Internet of Things

Hug the Rails

Trains, Team 11; Duck Developers
Team Members: Ayomide Omokanwaye, Ankit Patel, Faraz Pathan, Jan Gabriel Del Rosario

Section 1: Communication

1.1) Introduction:

Use principles of software engineering to successfully create a software application.

1.2) Problem Statement:

Hug the Rail has a problem: the software they use in their train systems is outdated and inefficient. Therefore, they have contracted Duck Developers to create a new and updated software system for them.

1.3) Purpose of the product (IoT Hugs The Rail):

Use IoT to make Hug The Rail safer, less costly, and more efficient. To automate the train decision without the use of the backoffices. To add more features and to increase efficiency and reliability of the train's software system.

1.4) We have talented team:

We have a talented and experienced team within the field that can supply the needs to implement and design this project. Everyone on the team has taken many courses in computer science, knows multiple programming languages, and has experience working on large-scale projects in the past. We are confident in our ability to deliver satisfying results to our client.

1.5) Evolving current operation:

Currently the train system has a need for backoffices to make decisions. With the IoT we can automate these decisions and by retrieving data about the train. This would make the whole process more efficient, less costly , and safer for the public.

1.6) Approach to the solution:

- Think critically
- Communicate (use zoom, groupme)
- Stay organized (google docs, GitHub/GitLab)
- Be ready to adapt

1.7) Reviews:

Once we get more details about the project requirements from the client, we will communicate with the software designers to get their feedback.

Section 2: Planning & Overview

Planning:

2.1) Roles and responsibilities:

- Ankit Patel: Team Lead for 2 weeks, designer, developer, tester, presenter
- Faraz Pathan: Team Lead for 2 weeks, designer, developer, tester, presenter
- Jan Gabriel: Team Lead for 2 weeks, designer, developer, tester, presenter
- Ayomide Omokanwaye: Team Lead for 2 weeks, designer, developer, tester, presenter

2.2) Process Model:

The process model we will be using is the unified model. The reason we chose this model is because the unified model is a very organized model that prioritizes documentation. And since we anticipate revisiting this project in the future, we think the unified model goes very well with this project.

2.3) Timeline:

- Feb 9 - March 1: (3 weeks) **Planning** out the project and constant **communication**
- March 2 - March 28: (4 weeks) Meet with coach/Reza and building **models** and make adjustments (UML, etc.)
- March 29 - April 11: (2 weeks) Software **Architecture** & **Design**
- April 12 - May 3: (3 weeks) **Construction** (2 weeks of coding & 1 week of testing) and **Sprint**
- May 4 - May 11: (1 week) **Deployment & Presentation**

Overview of IoT HTR:

2.4) Cellular Network:

Trains depend on WiFi/Cellular networks to receive live data about their environments and surrounding traffic. When a train loses its Wi-Fi/cellular network, it needs local information to operate safely.

The current train models are using Communications-based train control(CBTC) and they use telecommunications between trains and railways to accurately communicate with each other. Within the communication they can get the closest thing to the exact position as they can get. Trains systems usually have two different methods of moving with one another. The fixed block is a method when a train only moves ahead when the upcoming preset distant block is empty. The moving block has a more calculated position of the train and automatically calculates its brake curve and speed. While the train is moving it should always maintain a safe speed and braking curve. If communications are lost with either system the train can keep moving with a reduced speed until it is fixed, the train could stop until it is fixed, or have an entirely different transportation system take its place. With the train communication being out the other trains will still be able to run but at a lower speed if it is near the loss of communications of the train who lost communications.

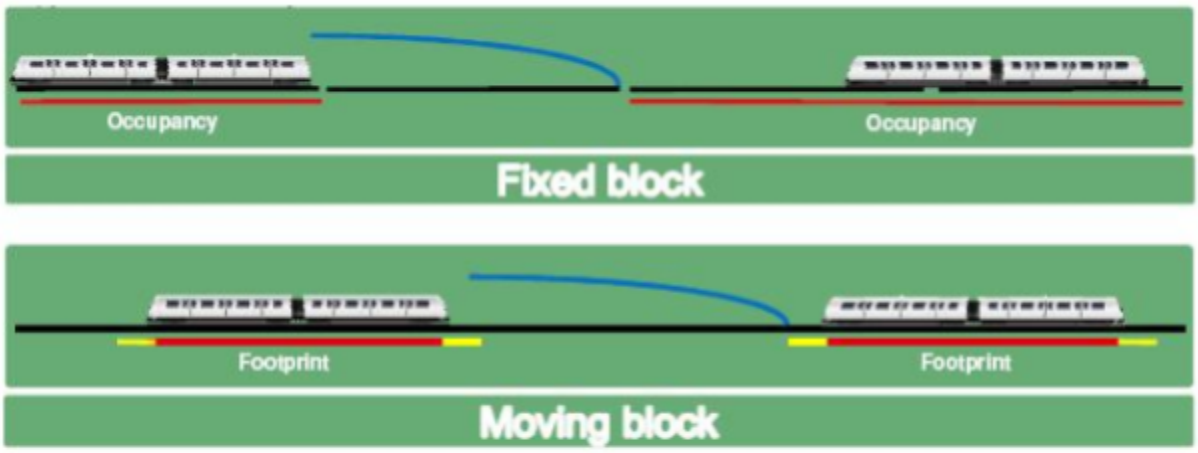


Diagram 1) Train movement comparison: “Fixed block” vs “Moving block”

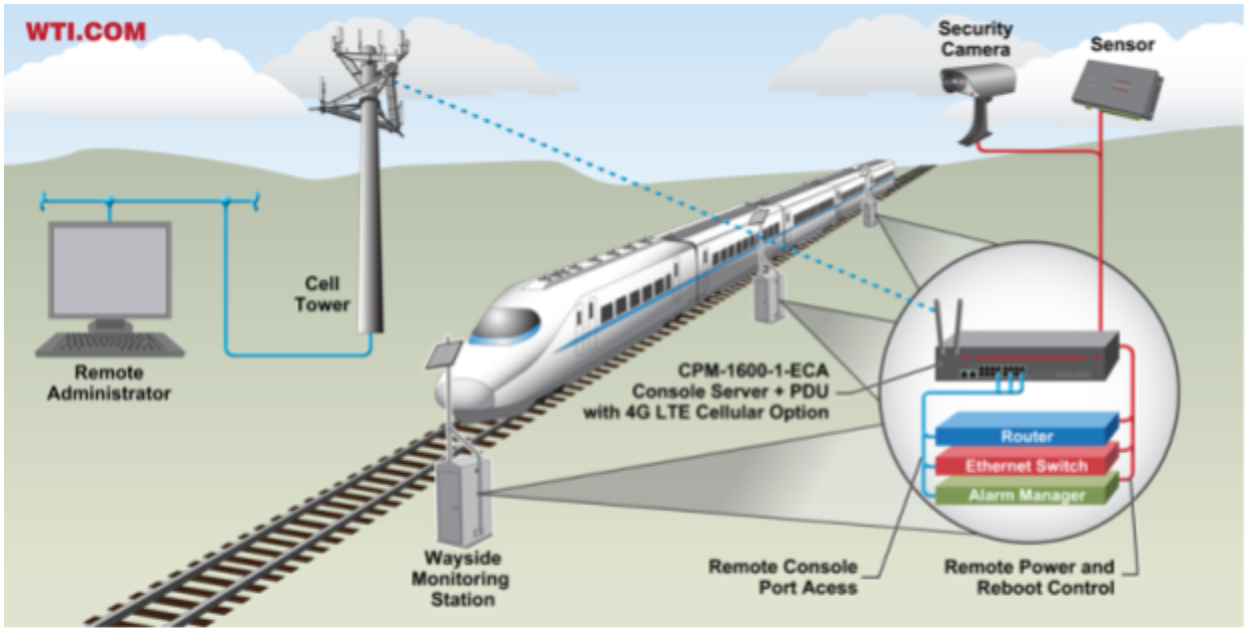


Diagram 2) High-level overview of HTR's IOT

2.5) Hazardous Conditions:

Due to the nature of how trains work, there are many hazardous conditions needed to be accounted for to ensure the safety of everyone who uses them. One type of hazard is the weather: like when it snows, rains, even dangerous wind speeds. Weather hazards can be dangerous due to impacting the tracks that the train is on, as well as affecting the conductor's ability to direct the train. Speaking of tracks, another type of hazard is structural. Old train tracks may be rusty and brittle, making them vulnerable to breaking during a train's route. This can lead to a catastrophic event such as derailment and will likely take some of the lives on board. Old train tracks may also be susceptible to operational failures such as failing to switch directions. Another example of a structural hazard is an accidental train decoupling; when part of a train detaches from the rest of the train. The last type of hazards we should also account for are human-made hazards/errors. These hazards are rather rare with some examples including head-on collisions (more of two trains bumping each other rather than crashing), obstruction of the track and a wrong signal sent by the track operator.

2.6) Use of Sensors can lead to improvement:

———Thankfully, many of the potential dangers explained in the previous paragraphs can be addressed through the use of sensors. The following paragraphs will explain the various different types of sensors and the ways they can be used to increase efficiency and the safety of passengers on the train.

The temperature sensors would be used to determine speed of the train. If the temperature is below freezing, the train would be programmed to travel at a slower speed due to the possibility of ice. The temperature sensors would also be used to determine the inside temperature. Depending on how cold or hot it is, it would use that data to calculate how much air conditioning or heating to use for the passengers. Pressure sensors would be used to determine the power for breaking and fuel engagement. Vibration sensors would be used to determine if the train is traveling at an alarming rate or if there are defects on the tracks. Position sensors can be used to determine the safest angle and placement of the train of the track, depending on the structure and path of the railing. There also would be proximity sensors to determine if something or someone is on the track to see if there is a need for braking. The operator would be recommended specific speeds, braking dynamics, routes, and levels depending on the circumstances. There also exists the chance of malfunctions.

Sensors can be employed to detect defects within the train system. These types of sensors are known as defect detectors, sensors which monitor vital train components and alert you if any of them have abnormal behavior. Furthermore, vehicle weight detectors, sensors which gauge the approximate weight of a cabin, are often used to detect if a train cabin has an appropriate cargo/passenger load. The use of these detectors will improve safety and ensure no weight regulations are violated. Additionally, many modern trains have rail break monitors. These are sensors whose entire purpose is to monitor the brakes and alert the train conductor if they are defective.

Lastly, there is one final line of defense in ensuring the safety of the train; the sensor monitor. The sensor monitor will closely observe all of the other sensors, monitors, and detectors on the train to ensure they are working properly. If a certain sensor isn’t working, then the sensor monitor will alert the train conductor.

In the event of an emergency, like a loss of connection, there would be sensors to detect such issues. This would require the operator to use emergency braking, backup power, and system reset.

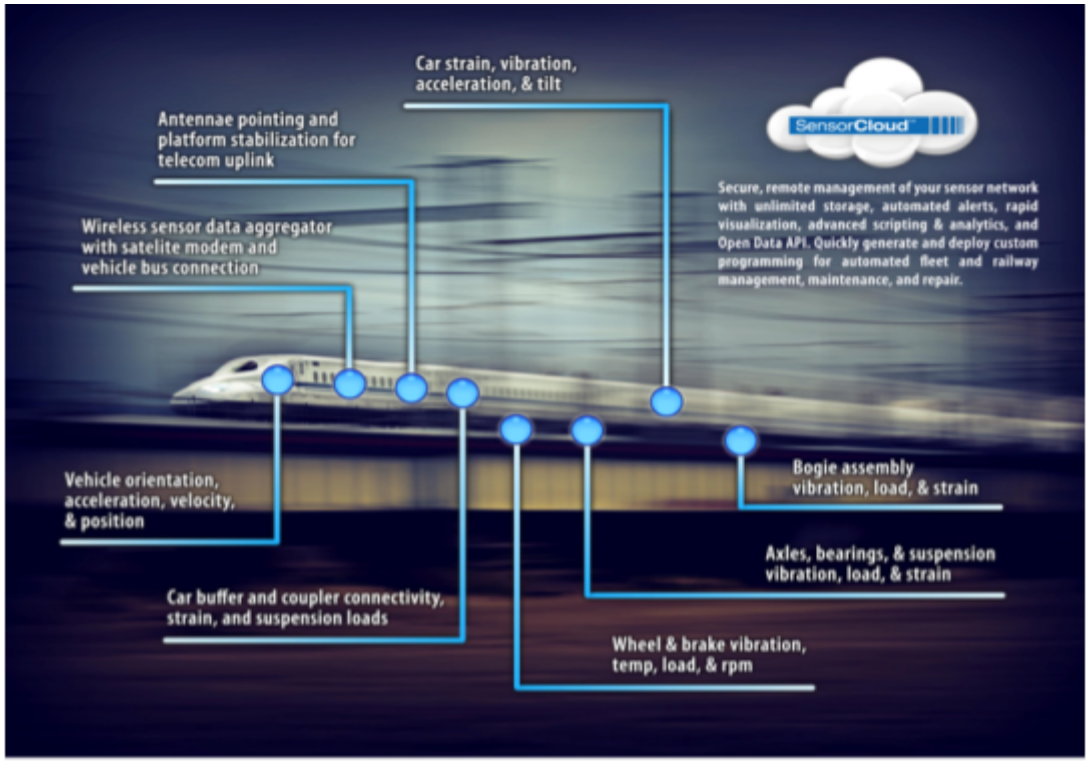


Diagram 3) Trains' complex engineering means our software should have a wide variety of uses.

Section 3: Requirements

3.1 Non-Functional

3.1.1 Hardware

R1) All sensors shall be water-proof.

R2) All sensors shall last for 1000 hours under water.

R3) All sensors shall operate 99.99% of the time.

R4) IoT HTR shall be supported by a hardwired LoRaWan network.

R5) IoT Hardware shall be able to support 1000 sensors.

R6) IoT shall support 5TB of data everyday.

R7) IoT shall summarize information about the status of the train (in real time) on a monitor inside the conductor's cabin.

3.1.2 Performance

R8) At most, IoT HTR shall process an event within 0.5 second of its occurrence.

R9) IoT HTR sensors shall detect events within 0.1 seconds.

3.1.3 Reliability

R10) IoT HTR reliability shall be no less than 99.9%.

3.1.4 Security

To account for potential sensor failure (reliability via redundancy),
R11) Each sensor shall have a backup sensor.

R12) The train conductor will require a User ID and password to access the train's software.

To avoid software updates on sensors mid-route and to avoid hacking:

R13) sensors will only connect to wifi when the train engine is off.

R14) sensors will only connect to wifi when the train is at a train station.

Furthermore,

R15) Once a train arrives at its destination, it will upload the data it collected from its journey to the HTR servers.

3.2 Functional Requirements

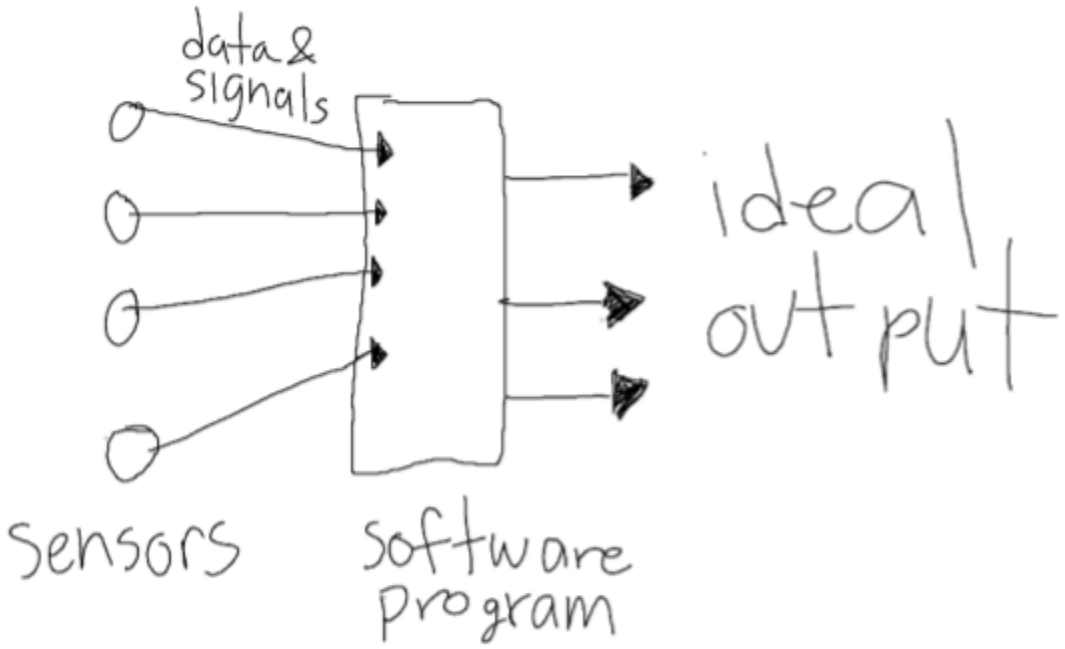


Diagram 4) Extremely simplified overview of most functional requirements

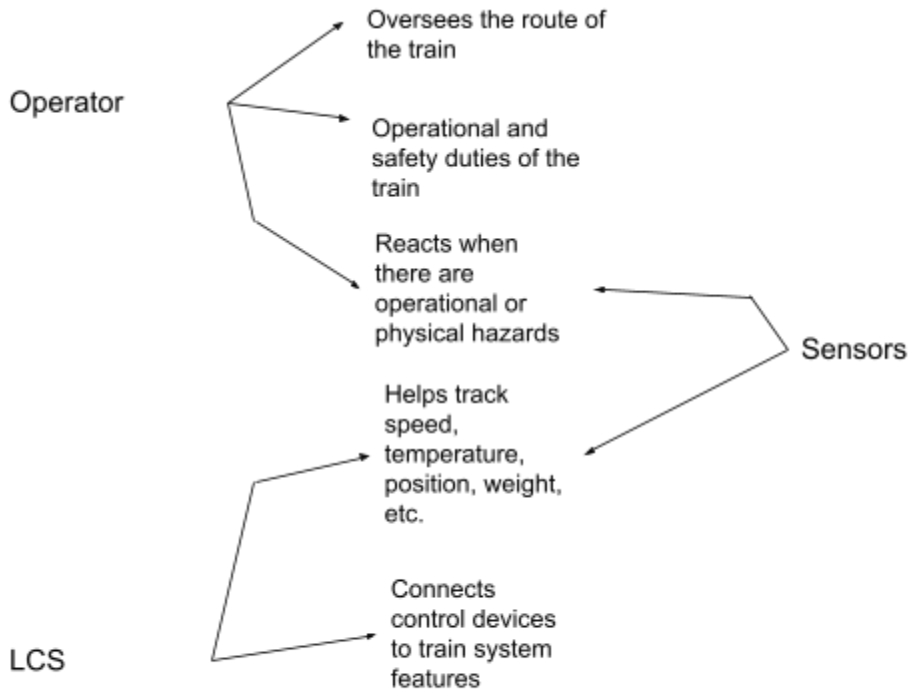


Diagram 5) A more detailed schematic of the relationship between sensors, operator and LCS.

3.2.1 Route Obstruction Prevention:

R16) Geographical Sensors (GPS) will also be able to track the position of the train within a margin of error no greater than 10 feet.

Upon meeting the CTO of HTR, it was concluded that at minimum the following four features are needed:

3.2.1.1) Detect standing objects on the path of the train with distance, time to collision, and recommend action to the operator for braking or increasing/decreasing the speed, or completely stopping.

R17) Detection of standing objects will be done using proximity sensors.

R18) Each proximity sensor will have a field of vision of 120 degrees angle.

R19) Each proximity sensor will be able to detect an object up to 2 miles away.

R20) Each proximity sensor will have the ability to detect objects with 98% accuracy.

R21) Apart from being placed in the front of the train, proximity sensors will be placed in other areas of the train to cover blind spots such as some parts of the side and the back.

R22) Immediately upon detection of an object, IOT will access data of the speed of the train and compare that with the distance of the object to the train, in order to give a precise plan of action to the operator.

R23) Once the IOT is done with the calculations, it shall signal the operator what the plan of action should be.

3.2.1.2) Detect moving objects, their distance (ahead and/or behind), their speed, and recommend to the operator braking or changing speed.

R24) Motion sensors on the front and back side of the train shall detect any moving objects that are within 2 miles of the train (both in front of and behind)

R25) The field of vision of these motion sensors shall be set to 120 degrees.

R26) The sensitivity of these motion sensors shall be set to detect objects that have a surface area greater than 1 square foot.

R27) Motion sensors shall be accurate at least 98% of the time.

R28) If a moving object is detected by a motion sensor, then the conductor shall be notified within 0.1 seconds.

And,

R29) The IOT shall give the conductor recommendations on whether to slow down, speed up, or completely stop the train.

3.2.1.3) Detect if the gate crossing is open/closed, its distance from the train, and recommend a speed change (increase, decrease, or stop) to the operator.

R30) GPS shall be used to determine the proximity of gate crossings with 99.9% accuracy.

R31) If a gate is open, the conductor shall be informed about it and recommended to proceed normally.

R32) Proximity sensors on the front side of the train shall detect if a gate crossing is closed.

R33) In the event a gate crossing is closed, the software shall recommend the conductor slow down the train to a complete stop.

R34) Proximity sensors will detect when the gate opens.

R35) If a gate opens, the conductor shall be recommended to start moving.

3.2.1.4) Detect wheel slippage using GPS speed data, compare it with the wheel RPM, and recommend the operator break or alter the speed.

R36) Tachometers, sensors which measure rotations per minute, shall be placed on the first four wheels at the front of the train.

R37) Tachometers shall measure the rpm of the wheels with 98% accuracy.

