SOFTWARE REQUIREMENTS SPECIFICATION

For

Cruise Control Module

By

MICCOTA Team

Anthony Bruno, Edward Yaroslavsky, Jared Follet and Adam Undus

At

Stevens Institute of Technology

CS 347 Software Development Process

Spring 2020

Table of Contents

Executive Summary

Over the course of this project, our team will be working on developing an efficient and innovative cruise control product. The cruise control device will act as a switch that can be enabled to maintain constant speed without the use of an accelerator. Through the use of agile process models, multiple frameworks, and continuous testing, our team will be able to successfully integrate our cruise control product to be used with every-day consumers. Ultimately, once the cruise control product is released for high-level customer use, a set of recurring maintenance will be taking place, further adapting and solidifying our project to the needs of the end-user.

Introduction

The product being developed is the software that operates the cruise control system of an automobile. The software will interpret continuously collected data from its environment and in response make decisions for the vehicle. This is a mission critical product, entailing that if any component of this product fails to perform successfully, then the product would pose serious safety hazards and concerns. As such, the system being developed must be highly reliable as to ensure the safety of those operating the vehicle. Further, as the software will be accepting and will process real time data, it must be highly responsive with the response times acting below 100 milliseconds. Therefore, it is essential that this cruise control product acts as a hard real-time mission critical system and so the language we will be choosing to utilize is Java, due to its adaptability.

The cruise control system will require both hardware and software components to be functional. As for the hardware, a switch, a clock, sensors, and a processor are required. There will also be a need for memory on which the software will be reliably stored and efficiently executed. In order to even utilize the cruise control, a switch built into the vehicle will have to be addressed. Once this step is completed, the functionality to turn the cruise control mode on and off will be enabled. Next, by using a hardware clock, a log of when the cruise control mode was turned on and off could be set up, detailing critical information when dealing with crashes and a loss of power. To continue, sensors will have to be put into place to check the status of the switch, monitor information about the clock, and set and adjust speeds. Furthermore, a sensor will have to determine if the vehicle is on cruise control while a brake is applied, terminating the cruise control mode. When dealing with the software, it is important to work with operating systems commonly used, such as the Microsoft Windows Embedded Automotive 7, Linux, and

QNX based systems. By working with the software to regulate the operations of the hardware, occasionally requesting the Engine Management System to perform various functional requirements, a successful cruise control product can be established.

Both the software and hardware components of the cruise control implementation must be exhaustively tested to ensure that they are safe and reliable when put into the consumer vehicles. Software should be tested for edge cases and unconventional circumstances to ensure that the correct action is taken when the conditions are not optimal. The hardware used in the system should be used repeatedly to check for reliability or defects. Further the system as a whole unit must be tested even more thoroughly than the individual pieces as to ensure that combining them did not introduce unexpected issues. As for security, preventing access to the main system code either remotely or directly is necessary to prevent malicious parties from tampering with the cruise control system. Safeguards to make physically and technically accessing the core of the system must be implemented such as physical protection within the vehicle and verification requirements for accessing or modifying the code within. This system will not be connected to the internet, removing the need to implement cyber security protocols to stop wireless access.

The primary actors will be individuals operating motor vehicles equipped with the Cruise Control Module. The goal of these actors will be to utilize the capabilities of the Cruise Control Module in order to safely and reliably maintain a certain speed, as well as have the ability to regain control of the vehicle at any moment in time with negligible delay. An actor would desire to use this feature in order to maintain a speed limit, or to possibly temporarily rest his/her leg on longer-duration drives. Each actor will be allowed to have four means of interaction with the system. The system can be engaged or disengaged, and the set speed can be increased or decreased. The actor will be responsible for informing the system about external changes, which will manifest themselves in one of the four possible interactions. The average actor will desire little to no information from the system, primarily just a display notification on the state of the system (engaged or disengaged). Therefore, the primary sensor that the actor will interact with will be the brake, which will disengage the system. The rest of the sensors will relay and modify a wealth of other information, such as engine operating power and acceleration, which will be used by the system administrator. Since the module will be implemented in a way to interact with existing hardware, it will have the ability to be updated, which offers a future-proof investment for a given actor, as it can constantly be updated and improved upon, and new features have the potential to be implemented, depending on the capabilities of the car. One of these possible modifications is Adaptive Cruise Control, which will adjust the speed of the car according to surrounding traffic. This will be possible, but dependent on the existence of external sensors in a given automobile.

In our system, the Actor will be responsible for engaging and disengaging the module. The actor will observe the current state of the internal system (velocity measurements) and the external conditions that will require the actor's response and input to the system in order to maintain safety. This will involve setting and adjusting the operating speed. The system admins will have access to the full hardware and software of the system, and will be responsible for maintaining, updating and configuring the system.

Eventually, this product should be able to be used for years to come. To do so, we have to ensure that the cruise control module is adaptable to possible innovations posed by the future. As a result, having a cruise control product that can evolve to the needs of the future will be one of the major fundamental factors of a successful project.

Requirements

1. Functional Requirements:

1.1. Input Requirements:

1.1.1. The Cruise Control System (CCS) shall activate upon user input (toggle on/off switch).

1.1.2. CCS shall deactivate upon user input (toggle on/off switch).

1.1.3. CCS shall deactivate when brake is engaged.

1.1.4. CSS shall momentarily deactivate while the accelerator is pressed, adjusting the speed, and reactivates when the new speed is set.

1.1.5. CCS shall maintain speed set by user.

1.1.6. CCS shall increase speed upon user input (dial) by 1 MPH.

1.1.7. CCS shall decrease speed upon user input (dial) by 1 MPH.

1.2. Output Requirements:

1.2.1. The CCS shall accept and displays acceptance of the driver's activation of the system.

1.2.2. The CCS shall display the current use status of CCS to the driver.

1.2.3. The CCS shall display the current velocity to the speed gauge.

1.2.4. CCS shall update state of use and velocity to driver in a timely manner upon receiving state-changing input from the driver.

1.2.5. CCS shall display to the driver when it is available for activation.

1.2.6. CCS shall have a log of anytime it is activated/deactivated or an update to the speed is made.

2. Nonfunctional Requirements:

2.1. Reliability Requirements

2.1.1. CCS shall be able to operate both from the alternator (when the car is running) and the battery (when the engine is off).

2.1.2. CSS shall have a brake sensor to tell when the user presses the brakes.

2.1.3. CSS shall have a throttle sensor to tell when the user pushes the throttle.

2.1.4. CS shall have a sensor in the front of the vehicle in order to tell when it is too close to another vehicle or object.

2.1.5. CSS shall have a speedometer sensor to maintain the correct speed.

2.1.6. CSS shall have a clock sensor for logging purposes.

2.1.7. The CSS sensors shall receive accurate information 99.99% of the time.

2.1.8. The CSS sensors shall respond accordingly 99.99% of the time.

2.1.9. CSS software shall operate successfully 99.99% of the time.

2.1.10. CSS shall have a sensor interface for the status of flat tires, curving roads, slippery roads, steep roads, and bumpy roads, not turning on if these conditions are too severe.

2.1.11. The margin for error for maintaining the user's speed should be within +/- 0.25 MPH.

2.2. **Performance Requirements**

2.2.1. CCS shall be ready for activation within 2 seconds after the engine startup (Design goal 0.5 second).

2.2.2. CSS shall be highly responsive with the response times acting below 100 milliseconds.

2.2.3. CSS shall operate within the range of speeds of 25 MPH and 150 MPH. Manufacturers can limit the max when producing.

2.2.4. CSS shall continuously receive feedback from the sensors whenever changes occur within 1 second.

2.2.5. CSS shall limit the use of the accelerator when moving downhill in order to maintain proper speeds.

2.3. **Security Requirements**

2.3.1. CCS shall sit on an isolated system within the car, not making any network connections, so securing from network attacks is unnecessary.

2.3.2. Only manufacturer specific software shall be able to interact with the CCS and logs.

2.3.3. CCS shall not communicate wirelessly by any means.

2.3.4. Software updates shall only be performed through the manufacturer specific software.

2.3.5. Log files shall only be read through the software and with an admin password.

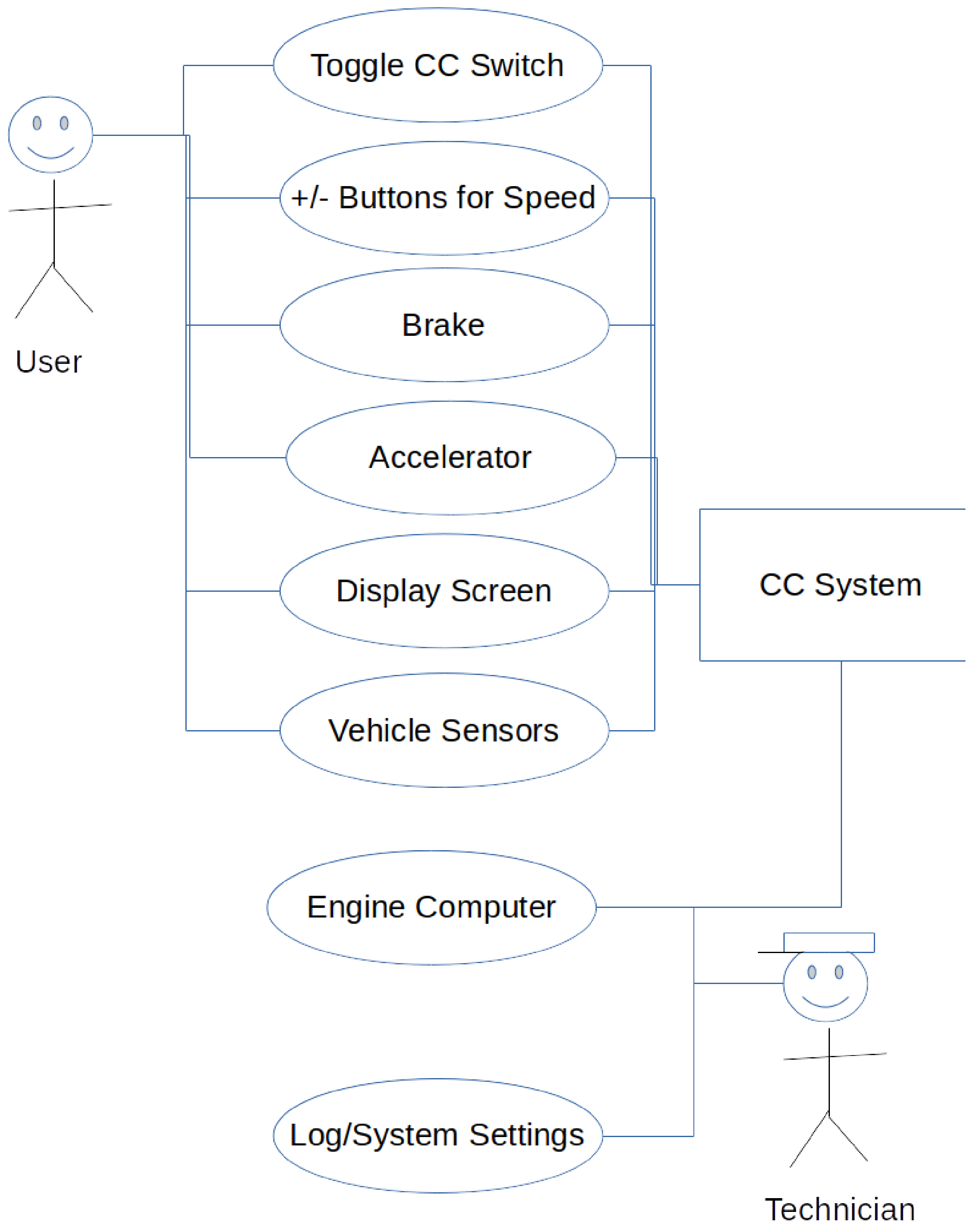
2.3.6. Log files shall only be written by the CCS itself.

