Stevens Institute of Technology

TrainX Architects
IoT Hug the Rail
v.5.0

Bonnie Nguyen, Roma Razdan, Matthew Viafora, and Michael Zylka
Team 3 Section D
Software Development Process
Professor Peyrovian

Table of Contents

Section 1:

Introduction:

Using the internet of things, our group will devise a method to allow decisions to be made locally in absence or failure of cellular and wifi. This will be implemented into the Train System (Hug the Rail) as a means of making it safer, less costly, and more efficient.

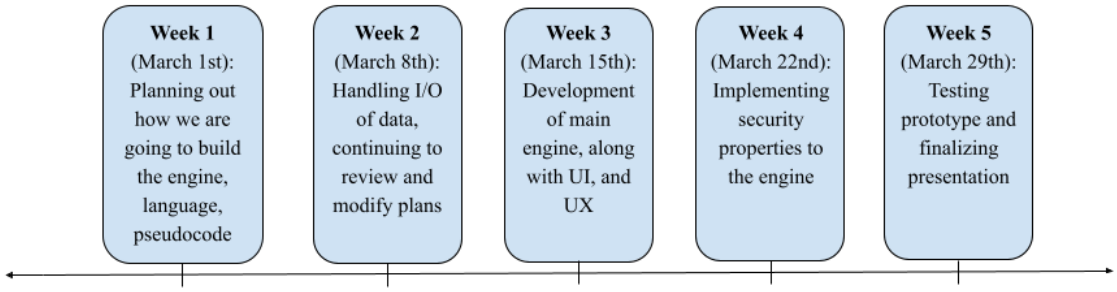
Our team consists of several smart, problem-solving, communicative individuals who strive to create an innovative design. Mike is a creative individual who will take the proper initiative to learn about the best possible solution for our project. Matt is a member who is very detail oriented and will ensure that the work we produce is truly working to create a safer, less costly, and more efficient solution. Bonnie is someone who is very task oriented and will help the overall flow of the project move smoothly, and Roma is someone who is people-oriented and will ensure that this model will be user-friendly and meet all the stakeholders' criteria.

Together this team, otherwise known as TrainX Architects, will work together to revolutionize the future of railroads as we know it today. We are looking to innovate the current solution, as well as looking for new ways to expedite the user's experience while maintaining a connection to our design.

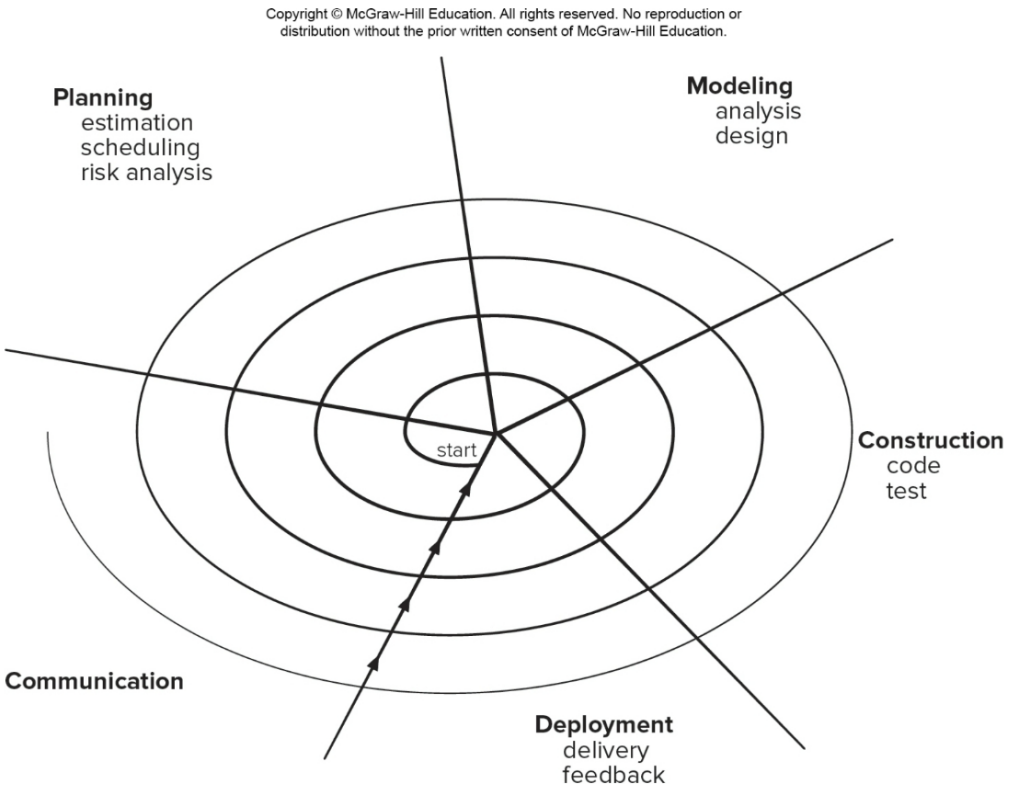
Our personal perspective on this project is to produce a system (engine) that informs the user of the train about current conditions, then to have an interface that allows the conductor to manipulate the train speeds according to that information. While this is rather general, we will work to pose a more detailed solution to our problem as we move through the planning process that this project will entail.

We see ourselves working to continue planning, then soon to begin modeling, construction, and deployment. After that we will reevaluate and communicate with our stakeholders and our group to ensure we are meeting their expectations. We have yet to create deliverables where we set due dates for each aspect of our project but we look forward to making more progress in reaching our goals.

Timeline:



Model:



(Spiral Process Model)

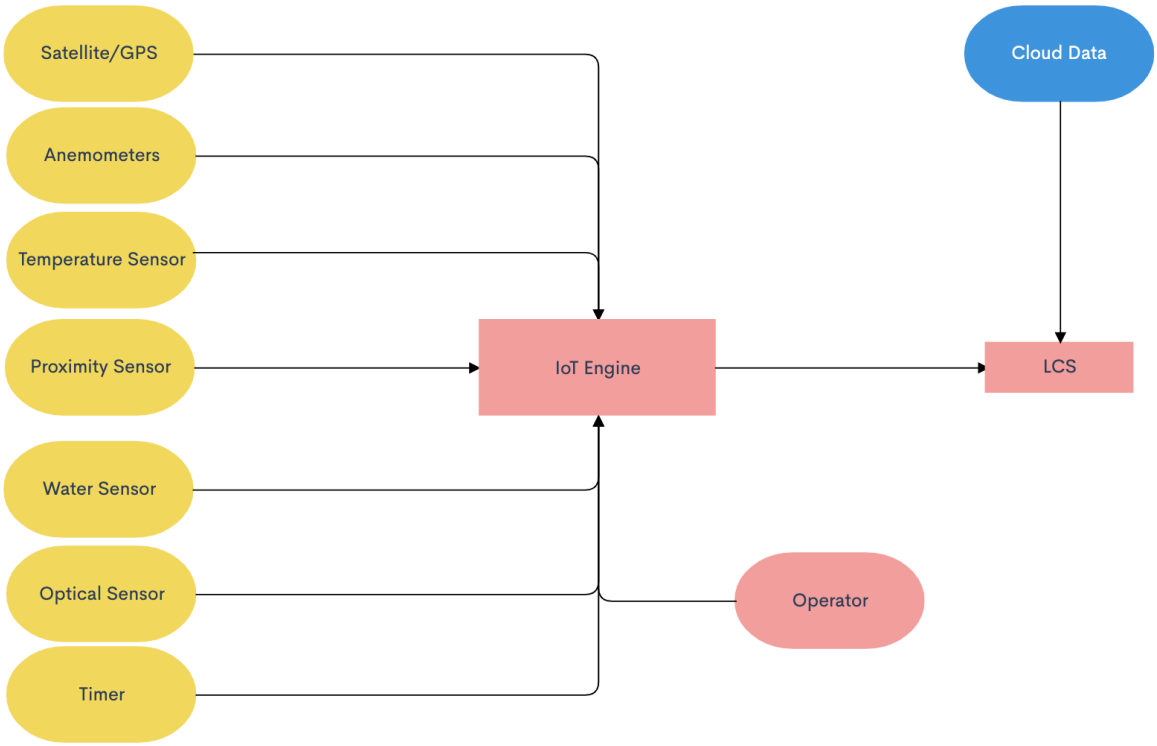
Pros	Cons
● Continuous customer involvement ● Development risks are managed ● Suitable for large, complex projects ● Allows to change ● Works well for extensible products	● Risk analysis failures can doom the project ● Project may be hard to manage ● Requires an expect development team

