Hug the Rails: Internet of Things

Version 5

Group 15

CS 347-D

Jason Ruan, Tomasz Borowiak, Maddie Johnson

Table of Contents

1) SECTION ONE

1.1 Introduction

1.1.1 Current Model

Hug the Rails (HTR) trains currently rely on a cellular network to download real-time weather reports, rail conditions, and more from a remote Central Operations Center. The downloaded information is then stored and used by the Locomotive Control System (LCS) in two main ways. First, LCS warns its train operators of unforeseen and hazardous conditions on their trip, such as a build up of ice on the rails or a train accident nearby. Secondly, LCS uses the downloaded information to adjust the train's movement for a safer and smoother trip, such as reducing the train's speed to compensate for the icy rails.

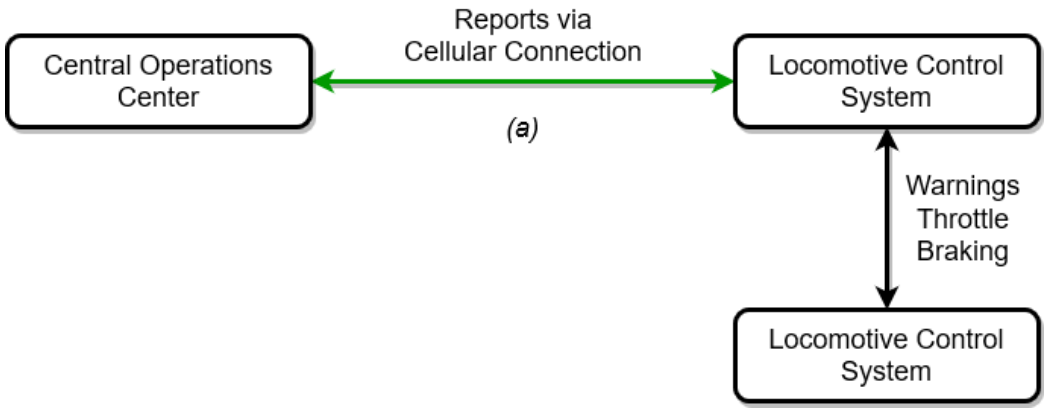


***Figure 1**: Current framework between the Central Operations Center and the Locomotive Control System via cellular connectivity.*

1.1.2 Problem

Unfortunately, LCS' efficiency in ensuring the safety of its train operators and passengers is entirely dependent on the cellular connectivity between the Central Operations Center and LCS. In the absence of cellular connectivity (severing arrow *(a)* in *Figure 1*), LCS cannot download critical information from the Central Operations Center. This leaves the train operators and their judgement to manually operate LCS. Failure to monitor sufficient speeds and critical warnings by the train operators may prove catastrophic in a preventable derailment or crash. As

a result, the safety and integrity of HTR train operators and its passengers will be put at risk in an otherwise safe system.

1.1.3 Purpose

In an effort to mitigate safety concerns caused by the absence of cellular connectivity, we propose the Internet of Things (IoT) to promote a safer trip and more cost efficient environment for HTR train operators and its passengers. IoT will be an independent software engine that assists HTR train operators in managing and operating LCS without the presence of a cellular network. Through communicating with local sensors installed on the train, IoT will be making speed recommendations to its train operators similar to how LCS gets its recommendations based on downloaded information from the Central Operations Center. More technical details on how IoT works will be outlined in Section Two.

1.1.4 Importance and Value to Operation

IoT's role as a local and independent system safeguards against an otherwise connection-reliant system. With IoT, HTR train operators will be able to confidently operate LCS knowing that IoT's recommendations are based on accurate and real-time data collected from around the train. As a result, HTR trips will become increasingly safe while mitigating the burden caused by high-decision making stress.

Furthermore, by decreasing reliance on the Central Operations Center, costs related to maintaining network towers and establishing secure connections over the web are decreased in favor of mechanical maintenance of the train's sensors and periodic updates to the IoT software. In fact, periodic maintenance of the sensors can simultaneously streamline the checkup and integrity of HTR's trains.

1.2 Planning

1.2.1 Our Team

We are a talented team of individuals who are proficient in several programming languages such as C++ and Java. The three of us will provide a diverse set of skills such as coding, writing, problem analysis, and problem planning. Additionally, some minor skills such as being task and detail-oriented will assist us in the development of IoT.

1.2.2 Stakeholders

Our team will work closely with the executives of HTR, its engineering team, Professor Peyrovian, and the CAs of CS-347 to ensure IoT's success as a product. To establish transparency and a common frame of thinking with our stakeholders, we will be holding weekly briefings to discuss what has been accomplished in the past week of IoT's development. As a result of these periodic updates, not only will we gain a deeper understanding of what is required of IoT, but our stakeholders will also be extensively involved in the development of their own product.

1.2.3 Timeline and Deadlines

2/9/21:
Establish the team. Discuss each member's strengths and weaknesses to effectively delegate roles and responsibilities, as outlined in section 1.2.4.

2/16/21:
Conference with the stakeholders to establish the requirements of IoT. Create a preliminary plan that outlines IoT's general usage and impact on HTR.

2/23/21:
Define the expected input and output of IoT. Find current technologies that will supply IoT with the appropriate information to prevent rail hazards.

3/2/21:
Start analyzing potential implementations and use cases of said sensors in order for IoT to effectively communicate and analyze local data.

3/9/21:
With the sensors implemented, create an interface that links the sensor's output to IoT's input.

3/16/21:
Create the rules and regulations that dictate how IoT responds to specific hazards and circumstances. The output of IoT should be measurable.

3/23/21:
Decide on how IoT generates its reduced speed recommendations and warnings. The recommendations should be simple but also informative.

3/30/21:
Select an architecture type that will support IoT's implementation. The architecture should focus on IoT's dynamic nature and ease of code maintenance.

4/6/21:
Use the selected architecture to code and implement IoT. Make sure that the code follows the defined requirements and use cases.

4/13/21:
Define and explore test cases to replicate IoT in a realistic environment. Edge cases must also be considered. Verify that IoT's output is regular and logical.

4/20/21:
Meet with stakeholders to ensure IoT meets the specifications and is in a satisfactory state. Make changes as needed in response to the stakeholder's feedback.

4/27/21:
Deploy IoT for public use on HTR.

1.2.4 Roles and Responsibilities

In order to maintain organization and ensure that we are on task while developing IoT, we will be delegating and alternating the following roles amongst ourselves every two weeks:

Communications Liaison: Facilitates communication between the team and the professor/CA. When the team encounters any issues, the liaison will communicate with them to hopefully work out an acceptable solution.

Project Manager: Maintains the project to ensure that tasks are completed according to the agreed upon timeline. Also writes a weekly progress report including what was accomplished and what the next steps are.

Software Developer: Leads all coding, problem solving, and analytical work involved in the project.

1.2.5 Communication and Organization

The team will communicate on a weekly basis and report any progress made. The team will also meet if there is a problem that needs to be addressed. We will operate according to the timeline and prioritize more difficult tasks first.

We will be utilizing GitLab to ensure version control and consistently review the code before merging to the main repository. We will also utilize branching and pull requests to maximize version control.

1.2.6 Process Model

We will be utilizing a hybrid of the Unified Process Model and the Agile Method while developing IoT. Due to lenient requirements defined by the stakeholders and a considerable amount of freedom while developing IoT, the team wanted to emphasize consistent and comprehensive customer involvement throughout the project. By choosing the Unified Process Model, the team will be able to establish clear communication with the stakeholders and implement continuous constructive feedback. Furthermore, because the requirements of this project may evolve before completion, it is imperative that there is some flexibility to account for future accommodations, changes, and maintenance of the project.

2) SECTION TWO

2.1 Overview of IoT

2.1.1 Purpose

To rectify the gap in accountability due to the absence of cellular connectivity, we propose the Internet of Things (IoT) to independently monitor and adjust train movement without jeopardizing the safety of train operators and their passengers. IoT is an advanced algorithm that will communicate with pre-installed sensors on the train. Using collected data from the sensors, IoT will independently form recommendations on the train's speed and provide them to the train operators for approval. This way, a rogue IoT will not be able to override the train's controls without manual approval and agreement between the train's operators.

2.1.2 Expected Output

The main output of IoT follows:

- If the RPM of the wheels exceed the speed that the train is currently traveling at, IoT will generate a wheel slip warning and a recommended train speed reduction to the train operators to reduce the effects of wheel slip.
- If there are obstructions on the tracks (moving or stationary), IoT will caution the train operators to slow down or stop the train accordingly to prevent a potential derailment. IoT will also provide the distance until the train reaches the object.
- If there is a Railroad Crossing ahead and the crossing gates have not fully closed, IoT will warn the train operators to stop the train. Additionally, IoT will provide the distance until the train reaches the Railroad Crossing while also providing a train horn recommendation.

2.2 Interfacing with Other Technologies

2.2.1 Sensors

In order to maintain the functionality of IoT, we need access to accurate and continuous data from around the train. Local data such as the train's location, the RPM of the wheels, and a scan of the tracks can

be readily collected through pre-installed sensors on the train for later processing by IoT. Thus, all HTR trains must be outfitted with 5 different types of sensors:

- 2 Lidar sensors mounted at the front of the train
- A proximity sensor mounted at the front of the train
- An optical sensor mounted at the front of the train
- A GPS unit mounted at the top of the train
- 4 RPM sensors mounted on each of the 4 wheels

Additionally, in order to maintain uniformity of the different sensor outputs, a Time Sensitive Networking Router (TSNR) will be used to combine sensor data into a single packet. TSNR will send this packet to IoT for processing outlined in Section 3.2.

2.2.2 Lidar Sensor

The 2 Lidar sensors will address issues related to track obstructions and Railroad Crossings. A Lidar sensor will rapidly shoot out light waves at the front of the train. When the light waves hit an object, they will be reflected and redirected back to the Lidar sensor. The time in between transmission and reception can then be used to measure the distance of all nearby objects in front of the train. Using this distance scan, IoT will accomplish two goals:

- The first Lidar sensor will be able to determine the distance from the nearest object on the tracks to the train's current position. Then, based on how far or close the object is to the train, IoT will be able to warn the train operators to slow down accordingly.
- The second Lidar sensor will be able to determine the distance from the nearest Railroad Crossing to the train's current position. Then, based on how far the Railroad Crossing is to the train, IoT will be able to warn the train operators to slow down accordingly.

2.2.3 Proximity Sensor

Similar to the Lidar sensor, the proximity sensor will address issues related to track obstructions. As the train moves, the proximity sensor will produce a large electromagnetic field in front of the train. When an object

moves through this field, the disruption of the field can then be detected by the proximity sensor to determine whether the object is stationary or moving. Therefore the proximity sensor and first Lidar sensor will provide important information to IoT in detecting stationary and moving objects.

2.2.4 Optical Sensor

When the second Lidar sensor detects the presence of a nearby Railroad Crossing, the optical sensor will play a critical role in determining how IoT proceeds with Railroad Crossing detection. The optical sensor will take periodic snapshots of the crossing gates and determine whether the crossing gates are in the fully opened or closed position. IoT will then generate a brake reccomendation if the crossing gates are opened, or a reduced speed recommendation if the crossing gates are closed.

2.2.5 GPS Unit

The GPS unit will periodically record the coordinates of the train's position as it moves. The GPS unit will then store the current position of the train as well as the previous position. Thus by using the change in position and change in time, the GPS unit will be able to accurately determine the speed of the train in miles per hour.

2.2.6 RPM Sensor

The 4 RPM sensors will address issues related to the weather. As the train moves, 4 RPM sensors will periodically record how many full rotations a wheel can make in a minute. Having 4 sensors ensures that IoT has an accurate average wheel RPM reading. Then, by comparing the wheel's average RPM to its average diameter, the average wheel speed can be calculated. By comparing the average wheel speed to the train's current speed, IoT can determine if there is wheel slippage on the rails.

2.2.7 Proposed Architecture of IoT

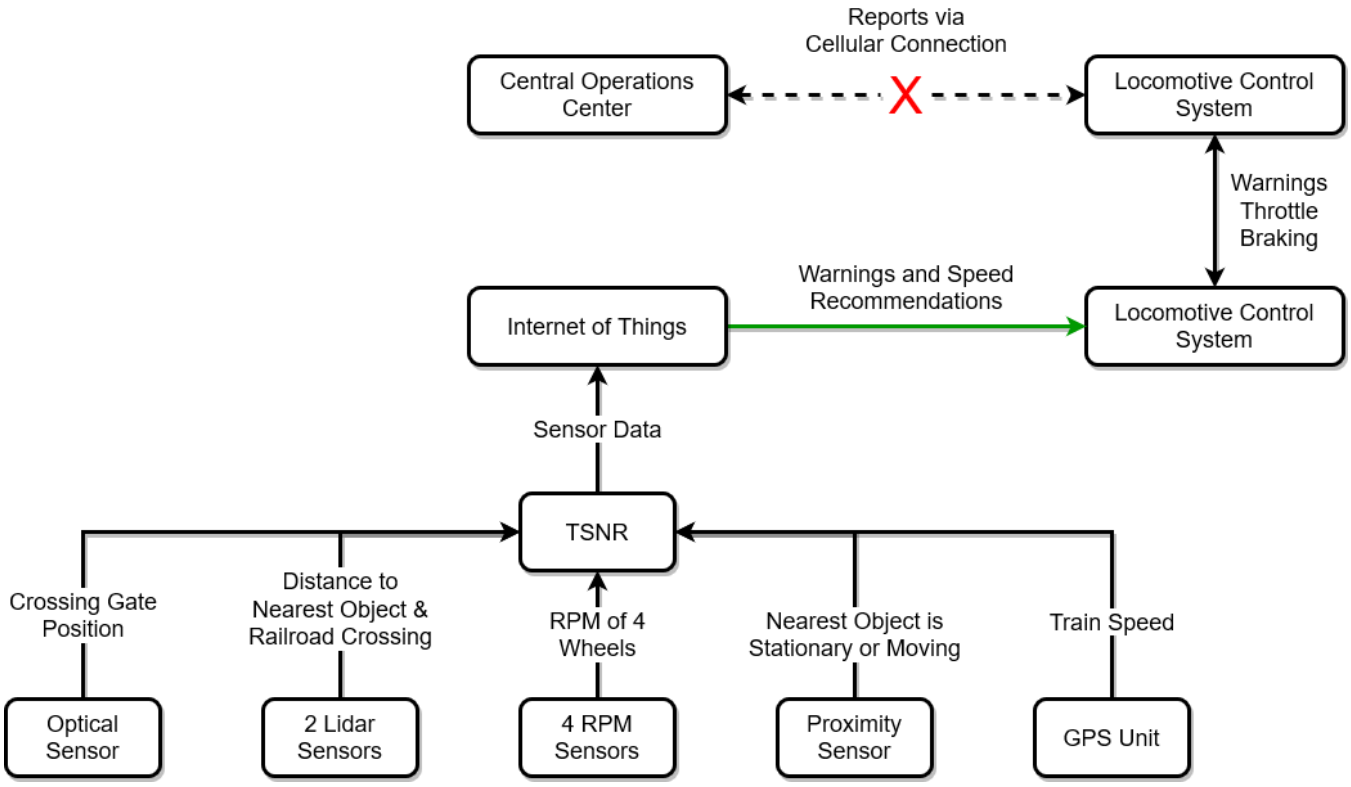


Figure 2: *Proposed framework between the Internet of Things and the Locomotive Control System in the absence of cellular connectivity.*

3) SECTION THREE

3.1 Non-Functional Requirements

3.1.1 Security

R1: All accessible software interfaces and sensor modules connected to IoT must be locked behind a username and password system.

R2: The HTR Network shall be secured by the LoRaWan protocol.

R3: Usernames shall only contain alphanumeric characters and must have a length from 3 to 16.

R4: Passwords shall only contain alphanumeric characters and must have a length from 5 to 25.

R5: IoT's login system must have 2 levels of clearance.

R6: Level 2 access shall be granted to HTR train operators who need IoT to drive and control LCS.

R7: Level 1 access shall be restricted to HTR admins or senior HTR engineers who need direct access to IoT's software and sensors.

R8: Level 1 users must have access to all modules accessible by level 2 users.

3.1.2 Terminal

R9: IoT's terminal screen must turn on when the train starts up.

R10: IoT's terminal shall consist of 3 panels: a console panel, an icon panel, and a prompt panel.

R11: The console panel shall display and handle messaging, errors, and login info.

R12: The console panel must handle color formatted text and display timestamps when necessary.

R13: The icon panel shall display any real-time distance warnings, reduced speed recommendations, and brake recommendations as specified in section 3.2.

R14: IoT shall display general warnings in a yellow color on the icon panel unless specified in another color.

R15: IoT shall display reduced speed recommendations in an orange color on the icon panel.

R16: IoT shall only display the lowest reduced speed recommendation in the icon panel.

R17: IoT shall not display a reduced speed recommendation when the train's current speed is below the reduced speed recommendation.

R18: IoT shall display brake recommendations in a red color on the icon panel.

R19: IoT shall display the icons in a white color when there are no warnings or recommendations.

R20: The prompt panel shall process command line inputs entered by level 1 and level 2 users.

R21: When IoT boots up, it must first prompt the user to enter their username and password into the terminal.

R22: When successfully logged on, IoT must identify the clearance level of the user.

R23: IoT must display a permission error message onto the terminal console if the user does not have the correct clearance level.

R24: IoT must display an error message onto the terminal console if the prompted request does not exist.

R25: Level 1 and level 2 users shall be able to log out of IoT from the terminal.

R26: IoT must prompt the user to enter their username and password into the terminal after a successful logout.

R27: Level 1 users shall be able to download a log file of all sensor outputs, IoT's outputs, generated recommendations, and any necessary calculations through a specified directory.

R28: IoT and the terminal shall remain on until the train is turned off.

3.1.3 Performance

R29: When IoT receives input data from a sensor, there must be at most 1 second of computation time to generate a recommendation for the train operator.

R30: When multiple sensors are running at the same, IoT must be able to handle at least 500 sensor requests per second.

R31: If IoT cannot handle requirements **R29** and **R30**, then IoT must increment its underperformance count.

3.1.4 Reliability

R32: IoT's performance to underperformance ratio must be above 99.5% for it to be considered reliable.

R33: IoT must only have 1 underperformance for every 1000 sensor inputs it processes.

R34: When IoT encounters an underperformance, the specified problem shall be saved in the log file.

3.2 Functional Requirements

3.2.1 Time Sensitive Networking Router

R35: At every second, the GPS unit shall give TSNR the current speed of the train in mph as a double.

R36: At every second, the first Lidar sensor shall give TSNR a double of the distance from the train to the closest object in feet.

R37: At every second, the proximity sensor shall tell TSNR whether the closest object is stationary, moving towards the train, or moving away from the train.

R38: At every second, the second Lidar sensor shall give TSNR a double of the distance from the train to the closest Railroad Crossing in feet.

R39: At every second, the optical sensor shall give TSNR a boolean if the Railroad Crossing gates are opened or closed.

R40: At every second, 4 RPM sensors shall give TSNR their respective wheel RPMs as a double.

R41: TSNR must combine the collected data from **R35** through **R40** into a single packet.

R42: At every second, TSNR must send the combined packet to IoT.

3.2.2 Moving Object Detection

R43: IoT must retrieve the distance and direction of the nearest moving object from TSNR's packet.

R44: IoT shall do nothing when the distance is NULL, as it signifies no moving object nearby.

R45: IoT shall also do nothing when the object is moving away from the train.

R46: IoT shall execute **R47** through **R52** when the object is moving towards the train.

R47: IoT must display an object warning icon with the object's distance from the train in feet when the moving object is within 3 miles of the train.

R48: IoT shall display the object warning icon in yellow when the moving object is within 2 to 3 miles of the train.

R49: IoT shall display the object warning icon in orange when the moving object is within 1 to 2 miles of the train.

R50: IoT shall display the object warning icon in red when the moving object is within 1 mile of the train.

R51: IoT must display an orange speed reduction icon with a recommended train speed of 30mph when the moving object is within 1 to 2 miles of the train.

R52: IoT must display a red emergency brake icon when the moving object is within 1 mile of the train.

3.2.3 Stationary Object Detection

R53: IoT must retrieve the distance and direction of the nearest moving object from TSNR's packet.

R54: IoT shall do nothing when the distance is NULL, as it signifies no stationary object nearby.

R55: IoT shall execute **R56** through **R61** when the object is stationary.

R56: IoT must display an object warning icon with the object's distance from the train in feet when the stationary object is within 3 miles of the train.

R57: IoT shall display the object warning icon in yellow when the stationary object is within 2 to 3 miles of the train.

R58: IoT shall display the object warning icon in orange when the stationary object is within 1 to 2 miles of the train.

R59: IoT shall display the object warning icon in red when the stationary object is within 1 mile of the train.

R60: IoT must display an orange speed reduction icon with a recommended train speed of 40mph when the stationary object is within 1 to 2 miles of the train.

R61: IoT must display a red emergency brake icon when the stationary object is within 1 mile of the train.

3.2.4 Railroad Crossing Detection

R62: IoT must retrieve the distance and crossing gate position of the nearest Railroad Crossing from TSNR's packet.

R63: IoT shall do nothing when the distance is NULL, as it signifies no Railroad Crossing nearby.

R64: IoT shall display a yellow horn warning for 15 seconds when the Railroad Crossing is within 3 miles of the train.

R65: IoT must also display a Railroad Crossing warning icon with the crossing's distance from the train in feet when the Railroad Crossing is within 3 miles of the train.

R66: IoT shall display the Railroad Crossing warning icon in yellow when the Railroad Crossing is within 2 to 3 miles of the train.

R67: IoT shall display the Railroad Crossing warning icon in orange when the crossing gates are in the closed position and are within 2 miles of the train.

R68: IoT shall display the Railroad Crossing warning icon in red when the crossing gates are in the opened position and are within 2 miles of the train.

R69: IoT must display an orange speed reduction icon with a recommended train speed of 40mph when the crossing gates are in the closed position and are within 2 miles of the train.

R70: IoT must display a red emergency brake icon when the crossing gates are in the opened position and are within 2 miles of the train.

R71: IoT shall display a yellow horn warning for 5 seconds when the Railroad Crossing is within 1 mile of the train.

3.2.5 Wheel Slip Detection

R72: IoT must retrieve the 4 wheel RPMs and current train speed from TSNR's packet.

R73: IoT must calculate the average wheel RPM using the 4 wheel RPMs.

R74: IoT shall then calculate the average wheel speed using the diameter of the wheel in feet and the average wheel RPM.

R75: IoT must display a yellow wheel slip warning icon when there is more than a 5mph difference between the average wheel speed and the train's current speed.

R76: IoT must display an orange speed reduction icon with a recommended train speed of 40mph when there is more than a 5mph difference between the average wheel speed and the train's current speed.

3.3 Hardware Requirements

3.3.1 Operating System

R77: IoT's operating system must be Linux.

R78: IoT's terminal must run as a Linux-based command line terminal.

3.3.2 Hardware Architecture

R79: IoT must handle at least 1000 sensors.

R80: IoT must support 5 terabytes of data storage everyday.

---

4) SECTION FOUR

4.1 Use Cases

4.1.1 Use Case 1: Activation of IoT

Primary Actor: Level 2 Train Operator or Level 1 Engineer.

Secondary Actors: IoT, Train engine.

Goal: To boot up and initialize IoT.

Preconditions: The train is off.

Trigger: The train is turned on.

Scenerio:

1. The user turns on the train by igniting the engine.
2. The train's engine supplies continuous power to IoT.
3. IoT boots up and turns on the terminal screen.
4. IoT verifies that all sensors are operational.
5. IoT creates a new log file with the current time and date.
6. IoT prompts the user for their username and password.
7. IoT remains active until power from the train's engine is lost.

Exceptions:

1. IoT will display an error message onto the terminal's console and shutdown after 30 seconds if a sensor fails to work.
2. IoT will display an error message if the log file cannot be created.

4.1.2 Use Case 2: Access to the Main Terminal

Primary Actor: Level 2 Train Operator or Level 1 Engineer.

Secondary Actors: IoT, Configuration file.

Goal: To access IoT's output and input requests via the main terminal.

Preconditions: IoT prompted the user for their login information. The user has a predefined username and password. The log file exists.

Trigger: The user enters their username and password onto the terminal prompt.

Scenerio:

1. IoT verifies the username and password through HTR's employee login database.
2. IoT gets the access level of the user if the username and password are valid.
3. IoT then greets the user by displaying a successful login message onto the terminal console.
4. IoT waits to recieve any generated recommendations/warnings or user input request on the terminal.
5. IoT displays a generated warning in a yellow color onto the terminal's icon panel.
6. IoT displays a generated reduced speed recommendation in an orange color onto the terminal's icon panel.
7. IoT displays a generated brake recommendation in a red color onto the terminal's icon panel.
8. IoT saves the current username, clearance level, and time of login to the log file on every user login.

Exceptions:

IoT will prompt the user for their username and password if the login information is invalid.

4.1.3 Use Case 3: Moving Object Detection

Primary Actor: TSNR.

Secondary Actors: IoT.

Goal: To generate a recommendation if an object is moving towards the train.