2.3.7. Only the CSS interface shall be upgradeable.

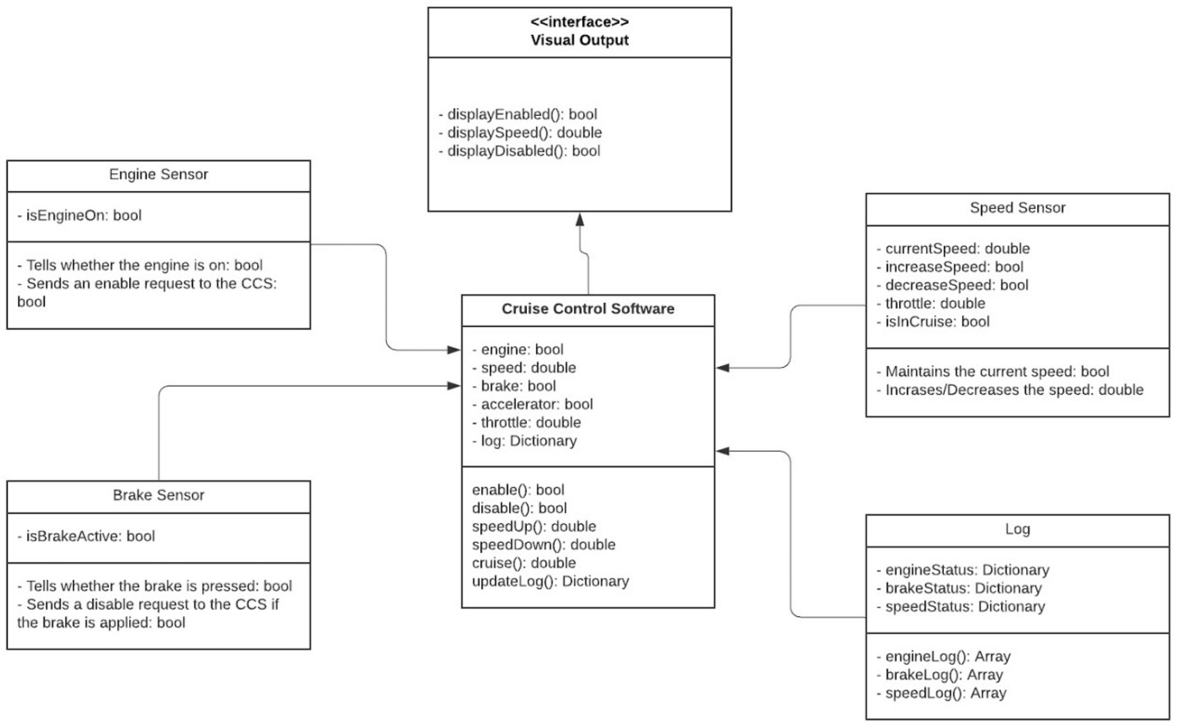
Use Cases

1. **User activates the cruise control system**
    - 1.1. User requests for the activation of the CCS via press of button.
    - 1.2. CCS provides visual feedback that it is ready for activation.
    - 1.3. CCS enters an enabled state.
    - 1.4. CCS provides visual feedback confirming it is enabled.
2. **User sets the speed**
    - 2.1. CCS gets values from the sensors.
    - 2.2. CCS calculates target speed based on user inputs and current speed from sensors.
    - 2.3. CCS requests the Engine Management System (EMS) set the speed to the target speed.
    - 2.4. EMS Speed (Throttle) is set at the current speed.
    - 2.5. CCS provides visual feedback to the user that the CCS is set and working.
    - 2.6. Sensors provide the changing environmental information to the cruise control unit (such as speed, request for increase/decrease speed, and brake).
    - 2.7. CCS detects the changes from the sensors and requests adjusting speed or deactivating CCS accordingly.
    - **2.8.** Speed is continuously reported to the CCS.
3. **User adjusts the speed**
    - 3.1. User requests an adjustment of CSS speed.
    - 3.2. CCS provides visual feedback that the system will alter the speed of the vehicle.
    - 3.3. CCS requests the speed to be adjusted by the EMS.
    - 3.4. CSS monitors speed via sensors and when the desired speed is reached, the CCS will provide visual feedback that the adjustment has been completed.
4. **Brake is applied while CCS is enabled**
    - **4.1.** If sensors report that the brake is applied while CCS is enabled, it will immediately follow the same steps as if the user deactivated CCS via a button press.
5. **User deactivates the cruise control system**
    - 5.1. User requests a deactivation of the CCS via button.
    - 5.2. CCS sends request to EMS to relinquish all control of speed back to user input
    - 5.3. CCS is disabled and provides visual confirmation that it has been deactivated.

UML Use Case Diagram



UML Class-Based Modeling



UML CRC Model Index Card

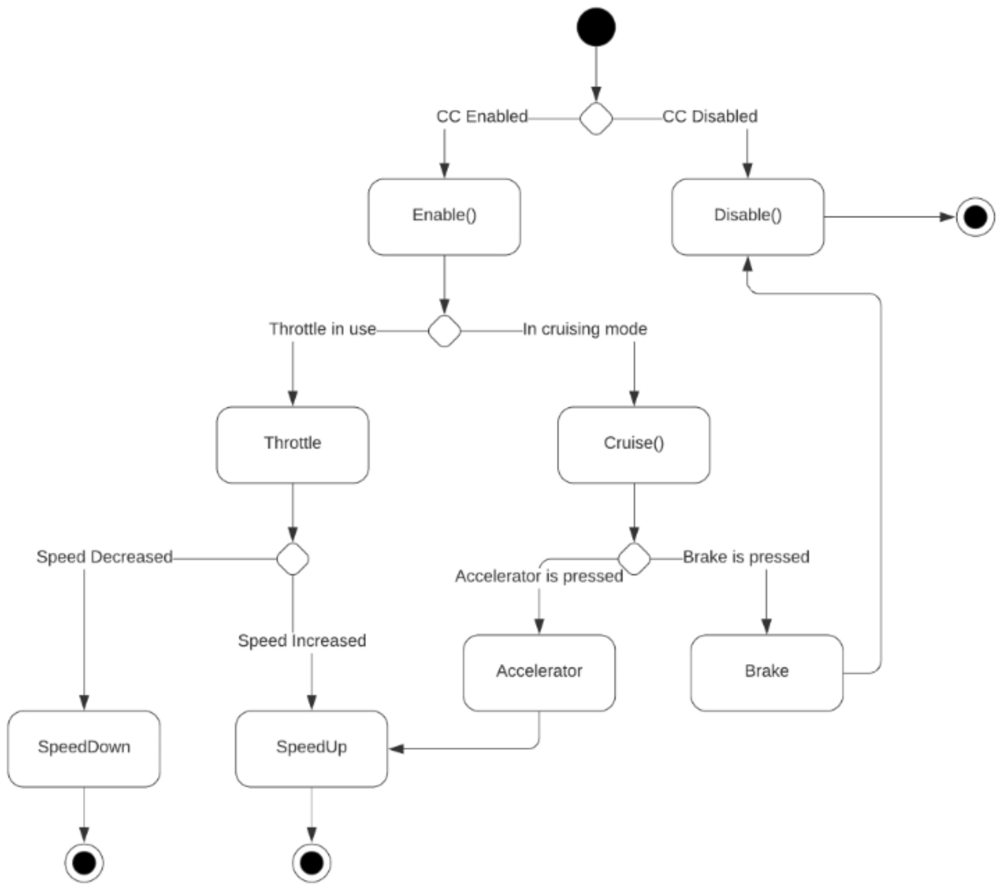
Off State
System Status = Off Display Message = None
Do: Wait for user input

CCS On(Not Activated)
System Status = Standby Display Message = Indicator
Activated Subsystems activated in standby Do: Check and propogate user input

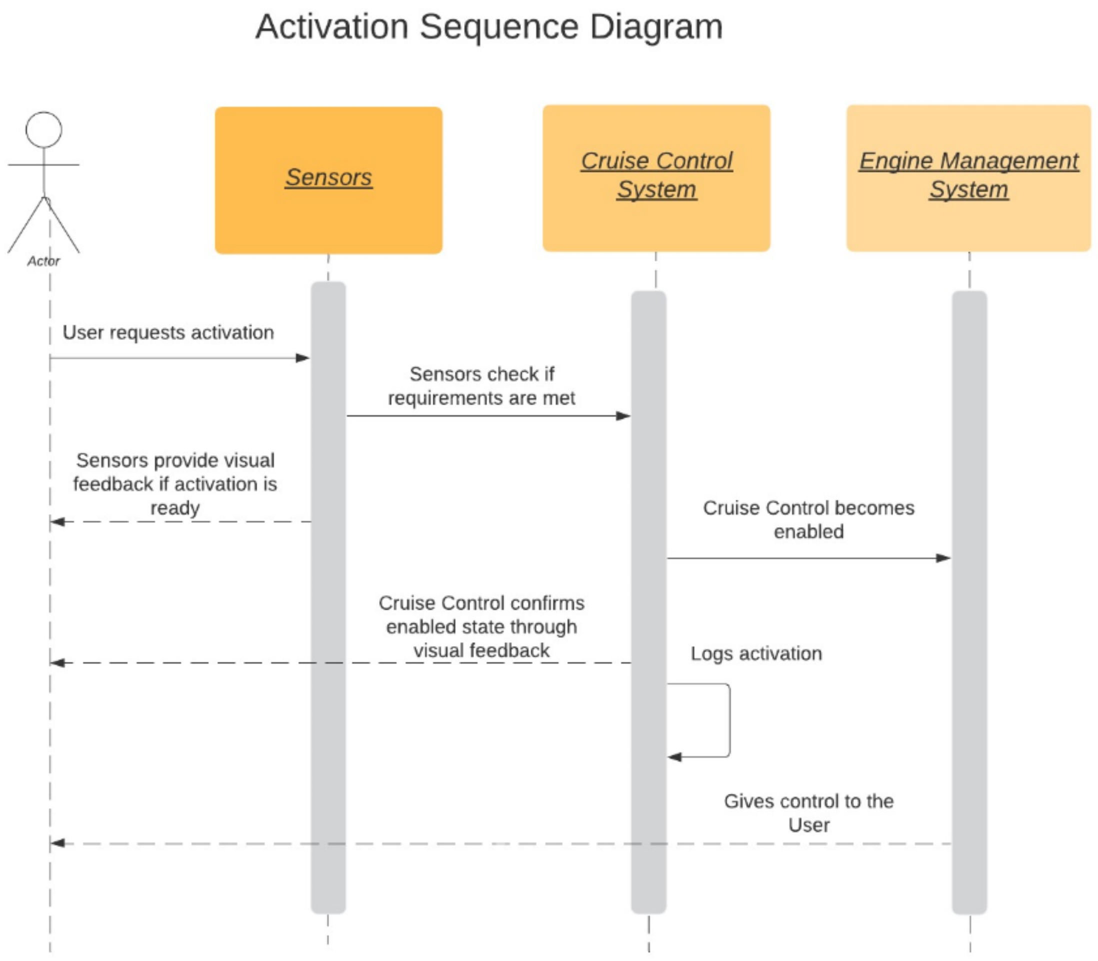
CCS Activated State
System Status = Activated Display Message = On + Current Speed
Do: Check for additional User Input Do: Check sensors for state of velocity Do: Check sensors for state of brake