Section 2:

Problem Statement:

The train operation depends on live data received from the Central Operation Center Servers via WiFi/Cellular network. We need to be able to operate the train when we lose WiFi/Celluar network. We need to develop a system to allow us to continue the trip safely based on local data that we can collect. We assume GPS data is available.

System Overview



- Using installed sensors and computers outlined in the flowchart above, the IoT engine will take in different inputs from the data collected by the external sensors.
- The operator also has input into the IoT engine as they have power to adjust any part of the IoT engine including the speed of the train. The operator may have access to information that is not collected by the external sensors via radio, visual, etc.
- The IoT engine utilizes the data collected from these different sensors and/or if the operator influences anything and dynamically adjusts the throttle to brake or speed up appropriately.
- The LCS also has access to cloud data previously collected before the train lost connection, any previous data of value including arrival and departure times can be used until connection is regained.

Data Required

- Weather condition (such as snow, ice, rain, and wind)
- Hazards on the track both front and back
- Gate opening and closing times
- Arrival and departure times

Technology Required

- Temperature
    - Temperature sensor will measure the temperature outside and work with the optical sensor to determine an ideal speed for the train while not connected.
- Optical sensor
    - Ensure that the train does not start “slipping” on tracks and the locomotive is kept in control the entire time.
- Proximity sensor
    - In order to detect hazards on the railway both in front and in back of the train, there will be proximity sensors placed on the train that will work with IoT in order to determine if a change in the locomotive speed should occur.
- Water Sensor
    - Water sensors should be placed in strategic locations on the train in order to properly monitor weather conditions outside. Depending on the severity of the weather conditions, the water sensor should interface with the IoT to properly determine the ideal travel speed for the train.
- Timer
    - In order to properly arrive at gates on time and be prepared to stop in case of a scheduled downed gate or terminal, all arrival and departure times should be downloaded before losing connection. The IoT will work with this pre-downloaded data in order to determine the proper speed to arrive on time and not too early or too late.
- Anemometers
    - In case of harsh weather conditions, the locomotive should be prepared to travel with caution. An anemometer to measure wind speeds should be programmed to collaborate with the IoT architecture to determine if the train should travel at a cautious speed.
- Satellite Technology/GPS
    - Rely on GPS technology to determine the current position and direction of the train. Data can be drawn from the GPS to ensure the train stays on its intended route and not conflict with other trains in the area.
    - GPS also provides data on the speed of the train to interface with the other sensors and IoT in order to accommodate for outside conditions.

Section 3:

Requirements

3.1 Non-Functional Requirements

3.1.1: IoT HTR shall only be accessed via Operator ID and Password.

- We have to make sure that our information can only be assessed by a secure ID and password to ensure that we are to have complete control over our project.

3.1.2:IoT HTR Admin shall have secured (Admin ID/Pwd) to all sensors and equipment.

- Additionally, we have to ensure that our ID and password also are incorporated into the sensors and equipment to ensure the same security for our devices. This will allow our whole system to work without a fail rate.

3.1.3: IoT HTR Network shall be secured by LoRaWan protocol

- We will use a lower power wide area connection which will be wireless which provides single hop links between our server and the engine allowing for secure connections.

3.1.4: IoT HTR shall process an event within 0.5 seconds.

- Between 0.1 second and 1 second is a comfortable range in system response time where the user feels as though the system is responding without interrupting flow of thought, so in order to maintain the usability of the product, the sensors as well as the system itself must provide 0.5 seconds or less of processing time.

3.1.5: IoT HTR shall process 1000 events per second without degrading service.

- The scalability of the system must still process an event within 0.5 even when processing 1000 events/second, including responding to the user and working simultaneously with other processes in the system over a WiFi/LTE connection.

3.1.6: IoT HTR shall operate with no failure 99.99% of the time.

- In order to ensure the safety of the operators and reliability of the IoT engine, extensive testing will be required until there are little to no failures whatsoever.
- The reliability of the IoT engine is mission critical to the safety and security of TrainX, its customers and the environment traveled.

3.2 Functional Requirements

3.2.1 Detect standing objects on the path of the train with distance, speed and suggestion to Conductor on braking or decreasing/increasing speed

- Proximity sensors shall be installed on both the front and rear of the train
- Proximity sensors report constant updates on standing hazards on the track to the IoT Engine
- IoT Engine makes needed calculations to recommend amount of braking required to possibly make an emergency stop

3.2.2 Detect moving objects ahead or behind and their speed, distance, with suggestion of brake, increase/decrease speed

- Proximity sensors report speed of moving objects ahead or behind train to IoT Engine
- IoT Engine makes needed calculations to determine whether and how much to increase or decrease speed

3.2.3 Detect gate crossing open/closed, distance, speed and suggestion of braking, speed increase /decrease

- IoT Engine shall utilize local timers, data, and GPS speed to determine appropriate speed to arrive at gate crossing at scheduled time

3.2.4 Detect wheel slippage, using GPS and wheel RPM, suggest braking or increase/ decrease speed

- Optical sensor and satellite GPS will send data to IoT Engine regarding to train speed and RPM to calculate whether the train is slipping on the tracks
- IoT Engine will suggest braking or increasing speed based on data from sensors

3.2.5 Detect severe weather conditions, suggest braking or increase/decrease speed

- Water sensor, Anemometer and Temperature sensor will send data to IoT Engine every 5 seconds relating to rainfall, wind speeds, and temperature
- IoT Engine will make appreciate calculations to determine if severe weather conditions warrants braking, increasing/decreasing speed

3.2.4 Display Requirements

- The IoT HTR Engine will be able to display the data in its own way they receive to the train operator in an interface.
- The operator and admin will have a terminal based display and interface that will allow them to use the engine.

3.2.5 Operator, and Admin privileges

- Operator: Has access to the polished result of sensor data.
    - Operator will access display via User ID and password
- Admin: Has access to the data received by all the sensors, can enable or disable sensors, create or delete other accounts, and change account privileges
    - Admin will access IoT HTR via Admin ID and password
    - Admin has access to log data, software update and configuration, etc.

