Cruise Control Software Document

By Team Boosted

Version 0.03

Albert Chen, Raj Gadhia, Simon Gao, and Jayson Infante

Table of Contents

1. Software Introduction Section

1.1 Executive Summary

Our modern software solution for cruise control will allow vehicles to have a speed-lock capability that can maintain the car's speed until the driver taps the brake pedal or turns off the system. Security issues with our solution will be deliberately investigated since cruise control is a mission-critical feature. The safety of our customers is our number one priority. We will utilize the agile development process along with an incremental build process to help us build a reliable and performant cruise control software.

1.2 Introduction

To begin, the stakeholders of our project include each team member. Our user base will include the customers of various vehicle manufacturers within the United States. As the solutions architects and developers of the cruise control software, we will hold ourselves to be deliberate and cautious in how we implement our solution in order to ensure that the feature can perform 99.9% of the time without failure.

Cruise control is a car feature that allows drivers to maintain a certain speed. So, instead of having to keep your foot on the gas and adjusting to maintain a certain speed limit, the cruise control feature automatically sets the car to your desired speed and maintains that speed until you press on the brake or the cancel button. The cruise control feature is extremely useful for long-distance drives, specifically on parkways and interstate highways and could even be used for short-distance drives. The feature makes the driving experience more relaxing for the driver since there is one less thing to concentrate on whilst driving. Cruise control can even reduce the chance of speeding, especially in average speed camera zones where it is easy to accidentally speed up and creep over the limit. It is possible that the driver may even save a little on fuel, as maintaining a set speed will always use less fuel than constantly accelerating and braking to change your speed repeatedly.

1.3 Safety Concerns

Although cruise control is useful, it also presents safety concerns. The cruise control software needs to account for various scenarios when it's used in production so that malfunctions have an extremely low to no chance of occurring. Therefore, when we develop our cruise control

feature we need to account for numerous potential bugs and malfunctions as it correlates to a matter of life and death in the real world.

As the cruise control is a mission-critical feature, and many cars are now having all digital interfaces and network reliant operations, security is of the utmost importance. Like mentioned before, there can not be any room for failure within the system itself, it must respond accurately and fail rarely. However, as cars get newer and introduce features like self-driving (i.e: Teslas), it opens a new threat to systems that wasn't an issue before. For example, if someone with malicious intent could gain access to your car's locking mechanism, they could very easily steal your car. Similarly, if that same person were to access the cars cruise control feature, they could do any number of things from letting the car drive off on its own, to even putting your life in danger. As such, during the process of creating the system, it should be rigorously tested against any system threats and continually be updated even after deployment so that any potential threats can be stopped before they become a reality.

1.4 Our Software Development Process

We will follow the agile software development process along with an incremental build process during the construction of the cruise control software. The tools that we will utilize during the development process include GitLab and a shared Google Drive folder amongst our team members. GitLab and Google Drive provides our team with project management tools that ease our collaboration efforts on documents, version control, and organization. Moreover, we will locally test our code before pushing to a shared repository on GitLab. As for communication, our team will use direct messaging and in-person meetings when necessary.

1.5 Proposed Functionality and Testing

As for the functionality of the cruise control feature, we need to simulate the tools that control it which include a brake, gas pedal, set button, cancel button, and increase/decrease speed button as these are the necessary features to simulate and test our cruise control software in a reliable environment. With this reliable environment, we will be able to adequately test our software's safety and reliability. We will utilize this testing environment every time we add a new feature or make any changes to the base code. Testing for the environment should be done as well, ensuring that our code does not falter because of any outside influences. If the new changes pass our test cases—which will be built off of user stories—then we will push the code to our shared repository and document our changes. We hope that in having a thorough testing process we can build a strong foundation for reliable and sustainable software.

2. Software Requirements Section

2.1. Input Requirements

2.1.1. General

2.1.1.1. The C.C. system shall accept electric power from the alternator.
2.1.1.2. The C.C system shall accept direct current from the car battery to support logging afterward.

2.1.2. Sensors

2.1.2.1. The C.C. system shall accept signals from sensors (clutch, brake, or any other environmental signals for approval of activation or cursing).
2.1.2.2. The C.C. system shall receive information from sensors about brake application continuously every 0.5 seconds.
2.1.2.3. The C.C. system shall receive the time and date from the car's clock every second.
2.1.2.4. The C.C. system shall deactivate upon the second press of the activation button or application of the brake.
2.1.2.5. The C.C. system shall ignore input from the gas pedal, rather, the vehicle simply speeding up for as long as it is pressed.
2.1.2.6. The C.C. system shall increase the speed by one mile per hour if the increase speed lever is pressed.
2.1.2.7. The C.C. system shall decrease the speed by one mile per hour if the decrease speed lever is pressed.

2.1.3. Engine Management System (EMS)

2.1.3.1. The C.C. system shall receive the current speed from Engine Management System (throttle) continuously every 1 second.
2.1.3.2. There shall be an automatic shutdown for the system following engine Shutdown.

-Shut down the C.C. system 30-60 seconds after the car is turned off.
-Must log diagnostic data of the car before the system shuts down.

2.1.4. Driver Input

2.1.4.1. The C.C. system shall accept the driver's activation of the system.
2.1.4.2. The C.C. system shall accept the driver's deactivation of the system.
2.1.4.3. The system will not activate if any of the following conditions are present: the transmission is in Park, Neutral, Reverse or Low gear, the vehicle is traveling less than 10 mph (or 90+), or there is no input from the vehicle speed sensor (which will also prevent the speedometer from working).

2.2 Output Requirements

2.2.1. User Feedback

2.2.1.1. The C.C. system shall provide visual feedback to the user about the setting of speed.

2.2.2. Graphical User Interface (Vehicle Interface)

2.2.2.1. The C.C. system shall provide an interface with a visual representation of the gas pedal, brake, and an activate/deactivate button for the cruise control feature.

2.3 Administration Requirements

2.3.1. Administrative Tasks

2.3.1.1. The C.C. system shall provide a physical interface for Technicians to access the unit.
2.3.1.2. The C.C. system software shall be configurable by the technician.
2.3.1.3. The C.C. system shall be able to log diagnostic data for the duration of the car's life.
2.3.1.4. The C.C. system shall log all C.C. state changes (events) with time.
2.3.1.5. The C.C. system shall store up to 1 TB of logs in memory for download to technicians later.
2.3.1.6. The data shall be downloadable through the interface.

2.4 Non-functional Requirements

2.4.1. Performance

2.4.1.1. The C.C. system shall accept all inputs within 2 seconds after the engine Starts.
2.4.1.2. The C.C. system shall be operable at any speed desired by the user.