Sensors
CCS State Sensor EMS State Sensor Velocity Sensor Brake Sensor
Do: Return state of CCS Do: Return state of Velocity Do: Return state of Brake Do: Update CCS in accordance with state of brake (If pressed, deactivate; else, continue)

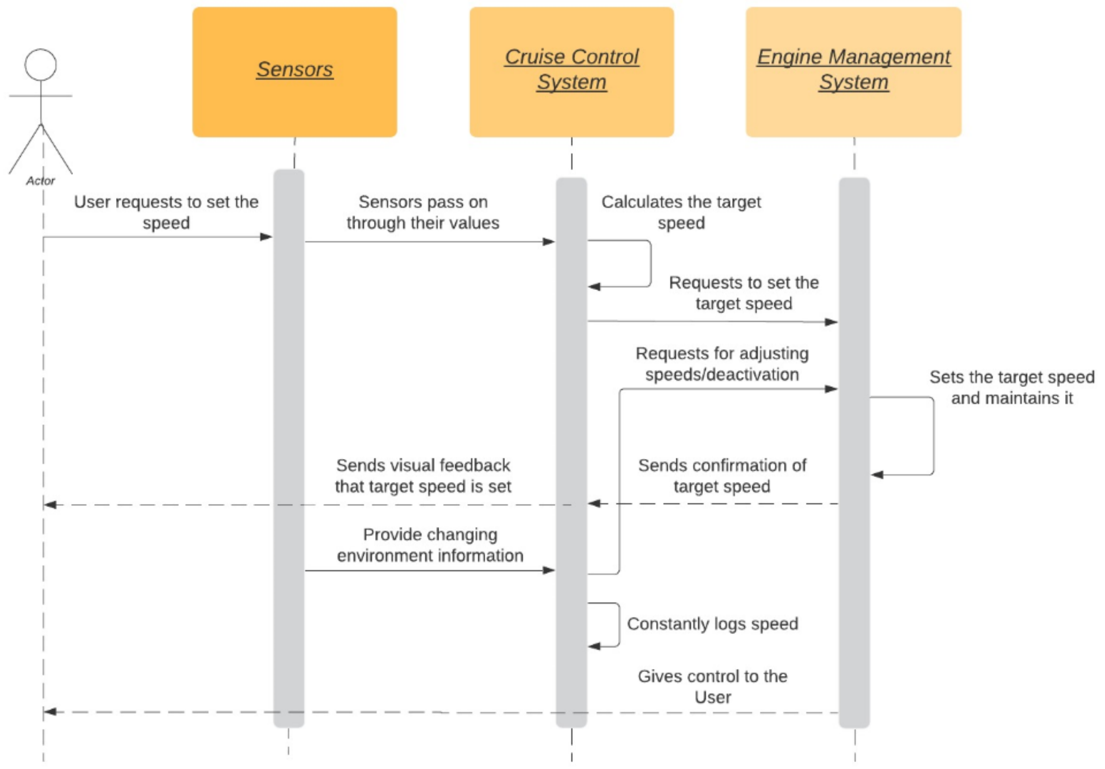
UML Activity Diagram



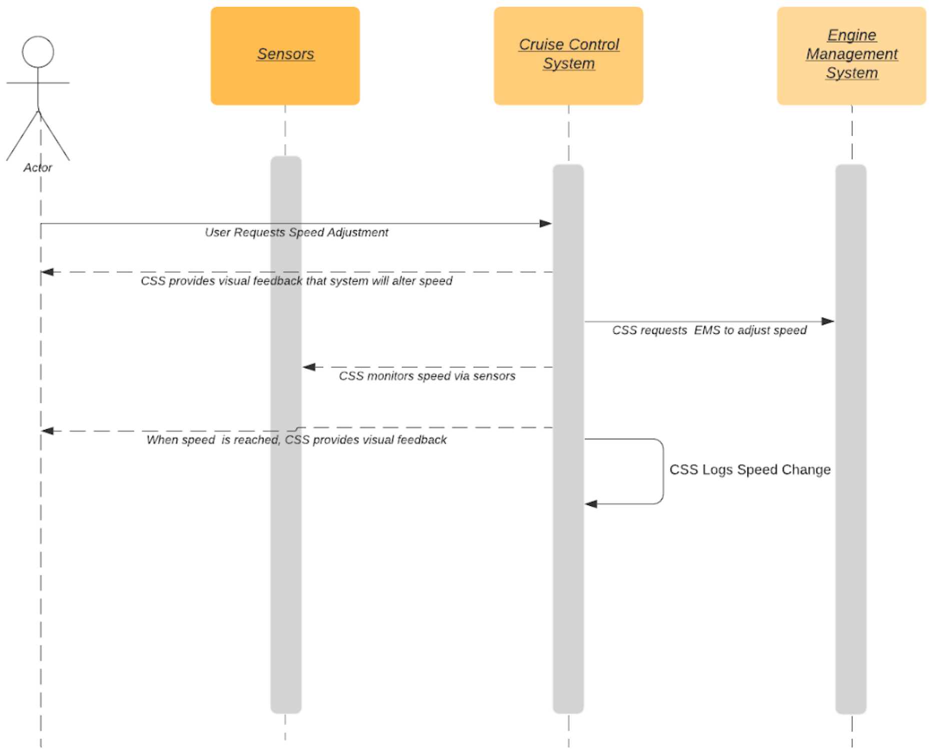
UML Sequence Diagrams

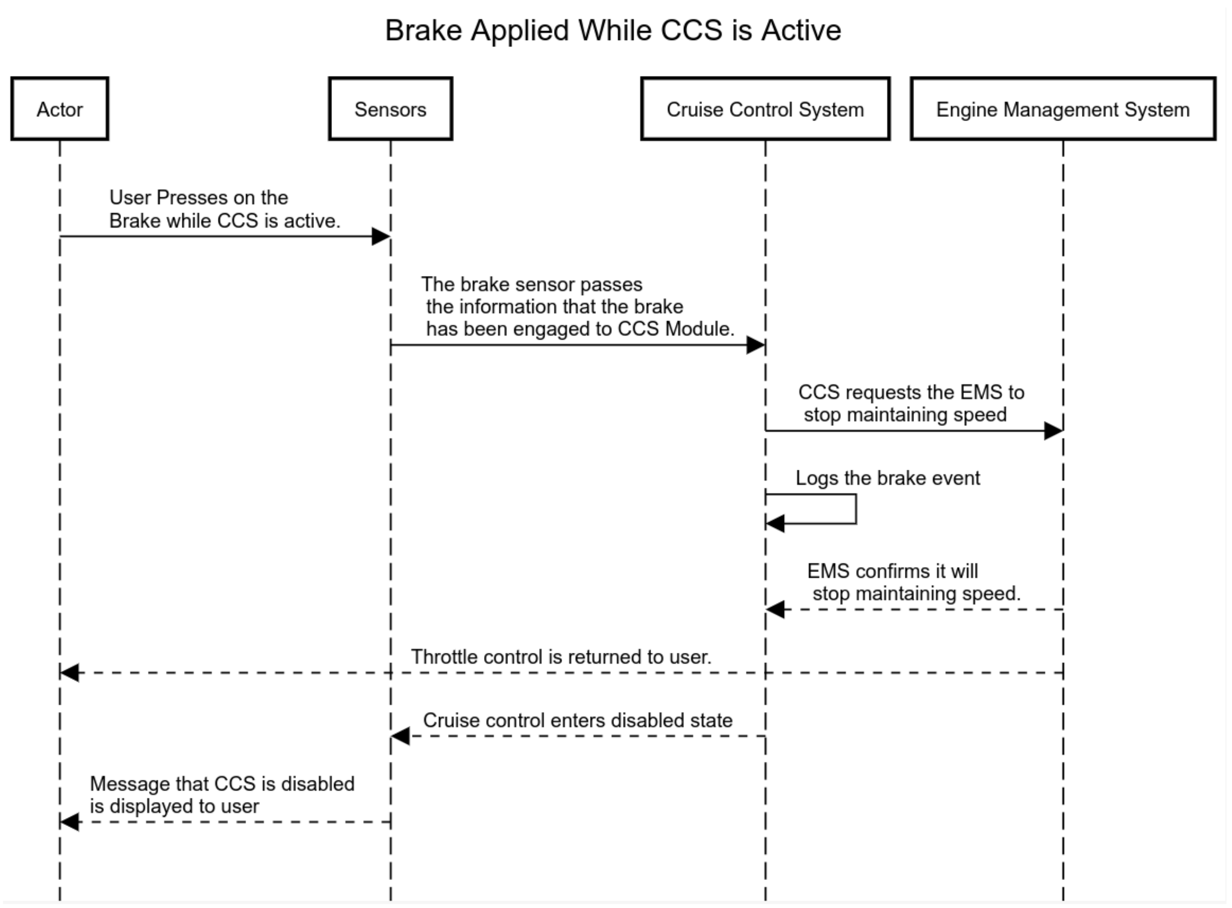

Activation Sequence Diagram
Actor
Sensors
Cruise Control System
Engine Management System
User requests activation
Sensors check if requirements are met
Sensors provide visual feedback if activation is ready
Cruise Control becomes enabled
Cruise Control confirms enabled state through visual feedback
Logs activation
Gives control to the User