Preconditions: TSNR generated a packet that includes the object's distance from the first Lidar sensor and the direction of the object from the proximity sensor. The log file exists.

Trigger: TSNR sends the packet to IoT.

Scenerio:

1. IoT retrieves the distance between the nearest object and the train from TSNR's packet.
2. IoT ends the Use Case if the distance is NULL or if the distance is out of the train's range specified in **R47**.
3. IoT retrieves the direction of the nearest object in front of the train from TSNR's packet.
4. IoT ends the Use Case if the object is not moving towards the train.
5. IoT displays an object warning icon with the object's distance from the train in feet when the moving object is within the train's range specified in **R47**.
6. IoT also displays the object warning icon in the color:
   a. Yellow when the moving object is within the train's range specified in **R48**.
   b. Orange when the moving object is within the train's range specified in **R49**.
   c. Red when the moving object is within the train's range specified in **R50**.
7. IoT displays an orange speed reduction icon with the recommended train speed specified in **R51** when the moving object is within the train's range specified in **R51**.
8. IoT displays a red emergency brake icon when the moving object is within the train's range specified in **R52**.
9. IoT saves all calculations and recommendations/warnings to the log file with the current time and date.

4.1.4 Use Case 4: Stationary Object Detection

Primary Actor: TSNR.

Secondary Actors: IoT.

Goal: To generate a recommendation if an object is stationary in front the train.

Preconditions: TSNR generated a packet that includes the object's distance from the first Lidar sensor and the direction of the object from the proximity sensor. The log file exists.

Trigger: TSNR sends the packet to IoT.

Scenerio:

1. IoT retrieves the distance between the nearest object and the train from TSNR's packet.
2. IoT ends the Use Case if the distance is NULL or if the distance is out of the train's range specified in **R56**.
3. IoT retrieves the direction of the nearest object in front of the train from TSNR's packet.
4. IoT ends the Use Case if the object is not stationary.
5. IoT displays an object warning icon with the object's distance from the train in feet when the stationary object is within the train's range specified in **R56**.
6. IoT also displays the object warning icon in the color:
    a. Yellow when the stationary object is within the train's range specified in **R57**.
    b. Orange when the stationary object is within the train's range specified in **R58**.
    c. Red when the stationary object is within the train's range specified in **R59**.
7. IoT displays an orange speed reduction icon with the recommended train speed specified in **R60** when the stationary object is within the train's range specified in **R60**.
8. IoT displays a red emergency brake icon when the stationary object is within the train's range specified in **R60**.
9. IoT saves all calculations and recommendations/warnings to the log file with the current time and date.

4.1.5 Use Case 5: Railroad Crossing Detection

Primary Actor: TSNR.

Secondary Actors: IoT.

Goal: To generate a recommendation if a Railroad Crossing gate is in the opened position.

Preconditions: TSNR generated a packet that includes the Railroad Crossing's distance from the second Lidar sensor and the crossing gate position from the optical sensor. The log file exists.

Trigger: TSNR sends the packet IoT.

Scenerio:

1. IoT retrieves the distance between the nearest Railroad Crossing and the train from TSNR's packet.
2. IoT also retrieves the crossing gate position from TSNR's packet.
3. IoT ends the Use Case if the distance is NULL or if the distance is out of the train's range specified in **R64**.
4. IoT displays a yellow horn warning for the time specified in **R64** when the Railroad Crossing is within the train's range specified in **R64**.
5. IoT displays a Railroad Crossing warning icon with the crossing's distance from the train in feet when the Railroad Crossing is within the train's range specified in **R65**.
6. IoT also displays the Railroad Crossing warning icon in the color:
   a. Yellow when the Railroad Crossing is within the train's range specified in **R66**.
   b. Orange when the Railroad Crossing is within the train's range specified in **R67** and the crossing gates are in the closed position.
   c. Red when the Railroad Crossing is within the train's range specified in **R68** and the crossing gates are in the opened position.
7. IoT displays an orange speed reduction icon with the recommended train speed specified in **R69** when the Railroad Crossing is within the train's range specified in **R69** and the crossing gates are in the closed position.

8. IoT displays a red emergency brake when the Railroad Crossing is within the train's range specified in **R70** and the crossing gates are in the open position.
9. IoT displays a yellow horn warning for the time specified in **R71** when the Railroad Crossing is within the train's range specified in **R71**.
10. IoT saves all calculations and recommendations/warnings to the log file with the current time and date.

4.1.6 Use Case 6: Wheel Slip Detection

Primary Actor: TSNR.

Secondary Actors: IoT.

Goal: To generate a recommendation if there is a disagreement between the wheel's average speed and the current speed of the train.

Preconditions: TSNR generated a packet that includes the 4 wheel RPMs from the 4 RPM sensors and the current train speed from the GPS unit. The log file exists.

Trigger: TSNR sends the packet to IoT.

Scenerio:

1. IoT retrieves the 4 wheel RPMs from TSNR's packet.
2. IoT calculates the average wheel RPM using the 4 wheel RPMs in the previous step.
3. IoT obtains the pre-defined average wheel diameter in feet.
4. IoT calculates the average wheel speed using the formula: average wheel RPM * pi * average wheel diameter in feet * (1mi / 5280ft) * (60min / 1hr).
5. IoT retrieves the current train speed from TSNR's packet.
6. IoT computes the difference between the average wheel speed to the train's current speed.
7. IoT displays a yellow wheel slip warning icon when the speed difference is greater than or equal to the difference specified in **R75**.

8. IoT displays an orange speed reduction icon with the recommended train speed specified in **R76** when the speed difference is greater than or equal to the difference specified in **R76**.
9. IoT saves all calculations and recommendations/warnings to the log file with the current time and date.

4.1.7 Use Case 7: Access to Log Files

Primary Actor: Level 1 Engineer.

Secondary Actors: IoT.

Goal: To download IoT's log files from the terminal.

Preconditions: The log file exists.

Trigger: The engineer enters a request to download IoT's log files on the terminal.

Scenerio:

1. IoT reads the input request from the terminal prompt.
2. IoT verifies that the input request matches "logger".
3. IoT verifies that the engineer has level 1 clearance.
4. IoT prints the log directory onto the terminal's console.
5. The engineer enters the specified log directory.
6. IoT logs all user requests with the current time and date.

Exceptions:

IoT will ignore the log file request and display a permission message onto the terminal if the user does not have level 1 clearance.

4.1.8 Use Case 8: Deactivation of IoT

Primary Actor: Level 2 Train Operator or Level 1 Engineer.

Secondary Actors: IoT, Train engine.

Goal: To shutdown IoT.

Preconditions: IoT is active. The train is on.

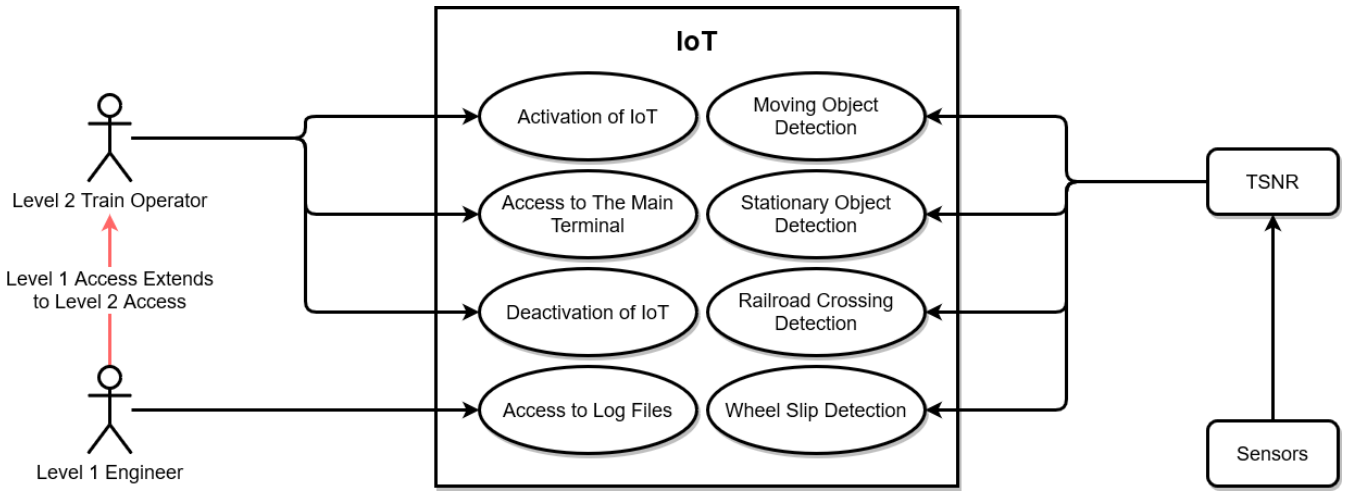
Trigger: The train turns off.

Scenerio:

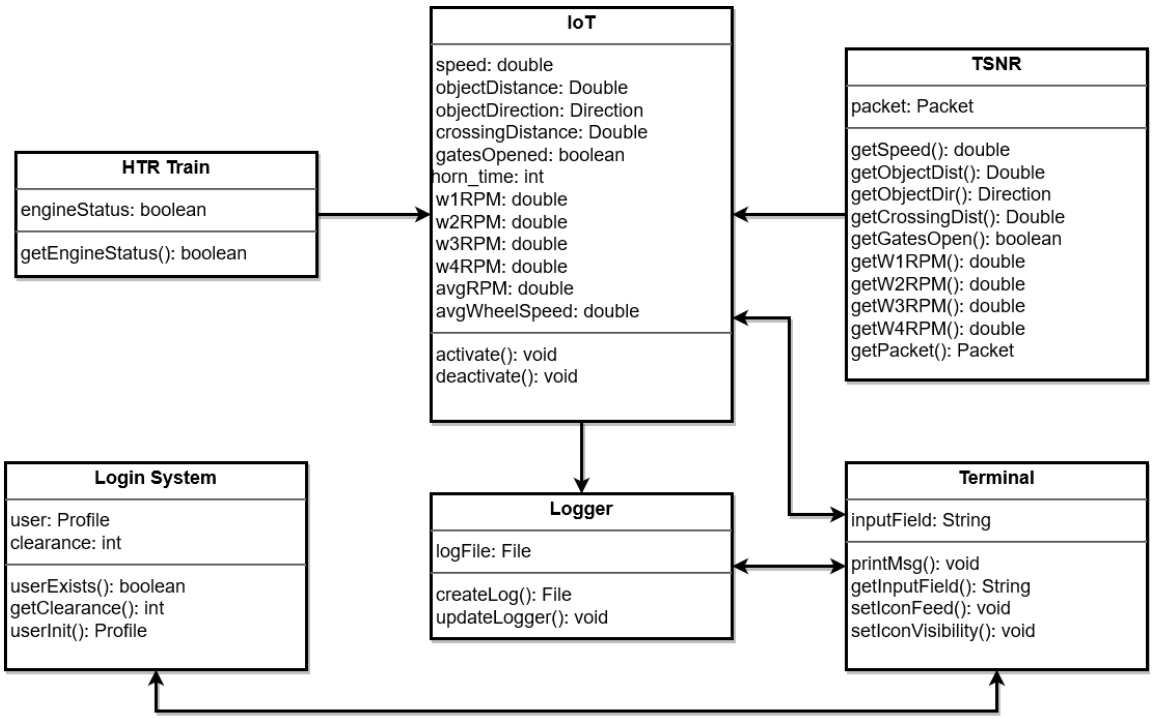
1. The user turns off the train.

2. The engine sends a deactivation request to IoT.
3. IoT saves and closes the log file.
4. IoT logs off the current user.
5. IoT turns off the terminal.
6. IoT shuts down.

4.2 Use Case Diagram



4.3 Class-Based Modeling



4.4 CRC Modeling Cards

4.4.1 HTR Train

Class Name: Train	
Description: *Access to hardware on the train.*	
Responsibility:	**Collaborators:**
getEngineStatus(): Determines whether the train's engine is on or off with true and false respectively. If the train engine is on, then power can be supplied to activate IoT.	IoT

4.4.2 IoT

Class Name: IoT	
Description: *Algorithm that uses sensor data to generate recommendations for the train operators.*	
Responsibility:	**Collaborators:**
activate() & deactivate(): Boots up and shuts down IoT respectively. IoT will only activate if the train engines are on. Additionally, IoT will deactivate when the engine turns off. While IoT is activated, it will repeatedly get packets from TSNR to execute Moving Object Detection, Stationary Object Detection, Railroad Crossing Detection, and Wheel Slip Detection.	HTR Train, TSNR

4.4.3 Logger

Class Name: Logger	
Description: *Logs all terminal requests, IoT outputs, and generated recommendations to a file.*	
Responsibility:	**Collaborators:**
createLog(): Creates a new log file containing the current time and date of creation. Log files are always created during IoT's boot up process.	IoT
updateLog(): Adds and saves a new line of text to the log file. The added text can be supplied from multiple classes such as login dates, generated recommendations, and sensor data calculations.	IoT, Login System, TSNR

4.4.4 Login System

Class Name: Login	
Description: *Allows HTR train operators and engineers to access IoT through a login system.*	
Responsibility:	**Collaborators:**
userExists(): Uses the HTR Employee database to verify the identity of the provided username and password. Access to IoT's main terminal will be granted once the login is successful.	Terminal
getClearance(): Uses the HTR Employee database to find the clearance level of the provided username. The clearance level will then be used by the terminal to accept or reject command line requests.	Terminal

userInit(): Will repeatedly prompt the user for their username and password. The prompting will only stop when the provided username and password is valid using the userExists() function. A success message will be displayed when the correct username and password is entered.	Terminal

4.4.5 Terminal

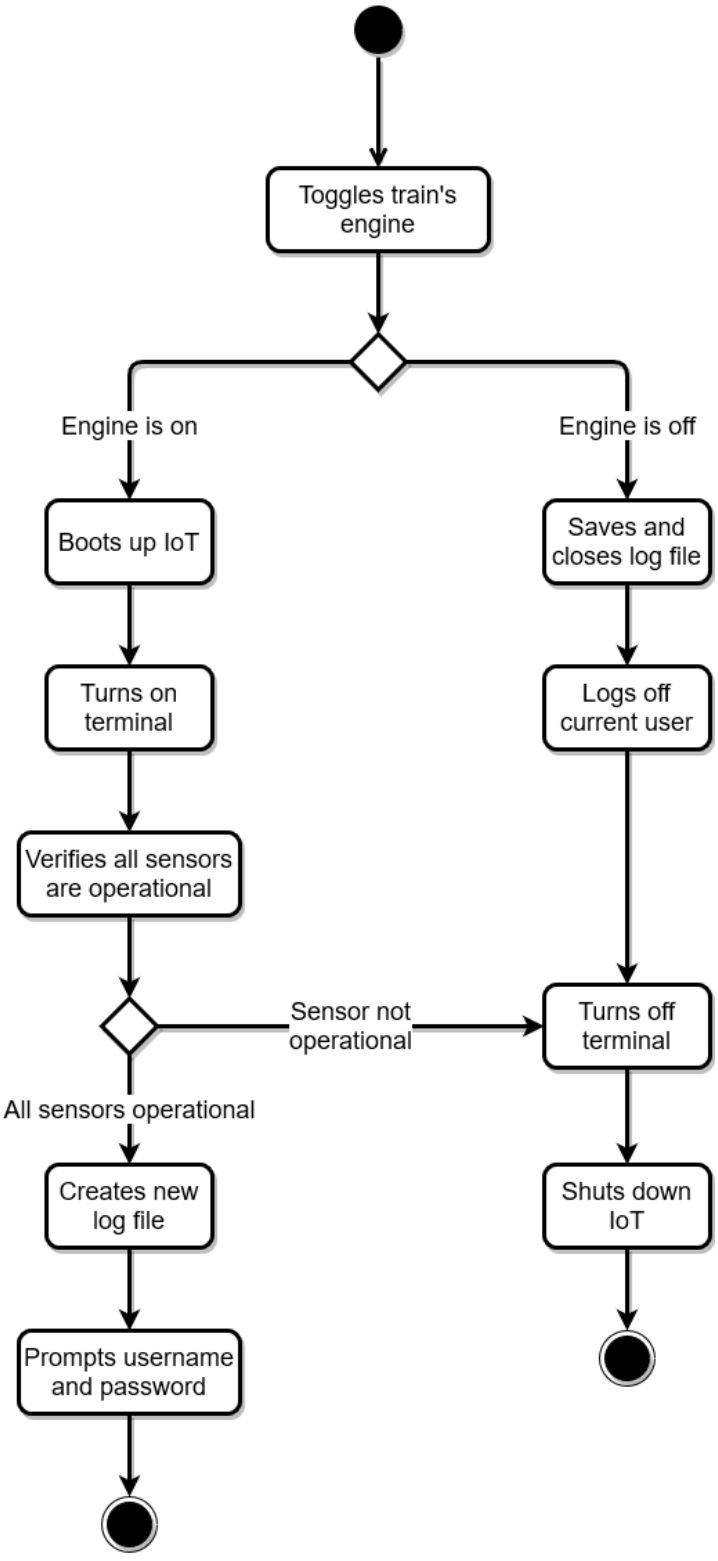
Class Name: Terminal	
Description: *Interface that connects the train operator and engineer to IoT.*	
Responsibility:	**Collaborators:**
printMsg(): Prints a message to the terminal's console. Mostly used by IoT to handle user prompting and outputting general messages not specific to icons.	IoT
getInputField(): Sends the user's entered command line request to IoT for parsing. IoT must check the user's clearance to execute requests.	IoT, Login System
setIconFeed(): Modifies the text associated with each icon. Will be useful for displaying specific speed, distance values, etc.	IoT
setIconVisibility(): Will display or hide the required warning/speed recommendation icon after IoT completes its detection algorithm. Colors can also be applied to the icons for a better user experience.	IoT

4.4.6 TSNR

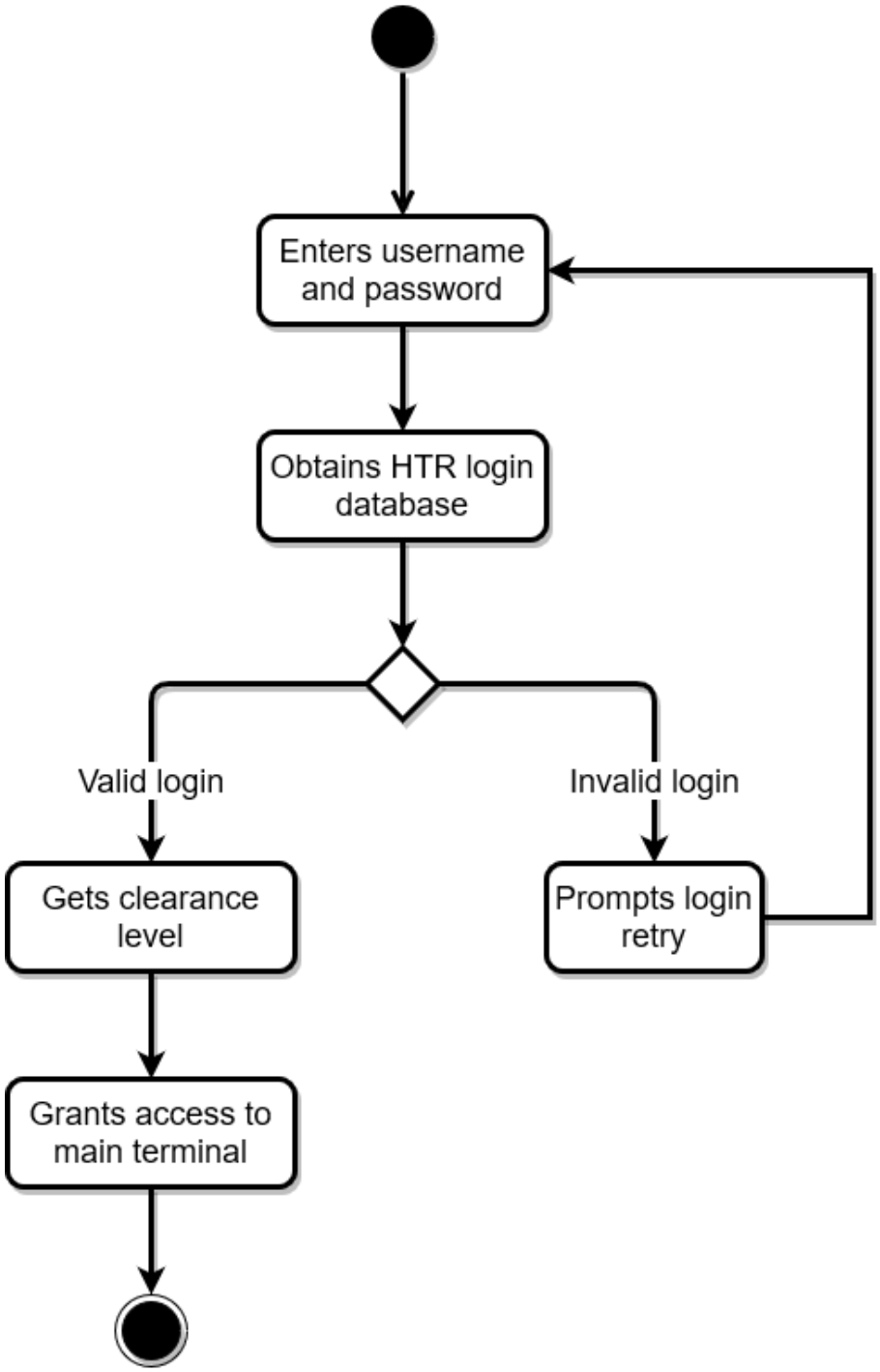
Class Name: TSNR	
Description: *Sends collected data from the sensors to IoT for processing.*	
Responsibility:	**Collaborators:**
getSpeed(): Obtains the current speed of the train outputted by the GPS unit.	IoT
getObjectDist(): Obtains the distance from the train to the nearest object outputted by the first Lidar sensor.	IoT
getObjectDir(): Obtains the direction that the nearest object is heading outputted by the proximity sensor.	IoT
getCrossingDist(): Obtains the distance from the train to the nearest Railroad Crossing outputted by the second Lidar sensor.	IoT
getGatesOpen(): Obtains the gate position of the nearest Railroad Crossing outputted by the optical sensor.	IoT
getW1RPM(), getW2RPM(), getW3RPM(), getW4RPM(): Obtains the RPM of wheels 1 through 4 based on their respective RPM sensors.	IoT
getPacket(): Combines the output of the GPS unit, 2 Lidar sensors, proximity sensor, optical sensor, and 4 wheel RPM sensors into a single packet. Ensures uniformity of sensor data for IoT.	IoT

4.5 Activity Diagrams

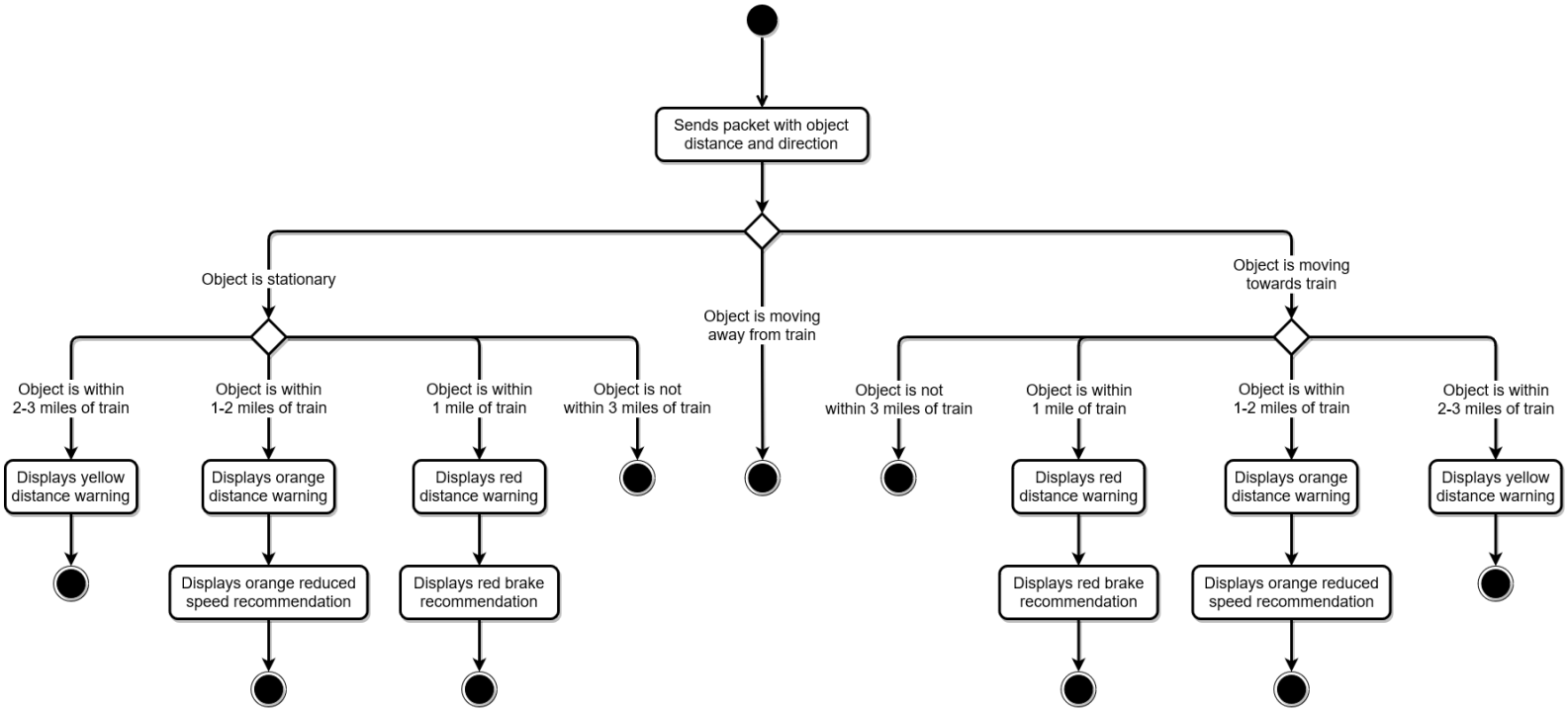
4.5.1 Use Cases 1 & 8: Activation & Deactivation of IoT