2.4.2. Safety and Security

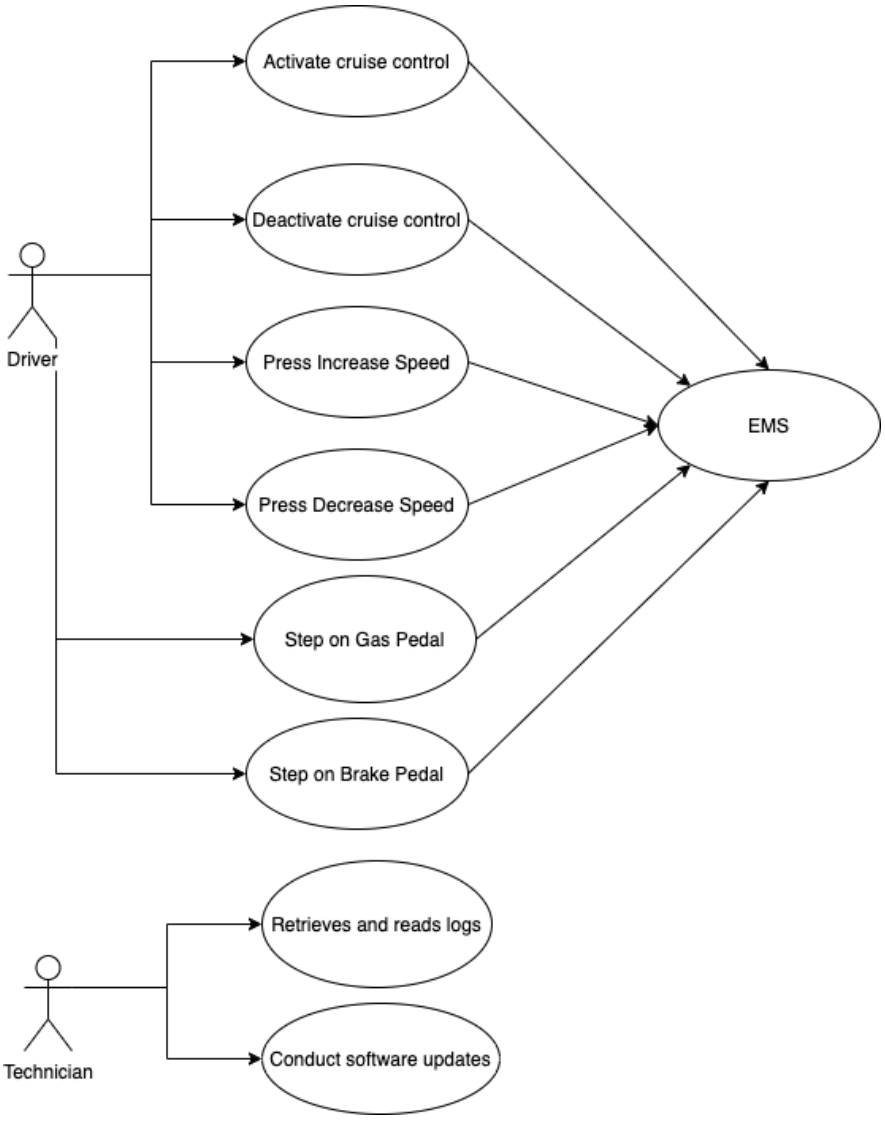
2.4.2.1. The C.C. system shall not provide any interface for remote access (such as Bluetooth).
2.4.2.2. The C.C. system shall only be accessible only to authorized dealers via a hard-wired interface and be password protected.
2.4.2.3. The C.C. System shall adhere to the safe and secure communication protocol specified by the manufacturer, industry, and governments.
2.4.2.4. The C.C. system cannot be activated or deactivated without the press of set/cancel button

2.4.3. Quality

2.4.3.1. The C.C. system hardware shall have a 4 nine (99.99%) availability.
2.4.3.2.The C.C. system software shall have a 5 nine (99.999%) availability.

3. Requirements Analysis Model Section

3.1 Use Case Diagram



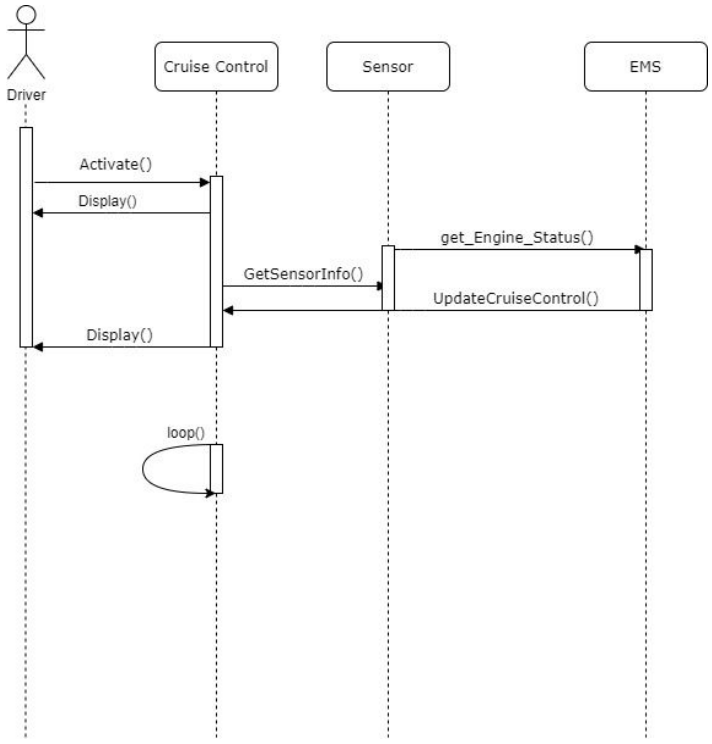
3.2 Use Cases/Sequence Diagrams

3.2.1 Use Case #1: User activates the cruise control feature.

Primary Actor: The driver of the vehicle.
Actor's Goal: To activate the car's cruise control feature.
Preconditions: The car/vehicle is already on.

1. The user presses the activation button for the car's cruise control (CC) feature.
2. Cruise control provides visual feedback that it is ready for activation.
3. The user requests cruise control to be set to the current speed.
4. CC requests values from sensors.
5. CC requests Engine Management System (EMS) to set its speed to its current speed.
6. CC provides visual feedback to the user to signify that CC is set properly and working.
7. CC detects any changes from the vehicle's sensors and requests adjusting speeds, or even deactivation of CC feature completely.
8. The car's acceleration continuously changes so that the speed remains constant.
9. Speed is continuously reported to the CC unit.

3.2.2 Sequence Diagram #1: User activates the cruise control feature.

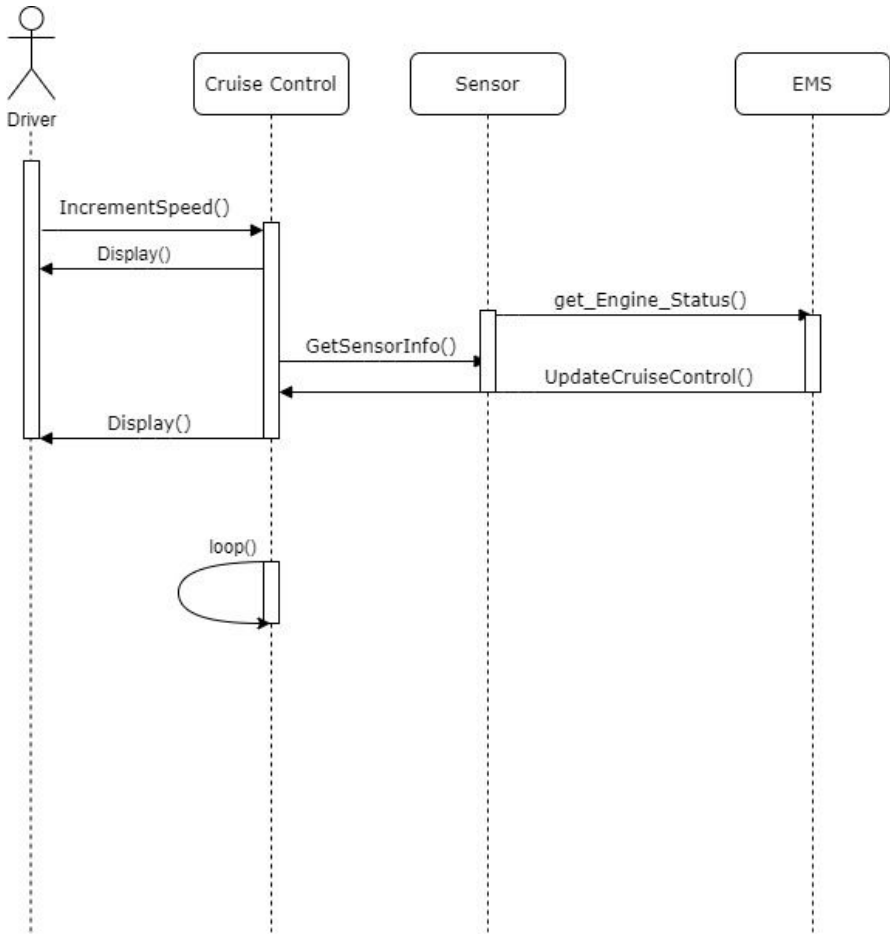


3.2.3 Use Case #2: The user presses the button to increase speed.

Primary Actor: The driver of the vehicle.
Actor's Goal: To increase the speed of the car while it is in cruise control.
Preconditions: The vehicle's cruise control feature is already active.

1. The user presses the button to increase the car's speed by 1.
2. CC provides visual feedback that the speed of the car has increased by 1.
3. CC requests values from sensors.
4. CC requests EMS to set its speed to (current speed + 1).
5. CC provides visual feedback to the user to signify that the new speed is set for CC.
6. CC detects any changes in sensors and requests adjustments in speed accordingly.
7. The car's acceleration continuously changes so that the new speed is maintained.
8. Speed is continuously reported to the CC unit.