R38) An odometer in the front of the train shall measure the speed of the train to the nearest 0.1 mph.

R39) The speed calculated by the odometer and the rpm measured by the tachometer shall be compared. If the ratio of these two measurements recommended the train is slipping/skidding, the train conductor shall be notified and recommended to lower the train speed.

R40) Such comparisons shall be made every 30 seconds.

3.2.2 Weather

Along with the minimum 4 requirements detailed by the CTO of HTR, our group has also decided to implement three more features that concerns weather and its impact on train service.

3.2.2.1) Detect severe wind gusts and recommend the operator to alter speed.

R41) Anemometers will be able to measure wind speeds with 99.9% accuracy.

R42) They will be placed in certain parts of the train to ensure all around measurement of the winds (front, sides, rear).

R43) Anemometers will automatically take new measurements every 30 seconds.

R44) If an anemometer obtains a wind speed measurement of 50 mph or more, the IOT will recommend the operator to slow down the train.

3.2.2.2) Detect heavy precipitation (rain or snow), track how strong it is and recommend the operator to alter speed and turn on train lights.

R45) Rain sensors installed at the front of the train will detect rainfall with 99.9% accuracy.

R46) Snow gauges on the sides of the train will be able to measure snowfall with 99.9% accuracy.

R47) Rain sensors and snow gauges will continuously update their data (by taking new measurements) every 30 seconds.

R48) If two consecutive 0.3 inch/hour measurements have been measured (this rule is for both rain and snow), the sensor will signal the operator to slow down and to turn on the train' lights.

3.2.2.3) Detect low visibility and recommend the operator to alter speed and turn on train lights.

R49) Visibility sensors will be able to measure up to 2 miles in front of the train with 99.9% accuracy.

R50) Visibility sensors will be installed in the front of the train.

R51) If measured distance is below 2 miles, signal the operator to lower speed and turn on train lights.

3.2.3 Horn Features

Towards the end of the project, the CTO of HTR decided to add a requirement dealing with horn recommendations.

R52) The train's horn shall be heard from at least 1.5 miles away.

3.2.3.1) In addition to detecting the status of an upcoming gate, now, when there is an upcoming gate one mile away from the train, LCS shall recommend the operator to honk the horn for 15 seconds. And when the train is about to pass through the gate, LCS shall recommend the operator to honk the horn for 5 seconds.

R53) When the train is 1 mile away from an open gate, LCS shall recommend the operator to honk the horn for 15 seconds.

R54) As soon as the train is about to pass through an open gate, LCS shall recommend the operator to honk the horn for 5 seconds.

Section 4: Requirement Modeling

4.1 Use Cases

4.1.1: System Administrator wants to access LCS via IOT

Use Case: System Administrator wants to access the LCS via IOT.
Primary Actor: System Administrator
Goal in Context: To allow the System Administrator to access the LCS (LCS is the user interface of the software system).
Preconditions: The LCS is on, but no user is logged in.
Trigger: The System Administrator turns on the computer, and enters their user ID and password.

Scenario:

1. The System Administrator turns on the LCS.
2. The System Administrator enters their user ID and password.
3. If their user ID and password are correct, then allow the System Administrator access to the LCS. If their user ID and password are incorrect, prompt the System Administrator to try again. Successful and failed login attempts are recorded in the log.

4.1.2: Train Operator wants to access LCS via IOT

Use Case: Train Operator wants to access the LCS via IOT.
Primary Actor: Train Operator
Goal in Context: To allow the Train Operator to access the LCS.
Preconditions: The LCS is on, but no user is logged in.
Trigger: The Train Operator turns on the computer, and enters their user ID and password.
Scenario:

1. The Train Operator turns on the LCS.
2. The Train Operator enters their user ID and password.
3. If their user ID and password are correct, then allow the Train Operator access to the LCS. If their user ID and password are incorrect, prompt the Train Operator to try again. Successful and failed login attempts are recorded in the log.

4.1.3: Train Operator accesses information about gate crossing

Use Case: The Train Operator wants to access information about an upcoming gate crossing.
Primary Actor: Train Operator
Goal in Context: To give the Train Operator knowledge about whether its okay to proceed through a gate crossing or if the train has to wait.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; the train must be traveling at least at the minimum speed.
Trigger: The Train Operator inputs a command to the LCS asking for information (status, distance) about an upcoming gate crossing.
Scenario:

1. The Train Operator inputs a command into the LCS asking for the distance of the next gate crossing, which is calculated by the GPS.
2. Once the data is given, the Train Operator shall then make a decision whether or not to slow the train down (if gate crossing is more than 2 miles away, the Train Operator could opt to keep the train at the same or even faster speed.)
3. Once the train gets within 2 miles of the next gate, the proximity sensors shall send data to the Train Operator via the IOT and LCS about the gate's status (open or close).
4. If the gate is open, the Train Operator shall proceed as normal through the gate. If it is closed, the Train Operator will be prompted to halt the train and wait for the gate to open.
5. The time of input command, input command, and sensor readings are stored in the log.

4.1.4: Train Operator accesses information about path obstruction

Use Case: Train Operator Spots an object obstructing the path.
Primary Actor: Train Operator
Goal in Context: Provide information to the Train Operator about the obstruction (stationary or moving) and recommend to them an action to take.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed; Train Operator spots the object from afar.
Trigger: The Train Operator inputs a command into the LCS to enable the proximity sensors to send information about the obstruction to them.
Scenario:

1. Once the Train Operator sees an obstruction in the distance, they input a command into the LCS to allow the motion and proximity sensors to gain information about it.
2. The motion and proximity sensors shall send the data to the Train Operator (stationary or moving object and distance of it to the train). Both the proximity sensor and motion sensor have a range of 2 miles.
3. The LCS will also display recommendations to the operator about which actions to take (slow down the train if applicable).
4. The time of input command, input command, and sensor readings are stored in the log.

4.1.5: Train Operator accesses information about wheel slippage

Use Case: Train Operator wants to get information about the train's wheels.
Primary Actor: Train Operator
Goal in Context: Provide information about the train's wheels so the Train Operator knows if they have to take any action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: The Train Operator inputs a command into the LCS to enable the tachometers and odometers to take data and display it on the LCS.
Scenario:

1. The Train Operator inputs a command into the LCS to get information about the status of the train's wheels (RPM). At the same time, the odometers shall report the train's current speed.
2. The IOT will then compare the data between each other.
3. If the IOT detects wheel slippage to be possible, it will prompt the Train Operator to either slow down or halt the train (depending on severity).
4. The time of input command, input command, and sensor readings are stored in the log.

4.1.6: Train Operator accesses wind speed information

Use Case: The Train Operator observes that the weather has become windy.
Primary Actor: Train Operator
Goal in Context: Obtain wind speed information, and if applicable, recommend the Train Operator take appropriate action.

Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: The Train Operator inputs a command into the LCS to enable the anemometers to send information about the wind speed to them by displaying it on the LCS.
Scenario:

1. The Train Operator inputs a command into the LCS to get information about the wind speed outside the train.
2. Almost instantaneously, the anemometers will return its most recent measurement of the wind speed.
3. The LCS will display the wind speed. If the wind speed is above 50 mph, then the LCS will display a recommendation to the Train Operator to reduce the speed of the train.
4. The time of input command, input command, and sensor readings are stored in the log.

4.1.7: Train Operator is informed of high wind speeds

Use Case: The LCS shall inform the Train Operator of high wind speeds (only if the operator hasn't already noticed) and recommends a course of action.
Primary Actor: Train Operator
Goal in Context: Detect high wind speeds, obtain wind speed information, and recommend the Train Operator take appropriate action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: Multiple anemometers obtain two consecutive 50 mph wind speed measurements.
Scenario:

1. An anemometer obtains a wind speed measurement of 50 mph or more.
2. The IOT will display the wind speed information on the LCS.
3. The IOT will recommend to the Train Operator that they reduce the speed of the train. This recommendation will be displayed on the LCS.
4. The sensor readings and time at which the readings were taken are stored in the log.

4.1.8: Train Operator accesses information about precipitation

Use Case: The Train Operator observes that there is precipitation outside the train.
Primary Actor: Train Operator
Goal in Context: Obtain precipitation information, and if applicable, recommend the Train Operator take appropriate action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: The Train Operator inputs a command into the LCS which tells the IOT to obtain the rate of rainfall and snowfall measured by the rain sensors & snow gauges and to display that information to the LCS.
Scenario:

1. The Train Operator inputs a command into the LCS to get information about the precipitation outside the train.
2. Almost instantaneously, the rain sensor will return the rainfall rate and the snow gauge will return the snowfall rate.
3. The rate of precipitation will be displayed on the LCS.
4. If either the rainfall rate or snowfall rate is greater than 0.3 inches per hour, then the LCS will recommend the Train Operator to reduce the speed of the train and turn on the train's headlights.
5. The time of input command, input command, and sensor readings are stored in the log.

4.1.9: Train Operator is informed of high rate of precipitation

Use Case: The LCS shall inform the Train Operator of high rates of precipitation (only if the operator hasn't already noticed) and recommends the appropriate actions.
Primary Actor: Train Operator
Goal in Context: Detect high precipitation rate, obtain precipitation rate information, and recommend the Train Operator take appropriate action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: Either multiple rain sensors or multiple snow gauges obtain two consecutive measurements greater than 0.3 inches/hour.
Scenario:

1. Either multiple rain sensors or multiple snow gauges obtain two consecutive measurements greater than 0.3 inches/hour.
2. The LCS will display the rate of precipitation on the screen which the Train Operator is using.

3. The LCS will recommend to the Train Operator that they reduce the speed of the train and turn on the train's headlights.
4. The sensor reading and time at which the readings were taken are stored in the log.

4.1.10: Train Operator accesses information about visibility

Use Case: The Train Operator notices that the visibility is low.
Primary Actor: Train Operator
Goal in Context: Obtain visibility information (such as fogginess), send it to the Train Operator, and have the LCS display recommendations that the Train Operator take appropriate action (only if the visibility is low).
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: The Train Operator inputs a command into the LCS which tells the IOT to obtain the measurements of the visibility sensors to display that information to the LCS.
Scenario:

1. The operator inputs a command into the LCS to get information about the visibility in front of the train.
2. The LCS displays the information about the visibility to the Train Operator.
3. If the visibility is less than 2 miles, the LCS will display a recommendation to the Train Operator that they turn on the train lights and that they reduce the speed of the train. If the visibility is greater than or equal to 2 miles, then the LCS will not make any recommendations.
4. The time of input command, input command, and sensor readings are stored in the log.

4.1.11: Train Operator is informed of low visibility

Use Case: The LCS shall inform the Train Operator of low visibility (only if the operator hasn't already noticed) and recommends the appropriate actions.
Primary Actor: Train Operator
Goal in Context: Detect fog, obtain visibility information, and recommend the Train Operator take appropriate action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.

Trigger: Visibility sensor detects that the visibility in front of the train is less than 2 miles.

Scenario:

1. Visibility sensor detects that the visibility in front of the train is less than 2 miles.
2. The LCS displays the information about the visibility to the Train Operator.
3. The LCS displays a recommendation to the Train Operator that they turn on the train lights and that they reduce the speed of the train.
4. The sensor reading and time at which the readings were taken are stored in the log.

4.1.12: Train Operator is informed of upcoming gate crossing

Use Case: The LCS shall inform the Train Operator of an upcoming gate crossing and recommend them to honk the train horn.

Primary Actor: Train Operator

Goal in Context: Detect upcoming gate crossing, and recommend the Train Operator take appropriate action.

Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.

Trigger: The proximity sensor detects there is an upcoming gate crossing exactly 1 mile away.

Scenario:

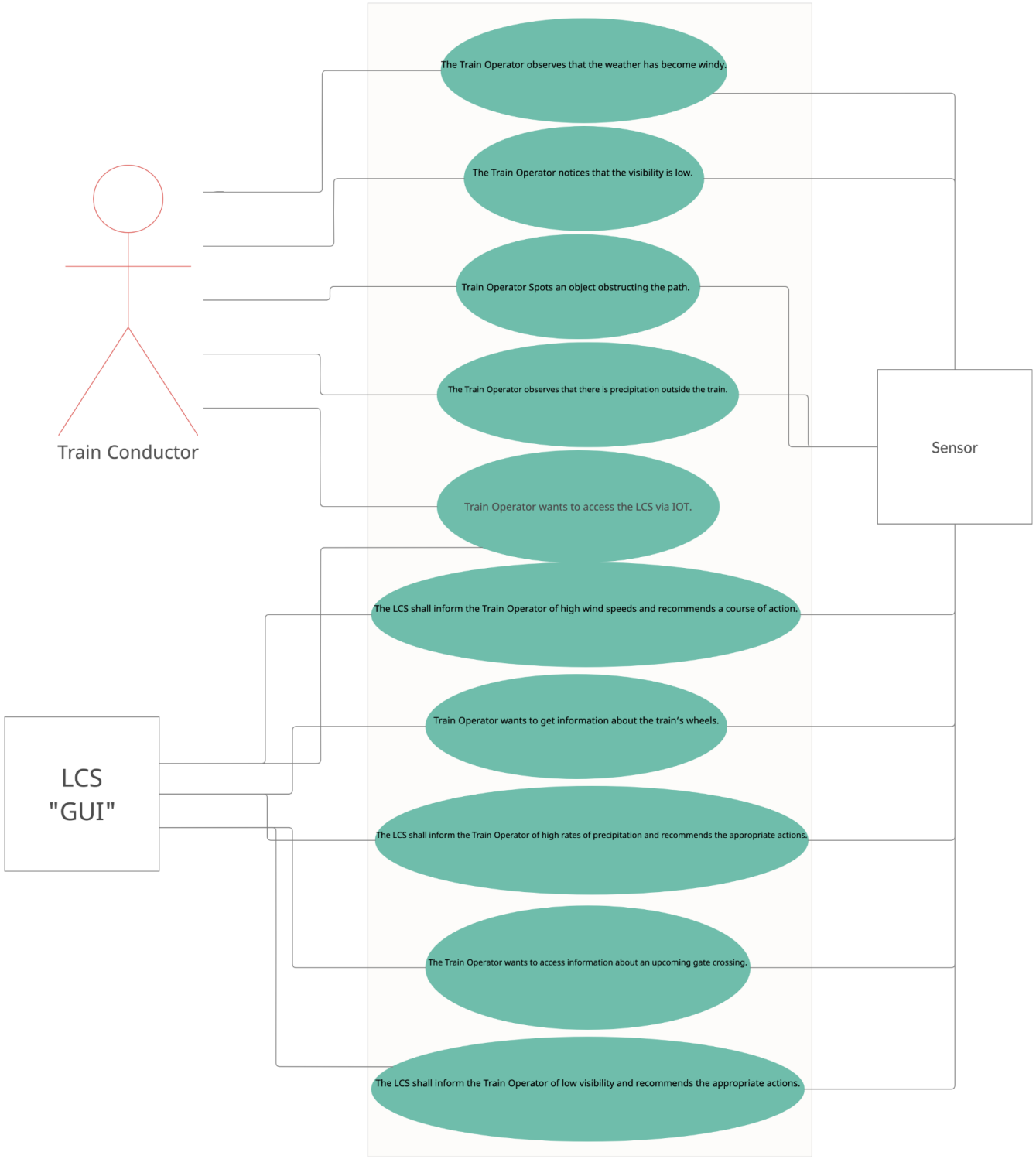
1. The proximity sensor detects there is an upcoming gate crossing exactly 1 mile away.
2. The LCS displays a recommendation to the Train Operator that they honk the train horn for 15 seconds.
3. The sensor readings and time at which the readings were taken are stored in the log.

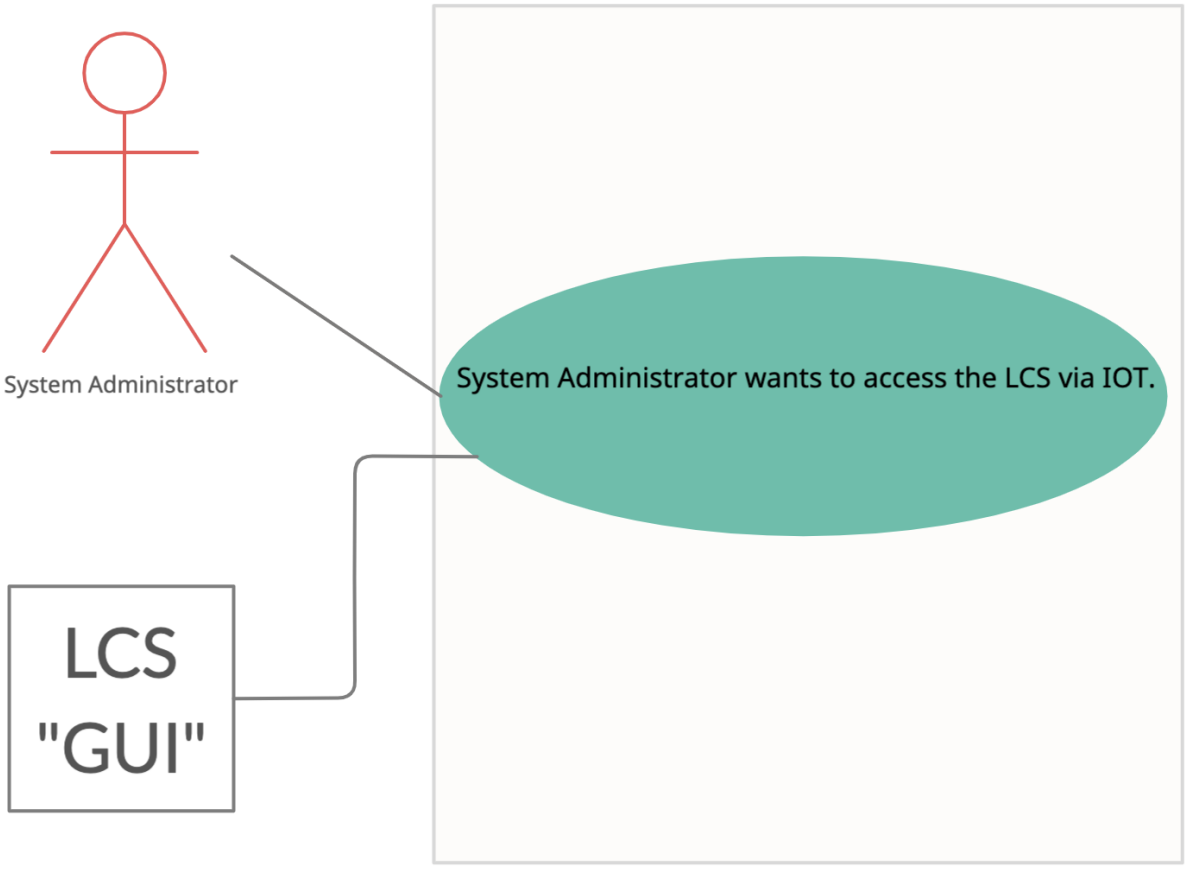
4.1.13: Train Operator is informed of immediate gate crossing

Use Case: If the train is immediately about to cross through a gate, the LCS shall inform the Train Operator and recommend them to honk the train horn.
Primary Actor: Train Operator
Goal in Context: Detect immediate gate crossing, and recommend the Train Operator take appropriate action.
Preconditions: Train Operator is logged into the LCS, sensors are already picking up data; train is running at least at the minimum speed.
Trigger: The proximity sensor detects there is an upcoming gate crossing between (0.0,1.0) miles away.
Scenario:

1. The proximity sensor detects that the train is about to cross a gate crossing, where the gate crossing is less than 1 mile in front of the train.
2. The LCS displays a recommendation to the Train Operator that they honk the train horn for 5 seconds.
3. The sensor readings and time at which the readings were taken are stored in the log.