3.3 Hardware & Operating System

3.3.1: IoT Hardware shall be able to support 10,000 sensors

- The compatibility of the hardware must ensure that all the sensors of the system can co-exist and function/interact properly within the same environment.

3.3.2: The Engine will be able to support up to 5TB of data a day.

- The engine will be able to transport up to 5TB of data a day between all of the sensors and throughout the Network.

Section 4:

Requirements Analysis Modeling

4.1 Use Cases

Use Case 1: User Logs Into IoT Engine

Primary Actor: Operator

Goal in Context: To log into and Initialize the IoT Engine so that if the locomotive were to lose connection to WiFi/Cellular, the IoT Engine will be activated.

Preconditions: Locomotive is running.

Trigger: The operator initializes the IoT Engine.

1. Operator requests login of IoT Engine.
2. The IoT Engine activates operator display with sensor statuses.
3. The IoT Engine connects with all sensors, trains and displays statuses.
4. The IoT Engine tests sensors to ensure all are in working order.

Exceptions:

1. Failed to initialize: Admin is required to troubleshoot IoT Engine

Use Case 2: IoT Engine is Activated

Primary Actor: Operator

Goal in Context: To activate the IoT Engine when the locomotive loses connection to the WiFi/Cellular.

Preconditions: Locomotive is moving and IoT Engine is Initialized.

Trigger: The locomotive loses connection to the WiFi/Cellular.

1. The Locomotive loses connection to the WiFi/Cellular.
2. The IoT Engine is activated from initialization.
3. The sensors that are connected to the IoT Engine are activated
4. The IoT Engine immediately begins to perform calculations to determine further action
5. Data from sensors is continuously sent to IoT Engine to monitor conditions
6. Log is initialized and starts logging to display.

Use Case 3: IoT Engine Suggests Immediate Braking Due to Standing Hazard

Primary Actor: Proximity Sensors

Goal in Context: To apply brakes immediately when a standing hazard is detected on the tracks

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: Proximity sensors detect standing hazard on track.

1. The proximity sensor detects hazards on track.
2. Standing Hazard distance data is reported to IoT Engine.
3. The IoT Engine makes needed calculations to determine how hard to safely brake.
4. The IoT Engine displays braking recommendations to operators.
5. Operator initializes braking according to IoT Engine recommendation.
6. Immediate braking recommendation is logged to log and display.

Exceptions:

1. Standing hazard is removed from tracks and locomotive can continue as normal.

Use Case 4: IoT Engine Suggests Decreasing/Increasing Speed Due to Moving Hazard

Primary Actor: Proximity Sensors

Goal in Context: To reduce speed of the train due to moving hazards on track.

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: Proximity sensors detect moving hazard on track.

1. The proximity sensor detects moving hazards on track.
2. Hazard distance and speed data is reported to IoT Engine.
3. Speed of train is reported to IoT Engine from GPS
4. The IoT Engine makes needed calculations to determine what speed should be set to in order to ensure a safe distance from moving hazards.
5. The IoT Engine displays speed recommendations to operators.
6. Operator adjusts speed according to IoT Engine recommendation.
7. Decrease/Increase speed recommendation is logged to log and display.

Exceptions:

1. Moving hazard speed becomes negligible to locomotive's and locomotive can continue as normal.

Use Case 5: IoT Engine Suggests Change of Speed to Accommodate for Gate Closing/Opening Times

Primary Actor: GPS Speed, Timer

Goal in Context: To adjust speed of the train so that the train safely proceeds in case of wheel slippage.

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: IoT Engine detects timing error with regard to gate closing/opening times.

1. The downloaded gate closing/opening times, timer data, GPS speed should all be reported to IoT Engine.
2. IoT Engine continuously monitors arrival time based on GPS speed, timer data, distance to destination.
3. IoT Engine calculates a discrepancy in arrival time based on sensor data.
4. IoT Engine recommends a speed on display.
5. Operator adjusts speed of train to match IoT Engine recommendation.
6. Change of speed is logged to log and display.

Exceptions:

1. Schedules are adjusted and locomotive can continue as normal.

Use Case 6: IoT Engine Suggests Change of Speed to Accommodate for Wheel Slippage.

Primary Actor: Optical Sensor

Goal in Context: To adjust speed of the train so that it arrives at the gate at an appropriate time.

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: IoT Engine detects slippage.

1. GPS speed data, optical sensor data will continuously report data to IoT Engine.
2. IoT Engine continuously makes calculations to monitor slippage.
3. IoT Engine detects slippage severity and makes speed recommendations based on sensor data.
4. Speed recommendation is reported on display.
5. Operator adjusts speed of train to match IoT Engine recommendation.
6. Change of speed is logged to log and display.

Use Case 7: IoT Engine Suggests Change of Speed to Accommodate for Weather Severity

Primary Actor: Weather Sensor

Goal in Context: To adjust speed of the train so that it safely proceeds through treacherous weather conditions.

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: IoT Engine detects severe weather.

1. Water sensor, Anemometer and Temperature sensor will send data to the IoT Engine continuously.
2. IoT Engine continuously determines “severe weather”.
3. IoT Engine detects “severe weather” and makes suggested calculations for recommended speed of train.
4. Speed recommendations are displayed to the operator.
5. Operator adjusts trains speeds to match recommendations from IoT Engine.
6. Change of speed is logged to log and display.

Use Case 8: IoT is Deactivated

Primary Actor: Operator

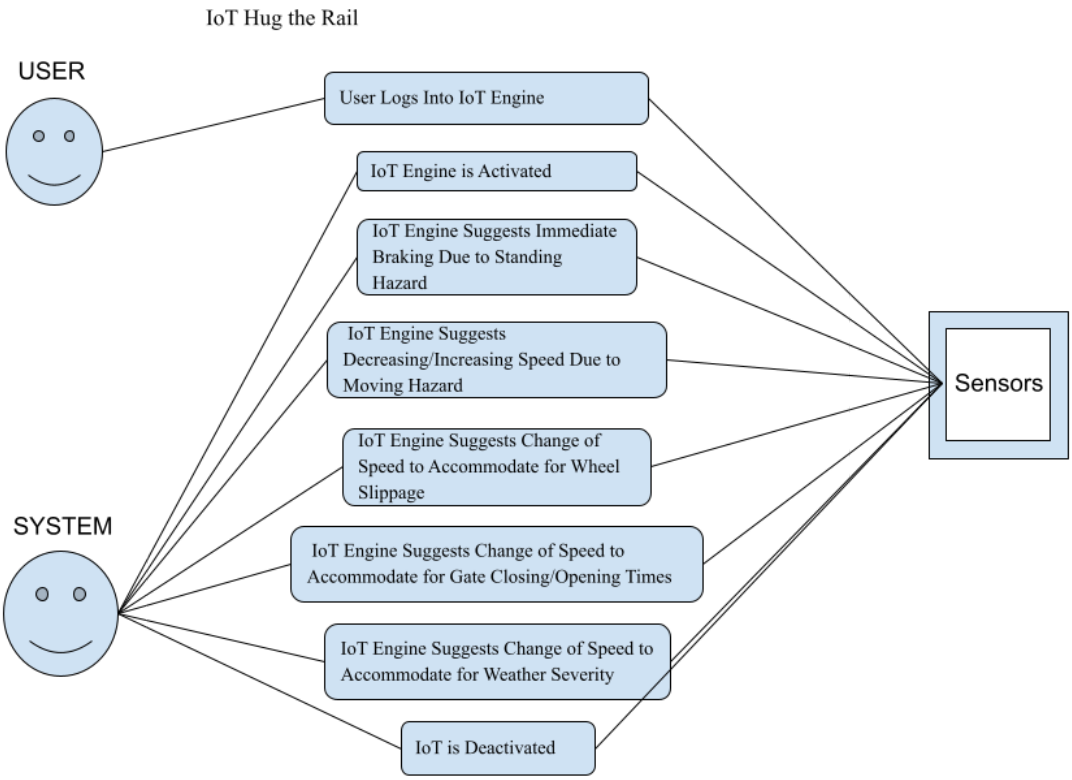
Goal in Context: To deactivate IoT Engine once cloud connection is returned.