3.2.4 Sequence Diagram #2: The user presses the button to increase speed.

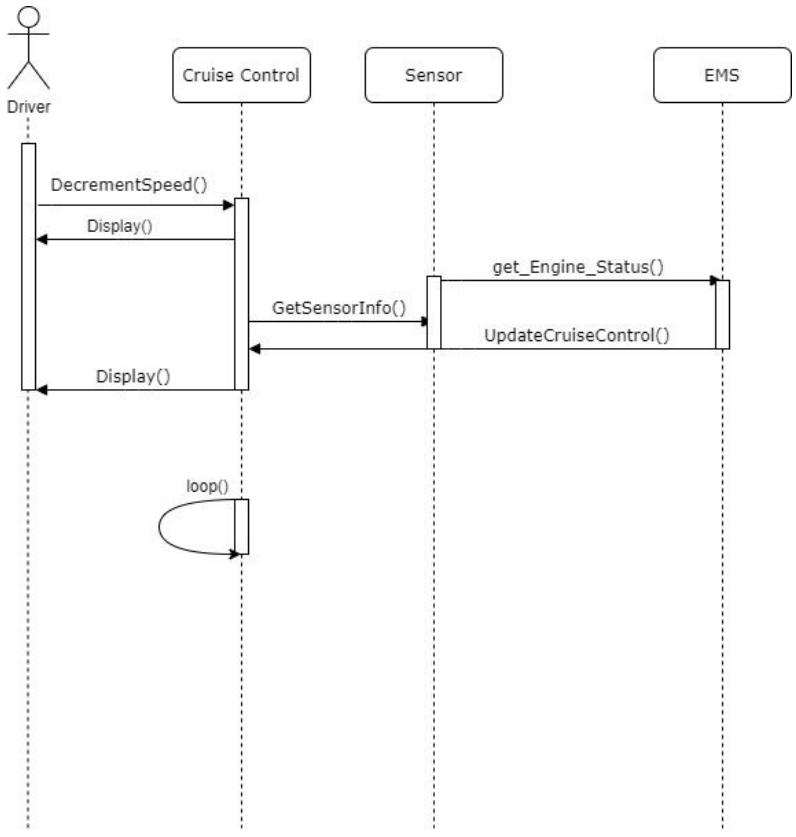


3.2.5 Use Case #3: The user presses the button to decrease speed.

Primary Actor: The driver of the vehicle.
Actor's Goal: To decrease the speed of the car while it is in cruise control.
Preconditions: The vehicle's cruise control feature is already active.

1. The user presses the button to decrease the car's current speed by 1.
2. CC provides visual feedback that the speed of the car has decreased by 1.
3. CC requests values from sensors.
4. CC requests EMS to set its speed to (current speed - 1).
5. CC provides visual feedback to the user to signify that the new speed is set for cc.
6. CC detects any changes in sensors and requests adjustments in speed accordingly.
7. The car's acceleration continuously changes so that the new speed is maintained
8. Speed is continuously reported to the CC unit

3.2.6 Sequence Diagram #3: The user presses the button to decrease speed.

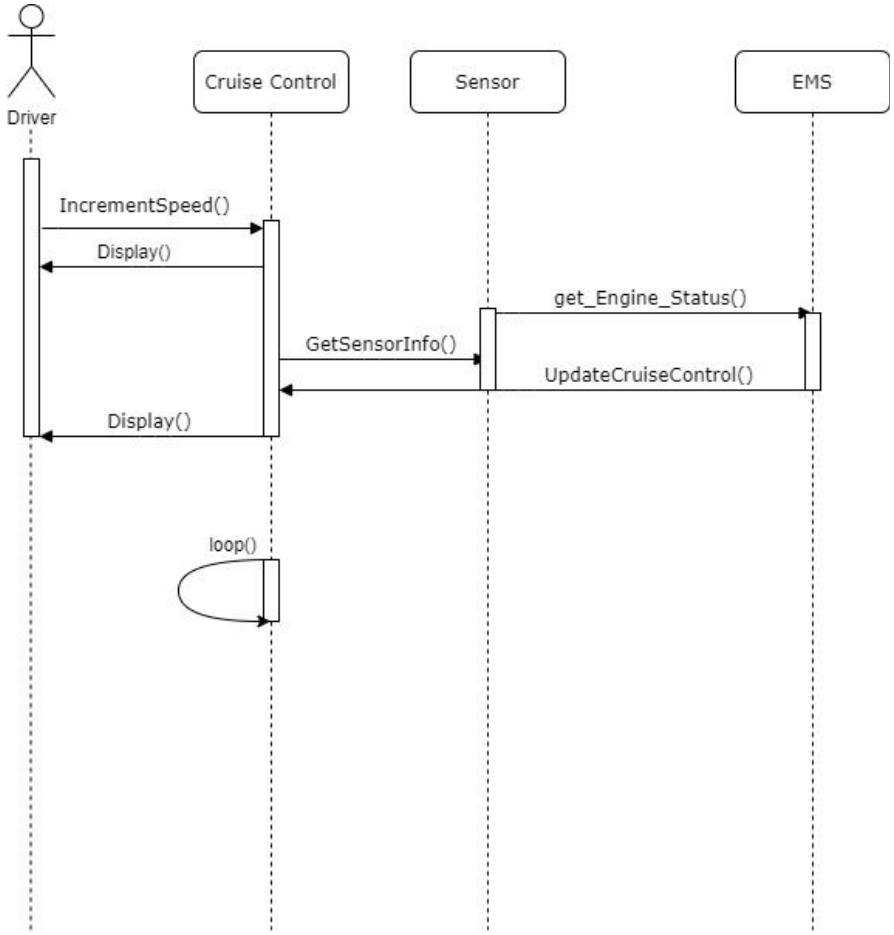


3.2.7 Use Case #4: The user presses the gas pedal.

Primary Actor: The driver of the vehicle.
Actor's Goal: To increase the speed of the car while it is in cruise control manually.
Preconditions: The vehicle's cruise control feature is already active.

1. The user presses on the gas pedal, or throttle.
2. CC provides visual feedback that the speed of the car has been increased manually.
3. CC requests values from sensors.
4. CC requests EMS to set its speed according to the values retrieved from the throttle.
5. CC provides visual feedback (blinking CC speed) to signify that it is following a new speed that is manually determined by the driver.
6. Speed is continuously reported to the CC unit.

3.2.8 Sequence Diagram #4: The user presses the gas pedal.

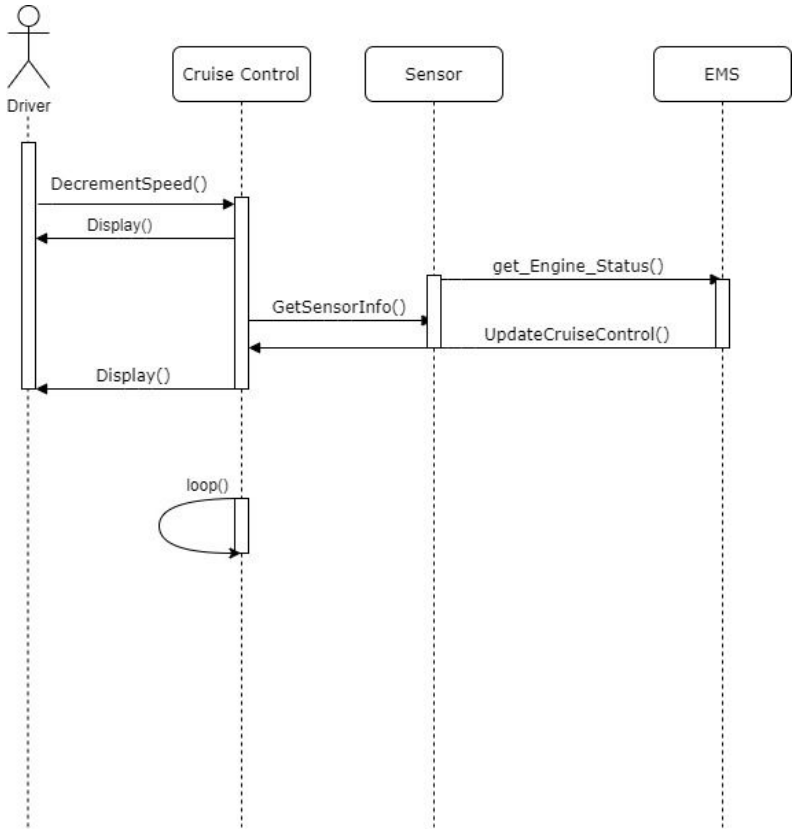


3.2.9 Use Case #5: The user releases the gas pedal.

Primary Actor: The driver of the vehicle.
Actor's Goal: To continue using CC after pressing on the gas pedal.
Preconditions: The vehicle's cruise control feature is already active.