Speed Sequence Diagram

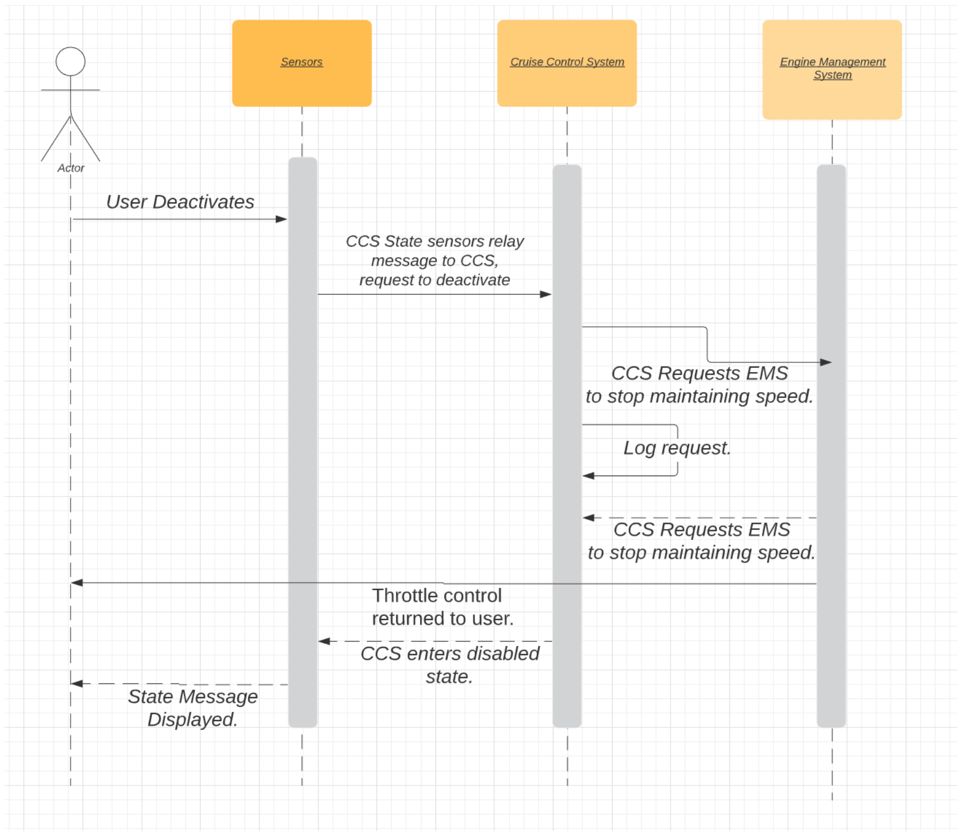

Sensors
Cruise Control System
Engine Management System
Actor
User requests to set the speed
Sensors pass on through their values
Calculates the target speed
Requests to set the target speed
Requests for adjusting speeds/deactivation
Sets the target speed and maintains it
Sends visual feedback that target speed is set
Sends confirmation of target speed
Provide changing environment information
Constantly logs speed
Gives control to the User

Adjusting Speed Sequence Diagram

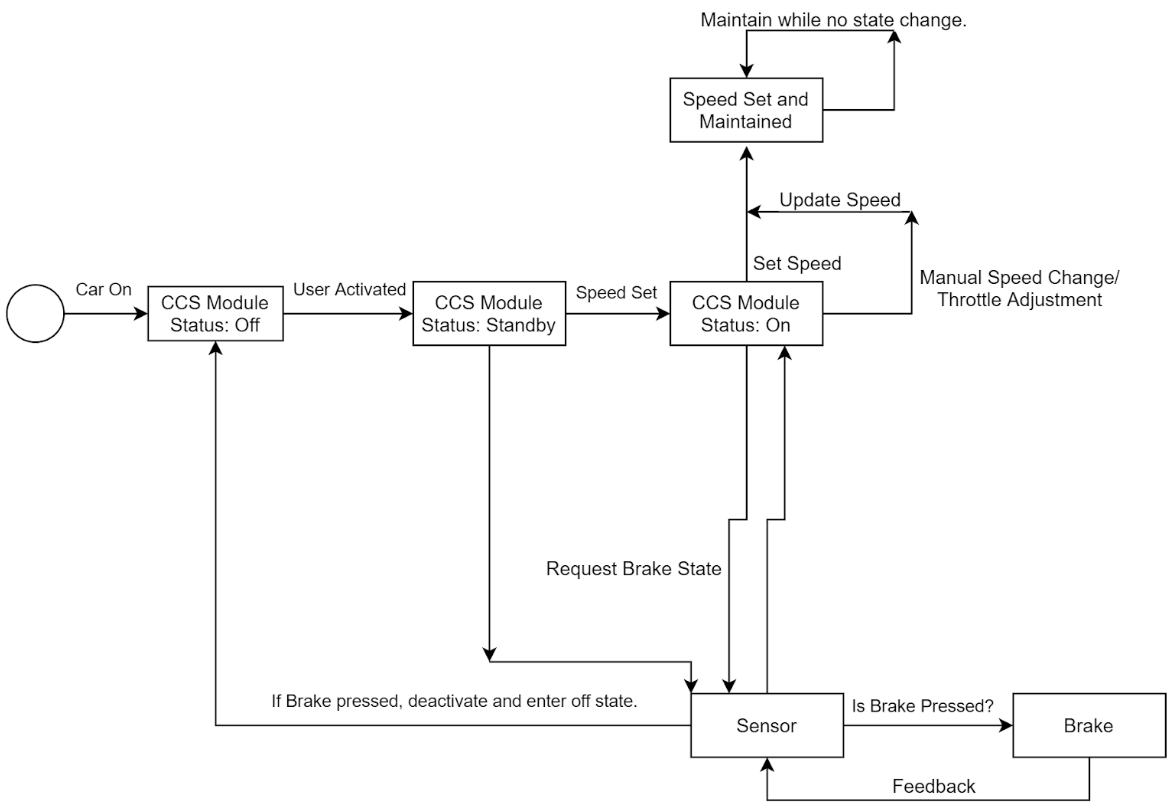

Sensors
Cruise Control System
Engine Management System
Actor
User Requests Speed Adjustment
CSS provides visual feedback that system will alter speed
CSS requests EMS to adjust speed
CSS monitors speed via sensors
When speed is reached, CSS provides visual feedback
CSS Logs Speed Change


Brake Applied While CCS is Active
Actor
Sensors
Cruise Control System
Engine Management System
User Presses on the Brake while CCS is active.
The brake sensor passes the information that the brake has been engaged to CCS Module.
CCS requests the EMS to stop maintaining speed
Logs the brake event
EMS confirms it will stop maintaining speed.
Throttle control is returned to user.
Cruise control enters disabled state
Message that CCS is disabled is displayed to user

Deactivation Sequence Diagram


Sensors
Cruise Control System
Engine Management System
Actor
User Deactivates
CCS State sensors relay message to CCS, request to deactivate
CCS Requests EMS to stop maintaining speed.
Log request.
CCS Requests EMS to stop maintaining speed.
Throttle control returned to user.
CCS enters disabled state.
State Message Displayed.

UML State Diagram



Software Architecture

1. Architecture Style

Style	**Pros**	**Cons**
Data Centered	This works well for multiple users accessing a single point of a data store.	A Cruise Control shouldn't rely on a single data store, mainly focused on its own dependency.
Data Flow	This is a strong method for innovating and manipulating data.	The data the Cruise Control will deal with will not be heavily innovated or manipulated.
Call Return	This is a more simplistic architecture that is easier to work with and understand.	While working with a Cruise Control System, this may complicate a Call Return model too much due to various functions, such as updating tasks, making it difficult to incorporate.
Object-Oriented	This is a familiar approach that fits in well based on the previously developed design choices on implementing the Cruise Control. This allows for greater management and ease in developing Cruise Control for our team.	This may be a tougher model to implement.
Layered	This is a clean and simple model architecture.	While working with a Cruise Control System, this model could be limited by various requirements and necessities.
Model View Controller	This design process allows for easy development of large applications.	This is a complex approach that would limit our team's capabilities to make an effective Cruise Control.

An appropriate architecture for a Cruise Control System would be the Object-Oriented style. Although we considered other styles such as the Data-Centered Architecture and the Call Return Architecture, we decided that the Object-Oriented architecture yields the most benefit to the project. While the Data-Centered has the advantage of having centralized data that can be used to deal with speed and sensor detection, the disadvantage lies within the components only having the shared data repository as the main connector between them. This means each component would be relatively independent and would have trouble interacting with each other when necessary. Furthermore, as a Call Return Architecture has the potential of a manageable hierarchy with efficient performance due to its thread-like control, where the constant running of the Cruise Control System would be depicted in the Main Program, it lacks the power to provide exception handling in a simple manner. Therefore, dealing with cases such as sudden braking would pose a challenge. As a result, even though Object-Oriented Architecture may not be as efficient as some other styles, it helps to maintain a centralized procedure that is easy to follow and can deal with complexities by breaking them down into separate classes, making it a strong choice to develop the architecture to our Cruise Control System with.

2. Architecture Components, Connectors, and Constraints

2.1. Components

Within our architecture, we would have our components as abstract units, involving classes, interfaces, and the functions inside them, supplying the transformation on our data elements through calculations. Some of these components of a software system would include:

- A Login System that allows for the access of the Cruise Control.
- A main Cruise Control program that acts based on the information of the child classes. This main program will be making decisions and enforcing changes through the data from the sensors.
- Sensors that act as children classes, gathering data and relaying that information back to the main Cruise Control program parent whenever requested. Furthermore, sensors will send out alerts when a prominent event is triggered.
- The Engine Management System that acts as a child process that communicates with the main Cruise Control program parent to either make changes to the vehicle's speed or just keep it constant.
- The User Interface that is a child of the main Cruise Control program parent and that uses its interface to read data and provide the various statuses back to the user.
- A Backend that helps to manage and to read the output logs or to update the Cruise Control System.

- A System Administrator that is an actor and manages certain permissions and restrictions.

2.2. Connectors

We will also have connectors that link the various pieces of the architecture together, representing individual classes linked to a parent or interface. These connectors will be mostly executed by function calls and through means of parameterization, where the data will be packed in specific structures. Being internal to the Cruise Control System, the connectors will help to pass messages and the information between the main Cruise Control program parent and its children classes. In this fashion, the system and connectors will operate similar to an API.

2.3. Constraints

Moreover, we would have a set of configurations integrated into the class and interface architecture to help with the constraint-focused design meant to either advise limits, restrict certain actions, or generally upkeep the requirements of the Cruise Control System. For example, we are constrained to use interaction with our sensors, to have the interaction with EMS, to be sound in performance and reliability, and to represent the feedback through a specifically-designed User Interface. Doing so will enable a successful architecture that can operate both safely and efficiently.

3. Control Management

Within the architecture, the main CCS class would have children that represent the different states the system can be in. This main class would take in information from connected sensors and distribute it to the child classes to perform calculations on this data and make decisions based on the current state of the CCS system. Information will then be returned back to the main class to be exported to the EMS and used to control the vehicle. In saying, a distinct control hierarchy emerges from this system. The role of the components within this hierarchy is to maintain control over the CCS when it is currently in the state corresponding to the class. The components transfer control upon state changes reported by the sensors. The classes will all eventually transfer control to the main system which performs operations and then delegates responsibilities to the EMS. The geometric form of the control topology resembles a pyramid, which operates in a synchronized fashion, the topmost point being the main class and the base consisting of the multiple child classes corresponding to the possible CCS states.

4. Data Architecture

Data is communicated between components via signal pathways where each end either transmits or handles the signal. The flow of data is continuous, which is transferred by an Interrupt- initiated mode. The data components within serve as signal transmitters and monitors. The functional components interact with active data components via sensors which create signals that are interpreted by intermediate I/O devices. Data directly interacts with control as the control is based upon the received data.