Preconditions: Locomotive is moving and IoT Engine is activated.

Trigger: Locomotive reconnections to WiFi/Cellular.

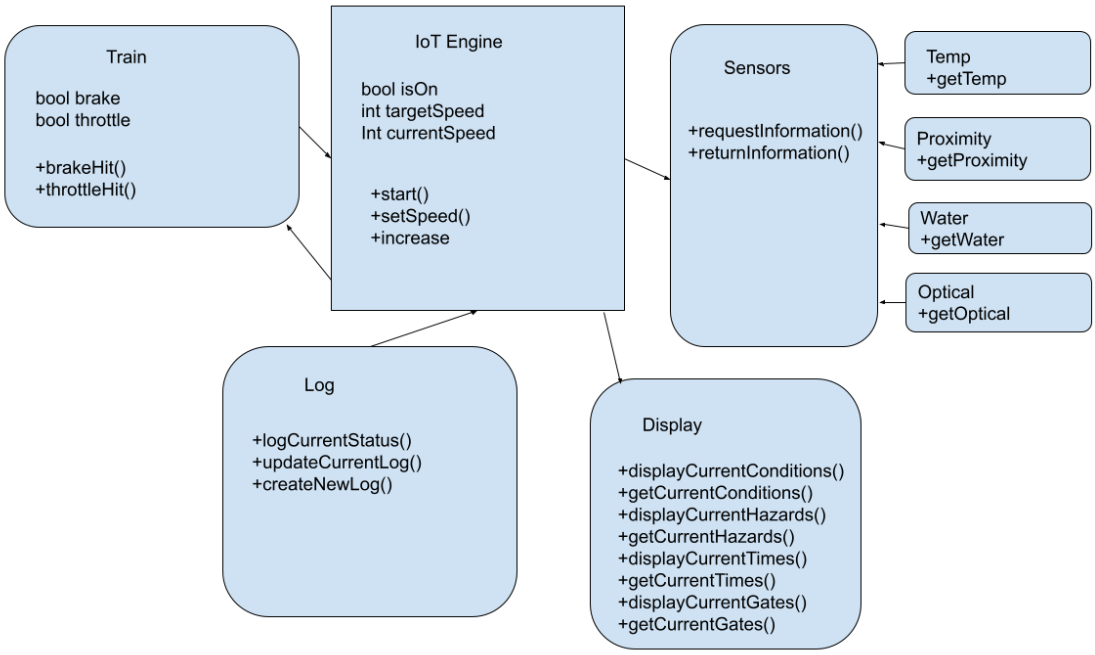
1. Locomotive reconnects to WiFi/Cellular.
2. Deactivation of the IoT Engine is displayed to the operator.
3. Deactivation is logged to log and display.
4. Log is saved into a database for records.

4.2 Use Case Diagram



For the Case Diagram we decided to take our eight main cases and then establish whether they were in relation to the User or the System which in our case is the Internet of Things Engine. For seven out of our eight cases we noticed the sensors would be the primary accessed by the Engine and not the user which makes sense because we want to make the life of the user easier so really the only thing they should handle is logging into the Engine. After that our Engine will take over all the tasks. Additionally for security reasons we feel we want to handle all the data.

4.3 Class-Based Modeling



For purposes of organization, we have divided up the key components of our project into five main classes. The first will focus on the brake and throttle of the Train which will access the IoT Engine. The IoT Engine will contain the speed including the current speed and target speed in order to ensure that the train does not surpass the speed it can handle. Additionally we will be using our Sensor class to keep track of our data where two main methods will take place - the requestInformation method and the returnInformation method. These two methods will seamlessly access the subclasses such as Temp, Proximity, Water, and Optical. These are the key components we felt would help assess the capabilities of our Train. Additionally we then needed a class to display all this information. Lastly, we wanted to keep a log of all of our data, specifically the status of the train at all times.

4.4 CRC Modeling/Cards

IoT Engine	
• Regulates Sensor data • Regulates Log data • Regulates Train controls • Regulates data to display	• Sensors • Log • Train • Display

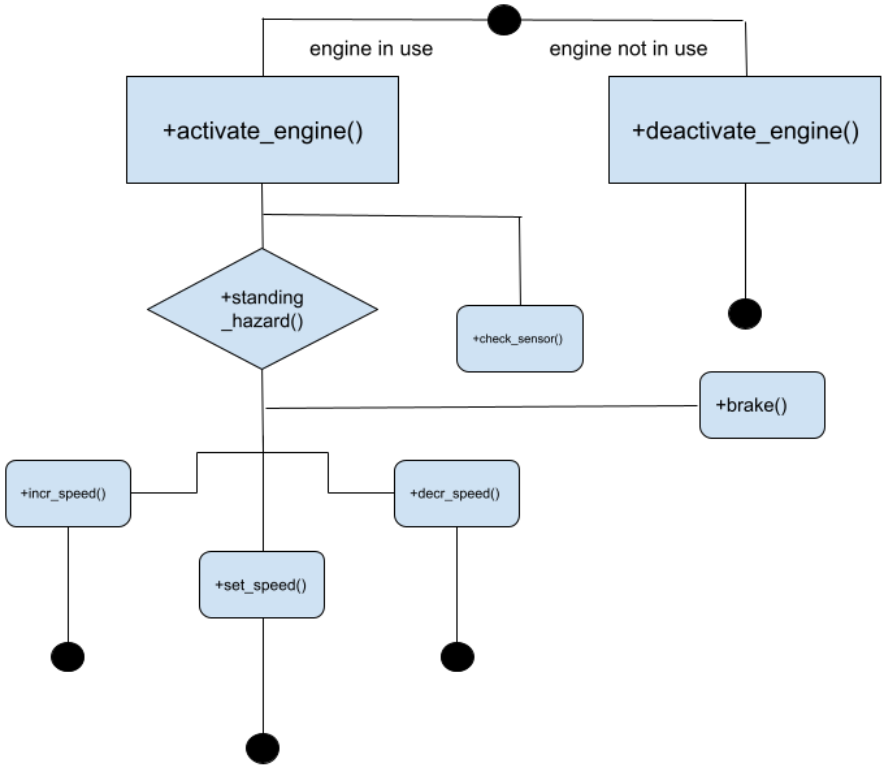
Sensors	
• Returns Information about a specific sensor • Requests Information from a specific sensor	• IoT Engine

Log	
• Updates the Current log with information • Creates a new Log with information • Logs the current status	• IoT Engine

Train	
• Controls the train's functionality • Allows the brakes to be activated • Allows the throttle to be activated	• IoT Engine