1. The user releases the gas pedal.
2. CC provides visual feedback that the speed of the car will follow the previous CC speed.
3. CC requests values from sensors.
4. CC requests EMS to set its speed to the previous CC speed (by most likely decreasing the current speed).
5. CC provides visual feedback to signify that it is following the previous CC speed.
6. CC detects any changes to the sensors and requests speed adjustments accordingly to EMS.
7. The car's acceleration changes to maintain the CC speed.
8. Speed is continuously reported to the CC unit.

3.2.10 Sequence Diagram #5: The user releases the gas pedal.

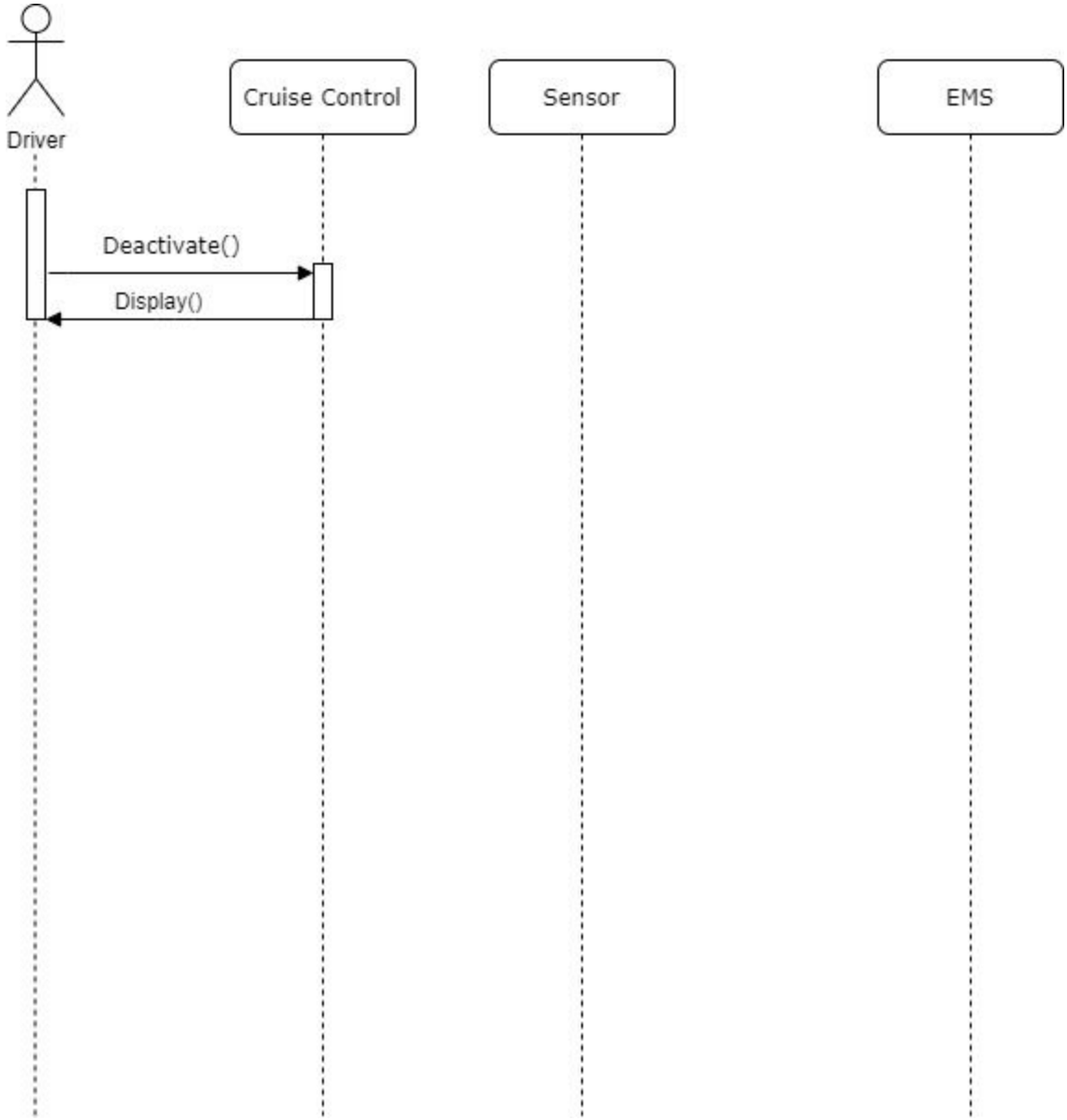


3.2.11 Use Case #6: The user deactivates cruise control.

Primary Actor: The driver of the vehicle.
Actor's Goal: To deactivate the cruise control feature.
Preconditions: The vehicle's cruise control feature is already active.

1. The user presses the deactivation button or presses on the brakes.
2. CC provides visual feedback that the feature is deactivated.
3. CC releases EMS from maintaining a certain speed.

3.2.12 Sequence Diagram #6: The user deactivates cruise control.

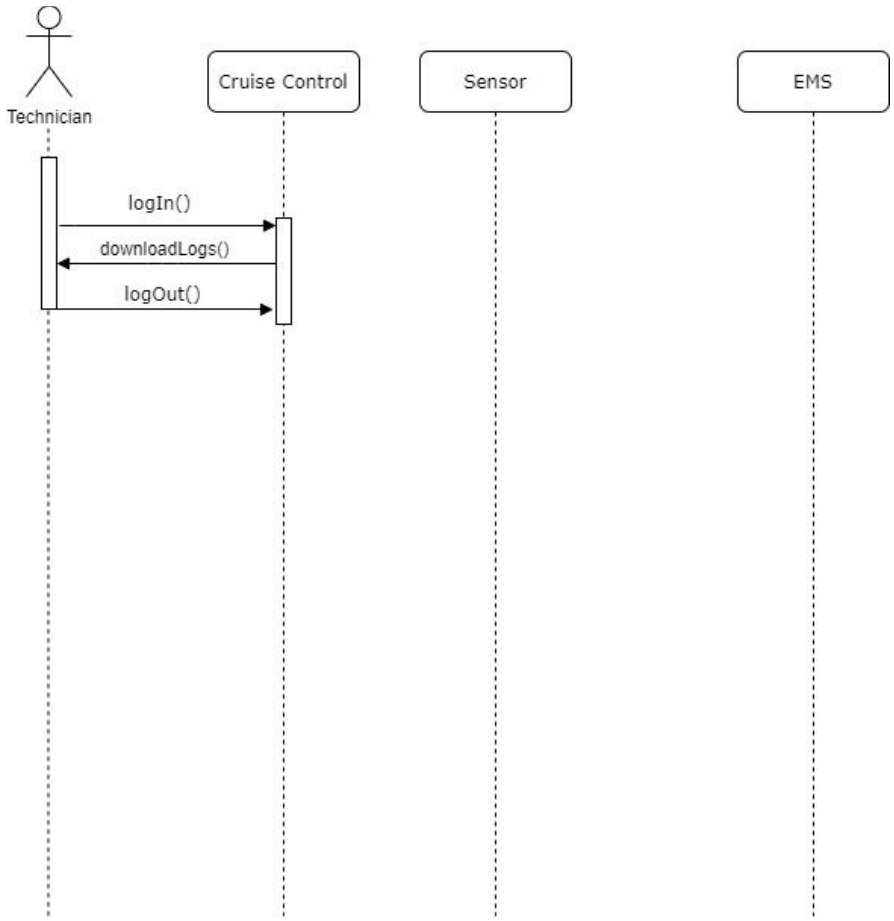


3.2.13 Use Case #7: The user retrieves the cruise control system logs.

Primary Actor: The technician of the vehicle's cruise control system.
Actor's Goal: To retrieve and read the logs for the cruise control system..
Preconditions: The car/vehicle is already on.