Data Components

- Sensor Data from EMS
- Driver input
- Input from vehicle diagnostics
- Cruise Control State

Functional Component Interaction with Data

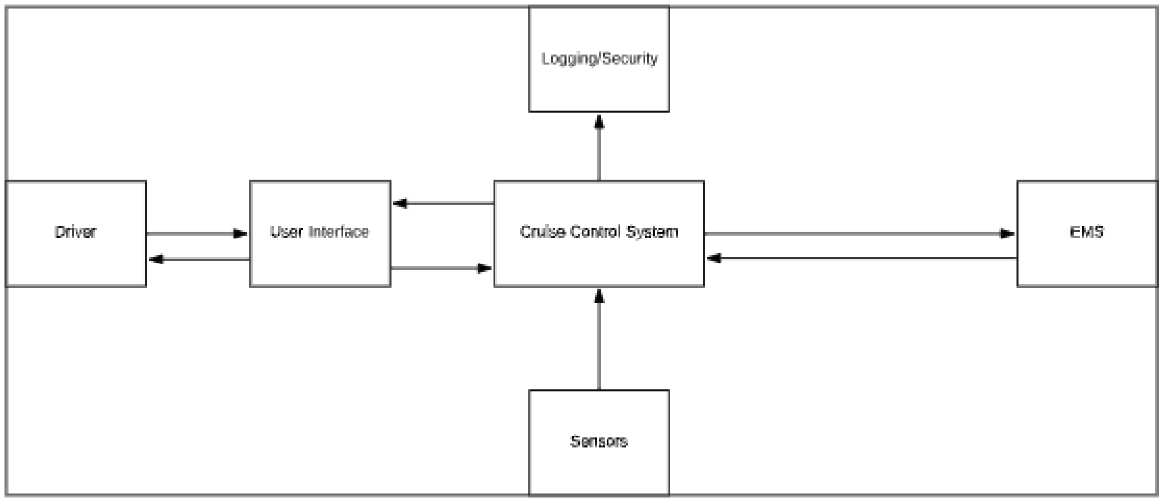
- Send target speed to EMS
- Receive current speed from EMS
- Receive Brake signal from EMS
- Send data to UI
- Update system state

Control of Data Flow

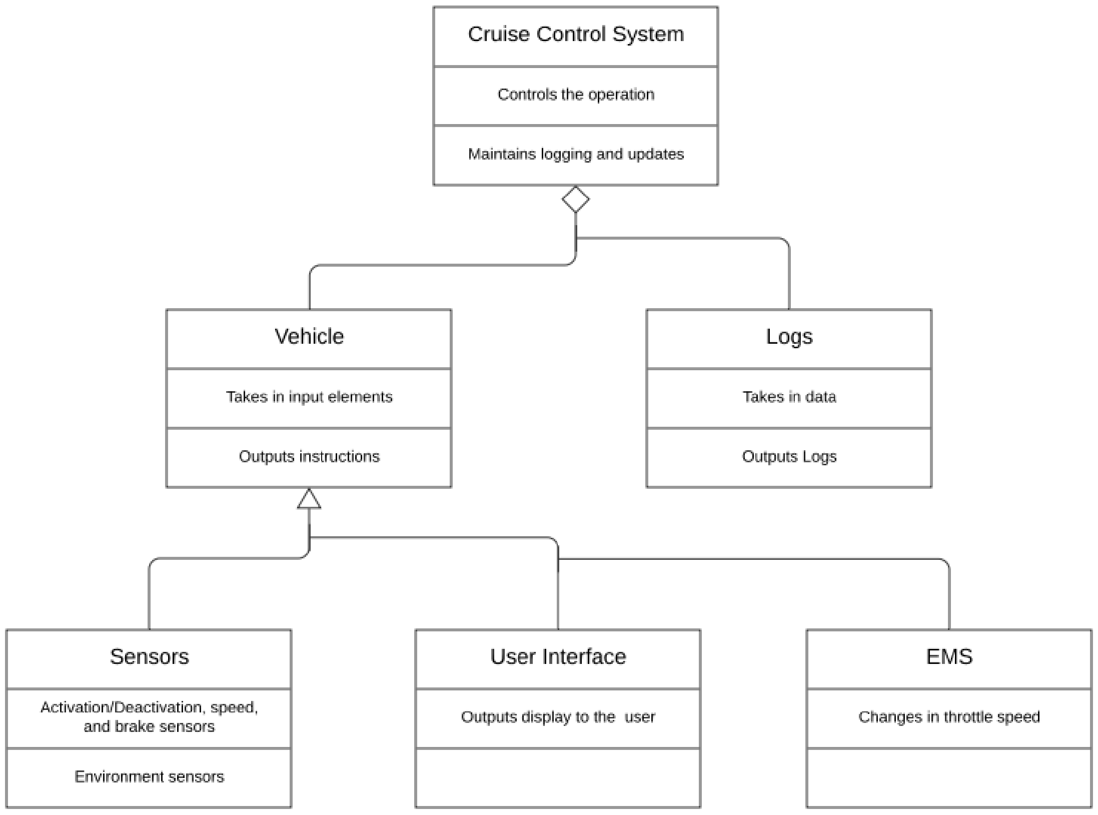
- User actions control the system state and communication with EMS
- Sensor data prevents system from entering certain conditions
- All incoming/outgoing data is logged.

5. Architectural Designs

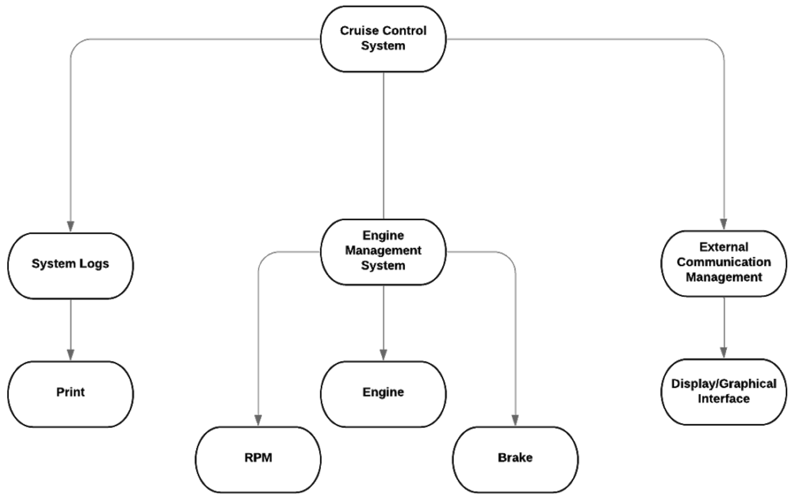
5.1. Architecture Context Diagram



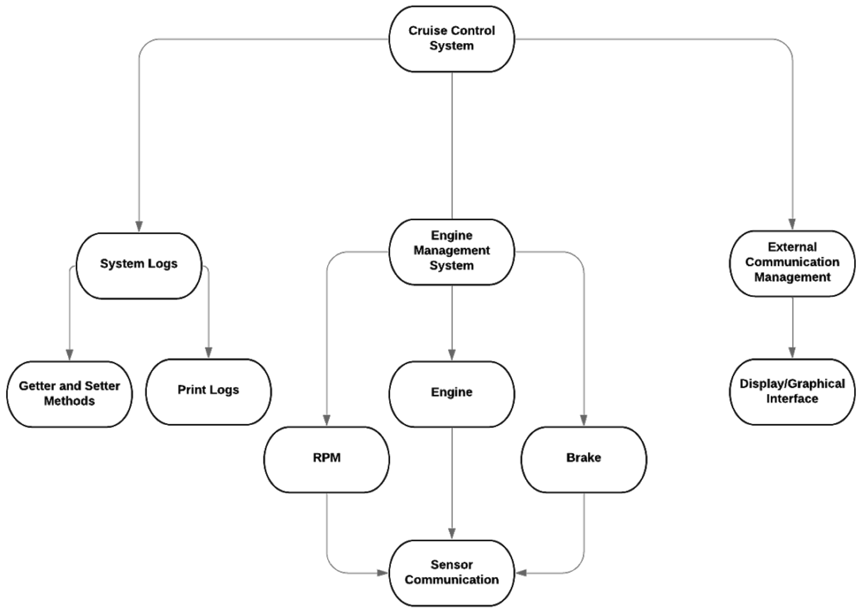
5.2. Cruise Control Function Archetype



5.3. Cruise Control Top-Level Component Architecture



5.4. Cruise Control Refined Component Architecture



6. Project Code

The Miccota team chose to go with Java in order to implement the module. We chose this language as a result of its OOP capabilities. Our Module has been broken down into the following classes with their respective synopsis':

- **CruiseControl.Java**
  - This class is the driver code for the CCM. It holds all methods regarding setting the current state of the CCM, as well as grabbing information from the EMS in order to appropriately function.
- **EMS.java**

  Our EMS class primarily serves as the backend code for simulating the actual vehicle functions, described below in the UI.java class synopsis.
- **Log.Java**
  - Our log class handles logging each event, such as status changes of the CCM, brake/accelerator actions, speed changes and other useful errors/exceptions.
- **UI.java**
  - This class creates a graphical interface to simulate a car, providing functionality such as turning the car on and off, manually changing vehicles speed via an accelerator and brake. It also provides a graphical interface to control the CCM. The CCM may be activated and deactivated, as well as increasing/decreasing the speed by 1. This UI also provides access to the log for each instance of our java executable. It is password protected to enforce the developer security policy we have instantiated above.
- **Status.java**
  - This class is very short and simply holds the enum structure for the three possible CCM statuses: Off, Ready or Engaged.