Display	
• Displays current Conditions • Gets current Conditions • Displays Hazards • Get necessary Hazard data • Display Current Gates • Get the Current dates	• IoT Engine

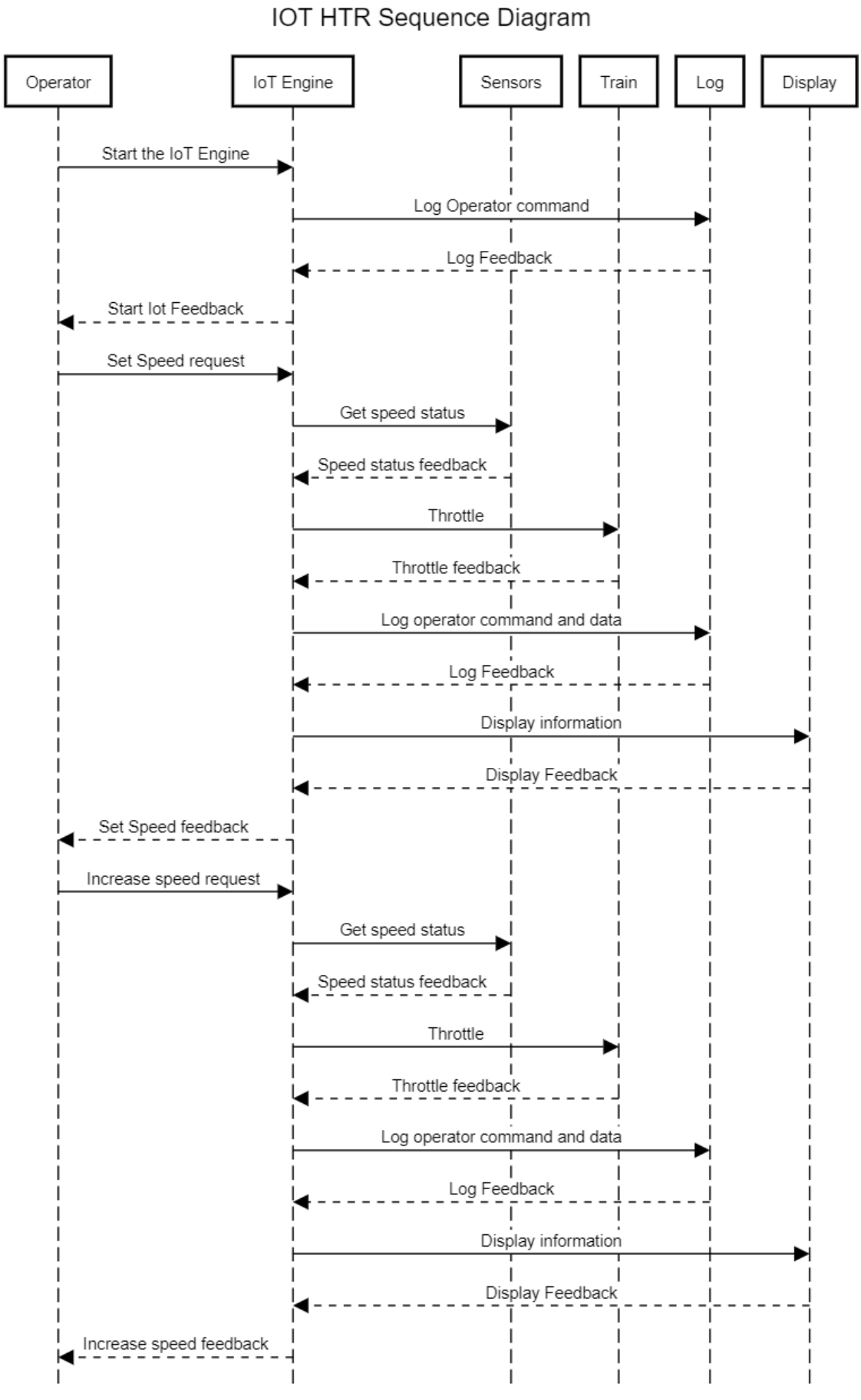
4.5 Activity Diagrams



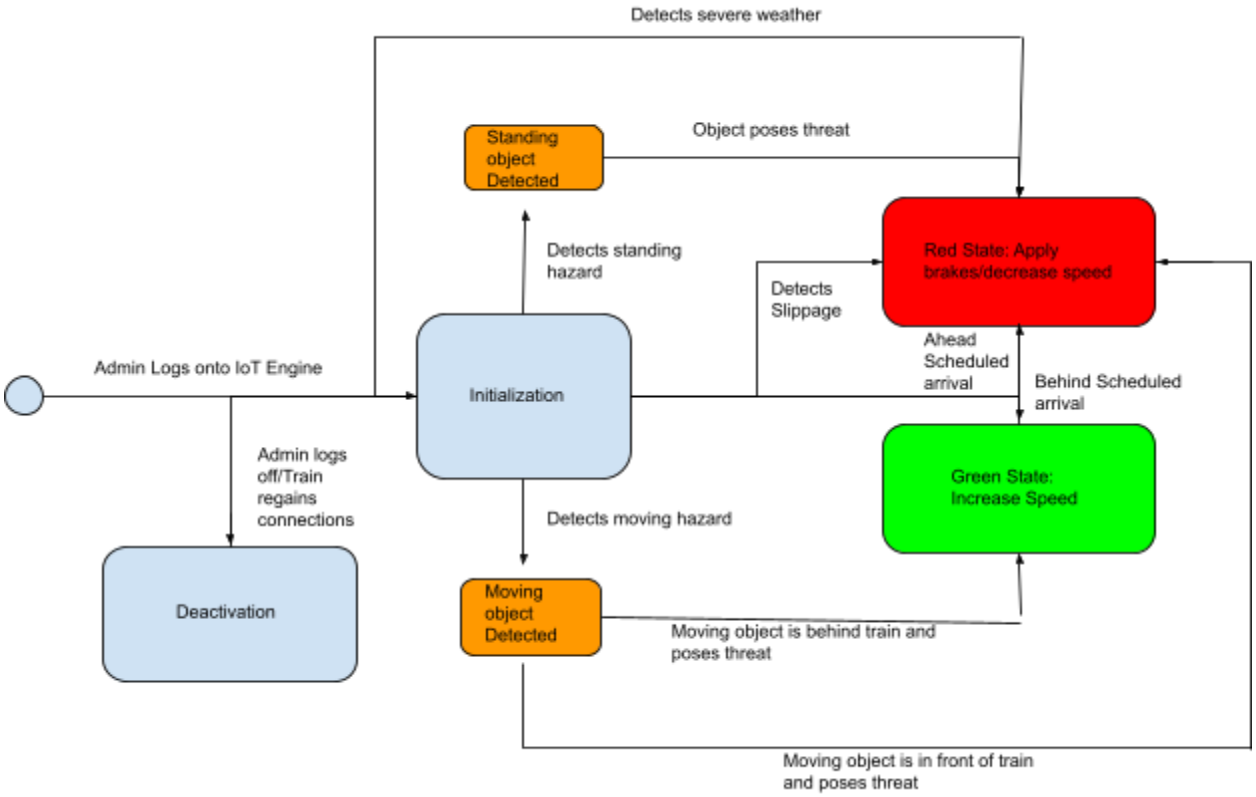
.