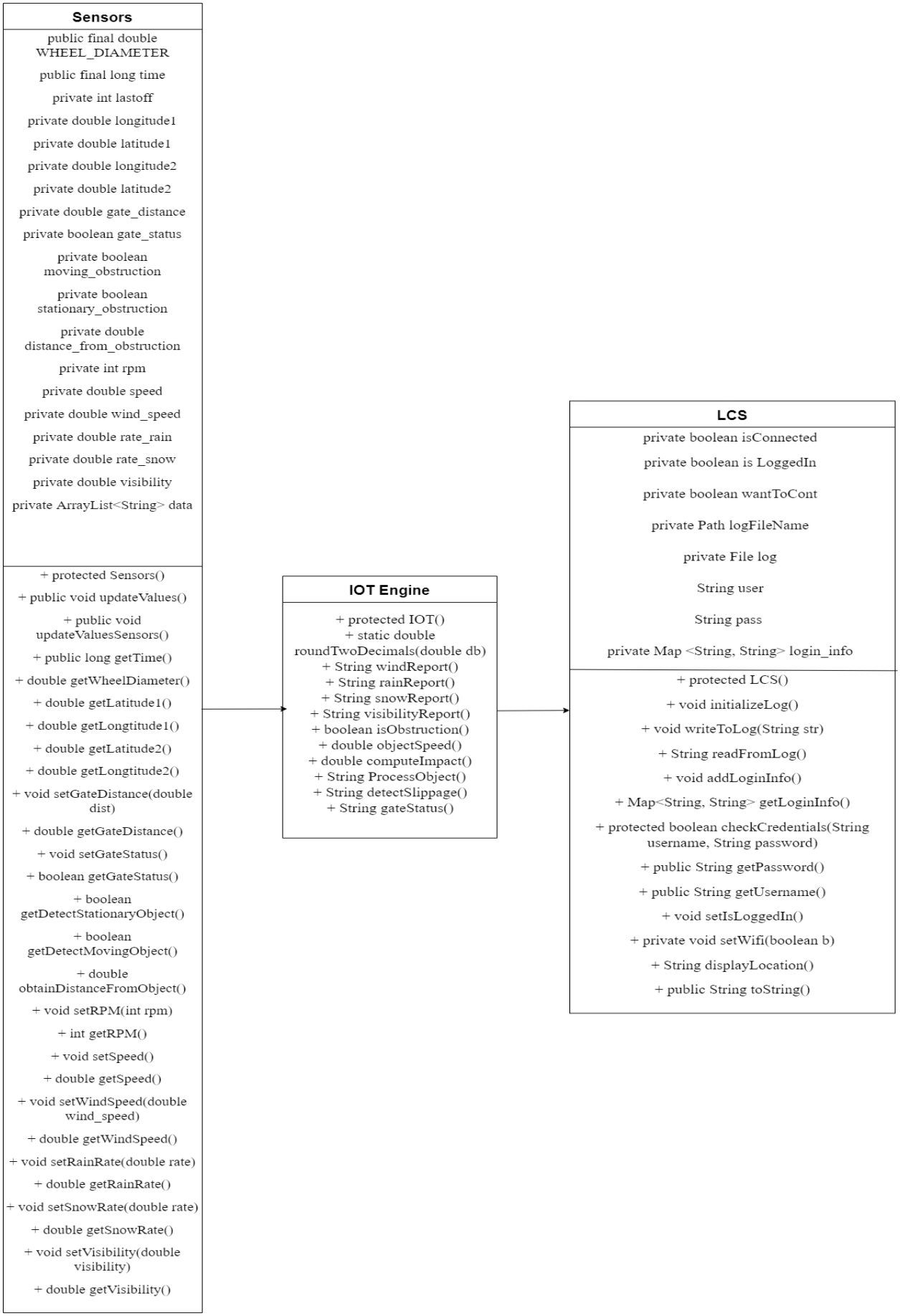
4.2 Use Case Diagrams

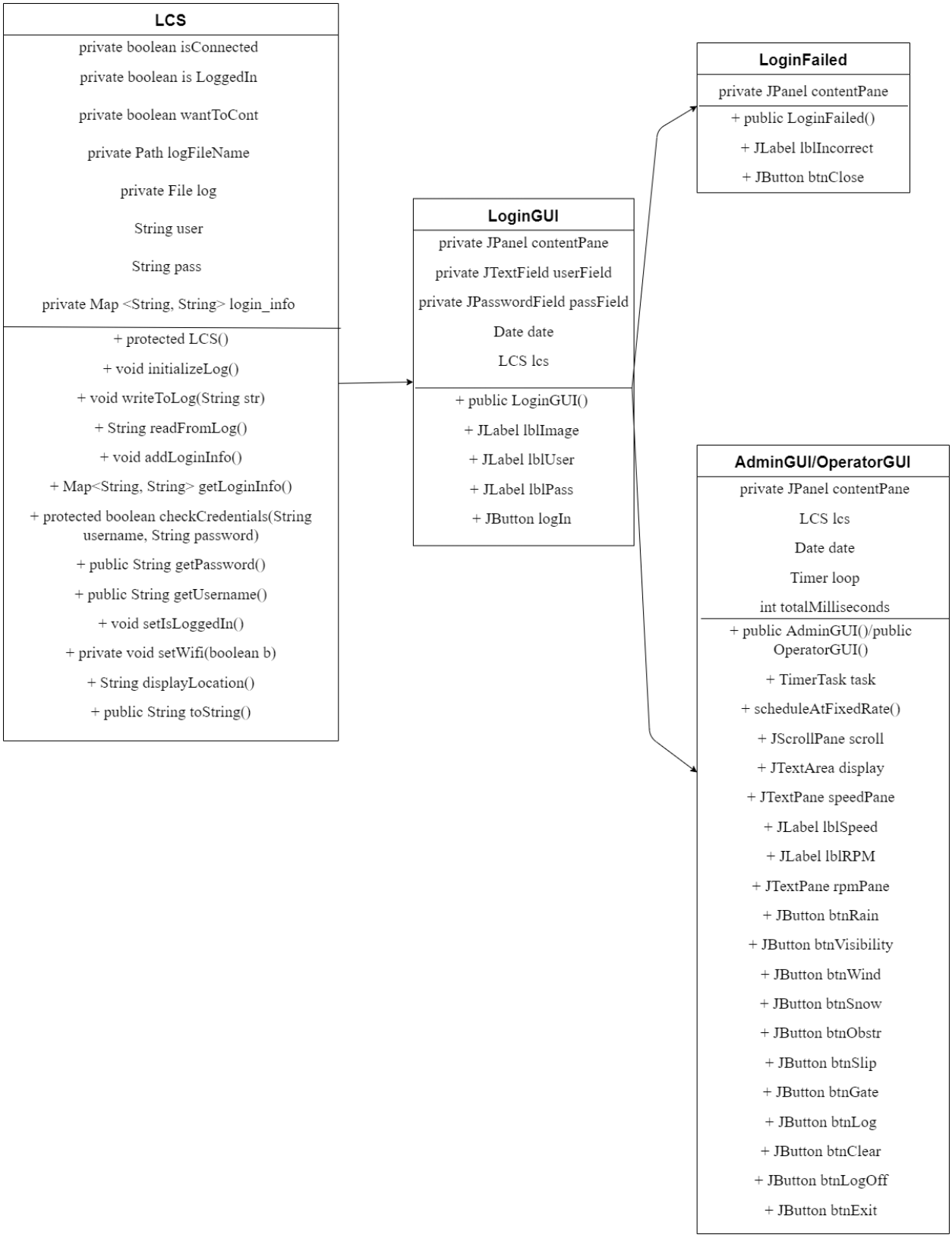



System Administrator
System Administrator wants to access the LCS via IOT.
LCS
"GUI"

4.3 Class-Based Diagram

The class diagrams below represent how our implementation of the software is connected. Sensors act as the parent, gathering the data to feed into the IOT engine, which is in charge of all the calculations with the data (ie. is it raining too hard, are the wheels slipping, etc.). The IOT engine then sends the result to the LCS, which essentially acts as a helper class for our GUIs (the LCS is outputted via the GUIs). The GUIs are split up into four classes: Login, LoginFailed, Admin and Operator. The Login GUI is where the user will login into the LCS with their credentials. If the credentials are not recognized by the system, it calls for the LoginFailed GUI to pop up. If the credentials match, it will open up to either Admin and Operator GUI (depending on the credentials fed into the Login). The Admin and Operator GUI has been merged in the class diagram for they are essentially the same, the only difference being the what commands the user has access to and what gets outputted by the LCS.


Sensors
public final double WHEEL_DIAMETER
public final long time
private int lastoff
private double longitude1
private double latitude1
private double longitude2
private double latitude2
private double gate_distance
private boolean gate_status
private boolean moving_obstruction
private boolean stationary_obstruction
private double distance_from_obstruction
private int rpm
private double speed
private double wind_speed
private double rate_rain
private double rate_snow
private double visibility
private ArrayList<String> data
+ protected Sensors()
+ public void updateValues()
+ public void updateValuesSensors()
+ public long getTime()
+ double getWheelDiameter()
+ double getLatitude1()
+ double getLongtitude1()
+ double getLatitude2()
+ double getLongtitude2()
+ void setGateDistance(double dist)
+ double getGateDistance()
+ void setGateStatus()
+ boolean getGateStatus()
+ boolean getDetectStationaryObject()
+ boolean getDetectMovingObject()
+ double obtainDistanceFromObject()
+ void setRPM(int rpm)
+ int getRPM()
+ void setSpeed()
+ double getSpeed()
+ void setWindSpeed(double wind_speed)
+ double getWindSpeed()
+ void setRainRate(double rate)
+ double getRainRate()
+ void setSnowRate(double rate)
+ double getSnowRate()
+ void setVisibility(double visibility)
+ double getVisibility()
IOT Engine
+ protected IOT()
+ static double roundTwoDecimals(double db)
+ String windReport()
+ String rainReport()
+ String snowReport()
+ String visibilityReport()
+ boolean isObstruction()
+ double objectSpeed()
+ double computeImpact()
+ String ProcessObject()
+ String detectSlippage()
+ String gateStatus()
LCS
private boolean isConnected
private boolean is LoggedIn
private boolean wantToCont
private Path logFileName
private File log
String user
String pass
private Map <String, String> login_info
+ protected LCS()
+ void initializeLog()
+ void writeToLog(String str)
+ String readFromLog()
+ void addLoginInfo()
+ Map<String, String> getLoginInfo()
+ protected boolean checkCredentials(String username, String password)
+ public String getPassword()
+ public String getUsername()
+ void setIsLoggedIn()
+ private void setWifi(boolean b)
+ String displayLocation()
+ public String toString()


LCS
private boolean isConnected
private boolean is LoggedIn
private boolean wantToCont
private Path logFileName
private File log
String user
String pass
private Map <String, String> login_info
+ protected LCS()
+ void initializeLog()
+ void writeToLog(String str)
+ String readFromLog()
+ void addLoginInfo()
+ Map<String, String> getLoginInfo()
+ protected boolean checkCredentials(String username, String password)
+ public String getPassword()
+ public String getUsername()
+ void setIsLoggedIn()
+ private void setWifi(boolean b)
+ String displayLocation()
+ public String toString()
LoginGUI
private JPanel contentPane
private JTextField userField
private JPasswordField passField
Date date
LCS lcs
+ public LoginGUI()
+ JLabel lblImage
+ JLabel lblUser
+ JLabel lblPass
+ JButton logIn
LoginFailed
private JPanel contentPane
+ public LoginFailed()
+ JLabel lblIncorrect
+ JButton btnClose
AdminGUI/OperatorGUI
private JPanel contentPane
LCS lcs
Date date
Timer loop
int totalMilliseconds
+ public AdminGUI()/public OperatorGUI()
+ TimerTask task
+ scheduleAtFixedRate()
+ JScrollPane scroll
+ JTextArea display
+ JTextPane speedPane
+ JLabel lblSpeed
+ JLabel lblRPM
+ JTextPane rpmPane
+ JButton btnRain
+ JButton btnVisibility
+ JButton btnWind
+ JButton btnSnow
+ JButton btnObstr
+ JButton btnSlip
+ JButton btnGate
+ JButton btnLog
+ JButton btnClear
+ JButton btnLogOff
+ JButton btnExit

4.4 CRC Model Index Cards

4.4.1 Sensors CRC Model

Sensors	
Description: Reads in data of it's surroundings and will supply it to the conductor (via the LCS) and to the IOT Engine when prompted.	
Responsibility:	**Collaborator:**
protected Sensors(), updateValues(), updateValuesSensors(), getTime() The constructor and methods that are in charge of receiving and tracking the data that the sensors take in.	**Sensors** (ALL Sensors)
getLatitude1(), getLongitude1(), getLatitude2(), getLongitude2(), setGateDistance(double dist), getGateDistance(), setGateStatus(), getGateStatus(), getDetectStationaryObject(), getDetectMovingObject(), obtainDistanceFromObject() Methods that will help gather and supply data regarding the train's current position, any form of obstruction and gate status.	**Sensors** (GPS, Proximity, Motion)
setRPM(int rpm), getRPM(), setSpeed(), getSpeed(), get Wheel Diameter Methods that will help gather and supply data regarding the train wheels' RPM as well as the train's current speed for calculating wheel slippage.	**Sensors** (Tachometer, Odometer)
setWindSpeed(double wind_speed), getWindSpeed(), setRainRate(double rate), getRainRate(), setSnowRate(double rate), getSnowRate(), setVisibility(double visibility), getVisibility() Methods that will help gather and supply data regarding current environmental factors.	**Sensors** (Anemometer, Rain sensor, Snow gauge, Visibility sensors)

4.4.2 IOT Engine CRC Model

IOT Engine	
Description: Takes in data and does calculations with them to output any necessary course of actions to the conductor when prompted.	
Responsibility:	**Collaborator:**
protected IOT(), roundTwoDecimals(double db) The constructor for the class that has access to the data read by the sensors. roundTwoDecimals is to ensure numbers are formatted properly and to avoid large floating numbers.	**Sensors**
windReport(), rainReport(), snowReport(), visibilityReport() Methods that gets the environmental data of the train, and uses them to determine if the train needs to slow down or brake.	**Sensors**
isObstruction(), objectSpeed(), computeImpact(), ProcessObject() Methods that takes in data about any upcoming obstruction (moving or stationary) and uses them determine whether or not the train should slow down or brake.	**Sensors**
detectSlippage(), gateStatus() Methods that take in data regarding RPM and gate status to determine slippage and the status of an upcoming gate crossing and whether or not the operator should honk the horn.	**Sensors**

4.4.3 LCS CRC Model

LCS	
Description: Takes in data from the IOT and helps with formatting to make it readable to the conductor. Acts as a helper to the visual interfaces, and helps with tracking data via the log.	
Responsibility:	**Collaborator:**
protected LCS(), initializeLog(), writeToLog(String str), readFromLog() Methods that initializes the LCS and the log where data will be written into.	**LCS**
addLoginInfo(), Map<String, String> getLoginInfo(), checkCredentials(String username, String password), getPassword(), getUsername(), setIsLoggedIn(), setWifi(boolean b) Methods in charge of the log in system of the software as well as tracking the status of WiFi.	**LCS**
displayLocation(), toString() Method that formats the data into a readable format for the operator and admin to read from the log.	**Sensors, IOT Engine, LCS**

4.4.4 LoginGUI CRC Model

LoginGUI	
Description: The visual interface of the login screen where the conductor will enter their credentials to log into the system.	
Responsibility:	**Collaborator:**
public LoginGUI() Constructor for the visual interface of the login screen.	**LoginGUI**
JLabel lblImage, JLabel lblUser, JLabel lblPass Components used to make the visual interface readable and appealing to the user.	**LoginGUI**
JButton logIn Button that enters the data inputted into the user and pass fields.	**LCS, LoginGUI**

4.4.5 LoginFailed CRC Model

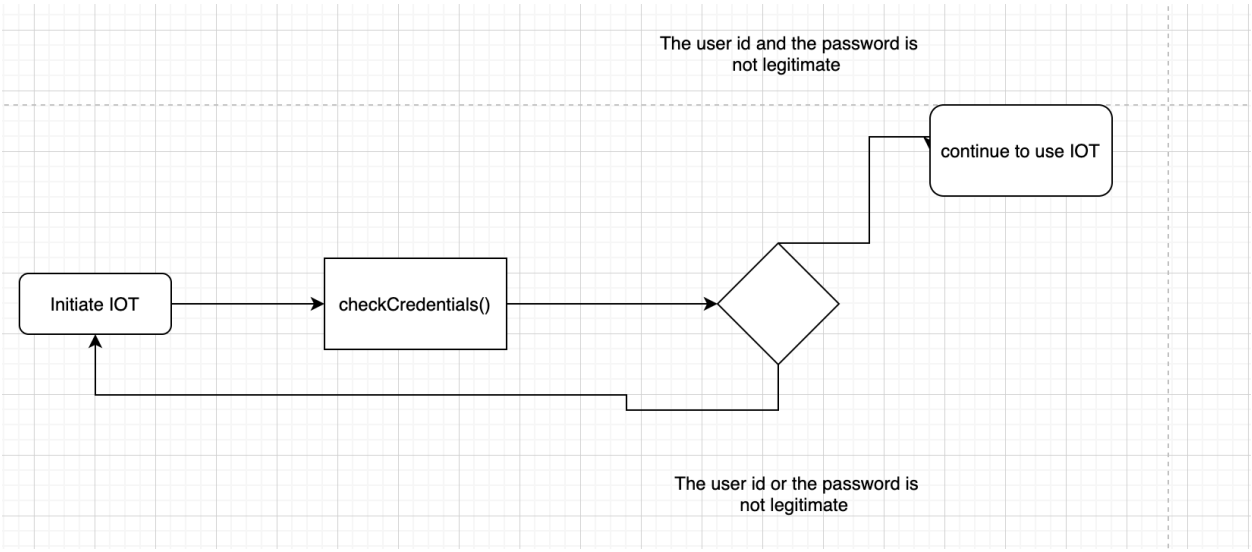
LoginFailed	
Description: The visual interface of the error message, which is prompted by incorrectly submitting wrong credentials from the LoginGUI.	
Responsibility:	**Collaborator:**
public LoginFailed() Constructor for the visual interface of the error message for incorrect credentials.	**LoginGUI, LoginFailed**
JLabel lblIncorrect, JButton btnClose Components used to show the error message as well as exit from the window.	**LoginGUI, LoginFailed**

4.4.6 AdminGUI/OperatorGUI CRC Model

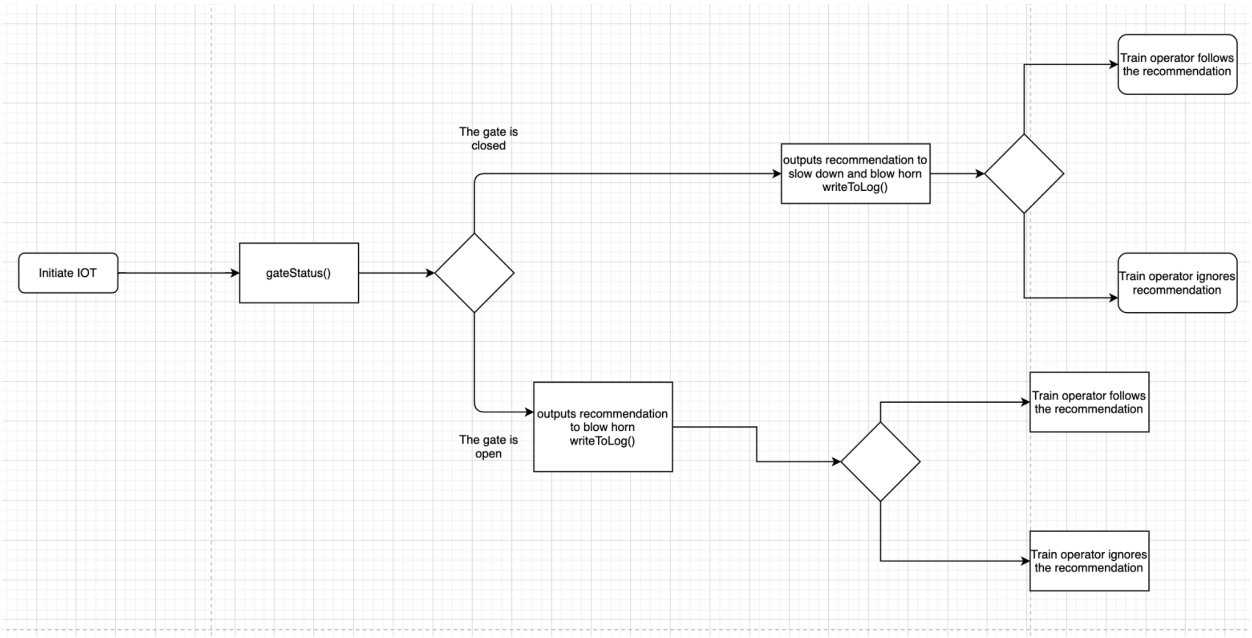
AdminGUI/OperatorGUI	
Description: The main GUIs where the conductor will be able to see the data, as well as have manual access to which data they want to see.	
Responsibility:	**Collaborator:**
public AdminGUI/OperatorGUI() The main visual interface for the LCS, where data and recommendations will be displayed.	**LCS, LoginGUI, AdminGUI/OperatorGUI**
TimerTask task, scheduleAtFixedRate() Object and method that aids in automating the visual data updates.	**AdminGUI/OperatorGUI**
JScrollPane scroll, JTextArea display, JTextPane speedPane, JTextPane rpmPane, JLabel lblSpeed, JLabel lblRPM Components that enable the data and recommendations to be outputted. Helps make the interface readable for the conductor.	**Sensors, IOT Engine, AdminGUI/OperatorGUI**
btnRain, btnVisibility, btnWind, btnSnow, btnObstr, btnSlip, btnGate, btnLog, btnClear, btnLogOff, btnExit Buttons that enable the conductor to manually call what data they want to look at. It also implements more trivial functionality like clearing the display screen to prevent data from clumping on it.	**Sensors, IOT Engine, AdminGUI/OperatorGUI**

4.5 Activity Diagrams

4.5.1: Use case 1 and 2

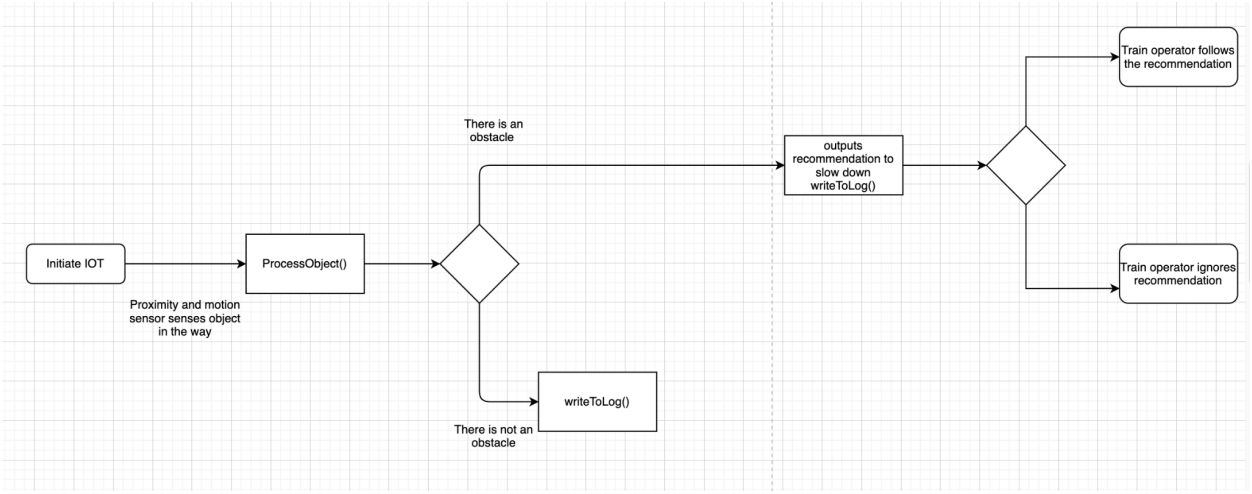


4.5.2: Use case 3



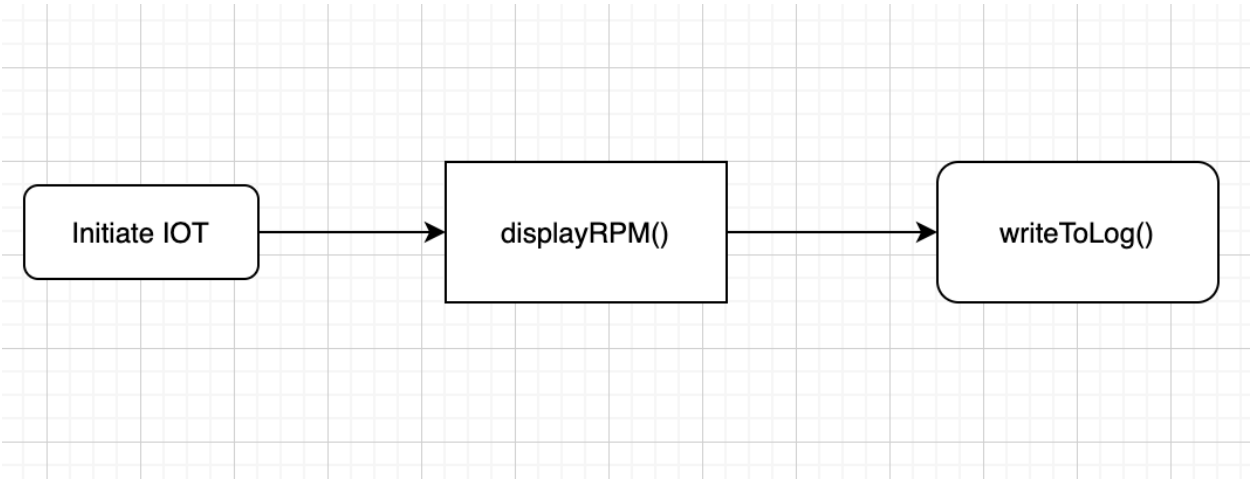
*Note: Implied that the user logged in.