1. The user enters their username and password in order to log into the cruise control system.
2. The user downloads the log information within the cruise control system.
3. The user logs out of the cruise control system.

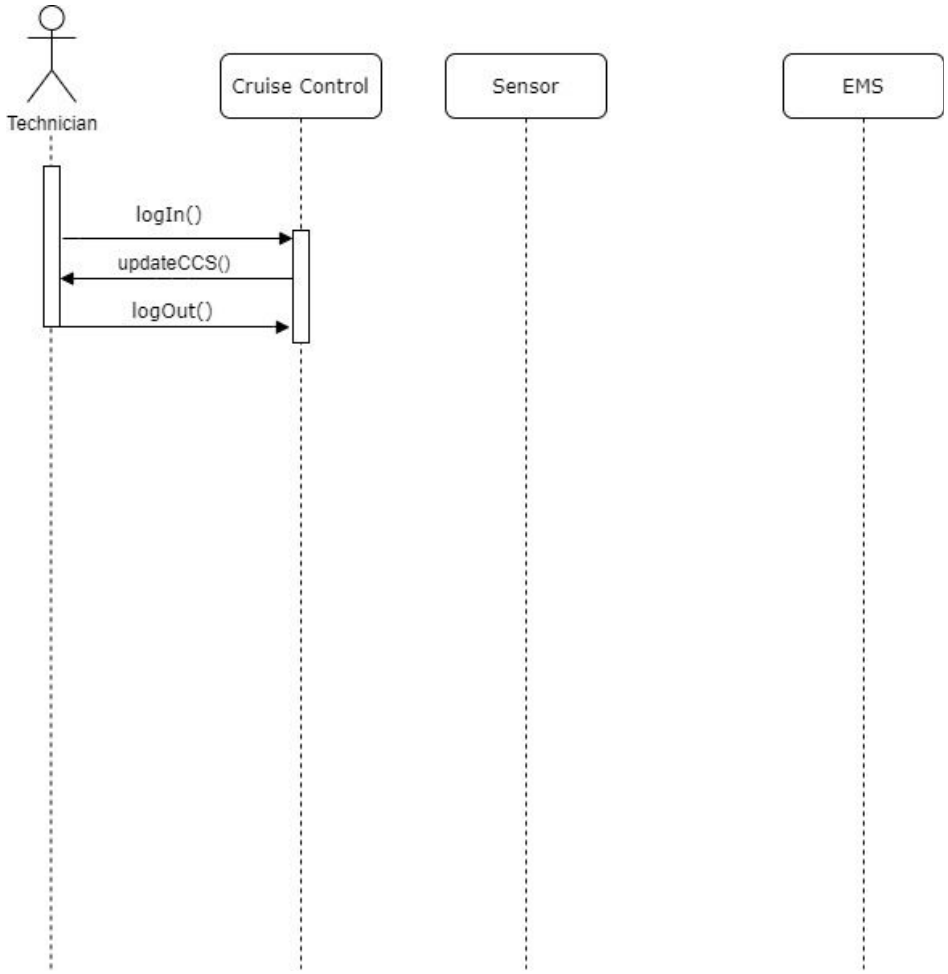
3.2.14 Sequence Diagram #7: The user retrieves the cruise control system logs.



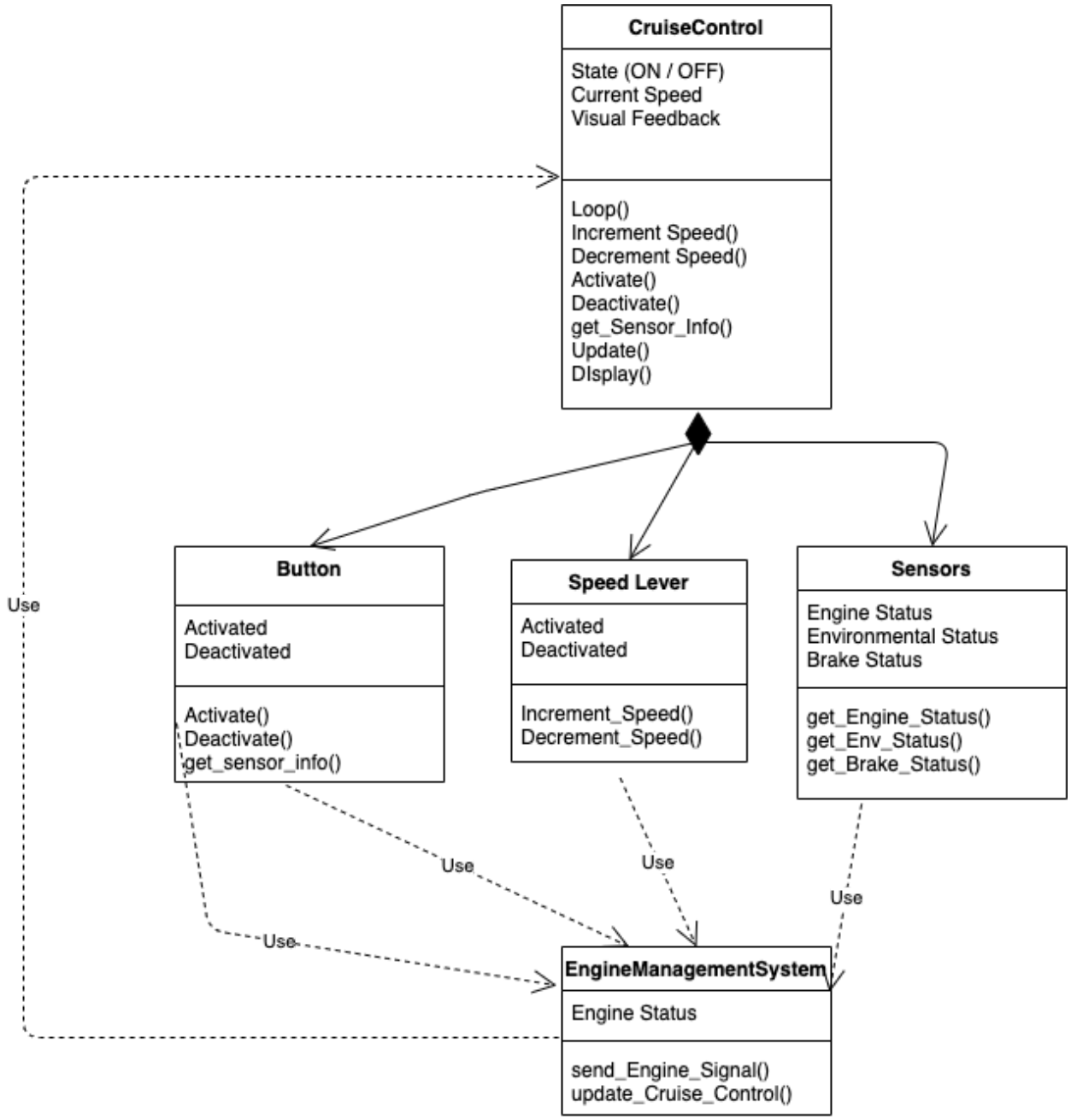
3.2.15 Use Case #8: The user conducts a software update on the cruise control system.

Primary Actor: The technician of the vehicle's cruise control system.
Actor's Goal: To conduct a software update on the cruise control system..
Preconditions: The car/vehicle is already on.
The user enters their username and password in order to log into the cruise control system.
The user initiates a software update to the cruise control system via external hardware.
The user logs out of the cruise control system.

