Autopilot Architects - IoT Hug the Lanes

Ruthie Mullisky, Cecilia Esteban, Ariana Beckford, and Kayla DePalma

Table of Contents

1.1 Introduction 2
1.2 Functional Architecture 4
1.3 Requirements 8
    1.3.1 Functional Requirements 8
        1.3.1.1 Driver Features 8
        1.3.1.2 Automated Features 10
        1.3.1.3 Technician Features 16
    1.3.2 Non-Functional Requirements 17
1.4 Requirement Modeling 19
    1.4.1 Use Case Scenarios 19
    1.4.2 Activity Diagrams 23
    1.4.3 Sequence Diagrams 26
    1.4.4 Classes 29
    1.4.5 State Diagrams 30
1.5 Design 32
    1.5.1 Software Architecture 32
    1.5.2 Interface Design 39
    1.5.3 Component-Level Design 41
1.6 Project Code 44
1.7 Testing 61
    1.7.1 Requirement Testing 61
    1.7.2 Use Case Scenario Testing 75

1.1 Introduction

With automation becoming a prevalent part of the lives of many, this project intends to plan, design, and implement software that helps usher in the future of self-driving technology. However, while this idea of automation acts as the driving force behind this project, the choice of the user is just as important. The software will provide the user the ability to drive by themselves or provide input as they wish throughout their drive. Furthermore, with the speed at which technology advances in the present day, it is imperative that this technology keep up with and implement features that accommodate an ever-changing society and market; therefore, this software will be updatable, allowing the features and applications to remain up-to-date with the needs of the consumer, all while keeping safety in mind. In doing so, this software will contribute to the future of vehicular automation.

1.1.1 Features and Scope

Important features include options for self-driving as well as manual driving of the car specified by the driver. The car will also have the capability to recognize stop signs, traffic lights, pedestrians, surrounding vehicles, and other obstacles on the road to stop accordingly. There will be options to employ adaptive speed control to automatically maintain a selectable distance between the driver's car and the car in front and turn on lane assistance to help keep the vehicle within its respective lane. When needed, a technician will have secure access to update or modify any of the car’s features. While these are several of the fundamental capabilities of a self-driving car, more extensive features and the full development of this product require additional incremental releases upon this one.

1.1.2 Architecture

The vast realm of the Internet of Things (IoT) is relied on for the advanced architecture behind this self-driving car. The car will be equipped with a wide range of sensors that will constantly collect and analyze data, allowing it to properly perform any features that are reliant on or related to the data. For example, the sensors will collect data that helps the car localize and perceive its environment; this includes data collected through the use of GPS, cameras, laser scanners, an electric speedometer, and more. While processing this data through the specific sections of the Architecture, such as Sensor Fusion, Planning, and Vehicle Control, the appropriate vehicle action will quickly be implemented to ensure the safety of the passengers of the car.

1.1.3 Mission-critical Real-time Embedded System

The term mission-critical refers to the fact that any mistake made in the production process can endanger lives and real-time refers to how the car will make responsive decisions based on the data it is provided. Embedded software within this context refers to the software controlling the automotive systems as an integral part of the hardware system. This project is a mission-critical embedded system because it is very easy for drivers and their passengers to be harmed while in a car, and this danger is increased by cars being self-driving. It is easy for people using self-driving cars to be harmed if the software is faulty, but the lives of other people on the road can also be at risk. Throughout this project, the priority is on making sure that there is a high

availability percentage and high reliability while fixing any errors that are encountered as soon as possible to make sure the product is safe.

1.1.4 Software Development Process

While undertaking such a large project, the Unified Process Model will provide an efficient manner to plan and develop this project. Its stages (inception, elaboration, construction, transition, and production) are the most adaptable and flexible to the vast requirements of the project, and as such, will provide a strong framework for setting and achieving goals for this self-driving car. Its iterative and incremental approach allows for continuous improvement and refinement. Additionally, its emphasis on quality and testing is crucial to minimize the risk of errors.