4.5.3: Use case 4



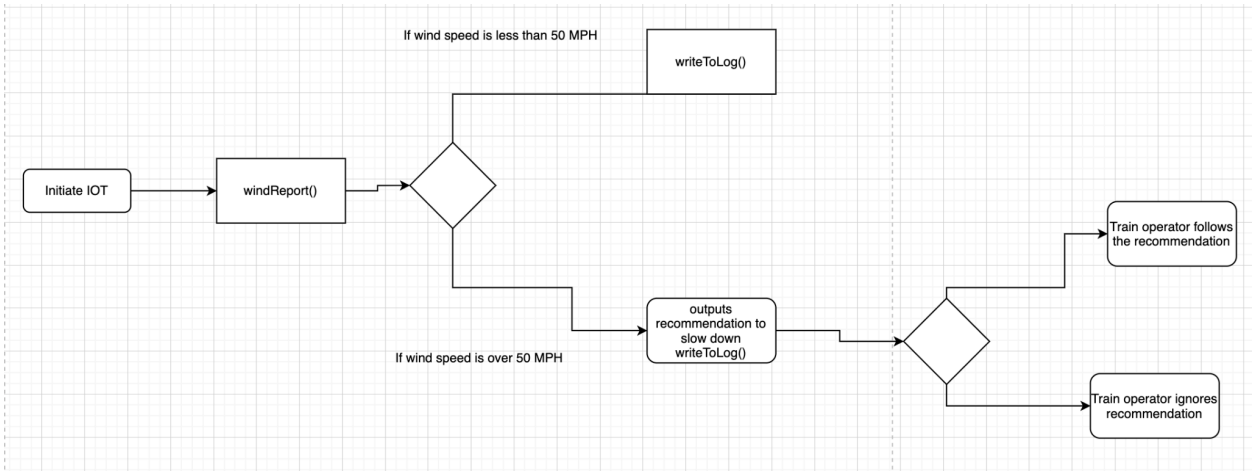
*Note: Implied that the user logged in.

4.5.3: Use case 5



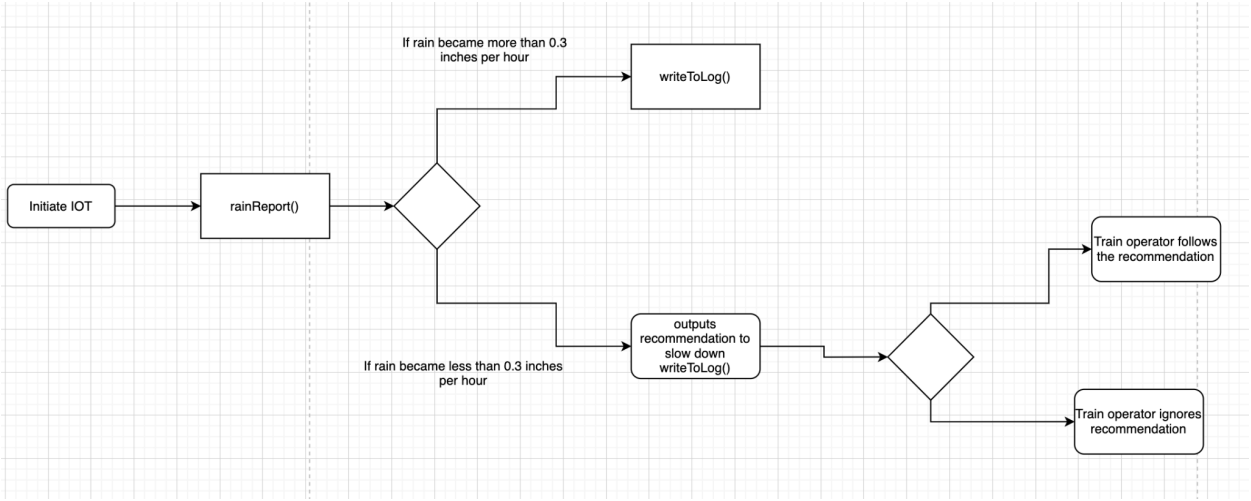
*Note: Implied that the user logged in.

4.5.4: Use cases 6-7



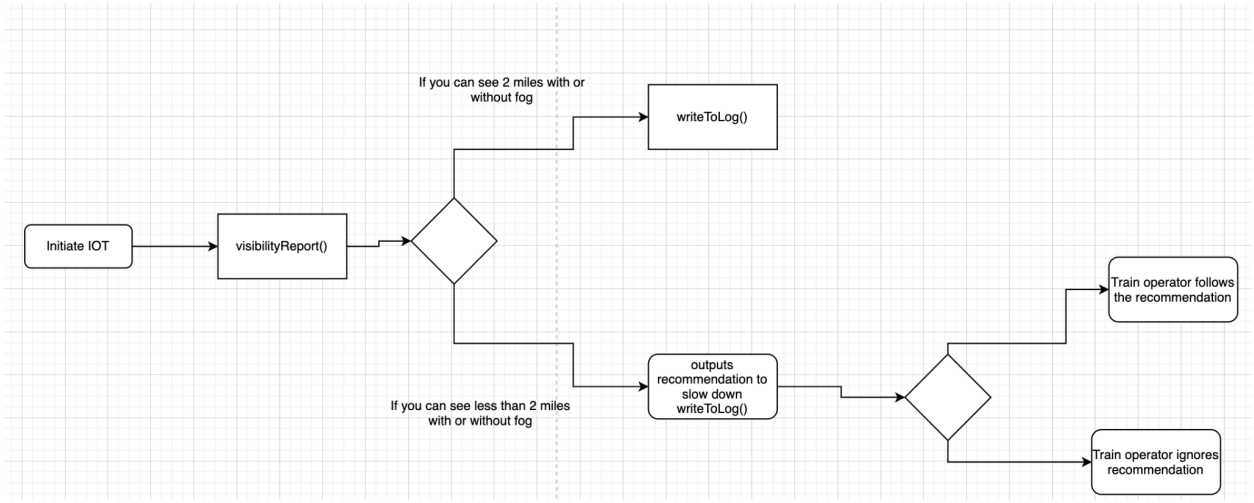
*Note: Implied that the user logged in.

4.5.5: Use cases 8-9



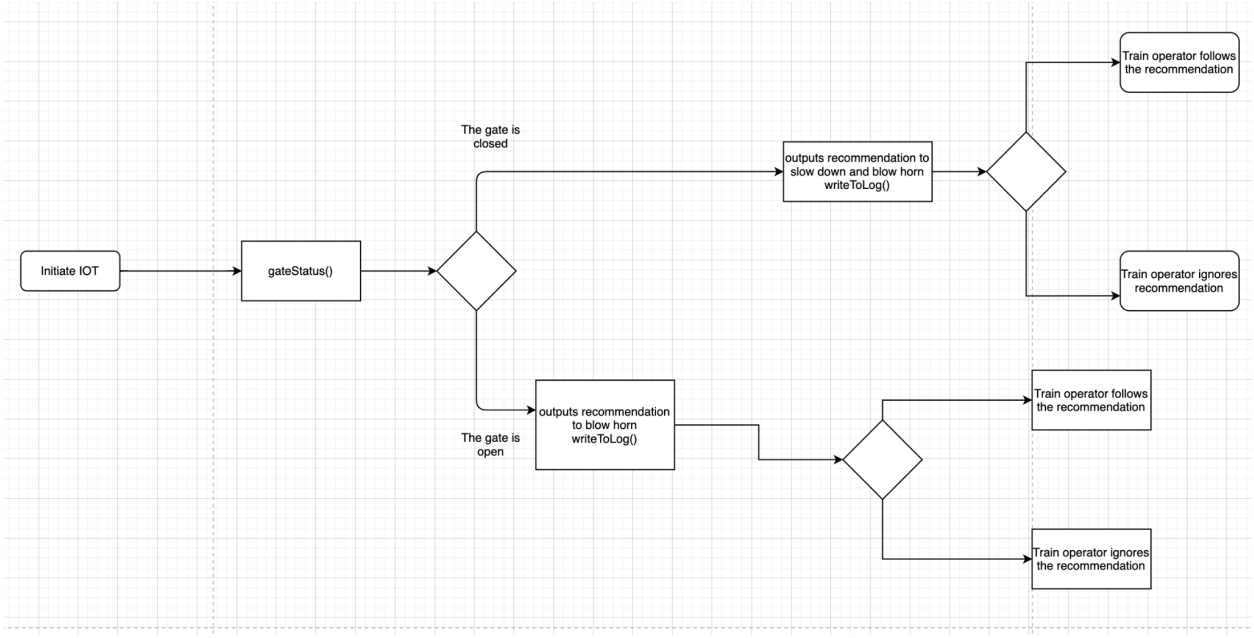
*Note: Implied that the user logged in.

4.5.6: Use cases 10-11



*Note: Implied that the user logged in.

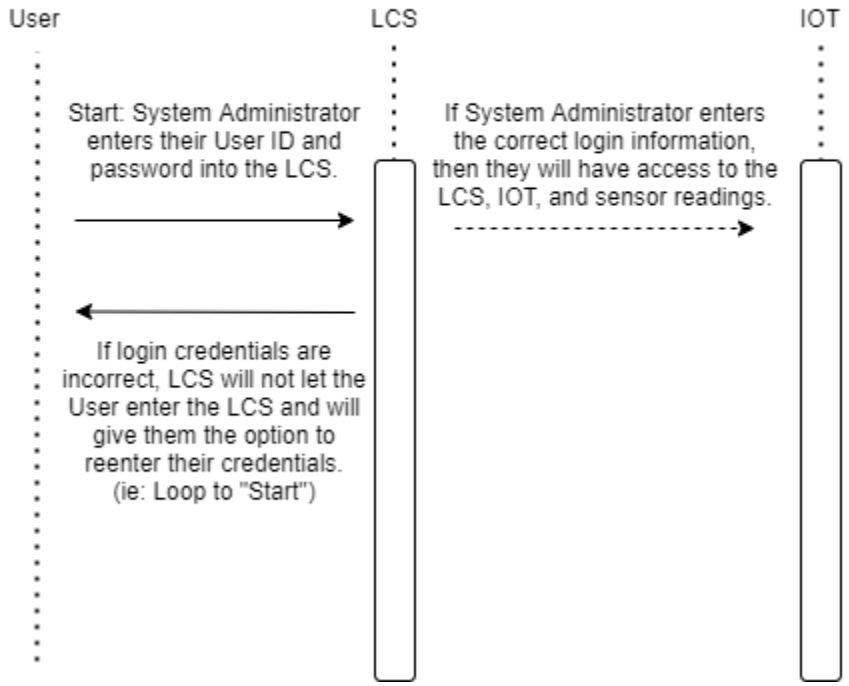
4.5.7: Use cases 12-13



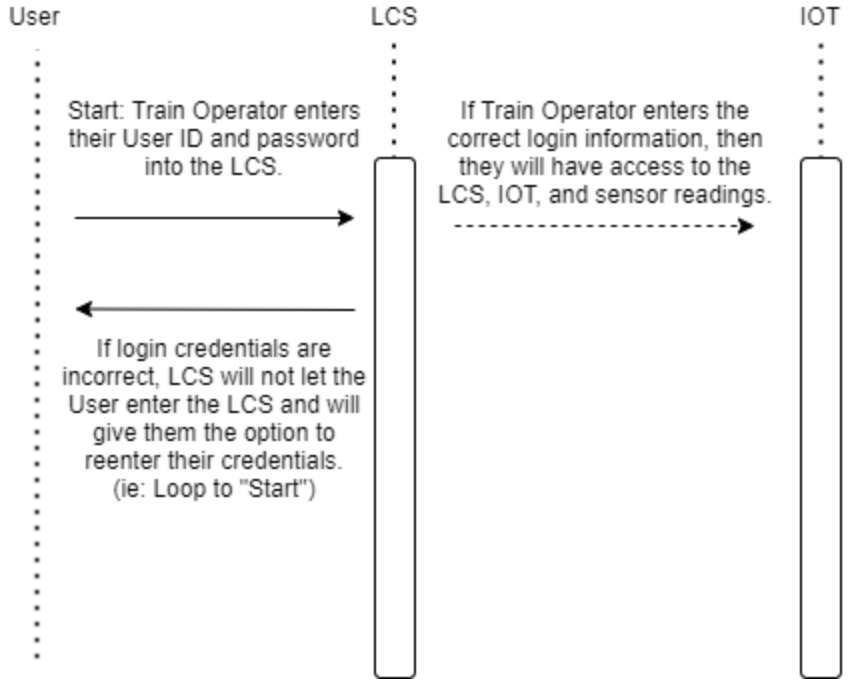
*Note: Implied that the user logged in.

4.6 Sequence Diagrams

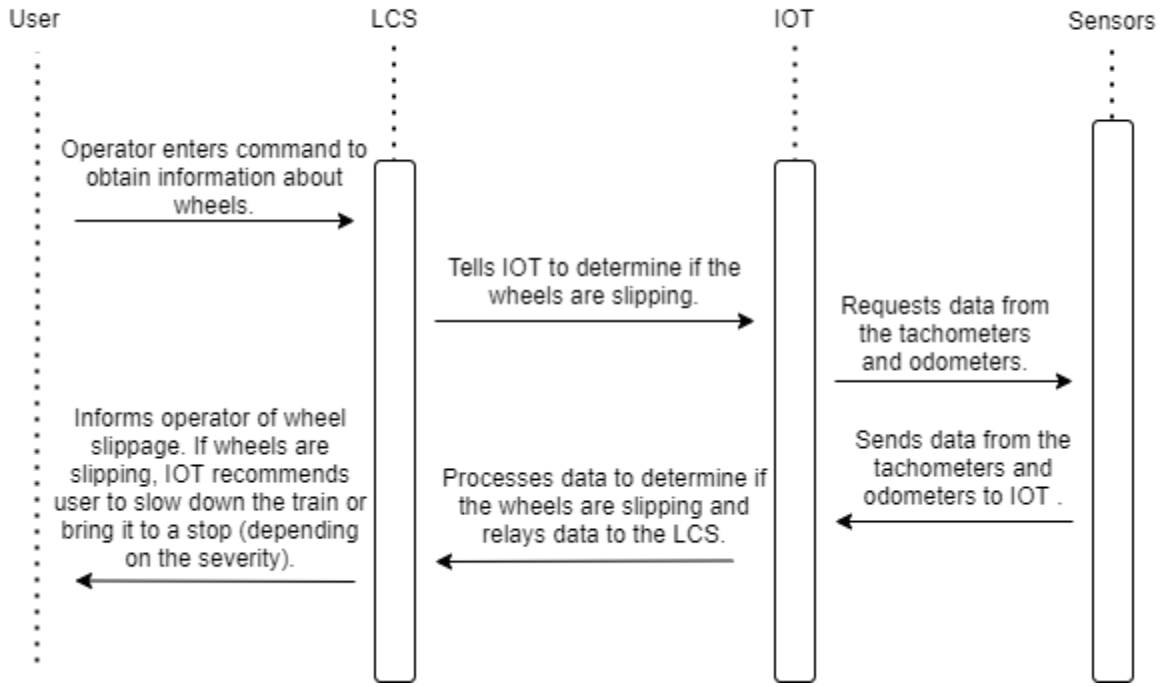
4.6.1: System administrator attempts to log in to LCS



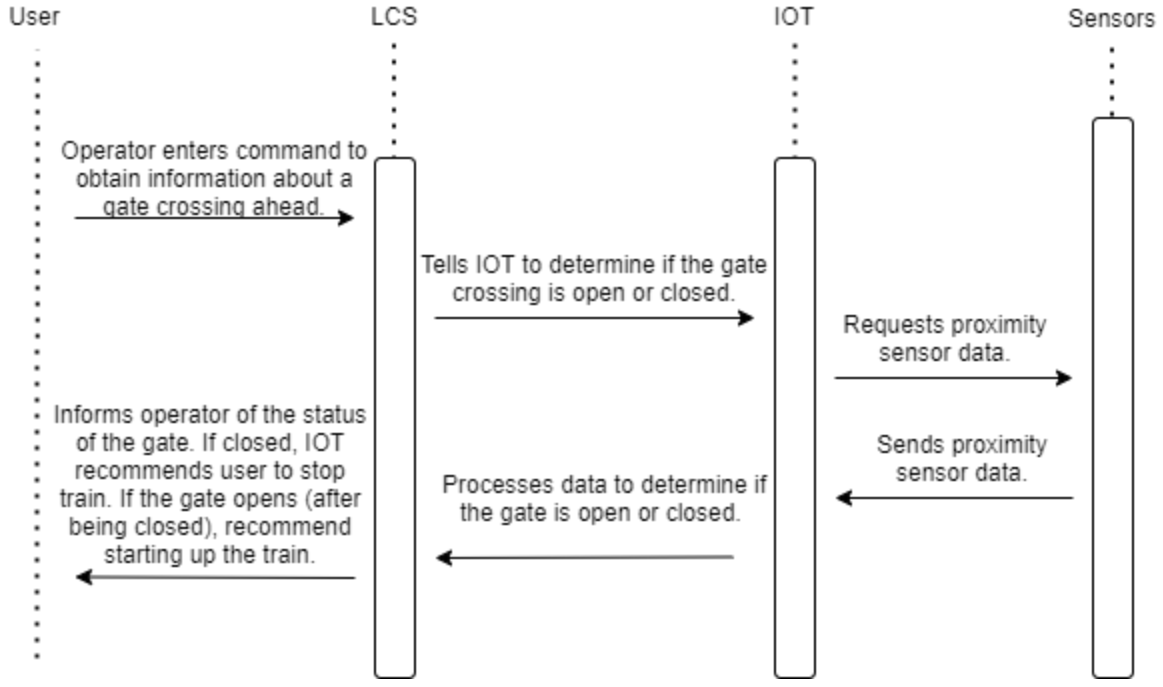
4.6.2: Train Operator attempts to log in to LCS