3.2.16 Sequence Diagram #8: The user conducts a software update on the cruise control system.



3.3 UML Class-Based Modeling



3.4 UML CRC Model Index Card

Class: CruiseControl Main module that will run continuously until deactivated	
Responsibility:	**Collaborator:**
Activates / Turns on	Activation button
Maintains speed	EMS / Throttle Sensors: Tire Pressure, Environment like slope
Increase / Decreases Current Speed	Increase / Decrease Speed Button
Shuts off / Deactivates	Brake Pedal Deactivation Button Engine Sensors: - Detects if the engine shuts off
Displays Current Speed / Visual Feedback	Dashboard

Class: Sensor Sensors responsible for information regarding environment and status of car/engine	
Responsibility:	**Collaborator:**
Returns information regarding engine	Engine Cruise Control
Returns information regarding environment values	EMS Cruise Control Tires
Returns information regarding brake status	Brake Pedal Cruise Control

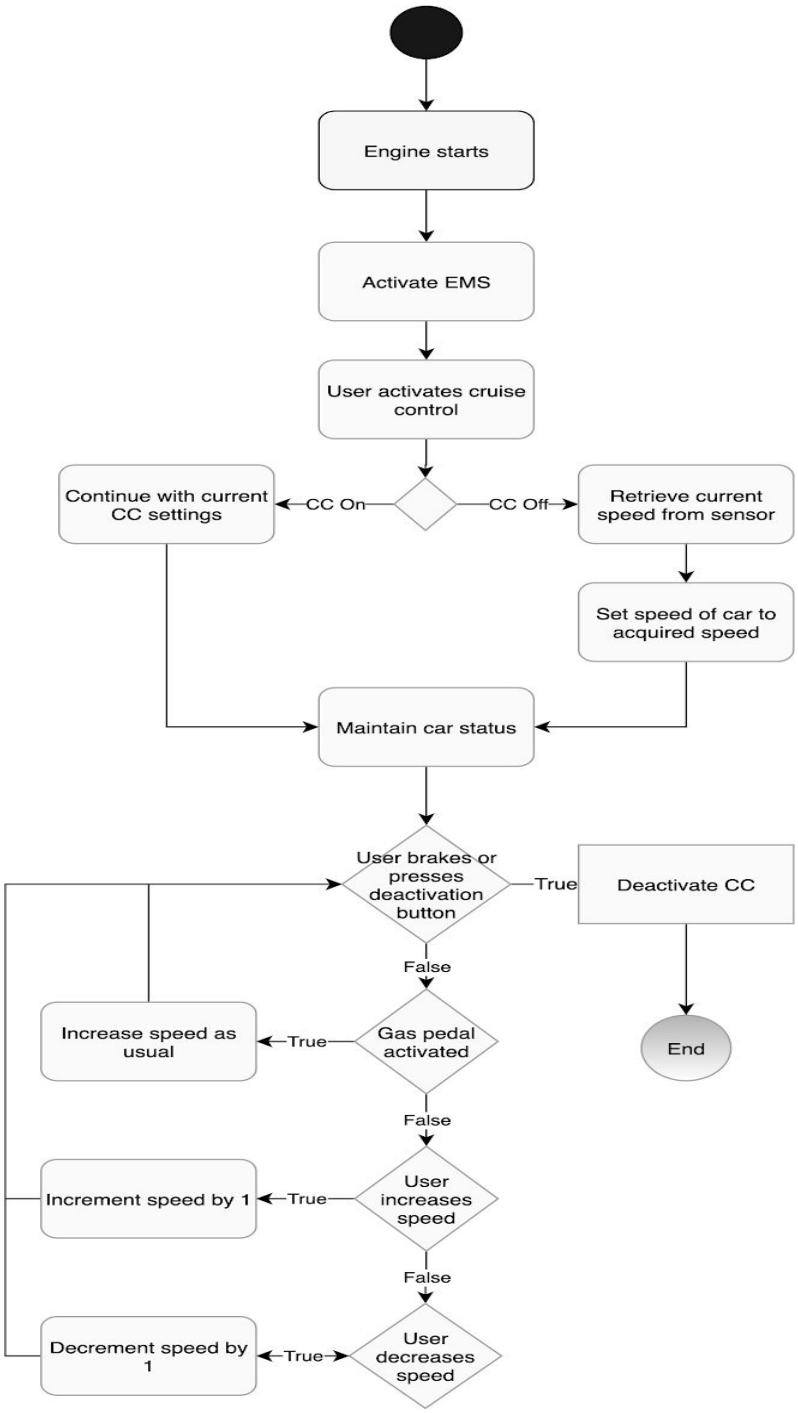
	Engine

Class: SpeedLever Class responsible for changing speed of cruise control by +/- 1mph	
Responsibility:	**Collaborator:**
Increases Speed	Cruise Control Dashboard
Decreases Speed	Cruise Control Dashboard

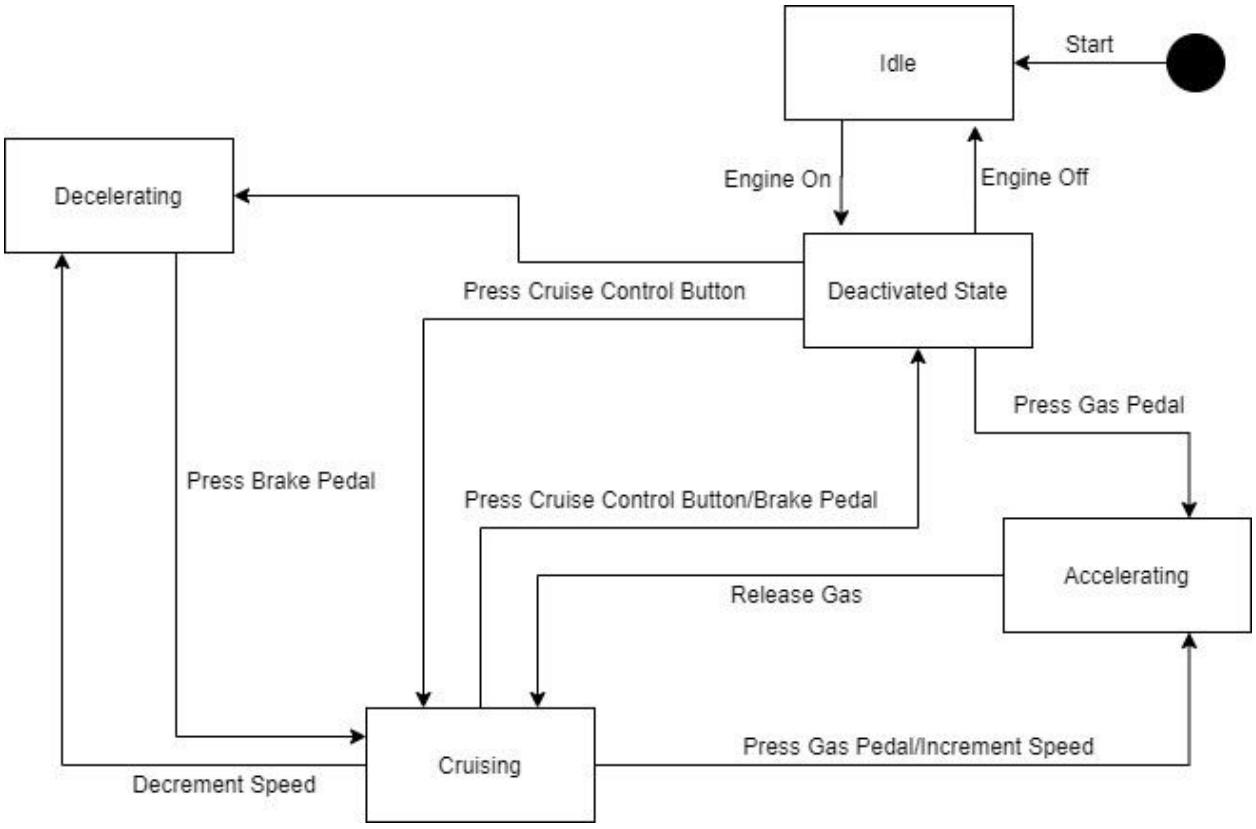
Class: Button In regards to only cruise control, this is the class responsible for activating and deactivating the software.	
Responsibility:	**Collaborator:**
Activates and deactivates cruise control	Cruise Control Dashboard

Class: EngineManagementSystem In regards to only cruise control, this is the class responsible for sending requests to the engine and updating cruise control with engine speed	
Responsibility:	**Collaborator:**
Sends request to maintain speed to engine	Cruise Control Engine
Updates Cruise Control system	Cruise Control

3.5 UML Activity Diagram



3.6 UML State Diagram



4. Software Architecture Section

4.1 Architecture Style

For our cruise control system, we considered having two separate architectures for both the driver and the technician. For the driver, we consider using either a **layered architecture, data centered architecture, data flow architecture, or call and return architecture, along with an object-oriented architecture** for our cruise control system. On the other hand, for the technician, we consider using solely an **object-oriented architecture**.