1.1.5 Team Qualifications

A cyclical workflow will be used where the team lead for the project changes periodically during the implementation of the features. Autopilot Architects is composed of four programmers (with experience programming in Python, Java, C, and C++), some with stronger problem-solving and analytical skills befitting the task at hand, and others whose strengths lie in communication and code implementation. Throughout the process, everybody's individual responsibilities and goals will be heavily documented, and the same will apply to any problems that are encountered at each stage and the eventual solution(s) implemented to solve them.

1.2 Functional Architecture

The functional architecture behind a self-driving car allows for a regulated, methodical framework for the intake of information that the car senses, such as its physical location or other physical objects in its vicinity, to be efficiently processed and appropriately acted upon. It is able to take both information from the physical environment and also the from the driver and process it in a way such that the vehicle smoothly and expeditiously takes action. In addition to this, this functional architecture allows for secure access for a technician to fix any issues that may arise in the vehicle, as well as an overarching administrative module tasked with overseeing the collection of all data that enters or leaves the vehicle's sensors or other modules and compiling it into a secured log file. Below is a physical representation of the framework behind the functional architecture for a self-driving vehicle:

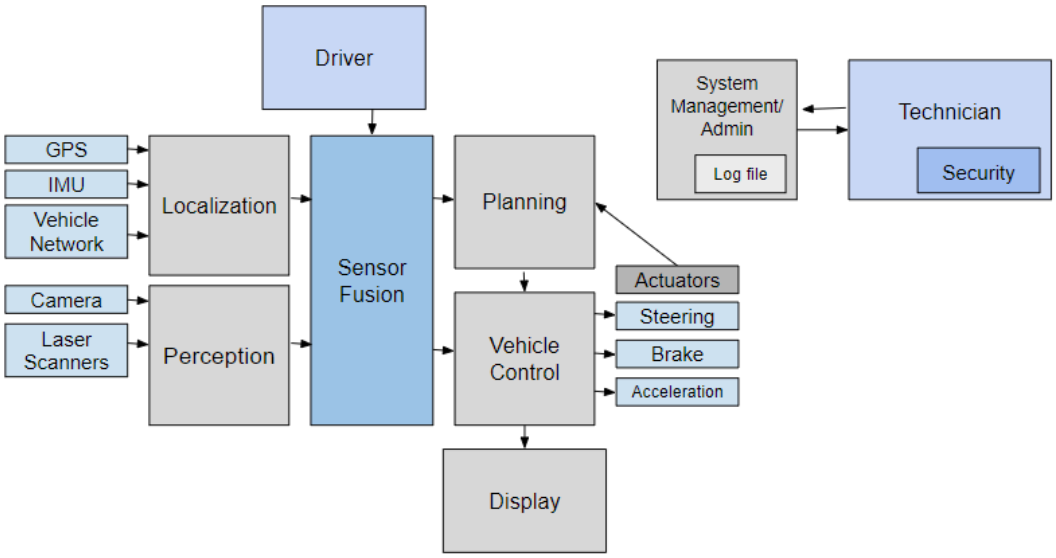


Figure 1: Functional Architecture

1.2.1 Localization and Perception

An integral part of the architecture of a self-driving car is its different types of sensors, such as its camera and laser scanners (or LiDAR–Light Detection and Ranging–technology). It is through these sensors that the vehicle is able to retrieve data from its surroundings, both in the form of visual information (through the camera) and models formed through the emission of a pulsed, near-infrared laser (LiDAR). This data is then interpreted and processed by the Perception module, the likes of which extracts environmental information that allows for the detection and tracking of objects within the vehicle's surroundings. This information is later sent to the Sensor Fusion module, which cleans up the data to then be transmitted to Planning to be used to determine which–if any–action should be taken. For example, the LiDAR sensors would essentially map the objects in the area surrounding the car, such as other vehicles on the road, buildings, sidewalks, pedestrians, and the like, and the car would use the information gained regarding how close or far away these objects are to later choose whether to continue driving, stop, speed up, or slow down. The cameras would focus on the visual surroundings of the car,