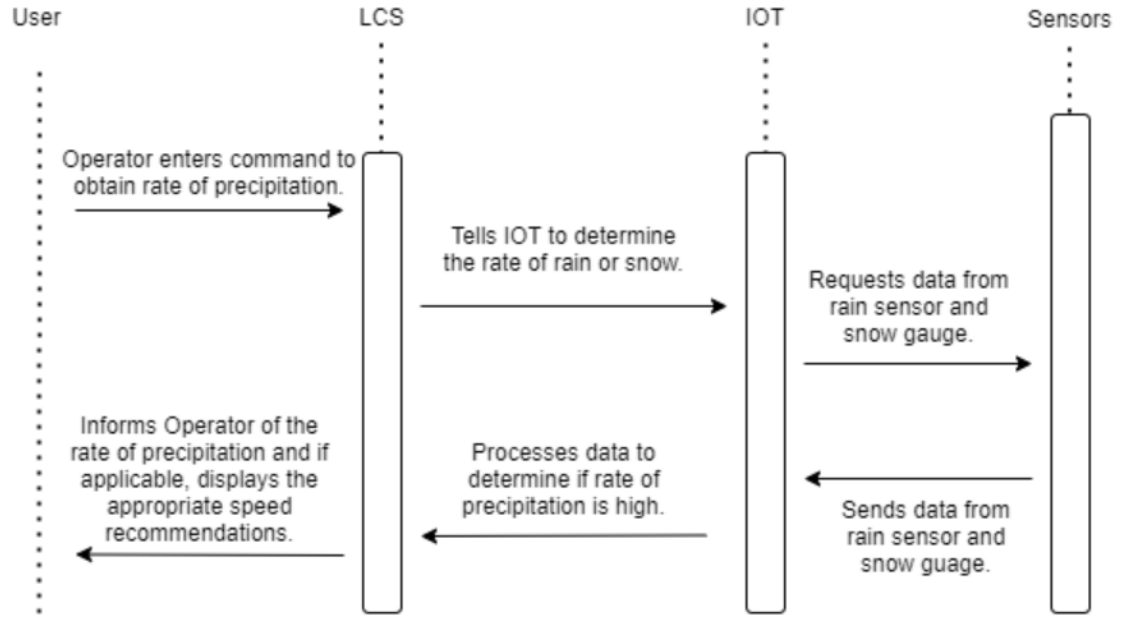
4.6.3: Determine if wheels are slipping



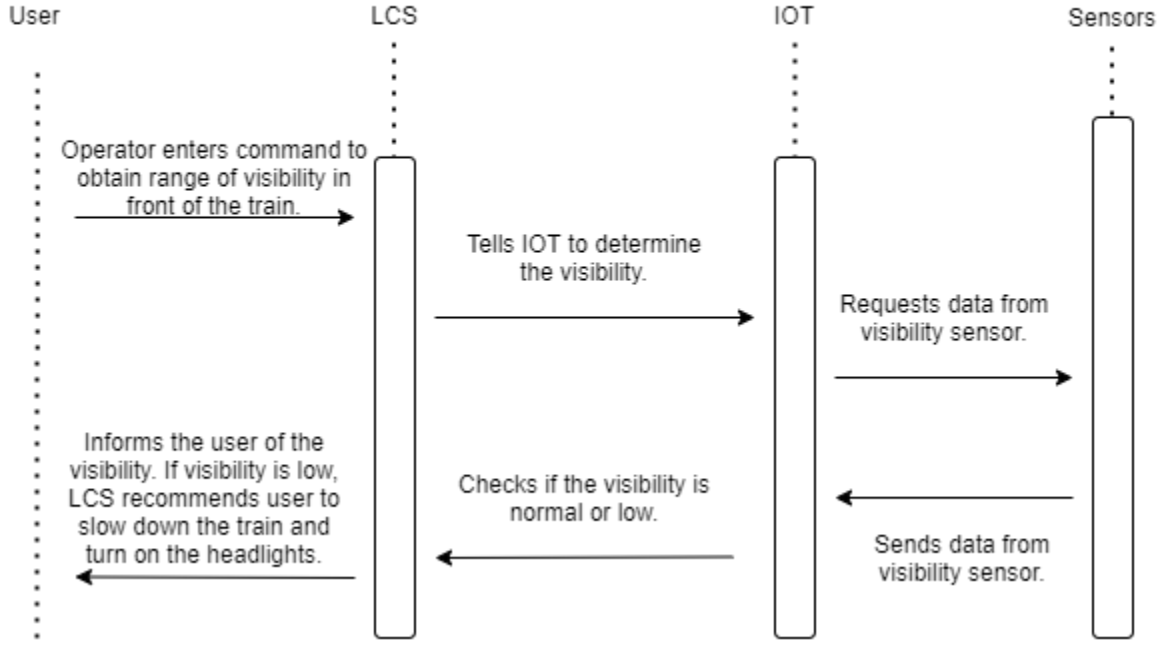
4.6.4: Determine if a gate crossing is open or closed



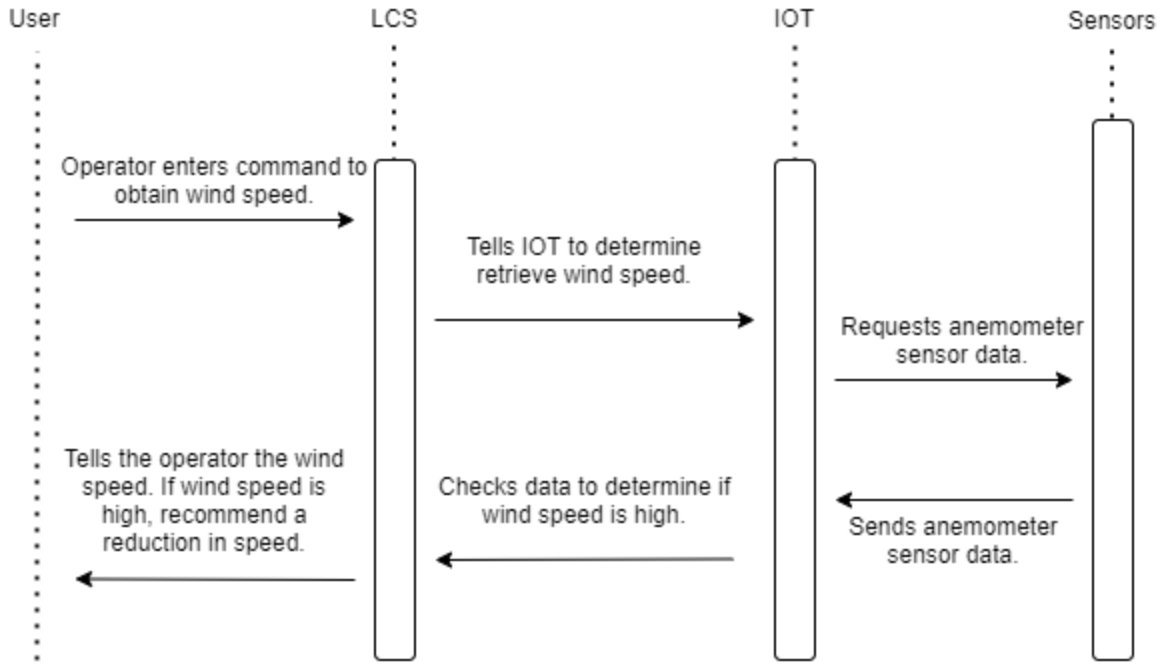
4.6.5: Determine rate of precipitation



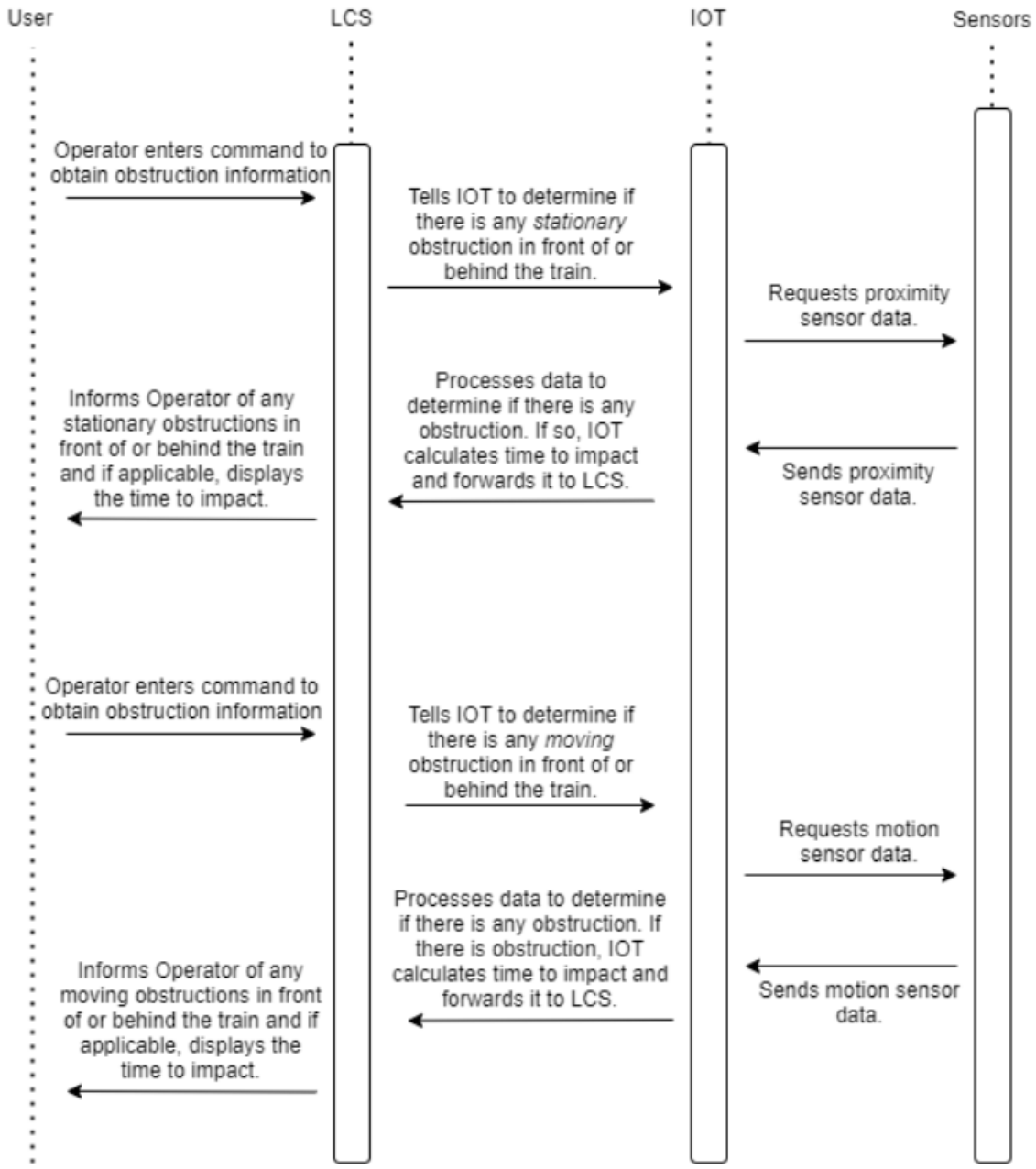
4.6.6: Determine visibility



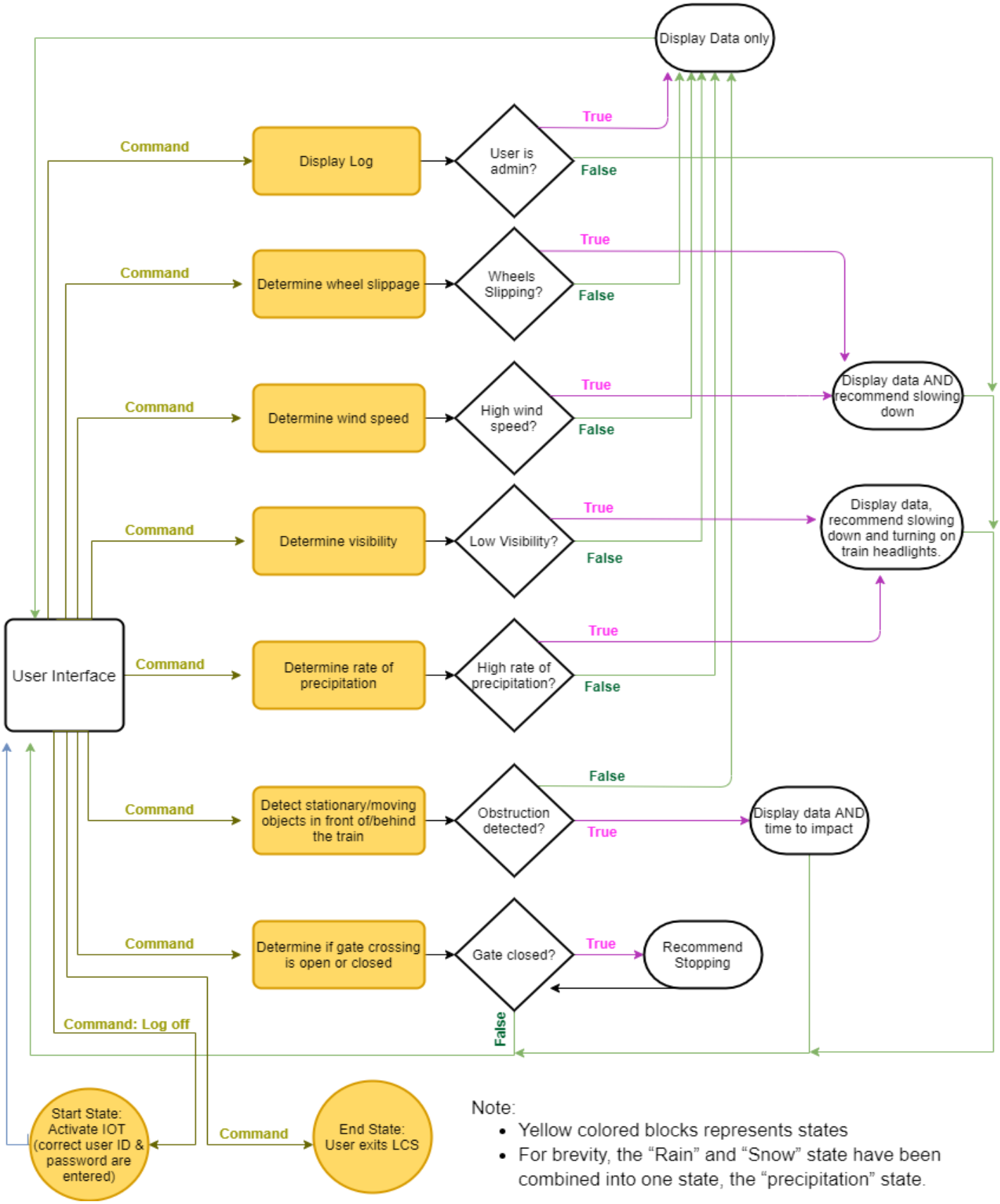
4.6.7: Determine wind speed



4.6.8: Detect moving/stationary objects in front of/behind the train



4.7 State Diagram



Note:

- Yellow colored blocks represents states
- For brevity, the "Rain" and "Snow" state have been combined into one state, the "precipitation" state.

Section 5: Software Architecture

5.1: Architectural Models Pros & Cons Table:

Types of architecture:	Pros	Cons
Data-Centered Architecture	-Simple abstraction -Client softwares (Requirements) operate independently -Promotes integrability -Easy data storage -IOT will always know where to get the data from	-Requirements represented as client softwares -Requires a centralized storage repository -Requires lots of processing, which can cause delays -In our implementation of IOT we do not need the multiple clients in our model.
Data-Flow architecture	-Efficient data usage -Filters work independently of each other -Data flow is compartmentalized -Data fed into IOT will be processed thoroughly by the filters	-Requires implementation of many filters & pipes -Will require a fair amount of data processing, which can cause delays -In our implementation of IOT it would be complicated to add pipes and filters.
Call-and-Return architecture	-Compartmentalized data flow -Easy to modify and scale -Simple abstraction -New functionality can be easily added to IOT	-Requires implementation of multiple subprograms -Subprograms dependent on main program -Parallel processing may be difficult to achieve -In our implementation of IOT we do not have many subprograms and instead have a class that can be more general.
Object-Oriented architecture	-Simple abstraction -Simple implementation for simple class hierarchies or connections (inheritance	-Class and relationship hierarchies can become complex -Communication between

	and polymorphism) -Complements how we currently have our program setup (Sensors, IOT, LCS, GUI classes)	components accomplished via message passing -Needs to explicitly reference the name and interface of other objects
Layered architecture	-Simple abstraction -Hierarchy and all classes are connected and easy to use each other. -IOT can be easily modified by changing a layer	-Multiple calls to get data -Data passed around too much -Some of the classes don't need to be connected to every class and make unnecessary links. -Performance of IOT may suffer due to multiple calls/data processing involved in said calls -Structuring the systems may be difficult
Model-View-Controller architecture	-Organized well with specific roles -Same amount of classes and use of the classes as the project. -Simple abstraction -Implementation of IOT can be easier to debug/modify/update	-Requires the output in HTML or any web app and it would be complicated to implement in our IOT. -Methods used must be strict in nature

5.2: Architectural Model Features Summary:

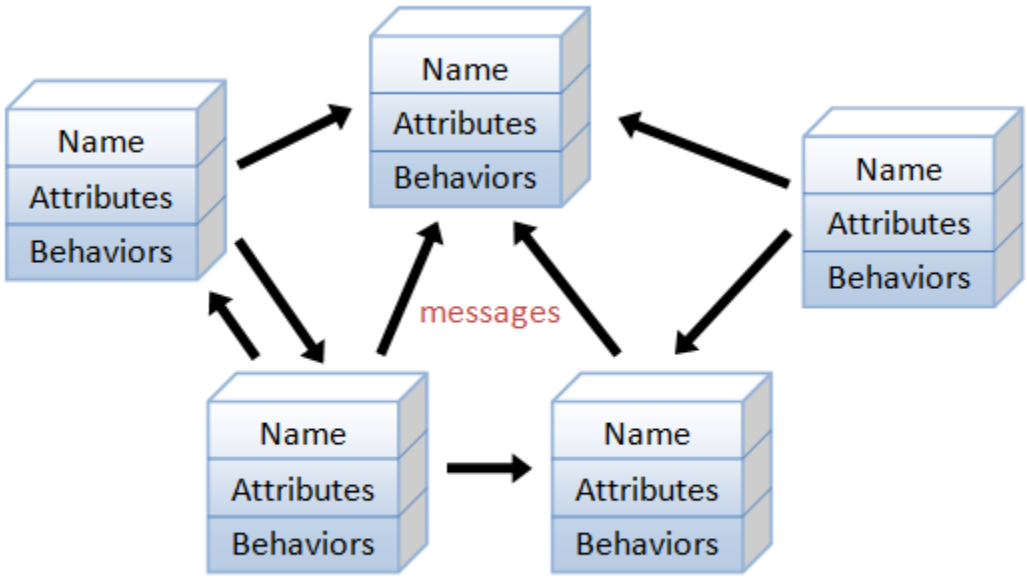
	Simple Abstraction?	Simple Implementation?	Efficient data usage?	Independent components[1]?
Data-Centered architecture	true	neutral	false	true
Data-Flow architecture	false	false	true	true
Call-and-Return architecture	true	false	true	false
Object-Oriented architecture	true	true	false	true
Layered architecture	true	neutral	false	false
Model-View-Controller architecture	true	false	true	false

[1]: Independent Components: includes classes, independent subprograms or independent client softwares.

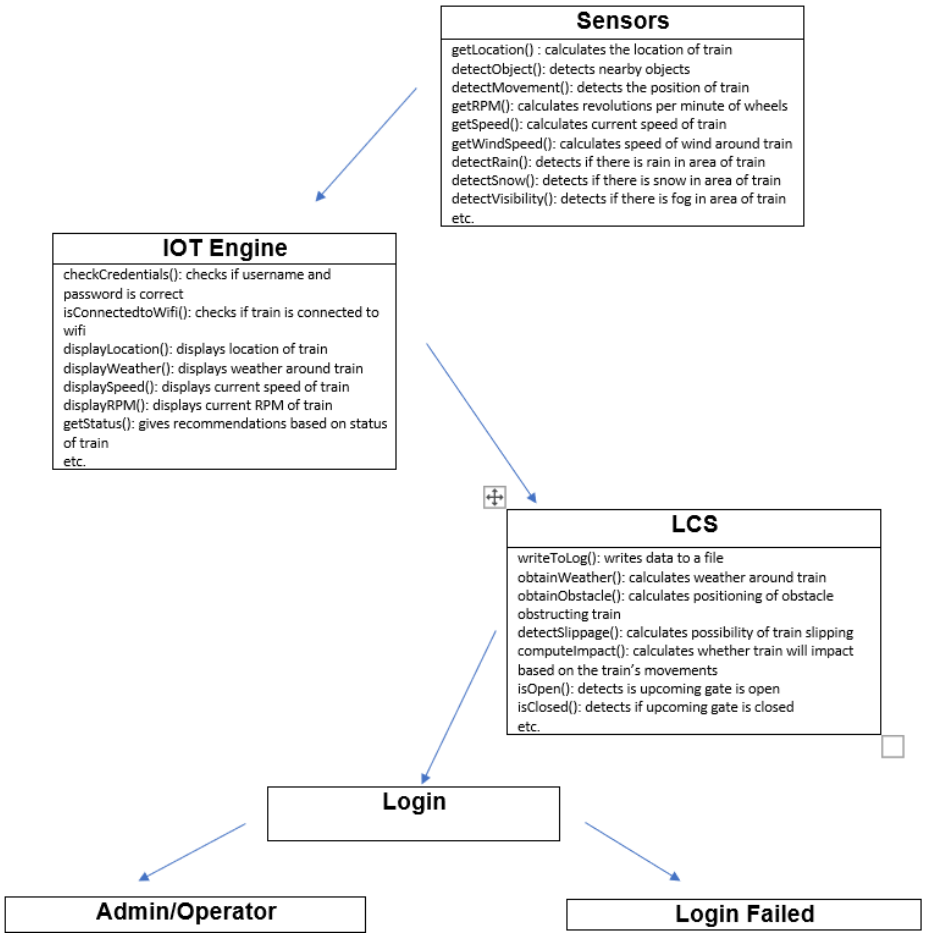
5.3: Tradeoff Analysis of Architectures:

Upon analyzing the different architectural models, we've decided to use the **Object Oriented Architecture** model for our project. The factors we considered are:

- **Economy** – software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility** – Architectural decisions and their justifications should be obvious to software engineers who review.
- **Spacing** – Separation of concerns in a design without introducing hidden dependencies.
- **Symmetry** – Architectural symmetry implies that a system is consistent and balanced in its attributes.
- **Emergence** – Emergent, self-organized behavior and control are key to creating scalable, efficient, and economic software architectures.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages



Economy: Object-Oriented Architecture reduces unnecessary detail and will result in an uncluttered software. Thus, in terms of the economy, OOA has a simple abstraction and simple implementation. Hence, our group is confident that we can complete the coding in the allotted time. Using OOA will reduce the likelihood of our project failing the deadlines, which in turn will prevent it from becoming too costly in terms of time. Overall, this architecture would be easy to implement because the classes are straightforward and easy to envision.