CruiseControl.Java

```
/*
            Created By: Jared Follet, Edward Yaroslavsky, Adam Undus and Anthony Bruno
            Modified On: 5/12/2020

*/
package miccota;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

public class CruiseControl {
    private Status status; //Status of the CC. Either Off, Ready or Engaged.
    private double targetSpeed; //The target speed set by the CCM after activation.
    private EMS ems; //
    private Log log; //Log files to keep track of all EMS and CCM events.

    public CruiseControl() {
        status = Status.READY;
        targetSpeed = 0;
        ems = new EMS();
        log = new Log();
    }

    public double getSpeed() {
        return ems.getSpeed(); //Return the current speed of the engine.
    }

    public boolean setTargetSpeed(double newSpeed) {
        if (status != Status.ENGAGED) { //Check if the CCM is engaged, if it is not you cannot set targetSpeed.
            System.out.println("Cannot set speed when CC is not in engaged state");
            return false;
        }
          if (newSpeed < 25 || newSpeed > 150) { //Make sure that the new targetSpeed is between the CCM operating
values, 25mph->150mph
            System.out.println("Invalid speed: " + newSpeed);
            return false;
        }

        targetSpeed = newSpeed;
        ems.requestSpeed(targetSpeed); //Update target speed.
        log.speedLog(targetSpeed); //Log the event.
        return true;
    }

    public boolean isCarOn() {
        return ems.isCarOn(); //Return the current state of the car. Either On(true) or Off(False).
    }
    public Status enable() { //Method to change the status of the CCM to Engaged from Ready.
            if (status == Status.READY) { //CCM status be ready, implying the car has a speed between 25mph and
150mph.
            status = Status.ENGAGED;
        }
        log.CClog(status); //Log the event to status log file.
        targetSpeed = ems.getSpeed(); //Update target speed.
        ems.requestSpeed(targetSpeed);
        return status;
    }

    public Status disable() { //Method to disable the CCM, which
        status = Status.READY;
        log.CClog(status); //Log the status update to the status log file.
        ems.stopAutoAcceleration(); //CCM is now off so Stop engine acceleration to a new target speed.
        return status;
    }

    public Status getStatus() { //Getter. Returns the current status.
        return status;
    }

    public boolean increaseSpeed() { //Method used by Increase Speed By 1 Button.
                if (status != Status.ENGAGED) { //Check if CCM is engaged, otherwise cant increase.
            System.out.println("Cannot increase speed if system is not in engaged status");
        }
        if (targetSpeed == 150) {
                                    System.out.println("Cannot increase speed more than the max speed");
        }
        targetSpeed++; //Update target speed +=1
        ems.requestSpeed(targetSpeed); //Send a request to the EMS to change current speed.
        log.speedLog(targetSpeed); //Log the change in speed to speed log file.
        return true;
    }

    public boolean decreaseSpeed() { //Method used by Increase Speed By 1 Button.
        if (status != Status.ENGAGED) { //Check if CCM is engaged, otherwise cant increase.
            System.out.println("Cannot decrease speed if system is not in engaged status");
        }
        if (targetSpeed == 0) {
                                    System.out.println("Cannot decrease speed if there is no set speed");
        }
        if (targetSpeed == 25) { //Ensure that the target speed does not fall below 25mph.
                                    System.out.println("Cannot decrease speed more than the min speed");
        }
        targetSpeed--; //decrease speed.
        ems.requestSpeed(targetSpeed); //Update ems.
        log.speedLog(targetSpeed); //Log the event.
        return true;
    }

    public void brakeApplied() { //Method used by Brake Button.
        if (status == Status.ENGAGED) { //Follow these actions when CCM engaged.
```

```
            status = Status.READY; //Dis-engage CCM.
            ems.startDeceleration(); //slow down vehicle.
            ems.stopAutoAcceleration();
            log.brakeLog(status); //Log the event.
        } else { //Else just apply brake.
            ems.startDeceleration();
            log.brakeLog(status); //Log the event to Brake log.
        }
    }

    public void brakeReleased() { //
        ems.stopDeceleration(); //Stop braking, maintain current speed.
    }

    public void acceleratorApplied() { //Begin to speed up vehicle.
        ems.startAcceleration();
    }

    public void acceleratorReleased() {
        ems.stopAcceleration(); //Stop acceleration, maintain current speed.
    }

    public void turnOnCar() {
        ems.turnOnCar();
        log.engineLog("On");
    }

    public void turnOffCar() {
        ems.turnOffCar();
        log.engineLog("Off");
    }
    //Print methods below to print the logs to text files. All are called upon entering correct password to get log.
    public void printLogsEngine(String fileName) {
        try {
            FileWriter fstream = new FileWriter(fileName);
            BufferedWriter out = new BufferedWriter(fstream);

            int count = 0;
              int recordsToPrint = log.getEngineLog().size(); //get the size of the hashmap to print every record to the text
file.
                Iterator<Entry<String, String>> it = log.getEngineLog().entrySet().iterator(); //Create iterator to loop over
HASHmap.
            while (it.hasNext() && count < recordsToPrint) { //Use iterator to loop over all entries.
                Map.Entry<String, String> pairs = it.next();
                   out.write(pairs.getKey() + ": " + pairs.getValue() + "\n"); //Print the Key which is the entry time and date
as well as the event that took place.
                count++;
            }
            out.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

```
  public void printLogsSpeed(String fileName) {
    try {
      FileWriter fstream = new FileWriter(fileName);
      BufferedWriter out = new BufferedWriter(fstream);

      int count = 0;
      int recordsToPrint = log.getSpeedLog().size();
      Iterator<Entry<String, Double>> it = log.getSpeedLog().entrySet().iterator();
      while (it.hasNext() && count < recordsToPrint) {
        Map.Entry<String, Double> pairs = it.next();
        out.write(pairs.getKey() + ": " + Double.toString(pairs.getValue()) + "\n");
        count++;
      }
      out.close();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  public void printLogsBrake(String fileName) {
    try {
      FileWriter fstream = new FileWriter(fileName);
      BufferedWriter out = new BufferedWriter(fstream);

      int count = 0;
      int recordsToPrint = log.getBrakeLog().size();
      Iterator<Entry<String, Status>> it = log.getBrakeLog().entrySet().iterator();
      while (it.hasNext() && count < recordsToPrint) {
        Map.Entry<String, Status> pairs = it.next();
        out.write(pairs.getKey() + ": Brake Activated" + "\n");
        count++;
      }
      out.close();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  public void printLogsCC(String fileName) {
    try {
      FileWriter fstream = new FileWriter(fileName);
      BufferedWriter out = new BufferedWriter(fstream);

      int count = 0;
      int recordsToPrint = log.getCCLog().size();
      Iterator<Entry<String, String>> it = log.getCCLog().entrySet().iterator();
      while (it.hasNext() && count < recordsToPrint) {
        Map.Entry<String, String> pairs = it.next();
        out.write(pairs.getKey() + ": " + pairs.getValue() + "\n");
        count++;
      }
```

```
            out.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```

EMS.Java

```
/*
            Created By: Jared Follet, Edward Yaroslavsky, Adam Undus and Anthony Bruno
            Modified On: 5/12/2020

*/
package miccota;

import java.util.Timer;
import java.util.TimerTask;

public class EMS extends TimerTask { //Extends TimerTask class
//A task that can be scheduled for one-time or repeated execution by a Timer.
    private double speed;
    private double targetSpeed;
    public final int ACCELERATION = 5; // us^2
    private Timer timer; //New timer instance.
    private final int REFRESH = 200; // ms
    private boolean acceleratorEngaged;
    private boolean brakeEngaged;
     private final double ACCELERATION_PER_REFRESH = ACCELERATION * ((double) REFRESH / 1000.0);
//declare vehicle accerlation rate
    private final double DRIFT_SLOW = 0.2;
    private boolean isCarOn; //Var to hold status of cars current state.
    private boolean ccOn; //Holds CCMs current state.
    // private Log logger;

    public EMS() { //Constructor.
        super(); //Call TimerTask constructor.
                        //Declare default variable values.
        speed = 0;
        targetSpeed = 0;
        timer = new Timer();
        acceleratorEngaged = false;
        brakeEngaged = false;
        isCarOn = false;
        ccOn = false;
        startAutoAcceleration();
        // TimerTask task = new EMS();

        // logger = new Log();
        // logger.engineLog("Engine Started.");
        // logger.speedLog(speed);
    }

    public boolean requestSpeed(double newSpeed) { //Method that updates the target speed.
        // logger.engineLog("New Target Speed Requested by CC: " + newSpeed);
        ccOn = true;
        targetSpeed = newSpeed;
        return true;
    }
```

```
  public double getSpeed() { //Getter for current ems speed.
    return speed;
  }

  private void startAutoAcceleration() { //Begin timer scheduling to be used to mimic vehicle acceleration.
    // logger.engineLog("Starting to Maintain Target Speed");
    System.out.println("starting task");
    timer.scheduleAtFixedRate(this, 0, (long) REFRESH);
  }

  public void stopAutoAcceleration() { //Set acceleration to off.
    ccOn = false;
  }

  @Override
  public void run() { //Run the timer, updating the speed according to the refresh rate.
    if (brakeEngaged) { // brake takes presedence over acceleration.
      this.speed -= ACCELERATION_PER_REFRESH;
      if (this.speed <= 0) {
        this.speed = 0;
      }
      return;
    } else if (acceleratorEngaged) { //if accelerating, increase speed.
      this.speed += ACCELERATION_PER_REFRESH;
      if (this.speed >= 150) {
        this.speed = 150;
      }
      return;
    }

    if (ccOn) { //IF CCM is engaged.
        //System.out.println("Here");
      if ((int) speed == (int) targetSpeed) {
//check if the new speed request is equal to ems current speed. If so, do nothing.
        return;
      } else if (targetSpeed > speed) {
        this.speed += ACCELERATION_PER_REFRESH; //begin vehicle acceleration up to target speeed.
      } else {
        this.speed -= ACCELERATION_PER_REFRESH; //begin vehicle deceleration down to target speed.
      }
    } else {
        if (speed >= 150) { //Cant go over 150mph
                this.speed = 150;
                return;
        }
        else if (speed > 0) {
        this.speed -= DRIFT_SLOW;
        return;
      }
      else {
        this.speed = 0;
        return;
      }
    }
```

```
        // logger.speedLog(speed);
    }

    public void startAcceleration() { //Engage Acceleration
        // logger.engineLog("Manual Acceleration Started.");
        acceleratorEngaged = true;
    }

    public void stopAcceleration() { //Disengage Acceleration, maintains current speed.
        // logger.engineLog("Manual Acceleration Stopped.");
        acceleratorEngaged = false;
    }

    public void startDeceleration() { //Engage Deceleration, decreasing speed.
        // logger.engineLog("Manual Deceleration Started.");
        brakeEngaged = true;
    }

    public void stopDeceleration() { //Stop decreasing speed, maintain current.
        // logger.engineLog("Manual Deceleration Stopped.");
        brakeEngaged = false;
    }

    public boolean turnOnCar() { //Turns Car/EMS on.
        if (isCarOn) {
            System.out.println("Cannot turn car on when it is already on.");
            return true;
        }
        isCarOn = true;
        return isCarOn;

    }

    public boolean turnOffCar() { //Turns Car/EMS off.
        if (!isCarOn) {
            System.out.println("Cannot turn car off when it is already off.");
            return false;
        }
        isCarOn = false;
        ccOn = false;
        acceleratorEngaged = false;
        return isCarOn;

    }

    public boolean isCarOn() { //returns if the car is on. True if On, false if Off.
        return isCarOn;
    }

}
```

Log.Java

```
/*
	Created By: Jared Follet, Edward Yaroslavsky, Adam Undus and Anthony Bruno
	Modified On: 5/12/2020

*/
package miccota;

import java.util.*;
import java.text.SimpleDateFormat;

public class Log {
	//Create indpendent hashmaps for 4 seperate logs. Helps organize and enables efficient debugging.
	private HashMap<String, String> engineLog;
	private HashMap<String, Double> speedLog;
	private HashMap<String, Status> brakeLog;
	private HashMap<String, String> ccLog;

	public Log() { //Constructor
		engineLog = new HashMap<String, String>();
		ccLog = new HashMap<String, String>();
		speedLog = new HashMap<String, Double>();
		brakeLog = new HashMap<String, Status>();
	}

	public String getDate() { //Get the current time and date in the format below. Example. 12/05/2020
02:39:12
		SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss"); //set the
format.
		Date date = new Date(); //request the current time and date.
		return formatter.format(date); //format the date to desired output and return.
	}

	public void CClog(Status status) { //CCM Log method that handles the status event changes.
		String date = getDate();
		String logMsg = status.toString();
		this.ccLog.put(date, logMsg);
	}

	public void CClog(Status status, String message) { //Other CCM Log method that handles the status event
changes with desired update/exception messages.
		String date = getDate();
		String logMsg = status.toString() + ": " + message;
		this.ccLog.put(date, logMsg);
	}

	public void engineLog(String message) {//Engine Log method that handles the EMS event changes.
		String date = getDate();
		this.engineLog.put(date, message);
	}

	public void speedLog(Double speed) {//Speed Log method that handles any changes in speed.
```

```
			String date = getDate();
			this.speedLog.put(date, speed);
		}

		public void brakeLog(Status status) { //Brake Log method that handles any brake presses.
			String date = getDate();
			this.brakeLog.put(date, status);
		}
		//Getters for respective logs.
		public HashMap<String, String> getEngineLog() {
		return engineLog;
    }

		public HashMap<String, Double> getSpeedLog() {
		return speedLog;
    }

		public HashMap<String, Status> getBrakeLog() {
		return brakeLog;
    }

		public HashMap<String, String> getCCLog() {
		return ccLog;
    }
}
```