This activity diagram depicts the general skeleton of the behavior of the IoT Engine when it is in use versus when it is not in use. Following the description for the usage of the engine, the engine can possibly be activated when it loses connection to WiFi. In that case, it would have to abide by thought-out steps to ensure that the system can still run smoothly even if internet connection is not optimal. In this case, sensors will need to be in place and checked even without connection, and the diagram illustrates that that is the first priority once the engine is activated. The feedback from this sensor check will then lead to requests to check speed and thus assist the engine in making suggestions to increment or decrement speed accordingly, and setting speed based on the conditions derived from the sensor check. This check to assess whether change in speed is necessary is crucial to most of the potential cases in the operation of the engine/locomotive. The last case illustrated is the engine being deactivated, and this will be displayed once it reconnects to Wifi or the internet.

4.6 Sequence Diagrams



4.7 State Diagram

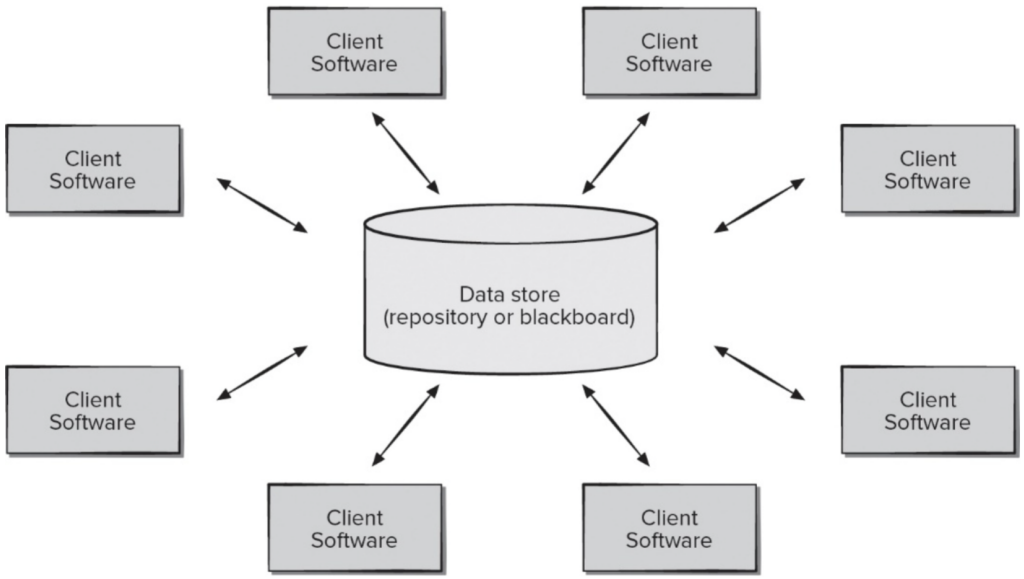


For our state diagram we have modeled our process by configuring the IoT Engine around one main “initialization” stage. Once the Admin logs onto the IoT Engine, the IoT Engine will remain in the initialization stage. From here all functional capabilities of the IoT Engine will be utilized. If the IoT Engine receives data from the sensors that there is a standing hazard it moves up to the standing object state and then if the object poses a threat then the IoT Engine will move into the “Apply brakes/decrease speed” state. Similarly, if the IoT Engine detects a moving hazard it moves to the “moving object” state where it will determine whether to move to the “Increase speed” or “Apply brakes/decrease speed” state. From the Initialization state the IoT Engine will also determine whether to move to the “Apply brakes/decrease speed” state or “Increase speed” state if it detects things like “Severe weather”, “Slippage”, “Behind schedule”, and “Ahead of schedule”. Once the Admin logs off of the IoT Engine, the state will be moved to “Deactivation”.

Section 5:

Software Architecture

5.1 Data Centered Architecture



Description:

A data store will resolve at the center of the architectural model. In the case of the IoT Engine, this data store will contain the log that displays for the operator to view. The data store will be accessed by the IoT Engine to get and input data from sensors and calculations made by functional requirements. This data centered architecture will promote integrability which will allow for any changes in the IoT Engine or its sensors to be easily integrated into the system. If needed, data could be passed among the clients. For example if the IoT Engine required data from the optical sensor and GPS speed, it can pass data from both sensors to one client software to make needed calculations, and then log data in the data store repository.

Pros:

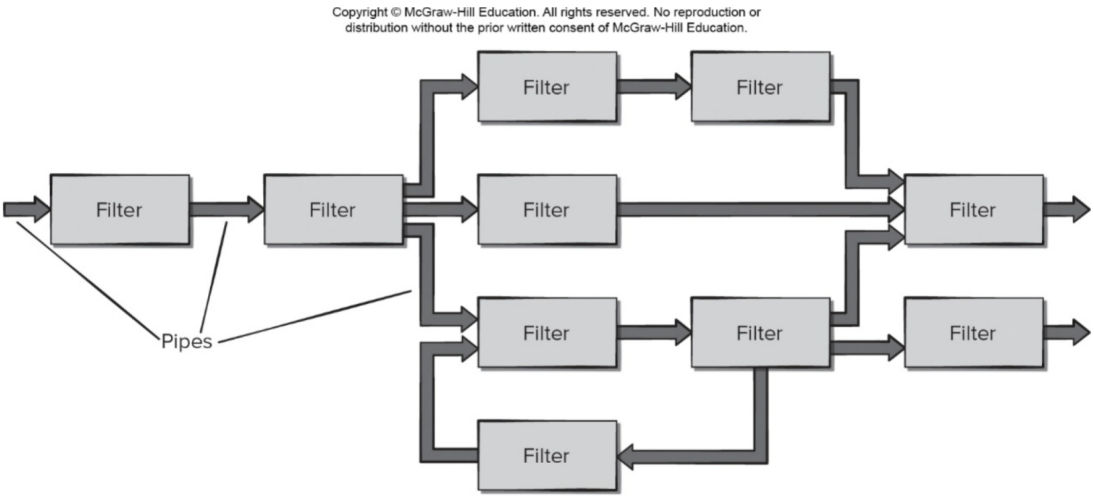
1. Data centered architecture is highly integratable
2. Data centered architecture can easily pass data around client softwares
3. Data centered architecture can easily get or store data in repository

Cons:

1. Data centered architecture is highly dependent on data store (log).

2. Data centered architecture changes in the data store can have a significant impact on client software.
3. Data centered architecture has a high cost of constantly moving data around.
4. Data centered architecture is vulnerable to data replication or duplication which can encumber data store and client software.

5.2 Data Flow Architecture



Description:

Data flow architecture can be used when the input data is to be transformed into output data through a series of computational filters. Control is passed through the architecture with pipes that connect filters together. The pipes transmit data from one filter to the next, where each filter is designed to work independently from the other. The filters do not require any type of knowledge of any of the other filters in the architecture. Data flow through this architecture is continuous as with the IoT Engine, data will constantly be pulled from sensors and passed through the filters in the architecture in order to compute different functional requirements. Most of the functional requirements of the IoT Engine require separate computations for different inputs, in this scenario, the data flow degenerates to batch sequential flow which is where there is only one flow of pipes and filters to an output which is displayed to the log.