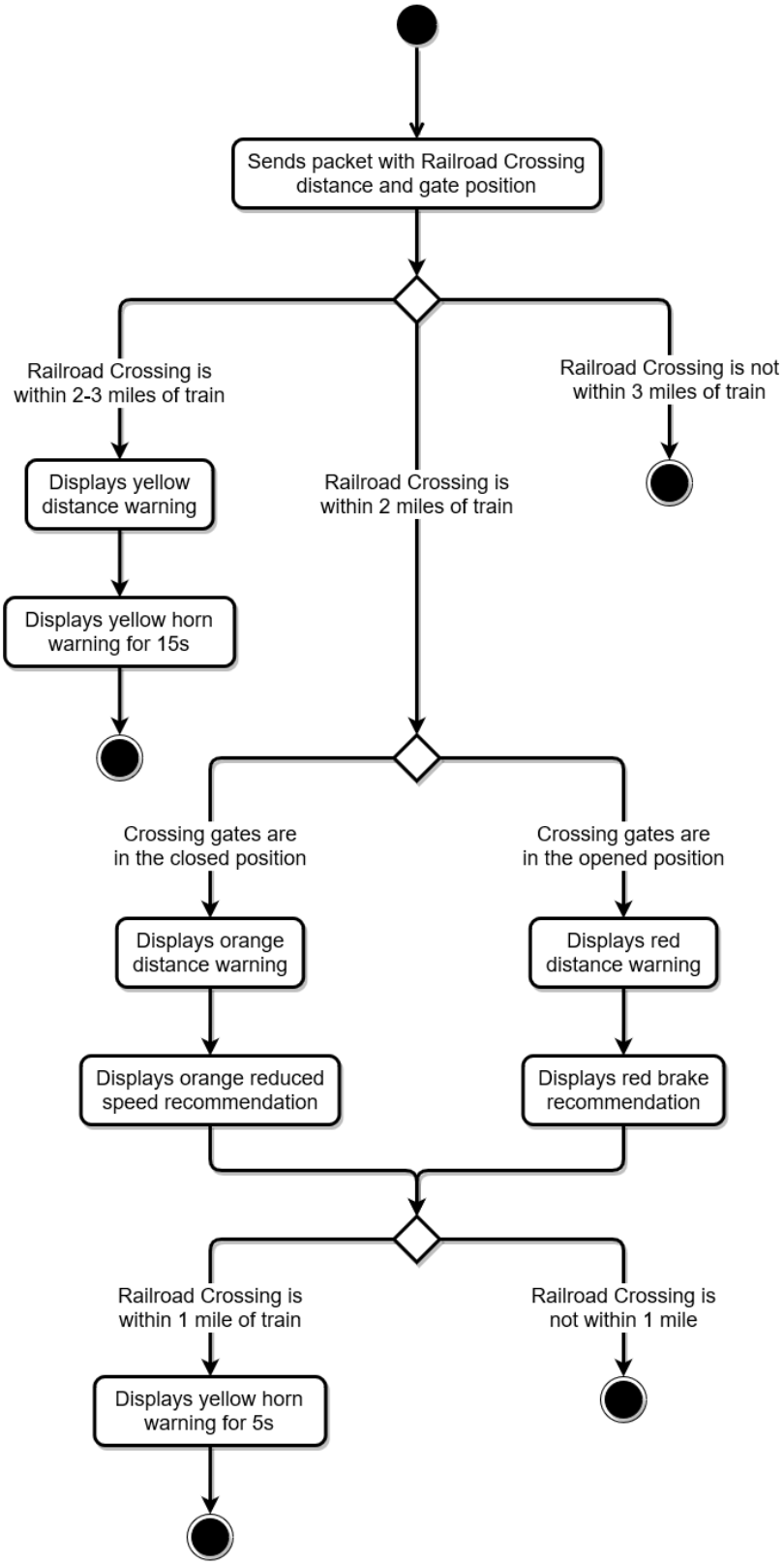
4.5.2 Use Case 2: Access to Main Terminal



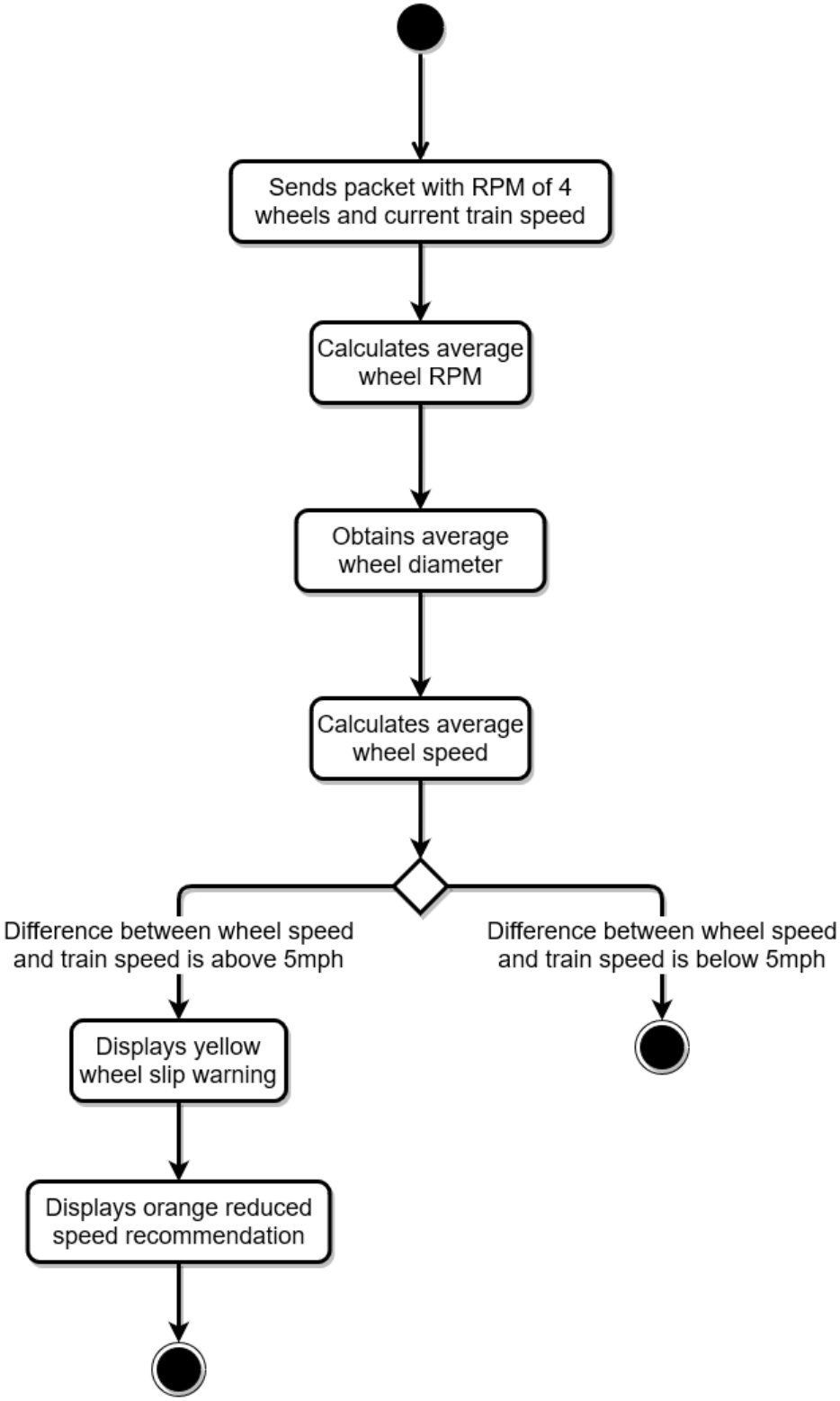
4.5.3 Use Cases 3 & 4: Moving & Stationary Object Detection



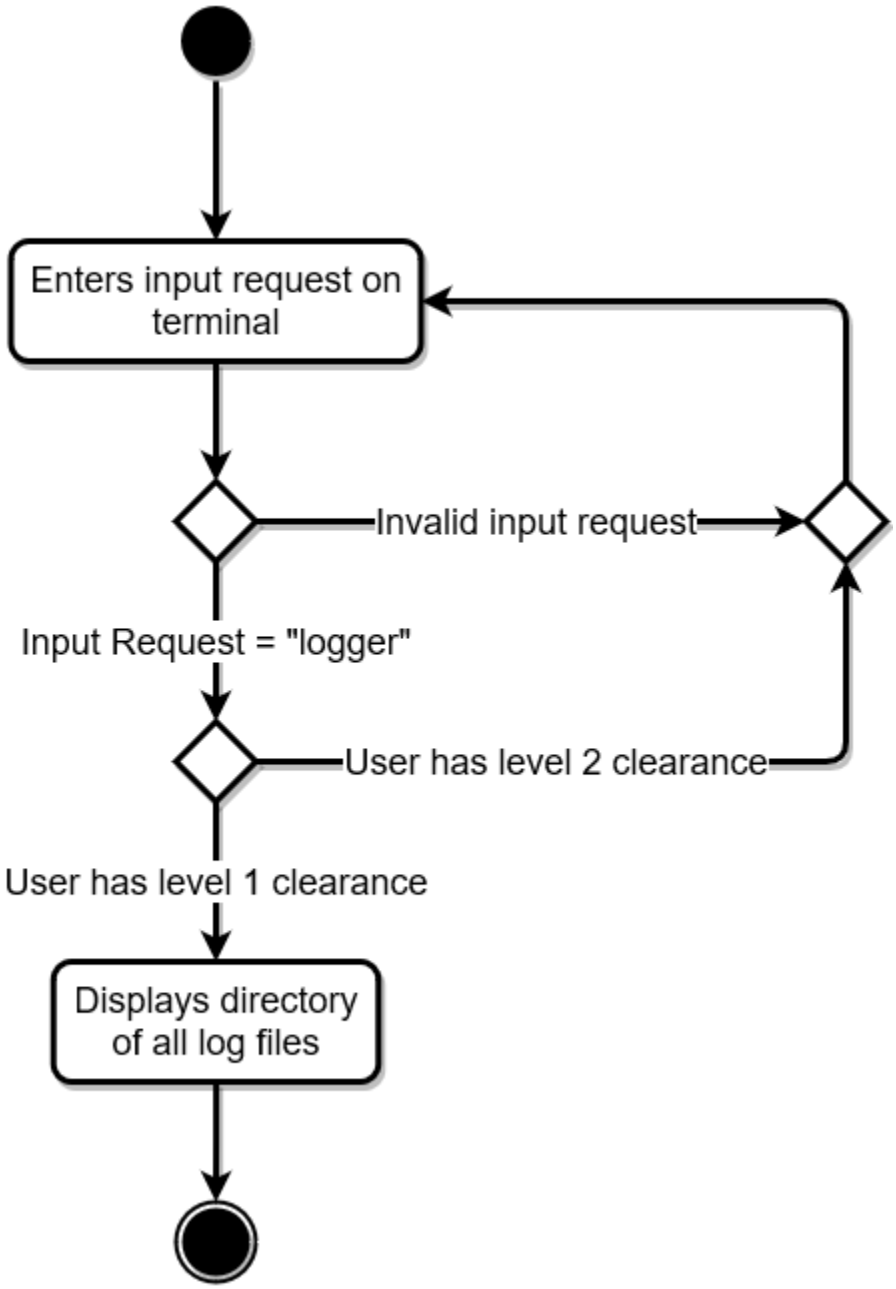
4.5.4 Use Case 5: Railroad Crossing Detection



4.5.5 Use Case 6: Wheel Slip Detection

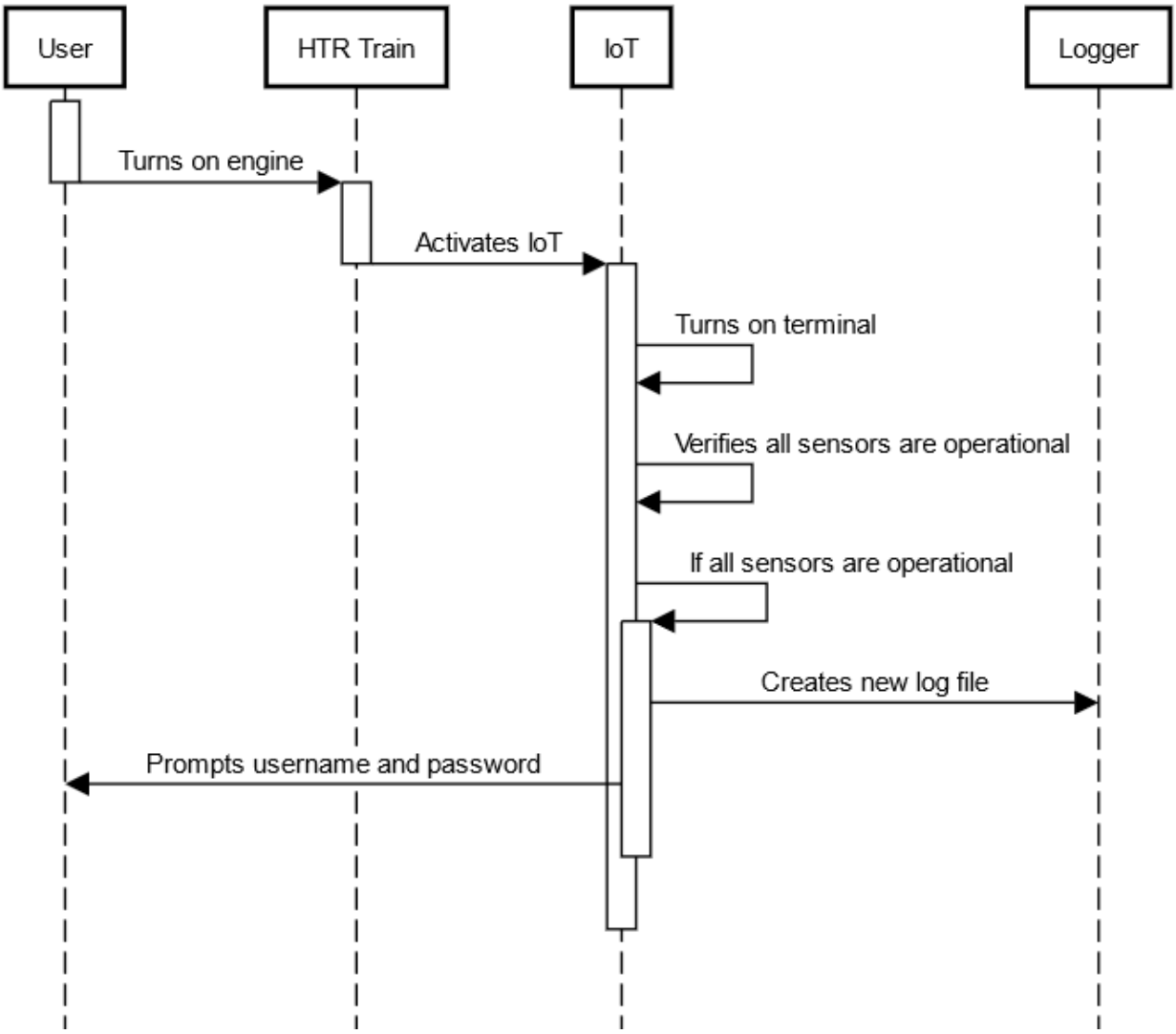


4.5.6 Use Case 7: Access to Log Files

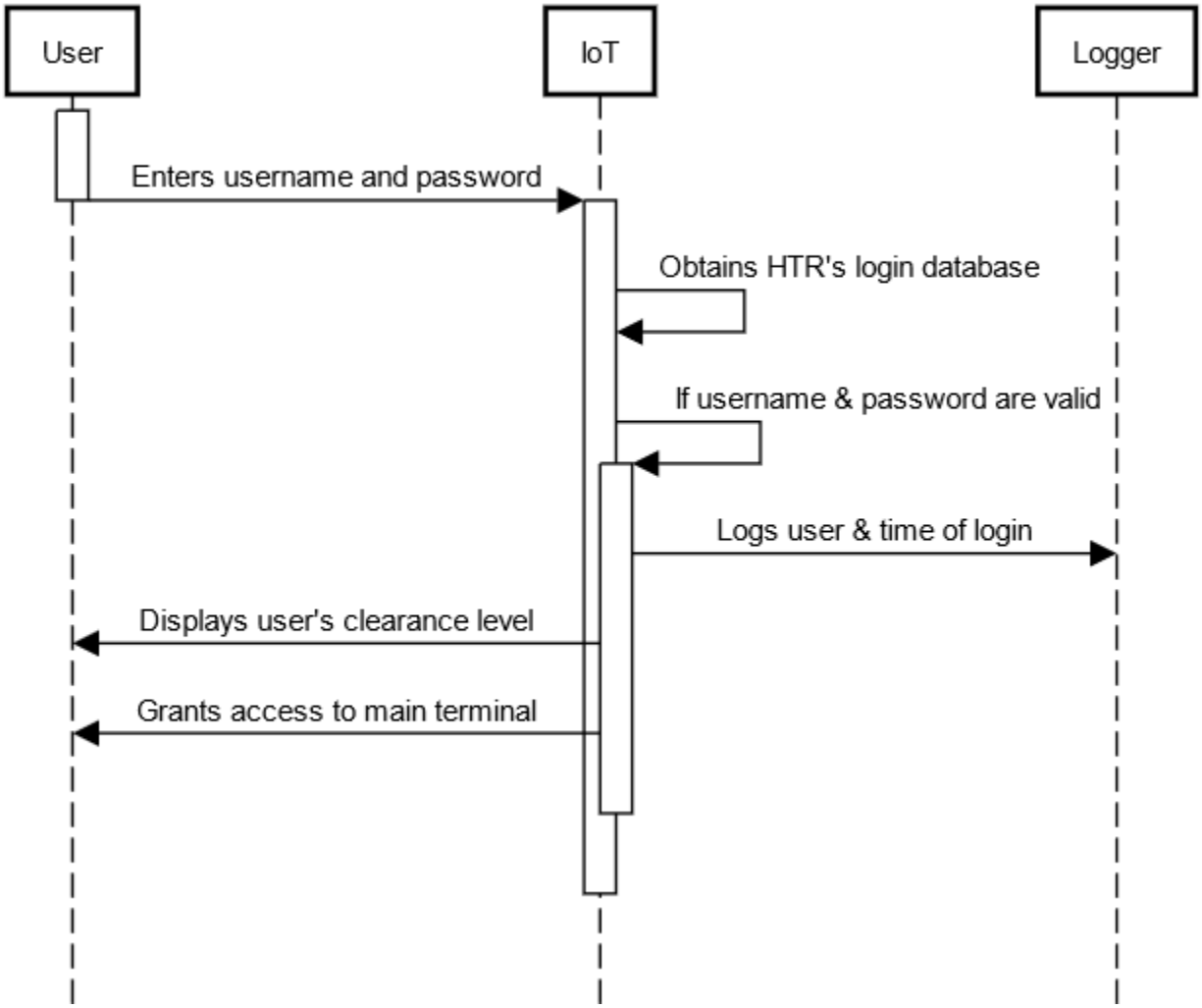


4.6 Sequence Diagrams

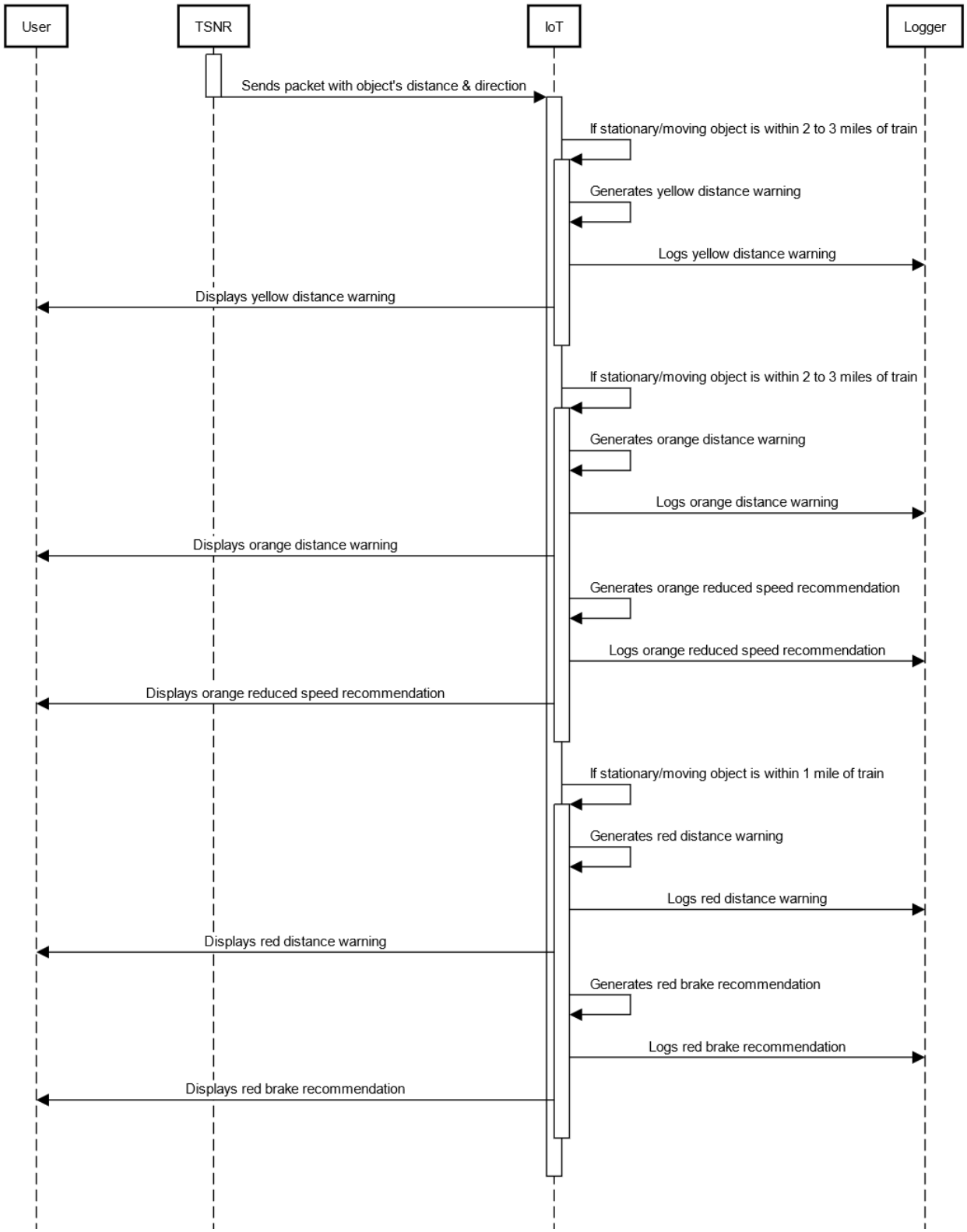
4.6.1 Use Case 1: Activation of IoT



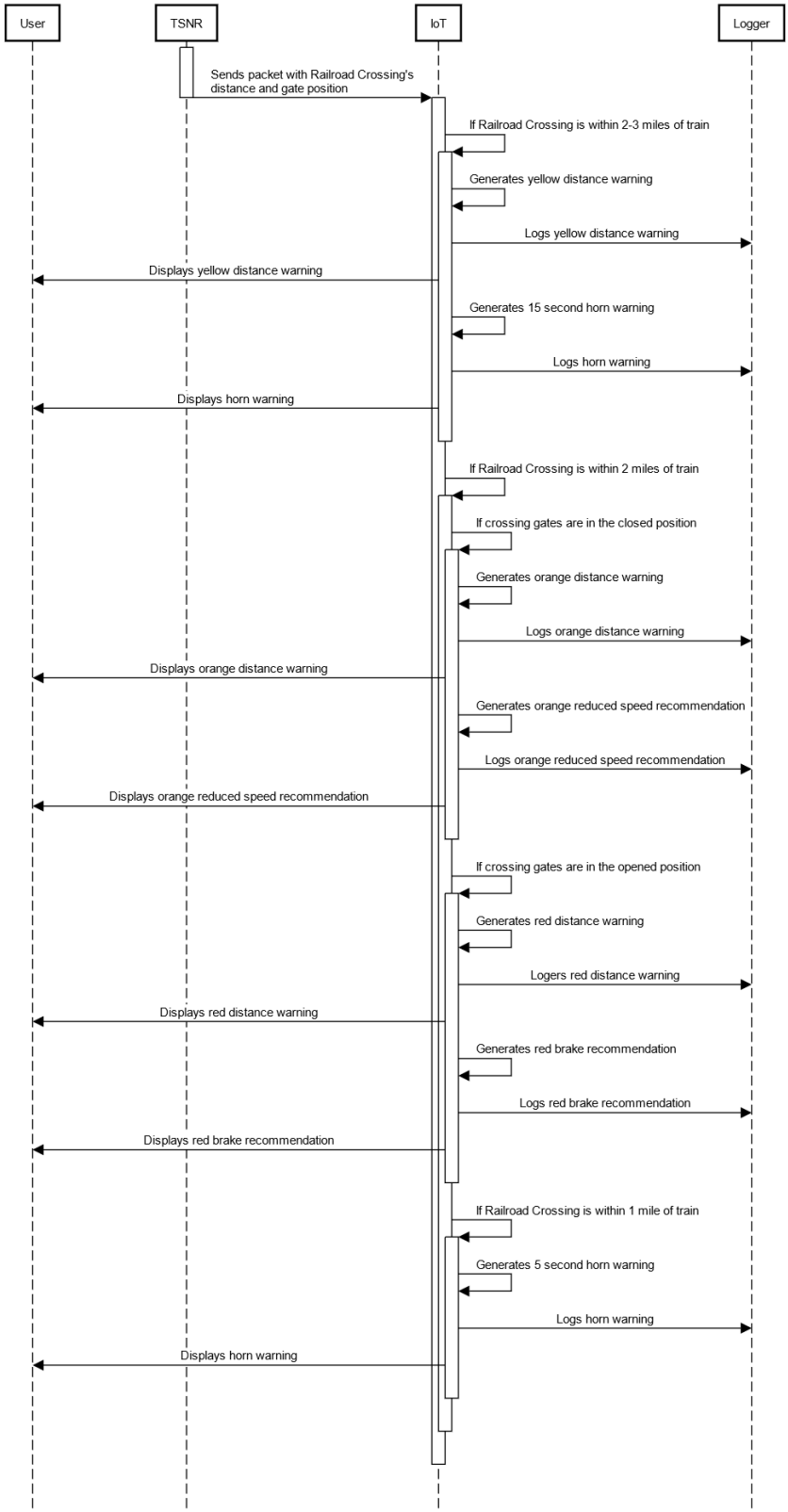
4.6.2 Use Case 2: Access to Main Terminal