and the information gathered would be used in cases where a car encounters, for example, a traffic light, a stop sign, or other form of traffic sign that indicates some kind of action should be taken by the vehicle, whether it be to slow down because of the speed limit, start slowing down to a stop because of a stop sign/yellow or red light/crosswalk, etc. Just as the camera and laser scanners are important to the vehicle's perception and capability of localization, so are GPS (global positioning system), IMU (inertial measurement unit), and the vehicle network. The GPS retrieves data regarding the vehicle's geolocation and its surroundings, while the IMU keeps track of the vehicle's acceleration and angular velocity, and the vehicle network hosts other critical information regarding the vehicle's immediate environment and notable patterns in the data. This information is then interpreted and processed by the Localization module, which–similarly to the Perception module–parses through the information provided by the GPS, IMU, and vehicle network. It then sends the data to the Sensor Fusion module, where it is prepared for its later use in the Planning module, which determines whether any action should be taken, and, if so, which. These modules would be most useful in situations where the driver wishes to, for example, know the quickest route to a particular destination, or how quickly or slowly the vehicle is moving and whether that needs to be adjusted. Given that the information from both the Perception module and the Localization module are then fed to the Sensor Fusion module, said information is used in conjunction with one another in order to make the informed decisions expected of a self-driving vehicle.

1.2.2 Display, Driver, Sensor Fusion, and Planning

Once data is received from the Localization, Perception, and Driver modules, the Sensor Fusion module seamlessly integrates the input to craft a more refined and dependable depiction of the dynamic environment. It does this through the normalization of raw sensor readings via meticulous data acquisition and alignment. At the initial level of Sensor Fusion abstraction, known as raw-data fusion, data from different sensors measuring identical physical parameters are combined. For instance, leveraging the complementary nature of images captured by distinct cameras or from a camera and LiDAR sensor by merging them will enhance the environment's completeness and precision. For redundant sensors that provide information about the same target, their outputs are combined to increase the reliability or confidence of the results. Subsequently, the feature-level fusion stage integrates extracted features and notable characteristics it identified from multiple sensors to create a detailed representation of the observed environment, filtering out extraneous noise. Then, the high-level fusion stage takes the fused data and features obtained from the lower-level fusion processes and integrates them into a coherent representation of the observed environment. It analyzes the integrated information to identify objects, recognize patterns, and assess risks, providing a comprehensive and accurate representation of the situation based on the combined input from all available sources. To enhance feature fusion, attention-based techniques will be employed. By directing the network to focus on relevant regions across the entire input space, rather than specific areas like those used for geometric fusion, a better understanding of the surroundings is achieved by considering the data from all sensors. This enables efficient detection of potential hazards. For instance, attention-based feature fusion will help the vehicle recognize and monitor the status of a traffic light and oncoming traffic simultaneously to decide when it is safe to turn.

Upon receiving data from Sensor Fusion, the Planning module meticulously processes this information to determine the most appropriate course of action, subsequently transmitting instructions to Vehicle Control. Since the driver retains full authority to override any suggested functions indicated by the sensors, their direct inputs take priority over sensor recommendations. For example, if the driver applies the brakes then the vehicle's speed will be reduced even if other sensors indicate no need to slow down. However, if the driver does not provide input that contradicts sensor data, the system will autonomously enact actions following sensor readings. The Planning module makes decisions about behaviors and trajectories based on top-level goals in addition to the perceived environment. For instance, the data of a stop sign, red traffic light, or pedestrian passed by Sensor Fusion will be processed in Planning to then decide to bring the vehicle to a stop. Additionally, Planning's submodules can include mission planning (route from current location to destination), behavior planning (deciding among operation modes such as "school zone," "crosswalk," and "parking"), and trajectory generation. The trajectory generation submodule is responsible for generating obstacle-free routes represented in coordinates while considering energy and directional constraints. It is often the last module in the Perception-Planning pipeline and is sent directly into Vehicle Control.