The first architecture we considered was the **layered architecture**, which comes with multiple advantages and disadvantages. In terms of advantages, the layered architecture allows us to create a smaller scope in each layer, making our problem of creating the cruise control solution more straightforward. Furthermore, each layer would also have less cases to test which would make it easier to test also. There is also more changeability in this type of architecture, meaning that if we are not satisfied with the implementation of a specific layer, we can replace it with another layer, as long as they're implemented using the same interface. However, there are a few cons that come along with this architecture. Firstly, the management cost gets increasingly large if you have too many different layers. The performance may also get slower if you continuously add more layers to your architecture. Overall, it may be difficult to optimize your solution if your codebase gets too large.

In terms of the **data centered architecture,** there are a few advantages and disadvantages to using this model for our software. The advantages of this model mainly revolve around the use of a central repository, which allows permanent data storage, ideal for updating and persistently storing the current speed needed for cruise control. With this idea of a central repository, multiple clients can send requests to this data storage to modify or update it, such as the driver increasing or decreasing the speed. However, this model also comes with a few disadvantages, such as the high dependency between the data storage and the clients, the vulnerability of the data storage failing and/or replicating data, and the high cost of evolving the data. Additionally, the Cruise Control software requires a consistent flow of data and would

require a large number of requests from client to server which is not ideal for our software due to potential latency issues.

A **data flow architecture** also comes with its unique advantages and disadvantages as a potential model for our software. The advantages of this model mainly revolve around that the actions of the driver, in relation to the cruise control software, are dynamic, constantly creating a new flow of data throughout the timeline of using cruise control which could lead to new data or no specific change in data. Furthermore, a data flow architecture provides concurrency and high throughput which is required due to our continuous stream of data within the CC system. Additionally, a data flow architecture promotes reusability which is essential to mitigating redundancy and improving the performance of our CC system. However, data flow architecture may not be the best for our given situation because it can be more difficult to implement and simulate in our program.

Another software architecture we considered was a **call-return architecture**. A call and return architecture would allow the main program to have control of various subprograms within a piece of software. In this case, a call-return architecture would allow us to model the CC as the main program while having various subprograms, or functions, to invoke components within our CC system—such as the sensors. Although this architecture provides us with a workable solution, it does not fulfill our high data throughput and organizational needs. This is mainly due to the fact that CC is a real-time system, so events must happen with time constraints. When you give the job to different subprograms, you lose the ability to sequence events. From an execution point of view, how would we know that hitting the brake doesn't finish before incrementing or decrementing the speed in this architecture?

As a result, we have picked an **object-oriented architecture**. An object-oriented architecture provides a means of information passing through parameters and objects. We can use objects to model our various components such as the CC system, sensors, and EMS. Moreover, object-oriented architecture is easiest to implement than the previous architectures and it is easier to map and apply to real-world objects. This architecture promotes the beneficial reusability of other architectures, such as call-return and data flow, while also allowing

consistent data flow through parameter passing. Overall, this architecture will serve us well for not only the Cruise Control system in the scope of the driver, but also the technician.

4.2 Architectural System Categories

The components in the software architecture consist of sensors, the CC system, the logger and the EMS. For each component, we will have a class modeling within our object-oriented architecture.

As for connectors, we will use class instance variables within a Car object to transfer information between our components. In a sense, this will mimic the data passing process with wires and hardware components. For instance, the logger will connect the CC system to the admin interface.

Our main technical constraint is the use of a programming language that supports object-oriented programming (Python, Java, C++, etc.). Additionally, we will need to use a library that supports graphical user interfaces so that we can build a front-end to interact with the CC software we develop. In addition, another constraint mentioned in part 1 of this section is the timing of how events are executed in our system. Cruise Control is a real-time system which means that events should happen with time constraints, so we will utilize event listeners and global flags to mimic the real-time responsiveness of a car . One final constraint to consider is the relationship between the CC system and the logger. The logger should *not* be able to request anything from the CC system since the relationship between the two entities is a one way direction. Rebounding off that idea, the brake should also only give information to the CC, but not the other way around. However, the CC and the EMS have a two way relationship through a sensor, which has two buffers (One which passes the data one way and one which receives the data).

4.3 Architectural Management Control

The control unit of our software will be the CarDriver program. We will utilize this as the central control unit to pass information between the sensors, CC, and EMS. The software will run in the main of the program which will consist of a Car object with a collection of sensors, a single CC system object, and a single EMS object. The control of the software will be primarily in the Car object, but it will be transferred over to the EMS when the EMS receives instructions to increment or decrement the speed of the vehicle to maintain the CC feature. The sensors are simply in the system to just pass information. Overall, the control of the system will remain within the main of the CarDriver program which will be our driver program, and will infinitely loop to get information from the sensors, make a decision, implement that decision and loop again. Furthermore, the control topology will be based around a tree diagram, where superclasses will reside at the top of the tree and will call methods from its subclasses, which reside in tiers below the top level. The pattern continues where these subclasses will act as the superclass, calling methods from its subclasses residing in tiers below that specific level. Overall, control management in our CC system is synchronous since events can only be executed one at a time.

4.4 Data Architecture

The data architecture is a composition of models, policies, and standards that dictate how data is collected, stored, and managed. For the CC system, we will mimic the transfer of data, which would typically be wires, through our program. The transfer of data will be between the sensors in the vehicle, the engine management system (EMS), and the cruise control system. Furthermore, the sensors will act like little computers, receiving data, and it will go through the protocol stack, where it will provide data to other units. The CC system will interact with the sensor and EMS components of the software and it will act as the central control unit to manage data and relay responses to these components accordingly based on the inputs. The data will be sent through parameter passing within our object-oriented software architecture. Therefore, data objects will be continuously fed into the CC system from the sensor inputs and then the CC system will respond accordingly and send the information to the EMS.

In order to have an object-oriented software architecture, we list our objects, methods and formal parameters below.

Car Object

Attributes:

- speed: an integer representation of the car's current traveling speed
- cruise_control: an instance of CruiseControl
- logs: a list of logs

Methods:

- get_current_speed(): returns the current speed of the car
- get_cc_status(): returns the status of the cruise control software
- get_cc_speed(): returns the cruise control speed
- get_logs(): returns a copy of logData (a copy to ensure that the logs cannot be externally changed)
- set_cc_speed(): sets the cruise control speed to the current speed of the car
- clear_cc_speed(): resets the cruise control speed to 0
- increment(): increments the speed of the car
- decrement(): decrements the speed of the car
- activate_cc(): activates the car's cruise control
- deactivate_cc(): deactivates the car's cruise control
- incremement_cc(): increments the car's cruise control speed
- decrement_cc(): decrements the car's cruise control speed

CruiseControl Object

Attributes:

- cc_speed: integer representation of the cruise control speed
- status: boolean representation of whether the cruise control is activated or deactivated

Methods:

- get_speed(): returns the cruise control speed

- get_status(): returns the cruise control status
- increment(): increment cc_speed
- decrement(): decrement cc_speed
- set_speed(new_speed): sets cc_speed equal to new_speed
- activate(): activates cruise control by changing the status to true
- deactivate(): deactivates cruise control by changing the status to false
- check_range(new_speed): returns true if new_speed is within range (>= 10 and <= 90); false otherwise

Log Object

Attributes:

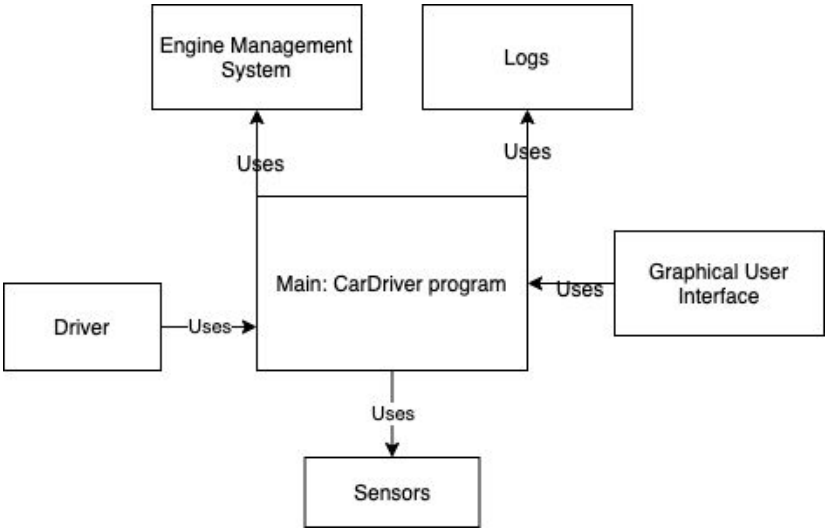
- event: string description of the event that occurred
- date_time: date and time representation of when the event occured

Methods:

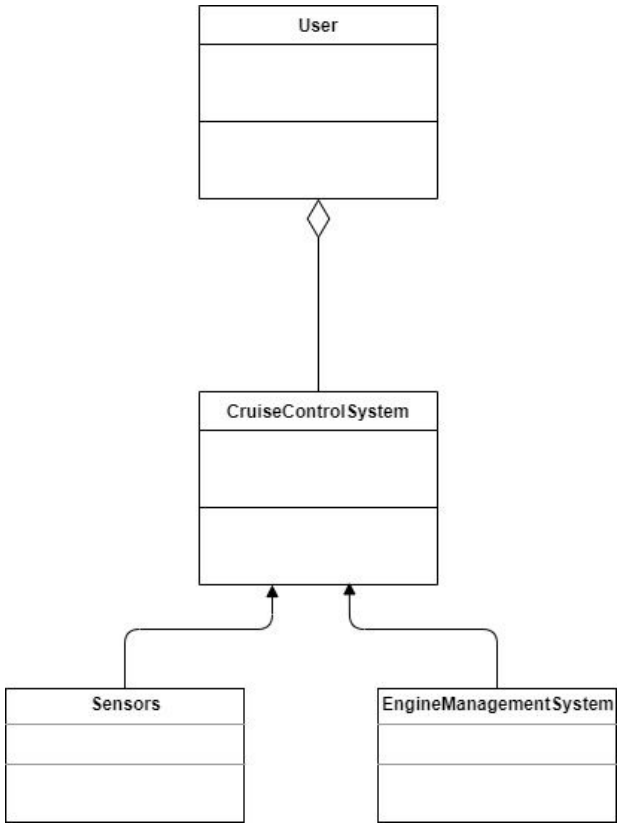
- get_event(): returns event
- get_date_time(): returns date_time
- __str__(): returns a string representation of the log object in the form: date_time + "\t" + event

4.5 Architectural Design

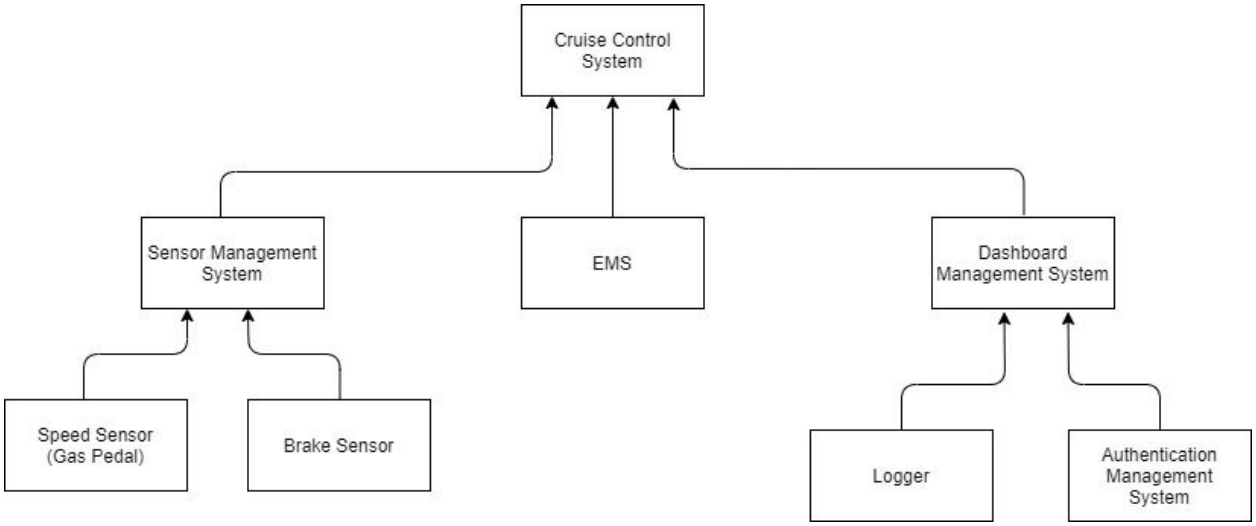
Architecture Context Diagram



Cruise Control Function Archetypes



Top-Level Component Architecture



4.6 Relevant Issues

The quality attributes of our software will be responsiveness, durability—low chance of significant errors that would cause the software to crash, and performance. We will measure responsiveness by testing how the software reacts to an abundance of inputs such as pressing the gas or brake, changing speeds, and activating or deactivating the CC feature in quick and successive cycles.

The architectural patterns that we see in our system structure are still relatively high level, but we felt that it still covered the overall requirements of Cruise Control quite well. The architecture patterns that we used to match the system's structure to the patterns' structure are the architecture context diagram, archetypes and the top-level component architecture. In terms of the architectural context diagram, we modelled the manner in which the software (Cruise Control System) interacts with any external entities (Driver, Sensors, EMS). Moving on to the archetype diagram, we were able to define and map out all of the relevant classes needed to create this system. From there, we were able to refine the architecture into components as seen in our top-level component architecture. We chose our components based on the classes we made in our requirements analysis modeling section. These components addressed multiple functionalities of our cruise control system, such as logging, EMS processing, security, and external communication management.

Overall, this architecture review helped us to see the big picture for our cruise control system, and also helped to clarify and unify our team's vision for the system. We also learned more about the architecture patterns that we used and how they relate to the quality attributes. However, based on our existing documentation and use cases, we can determine that there are some quality issues raised by the architectural patterns that we used in our design which are listed out below.

Issue #1

We must clarify the components in the Top-Level Component Architecture and organize the structure based on major functionalities of the Cruise Control system. This issue was brought up on 4/17/20 and will be fixed by Jayson Infante. It will be fixed by organizing the structure based on components that act as management systems. For instance, the logger and authentication management components can interact with a dashboard management component. The final diagram will be approved by all team members of Team Turtles.

Section 5: Formal Review

Work Product being reviewed: Requirements Analysis Modeling and Software Architecture section of Cruise Control Software Document

Reviewers: Albert Chen, Raj Gadhia, Simon Gao, Jayson Infante

Findings: Issues/Errors listed below

Issue #1 (Minor):
We must clarify the components in the Top-Level Component Architecture and organize the structure based on major functionalities of the Cruise Control system.

Issue #2 (Minor):
The Car Object and Bridge Object appear to be unnecessary duplicates since they both serve the same purpose of holding the various states for each attribute of the Cruise Control object. In addition, some of our classes/methods are

Issue #3 (Major):
We did not include a user interface (UI) for the technician to review the car's logs. We will need to add a UI to display the logs in a table format and allow the technician to request any specific log given an ID number.

Issue #4:
Our knowledge of software security is not extensive so that may be an issue upon creating the software. The extent of our knowledge, in regards to OOP, is data encapsulation and ensuring that mutable lists are immutable to any external calls made to our software. For example, the list of logs should not be returned, but they should be displayed or a copy should be made and returned.

Issue #5 (Minor):

There are no constraints in our document regarding race conditions, however, we will need to account for that during our testing phase. The structure of the code should run sequentially, so we may not have to worry about this during implementation; however, we must ensure/add these constraints to our document.

Conclusion: As a team, we will accept the product provisionally (Minor errors have been encountered and will be corrected, but no additional review will be required).