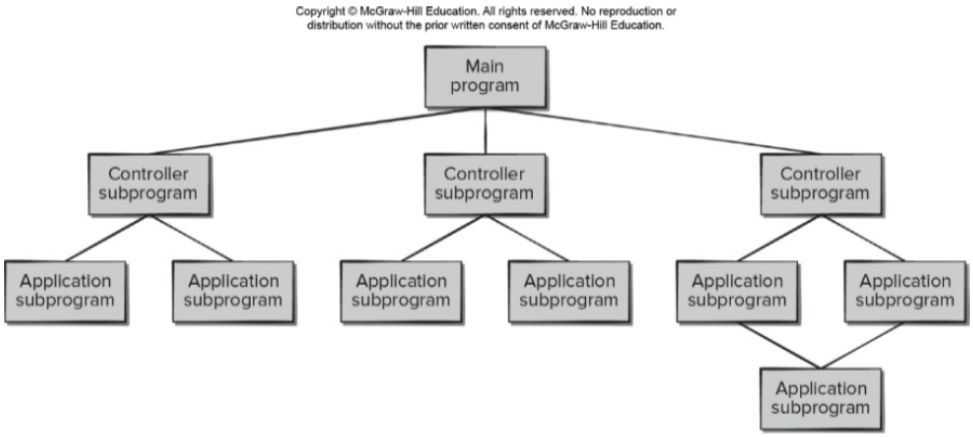
Pros:

1. Data flow architecture provides high throughput for tons of easy data processing.
2. Data flow architecture is simple and easy to understand and debug errors.
3. Data flow architecture is flexible between batch sequential flow and parallel processing.

Cons:

1. Data flow architecture is hard to integrate into IoT Engine as there are many different computations that need to be made and the architecture does not provide a suitable way for cooperative interactions.
2. Data flow architecture is difficult to configure with multiple inputs and outputs.
3. Data flow architecture does not provide a centralized repository similar to the data centered architecture.

5.3 Call Return Architecture



Description:

This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category. The main program/subprogram architecture is a classic program structure that decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components. The remote procedure call architectures are components of the main program/subprogram architecture and are distributed across multiple computers on a network.

Pros:

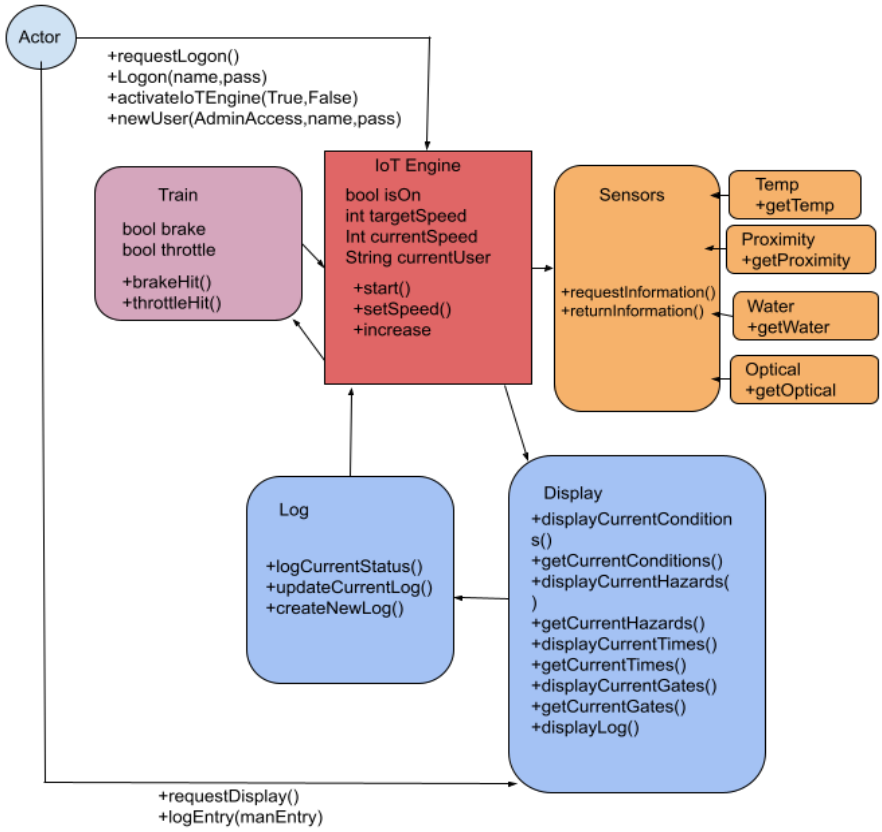
1. We are able to see each individual call our system makes
2. We are able to see all of the return values made by each call
3. We are able to see where our system will terminate
4. Each of the above reasons make this architecture easy to use

Cons:

1. We cannot identify our UML class diagrams
2. May become needlessly complex and difficult to manage

3. Makes it difficult to update a singular portion of the system without updating other parts of it as well

5.4 Object-Oriented Architecture (Our finalized choice i.e. architecture we selected)



Description:

The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing. We believe that this architecture fits best for our implementation of the IoT Engine due to its alignment and integrability with our class based modeling.

Pros:

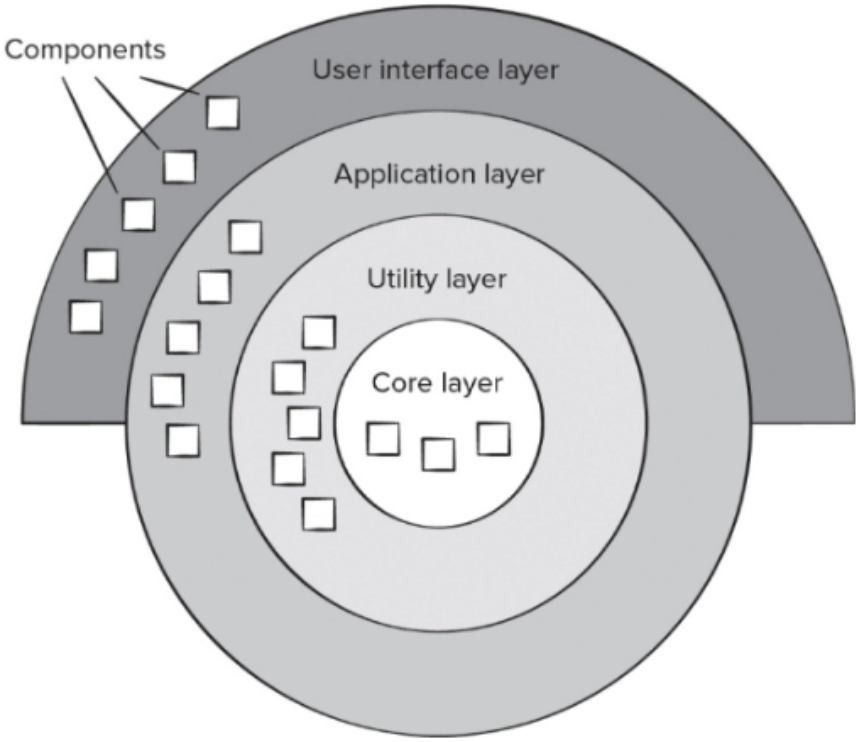
1. It fits best with the UML class diagrams that we had described in section 4 of the document
2. IoT system best fits an object oriented design
3. All of the team members are familiar with OOP therefore we will be able to implement this with confidence
4. Our object will include the logger, accelerator, brake, tire sensor, IoT, and the display

5. Object oriented architecture leads to increased reusability of code which will save on development time.

Cons:

1. The learning curve can be a bit intense, however our group is already familiar with OOP
2. It can cause duplication - however we have very thoroughly outlined our code so we should be ok.