Moreover, a comprehensive Display interface will communicate the adjustments made to vehicle features and controls post-data processing, such as activating or deactivating headlights or windshield wipers. This Display will also incorporate a digital speedometer displaying the current speed of the vehicle, an estimate of remaining fuel mileage, lane assistance indicators to gauge the vehicle's alignment, and illuminated color-coded icons signaling warnings like low tire pressure or engine failure. The Display selects which warning lights to activate based on the data passed to Planning indicating potential feature malfunctions, and there is an array of warning lights and colors to signify severity. Overall, this Display's purpose is to keep the driver informed on any decisions the self-driving car makes and of any warnings that will impact their safety.

1.2.3 Vehicle Control

The majority of autonomous vehicle literature uses a three-part system in which the Vehicle Control module focuses on breaking down the planned motion into duration, strength, and direction of acceleration. Duration, strength, and the direction of acceleration are implemented as the vehicle's actuators. There are three categories of Vehicle Control: propulsion and braking, steering, and collision avoidance and emergency braking. These categories are referred to as longitudinal control, lateral control, and reactive control, respectively. There is another possible system to implement known as the world model, which is a model with a real-time nature that contains current and historical knowledge about images, maps, entities, events, and how they relate to each other. Under the world model, the Control and Planning modules also encompass behavior generation. Behavior generation is dependent on not only the data from the current and historical knowledge the world model has, but it is also dependent on feedback.

Vehicle Control is mostly based on the principle of feedback, the Control modules monitor the completion of past commands and issue new commands based on the results of the completion

of the past commands. Many Control modules do not require a high level of abstraction as controllers usually get their information directly from the corresponding sensor; because of this many controllers are controlled by minimal intelligence electronic control units (ECUs). For example, the brake pressure controller doesn't need any more information than the brake pressure, which is a bottom-level sensor. Minimal intelligence ECUs are responsible for Control modules such as braking and steering. Feedback from actuators usually goes to sensors, but there are many situations in which this way of utilizing feedback can be unreliable. An example of Perception sensors not always able to provide reliable information is the case of extreme environmental conditions. Even excluding the case of extreme environmental conditions, waiting for sensors to get feedback from actuators, and then the information from sensors going to planning takes a lot of time. A more efficient way to utilize feedback would be for the Planning module to get feedback from actuators directly. An example of the benefits of the Planning module getting feedback from the actuators directly would be when the controllers cause substantial changes from the expected position due to not executing the generated trajectory. In the example just mentioned, it would be faster for there to be direct feedback to the Planning module because it would take time for the Perception modules to detect it.

1.2.4 System Management/Admin and Technician

The System Management/Admin module is responsible for fault management and the crucial logging facility for all the data that comes into and out of the car's scanners and features. The log file is one of the most important aspects of this module, as it serves as a database for the car's functionality. It collects and stores the incoming data from the sensors, as well as all actions taken by the car as a result of the data from the sensors. Additionally, it keeps track of any faults or mistakes that may have occurred at any stage throughout the automation of the car's motion in any of the modules. For example, if there was a mistake that occurred during the processing of when braking should have occurred, the log file would have this mistake on record, among all other successfully completed data, ready to be accessed by the Technician module for analysis. It is important that this log file is kept up to date and kept secure, as its mistakes may be able to be taken advantage of by malicious actors. On top of this, the System Management/Admin module is responsible for the connection to the cloud, as in consistently uploading vehicle information to the cloud for safekeeping, as well as maintaining a strong network connection while the vehicle is running.

The Technician module is configured in such a way that it has very secure access only to necessary information, such as the information that is stored in the log file of the System Management/Admin module. By limiting such availability to the vehicle's data and the access to critical data, the security of the car and its driver is ensured. Additionally, there is only authorized access to the Technician module, meaning the integrity of the data is ensured. Technicians are able to look in the log file and find the occurrence of a fault or mistake and see exactly what happened–strictly through access to the log file. From this, they can then appropriately modify the vehicle to fix the fault or mistake from happening in the future. Additionally, the Technician has the ability to provide software updates as new software increments are released, as well as having the capability to perform analysis on the vehicle and among the software in order to identify and isolate an issue and properly replace whatever might be causing it.