Status.Java

```
/*
        Created By: Jared Follet, Edward Yaroslavsky, Adam Undus and Anthony Bruno
        Modified On: 5/12/2020
*/
package miccota;

public enum Status {
   OFF, READY, ENGAGED
}
```

UI.Java

```
/*
        Created By: Jared Follet, Edward Yaroslavsky, Adam Undus and Anthony Bruno
        Modified On: 5/12/2020

*/
package miccota;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JSlider;
import javax.swing.JTextArea;

public class UI {

   private JFrame frame; //Entire program UI.
   private JLabel lblstat = new JLabel("CC: Off"); //Default CCM status to off.
   private JLabel currspeed = new JLabel("0"); //Default engine speed to 0.
   private static JLabel speedo = new JLabel("Current Speed: 0.0 MPH"); //Default CCM targetSpeed to 0
   private JSlider speed; //Slider to adjust speed of engine manually.
   private JTextArea console; //CCM UI Console.
   private static CruiseControl cc = new CruiseControl(); //New instance of CCM class.
   private static boolean brakeDown = false;
   private static boolean acceleratorDown = false; //Neither brake nor accelator applied.
    private static JPasswordField pass = new JPasswordField(10); //Password for get log function to enforce security
policy.
```

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

        while (true) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // System.out.println("Speedometer Run Time " + count);
            double speedd = cc.getSpeed(); //Constantly update speed.
            double scale = Math.pow(10, 1);
                    speedo.setText("Current Speed: " + Double.toString(Math.round(speedd * scale) / scale) + " MPH");
//Display CCM current speed, adjusting for new values.
        }
    }

    /**
     * Create the application.
     */
    public UI() {
        initialize();
    }

     private boolean validSpeed() { //Checks if the desired speed for the CCM is between its correct operating speeds,
25mph and 150mph.
        return cc.getSpeed() >= 25 && cc.getSpeed() <= 150;
    }

    /**
     * Initialize the contents of the frame.
     */
    private void initialize() {
                              //Format UI
        frame = new JFrame();
        frame.setTitle("MICCOTA Cruise Control"); //Set UI Window title.
        frame.setBounds(100, 100, 1024, 576);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setBackground(Color.black);
        frame.setPreferredSize(new Dimension(1024, 576));
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
        frame.getContentPane().setLayout(null);
                    //JFrame icon in top left corner. Set to a car for relevance.
        ImageIcon img = new ImageIcon("tcicon.jpeg");
        frame.setIconImage(img.getImage());
              //Create the console/text area to visually display current status and actions from CCM.
        console = new JTextArea();
        console.setBounds(650, 100, 275, 200);
        console.setEditable(false);
        frame.getContentPane().add(console);
        //Adjust CCM display properties.
        lblstat.setBounds(750, 31, 200, 20);
        lblstat.setFont(new Font("Monospaced", Font.PLAIN, 20));
        frame.getContentPane().add(lblstat);
                    //Adjust current speed display properties
        currspeed.setBounds(60, 85, 300, 100);
        currspeed.setFont(new Font("Monospaced", Font.BOLD, 20));
        frame.getContentPane().add(currspeed);
                    //Adjust CCM current speed display properties
        speedo.setBounds(60, 160, 300, 100);
        speedo.setFont(new Font("Monospaced", Font.BOLD, 20));
        frame.getContentPane().add(speedo);
                    //Create CCM Actiate button, lower left of UI
        JButton On = new JButton("Activate");
        On.setBounds(50, 400, 175, 100);
        On.setFont(new Font("Monospaced", Font.BOLD, 20));
        On.setBackground(Color.GREEN);
        frame.getContentPane().add(On);
                    //Create CCM Deactivate button, right of Activate button.
        JButton Off = new JButton("Deactivate");
        Off.setBounds(275, 400, 175, 100);
        Off.setFont(new Font("Monospaced", Font.BOLD, 20));
        Off.setBackground(Color.red);
        frame.getContentPane().add(Off);
                    //Create CCM Increase Speed by 1 button, above activation controls.
        JButton Up = new JButton("Increase Speed by 1");
        Up.setBounds(100, 300, 150, 50);
        Up.setFont(new Font("Monospaced", Font.BOLD, 10));
        Up.setBackground(Color.lightGray);
        frame.getContentPane().add(Up);
                    //Create CCM Decrease Speed by 1 button, above activation controls.
        JButton Down = new JButton("Decrease Speed by 1");
        Down.setBounds(250, 300, 150, 50);
        Down.setFont(new Font("Monospaced", Font.BOLD, 10));
        Down.setBackground(Color.LIGHT_GRAY);
        frame.getContentPane().add(Down);
                    //Create UI Turn Car On button, bottom right
        JButton CarOn = new JButton("Turn Car On");
        CarOn.setBounds(625, 425, 150, 50);
        CarOn.setFont(new Font("Monospaced", Font.BOLD, 15));
        frame.getContentPane().add(CarOn);
                    //Create UI Turn Car Off button, right of Turn Car On Button
        JButton CarOff = new JButton("Turn Car Off");
        CarOff.setBounds(800, 425, 150, 50);
        CarOff.setFont(new Font("Monospaced", Font.BOLD, 15));
        frame.getContentPane().add(CarOff);
                    //Create UI Turn Car On button, above car on/off buttons.
        JButton Brake = new JButton("Brake");
        Brake.setBounds(625, 350, 150, 50);
        Brake.setFont(new Font("Monospaced", Font.BOLD, 15));
        Brake.setBackground(Color.white);
        frame.getContentPane().add(Brake);
                    //Create UI Get Log Button, below turn car on/off buttons.
        JButton getlog = new JButton("Get Logs");
        getlog.setBounds(740, 500, 100, 30);
        getlog.setFont(new Font("Monospaced", Font.BOLD, 14));
        getlog.setBackground(Color.white);
        frame.getContentPane().add(getlog);

        pass.setBounds(845, 505, 100, 20); //add password field.
        frame.getContentPane().add(pass); //Grab users inputted password to be handled and authorized(or not).
                    //Create Accelerator Button for Car.
        JButton Accelerator = new JButton("Accelerator");
        Accelerator.setBounds(800, 350, 150, 50);
        Accelerator.setFont(new Font("Monospaced", Font.BOLD, 15));
        Accelerator.setBackground(Color.white);
        frame.getContentPane().add(Accelerator);
                    //Create slider properties for setting vehicle speed outside of CCM.
        speed = new JSlider(JSlider.HORIZONTAL, 0, 150, 0);
        speed.setBounds(100, 100, 300, 100);
        speed.setMajorTickSpacing(25);
        speed.setMinorTickSpacing(1);
        speed.setPaintTicks(true);
        speed.setPaintLabels(true);
        speed.addChangeListener(new SliderListener());
        frame.getContentPane().add(speed);

        // LISTENERS

        On.addActionListener(new ActionListener() { //Lisens for the CCM activate button.
            public void actionPerformed(ActionEvent e) {
                if (!cc.isCarOn()) {
                    console.setText("Car must be on to enable CC.");
                }
                if (!validSpeed()) {
                    console.setText("Speed Must be between 25 and 150");

                } else if (cc.getStatus() == Status.READY) {
                    cc.enable();
                    lblstat.setText("CC: On");
                    console.setText("Status is Engaged");
                } else {
                    console.setText("CC is already Engaged");
                }

            }
        });
        Off.addActionListener(new ActionListener() { //Lisens for the CCM Deactivate button.
            public void actionPerformed(ActionEvent e) {
                if (cc.getStatus() == Status.ENGAGED) {
                    cc.disable();
                    lblstat.setText("CC: Off");
                    console.setText("Cruise Control deactivated");
                } else {
                    console.setText("Cruise Control is already deactivated");
                }
            }
        });

        Up.addActionListener(new ActionListener() { //Lisens for the Increase Speed By 1 Button press.
            public void actionPerformed(ActionEvent e) {
                if (!cc.isCarOn()) {
                    console.setText("Turn the car on first");
                } else if (cc.getStatus() != Status.ENGAGED) {
                    console.setText("The Cruise Control must be Engaged");
                } else if (cc.getSpeed() >= 150) {
                    console.setText("Cannot increase speed past 150 MPH");
                } else {
                    cc.increaseSpeed();
                    console.setText("Speed Increased.");
                }

            }
        });

        Down.addActionListener(new ActionListener() { //Lisens for the Decrease Speed By 1 Button press.
            public void actionPerformed(ActionEvent e) {
                if (!cc.isCarOn()) {
                    console.setText("Turn the car on first");
                } else if (cc.getStatus() != Status.ENGAGED) {
                    console.setText("The Cruise Control must be Engaged");
                } else if (cc.getSpeed() <= 25) {
                    console.setText("Cannot decrease speed past 25 MPH");
                } else {
                    cc.decreaseSpeed();
                    console.setText("Speed Decreased.");
                }
            }
        });

        Brake.addActionListener(new ActionListener() { //Lisens for the Brake Button press.
            public void actionPerformed(ActionEvent e) {
                if (cc.isCarOn() == false) {
                    console.setText("Car is off. Cannot use brake");
                } else {
                    if (brakeDown) {
                        brakeDown = false;
                        cc.brakeReleased();
                    } else {
                        brakeDown = true;
                    cc.brakeApplied();
                    lblstat.setText("CC: Off");
                }
            }
        }
});

Accelerator.addActionListener(new ActionListener() { //Lisens for the Accelerator Button press.
    public void actionPerformed(ActionEvent e) {
        if (cc.isCarOn() == false) {
            console.setText("Car is off. Cannot use Accelerator");
        } else {
            if (acceleratorDown) {
                acceleratorDown = false;
                cc.acceleratorReleased();
                lblstat.setText("CC: On");
            } else {
                acceleratorDown = true;
                cc.acceleratorApplied();
                lblstat.setText("CC: Off");
            }
        }
    }
});

CarOn.addActionListener(new ActionListener() { //Lisens for the Turn Car On Button press.
    public void actionPerformed(ActionEvent e) {
        if (cc.isCarOn()) {
            console.setText("Car is already on");
        } else {
            cc.turnOnCar();
            console.setText("Car is on\n");
        }
    }
});