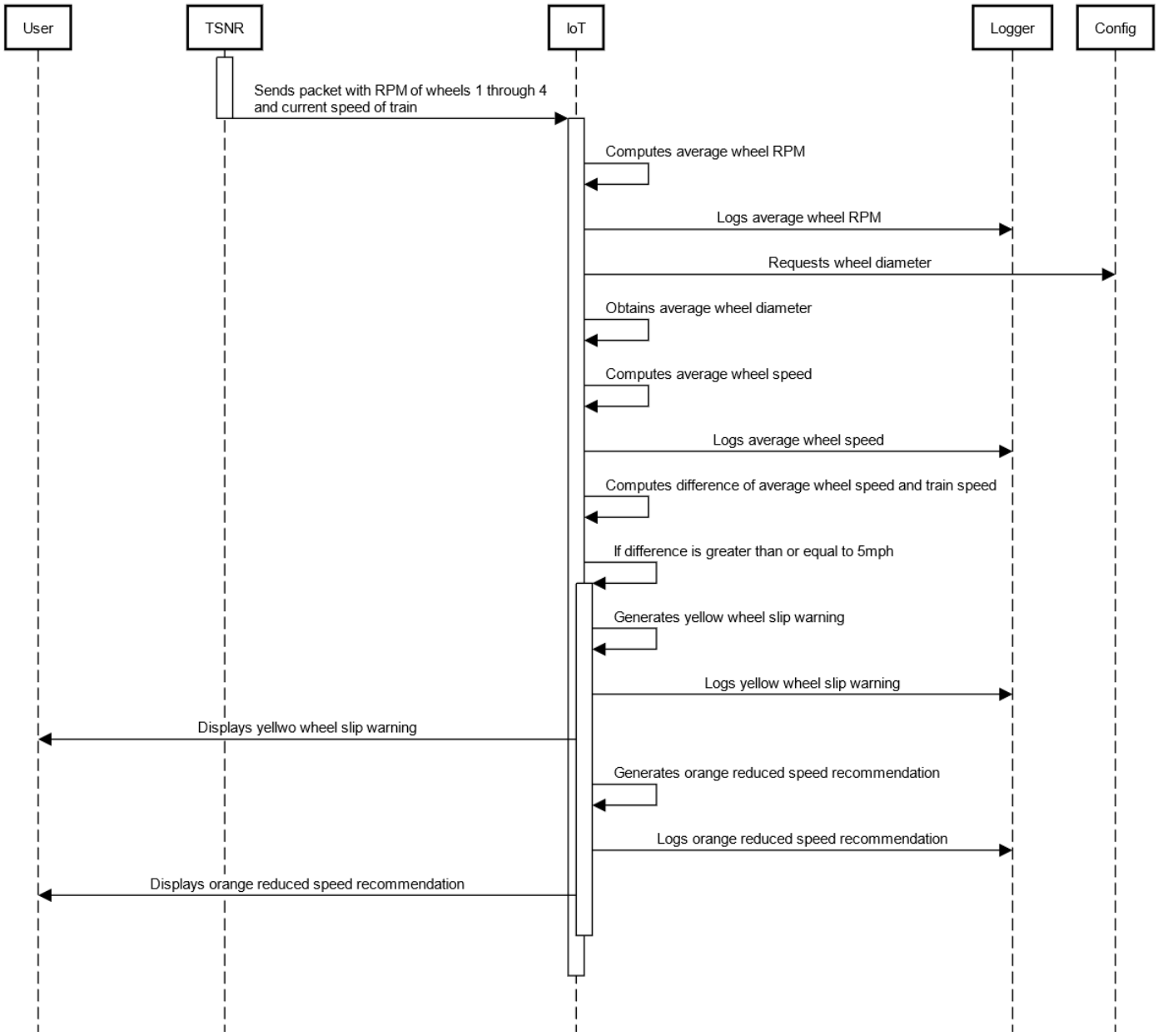
4.6.3 Use Cases 3 & 4: Moving & Stationary Object Detection



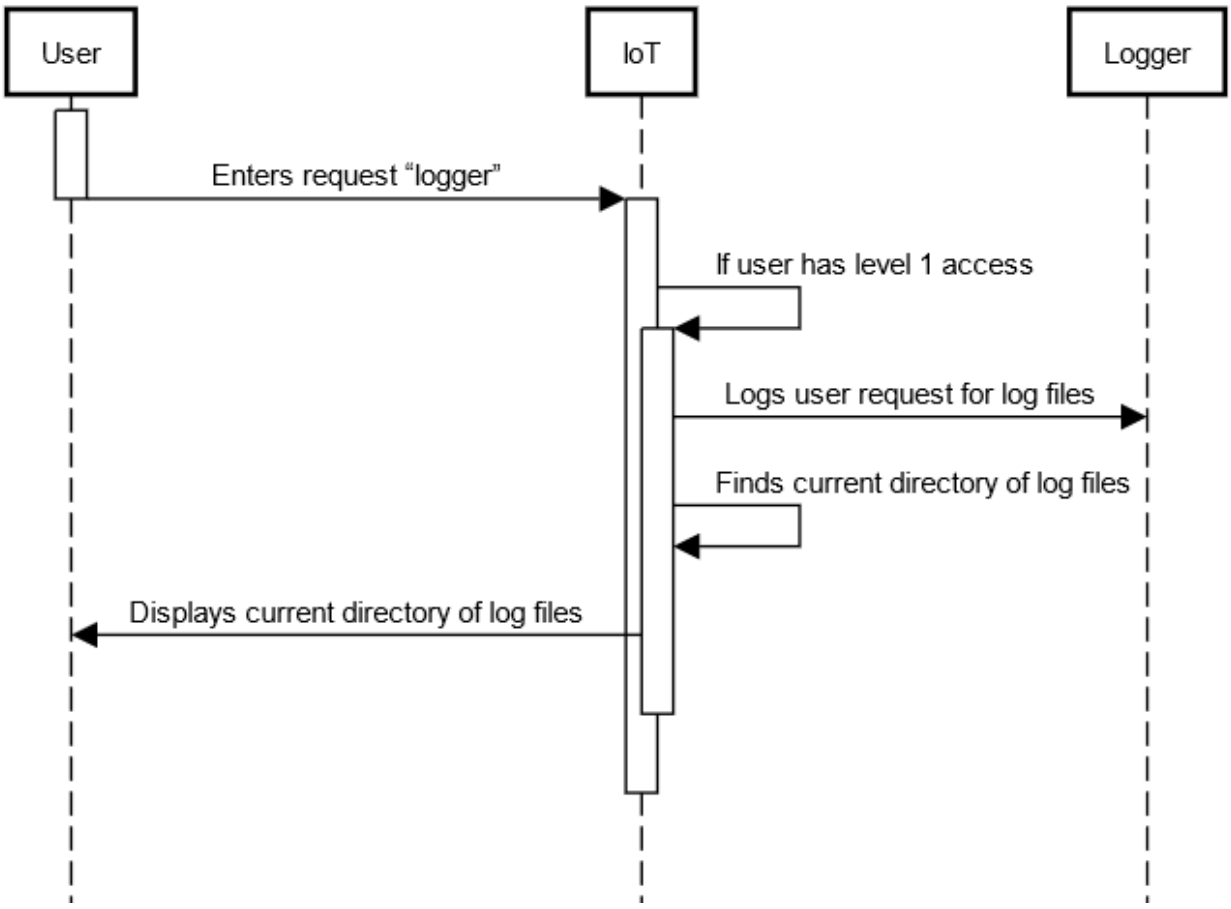
4.6.4 Use Case 5: Railroad Crossing Detection



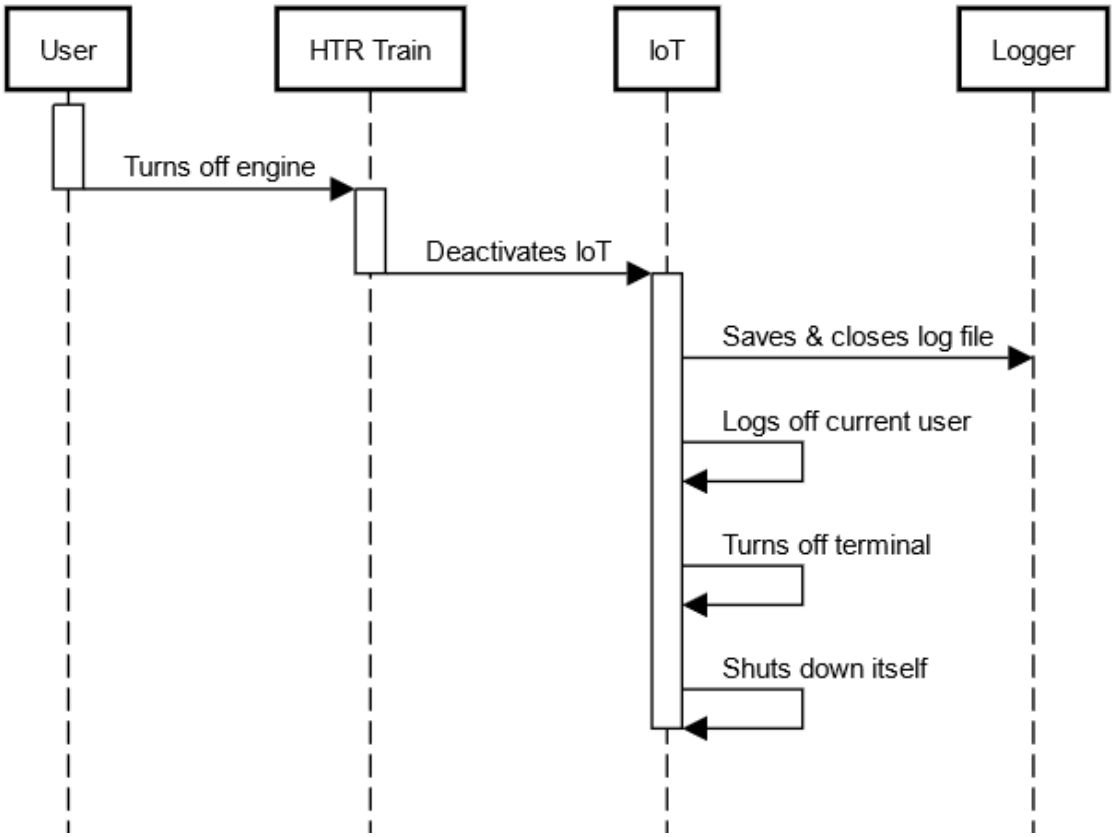
4.6.5 Use Case 6: Wheel Slip Detection



4.6.6 Use Case 7: Access to Log Files

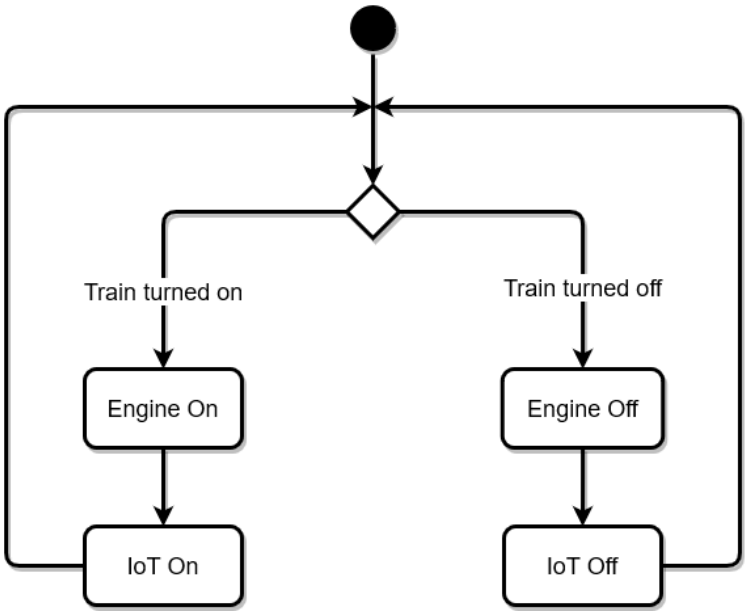


4.6.7 Use Case 8: Deactivation of IoT

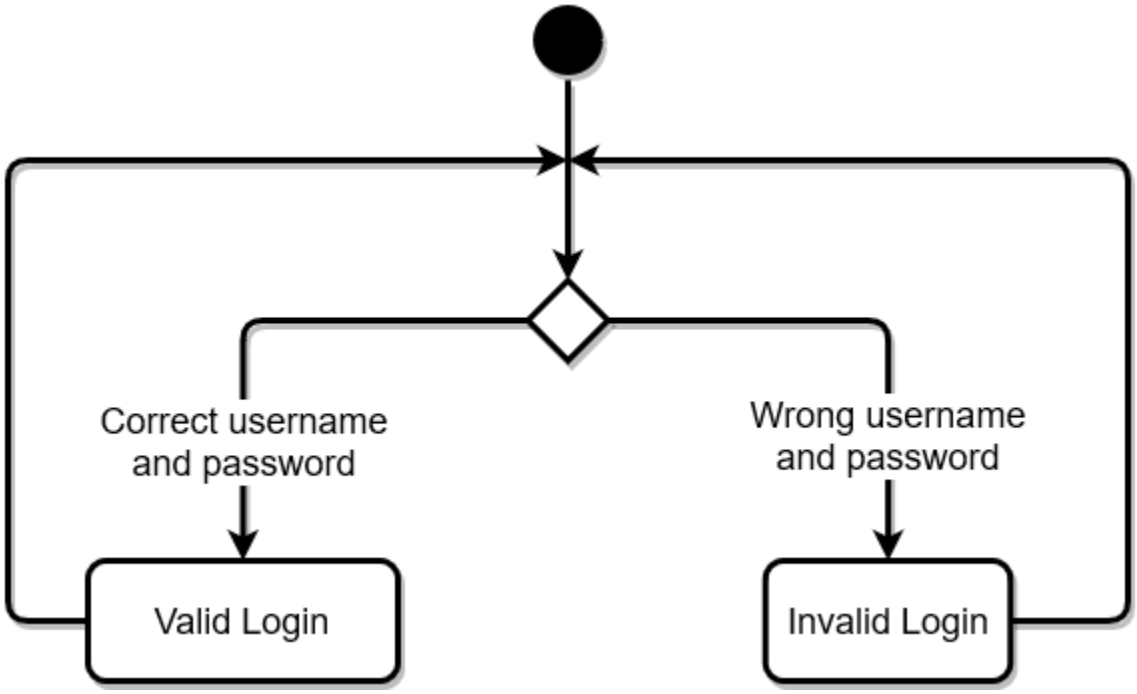


4.7 State Diagrams

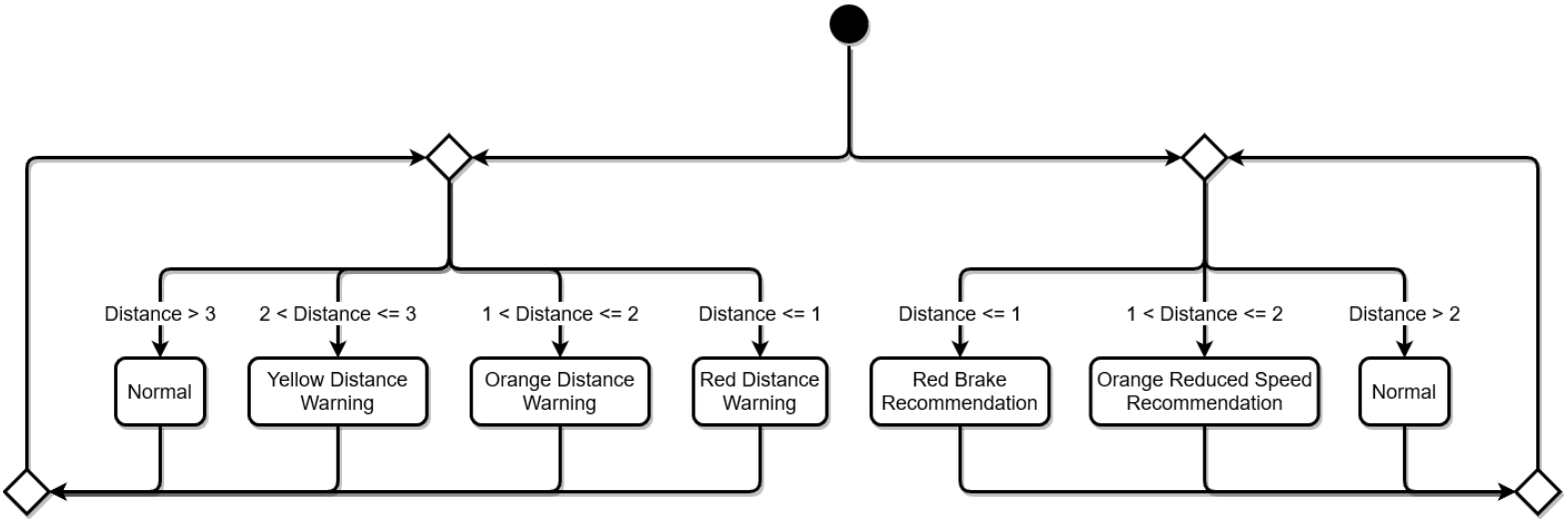
4.7.1 IoT State



4.7.2 Login State

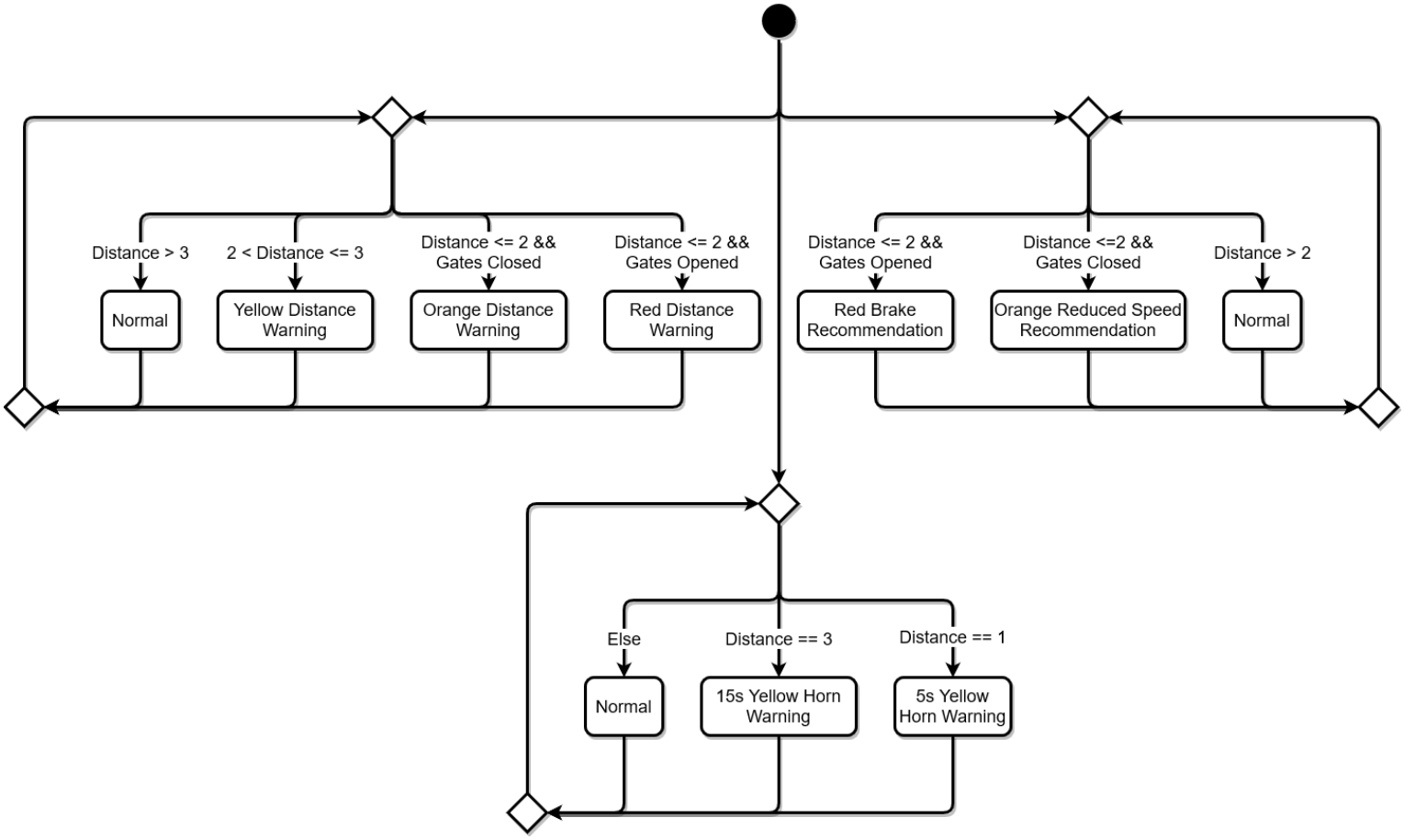


4.7.3 Object Obstruction State



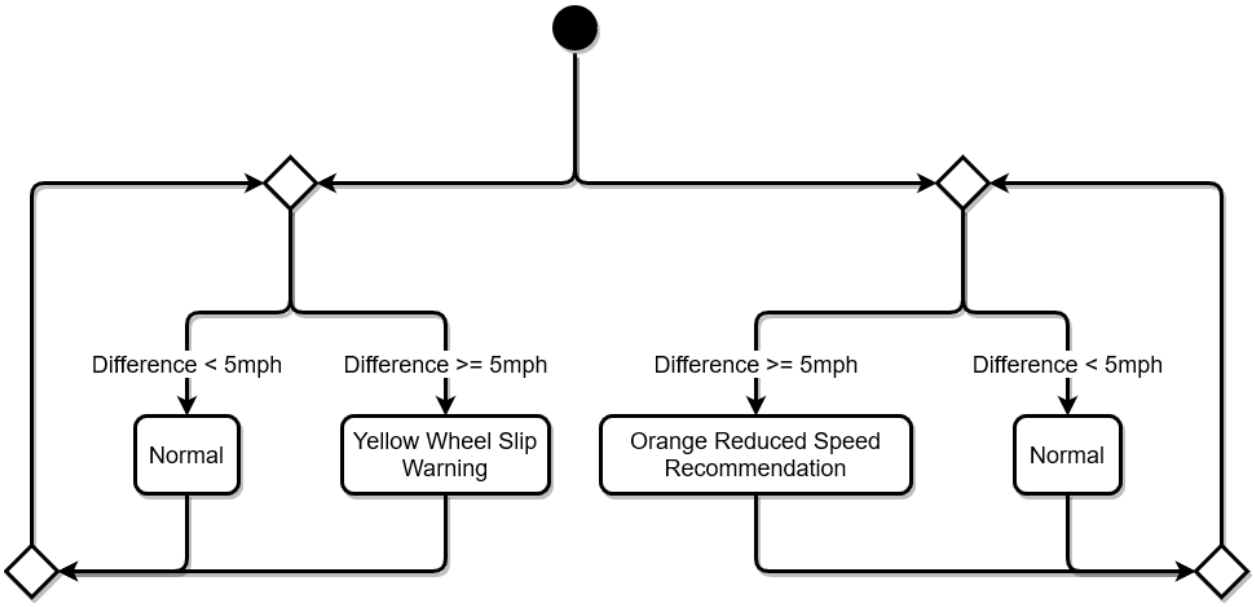
Note: Distance refers to the object's distance from the train.