5.5 Layered Architecture



Description:

The components within the layered architecture model are organized into a pattern that is a standard for most IT communication and organizational structures found in most companies. Each layer forms a specific role within the application; although the pattern does not specify the number and types of layers that exist in the pattern, most models consist of four: the user interface, application, utility, and core layers. This architecture is efficient and simple, but it relies on each individual component performing their job without assisting or needing assistance from other sensors within the IoT engine, which would be impractical due to the integrated nature of the system.

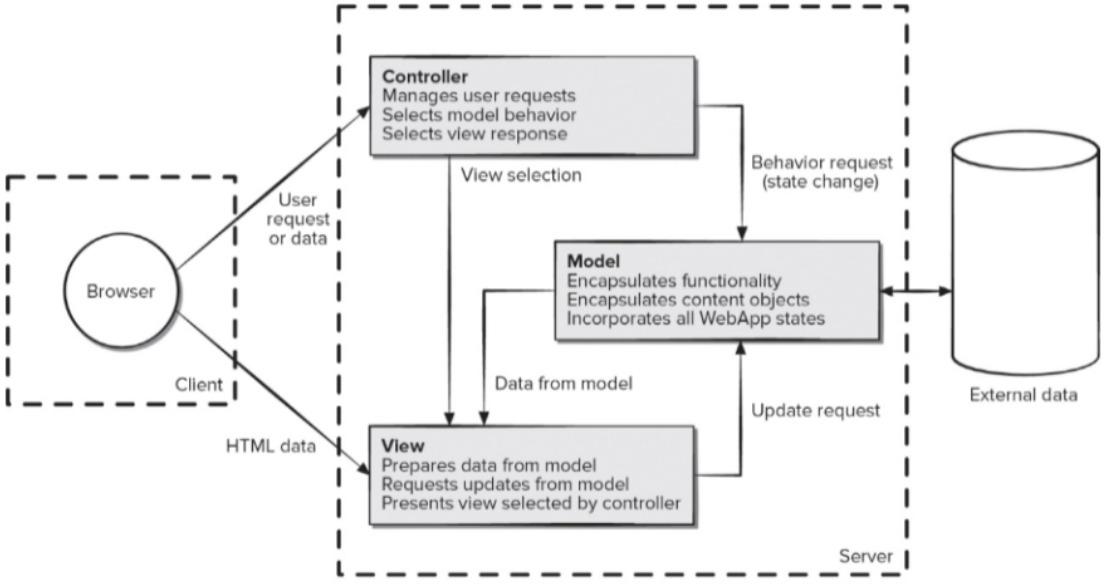
Pros:

1. Because there is separation of concern, there is a smaller scope needed to be considered by each layer, making the problems encountered by each layer more surmountable, so the process of the engine could potentially be faster.
2. Layered architecture would be easy to implement as the IoT engine's layout due to its simplicity.
3. Because of the clear organization, the code organization of the IoT engine will also be consistent across all the layered projects.
4. It is easier to find a specific sensor or part because all the objects of the same type are kept together within their respective layers.

Cons:

1. There is greater accumulation of management cost to keep the IoT engine in service if there are too many layers added due to the engine's complexity.
2. Message passing, or the mode of communication/coordination between each layer might cause slower performance if there are too many layers added; this would diminish the efficiency of the engine.
3. Usability is inhibited because of additional layers of controls, causing potential difficulties for users and slowing the engine's performance.

5.6 Model View Controller Architecture



Source: Adapted from Jacyntho, Mark Douglas, Schwabe, Daniel and Rossi, Gustavo, "An Architecture for Structuring Complex Web Applications," 2002, available at http://www-di.inf.puc-rio.br/schwabe/papers/OOHDMJava2%20Report.pdf

Model view controller architecture utilizes 3 components: Model, View, and Controller. The model component handles all of the data-related logic that the application uses. This can be the data that is transferred between the view and the controller or even just the data from the database that the user uses. The View component handles everything that has to do with the user interface or the user experience. Finally, the controller acts like an interface between the model

and the view. The controller handles most of the logic of the program such as manipulating data or telling the view what to display.

Pros:

1. Fast Development of application for the IoT engine
2. Easy for multiple collaborators to work together
3. Easy application updates for the IoT in the future
4. Easier to debug problems because the coding is separated into the 3 collections

Cons:

1. Hard to learn the architecture
2. Really strict rules have to be made beforehand and followed
3. The view of the IoT is a command line interface

Section 6:

Project Code

IoT Engine Class

```
1   public class IoTEngine {
2
3       public boolean isOn;
4       private int targetSpeed;
5       public int currentSpeed;
6       private Log log;
7       private Train train;
8
9       //starts the IoTEngine and creates a connection to each sensor
10      private void start() {
11          isOn = true;
12
13          this.log = new Log();
14          this.train = new Train();
15
16          log.createNewLog("IoTEngine");
17          log.createNewLog("Train");
18      }
19
20      //sets the new target speed of the train. returns the s
21      private void setSpeed(int speed) {
22          this.targetSpeed = speed;
23          this.log.updateCurrentLog("IoTEngine", "Train target speed set to " + String.valueOf(speed) + ": ");
24      }
25
        Run | Debug
26      public static void main(String[] args) throws Exception {
27          IoTEngine engine = new IoTEngine();
28          engine.start();
29          engine.setSpeed(10);
30      }
31  }
32
```

Train Class

```java
public class Train {
    private boolean brake;
    private boolean throttle;

    public Train() {
        this.brake = false;
        this.throttle = false;
    }

    public void brakeHit(boolean isBreaking) {
        if (isBreaking) {
            this.brake = true;
            this.throttle = false;
        }
        else {
            this.brake = false;
        }
    }

    public void throttle(boolean isThrottling) {
        if (isThrottling) {
            this.throttle = true;
            this.brake = false;
        }
        else {
            this.throttle = false;
        }
    }
}

```

Log Class

```java
import java.io.File;
import java.util.Date;
import java.text.SimpleDateFormat;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.Hashtable;
import java.io.IOException;

public class Log {
    public Hashtable<String, File> logList = new Hashtable<String, File>();

    public Log() {
        this.createNewLog("MainLog");
    }

    //updates the log with the argument supplied
    public void updateCurrentLog(String logToUpdate, String logged_data) {
        try {
            File file = this.logList.get(logToUpdate);
            FileWriter fw = new FileWriter(file, true);
            BufferedWriter logFile = new BufferedWriter(fw);
            Date currentTime = new Date();
            SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
            logFile.write(logged_data + formatter.format(currentTime) + "\n");
            logFile.close();
        }
        catch (IOException e) {
            System.err.println("An error occured.");
            e.printStackTrace();
        }
    }
    //creates a new log
    public void createNewLog(String logName) {
        try {
            String fullLogName = logName + ".log";
            File newLog = new File(fullLogName);
            if (newLog.createNewFile()) {
                this.logList.put(logName, newLog);
                this.updateCurrentLog("MainLog", "File '" + fullLogName + "' created: ");
            }
            else {
                if (!this.logList.containsKey(logName)) {
                    this.logList.put(logName, newLog);
                }
            }
        }
        catch (IOException e) {
            System.err.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```