Visibility: Because of the simplicity of OOA, it's easy to “filter” out the small details and instead focus on the bigger picture. This simple abstraction will make it easier for software engineers to conceptualize the project, which in turn will make implementation easier.

Spacing: In OOA, the hierarchy is straightforward which means that there should not be any unwanted or unnecessary dependencies introduced to the software. Thus, any concerns or issues would be limited within the individual classes rather than the relationships between classes.

Symmetry: The Object-Oriented Architecture has a very symmetrical abstraction (meaning that it's balanced & consistent). This symmetry not only simplifies the abstraction, but would also allow us to implement the software efficiently and within the deadline.

Emergence: OOA allows for decent scalability and high organization. If in the future, we need to add new features to the software, we will be able to do so because of OOA. Following an architecture that promotes organization and scalability, future-proofs our software and will extend its lifespan.

Although Object-Oriented Architecture isn't perfect, when considering the five factors mentioned above, it is still the best choice for this project. The following paragraphs will explain the reasons why we did not go with the other architectures.

Data Centered Architecture is not an ideal option for the project because of the poor emergence and economic factor. Although DCA is highly symmetrical and has independent components (which promotes modularity and integrability), it is not necessarily guaranteed to yield a simple implementation. A difficult implementation process has the potential to completely derail our timeline for the project. This is a risk that we are unwilling to take. Furthermore, the client-based model with centralized data storage could run the risk of inefficiently using data.

Data Flow Architecture is perhaps the least beneficial option for us. It has neither a simple abstraction nor a simple implementation. In other words, this architecture has poor economy, poor visibility, and has the potential to have poor spacing. Although DFA has efficient data usage and independent components, we do not believe that using this architecture is worth the risk of not meeting our deadline.

Call & Return architecture has simple abstraction and efficient data usage. However, it does not have a simple implementation. In other words, while this architecture's economy is good, its visibility and spacing are not.

Layered architecture is another option that offers little to no benefit for our project. While it has simple abstraction, its implementation can be trickier to do and thus, has a poor economic factor. Its inefficient data usage, as well as dependencies between components also make the architecture unviable in terms of spacing, symmetry, and emergence.

Model-view-controller Architecture has simple implementation and is efficient with the data. However, the architecture would be hard to implement due to the usage of HTML. The group will be planning to code in Java and it will be complex to implement HTML UI. This would be poor economically due to the program being harder to implement and more time and effort will be needed. Also, MVCA does not have independent components and will lead to hidden dependencies and non-balanced attributes. This would lead to poor spacing and symmetry.

Overall, object-oriented architecture is the only architecture that has both a simple abstraction and a simple implementation. However, this architecture does have its limitations. More specifically, while its economy, visibility, and spacing are good, its symmetry and emergence are mediocre compared to some of the other architectures.

5.4: Output Display

The display of the output to the LCS will depend on the output of the IOT. Below is a comprehensive list of all of the possible inputs & outputs to/from the LCS.

Input into LCS	**Conditions**	**Output Displayed to LCS**
Train operator or System administrator enters their login credentials.	Login credentials are invalid.	Popup: “Incorrect Credentials”
Train operator or System administrator enters their login credentials.	Login credentials are valid.	Displays LCS user interface.
User enters a command to access the log.	User is the system administrator.	LCS displayed on the screen.
User enters a command to access the log.	User is not a system administrator.	“Error: Only the admin can access the log.”
Train operator enters command to access gate crossing information.	Next gate crossing is more than two miles away and is open.	“The next gate, which is [distance] miles away, is open.”
Train operator enters command to access gate crossing information.	Next gate crossing is more than two miles away and is closed.	“The next gate, which is [distance] miles away, is closed.”
[None]	Next gate crossing is open and is less than 2 miles away.	“The next gate, which is [distance] miles away, is open. You may proceed.”
[None]	Next gate crossing is closed and is less than 2 miles away.	“The next gate, which is [distance] miles away, is closed. Recommendation: Stop the train immediately & wait for the gate to open.”
[None]	A gate crossing which is less than 2 miles away opens after being closed.	“The next gate, which is [distance] miles away, is open. You may proceed.”
Train operator enters a	There is a stationary object in	“There is an obstruction

command to determine if there is an object in front of or behind the train.	front of the train.	[distance] miles in front of the train. Obstruction is stationary. Estimated time to impact: [time] minutes.”
Train operator enters a command to determine if there is an object in front of or behind the train.	There is a stationary object behind the train.	“There is an obstruction [distance] miles behind the train. Obstruction is stationary.”
Train operator enters a command to determine if there is an object in front of or behind the train.	There is a moving object in front of the train.	“There is an obstruction [distance] miles in front of the train. Obstruction is moving. Its speed is [speed] mph. Estimated time to impact: [time] minutes.”
Train operator enters a command to determine if there is an object in front of or behind the train.	There is a moving object behind the train.	“There is an obstruction [distance] miles behind the train. Obstruction is moving. Its speed is [speed] mph.”
Train operator enters a command to determine if there is an object in front of or behind the train.	There is no obstruction in front of or behind the train.	“There is no obstruction.”
Train operator enters a command to access information about wheel slippage.	The wheels are slipping.	“The wheels are slipping. Recommendation: slow down or halt the train.”
Train operator enters a command to access information about wheel slippage.	The wheels are not slipping.	“The wheels are not slipping.”
Train operator enters a command to access wind speed information.	The wind speed is less than 50 mph.	“The wind speed is [speed] mph.”
Train operator enters a command to access wind speed information.	The wind speed is greater than or equal to 50 mph.	“The wind speed is [speed] mph. Recommendation: Reduce the speed of the train.”
[None]	The sensors detect that the wind speed is greater than 50 mph.	“The wind speed is [speed] mph. Recommendation: Reduce the speed of the train.”

Train operator enters a command to access information about rain.	The rate of rainfall is greater than or equal to 0.3 inches/hour.	“The rate of rainfall is [rate] inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.”
Train operator enters a command to access information about snow.	The rate of snowfall is greater than or equal to 0.3 inches/hour.	“The rate of snowfall is [rate] inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.”
Train operator enters a command to access information about rain.	The rate of rainfall is less than 0.3 inches/hour.	“The rate of rainfall is [rate] inches per hour.”
Train operator enters a command to access information about snow.	The rate of snowfall is less than 0.3 inches/hour.	“The rate of snowfall is [rate] inches per hour.”
[None]	The rain sensors detect that the rate of rainfall is greater than or equal to 0.3 inches/hour.	“The rate of rainfall is [rate] inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.”
[None]	The snow gauges detect that the rate of snowfall is greater than or equal to 0.3 inches/hour.	“The rate of snowfall is [rate] inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.”
Train operator enters a command to access information about the visibility.	The visibility is greater than or equal to 2 miles.	“The visibility is at least 2 miles.”
Train operator enters a command to access information about the visibility.	The visibility is less than 2 miles.	“The visibility is [distance] miles. Recommendation: Turn on the train headlights and reduce the speed of the train.”

[None]	The visibility sensor detects that the visibility is less than 2 miles.	“The visibility is [distance] miles. Recommendation: Turn on the train headlights and reduce the speed of the train.”

Section 6: Code

Sensors.java

```
package htrPackage;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;

public class Sensors {

	/* hardware: */
	public final double WHEEL_DIAMETER = 85.0; /* Wheel diameter in inches */
	public final long time = 10; /* How often to refresh the data (in seconds) */

	/* Data fields: */
	private int lastoff = 0;
	private double longitude1; /* miles from origin */
	private double latitude1; /* miles from origin */

	private double longitude2; /* miles from origin */
	private double latitude2; /* miles from origin */

	private double gate_distance; /* miles */
	private boolean gate_status; /* true = gate open. false = gate closed. */

	private boolean moving_obstruction; /* true = is obstruction. */
	private boolean stationary_obstruction; /* true = is obstruction. */
	private double distance_from_obstruction; /* [-2,2] miles */

	private int rpm;

	private double speed; /* mph */
	private double wind_speed; /* mph */

	private double rate_rain; /* inches per hour */
	private double rate_snow; /* inches per hour */

	private double visibility; /* miles */
	private ArrayList<String> data;
```

```
/**
 * constructor for Sensor object.
 */
protected Sensors() {
	/* Initialize the sensor object with the following default values */

	this.longitude1 = 0.00000; /* most recent longitude */
	this.latitude1 = 0.00000; /* most recent latitude */
	this.longitude2 = 2.0000; /* older longitude */
	this.latitude2 = 3.0000; /* older latitude */
	this.gate_distance = 4.58; /* miles */
	this.gate_status = true; /* false = closed. true = open. */
	this.moving_obstruction = false;
	this.stationary_obstruction = false;
	this.distance_from_obstruction = 2.0;
	this.rpm = 514;
	this.speed = 130.0; /* mph */
	this.wind_speed = 20.0; /* mph */
	this.rate_rain = 0.0; /* inches per hour */
	this.rate_snow = 0.0; /* inches per hour */
	this.visibility = 2.0; /* miles */
	updateValues();
}

public void updateValues() {
	data = new ArrayList<String>();

	try {

		Scanner input = new Scanner(System.in);

		File file = new File("test.txt");

		input = new Scanner(file);

		while (input.hasNextLine()) {
			String line = input.nextLine();
			data.add(line);
		}
	} catch (Exception ex) {
		ex.printStackTrace();
	}
}
```

```
/**
 * Map of data's indices to the data it represents. 0 - rpm 1-lat 2 long 3- gate
 * distance 4 - gate status 5- moving obstruction 6- stationary 7-distance 8-
 * rain 9- snow 10- wind 11- visibility
 */
public void updateValuesSensors() {

	rpm = Integer.valueOf((data.get(lastoff)));
	lastoff++;
	longitude2 = longitude1;
	longitude1 = Double.valueOf((data.get(lastoff)));
	lastoff++;
	latitude2 = latitude1;
	latitude1 = Double.valueOf((data.get(lastoff)));
	lastoff++;
	gate_distance = Double.valueOf((data.get(lastoff)));
	lastoff++;
	if (data.get(lastoff).equals("True")) {
		gate_status = true;
		lastoff++;
	} else {
		gate_status = false;
		lastoff++;
	}
	if (data.get(lastoff).equals("True")) {
		moving_obstruction = true;
		lastoff++;
	} else {
		moving_obstruction = false;
		lastoff++;
	}
	if (data.get(lastoff).equals("True")) {
		stationary_obstruction = true;
		lastoff++;
	} else {
		stationary_obstruction = false;
		lastoff++;
	}
	distance_from_obstruction = Double.valueOf((data.get(lastoff)));
	lastoff++;
	rate_rain = Double.valueOf((data.get(lastoff)));
	lastoff++;
	rate_snow = Double.valueOf((data.get(lastoff)));
	lastoff++;
```

```
            wind_speed = Double.valueOf((data.get(lastoff)));
            lastoff++;
            visibility = Double.valueOf((data.get(lastoff)));
            lastoff++;
            setSpeed();
    }

    /**
     * Getter for time.
     */
    public long getTime() {
            return this.time;
    }

    /**
     * @return the wheel diameter (in inches).
     */
    double getWheelDiameter() {
            return WHEEL_DIAMETER;
    }

    /**
     * @return the most recent latitude.
     */
    double getLatitude1() {
            return this.latitude1;
    }

    /**
     * @return the most recent longitude.
     */
    double getLongitude1() {
            return this.longitude1;
    }

    /**
     * @return the old latitude.
     */
    double getLatitude2() {
            return this.latitude2;
    }

    /**
     * @return the old longitude.
```

```
 */
double getLongitude2() {
	return this.longitude2;
}

/**
 * Set the distance to the next gate.
 */
void setGateDistance(double dist) {
	this.gate_distance = dist;
}

/**
 * @return get the distance to the next gate opening (in miles).
 */
double getGateDistance() {
	return this.gate_distance;
}

/**
 * Set the gate_status data field to true if next gate is open, otherwise false.
 */
void setGateStatus() {
	this.gate_status = true;
}

/**
 * @return true if next gate is open, false otherwise.
 */
boolean getGateStatus() {
	return this.gate_status;
}

/**
 * @return true if a a stationary object is detected, false otherwise.
 */
boolean getDetectStationaryObject() {
	return this.stationary_obstruction;
}

/**
 * @return true if a a moving object is detected, false otherwise.
 */
boolean getDetectMovingObject() {
```

```
		return this.moving_obstruction;
	}

	/**
	 * Sets then gets distance from object.
	 * @return a negative double to signify obstruction's distance behind the train,
	 *	and a non-negative double to signify distance in front of the train.
	 */
	double obtainDistanceFromObject() {
		return this.distance_from_obstruction;
	}

	/**
	 * Set the rpm.
	 */
	void setRPM(int rpm) {
		if (rpm < 0 || rpm > 5000) {
			return;
		}
		this.rpm = rpm;
	}

	/**
	 * @return the rpm.
	 */
	int getRPM() {
		return this.rpm;
	}

	/**
	 * Set the speed.
	 */
	void setSpeed() {
		double deltaLat = Math.abs(this.latitude1 - this.latitude2);
		double deltaLong = Math.abs(this.longitude1 - this.longitude2);
		this.speed = Math.rint((3600/this.getTime()) * Math.sqrt(Math.pow(deltaLat, 2.0) +
Math.pow(deltaLong, 2.0)));
	}

	/**
	 * @return the speed in miles per hour.
	 */
	double getSpeed() {
		return this.speed;
```

```
}

/**
 * Set the speed.
 */
void setWindSpeed(double wind_speed) {
	if (wind_speed < 0 || wind_speed > 400) {
		return;
	}
	this.wind_speed = wind_speed;
}

/**
 * @return the wind speed (miles per hour).
 */
double getWindSpeed() {
	return this.wind_speed;
}

/**
 * Set the rain rate.
 */
void setRainRate(double rate) {
	this.rate_rain = rate;
}

/**
 * @return the rain rate (inches per hour).
 */
double getRainRate() {
	return this.rate_rain;
}

/**
 * Set the snow rate.
 */
void setSnowRate(double rate) {
	this.rate_snow = rate;
}

/**
 * @return the snow rate (inches per hour).
 */
double getSnowRate() {
```

```
		return this.rate_snow;
	}

	/**
	 * Set the visibility.
	 */
	void setVisibility(double visibility) {
		if (visibility < 0 || visibility > 2.0) {
			return;
		}
		this.visibility = visibility;
	}

	/**
	 * @return the visibility (in miles). Range of visibility is b/w 0.0 & 2.0 miles
	 *	inclusive.
	 */
	double getVisibility() {
		return this.visibility;
	}

}
```

IOT.java

```
package htrPackage;

public class IOT extends Sensors {

	/**
	 * constructor for IOT object.
	 */
	protected IOT() {
		super();
	}

	/**
	 * Rounds a double to 2 decimal places.
	 */
	static double roundTwoDecimals(double db) {
		return ((double) ((int) (db * 100))) / 100;
	}
```

```
	/**
	 * @return the windreport. Includes recommendation if windspeed is >= 50.
	 */
	String windReport() {
		String data = "The wind speed is " + roundTwoDecimals(this.getWindSpeed()) + "
mph. ";
		String rec = "";
		if (this.getWindSpeed() >= 50.0) {
			rec += "\nRecommendation: Reduce the speed of the train.";
		}
		return data + rec;
	}

	/**
	 * @return the rain report. Includes recommendation if rainRate is >= 0.3.
	 */
	String rainReport() {
		String data = "The rate of rainfall is " + roundTwoDecimals(this.getRainRate()) + "
inches per hour.";
		String rec = "";
		if (this.getRainRate() >= 0.3) {
			rec += "\nRecommendation: Turn on the train headlights and reduce the
speed of the train.";
			return "The rate of rainfall is " + roundTwoDecimals(this.getRainRate()) +
" inches per hour."
					+ "\nRecommendation: Turn on the train headlights and
reduce the speed of the train.";
		}
		return data + rec;
	}

	/**
	 * @return the snow report. Includes recommendation if snowRate is >= 0.3.
	 */
	String snowReport() {
		String data = "The rate of snowfall is " + roundTwoDecimals(this.getSnowRate())
+ " inches per hour.";
		String rec = "";
		if (this.getSnowRate() >= 0.3) {
			rec += "\nRecommendation: Turn on the train headlights and reduce the
speed of the train.";
		}
		return data + rec;
```

```
	}

	/**
	 * @return the visibility report. Includes recommendation if visibility is < 2.0
	 *         miles.
	 */
	String visibilityReport() {
		if (this.getVisibility() < 2.0) {
			return "The visibility is " + roundTwoDecimals(this.getVisibility())
					+ " miles.\nRecommendation: Turn on the train" + "
headlights and reduce the speed of the train.";
		} else {
			return "The visibility is at least 2 miles.";
		}
	}

	/**
	 * @return true if there is an obstruction. False otherwise.
	 */
	boolean isObstruction() {
		// return true;
		return this.getDetectStationaryObject() || getDetectMovingObject();
	}

	/**
	 * @returns the speed of the object relative to the train.
	 */
	double objectSpeed() {
		return (1 + (Math.random() * ((15 - 1) + 1))); /* generate random speed b/w [1,15]
*/
	}

	/**
	 *
	 * @return the number of seconds to impact.
	 */
	double computeImpact() {
		return (((this.obtainDistanceFromObject() * 5280) / (88 * this.getSpeed())));
	}

	/**
	 *
	 * Determines if there is an obstruction.
	 * If there is, determine:
```

```
	 *		whether its in front of or behind the train,
	 *		its speed,
	 *		whether the obstruction is moving or stationary,
	 *		its distance away from the train,
	 *		and the time to collision, if applicable.
	 */
	String ProcessObject() {
		StringBuilder str = new StringBuilder();

		if (!this.isObstruction()) {
			str.append("There is no obstruction.");
			return str.toString();
		} else {
			if (this.obtainDistanceFromObject() < 0) { /* obstruction behind train */
				str.append("There is an obstruction " +
Math.abs(roundTwoDecimals(this.obtainDistanceFromObject()))
						+ " miles behind the train.");
				if (this.getDetectMovingObject()) {
					str.append(" Obstruction is moving. Its speed is " +
roundTwoDecimals(this.objectSpeed()) + " mph.");
				} else {
					str.append(" Obstruction is stationary.");
				}
				return str.toString();
			} else { /* obstruction in front of train */
				str.append("There is an obstruction " +
roundTwoDecimals(this.obtainDistanceFromObject())
						+ " miles in front of the train.");
				if (this.getDetectMovingObject()) {
					str.append(" Obstruction is moving. Its speed is " +
roundTwoDecimals(this.objectSpeed()) + " mph.");
				} else {
					str.append(" Obstruction is stationary.");
				}
				if (this.getSpeed() != 0) {
					str.append(" Estimated time to impact: " +
roundTwoDecimals(this.computeImpact()) + " minutes.");
				}
				return str.toString();
			}
		}

	}
```

```
	/**
	 * Calculate wheel slippage. If the wheels are slipping, then print out a
	 * recommendation to slow down the train.
	 */
	String detectSlippage() {
//		double rpmSpeed = (double) this.getRPM() * this.getWheelDiameter() * Math.PI *
(double) 60 / (double) 63360;
//		if (Math.abs(
//				this.getSpeed() - rpmSpeed) > (this.getSpeed() * 0.05)) {
//			return "The wheels are slipping. Recommendation: slow down or halt the
train.";
//		} else {
//			return "The wheels are not slipping.";
//		}
		if (this.getSnowRate() >= 0.2 || this.getRainRate() >= 0.3) {
			return "The wheels are slipping.\nRecommendation: slow down or halt the
train.";
		} else {
			return "The wheels are not slipping.";
		}
	}

	/**
	 * Processed the gate status.
	 *
	 * @returns a string which contains details about upcoming gate crossings.
	 *		String will contain the appropriate recommendations, if any.
	 */
	String gateStatus() {
		StringBuilder str = new StringBuilder();

		if (this.getGateDistance() > 2.0) {
			if (this.getGateStatus()) { // true if gate is open, false if closed.
				str.append("The next gate, which is " +
roundTwoDecimals(this.getGateDistance())
						+ " miles away, is open. ");
			} else {
				str.append("The next gate, which is " +
roundTwoDecimals(this.getGateDistance())
						+ " miles away, is closed.");
			}
		} else {
			if (this.getGateStatus()) {
```

```
                        str.append("The next gate, which is " +
roundTwoDecimals(this.getGateDistance())
                                        + " miles away, is open. ");
                        if (!this.isObstruction()) {
                                str.append("You may proceed.");
                        }
                } else {
                        str.append("The next gate, which is " +
roundTwoDecimals(this.getGateDistance())
                                        + " miles away, is closed."
                                        + "\nRecommendation: Stop the train immediately
& wait for the gate to open.\n");
                }
        }

        /* Dont honk if the gate is closed & there is an obstruction */
        if (this.getSpeed() != 0) {
                double deltaDist = 0.0;
                /* approximation of distance the train will travel in miles in the next 'time'
seconds. */
                deltaDist = (this.getTime() * this.getSpeed() / 3600);

                /*
                 * We have to approximate the gate dist w/ our curr speed. yes. Gate dist
after
                 * 'time' sec.
                 *
                 * Honk the horn if the gate distance is b/w [1, 1 + deltaDist).
                 *   1 + deltaDist is the approximation of the gateDistance after 15
seconds.
                 */
                if (this.getGateDistance() >= 1 && this.getGateDistance() < 1 + deltaDist)
{
                        str.append("Gate is about 1 mile away.\nRecommendation: Honk
the horn for 15 seconds.");
                }

                /*
                 * Honk the horn if the gate distance is b/w [0, deltaDist).
                 * deltaDist is the approximation of the gateDistance after 15 seconds.
                 */
                if ( (this.getGateDistance() >= 0 && this.getGateDistance() <  deltaDist) ||
(this.getGateDistance() <= 0.1)) {
```

```
                                str.append(" About to cross a gate.\nRecommendation: Honk the
horn for 5 seconds.");
                        }
                }

                return str.toString();
        }
}
```