4.7.4 Railroad Crossing State



Note: Distance refers to the Railroad Crossing's distance from the train.

4.7.5 Wheel Slip State



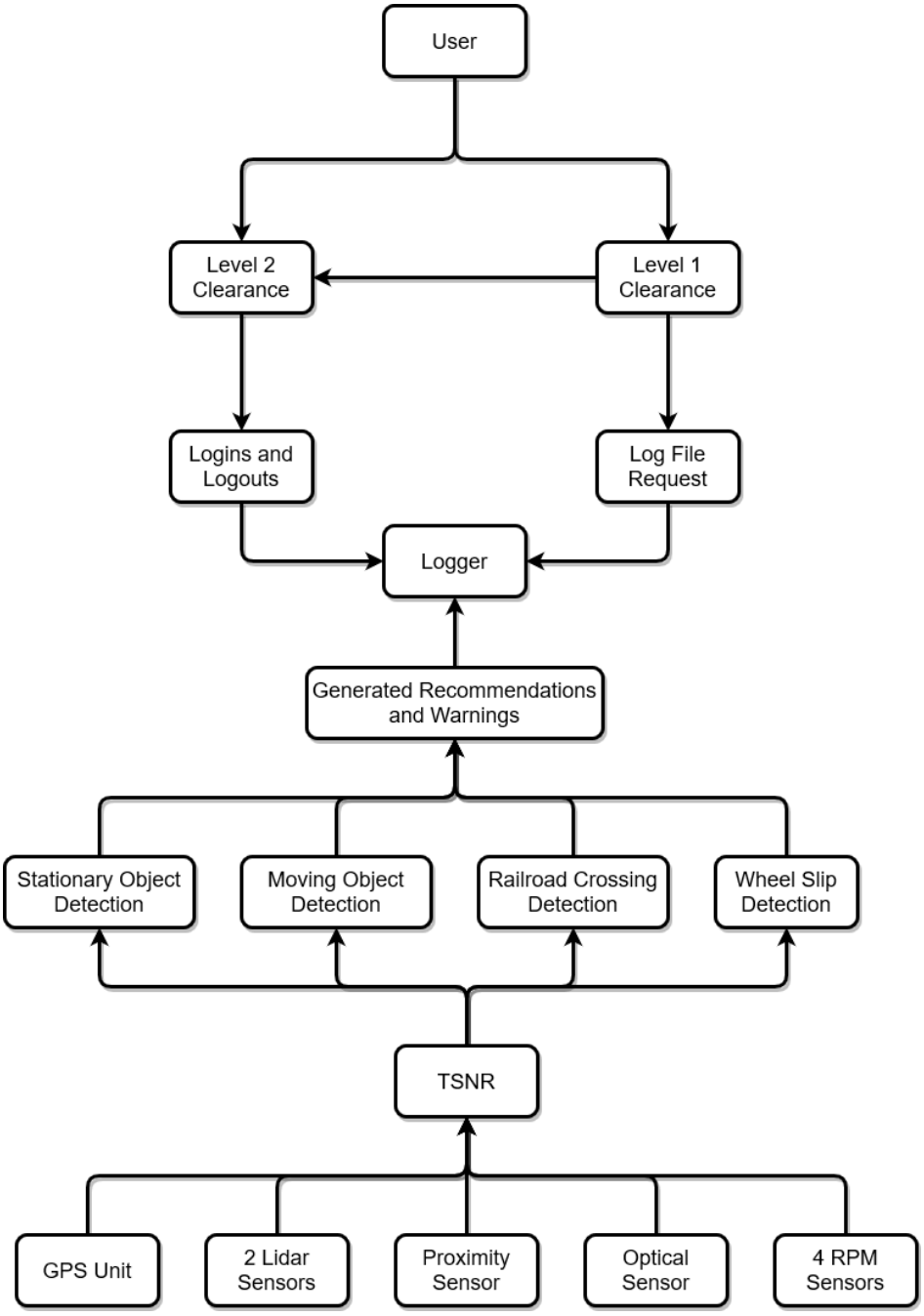
Note: Difference refers to the difference between the wheel speed and train speed.

---

5) SECTION FIVE

5.1 Data Centered Architecture

5.1.1 Model



5.1.2 Pros

1. All IoT data is logged to a central system
2. Logger is easily accessed from a single location
3. Very modular and scalable in adding more software modules

5.1.3 Cons

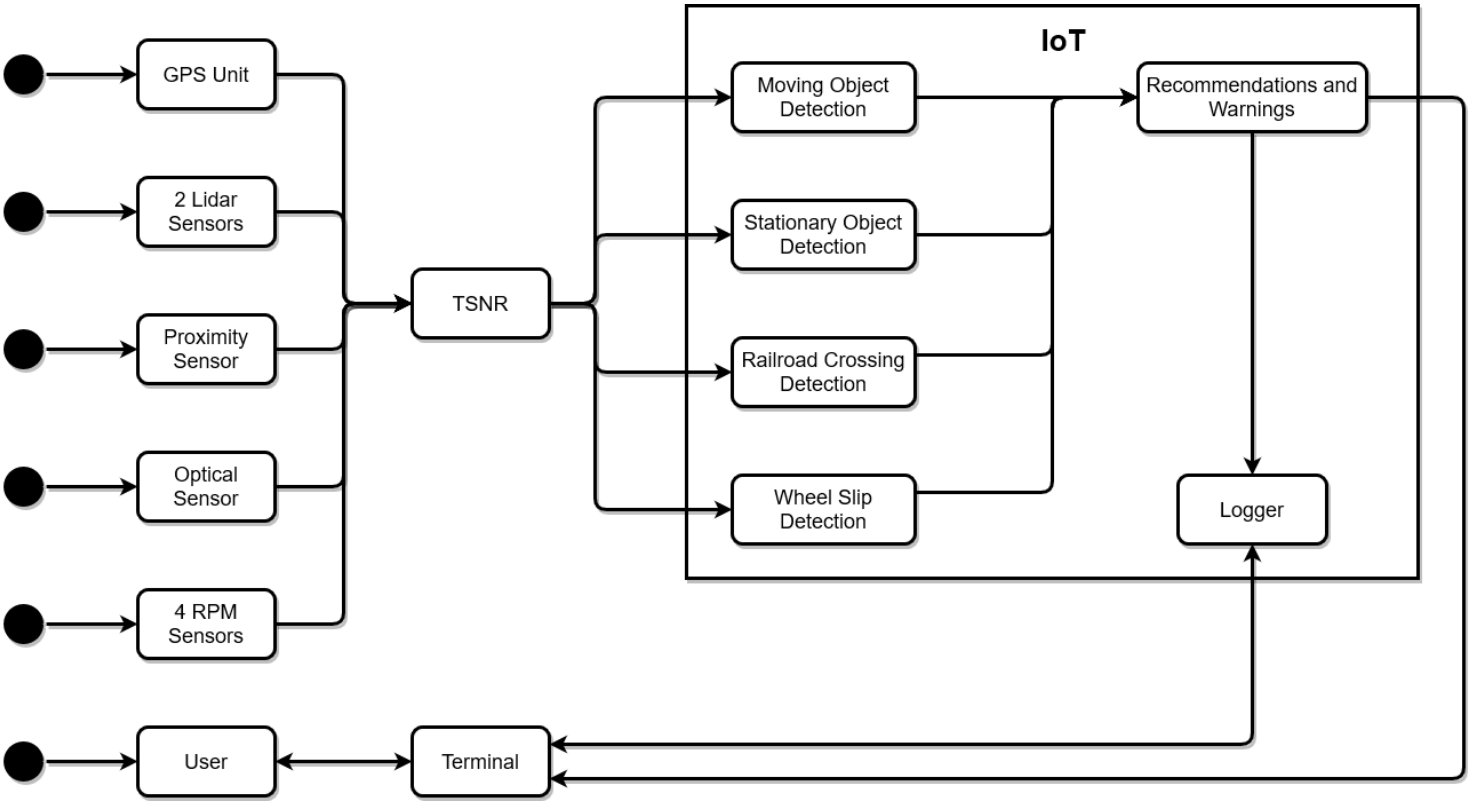
1. High dependency between the central system and individual software
2. Failure of central system propagates into a failure of the whole logger
3. Data structure of central system must be consistent for all software
4. Logger must manage high traffic requests

5.1.4 Remarks

The Data Centered Architecture does not suit IoT's dynamic nature. Being primarily a storage-based architecture, the only IoT module that somewhat fits this role is the logger. Even then, the logger is not a suitable choice for utilizing the Data Centered Architecture. Each log file stores IoT's data from turning on to turning off the train, which does not require massive storage space typically found in a large database. Additionally, the logger constantly updates and changes its stored values, which is not a feature of long-term database storage.

5.2 Data Flow Architecture

5.2.1 Model



5.2.2 Pros

1. Allows IoT to have high throughput of data processing
2. Simplifies IoT's software execution into block chains
3. Each block chain can run sequentially or independently in parallel
4. Individual software filters are easy to maintain and modify

5.2.3 Cons

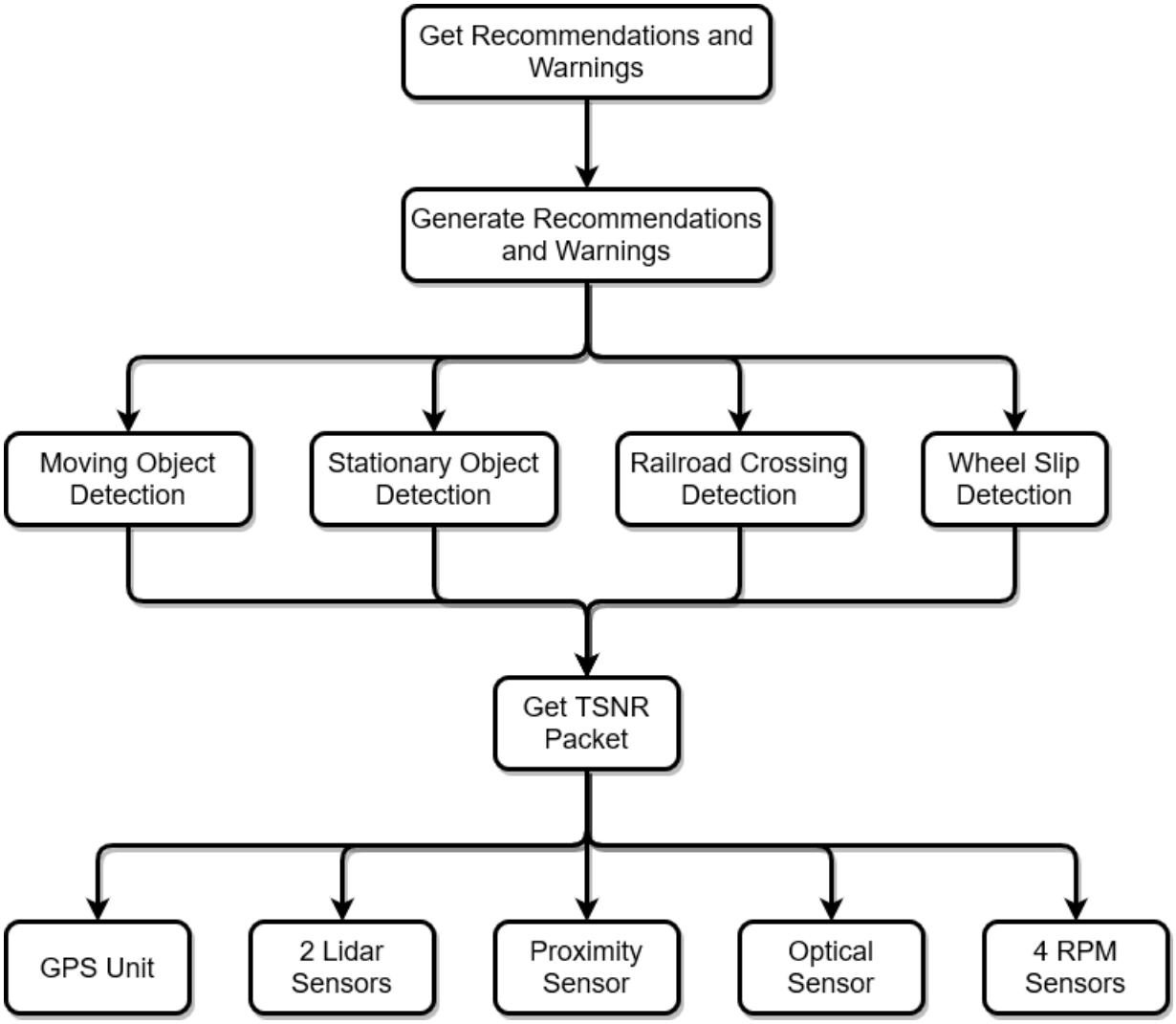
1. More sequential software filters increases IoT's processing time
2. Does not allow IoT to modify/redirect existing filters dynamically
3. Failure in a software filter will create bottlenecks in subsequent filters
4. Software filters have a hard time working on a large problem
5. Software filters cannot effectively store calculations

5.2.4 Remarks

The Data Flow Architecture is a strong option for IoT. One of the highlights of this architecture is the ability for IoT to run data processes in parallel. By running the 6 sensors and 4 detection functions in parallel, IoT would have a higher throughput of data. Additionally, chains of parallel filters create distinct blocks for the data to flow. This simplifies the development of IoT's software because our team will be able to independently and simultaneously code each block sequence of the Data Flow Architecture. Furthermore, these blocks of filters enable easier readability and maintenance of IoT.

5.3 Call-Return Architecture

5.3.1 Model



5.3.2 Pros

1. Reduces complexity of IoT into smaller and simpler subprograms
2. Subprograms can be hidden from one another
3. Groups of subprograms create easy to understand hierarchy of IoT
4. Modular in adding new subprograms to IoT

5.3.3 Cons

1. High dependency between output of one subprogram and the input of another subprogram
2. Failure of subprogram will propagate up into IoT's main process
3. Does not scale well as IoT's complexity increases
4. More subprograms increases process time of IoT

5.3.4 Remarks

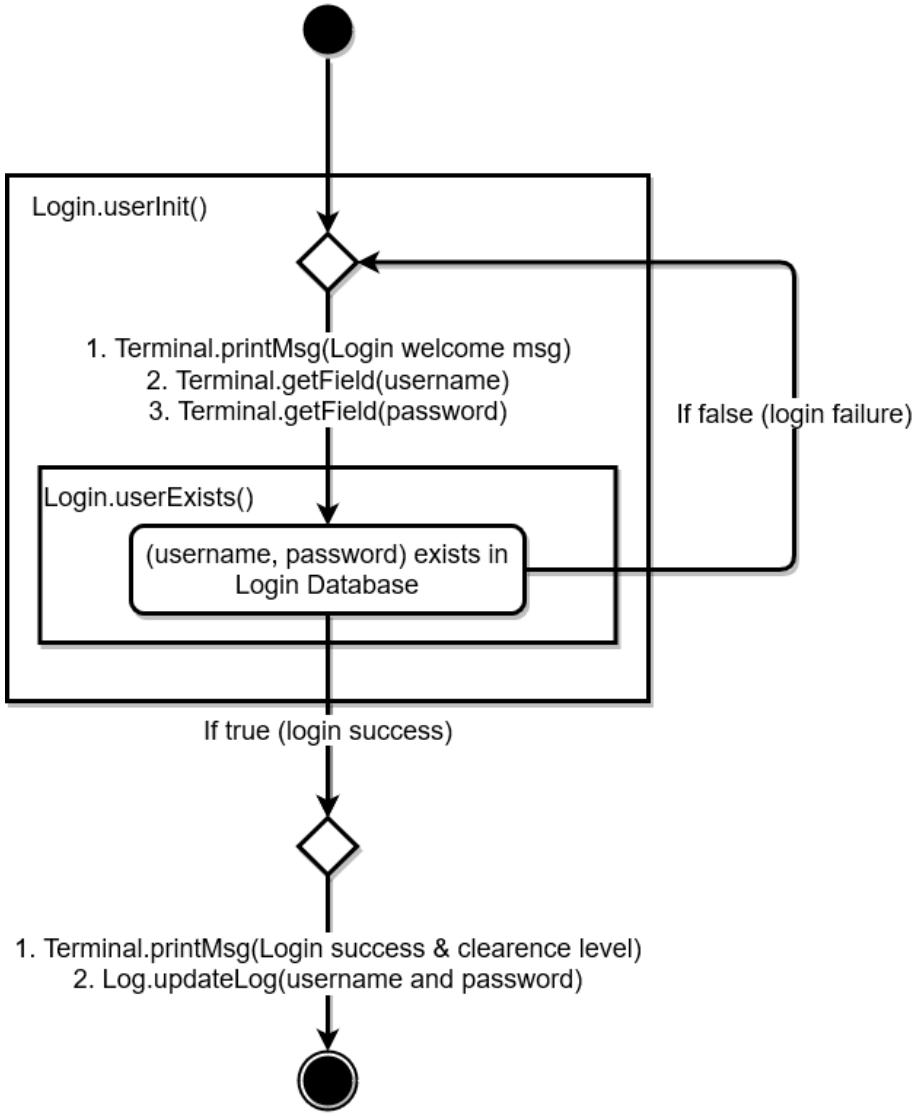
The Call-Return Architecture is a superb option for IoT because of its hierarchical system. Through visualizing IoT from a hierarchical perspective, complex processes such as recommendation and warning generation can be further broken down into much simpler and easier to manage subproblems. For example, all terminal requests in IoT can be grouped and divided into Level 1 and Level 2 requests, and speed recommendations can be derived from the 4 detection functions.

Additionally, the hierarchical approach in the Call-Return Architecture satisfies IoT's computation requirement. At every second, IoT must process TSNR's packet which includes data from 6 sensors. Then, IoT must complete calculations for each individual sensor data. This requires IoT to be quick in computing, storing, and retrieving values to anticipate the next TSNR packet a second later. Because this architecture is built on sharing and returning data between subprograms, this scenario is perfect for supporting IoT's calculation process.

The only fault in this architecture is complications arising from scaling up IoT. When IoT becomes bigger, there must be exponentially more subprograms in the hierarchy, which will slow down development and computation time.

5.4 Object-Oriented Architecture

5.4.1 Model



5.4.2 Pros

1. Uses abstraction to reduce complexity of IoT
2. Objects and methods are reusable, modular, and dynamic
3. Implementation of IoT objects are hidden from one another
4. IoT objects can be constructed and executed independently

5.4.3 Cons

1. Inefficient for high performance calculations in IoT
2. Prone to excess and large amounts of code
3. Difficult implementation without a high level plan of IoT

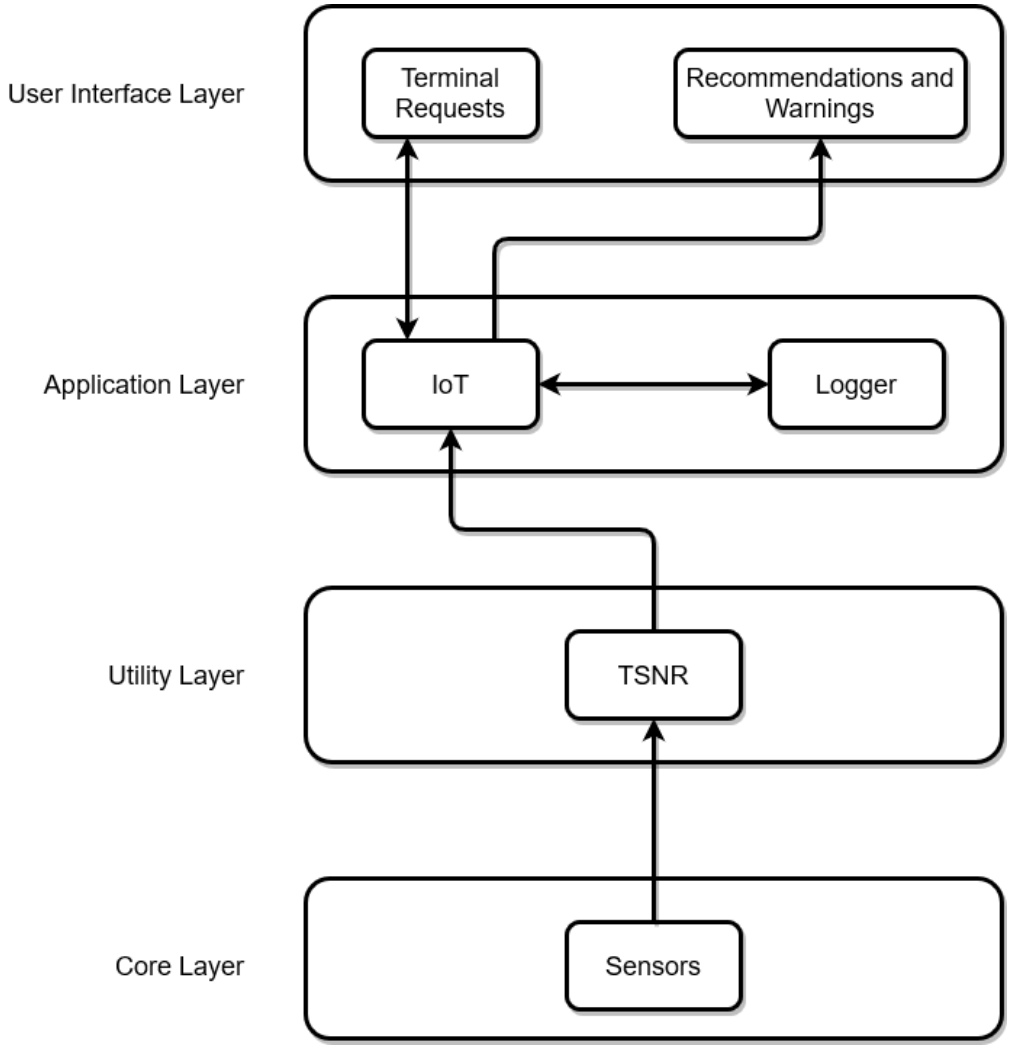
5.4.4 Remarks

The Object-Oriented Architecture is a strong fit for IoT. The main advantage of utilizing this architecture is using abstraction to minimize and simplify IoT's software. By only specifying required classes and methods in IoT's objects, abstraction enables only the bare minimum and necessary amount of code to sufficiently run IoT. Thus, the run time of IoT can be reduced while simultaneously increasing overall efficiency, readability, and performance of IoT. Furthermore, by creating distinct classes such as the Config or Logger, IoT's objects can run independently without the risk of one object's dependencies clashing with another object.

However, the Object-Oriented Architecture assumes that IoT's abstracted code is well optimized for performance and thoroughly planned out. Objects require an abundance of space to store methods and stored data. When the code is unoptimized, duplicate and unneeded objects may still linger after its initial use, taking up valuable space in IoT. Additionally, it is difficult to optimize IoT's code without a thorough understanding and high level software plan of IoT. There needs to be a balance between having the bare minimum amount of code and a sufficient amount to successfully execute IoT.

5.5 Layered Architecture

5.5.1 Model



5.5.2 Pros

1. Distinct layers simplifies complexity and understanding of IoT
2. Divides IoT into 4 unique hardware, software, and interface layers
3. Changes in one layer do not significantly impact development of other layers
4. Robust for grouping and customizing IoT's functionality, specs, and interface

5.5.3 Cons

1. More independent layers make it harder to maintain and modify IoT
2. Not effective for higher modularity and scalability of IoT
3. Moving through multiple layers lowers performance and increases process time for IoT
4. Layers that are far apart have a hard time working together

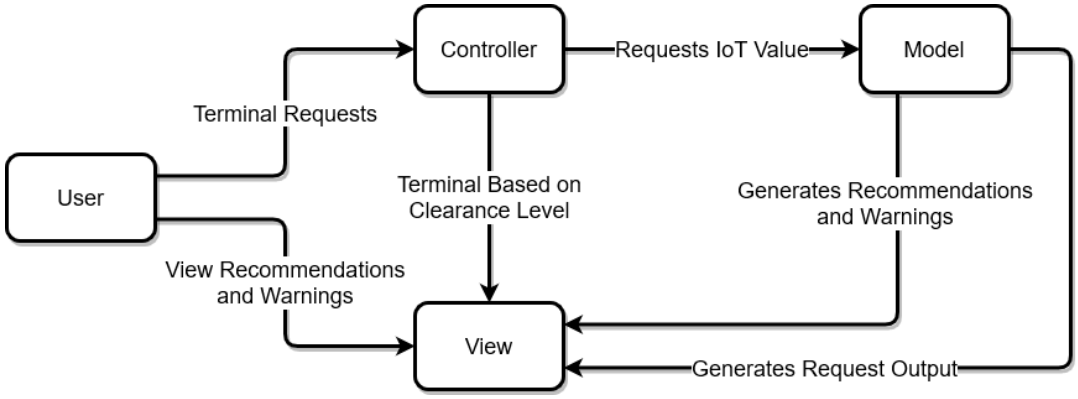
5.5.4 Remarks

The Layered Architecture is also an excellent fit for IoT due to its powerful organization of IoT's design layout. By separating IoT's hardware, software, and interface into 4 unique layers, each layer will have a designated purpose in IoT that makes customization and maintenance of the software easier. Furthermore, having unique layers enables independence during the software development process. For example, modifying existing or adding new Terminal requests in the User Interface Layer will not have a significant impact on the Utility and Core Layers, as these layers are far away from the User Interface Layer. This ensures faster development of IoT without severely altering development of other layers.