CarOff.addActionListener(new ActionListener() { //Lisens for the Turn Car Off Button press.
    public void actionPerformed(ActionEvent e) {
        if (cc.isCarOn() == false) {
            console.setText("Car is already off");
        } else if (cc.getSpeed() != 0) {
            console.setText("Cannot turn off the car while in motion");
        } else {
            currspeed.setText(Integer.toString(0));
            speed.setValue(0);
            cc.disable();
            cc.turnOffCar();
            console.setText("Car Turned Off.");
            cc.setTargetSpeed(0);
        }
    }
});

getlog.addActionListener(new ActionListener() { //Listens for the Get Log Button Press.
                public void actionPerformed(ActionEvent e) {
                    String CP = "miccota"; //Correct Password.
                    String check = new String(pass.getPassword());

                     if (CP.equals(check)) { //Check if the user inputted password matches the correct password, if so, print
logs to text files.
                    cc.printLogsEngine("engine.txt");
                    cc.printLogsSpeed("speed.txt");
                    cc.printLogsBrake("brake.txt");
                    cc.printLogsCC("cc.txt");
                    console.setText("Log Files Printed");
                } else { //If passwords do not match, reject users request.
                    console.setText("Incorrect admin password. Cannot print logs.");
                }

            }
        });

    }

    class SliderListener implements ChangeListener { //Seperate Class listens for changes/activity with the slider.
        public void stateChanged(ChangeEvent e) {
            int speednum = speed.getValue(); //get the new value of the slider.
            currspeed.setText(Integer.toString(speednum));
            if (cc.getStatus() == Status.ENGAGED) { //Check if the CCM module is engaged.
                if (speednum >= 25 && speednum <= 150 && cc.isCarOn()) { //Confirm targetSpeed is in valid operating
range.
                    cc.setTargetSpeed(speednum); //Adjust speed
                    lblstat.setText("CC: On"); //Update CCM state display.
                } else {
                    lblstat.setText("Invalid Speed");
                }
            } else {
                console.setText("Cruise control must be enganged.");
            }
        }
    }

}
```

7. Test Cases for Requirements

Functional Requirements:

2.4. Input Requirements:

2.4.1. The Cruise Control System (CCS) shall activate upon user input (toggle on/off switch).

- **TEST CASES:**
  1. Implemented Activate/Deactivate buttons which passed activation tests for both initial states.
  2. Pressing activate when status is ready will engage CC;
  3. pressing activate when status is engaged will do nothing, and vice versa for deactivate.

2.4.2. CCS shall deactivate upon user input (toggle on/off switch).

- **TEST CASES:** See above. Passed.

2.4.3. CCS shall deactivate when brake is engaged.

- **TEST CASES:**
  1. Passed tests, CC will disengage when brake is enabled and,
  2. can be reactivated instantly if speed is greater than 25mph.

2.4.4. CSS shall momentarily deactivate while the accelerator is pressed, adjusting the speed, and reactivates when the new speed is set.

- **TEST CASES:**
  1. Implemented accelerator input from UI. Passed Tests.
  2. Accelerator takes precedence over CC in handling by the classes so CC is temporarily disabled when accelerator is pressed and re-enabled when accelerator is let go of.

2.4.5. CCS shall maintain the speed set by the user.

- **TEST CASES:**
  1. Passed Tests. Tested for various intervals of time, car will remain at speed set by CCS until an interrupt occurs such as brake press or accelerator.

2.4.6. CCS shall increase speed upon user input (dial) by 1 MPH.

- **TEST CASES:**
  1. Implemented Speed increase and Decrease buttons.
  2. Passed tests, buttons respectively increase and decrease speed by 1 mph.
  3. Pressing both buttons rapidly in sequence will effectively cancel out, as it takes time for the accelerator to speed up the car. Passed.

4. Tested pressing one button over and over again. Passed. Will function until the car moves outside the CCM operating speeds of >25mph and <150mph.

2.4.7. CCS shall decrease speed upon user input (dial) by 1 MPH.

- **TEST CASES:**
  1. See Above. Passed.

2.5. Output Requirements:

2.5.1. The CCS shall accept and displays acceptance of the driver's activation of the system.

- **TEST CASES:**
  1. Implemented a console to visually display messages from the CCM to the user.
  2. Tested various combinations of user actions such as: Car on, car off, car on, activate, deactivate, Activate, etc. CCS accepted users input when conditions met and rejected when conditions did not meet(Car was off, speed was not in operating range). Passed Tests.
  3. Console visual display succeeded in the above mentioned tests as well.

2.5.2. The CCS shall display the current use status of CCS to the driver.

- **TEST CASES:**
  1. Implemented a seperate status window for the state of CCS.
  2. See above tests, CCS Status display passed static and dynamic update tests.

2.5.3. The CCS shall display the current velocity to the speed gauge.

- **TEST CASES:**
  1. CCS has a dedicated speedometer, which correctly updates according to:
  2. Acceleration
  3. Deceleration
  4. Brake
  5. CCS Deactivation/Activation
  6. All tests passed.

2.5.4. CCS shall update state of use and velocity to the driver in a timely manner upon receiving state-changing input from the driver.

- **TEST CASES:**

1. Our EMS class has an implemented timer which feeds the CruiseControl class information at every refresh rate, such as
2. velocity and,
3. the state of the car
4. Tested only manual acceleration/deceleration vs. the aforementioned actions done by cruise control. The EMS feeds the CC the correct information with 99.99% reliability. Passed.

2.5.5. CCS shall display to the driver when it is available for activation.

- **TEST CASES:**

1. Handled by console/dedicated status window. Passed.

2.5.6. CCS shall have a log of anytime it is activated/deactivated or an update to the speed is made.

- **TEST CASES:**

1. Implemented in Log class. Output contains time, date and specific action. Passed.

3. Nonfunctional Requirements:

3.1. Reliability Requirements

3.1.1. CSS shall have a brake sensor to tell when the user presses the brakes.

- **TEST CASES:**

1. EMS class contains a dedicated public method to feeding CC information about the state of the brakes. This mimics a break.
2. Tested multiple rapid brake presses. Passed
3. Tested Break to CC Activate to Break tests. Passed.

3.1.2. CSS shall have a throttle sensor to tell when the user pushes the throttle.

- **TEST CASES:**

1. See above, implemented the same method but mimicking a throttle sensor. Passed tests.

3.1.3. CSS shall have a speedometer sensor to maintain the correct speed.

- **TEST CASES:**

1. Implemented features. Visual display available to user and backend updates provided by EMS available to CruiseControl.

3.1.4. CSS shall have a clock sensor for logging purposes.

- **TEST CASES:**

1. EMS inherits these features from the TimerTask class. Requirement handled by EMS and made available to CruiseControl

3.1.5. The CSS sensors shall receive accurate information 99.99% of the time.
- **TEST CASES:**
    1. Implemented features. Passed tests. Experienced No inaccurate results.

3.1.6. The CSS sensors shall respond accordingly 99.99% of the time.
- **TEST CASES:**
    1. Implemented features. Passed tests. Experienced No down time.

3.1.7. CSS software shall operate successfully 99.99% of the time.
- **TEST CASES:**
    1. Implemented features. Passed tests. Experienced No down time.

3.1.8. The margin for error for maintaining the user's speed should be within +/- 0.25 MPH.

3.2. Performance Requirements

3.2.1. CCS shall be ready for activation within 2 seconds after the engine startup (Design goal 0.5 second).
- **TEST CASES:**
    1. Design goal met, CCS ready for activation(changing from Ready and Engaged status) within .5 seconds upon proper criteria met. Passed.

3.2.2. CSS shall be highly responsive with the response times acting below 100 milliseconds.
- **TEST CASES:**
    1. Design goal met, CCS responds highly rapidly.

3.2.3. CSS shall operate within the range of speeds of 25 MPH and 150 MPH. Manufacturers can limit the max when producing.
- **TEST CASES:**
    1. Implemented manual policy enforcement.
    2. Tested CC below speeds of 25mph.
    3. Vehicle max speed 150, passed.

3.2.4. CSS shall continuously receive feedback from the sensors whenever changes occur within 1 second.
- **TEST CASES: Passed.**

3.3. Security Requirements

3.3.1. CCS shall sit on an isolated system within the car, not making any network connections, so securing from network attacks is unnecessary.

- **TEST CASES:**
    1. Tested by Jared and Bruno. No external APIs, REST, TCP/IP or HTTP requests are made by the module. Module is self contained within the 5 classes and a package library. Passed.

3.3.2. Log files shall only be read through the software and with an admin password.

- **TEST CASES:**
    1. Implemented a get log feature from UI. An administrator/developer password is required to gain access to the logs, thus enforcing the security policy.
    2. Only the correct password will allow access.
    3. Password is handled securely.

3.3.3. Log files shall only be written by the CCS itself.

- **TEST CASES:**
    1. CCS handles all commands for writing events to logs.
    2. No external module from the vehicle can write to the CCs Logs. Passed.

8. Issues

1. Currently, we do not know the form in which our sensors transmit data. We must test the sensors, interpret the signals and convert whatever format they use to a form understandable by our Java code. (Adam).
    - Resolved: Implemented Getters/Setters in our EMS class which simulate sensors and pass data to the CruiseControl class in an interpretable form.

2. Our data architecture needs to be more explicitly defined. Exactly what data will be stored, where it will be stored and how it will be stored needs to be decided on and recorded. How this is all done will influence how the CSS is programmed so this is an urgent issue to resolve. (Ed).
    - Resolved: In all, the program directly stores only the state of the respective controls(such as brake, accelerator, whether or not the car is on, whether CCS is on, etc.) as well as current speed and current CCM target speed. All of these properties are then written to logs which compiles all of this information into a document that can be used to efficiently debug.

3. Security is an important feature in the CCS and while we have outlined top level security requirements, we need to implement actual programming level security features to ensure the safety of the CCS and log data. Possible data encryption and implementation of password protected data write and reads must be organized and executed. (Jared + Bruno).
    - Resolved: Ensured there were no external APIs or TCP/IP connections. The CCS is self-contained across the 5 classes in a package and the TimerTask library. Furthermore, implemented a password protected log system that ensures only CCS admins can access the contents.

4. In our 5.2 diagram, we need to identify the object's exact attributes and methods. We need to define all sensors, attributes and methods, and clearly state what data the sensor has, and from where it gets it from. (Everyone).
    - Resolved: Re-designed the diagram.

5. While the backend code is the crux of this project, developing an intuitive UI is important to us in order to mimic a car and allow us and the user to get a much better representation of a car and the capabilities of our module.
    - Resolved: Spent countless hours adjusting UI to be the most convenient to use and implied logic via the location of certain buttons.

6. Error Handling needs to be extremely extensive and reliable. Need to minimize the occurrence of errors, but need to be thorough in our coverage of them.
    - Resolved: Some errors received a tag of ignore, such as activating cruise control after it had already been activated, or vice versa. These errors are simply caught and then ignored by the CCS.

Pledge

I pledge my honor that I have abided by the Stevens Honor System.

- Anthony Bruno, Jared Follet, Adam Undus, Edward Yaroslavsky