LCS.java

```
package htrPackage;

import java.util.HashMap;
import java.util.Map;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Scanner;

public class LCS extends IOT {

        /* data fields */
        private boolean isConnected; /* tracks if the LCS system is connected to wifi */
        private boolean isLoggedIn; /* tracks if the user is logged into LCS */
        private boolean wantToCont; /* tracks if the user wants to exit from LCS */

        private Path logFileName;
        private File log; /* LCS log stores lots of information */

        String user = "";
        String pass = "";

        /* Use a HashMap to store Login information. */
        /* Add default login info for operator & administrator */
        private Map<String, String> login_info = new HashMap<String, String>();

        /**
         * Constructor for LCS object.
         */
```

```
	protected LCS() {
		super();
		this.isConnected = false; /* By default, assume there is no wifi connection. */
		this.isLoggedIn = false;
		this.wantToCont = true;

		/* Initialize the default login credentials. */
		this.addLoginInfo("operator", "qwerty");
		this.addLoginInfo("admin", "password");

		/* Initialize the log for this session. */
		try {
			this.initializeLog();
		} catch (Exception e) {
			System.out.println("Error: Unable to initialize log.\n");
			e.printStackTrace();
		}
	}

	/* Regular methods: */

	/* Log-related methods */

	void initializeLog() throws Exception {
		String filename = "LCS_Log.txt";

		/* create the log file: */
		try {
			this.log = new File(filename);
			this.log.createNewFile();
		} catch (Exception e) {
			System.out.println("An error occurred.\n");
		}

		/* set its filename/path and write a line to it */
		this.logFileName = Path.of(filename);
		try {
			/* Initialize the log to be written to & read from */
			Files.write(this.logFileName,
"****************************************\n".getBytes(),
					StandardOpenOption.APPEND);
		} catch (IOException e) {
			/* Unable to initialize log */
			e.printStackTrace();
```

```
		}

	}

	void writeToLog(String str) {
		try {
			Files.write(this.logFileName, str.getBytes(),
StandardOpenOption.APPEND);
		} catch (IOException e) {
			/* Unable to write to log */
			e.printStackTrace();
		}
	}

	String readFromLog() {
		String data = "";
		try {
			Scanner myReader = new Scanner(this.log);
			while (myReader.hasNextLine()) {
				data += myReader.nextLine();
				data += "\n";
			}
			data += "*************** END OF LOG ***************\n";
			myReader.close();
		} catch (FileNotFoundException e) {
			System.out.println("Error: Unable to read from the log.\n");
			e.printStackTrace();
		}
		return data;
	}

	/* Log in Credential Methods */

	/**
	 * If the given username and password are valid (at least 5 chars long), then
	 * add the (username,password) pair to the table of valid login credentials.
	 */
	void addLoginInfo(String username, String password) {
		if (username.length() < 5 || password.length() < 5) {
			System.out
					.println("Error: Failed to add new user. Username " + "&
password must be at least 5 characters.");
			return;
		}
```

```
		this.getLoginInfo().put(username, password);

	}

	/**
	 * @returns the Map containing all usernames & passwords.
	 */
	Map<String, String> getLoginInfo() {
		return this.login_info;
	}

	/**
	 * @param username
	 * @param password
	 * @return true if the pair (username,password) is a valid login credential,
	 *	false otherwise.
	 */
	protected boolean checkCredentials(String username, String password) {
		if (this.getLoginInfo().getOrDefault(username, "").equals(password)) {
			this.user = username;
			this.pass = password;
			return true;
		} else {
			return false;
		}
	}

	public String getPassword() {
		return this.pass;
	}

	public String getUsername() {
		return this.user;
	}

	void setIsLoggedIn(boolean connection) {
		this.isLoggedIn = connection;
	}

	/* Wifi Connection Methods */
	private void setWifi(boolean b) {
		this.isConnected = b;
	}
```

```
	/**
	 *
	 * @return a string representation of the location in the form: (long,lat)
	 */
	String displayLocation() {
		return "(" + this.getLatitude1() + ", " + this.getLongitude1() + ")";
	}

	/**
	 * ToString for an LCS object.
	 */
	public String toString() {
		StringBuilder str = new StringBuilder();
		str.append("\nLocation: " + this.displayLocation());
		str.append("\nGate Distance: " + roundTwoDecimals(this.getGateDistance()));
		str.append("\nGate Status: " + this.getGateStatus());
		str.append("\nObstruction Present?: " + this.isObstruction());
		str.append("\nRPM: " + this.getRPM());
		str.append("\nSpeed: " + roundTwoDecimals(this.getSpeed()));
		str.append("\nWind Speed: " + roundTwoDecimals(this.getWindSpeed()));
		str.append("\nRain Rate: " + roundTwoDecimals(this.getRainRate()));
		str.append("\nSnow Rate: " + roundTwoDecimals(this.getSnowRate()));
		str.append("\nVisibility: " + roundTwoDecimals(this.getVisibility()) + "\n");
		return str.toString();
	}
}
```

LoginGUI.java

```
package htrPackage;

import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.io.File;
import java.awt.event.ActionEvent;
import java.awt.Color;
import javax.swing.JLabel;
```

```
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import java.util.Date;

public class LoginGUI extends JFrame {

	/**
	 * Data fields
	 */
	private JPanel contentPane;
	private JTextField userField;
	private JPasswordField passField;
	Date date = java.util.Calendar.getInstance().getTime();
	LCS lcs;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					LoginGUI frame = new LoginGUI();
					frame.setTitle("Login");
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public LoginGUI() {

		/**
		 * Create new intance of LCS Update date Write to log that session has started
		 */
		lcs = new LCS();
		date = java.util.Calendar.getInstance().getTime();
		lcs.writeToLog("" + date + "-- LCS session started.\n");
		lcs.writeToLog(lcs.toString());
```

```
		/**
		 * Default block of code with JFrame
		 */
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(325, 100, 900, 600);
		contentPane = new JPanel();
		contentPane.setBackground(Color.GRAY);
		contentPane.setForeground(Color.WHITE);
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);

		/**
		 * Code that adds a train to our login window
		 */
		File img = new File("train.png");
		JLabel lblImage = new JLabel();
		lblImage.setBounds(310, 60, 256, 169);
		lblImage.setText("");
		lblImage.setIcon(new ImageIcon(img.getAbsolutePath()));

		/**
		 * Sets up the username and password labels
		 */
		JLabel lblUser = new JLabel("USERNAME:");
		lblUser.setForeground(Color.GREEN);
		lblUser.setBounds(226, 269, 88, 33);
		JLabel lblPass = new JLabel("PASSWORD:");
		lblPass.setForeground(Color.GREEN);
		lblPass.setBounds(226, 312, 88, 33);
		userField = new JTextField(20);
		userField.setBounds(335, 276, 231, 19);
		passField = new JPasswordField(20);
		passField.setBounds(335, 319, 231, 19);

		/**
		 * Code for the log in button (style, size, functionality, etc.)
		 */
		JButton logIn = new JButton("LOG IN");
		logIn.setBounds(373, 359, 150, 40);
		logIn.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (lcs.checkCredentials(userField.getText(),
String.valueOf(passField.getPassword()))
```

```
                                        && (userField.getText().length() != 0 &&
String.valueOf(passField.getPassword()).length() != 0)
                                        && (userField.getText().equals("operator"))) {
                                    date = java.util.Calendar.getInstance().getTime();
                                    lcs.writeToLog("" + date + "-- User \'" + userField.getText()
+ "\' logged in.\n");
                                    OperatorGUI op = new OperatorGUI();
                                    op.setVisible(true);
                                    dispose();
                                } else if (lcs.checkCredentials(userField.getText(),
String.valueOf(passField.getPassword()))
                                        && (userField.getText().length() != 0 &&
String.valueOf(passField.getPassword()).length() != 0)
                                        && (userField.getText().equals("admin"))) {
                                    date = java.util.Calendar.getInstance().getTime();
                                    lcs.writeToLog("" + date + "-- User \'" + userField.getText()
+ "\' logged in.\n");
                                    AdminGUI ad = new AdminGUI();
                                    ad.setVisible(true);
                                    dispose();
                                } else {
                                    date = java.util.Calendar.getInstance().getTime();
                                    lcs.writeToLog("" + date + "-- Failed login attempt.\n");
                                    LoginFailed fail = new LoginFailed();
                                    fail.setVisible(true);
                                }
                            }
                        });

                        /**
                         * Block of code that adds the components onto the GUI
                         */
                        contentPane.setLayout(null);
                        contentPane.add(lblPass);
                        contentPane.add(lblUser);
                        contentPane.add(passField);
                        contentPane.add(userField);
                        contentPane.add(logIn);
                        contentPane.add(lblImage);
                }
}
```

LoginFailed.java

```
package htrPackage;

import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import java.awt.Color;
import java.awt.Font;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class LoginFailed extends JFrame {

	private JPanel contentPane;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					LoginFailed frame = new LoginFailed();
					frame.setTitle("Error");
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public LoginFailed() {

		/**
```

```
		 * Default block of code with JFrame
		 */

		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(600, 300, 350, 200);
		contentPane = new JPanel();
		contentPane.setBackground(Color.GRAY);
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);

		/**
		 * The text to denote wrong credentials.
		 */

		JLabel lblIncorrect = new JLabel("Incorrect credentials.");
		lblIncorrect.setFont(new Font("Tahoma", Font.BOLD, 14));
		lblIncorrect.setForeground(Color.YELLOW);
		lblIncorrect.setHorizontalAlignment(JLabel.CENTER);
		lblIncorrect.setBounds(63, 49, 212, 33);

		/**
		 * Code for the close button (style, size, functionality)
		 */

		JButton btnClose = new JButton("CLOSE");
		btnClose.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				dispose();
			}
		});
		btnClose.setBounds(123, 92, 85, 21);

		contentPane.add(lblIncorrect);
		contentPane.add(btnClose);
	}
}
```

OperatorGUI.java

```
package htrPackage;
```

```
import java.awt.Color;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextPane;
import javax.swing.border.EmptyBorder;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.UIManager;

public class OperatorGUI extends JFrame {

	/**
	 * Data fields
	 */
	private JPanel contentPane;
	LCS lcs;
	Date date = java.util.Calendar.getInstance().getTime();
	Timer loop;
	int totalMilliseconds;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					OperatorGUI frame = new OperatorGUI();
					frame.setTitle("LCS");
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
```

```
		});
}

/**
 * Create the frame.
 */
public OperatorGUI() {
		/**
		 * Create new instances of LCS and Timer Set isLoggedIn() to true Set date
		 * variable
		 */
		lcs = new LCS();
		lcs.setIsLoggedIn(true);
		loop = new Timer();
		date = java.util.Calendar.getInstance().getTime();

		/**
		 * Sets up the main display area for the outputs and recommendations
		 */
		JScrollPane scroll = new JScrollPane();
		scroll.setBounds(285, 42, 576, 245);
		JTextArea display = new JTextArea();
		display.setForeground(Color.GREEN);
		scroll.setViewportView(display);
		display.setEditable(false);
		display.setWrapStyleWord(true);
		display.setLineWrap(true);
		display.setBackground(Color.BLACK);
		display.setFont(new Font("Monospaced", Font.BOLD, 16));
		display.setText("Welcome, Operator.");

		/**
		 * Sets up the label and display for the speed data
		 */
		JTextPane speedPane = new JTextPane();
		speedPane.setBounds(57, 83, 152, 67);
		speedPane.setEditable(false);
		speedPane.setBackground(Color.BLACK);
		speedPane.setForeground(Color.GREEN);
		speedPane.setFont(new Font("Tahoma", Font.PLAIN, 50));
		speedPane.setText(String.valueOf(lcs.getSpeed()));

		JLabel lblSpeed = new JLabel("SPEED");
		lblSpeed.setBounds(92, 39, 87, 34);
```

```
		lblSpeed.setForeground(Color.YELLOW);
		lblSpeed.setFont(new Font("Tahoma", Font.BOLD, 14));
		lblSpeed.setHorizontalAlignment(JLabel.CENTER);

		/**
		 * Sets up the label and display for the RPM data
		 */
		JLabel lblRPM = new JLabel("RPM");
		lblRPM.setBounds(72, 160, 128, 34);
		lblRPM.setForeground(Color.YELLOW);
		lblRPM.setFont(new Font("Tahoma", Font.BOLD, 14));
		lblRPM.setHorizontalAlignment(JLabel.CENTER);

		JTextPane rpmPane = new JTextPane();
		rpmPane.setBounds(82, 193, 103, 61);
		rpmPane.setEditable(false);
		rpmPane.setBackground(Color.BLACK);
		rpmPane.setForeground(Color.GREEN);
		rpmPane.setFont(new Font("Tahoma", Font.PLAIN, 50));
		rpmPane.setText(String.valueOf(lcs.getRPM()));

		/**
		 * Create and define a TimerTask() to automate file reading, updating, the
		 * output printing
		 */
		TimerTask task = new TimerTask() {
			public void run() {
				totalMilliseconds += lcs.getTime() * 1000;
				lcs.updateValuesSensors();

				speedPane.setText(String.valueOf(lcs.getSpeed()));
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- Speed data has been updated \'" +
"\'.\n");

				rpmPane.setText(String.valueOf(lcs.getRPM()));
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- RPM data has been updated \'" +
"\'.\n");

				display.setText("");

				if (lcs.getRainRate() >= 0.3) {
					display.append(lcs.rainReport() + "\n");
```

```
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Rain data has been updated \'" +
"\'.\n");

                        if (lcs.getWindSpeed() >= 50.0) {
                                display.append(lcs.windReport() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Wind data has been updated \'" +
"\'.\n");

                        if (lcs.getSnowRate() >= 0.3) {
                                display.append(lcs.snowReport() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Snow data has been updated \'" +
"\'.\n");

                        if (lcs.getVisibility() < 2.0) {
                                display.append(lcs.visibilityReport() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Visibility data has been updated \'" +
"\'.\n");

                        if (lcs.isObstruction()) {
                                display.append(lcs.ProcessObject() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Obstruction data has been updated
\'" + "\'.\n");

                        if (lcs.getSnowRate() >= 0.2 || lcs.getRainRate() >= 0.3) {
                                display.append(lcs.detectSlippage() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Wheels' status has been updated \'"
+ "\'.\n");

                        display.append(lcs.gateStatus() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Gate status has been updated \'" +
"\'.\n");
```

```
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog(lcs.toString());

                        if (totalMilliseconds == lcs.getTime() * 39000) {
                                loop.cancel();
                        }
                }
        };

        /**
         * Schedules the task to occur over a span of 6min 30s with 10 second intervals
         */
        loop.scheduleAtFixedRate(task, lcs.getTime() * 1000, lcs.getTime() * 1000);

        /**
         * Default block of code with JFrame
         */
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(325, 100, 900, 600);
        contentPane = new JPanel();
        contentPane.setBackground(Color.GRAY);
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);

        /**
         * Codes for our button components (style, size, functionality, etc.)
         */
        JButton btnRain = new JButton("RAIN");
        btnRain.setForeground(Color.BLACK);
        btnRain.setBounds(443, 311, 118, 71);
        btnRain.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Operator."))
                                display.setText("");
                        display.append(lcs.rainReport() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "RAIN" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnVisibility = new JButton("VISIBILITY");
```

```
		btnVisibility.setBounds(599, 311, 118, 71);
		btnVisibility.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.visibilityReport() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "VISIBILITY" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnWind = new JButton("WIND");
		btnWind.setBounds(599, 409, 118, 71);
		btnWind.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.windReport() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "WIND" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnSnow = new JButton("SNOW");
		btnSnow.setForeground(Color.BLACK);
		btnSnow.setBounds(443, 409, 118, 71);
		btnSnow.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.snowReport() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "SNOW" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnObstr = new JButton("OBSTRUCTION");
		btnObstr.setForeground(Color.BLACK);
```

```
		btnObstr.setBackground(UIManager.getColor("Button.background"));
		btnObstr.setBounds(285, 311, 118, 71);
		btnObstr.setFont(new Font("Tahoma", Font.BOLD, 10));
		btnObstr.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.ProcessObject() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "OBSTRUCTION" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnSlip = new JButton("SLIPPAGE");
		btnSlip.setForeground(Color.BLACK);
		btnSlip.setBounds(285, 411, 118, 71);
		btnSlip.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.detectSlippage() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "SLIPPAGE" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnGate = new JButton("GATE");
		btnGate.setBounds(743, 313, 118, 71);
		btnGate.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Operator."))
					display.setText("");
				display.append(lcs.gateStatus() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "GATE" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});
```

```
		JButton btnLog = new JButton("LOG");
		btnLog.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				display.setText("");
				display.setText("Error: Only the admin can access the log.\n");
			}
		});
		btnLog.setBounds(743, 411, 118, 71);

		JButton btnClear = new JButton("CLEAR");
		btnClear.setForeground(Color.RED);
		btnClear.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				display.setText("");
			}
		});
		btnClear.setFont(new Font("Tahoma", Font.BOLD, 14));
		btnClear.setBounds(87, 311, 118, 71);

		JButton btnLogOff = new JButton("LOG OFF");
		btnLogOff.setForeground(Color.RED);
		btnLogOff.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				lcs.setIsLoggedIn(false);
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User \'" + "operator" + "\' logged off
successfully.\n");
				lcs.writeToLog(lcs.toString());
				loop.cancel();
				LoginGUI login = new LoginGUI();
				login.setVisible(true);
				dispose();
			}
		});
		btnLogOff.setFont(new Font("Tahoma", Font.BOLD, 14));
		btnLogOff.setBounds(87, 411, 118, 71);

		JButton btnExit = new JButton("X");
		btnExit.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				lcs.writeToLog("" + date + "-- User \'" + "operator" + "\' terminated
LCS session.\n");
				loop.cancel();
				dispose();
```

```
                }
            });
            btnExit.setForeground(Color.BLACK);
            btnExit.setBackground(Color.RED);
            btnExit.setFont(new Font("Tahoma", Font.BOLD, 12));
            btnExit.setBounds(839, 0, 47, 36);

            /**
             * Block of code that adds the components onto the GUI
             */
            contentPane.setLayout(null);
            contentPane.add(rpmPane);
            contentPane.add(lblRPM);
            contentPane.add(btnSlip);
            contentPane.add(btnObstr);
            contentPane.add(btnSnow);
            contentPane.add(btnRain);
            contentPane.add(btnVisibility);
            contentPane.add(btnWind);
            contentPane.add(btnGate);
            contentPane.add(speedPane);
            contentPane.add(scroll);
            contentPane.add(lblSpeed);
            contentPane.add(btnLog);
            contentPane.add(btnExit);
            contentPane.add(btnLogOff);
            contentPane.add(btnClear);
        }

}
```