Similar to the Call-Return Architecture, the only major flaw of the Layered Architecture is when scaling up IoT. Because there are only 4 unique layers, adding new modules to IoT will overflow their respective layers until it will become hard to organize and maintain each individual layer. However, as stated in the Call-Return Architecture, IoT at this moment is not large enough to encounter this scenario. Thus, the Layered Architecture is a powerful organizer of IoT's layout.

5.6 Model View Controller (MVC) Architecture

5.6.1 Model



5.6.2 Pros

1. Oriented for IoT's user interface
2. Splits IoT into Controller, Model, and View units enables for easier maintenance and readability
3. Changes in one unit do not significantly impact the other two units
4. 3 seperate units ensures independence

5.6.3 Cons

1. Not expandable outside of user interface
2. Difficult implementation without a high level plan of IoT's interface
3. IoT data follows a strict flow in this architecture
4. Model unit becomes more complex as functionalities are added to IoT

5.6.4 Remarks

The Model View Controller Architecture is a decent option for IoT. Similar to how the Layered Architecture improves organization of IoT by separating the hardware, software, and interface into 4 different layers, the MVC Architecture separates IoT's terminal interface into a Controller, View, and Model unit. Each unit is independent and serves its designated role in IoT. For example, the Controller unit is responsible for maintaining user Terminal requests, while the View unit is responsible for displaying recommendations and warnings onto the Terminal screen for the user.

5.7 Architecture of Choice

5.7.1 Our Architecture for IoT

The best architecture for our implementation of IoT is a combination of the Data Flow Architecture and the Object-Oriented Architecture.

5.7.2 Why We Chose the Data Flow Architecture

We first chose the Data Flow Architecture because the architecture is simple to implement and allows the power of parallel computing. By organizing IoT's functions into blocks of filters, our team will have an easier time understanding IoT's complex detection system through breaking down each detection function based on their sensor type. Organizing IoT's detection system through this architecture also enables us to easily maintain IoT during development, as changes can be quickly isolated in each filter.

Furthermore, parallel processing in the Data Flow Architecture is an important advantage. TSNR sends IoT sensor packets at every second, which makes IoT very time sensitive and computation heavy. Thus, it is critical that we reduce data calculation times and loads in IoT. Through parallel processing, we are able to achieve this high performance since parallel pipes increase computational data throughput while simultaneously increasing time efficiency in IoT.

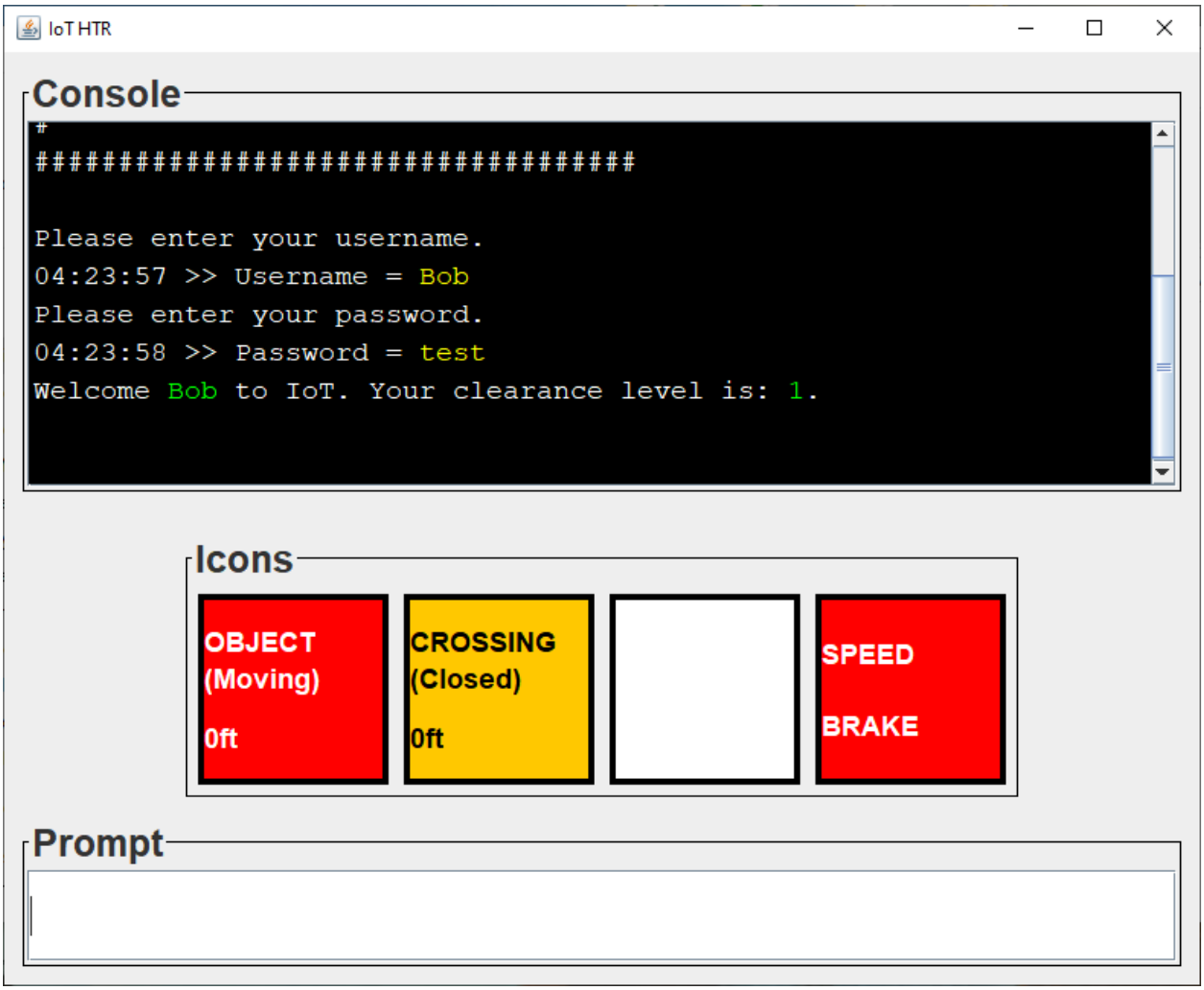
5.7.3 Why We Chose the Object-Oriented Architecture

We then chose the Object-Oriented Architecture mainly because of software abstraction. Managing the software for the 6 different sensors and 4 detection functions requires high organization and an immense understanding of IoT. For example, in our Login System implementation, we must verify a username and password, check the number of login attempts, and see if the user is locked out. However, through abstraction, we may reduce the code's complexity by only implementing what is absolutely necessary for IoT to run. In the above login example, we may simply hide all the calculations and verification steps through a function that parses the entered username and password and returns the login success.

While the Object-Oriented Architecture is not as efficient and performant as other architectures (objects usually require more space to

store attributes and methods), we felt that abstraction and an object's dynamic nature outweighed the performance drawbacks present in this architecture. Additionally, we hoped that parallel processing from the Data Flow Architecture would alleviate some of the performance losses in the Object-Oriented Architecture.

5.7.4 Mockup of the Terminal Screen



6) SECTION SIX

6.1 Sensors

6.1.1 GPS Unit Class

```java
// the purpose of the GPS class is to output the train's current speed in mph to TSNR.
public class GPS {
    // The GPS class holds 2 attributes:
    // - an array of doubles 'speed' that holds a train speed at each array index.
    //   this array enables easy simulation and modification of the GPS unit's train speed output.
    // - an int 'index' that tracks the current array index as the GPS unit outputs the train speed.
    private int index;
    private double[] speed;

    // initializing the GPS class requires a predefined array of doubles that stores a simulated train speed interval.
    // then, it sets 'index' to start at position 0, and sets the 'speed' according to the predefined train speed array.
    public GPS(double[] _speed) {
        this.index = 0;
        this.speed = _speed;
    }

    // resets 'index' back to position 0 of the 'speed' array.
    public void reset() {
        this.index = 0;
    }

    // returns the current train speed outputted by the GPS unit as determined by the simulation.
    // the 'index' will keep incrementing until it reaches the end of the 'speed' array or if it is reseted.
    public double getGPSSpeed() {
        double val = this.speed[index];
        if (index < this.speed.length - 1) {
            index++;
        }

        return val;
    }
}
```

6.1.2 Lidar Sensor Class

```java
// the purpose of the Lidar class is to output an object's distance in front of the train in feet to TSNR.
public class Lidar {
    // The Lidar class holds 2 attributes:
    // - an array of Doubles 'distance' that holds an object's distance from the train at each array index.
    //   this array enables easy simulation and modification of the Lidar sensor's distance scan output.
    // - an int 'index' that tracks the current array index as the Lidar sensor outputs the object's distance from the train.
    private int index;
    private Double[] distance;

    // initializing the Lidar class requires a predefined array of Doubles that stores a simulated object distance interval.
    // then, it sets 'index' to start at position 0, and sets the 'distance' according to the predefined object distance array.
    public Lidar(Double[] _distance) {
        this.index = 0;
        this.distance = _distance;
    }

    // resets 'index' back to position 0 of the 'distance' array.
    public void reset() {
        this.index = 0;
    }

    // returns the current object's distance from the train outputted by the Lidar sensor as determined by the simulation.
    // the 'index' will keep incrementing until it reaches the end of the 'distance' array or if it is reseted.
    public Double getLidarScan() {
        Double val = this.distance[index];
        if (index < this.distance.length - 1) {
            index++;
        }

        return val;
    }
}
```

6.1.3 Proximity Sensor Class

```java
// the purpose of the Proximity class is to output an object's direction relative to the train to TSNR.
public class Proximity {
    // the Direction enum specifies the direction in which the object in front of the train is heading:
    // - Direction.AWAY translates to the object moving away the train.
    // - Direction.STATIONARY translates to the object being stationary.
    // - Direction.TOWARDS translates to the object moving towards the train.
    public enum Direction {
        AWAY,
        STATIONARY,
        TOWARDS
    }

    // The Proximity class holds 2 attributes:
    // - an array of Directions  'dir' that holds the direction in which the object is heading at each array index.
    //   this array enables easy simulation and modification of the proximity sensor's object direction output.
    // - an int 'index' that tracks the current array index as the proximity sensor outputs the direction the object is heading.
    private int index;
    private Direction[] dir;

    // initializing the Proximity class requires a predefined array of Directions that stores a simulated object direction interval.
    // then, it sets 'index' to start at position 0, and sets the 'dir' according to the predefined object direction array.
    public Proximity(Direction[] _dir) {
        this.index = 0;
        this.dir = _dir;
    }

    // resets 'index' back to position 0 of the 'dir' array.
    public void reset() {
        this.index = 0;
    }

    // returns the current direction in which the object is heading outputted by the proximity sensor as determined by the simulation.
    // the 'index' will keep incrementing until it reaches the end of the 'dir' array or if it is reseted.
    public Direction getProximityScan() {
        Direction val = this.dir[index];
        if (index < this.dir.length - 1) {
            index++;
        }

        return val;
    }
}
```

6.1.4 Optical Sensor Class

```java
// the purpose of the Optical class is to output whether the crossing gates are in the opened position to TSNR.
public class Optical {
    // The Optical class holds 2 attributes:
    // - an array of booleans 'gatesOpen' that holds a crossing gate position at each array index.
    //   this array enables easy simulation and modification of the optical sensor's gate position output.
    // - an int 'index' that tracks the current array index as the optical sensor outputs the crossing gate position.
    private int index;
    private boolean[] gatesOpen;

    // initializing the Optical class requires a predefined array of booleans that stores a simulated crossing gate position interval.
    // then, it sets 'index' to start at position 0, and sets the 'gatesOpen' according to the predefined gate position array.
    public Optical(boolean[] _gatesOpen) {
        this.index = 0;
        this.gatesOpen = _gatesOpen;
    }

    // resets 'index' back to position 0 of the 'gatesOpen' array.
    public void reset() {
        this.index = 0;
    }

    // returns the crossing gate position outputted by the optical sensor as determined by the simulation.
    // the 'index' will keep incrementing until it reaches the end of the 'gatesOpen' array or if it is reseted.
    public boolean getOpticalScan() {
        boolean val = this.gatesOpen[index];
        if (index < this.gatesOpen.length - 1) {
            index++;
        }

        return val;
    }
}
```

6.1.5 Wheel RPM Class

```java
// the purpose of the RPM class is to output the wheel RPM to TSNR.
public class RPM {
    // The RPM class holds 2 attributes:
    // - an array of doubles 'rpm' that holds the wheel RPM at each array index.
    //   this array enables easy simulation and modification of the RPM sensor's wheel RPM output.
    // - an int 'index' that tracks the current array index as the RPM sensor outputs the wheel RPM.
    private int index;
    private double[] rpm;

    // initializing the RPM class requires a predefined array of doubles that stores a simulated wheel RPM interval.
    // then, it sets 'index' to start at position 0, and sets the 'rpm' according to the predefined wheel RPM array.
    public RPM(double[] _rpm) {
        this.index = 0;
        this.rpm = _rpm;
    }

    // resets 'index' back to position 0 of the 'rpm' array.
    public void reset() {
        this.index = 0;
    }

    // returns the current wheel RPM outputted by the RPM sensor as determined by the simulation.
    // the 'index' will keep incrementing until it reaches the end of the 'rpm' array or if it is reseted.
    public double getWheelRPM() {
        double val = this.rpm[index];
        if (index < this.rpm.length - 1) {
            index++;
        }

        return val;
    }
}
```

6.2 TSNR Class

6.2.1 Packet Constructors

```java
// the purpose of the Packet class is to centralize all sensor data into a single object.
public static class Packet {
    // the Packet class holds 9 attributes:
    // - a double 'speed' which is the current train speed outputted by the GPS unit.
    // - a Double 'objectDist' which is an object's distance from the front of the train outputted by the first Lidar sensor.
    // - a Direction 'objectDir' which is an object's direction relative to the train outputted by the proximity sensor.
    // - a Double 'crossingDist' which is a Railroad Crossing's distance from the front of the train outputted by the second Lidar sensor.
    // - a boolean 'gatesOpen' which is the crossing gate position outputted by the optical sensor.
    // - a double 'w1RPM' which is the wheel RPM outputted by the 1st RPM sensor.
    // - a double 'w2RPM' which is the wheel RPM outputted by the 2nd RPM sensor.
    // - a double 'w3RPM' which is the wheel RPM outputted by the 3rd RPM sensor.
    // - a double 'w4RPM' which is the wheel RPM outputted by the 4th RPM sensor.
    private double speed;
    private Double objectDist;
    private Direction objectDir;
    private Double crossingDist;
    private boolean gatesOpen;
    private double w1RPM;
    private double w2RPM;
    private double w3RPM;
    private double w4RPM;

    // initializing the Packet class requires the above 9 attributes to be specified.
    public Packet(double _speed, Double _objectDist, Direction _objectDir, Double _crossingDist, boolean _gatesOpen,
            double _w1RPM, double _w2RPM, double _w3RPM, double _w4RPM) {
        this.speed = _speed;
        this.objectDist = _objectDist;
        this.objectDir = _objectDir;
        this.crossingDist = _crossingDist;
        this.gatesOpen = _gatesOpen;
        this.w1RPM = _w1RPM;
        this.w2RPM = _w2RPM;
        this.w3RPM = _w3RPM;
        this.w4RPM = _w4RPM;
    }

    ...
}
```

6.2.2 Packet Getters

```
// the purpose of the Packet class is to centralize all sensor data into a single object.
public static class Packet {
    ...

    // returns the current train speed outputted by the GPS unit from the packet.
    public double getSpeed() {
        return this.speed;
    }

    // returns the object's distance from the front of the train outputted by the first Lidar sensor from the packet.
    public Double getObjectDist() {
        return this.objectDist;
    }

    //returns the object's direction relative to the train outputted by the proximity sensor from the packet.
    public Direction getObjectDir() {
        return this.objectDir;
    }

    // returns the Railroad Crossing's distance from the front of the train outputted by the second Lidar sensor from the packet.
    public Double getCrossingDist() {
        return this.crossingDist;
    }

    // returns the crossing gate position outputted by the optical sensor from the packet.
    public boolean getGatesOpen() {
        return this.gatesOpen;
    }

    // returns the wheel RPM outputted by the 1st RPM sensor from the packet.
    public double getW1RPM() {
        return this.w1RPM;
    }

    // returns the wheel RPM outputted by the 2nd RPM sensor from the packet.
    public double getW2RPM() {
        return this.w2RPM;
    }

    // returns the wheel RPM outputted by the 3rd RPM sensor from the packet.
    public double getW3RPM() {
        return this.w3RPM;
    }

    // returns the wheel RPM outputted by the 4th RPM sensor from the packet.
    public double getW4RPM() {
        return this.w4RPM;
    }
}
```

6.2.3 TSNR Constructors

```java
// the purpose of the TSNR class is to combine all sensor data into a single packet for IoT processing.
public class TSNR {

    // the purpose of the Packet class is to centralize all sensor data into a single object.
    public static class Packet {
        ...
    }

    // getting TSNR's packet requires input from the GPS unit, 2 Lidar sensors, proximity sensor, optical sensor, and 4 RPM sensors.
    // then, each sensor's data is combined into a single packet to be returned.
    public static Packet getPacket(
            GPS gps, Lidar objectLidar, Proximity proximity,
            Lidar crossingLidar, Optical optical,
            RPM RPM1, RPM RPM2, RPM RPM3, RPM RPM4) {

        return new Packet(
                gps.getGPSSpeed(), objectLidar.getLidarScan(), proximity.getProximityScan(),
                crossingLidar.getLidarScan(), optical.getOpticalScan(),
                RPM1.getWheelRPM(), RPM2.getWheelRPM(), RPM3.getWheelRPM(), RPM4.getWheelRPM());
    }
}
```

6.3 Testable Class

6.3.1 Constructors

```java
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    // the Testable class holds 10 attributes:
    // - an int 'steps' that holds how many packets should be created.
    // - an array of doubles 'speed' that holds a train speed at each array index.
    // - an array of Doubles 'objectDist' that holds an object's distance from the train at each array index.
    // - an array of Directions 'objectDir' that holds the direction in which the object is heading at each array index.
    // - an array of Doubles 'crossingDist' that holds a Railroad Crossing's distance from the train at each array index.
    // - an array of booleans 'gatesOpen' that holds a crossing gate position at each array index.
    // - an array of doubles 'w1RPM' that holds wheel 1's RPM at each array index.
    // - an array of doubles 'w2RPM' that holds wheel 2's RPM at each array index.
    // - an array of doubles 'w3RPM' that holds wheel 3's RPM at each array index.
    // - an array of doubles 'w4RPM' that holds wheel 4's RPM at each array index.
    private int steps;
    private double[] speed;
    private Double[] objectDist;
    private Direction[] objectDir;
    private Double[] crossingDist;
    private boolean[] gatesOpen;
    private double[] w1RPM;
    private double[] w2RPM;
    private double[] w3RPM;
    private double[] w4RPM;

    // initializing the Testable class requires a predefined step amount correlating to how many packets will be tested.
    // then each sensor data array will be initialized with 'step' number of indices.
    public Testable(int _steps) {
        this.steps = _steps;
        this.speed = new double[_steps];
        this.objectDist = new Double[_steps];
        this.objectDir = new Direction[_steps];
        this.crossingDist = new Double[_steps];
        this.gatesOpen = new boolean[_steps];
        this.w1RPM = new double[_steps];
        this.w2RPM = new double[_steps];
        this.w3RPM = new double[_steps];
        this.w4RPM = new double[_steps];
    }

    ...
}
```

6.3.2 Testing GPS Unit

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing GPS unit
    //
    ////////////////////////////////////////

    // sets up the simulation for the train speed outputted by the GPS unit with linear growth over the steps.
    // setup requires the train's starting speed and the train's final speed.
    public void setupSpeed(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.speed[i] = curr_val;
        }
    }

    // modifies the current train speed at a specified 'index' of the simulated train speed array.
    public void setSpeed(double val, int index) {
        this.speed[index] = val;
    }

    // returns the simulated train speed array.
    public double[] getSpeed() {
        return this.speed;
    }

    ...
}

```

6.3.3 Testing First Lidar Sensor

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing Lidar sensor for object distance
    //
    ////////////////////////////////////////

    // sets up the simulation for the object's distance from the train outputted by the first Lidar sensor with linear growth over the steps.
    // setup requires the object's starting distance from the train and the object's final distance from the train.
    public void setupObjectDist(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.objectDist[i] = curr_val;
        }
    }

    // modifies the current object's distance from the train at a specified 'index' of the simulated object distance array.
    public void setObjectDist(Double val, int index) {
        this.objectDist[index] = val;
    }

    // returns the simulated object distance array.
    public Double[] getObjectDist() {
        return this.objectDist;
    }

    ...
}

```

6.3.4 Testing Proximity Sensor

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing proximity sensor
    //
    ////////////////////////////////////////

    // sets up the simulation for the object's direction of heading outputted by the proximity sensor.
    // setup requires the object's direction of heading relative to the train.
    public void setupObjectDir(Direction val) {
        for (int i = 0; i < this.steps; i++) {
            this.objectDir[i] = val;
        }
    }

    // modifies the current object's direction from the train at a specified 'index' of the simulated object direction array.
    public void setObjectDir(Direction val, int index) {
        this.objectDir[index] = val;
    }

    // returns the simulated object direction array.
    public Direction[] getObjectDir() {
        return this.objectDir;
    }

    ...
}
```

6.3.5 Testing Second Lidar Sensor

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing Lidar sensor for crossing distance
    //
    ////////////////////////////////////////

    // sets up the simulation for the Railroad Crossing's distance from the train outputted by the second Lidar sensor with linear growth over
the steps.
    // setup requires the crossing's starting distance from the train and the crossing's final distance from the train.
    public void setupCrossingDist(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.crossingDist[i] = curr_val;
        }
    }

    // modifies the current Railroad Crossing's distance from the train at a specified 'index' of the simulated crossing distance array.
    public void setCrossingDist(Double val, int index) {
        this.crossingDist[index] = val;
    }

    // returns the simulated crossing distance array.
    public Double[] getCrossingDist() {
        return this.crossingDist;
    }

    ...
}
```

6.3.6 Testing Optical Sensor

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing optical sensor
    //
    ////////////////////////////////////////

    // sets up the simulation for the crossing gate position outputted by the optical sensor.
    // setup requires whether the Railroad Crossing gate's are in the opened position.
    public void setupGatesOpen(boolean val) {
        for (int i = 0; i < this.steps; i++) {
            this.gatesOpen[i] = val;
        }
    }

    // modifies the current crossing gate position at a specified 'index' of the simulated gate position array.
    public void setGatesOpen(boolean val, int index) {
        this.gatesOpen[index] = val;
    }

    // returns the simulated gate position array.
    public boolean[] getGatesOpen() {
        return this.gatesOpen;
    }

    ...
}
```

6.3.7 Testing RPM Sensor 1

```
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing RPM sensor 1
    //
    ////////////////////////////////////////

    // sets up the simulation for the wheel RPM outputted by the 1st RPM sensor with linear growth over the steps.
    // setup requires the wheel's starting RPM and the wheel's final RPM.
    public void setupW1RPM(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.w1RPM[i] = curr_val;
        }
    }

    // modifies the current RPM value at a specified 'index' of the simulated wheel 1 RPM array.
    public void setW1RPM(double val, int index) {
        this.w1RPM[index] = val;
    }

    // returns the simulated wheel 1 RPM array.
    public double[] getW1RPM() {
        return this.w1RPM;
    }

    ...
}
```

6.3.8 Testing RPM Sensor 2

```java
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing RPM sensor 2
    //
    ////////////////////////////////////////

    // sets up the simulation for the wheel RPM outputted by the 2nd RPM sensor with linear growth over the steps.
    // setup requires the wheel's starting RPM and the wheel's final RPM.
    public void setupW2RPM(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.w2RPM[i] = curr_val;
        }
    }

    // modifies the current RPM value at a specified 'index' of the simulated wheel 2 RPM array.
    public void setW2RPM(double val, int index) {
        this.w2RPM[index] = val;
    }

    // returns the simulated wheel 2 RPM array.
    public double[] getW2RPM() {
        return this.w2RPM;
    }

    ...
}

```

6.3.9 Testing RPM Sensor 3

```java
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing RPM sensor 3
    //
    ////////////////////////////////////////

    // sets up the simulation for the wheel RPM outputted by the 3rd RPM sensor with linear growth over the steps.
    // setup requires the wheel's starting RPM and the wheel's final RPM.
    public void setupW3RPM(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.w3RPM[i] = curr_val;
        }
    }

    // modifies the current RPM value at a specified 'index' of the simulated wheel 3 RPM array.
    public void setW3RPM(double val, int index) {
        this.w3RPM[index] = val;
    }

    // returns the simulated wheel 3 RPM array.
    public double[] getW3RPM() {
        return this.w3RPM;
    }

    ...
}

```

6.3.10 Testing RPM Sensor 4

```java
// the purpose of the Testable class is to manually edit and simulate specific sensor data inputs into IoT.
public class Testable {
    ...

    ////////////////////////////////////////
    //
    // Testing RPM sensor 4
    //
    ////////////////////////////////////////

    // sets up the simulation for the wheel RPM outputted by the 4th RPM sensor with linear growth over the steps.
    // setup requires the wheel's starting RPM and the wheel's final RPM.
    public void setupW4RPM(double start_val, double end_val) {
        double step = (end_val - start_val) / (this.steps - 1), curr_val = start_val;
        for (int i = 0; i < this.steps; i++, curr_val += step) {
            this.w4RPM[i] = curr_val;
        }
    }

    // modifies the current RPM value at a specified 'index' of the simulated wheel 4 RPM array.
    public void setW4RPM(double val, int index) {
        this.w4RPM[index] = val;
    }

    // returns the simulated wheel 4 RPM array.
    public double[] getW4RPM() {
        return this.w4RPM;
    }
}
```

6.4 Terminal Class

6.4.1 Constructors

```java
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    // customization attributes for each panel
    private final Font TITLE_FONT = new Font("Helvetica", Font.BOLD, 25);
    private final Font CONSOLE_FONT = new Font("Courier", Font.PLAIN, 18);
    private final Font ICON_TITLE_FONT = new Font("Helvetica", Font.BOLD, 18);
    private final Font ICON_TEXT_FONT = new Font("Helvetica", Font.BOLD, 18);
    private final Font PROMPT_FONT = new Font("Comic Sans MS", Font.PLAIN, 18);
    private final EmptyBorder PADDING = new EmptyBorder(10, 10, 10, 10);
    private final Color[] COLORS = {Color.WHITE, Color.RED, Color.ORANGE, Color.YELLOW, Color.GREEN, Color.BLUE, Color.MAGENTA, Color.BLACK};
    private String[] ICON_TITLES = {"OBJECT", "CROSSING", "HORN", "WHEEL SLIP", "SPEED"};

    // the Terminal class holds 8 attributes:
    // - a FileWriter 'logFile' that holds a pointer to the log file.
    // - a JTextPane "console_feed" that emulates a console environment.
    // - an array of JPanels "icon_square" that holds each icon for distance warnings, reduced speed recommendations, etc.
    // - an array of JLabels "icon_head" that holds the title text for each icon.
    // - an array of JLabels "icon_subhead" that holds the subtitle text for each icon.
    // - an array of JLabels "icon_feed" that holds the sensor data text for each icon.
    // - a JTextField "prompt_feed" that emulates a prompt input box.
    // - a String "inputField" that holds a command line argument entered from terminal prompt.
    private FileWriter logFile;
    private JTextPane console_feed;
    private JPanel[] icon_square;
    private JLabel[] icon_head;
    private JLabel[] icon_subhead;
    private JLabel[] icon_feed;
    private JTextField prompt_feed;
    private String inputField = "";

    // initializing the Terminal class has the following pseudocode:
    // 1. locate the log file directory.
    // 2. create a new log file in said directory.
    // 3. create a new JFrame for the terminal's application body.
    // 4. create the console panel by initializing 'console_feed'.
    // 5. create the icon panel by initializing 'icon_square', 'icon_head', 'icon_subhead', and 'icon_feed'.
    // 6. create the prompt panel by initializing 'prompt_feed'.
    // 7. add the console, icon, and prompt panels to the main JFrame.
    public Terminal() {
        ...
    }

    ...
}
```

6.4.2 Initialization Parts 1-3

```java
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // initializing the Terminal class has the following pseudocode:
    // 1. locate the log file directory.
    // 2. create a new log file in said directory.
    // 3. create a new JFrame for the terminal's application body.
    // 4. create the console panel by initializing 'console_feed'.
    // 5. create the icon panel by initializing 'icon_square', 'icon_head', 'icon_subhead', and 'icon_feed'.
    // 6. create the prompt panel by initializing 'prompt_feed'.
    // 7. add the console, icon, and prompt panels to the main JFrame.
    public Terminal() {
        // 1. locate the log file directory.
        // if the log file directory does not exist, create a new folder '/log' attached to the current directory of the application.
        String dir_name = System.getProperty("user.dir") + "\\logs";
        File dir = new File(dir_name);
        if (!dir.exists()) {
            dir.mkdir();
        }

        // 2. create a new log file in said directory.
        // the log file will be named with the current time and date.
        try {
            Date date = new Date();
            SimpleDateFormat time = new SimpleDateFormat("MM.dd.yyyy HH-mm-ss");
            logFile = new FileWriter(dir_name + "/" + time.format(date) + ".txt");
            updateLogger("Log file created on " + time.format(date) + ". Log messages are broken by (C): Console, (S): System, (U): User.",
false);
        } catch (IOException e1) {
            System.out.println("Error: Cannot create log file.");
            e1.printStackTrace();
        }

        // 3. create a new JFrame for the terminal's application body.
        // setup the dimensions of the Java application and window close behavior.
        JFrame frame = new JFrame("IoT HTR");
        frame.setSize(800, 650);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        ...
    }

    ...
}

```

6.4.3 Initialization Parts 4-5

```java
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // initializing the Terminal class has the following pseudocode:
    // 1. locate the log file directory.
    // 2. create a new log file in said directory.
    // 3. create a new JFrame for the terminal's application body.
    // 4. create the console panel by initializing 'console_feed'.
    // 5. create the icon panel by initializing 'icon_square', 'icon_head', 'icon_subhead', and 'icon_feed'.
    // 6. create the prompt panel by initializing 'prompt_feed'.
    // 7. add the console, icon, and prompt panels to the main JFrame.
    public Terminal() {
        ...

        // 4. create the console panel by initializing 'console_feed'.
        // setup the console panel with a black background and a vertical scroll bar.
        JPanel console = new JPanel();
        Dimension console_size = console.getPreferredSize();
        console_size.height = 300;
        console.setPreferredSize(console_size);
        TitledBorder console_title = BorderFactory.createTitledBorder(BorderFactory.createLineBorder(Color.BLACK), "Console");
        console_title.setTitleFont(TITLE_FONT);
        console.setBorder(new CompoundBorder(PADDING, console_title));
        console.setLayout(new BorderLayout());
        console_feed = new JTextPane();
        console_feed.setEditable(false);
        console_feed.setBackground(Color.BLACK);
        console_feed.setFont(CONSOLE_FONT);
        JScrollPane console_scroll = new JScrollPane(console_feed);
        console_scroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
        console_scroll.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        console.add(console_scroll);

        // 5. create the icon panel by initializing 'icon_square', 'icon_head', 'icon_subhead', and 'icon_feed'.
        JPanel center_icon = new JPanel();
        JPanel icon = new JPanel();
        TitledBorder icon_title = BorderFactory.createTitledBorder(BorderFactory.createLineBorder(Color.BLACK), "Icons");
        icon_title.setTitleFont(TITLE_FONT);
        icon.setBorder(new CompoundBorder(PADDING, icon_title));
        icon.setLayout(new GridLayout(0, ICON_TITLES.length));
        center_icon.add(icon);

        // initialize each panel with a title from the 'ICON_TITLES' array and hide the icon's visibility by default.
        icon_square = new JPanel[ICON_TITLES.length];
        icon_head = new JLabel[ICON_TITLES.length];
        icon_subhead = new JLabel[ICON_TITLES.length];
        icon_feed = new JLabel[ICON_TITLES.length];
        for (int i = 0; i < ICON_TITLES.length; i++) {
            JPanel icon_col = new JPanel();
            icon_square[i] = new JPanel();
            icon_square[i].setPreferredSize(new Dimension(125, 125));
            icon_square[i].setLayout(new BoxLayout(icon_square[i], BoxLayout.Y_AXIS));
            icon_square[i].setBackground(Color.WHITE);
            icon_square[i].setBorder(BorderFactory.createLineBorder(Color.BLACK, 4));
            icon_head[i] = new JLabel(ICON_TITLES[i]);
            icon_head[i].setFont(ICON_TITLE_FONT);
            icon_subhead[i] = new JLabel(String.valueOf(i));
            icon_subhead[i].setFont(ICON_TEXT_FONT);
            icon_feed[i] = new JLabel(String.valueOf(i));
            icon_feed[i].setFont(ICON_TEXT_FONT);
            icon_square[i].add(Box.createVerticalGlue());
            icon_square[i].add(icon_head[i]);
            icon_square[i].add(icon_subhead[i]);
            icon_square[i].add(Box.createVerticalGlue());
            icon_square[i].add(icon_feed[i]);
            icon_square[i].add(Box.createVerticalGlue());
            icon_col.add(icon_square[i]);
            icon.add(icon_col);
        }
        hideAllIcons();

        ...
    }

    ...
}
```

6.4.4 Initialization Parts 6-7

```
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // initializing the Terminal class has the following pseudocode:
    // 1. locate the log file directory.
    // 2. create a new log file in said directory.
    // 3. create a new JFrame for the terminal's application body.
    // 4. create the console panel by initializing 'console_feed'.
    // 5. create the icon panel by initializing 'icon_square', 'icon_head', 'icon_subhead', and 'icon_feed'.
    // 6. create the prompt panel by initializing 'prompt_feed'.
    // 7. add the console, icon, and prompt panels to the main JFrame.
    public Terminal(IoT iot) {
        ...

        // 6. create the prompt panel by initializing 'prompt_feed'.
        // add a listener to the prompt panel that pipes the user's request into 'inputField' whenever the user presses [ENTER] in the terminal prompt.
        JPanel prompt = new JPanel();
        Dimension prompt_size = prompt.getPreferredSize();
        prompt_size.height = 120;
        prompt.setPreferredSize(prompt_size);
        TitledBorder prompt_title = BorderFactory.createTitledBorder(BorderFactory.createLineBorder(Color.BLACK), "Prompt");
        prompt_title.setTitleFont(TITLE_FONT);
        prompt.setBorder(new CompoundBorder(PADDING, prompt_title));
        prompt.setLayout(new BorderLayout());
        prompt_feed = new JTextField();
        prompt_feed.setFont(PROMPT_FONT);
        prompt.add(prompt_feed);
        prompt_feed.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                inputField = prompt_feed.getText();
                prompt_feed.setText("");
            }
        });

        // 7. add the console, icon, and prompt panels to the main JFrame.
        // add a listener to the Java application so that when the application closes, the log file successfully saves and closes.
        Container c = frame.getContentPane();
        c.add(console, BorderLayout.NORTH);
        c.add(center_icon, BorderLayout.CENTER);
        c.add(prompt, BorderLayout.SOUTH);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                iot.invokeClose(true);

                try {
                    logFile.close();
                } catch (IOException e1) {
                    System.out.println("Error: Could not save & close log file.");
                    e1.printStackTrace();
                }

                System.exit(0);
            }
        });
    }

    ...
}

```

6.4.5 Printing Methods

```
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // returns the string 'msg' without any of the colored prefixes &[0-7].
    private String sanatize(String msg) {
        Pattern pattern = Pattern.compile("&[0-7]");
        Matcher matcher = pattern.matcher(msg);
        while (matcher.find()) {
            String str = msg.substring(matcher.start(), matcher.end());
            msg = msg.replace(str, "");
            matcher = pattern.matcher(msg);
        }

        return msg;
    }

    // prints 'msg' onto the terminal console. If 'includeColor' is enabled, then using a prefix &[0-7] changes the message's color.
    // if 'includeTime' is enabled, then 'msg' will be prepended by the current time.
    // all calls to printMsg() and their messages will be saved to the log file.
    public void printMsg(String msg, boolean includeColor, boolean includeTime) {
        // sanatize the message and save it to the log file.
        Date date = new Date();
        SimpleDateFormat time = new SimpleDateFormat("HH:mm:ss");
        updateLogger(sanatize(time.format(date) + " (" + (includeTime ? "U" : "C") + ") >> " + msg), true);

        // if 'includeTime' is enabled, then 'msg' will be prepended by the current time.
        if (includeTime) {
            msg = time.format(date) + " >> " + msg;
        }

        try {
            StyledDocument doc = console_feed.getStyledDocument();
            Style style = console_feed.addStyle("", null);

            // if 'includeColor' is enabled, then using a prefix &[0-7] changes the message's color.
            // else format the message in a white color.
            if (includeColor) {
                String[] parts = msg.split("(?=&[0-7])");
                Pattern pattern = Pattern.compile("&[0-7]");
                Matcher matcher = pattern.matcher(parts[0]);
                if (!matcher.find()) {
                    parts[0] = "&0" + parts[0];
                }

                for (int i = 0; i < parts.length; i++) {
                    int color = Character.getNumericValue(parts[i].charAt(1));
                    StyleConstants.setForeground(style, COLORS[color]);
                    doc.insertString(doc.getLength(), parts[i].substring(2, parts[i].length()), style);
                }

                StyleConstants.setForeground(style, COLORS[0]);
                doc.insertString(doc.getLength(), "\n", style);
            }
            else {
                StyleConstants.setForeground(style, COLORS[0]);
                doc.insertString(doc.getLength(), msg + "\n", style);
            }

            // display the message onto the terminal console.
            console_feed.setCaretPosition(console_feed.getDocument().getLength());
        }
        catch (BadLocationException e) {
            System.out.println("Error: Could not print message '" + msg + "'.");
            e.printStackTrace();
        }
    }

    // prints a newline onto the terminal console.
    public void printNewline() {
        printMsg("", false, false);
    }

    ...
}
```

6.4.6 Icon Methods

```
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // hides or displays the icon text at the specified icon index 'i'. Indices match with the 'ICON_TITLES' array.
    public void setIconVisibility(boolean isVisible, int i) {
        if (!isVisible) {
            setIconBGColor(0, i);
        }
        icon_head[i].setVisible(isVisible);
        icon_subhead[i].setVisible(isVisible);
        icon_feed[i].setVisible(isVisible);
    }

    // hides all icons from view.
    public void hideAllIcons() {
        for (int i = 0; i < ICON_TITLES.length; i++) {
            setIconVisibility(false, i);
        }
    }

    // sets the background color of the icon at the specified icon index 'i'. Indices match with the 'ICON_TITLES' array.
    // indices of colors can be viewed from the 'COLORS' array.
    public void setIconBGColor(int color_index, int i) {
        icon_square[i].setBackground(COLORS[color_index]);
    }

    // sets the color of the icon text at the specified icon index 'i'. Indices match with the 'ICON_TITLES' array.
    // indices of colors can be viewed from the 'COLORS' array.
    public void setIconTxtColor(int color_index, int i) {
        icon_head[i].setForeground(COLORS[color_index]);
        icon_subhead[i].setForeground(COLORS[color_index]);
        icon_feed[i].setForeground(COLORS[color_index]);
    }

    // replaces the icon text with 'msg' at the specified icon index 'i'. Indices match with the 'ICON_TITLES' array.
    public void setIconFeed(String subhead_msg, String feed_msg, int i) {
        icon_subhead[i].setText(subhead_msg);
        icon_feed[i].setText(feed_msg);
    }

    ...
}

```

6.4.7 Prompt and Logger Methods

```
// the purpose of the Terminal class is to connect the user to IoT via a console, icon, and prompt interface.
public class Terminal {
    ...

    // returns the input string after the user presses [ENTER] in the prompt window.
    // caution: the input string may not necessarily have been cleared using resetInputField(), so old inputs may still remain.
    public String getInputField() {
        return this.inputField;
    }

    // resets the input string.
    public void resetInputField() {
        this.inputField = "";
    }

    // writes 'msg' to a new line of the log file.
    public void updateLogger(String msg, boolean fromConsole) {
        try {
            if (fromConsole) {
                logFile.write(msg + "\n");
            }
            else {
                Date date = new Date();
                SimpleDateFormat time = new SimpleDateFormat("HH:mm:ss");
                logFile.write(time.format(date) + " (S) >> " + msg + "\n");
            }
        } catch (IOException e) {
            System.out.println("Error: Could not write '" + "' to the log file.");
            e.printStackTrace();
        }
    }
}

```

6.5 IoT Class

6.5.1 Profile Class

```java
// the purpose of the Profile class is to centralize all user information (username and password) into a single object.
public class Profile {
    // the Profile class holds 2 attributes:
    // - a String 'username' that holds the user's username.
    // - a String 'password' that holds the user's password.
    private String username;
    private String password;

    // initializing the Profile class requires the above 2 attributes to be specified.
    public Profile(String _username, String _password) {
        this.username = _username;
        this.password = _password;
    }

    // resets the username and password of the Profile to an empty string.
    public void reset() {
        this.username = "";
        this.password = "";
    }

    // getters and setters for username and password.
    public void setUsername(String _username) {
        this.username = _username;
    }

    public void setPassword(String _password) {
        this.password = _password;
    }

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

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

    // this override function enables the Profile class to work properly with a hashmap's key hash.
    @Override
    public int hashCode() {
        return 31 * username.hashCode() + password.hashCode();
    }

    // this override function enables the Profile class to determine if 2 profiles have the same username and password.
    @Override
    public boolean equals(Object obj) {
        // return false if the object is not a Profile
        if (!(obj instanceof Profile)) {
            return false;
        }

        // convert object to Profile
        Profile p = (Profile) obj;
        return this.username.equals(p.getUsername()) && this.password.equals(p.getPassword());
    }
}
```

6.5.2 IoT Constructors

```java
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    // the IoT class holds 3 attributes:
    // - an instance of the Terminal object 'terminal'.
    // - a Profile 'user' that holds the current user's username and password.
    // - an int "clearance" that holds the clearance level of the current user.
    // - a boolean 'closeInvoked' that holds the status of the application has been closed.
    private Terminal terminal;
    private Profile user;
    private int clearance;
    private boolean closeInvoked;

    // the IoT class holds 2 constants:
    // - a double 'WHEEL_DIAMETER_FT' that holds the average wheel diameter in feet.
    // - a Hashmap<Profile, Integer> 'USERS' that maps a user's profile to their clearance level.
    // the 'USERS' Hashmap emulates HTR's login database.
    private final double WHEEL_DIAMETER_FT = 4.166;
    private final Map<Profile, Integer> USERS = Map.of(
            new Profile("Bob", "testing"), 1,
            new Profile("John", "12345"), 2
            );

    // returns true if 'user' exists in HTR's login database.
    public boolean userExists(Profile user) {
        return USERS.containsKey(user);
    }

    // returns the clearance level of 'user'.
    public int getClearance(Profile user) {
        return USERS.get(user);
    }

    // initializing the IoT class creates a new instance of the Terminal, initiates the login process, and sets closeInvoked to false.
    public IoT() {
        this.terminal = new Terminal(this);
        this.user = userInit();
        this.closeInvoked = false;
    }

    // getter for the terminal.
    public Terminal getTerminal() {
        return this.terminal;
    }

    // returns whether the application has been closed.
    public boolean getCloseStatus() {
        return this.closeInvoked;
    }

    // sets the closeInvoked status.
    public void invokeClose(boolean bool) {
        this.closeInvoked = bool;
    }

    ...
}
```

6.5.3 IoT Login Initialization

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    // returns the profile of the user.
    // the terminal will repeatedly prompt the user for their username and password until the profile has been successfully validated using
HTR's login database.
    // initializing the user profile has the following pseudocode:
    // 1. create an empty profile and display IoT's welcome message onto the console.
    // 2. prompt the user for their username and store the input to inputField.
    // 3. prompt the user for their password and store the input to inputField.
    // 4. if the username does not have 3 to 16 alphanumeric characters, goto step 2.
    // 5. if the password does not have 5 to 25 alphanumeric characters, goto step 2.
    // 6. if the username and password are invalid in HTR's login database, goto step 2.
    // 7. if the username and password are valid in HTR's login database, display the clearance level of the user.
    // 8. return the profile of the user with the valid username and password.
    public Profile userInit() {
        // 1. create an empty profile and display IoT's welcome message onto the console.
        Profile profile = new Profile("", "");
        this.getTerminal().printMsg("#################################", true, false);
        this.getTerminal().printMsg("#   &4  __   ______   ______  ", true, false);
        this.getTerminal().printMsg("#   &4 /\\ \\ /\\  __ \\ /\\__  _\\ ", true, false);
        this.getTerminal().printMsg("#   &4 \\ \\ \\\\ \\ \\\\ \\ \\/\\ \\\\/_/\\ \\/ ", true, false);
        this.getTerminal().printMsg("#   &4  \\ \\_\\\\ \\_\\\\ \\_____\\  \\ \\_\\ ", true, false);
        this.getTerminal().printMsg("#   &4   \\/_/ \\/_____/   \\/_/ ", true, false);
        this.getTerminal().printMsg("#", true, false);
        this.getTerminal().printMsg("#################################", true, false);
        this.getTerminal().printNewline();

        while (true) {
            // 2. prompt the user for their username and store the input to inputField.
            profile.reset();
            this.getTerminal().resetInputField();
            this.getTerminal().printMsg("Please enter your username.", false, false);
            while (profile.getUsername().equals("")) {
                profile.setUsername(this.getTerminal().getInputField());
            }
            this.getTerminal().printMsg("Username = &3" + profile.getUsername(), true, true);

            // 3. prompt the user for their password and store the input to inputField.
            this.getTerminal().resetInputField();
            this.getTerminal().printMsg("Please enter your password.", true, false);
            while (profile.getPassword().equals("")) {
                profile.setPassword(this.getTerminal().getInputField());
            }
            this.getTerminal().resetInputField();
            this.getTerminal().printMsg("Password = &3" + profile.getPassword(), true, true);

            // 4. if the username does not have 3 to 16 alphanumeric characters, goto step 2.
            if (profile.getUsername().length() < 3 || profile.getUsername().length() > 16 || !profile.getUsername().matches("^[a-zA-Z0-9]*$"))
{
                this.getTerminal().printNewline();
                this.getTerminal().printMsg("&1Error&0: Username must be between 3 to 16 alphanumeric characters.", true, false);
                continue;
            }

            // 5. if the password does not have 5 to 25 alphanumeric characters, goto step 2.
            if (profile.getPassword().length() < 5 || profile.getPassword().length() > 25 || !profile.getPassword().matches("^[a-zA-Z0-9]*$"))
{
                this.getTerminal().printNewline();
                this.getTerminal().printMsg("&1Error&0: Password must be between 5 to 25 alphanumeric characters.", true, false);
                continue;
            }

            // 7. if the username and password are valid in HTR's login database, display the clearance level of the user.
            if (userExists(profile)) {
                this.clearance = getClearance(profile);
                this.getTerminal().printMsg("Welcome &4" + profile.getUsername() + "&0 to IoT. Your clearance level is: &4" + this.clearance +
"&0.", true, false);
                if (this.clearance == 1) {
                    this.getTerminal().printMsg("Level 1 commands:", false, false);
                    this.getTerminal().printMsg("- &4logger", true, false);
                }
                this.getTerminal().printMsg("Level 2 commands:", false, false);
                this.getTerminal().printMsg("- &4logout", true, false);
                this.getTerminal().printNewline();
                break;
            }

            // 6. if the username and password are invalid in HTR's login database, goto step 2.
            this.getTerminal().printNewline();
            this.getTerminal().printMsg("&1Error&0: Invalid username and password.", true, false);
        }

        // 8. return the profile of the user with the valid username and password.
        return profile;
    }

    // getter and setter for user profile.
    private Profile getUser() {
        return this.user;
    }

    private void setUser(Profile _user) {
        this.user = _user;
    }

    ...
}
```

6.5.4 IoT Test Environment

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    // returns a string of the double 'x' truncated to 2 decimal places.
    private static String trunc(double x) {
        double y = BigDecimal.valueOf(x).setScale(2, RoundingMode.HALF_UP).doubleValue();
        DecimalFormat df = new DecimalFormat("#.##");
        return df.format(y);
    }

    public static void main(String[] args) {
        // create new instance of IoT.
        IoT IoT = new IoT();

        // create a new test simulation environment by predefining the sensor data.
        Simulation test = new Simulation();

        // initialize the GPS unit, 2 Lidar sensors, proximity sensor, optical sensor, and 4 RPM sensors.
        // feed the test simulation data into each sensor.
        GPS gps = new GPS(test.getSpeed());
        Lidar objectLidar = new Lidar(test.getObjectDist());
        Proximity proximity = new Proximity(test.getObjectDir());
        Lidar crossingLidar = new Lidar(test.getCrossingDist());
        Optical optical = new Optical(test.getGatesOpen());
        RPM RPM1 = new RPM(test.getW1RPM());
        RPM RPM2 = new RPM(test.getW2RPM());
        RPM RPM3 = new RPM(test.getW3RPM());
        RPM RPM4 = new RPM(test.getW4RPM());

        // holds how many seconds left to display horn warning
        int horn_time = 0;
        int horn_dist = 0;

        ...
}

```

6.5.5 Input Parsing

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    public static void main(String[] args) {
        ...

        // infinitely loop to simulate TSNR sending a packet to IoT at every second.
        // exit the loop when the application closes.
        while (!IoT.getCloseStatus()) {
            // track the current time to test algorithm runtime.
            long startTime = System.nanoTime();

            // process command line inputs from the user.
            // there are currently 2 commands: "logger" with level 1 clearance, and "logout" with level 2 clearance.
            String input = IoT.getTerminal().getInputField();
            if (!input.equals("")) {
                IoT.getTerminal().printMsg(input, false, true);
                switch (input) {
                case "logger":
                    // display the directory containing the log files if the user has level 1 clearance.
                    if (IoT.clearance == 1) {
                        IoT.getTerminal().printMsg("Logger directory: '&4" + System.getProperty("user.dir") + "\\logs&0'.", true, false);
                    }
                    else {
                        IoT.getTerminal().printMsg("&1Error&0: You do not have permission to use this command.", true, false);
                    }
                    break;
                case "logout":
                    // hide all current icons and reset the test simulation back to index 0.
                    IoT.getTerminal().hideAllIcons();
                    gps.reset();
                    objectLidar.reset();
                    proximity.reset();
                    crossingLidar.reset();
                    optical.reset();
                    RPM1.reset();
                    RPM2.reset();
                    RPM3.reset();
                    RPM4.reset();

                    horn_time = 0;

                    // display a logout success message onto the console and reprompt the user for their username and password.
                    IoT.getTerminal().printMsg("Successfully logged out &4" + IoT.getUser().getUsername() + "&0.", true, true);
                    IoT.setUser(IoT.userInit());
                    break;
                default:
                    // error handler for invalid command.
                    IoT.getTerminal().printMsg("&1Error&0: Unknown command '" + input + "'.", true, false);
                }

                IoT.getTerminal().resetInputField();
            }

            ...
        }
    }
}

```

6.5.6 IoT Moving & Stationary Object Detection

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    public static void main(String[] args) {
        ...