AdminGUI.java

```
package htrPackage;

import java.awt.Color;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextPane;
import javax.swing.border.EmptyBorder;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.UIManager;

public class AdminGUI extends JFrame {

	/**
	 * Data fields
	 */
	private JPanel contentPane;
	LCS lcs;
	Date date = java.util.Calendar.getInstance().getTime();
	Timer loop;
	int totalMilliseconds;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					AdminGUI frame = new AdminGUI();
					frame.setTitle("LCS");
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public AdminGUI() {
		/**
```

```
 * Create new instances of LCS and Timer Set isLoggedIn() to true Set date
 * variable
 */
lcs = new LCS();
lcs.setIsLoggedIn(true);
loop = new Timer();
date = java.util.Calendar.getInstance().getTime();

/**
 * Sets up the main display area for the outputs and recommendations
 */
JScrollPane scroll = new JScrollPane();
scroll.setBounds(285, 42, 576, 245);
JTextArea display = new JTextArea();
display.setForeground(Color.GREEN);
scroll.setViewportView(display);
display.setEditable(false);
display.setWrapStyleWord(true);
display.setLineWrap(true);
display.setBackground(Color.BLACK);
display.setFont(new Font("Monospaced", Font.BOLD, 16));
display.setText("Welcome, Admin.");

/**
 * Sets up the label and display for the speed data
 */
JTextPane speedPane = new JTextPane();
speedPane.setBounds(57, 83, 152, 67);
speedPane.setEditable(false);
speedPane.setBackground(Color.BLACK);
speedPane.setForeground(Color.GREEN);
speedPane.setFont(new Font("Tahoma", Font.PLAIN, 50));
speedPane.setText(String.valueOf(lcs.getSpeed()));

JLabel lblSpeed = new JLabel("SPEED");
lblSpeed.setBounds(92, 39, 87, 34);
lblSpeed.setForeground(Color.YELLOW);
lblSpeed.setFont(new Font("Tahoma", Font.BOLD, 14));
lblSpeed.setHorizontalAlignment(JLabel.CENTER);

/**
 * Sets up the label and display for the RPM data
 */
JLabel lblRPM = new JLabel("RPM");
```

```
		lblRPM.setBounds(72, 160, 128, 34);
		lblRPM.setForeground(Color.YELLOW);
		lblRPM.setFont(new Font("Tahoma", Font.BOLD, 14));
		lblRPM.setHorizontalAlignment(JLabel.CENTER);

		JTextPane rpmPane = new JTextPane();
		rpmPane.setBounds(82, 193, 103, 61);
		rpmPane.setEditable(false);
		rpmPane.setBackground(Color.BLACK);
		rpmPane.setForeground(Color.GREEN);
		rpmPane.setFont(new Font("Tahoma", Font.PLAIN, 50));
		rpmPane.setText(String.valueOf(lcs.getRPM()));

		/**
		 * Create and define a TimerTask() to automate file reading, updating, the
		 * output printing
		 */
		TimerTask task = new TimerTask() {
			public void run() {
				totalMilliseconds += lcs.getTime() * 1000;
				lcs.updateValuesSensors();

				speedPane.setText(String.valueOf(lcs.getSpeed()));
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- Speed data has been updated \'" +
"\'.\n");

				rpmPane.setText(String.valueOf(lcs.getRPM()));
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- RPM data has been updated \'" +
"\'.\n");

				display.setText("");

				if (lcs.getRainRate() >= 0.3) {
					display.append(lcs.rainReport() + "\n");
				}
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- Rain data has been updated \'" +
"\'.\n");

				if (lcs.getWindSpeed() >= 50.0) {
					display.append(lcs.windReport() + "\n");
				}
```

```
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Wind data has been updated \'" +
"\'.\n");

                        if (lcs.getSnowRate() >= 0.3) {
                                display.append(lcs.snowReport() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Snow data has been updated \'" +
"\'.\n");

                        if (lcs.getVisibility() < 2.0) {
                                display.append(lcs.visibilityReport() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Visibility data has been updated \'" +
"\'.\n");

                        if (lcs.isObstruction()) {
                                display.append(lcs.ProcessObject() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Obstruction data has been updated
\'" + "\'.\n");

                        if (lcs.getSnowRate() >= 0.2 || lcs.getRainRate() >= 0.3) {
                                display.append(lcs.detectSlippage() + "\n");
                        }
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Wheels' status has been updated \'"
+ "\'.\n");

                        display.append(lcs.gateStatus() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- Gate status has been updated \'" +
"\'.\n");

                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog(lcs.toString());

                        if (totalMilliseconds == lcs.getTime() * 39000) {
                                loop.cancel();
                        }
                }
```

```
		};

		/**
		 * Schedules the task to occur over a span of 6min 30s with 10 second intervals
		 */
		loop.scheduleAtFixedRate(task, lcs.getTime() * 1000, lcs.getTime() * 1000);

		/**
		 * Default block of code with JFrame
		 */
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(325, 100, 900, 600);
		contentPane = new JPanel();
		contentPane.setBackground(Color.GRAY);
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);

		/**
		 * Codes for our button components (style, size, functionality, etc.)
		 */
		JButton btnRain = new JButton("RAIN");
		btnRain.setForeground(Color.BLACK);
		btnRain.setBounds(443, 311, 118, 71);
		btnRain.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Admin."))
					display.setText("");
				display.append(lcs.rainReport() + "\n");
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User entered the following command
\'" + "RAIN" + "\'.\n");
				lcs.writeToLog(lcs.toString());
			}
		});

		JButton btnVisibility = new JButton("VISIBILITY");
		btnVisibility.setBounds(599, 311, 118, 71);
		btnVisibility.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				if (display.getText().equals("Welcome, Admin."))
					display.setText("");
				display.append(lcs.visibilityReport() + "\n");
				date = java.util.Calendar.getInstance().getTime();
```

```
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "VISIBILITY" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnWind = new JButton("WIND");
        btnWind.setBounds(599, 409, 118, 71);
        btnWind.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Admin."))
                                display.setText("");
                        display.append(lcs.windReport() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "WIND" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnSnow = new JButton("SNOW");
        btnSnow.setForeground(Color.BLACK);
        btnSnow.setBounds(443, 409, 118, 71);
        btnSnow.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Admin."))
                                display.setText("");
                        display.append(lcs.snowReport() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "SNOW" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnObstr = new JButton("OBSTRUCTION");
        btnObstr.setForeground(Color.BLACK);
        btnObstr.setBackground(UIManager.getColor("Button.background"));
        btnObstr.setBounds(285, 311, 118, 71);
        btnObstr.setFont(new Font("Tahoma", Font.BOLD, 10));
        btnObstr.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Admin."))
                                display.setText("");
```

```
                        display.append(lcs.ProcessObject() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "OBSTRUCTION" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnSlip = new JButton("SLIPPAGE");
        btnSlip.setForeground(Color.BLACK);
        btnSlip.setBounds(285, 411, 118, 71);
        btnSlip.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Admin."))
                                display.setText("");
                        display.append(lcs.detectSlippage() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "SLIPPAGE" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnGate = new JButton("GATE");
        btnGate.setBounds(743, 313, 118, 71);
        btnGate.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        if (display.getText().equals("Welcome, Admin."))
                                display.setText("");
                        display.append(lcs.gateStatus() + "\n");
                        date = java.util.Calendar.getInstance().getTime();
                        lcs.writeToLog("" + date + "-- User entered the following command
\'" + "GATE" + "\'.\n");
                        lcs.writeToLog(lcs.toString());
                }
        });

        JButton btnLog = new JButton("LOG");
        btnLog.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        display.setText("");
                        display.setText(lcs.readFromLog());
                        lcs.writeToLog(lcs.toString());
                }
```

```
		});
		btnLog.setBounds(743, 411, 118, 71);

		JButton btnClear = new JButton("CLEAR");
		btnClear.setForeground(Color.RED);
		btnClear.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				display.setText("");
			}
		});
		btnClear.setFont(new Font("Tahoma", Font.BOLD, 14));
		btnClear.setBounds(87, 311, 118, 71);

		JButton btnLogOff = new JButton("LOG OFF");
		btnLogOff.setForeground(Color.RED);
		btnLogOff.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				lcs.setIsLoggedIn(false);
				date = java.util.Calendar.getInstance().getTime();
				lcs.writeToLog("" + date + "-- User \'" + "admin" + "\' logged off
successfully.\n");
				lcs.writeToLog(lcs.toString());
				loop.cancel();
				LoginGUI login = new LoginGUI();
				login.setVisible(true);
				dispose();
			}
		});
		btnLogOff.setFont(new Font("Tahoma", Font.BOLD, 14));
		btnLogOff.setBounds(87, 411, 118, 71);

		JButton btnExit = new JButton("X");
		btnExit.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				lcs.writeToLog("" + date + "-- User \'" + "admin" + "\' terminated
LCS session.\n");
				loop.cancel();
				dispose();
			}
		});
		btnExit.setForeground(Color.BLACK);
		btnExit.setBackground(Color.RED);
		btnExit.setFont(new Font("Tahoma", Font.BOLD, 12));
		btnExit.setBounds(839, 0, 47, 36);
```

```
            /**
             * Block of code that adds the components onto the GUI
             */
            contentPane.setLayout(null);
            contentPane.add(rpmPane);
            contentPane.add(lblRPM);
            contentPane.add(btnSlip);
            contentPane.add(btnObstr);
            contentPane.add(btnSnow);
            contentPane.add(btnRain);
            contentPane.add(btnVisibility);
            contentPane.add(btnWind);
            contentPane.add(btnGate);
            contentPane.add(speedPane);
            contentPane.add(scroll);
            contentPane.add(lblSpeed);
            contentPane.add(btnLog);
            contentPane.add(btnExit);
            contentPane.add(btnLogOff);
            contentPane.add(btnClear);
        }

}
```

Section 7: Testing

7.1: Non-functional Requirements Testing

Testing Requirement 1 ... All sensors shall be water-proof – Test case has passed.

Testing Requirement 2 ... All sensors shall last for 1000 hours under water – Test case has passed.

Testing Requirement 3 ... All sensors shall operate 99.99% of the time – Test case has passed.

Testing Requirement 4 ... IoT HTR shall be supported by a hardwired LoRaWan network – Test case has passed.

Testing Requirement 5 ... IoT Hardware shall be able to support 1000 sensors – Test case has passed.

Testing Requirement 6 ... IoT shall support 5TB of data everyday – Test case has passed.

Testing Requirement 7 ... IoT shall summarize information about the status of the train (in real time on a monitor inside the conductor's cabin – Test case has passed.

Testing Requirement 8 ... At most, IoT HTR shall process an event within 0.5 second of its occurrence – Test case has passed.

Testing Requirement 9 ... IoT HTR sensors shall detect events within 0.1 seconds – Test case has passed.

Testing Requirement 10 ... IoT HTR reliability shall be no less than 99.9% – Test case has passed.

Testing Requirement 11 ... Each sensor shall have a backup sensor – Test case has passed.

Testing Requirement 12 ... The train conductor will require a User ID and password to access the train's software – Test case has passed.

Testing Requirement 13 ... sensors will only connect to wifi when the train engine is off – Test case has passed.

Testing Requirement 14 ... sensors will only connect to wifi when the train is at a train station – Test case has passed.

Testing Requirement 15 ... Once a train arrives at its destination, it will upload the data it collected from its journey to the HTR servers – Test case has passed.

7.2: Functional Requirements Testing

Use Cases	**Commands & Scenarios**	**Recommendations**	**Results**
Use Case 4.1.1: System Administrator wants to access the LCS via IOT.	Correct Username and Password	Wed April 21 14:14:06 EEST 2021-- User 'admin' logged in.	Success
Use Case 4.1.1: Train Operator wants to access the LCS via IOT.	Incorrect Username and Password	Incorrect credentials. Wed April 21 14:14:06 EEST 2021-- Failed login attempt.	Success
Use Case 4.1.2: Train Operator wants to access the LCS via IOT.	Correct Username and Password	Wed April 21 14:14:06 EEST 2021-- User 'admin' logged in.	Success

Use Case 4.1.2: Train Operator wants to access the LCS via IOT.	Incorrect Username and Password	Incorrect credentials. Wed April 21 14:14:06 EEST 2021-- Failed login attempt.	Success
Use Case 4.1.3: The Train Operator wants to access information about an upcoming gate crossing.	Status (open gate 5.2 miles away)	The next gate, which is 5.2 miles away, is open.	Success
Use Case 4.1.3: The Train Operator wants to access information about an upcoming gate crossing.	Status (closed gate 1.3 miles away)	The next gate, which is 1.3 miles away, is closed. Recommendation: Stop the train immediately & wait for the gate to open.	Success
Use Case 4.1.4: Train Operator Spots an object obstructing the path.	(no input) (object 1 mile away)	An object is 1 mile away. Recommendation: Stop the train immediately & wait for object to clear path & sound horn.	Success
Use Case 4.1.4: Train Operator Spots an object obstructing the path.	obstruction (no object in the way)	There is no object obstructing the path.	Success

Use Case 4.1.4: Train Operator Spots an object obstructing the path.	obstruction (no object in the way) (Object 1.5 miles behind)	There is no object obstructing the path. Object is 1.5 miles behind train.	Success
Use Case 4.1.5: Train Operator wants to get information about the train's wheels.	rpm (rpm is 800)	Wheel rpm: 800	Success
Use Case 4.1.5: Train Operator wants to get information about the train's wheels.	rpm (rpm is 900)	Wheel rpm: 900	Success
Use Case 4.1.5: Train Operator wants to get information about the train's wheels.	(no input) (wheels are slipping) (speed 100 mph, rpm speed 50)	The wheels are slipping. Recommendation: slow down or halt the train.	Success
Use Case 4.1.5: Train Operator wants to get information about the train's wheels.	slippage (wheels are not slipping) (speed 60 mph, rpm speed 60)	The wheels are not slipping.	Success

Use Case 4.1.6: The Train Operator observes that the weather has become windy.	wind (no precipitation) (wind 70 mph) (visibility 2.5 miles)	The wind speed is 70 mph. Recommendation: Reduce the speed of the train.	Success
Use Case 4.1.6: The Train Operator observes that the weather has become windy.	wind (no precipitation) (wind 40 mph) (visibility 2.5 miles)	The wind speed is 40 mph.	Success
Use Case 4.1.7: The LCS shall inform the Train Operator of high wind speeds (only if the operator hasn't already noticed) and recommends a course of action.	(no input) (wind 70 mph)	The wind speed is 70 mph. Recommendation: Reduce the speed of the train.	Success

Use Case 4.1.8: The Train Operator observes that there is precipitation outside the train.	rain (.1 inch of rain per hr) (no wind) (no snow) (visibility 2.5 miles)	The rate of rainfall is 0.1 inches per hour.	Success
Use Case 4.1.8: The Train Operator observes that there is precipitation outside the train.	snow (.1 inch of snow per hr) (no wind) (no rain) (visibility 2.5 miles)	The rate of snowfall is 0.1 inches per hour.	Success
Use Case 4.1.8: The Train Operator observes that there is precipitation outside the train.	rain (1.2 inch of rain per hr) (no wind) (no snow) (visibility 2.5 miles)	The rate of rainfall is 1.2 inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success

Use Case 4.1.8: The Train Operator observes that there is precipitation outside the train.	snow (1.2 inch of snow per hr) (no wind) (no rain) (visibility 2.5 miles)	The rate of snowfall is 1.2 inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success
Use Case 4.1.9: The LCS shall inform the Train Operator of high rates of precipitation (only if the operator hasn't already noticed) and recommends the appropriate actions.	(no input) (1.2 inch of snowfall per hr)	The rate of snowfall is 1.2 inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success
Use Case 4.1.9: The LCS shall inform the Train Operator of high rates of precipitation (only if the operator hasn't already noticed) and recommends the appropriate actions.	(no input) (1.3 inch of rainfall per hr)	The rate of rainfall is 1.3 inches per hour. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success

Use Case 4.1.10: The Train Operator notices that the visibility is low.	visibility (visibility 0.9 miles) (no precipitation) (no wind)	The visibility is 0.9 miles. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success
The Train Operator notices that the visibility is low.	visibility (visibility 2.5 miles) (no wind) (no precipitation)	The visibility is at least 2 miles.	Success
Use Case 4.1.11: The LCS shall inform the Train Operator of low visibility (only if the operator hasn't already noticed) and recommends the appropriate actions.	(no input) (visibility 0.8 miles) (no precipitation) (no wind)	The visibility is 0.8 miles. Recommendation: Turn on the train headlights and reduce the speed of the train.	Success

Use Case 4.1.12: The LCS shall inform the Train Operator of an upcoming gate crossing and recommend them to honk the train horn.	(no input) (open gate 1.02 miles away)	Gate is about 1 mile away. Recommendation: Honk the horn for 15 seconds.	Success
Use Case 4.1.12: The LCS shall inform the Train Operator of an upcoming gate crossing and recommend them to honk the train horn.	(no input) (open gate 0.99 miles away)	Gate is about 1 mile away. Recommendation: Honk the horn for 15 seconds.	Success
Use Case 4.1.13: If the train is immediately about to cross through a gate, the LCS shall inform the Train Operator and recommend them to honk the train horn.	(no input) (open gate .1 miles away)	About to cross a gate. Recommendation: Honk the horn for 5 seconds.	Success

Use Case 4.1.13: If the train is immediately about to cross through a gate, the LCS shall inform the Train Operator and recommend them to honk the train horn.	(no input) (open gate .05 miles away)	About to cross a gate. Recommendation: Honk the horn for 5 seconds.	Success

Testing Requirement 16 ... Geographical Sensors (GPS) will also be able to track the position of the train within a margin of error no greater than 10 meters – Test case has passed.

Testing Requirement 17 ... Detect standing objects on the path of the train with

distance, time to collision, and recommend action to the operator for braking or increasing/decreasing the speed, or completely stopping – Test case has passed.

Testing Requirement 18 ... Detection of standing objects will be done using proximity sensors – Test case has passed.

Testing Requirement 19 ... Each proximity sensor will have a field of vision of 120 degrees angle – Test case has passed.

Testing Requirement 20 ... Each proximity sensor will be able to detect an object up to 2 miles away – Test case has passed.

Testing Requirement 21 ... Each proximity sensor will have the ability to detect objects with 98% accuracy – Test case has passed.

Testing Requirement 22 ... Apart from being placed in the front of the train, proximity sensors will be placed in other areas of the train to cover blind spots such as some parts of the side and the back – Test case has passed.

Testing Requirement 23 ... Immediately upon detection of an object, IOT will access data of the speed of the train and compare that with the distance of the object to the train, in order to give a precise plan of action to the operator – Test case has passed.

Testing Requirement 24 ... Once the IOT is done with the calculations, it shall signal the operator what the plan of action should be – Test case has passed.

Testing Requirement 25 ... Detect moving objects, their distance (ahead and/or behind), their speed, and recommend to the operator braking or changing speed – Test case has passed.

Testing Requirement 26 ... Motion sensors on the front and back side of the train shall detect any moving objects that are within 2 miles of the train (both in front of and behind)

Testing Requirement 27 ... The field of vision of these motion sensors shall be set to 120 degrees – Test case has passed.

Testing Requirement 28 ... The sensitivity of these motion sensors shall be set to detect objects that have a surface area greater than 1 square foot – Test case has passed.

Testing Requirement 29 ... Motion sensors shall be accurate at least 98% of the time. – Test case has passed.

Testing Requirement 30 ... If a moving object is detected by a motion sensor, then the conductor shall be notified within 0.1 seconds – Test case has passed.

Testing Requirement 31 ... The IOT shall give the conductor recommendations on whether to slow down, speed up, or completely stop the train – Test case has passed.

Testing Requirement 32 ... Detect if the gate crossing is open/closed, its distance from the train, and recommend a speed change (increase, decrease, or stop) to the operator – Test case has passed.

Testing Requirement 33 ... GPS shall be used to determine the proximity of gate crossings with 99.99% accuracy – Test case has passed.

Testing Requirement 34 ... If a gate is open, the conductor shall be informed about it and recommended to proceed normally – Test case has passed.

Testing Requirement 35 ... Proximity sensors on the front side of the train shall detect if a gate crossing is closed – Test case has passed.

Testing Requirement 36 ... In the event a gate crossing is closed, the software shall recommend the conductor slow down the train to a complete stop – Test case has passed.

Testing Requirement 37 ... Proximity sensors will detect when the gate opens – Test case has passed.

Testing Requirement 38 ... If a gate opens, the conductor shall be recommended to start moving – Test case has passed.

Testing Requirement 39 ... Detect wheel slippage using GPS speed data, compare it with the wheel RPM, and recommend the operator break or alter the speed – Test case has passed.

Testing Requirement 40 ... Tachometers, sensors which measure rotations per minute, shall be placed on the first four wheels at the front of the train – Test case has passed.

Testing Requirement 41 ... Tachometers shall measure the rpm of the wheels with 98% accuracy – Test case has passed.

Testing Requirement 42 ... An odometer in the front of the train shall measure the speed of the train to the nearest 0.1 mph – Test case has passed.

Testing Requirement 43 ... The speed calculated by the odometer and the rpm measured by the tachometer shall be compared. If the ratio of these two measurements recommended the train is slipping/skidding, the train conductor shall be notified and recommended to lower the train speed and start braking from a farther distance – Test case has passed.

Testing Requirement 44 ... Such comparisons shall be made every 5 seconds – Test case has passed.

Testing Requirement 45 ... Detect severe wind gusts and recommend the operator to alter speed – Test case has passed.

Testing Requirement 46 ... Anemometers will be able to measure wind speeds with 99.9% accuracy – Test case has passed.

Testing Requirement 47 ... They will be placed in certain parts of the train to ensure all around measurement of the winds (front, sides, rear) – Test case has passed.

Testing Requirement 48 ... Anemometers will automatically take new measurements every 20 seconds – Test case has passed.

Testing Requirement 49 ... If two anemometers measure 50 mph wind speeds, the sensors will signal to the operator to slow down the train – Test case has passed.

Testing Requirement 50 ... Detect heavy precipitation (rain or snow), track how strong it is and recommend the operator to alter speed and turn on train lights – Test case has passed.

Testing Requirement 51 ... Rain sensors installed at the front of the train will detect rainfall with 99.9% accuracy – Test case has passed.

Testing Requirement 52 ... Snow gauges on the sides of the train will be able to measure snowfall with 99.9% accuracy – Test case has passed.

Testing Requirement 53 ... Rain sensors and snow gauges will continuously update their data (by taking new measurements) every minute – Test case has passed.

Testing Requirement 54 ... If two consecutive 0.3 inch/hour measurements have been measured (this rule is for both rain and snow), the sensor will signal the operator to slow down and to turn on the train' lights – Test case has passed.

Testing Requirement 55 ... Detect low visibility and recommend the operator to alter speed and turn on train lights – Test case has passed.

Testing Requirement 56 ... Visibility sensors will be able to measure the distance an average human eye can see (about 2 miles) under given weather conditions (ie. fog) with 99.9% accuracy – Test case has passed.

Testing Requirement 57 ... They will be installed in the front of the train – Test case has passed.

Testing Requirement 58 ... If measured distance is below 4000 meters (~13123 ft), signal the operator to lower speed and turn on train lights – Test case has passed.

Testing Requirement 59 ... The train's horn shall be heard from at least 1.5 miles away. – Test case has passed.

Testing Requirement 60 … In addition to detecting the status of an upcoming gate, now, when there is an upcoming gate one mile away from the train, LCS shall recommend the operator to honk the horn for 15 seconds. And when the train is about to pass through the gate, LCS shall recommend the operator to honk the horn for 5 seconds. – Test case has passed.

Testing Requirement 61 … When the train is 1 mile away from an open gate, LCS shall recommend the operator to honk the horn for 15 seconds. – Test case has passed.

Testing Requirement 62 … As soon as the train is about to pass through an open gate, LCS shall recommend the operator to honk the horn for 5 seconds. – Test case has passed.