        // infinitely loop to simulate TSNR sending a packet to IoT at every second.
        // exit the loop when the application closes.
        while (!IoT.getCloseStatus()) {
            ...

            // get the packet from TSNR and update the log with all sensor data.
            Packet packet = TSNR.getPacket(gps, objectLidar, proximity, crossingLidar, optical, RPM1, RPM2, RPM3, RPM4);
            IoT.getTerminal().updateLogger("Packet: ", false);
            IoT.getTerminal().updateLogger(" - Train speed: " + packet.getSpeed() + "mph", false);
            IoT.getTerminal().updateLogger(" - Object distance: " + (packet.getObjectDist() == null ? "No nearby object detected" :
packet.getObjectDist() + "ft"), false);
            IoT.getTerminal().updateLogger(" - Object direction: " + packet.getObjectDir(), false);
            IoT.getTerminal().updateLogger(" - Crossing distance: " + (packet.getCrossingDist() == null ? "No nearby Railroad Crossing
detected" : packet.getCrossingDist() + "ft"), false);
            IoT.getTerminal().updateLogger(" - Gate position: " + packet.getGatesOpen(), false);
            IoT.getTerminal().updateLogger(" - Wheel 1 RPM: " + packet.getW1RPM(), false);
            IoT.getTerminal().updateLogger(" - Wheel 2 RPM: " + packet.getW2RPM(), false);
            IoT.getTerminal().updateLogger(" - Wheel 3 RPM: " + packet.getW3RPM(), false);
            IoT.getTerminal().updateLogger(" - Wheel 4 RPM: " + packet.getW4RPM(), false);

            // holds minimum speed between the current train speed and speed recommendations
            double speed = packet.getSpeed();
            double min_speed = speed;

            // decrement horn timer
            if (horn_time > 0) {
                horn_time--;
            }

            // Object Detection:
            // object detection has the following pseudocode:
            // 1. if the object's distance from the train > 3 miles or if the object is moving away from the train, do nothing.
            // 2. if the object is within 2 to 3 miles of the train, display a yellow object distance warning.
            // 3. if the object is within 1 to 2 miles of the train:
            //    - display an orange object distance warning.
            //    - display an orange reduced speed recommendation of:
            //      - 30mph if the object is moving towards the train.
            //      - 40mph if the object is stationary.
            // 4. if the object is within 1 mile of the train:
            //    - display a red object distance warning.
            //    - display a red brake recommendation.
            // 5. log the warnings/recommendations.
            Double objectDistance = packet.getObjectDist();
            Direction objectDirection = packet.getObjectDir();
            if (objectDistance != null && objectDistance <= 3 * 5280 && objectDirection != Direction.AWAY) {
                IoT.getTerminal().setIconFeed((objectDirection == Direction.TOWARDS) ? "(Moving)" : "(Stationary)", trunc(objectDistance) +
"ft", 0);

                if (objectDistance <= 3 * 5280 && objectDistance > 2 * 5280) {
                    IoT.getTerminal().setIconTxtColor(7, 0);
                    IoT.getTerminal().setIconBGColor(3, 0);
                }
                else if (objectDistance <= 2 * 5280 && objectDistance > 1 * 5280) {
                    IoT.getTerminal().setIconTxtColor(7, 0);
                    IoT.getTerminal().setIconBGColor(2, 0);
                    min_speed = (objectDirection == Direction.TOWARDS) ? Math.min(30, min_speed) : Math.min(40, min_speed);
                }
                else if (objectDistance <= 1 * 5280) {
                    IoT.getTerminal().setIconTxtColor(0, 0);
                    IoT.getTerminal().setIconBGColor(1, 0);
                    min_speed = Math.min(0, min_speed);
                }

                IoT.getTerminal().setIconVisibility(true, 0);
                IoT.getTerminal().updateLogger(((objectDirection == Direction.TOWARDS) ? "MOVING" : "STATIONARY") + " OBJECT DISTANCE WARNING:
" + objectDistance + "ft", false);
            }
            else {
                IoT.getTerminal().setIconVisibility(false, 0);
            }

            ...
        }
    }
}
```

6.5.7 IoT Railroad Crossing Detection

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    public static void main(String[] args) {
        ...

        // infinitely loop to simulate TSNR sending a packet to IoT at every second.
        // exit the loop when the application closes.
        while (!IoT.getCloseStatus()) {
            ...

            // Railroad Crossing Detection:
            // Railroad Crossing detection has the following pseudocode:
            // 1. if the crossing's distance from the train > 3 miles, do nothing.
            // 2. if the crossing is within 2 to 3 miles of the train:
            //    - display a yellow crossing distance warning.
            //    - display a yellow horn warning for 15 seconds.
            // 3. if the crossing is within 2 miles of the train:
            //    - if the crossing gates are closed:
            //      - display an orange crossing distance warning.
            //      - display an orange reduced speed recommendation of 30mph.
            //    - if the crossing gates are opened:
            //      - display a red crossing distance warning.
            //      - display a red brake recommendation.
            // 4. if the crossing is within 1 mile of the train, display a yellow horn warning for 5 seconds.
            // 5. log the warnings/recommendations.
            Double crossingDistance = packet.getCrossingDist();
            boolean gatesOpened = packet.getGatesOpen();
            if (crossingDistance != null && crossingDistance <= 3 * 5280) {
                IoT.getTerminal().setIconFeed(gatesOpened ? "(Open)" : "(Closed)", trunc(crossingDistance) + "ft", 1);

                if (crossingDistance <= 3 * 5280 && crossingDistance > 2 * 5280) {
                    IoT.getTerminal().setIconTxtColor(7, 1);
                    IoT.getTerminal().setIconBGColor(3, 1);

                    if (horn_dist != 3) {
                        horn_time = 15;
                        horn_dist = 3;
                    }
                }
                else if (crossingDistance <= 2 * 5280) {
                    if (gatesOpened) {
                        IoT.getTerminal().setIconTxtColor(0, 1);
                        IoT.getTerminal().setIconBGColor(1, 1);
                        min_speed = Math.min(0, min_speed);
                    }
                    else {
                        IoT.getTerminal().setIconTxtColor(7, 1);
                        IoT.getTerminal().setIconBGColor(2, 1);
                        min_speed = Math.min(40, min_speed);
                    }

                    if (crossingDistance <= 1 * 5280 && horn_dist != 1) {
                        horn_time = 5;
                        horn_dist = 1;
                    }
                }

                IoT.getTerminal().setIconVisibility(true, 1);
                IoT.getTerminal().updateLogger((gatesOpened ? "OPENED" : "CLOSED") + " RAILROAD CROSSING DISTANCE WARNING: " + crossingDistance
+ "ft", false);
            }
            else {
                IoT.getTerminal().setIconVisibility(false, 1);
            }

            // make sure to only display the yellow horn warning when the horn time is positive.
            if (horn_time > 0) {
                IoT.getTerminal().setIconFeed("", horn_time + "s", 2);
                IoT.getTerminal().setIconTxtColor(7, 2);
                IoT.getTerminal().setIconBGColor(3, 2);
                IoT.getTerminal().updateLogger("HORN WARNING: " + horn_time + "s", false);
                IoT.getTerminal().setIconVisibility(true, 2);
            }
            else {
                IoT.getTerminal().setIconVisibility(false, 2);
            }

            ...
        }
    }
}

```

6.5.8 IoT Wheel Slip Detection

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    public static void main(String[] args) {
        ...

        // infinitely loop to simulate TSNR sending a packet to IoT at every second.
        while (true) {
            ...

            // Wheel Slip Detection:
            // wheel slip detection has the following pseudocode:
            // 1. take the average of the 4 wheel RPMs using (RPM1 + RPM2 + RPM3 + RPM4) / 4.
            // 2. convert the average wheel RPM to wheel speed using wheel RPM * pi * wheel diameter in ft * (1 mile / 5280ft) * (60 min / 1hr).
            // 3. calculate the absolute difference between the wheel speed and the current train speed.
            // 4. if the difference >= 5mph:
            //    - display an yellow wheel slip distance warning.
            //    - display an orange reduced speed recommendation of 40mph.
            // 5. log the warnings/recommendations.
            double w1RPM = packet.getW1RPM();
            double w2RPM = packet.getW2RPM();
            double w3RPM = packet.getW3RPM();
            double w4RPM = packet.getW4RPM();

            double avgRPM = (w1RPM + w2RPM + w3RPM + w4RPM) / 4;
            double avgWheelSpeed = avgRPM * Math.PI * IoT.WHEEL_DIAMETER_FT / 5280 * 60;
            double difference = Math.abs(speed - avgWheelSpeed);

            IoT.getTerminal().updateLogger("Average wheel RPM = " + avgRPM, false);
            IoT.getTerminal().updateLogger("Average wheel speed = " + avgWheelSpeed + "mph", false);

            if (difference >= 5) {
                IoT.getTerminal().setIconFeed("", "", 3);
                    IoT.getTerminal().setIconTxtColor(7, 3);
                    IoT.getTerminal().setIconBGColor(3, 3);
                    IoT.getTerminal().setIconVisibility(true, 3);
                    min_speed = Math.min(40, min_speed);
                IoT.getTerminal().updateLogger("WHEEL SLIP WARNING: " + difference + "mph difference", false);
            }
            else {
                IoT.getTerminal().setIconVisibility(false, 3);
            }

            ...
        }
    }
}
```

6.5.9 IoT Speed Recommendation Icon

```
// the purpose of the IoT class is to generate warnings/recommendations based on sensor data at every second.
public class IoT {
    ...

    public static void main(String[] args) {
        ...

        // infinitely loop to simulate TSNR sending a packet to IoT at every second.
        // exit the loop when the application closes.
        while (!IoT.getCloseStatus()) {
            ...

            // make sure to only display the lowest orange reduced speed recommendation.
            // the reduced speed recommendation should also be less than the current train speed.
            if (min_speed < speed) {
                if (min_speed == 0) {
                    // a speed of 0mph is just a red brake recommendation.
                    IoT.getTerminal().setIconFeed("", "BRAKE", 4);
                    IoT.getTerminal().setIconTxtColor(0, 4);
                    IoT.getTerminal().setIconBGColor(1, 4);
                    IoT.getTerminal().updateLogger("EMERGENCY BRAKE RECOMMENDATION", false);
                }
                else {
                    // else orange reduced speed recommendation.
                    IoT.getTerminal().setIconFeed("", min_speed + "mph", 4);
                    IoT.getTerminal().setIconTxtColor(7, 4);
                    IoT.getTerminal().setIconBGColor(2, 4);
                    IoT.getTerminal().updateLogger("REDUCED SPEED RECOMMENDATION: " + min_speed + "mph", false);
                }

                IoT.getTerminal().setIconVisibility(true, 4);
            }
            else {
                IoT.getTerminal().setIconVisibility(false, 4);
            }

            // calculate total execution time.
            long endTime = System.nanoTime();
            IoT.getTerminal().updateLogger("Total execution time: " + (((double) endTime - startTime) / 1000000000) + "s", false);

            // sleep for 1 second to simulate waiting for the next TSNR packet a second later.
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Error: Cannot sleep for 1 second.");
                e.printStackTrace();
            }
        }
    }
}
```

7) SECTION SEVEN

7.1 Validation Testing

7.1.1 Security

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R1	All accessible software interfaces and sensor modules connected to IoT must be locked behind a username and password system.	**X**		
R2	The HTR Network shall be secured by the LoRaWan protocol.		**X**	Implementing the LoRaWan protocol is not possible with Java.
R3	Usernames shall only contain alphanumeric characters and must have a length from 3 to 16.	**X**		
R4	Passwords shall only contain alphanumeric characters and must have a length from 5 to 25.	**X**		
R5	IoT's login system must have 2 levels of clearance.	**X**		
R6	Level 2 access shall be granted to HTR train operators who need IoT to drive and control LCS.	**X**		Commands: "logout"

R7	Level 1 access shall be restricted to HTR admins or senior HTR engineers who need direct access to IoT's software and sensors.	**X**		Commands: "logger", "logout"
R8	Level 1 users must have access to all modules accessible by level 2 users.	**X**		

7.1.2 Terminal

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R9	IoT's terminal screen must turn on when the train starts up.	**X**		
R10	IoT's terminal shall consist of 3 panels: a console panel, an icon panel, and a prompt panel.	**X**		Not a specified requirement, but the terminal tears when resized to a small window.
R11	The console panel shall display and handle messaging, errors, and login info.	**X**		
R12	The console panel must handle color formatted text and display timestamps when necessary.	**X**		
R13	The icon panel shall display any real-time distance warnings, reduced speed recommendations, and brake	**X**		

	recommendations as specified in section 3.2.			
R14	IoT shall display general warnings in a yellow color on the icon panel unless specified in another color.	**X**		
R15	IoT shall display reduced speed recommendations in an orange color on the icon panel.	**X**		
R16	IoT shall only display the lowest reduced speed recommendation in the icon panel.	**X**		
R17	IoT shall not display a reduced speed recommendation when the train's current speed is below the reduced speed recommendation.	**X**		
R18	IoT shall display brake recommendationS in a red color on the icon panel.	**X**		
R19	IoT shall display the icons in a white color when there are no warnings or recommendations.	**X**		
R20	The prompt panel shall process command line inputs entered by level 1 and level 2 users.	**X**		
R21	When IoT boots up, it must first prompt the user to enter their username and password into the terminal.	**X**		

R22	When successfully logged on, IoT must identify the clearance level of the user.	**X**		
R23	IoT must display a permission error message onto the terminal console if the user does not have the correct clearance level.	**X**		
R24	IoT must display an error message onto the terminal console if the prompted request does not exist.	**X**		
R25	Level 1 and level 2 users shall be able to log out of IoT from the terminal.	**X**		
R26	IoT must prompt the user to enter their username and password into the terminal after a successful logout.	**X**		
R27	Level 1 users shall be able to download a log file of all sensor outputs, IoT's outputs, generated recommendations, and any necessary calculations through a specified directory.	**X**		The log files are contained in a single directory that changes depending on where the application is located in.
R28	IoT and the terminal shall remain on until the train is turned off.	**X**		

7.1.3 Performance

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R29	When IoT receives input data from a sensor, there must be at most 1 second of computation time to generate a recommendation for the train operator.	**X**		Average computation time is about 0.005 seconds.
R30	When multiple sensors are running at the same, IoT must be able to handle at least 500 sensor requests per second.		**X**	IoT currently has 5 sensors installed.
R31	If IoT cannot handle requirements **R29** and **R30**, then IoT must increment its underperformance count.	**X**		

7.1.4 Reliability

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R32	IoT's performance to underperformance ratio must be above 99.5% for it to be considered reliable.	**X**		Ratio is 100%.

R33	IoT must only have 1 underperformance for every 1000 sensor inputs it processes.	**X**		No underperformance issues so far.
R34	When IoT encounters an underperformance, the specified problem shall be saved in the log file.	**X**		

7.1.5 Time Sensitive Networking Router

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R35	At every second, the GPS unit shall give TSNR the current speed of the train in mph as a double.	**X**		
R36	At every second, the first Lidar sensor shall give TSNR a double of the distance from the train to the closest object in feet.	**X**		
R37	At every second, the proximity sensor shall tell TSNR whether the closest object is stationary, moving towards the train, or moving away from the train.	**X**		
R38	At every second, the second Lidar sensor shall give TSNR a double of the distance from the train to the closest Railroad Crossing in feet.	**X**		

R39	At every second, the optical sensor shall give TSNR a boolean if the Railroad Crossing gates are opened or closed.	**X**		
R40	At every second, 4 RPM sensors shall give TSNR their respective wheel RPMs as a double.	**X**		
R41	TSNR must combine the collected data from **R35** through **R40** into a single packet.	**X**		
R42	At every second, TSNR must send the combined packet to IoT.	**X**		

7.1.6 Moving Object Detection

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R43	IoT must retrieve the distance and direction of the nearest moving object from TSNR's packet.	**X**		
R44	IoT shall do nothing when the distance is NULL, as it signifies no moving object nearby.	**X**		
R45	IoT shall also do nothing when the object is moving away from the train.	**X**		

R46	IoT shall execute **R47** through **R52** when the object is moving towards the train.	**X**		
R47	IoT must display an object warning icon with the object's distance from the train in feet when the moving object is within 3 miles of the train.	**X**		
R48	IoT shall display the object warning icon in yellow when the moving object is within 2 to 3 miles of the train.	**X**		
R49	IoT shall display the object warning icon in orange when the moving object is within 1 to 2 miles of the train.	**X**		
R50	IoT shall display the object warning icon in red when the moving object is within 1 mile of the train.	**X**		
R51	IoT must display an orange speed reduction icon with a recommended train speed of 30mph when the moving object is within 1 to 2 miles of the train.	**X**		
R52	IoT must display a red emergency brake icon when the moving object is within 1 mile of the train.	**X**		

7.1.7 Stationary Object Detection

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R53	IoT must retrieve the distance and direction of the nearest moving object from TSNR's packet.	**X**		
R54	IoT shall do nothing when the distance is NULL, as it signifies no stationary object nearby.	**X**		
R55	IoT shall execute **R56** through **R61** when the object is stationary.	**X**		
R56	IoT must display an object warning icon with the object's distance from the train in feet when the stationary object is within 3 miles of the train.	**X**		
R57	IoT shall display the object warning icon in yellow when the stationary object is within 2 to 3 miles of the train.	**X**		
R58	IoT shall display the object warning icon in orange when the stationary object is within 1 to 2 miles of the train.	**X**		

R59	IoT shall display the object warning icon in red when the stationary object is within 1 mile of the train.	**X**		
R60	IoT must display an orange speed reduction icon with a recommended train speed of 40mph when the stationary object is within 1 to 2 miles of the train.	**X**		
R61	IoT must display a red emergency brake icon when the stationary object is within 1 mile of the train.	**X**		

7.1.8 Railroad Crossing Detection

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R62	IoT must retrieve the distance and crossing gate position of the nearest Railroad Crossing from TSNR's packet.	**X**		
R63	IoT shall do nothing when the distance is NULL, as it signifies no Railroad Crossing nearby.	**X**		
R64	IoT shall display a yellow horn warning for 15 seconds when the Railroad Crossing is within 3 miles of the train.	**X**		

R65	IoT must also display a Railroad Crossing warning icon with the crossing's distance from the train in feet when the Railroad Crossing is within 3 miles of the train.	**X**		
R66	IoT shall display the Railroad Crossing warning icon in yellow when the Railroad Crossing is within 2 to 3 miles of the train.	**X**		
R67	IoT shall display the Railroad Crossing warning icon in orange when the crossing gates are in the closed position and are within 2 miles of the train.	**X**		
R68	IoT shall display the Railroad Crossing warning icon in red when the crossing gates are in the opened position and are within 2 miles of the train.	**X**		
R69	IoT must display an orange speed reduction icon with a recommended train speed of 40mph when the crossing gates are in the closed position and are within 2 miles of the train.	**X**		
R70	IoT must display a red emergency brake icon when the crossing gates are in the opened position and are within 2 miles of the train.	**X**		
R71	IoT shall display a yellow horn warning for 5 seconds when the Railroad Crossing is within 1 mile of the train.	**X**		

7.1.9 Wheel Slip Detection

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R72	IoT must retrieve the 4 wheel RPMs and current train speed from TSNR's packet.	**X**		
R73	IoT must calculate the average wheel RPM using the 4 wheel RPMs.	**X**		
R74	IoT shall then calculate the average wheel speed using the diameter of the wheel in feet and the average wheel RPM.	**X**		
R75	IoT must display a yellow wheel slip warning icon when there is more than a 5mph difference between the average wheel speed and the train's current speed.	**X**		
R76	IoT must display an orange speed reduction icon with a recommended train speed of 40mph when there is more than a 5mph difference between the average wheel speed and the train's current speed.	**X**		

7.1.10 Operating System

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R77	IoT's operating system must be Linux.		**X**	Java cannot emulate a Linux environment.
R78	IoT's terminal must run as a Linux-based command line terminal.		**X**	Java cannot emulate a Linux environment.

7.1.11 Hardware Architecture

Requirement Number	Requirement Description	Passed by Simulation	Not Testable	Additional Notes
R79	IoT must handle at least 1000 sensors.		**X**	IoT currently has 5 sensors installed.
R80	IoT must support 5 terabytes of data storage everyday.		**X**	The Java application does not allow for such a high storage amount.

7.2 Scenario-Based Testing

7.2.1 Use Case 1: Activation of IoT

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To boot up and initialize IoT.	**X**		Reproduce by running the Java IoT application. The application emulates the terminal screen turning on.

7.2.2 Use Case 2: Access to the Main Terminal

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To access IoT's output and input requests via the main terminal.	**X**		Reproduce by entering a valid level 1 account with username "Bob" and password "testing". Then, the command line argument "logger" should execute successfully from the terminal prompt.

7.2.3 Use Case 3: Moving Object Detection

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To generate a recommendation if an object is moving towards the train.	X		Reproduce by changing the Testable object arguments. Set the object distance to run from 26400 feet to 0 feet, and the object direction to always be towards the train. Then, IoT should display a red distance warning and a red brake warning at the 0 feet mark.

7.2.4 Use Case 4: Stationary Object Detection

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To generate a recommendation if an object is stationary in front the train.	X		Reproduce by changing the Testable object arguments. Set the object distance to run from 26400 feet to 0 feet, and the object direction to always be stationary. Then, IoT should display a red distance warning and a red brake warning at the 0 feet mark.

7.2.5 Use Case 5: Railroad Crossing Detection

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To generate a recommendation if a Railroad Crossing gate is in the opened position.	X		Reproduce by changing the Testable object arguments. Set the Railroad Crossing distance to run from 26400 feet to 0 feet, and the crossing gate position to be always open. Then, IoT should display a red distance warning and a red brake warning at the 0 feet mark.

7.2.6 Use Case 6: Wheel Slip Detection

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To generate a recommendation if there is a disagreement between the wheel's average speed and the current speed of the train.	X		Reproduce by changing the Testable object arguments. Set the train speed to always be at 60mph and modify the wheel RPMs so that the wheel speed computes to 70mph. Then, IoT should display a yellow wheel slip warning and an orange reduced speed recommendation.

7.2.7 Use Case 7: Access to Log Files

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To download IoT's log files from the terminal.	**X**		Reproduce by entering a valid level 1 account with username "Bob" and password "testing". Then, executing the command line argument "logger" should display the directory name containing all the log files. When going to that directory, all the timed log files should be there.

7.2.8 Use Case 8: Deactivation of IoT

Use Case Goal	Passed by Simulation	Not Testable	Additional Notes
To shutdown IoT.	**X**		Reproduce by pressing [x] on the Java IoT application. The application window closing emulates the terminal screen turning off.