1.3 Requirements

The requirements detailed below provide a comprehensive and exhaustive description of exactly under what conditions a certain event must take place and exactly what must occur as a result thereof. The requirements are split into two separate categories: functional requirements and non-functional requirements. While they are both of equal importance, they provide substantially different specifications which need to be implemented accordingly. The functional requirements provide a complete outline of what must take place for an event to occur. On the other hand, the nonfunctional requirements provide strict guidelines regarding the metrics for the vehicle to operate within, such as performance, reliability, and security. Together, these requirements allow the vehicle to operate safely and efficiently.

1.3.1 Functional Requirements

1.3.1.1 Driver Features

1.3.1.1.1 Driver Starts the Car

Pre-conditions

- The vehicle is off and the engine is not running.
- Driver has the key and it is within 2 feet of the vehicle.
- Driver is pressing the brake.
- Driver is sitting in the driver's seat.

Post-conditions:

- The engine is running.
- The Display lights up.

1.3.1.1.1.1 Door sensors detect a valid key within 2 feet and send the data to Sensor Fusion.

1.3.1.1.1.2 Driver's Seat sensors detect the driver's weight indicating if they are sitting in the driver's seat (weight on the seat is greater than 0) and send the data to Sensor Fusion.

1.3.1.1.1.3 Driver pushes the engine "start/stop" button.

1.3.1.1.1.4 Start/Stop sensor sends a "turn-on" signal to Sensor Fusion.

1.3.1.1.1.5 Sensor Fusion normalizes the data from the Door, Driver's Seat, and Start/Stop sensors and sends an "engine start" request to Planning.

1.3.1.1.1.6 Planning reevaluates the preconditions and sends a "turn-on" request to Vehicle Control accordingly.

1.3.1.1.1.7 If Planning sends a "turn-on" request, Vehicle Control transmits the "turn-on" signal to the Start actuator to start the engine.

1.3.1.1.1.8 If Vehicle Control successfully turns on the engine, it sends a message to light up the Display.

1.3.1.1.1.9 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.1.2 Remote Start

Pre-conditions

- The vehicle is off and the engine is not running.
- Driver has the key and it is within 15 feet of the vehicle.

Post-conditions:

- The vehicle is on and the engine is running.
- The vehicle's heater or A/C is on depending on the settings the driver last left the heater or A/C on.
- The Display and all the buttons light up.

1.3.1.1.2.1 Driver hits the button that is responsible for the remote start on the key fob.
1.3.1.1.2.2 The engine detects and processes the "remote start" signal.
1.3.1.1.2.3 Start/Stop sensor sends a "turn-on" signal to Sensor Fusion.
1.3.1.1.2.4 Sensor Fusion normalizes the data (i.e. the "turn-on" signal) from the Start/Stop sensor and sends an "engine start" request to Planning.
1.3.1.1.2.5 Planning confirms "turn-on" and sends a request to Vehicle Control.
1.3.1.1.2.6 Vehicle Control transmits the "turn on" signal to the Display and to the Start actuator to start the engine and the heater or A/C.
1.3.1.1.2.7 Display indicates that the engine and the heater or A/C are on and all of the buttons on the dashboard light up.
1.3.1.1.2.8 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.1.3 Driver Turns Off the Car

Pre-conditions

- The vehicle is on and the engine is running.
- The vehicle is not in motion so the speed is 0.
- The vehicle is in park.

Post-conditions:

- The engine is off and no longer running.
- The vehicle is off.

1.3.1.1.3.1 The Vehicle Speed sensors record the vehicle's speed and send it to Sensor Fusion.
1.3.1.1.3.2 The Park sensor records if the vehicle is in park and sends it to Sensor Fusion.
1.3.1.1.3.3 Driver pushes the engine "start/stop" button.
1.3.1.1.3.4 Start/Stop sensor sends a "turn-off" signal to Sensor Fusion.
1.3.1.1.3.5 Sensor Fusion normalizes the data on the vehicle's speed, whether the vehicle is in park, and the "turn-off" signal and sends an "engine stop" request to Planning.

1.3.1.1.3.6 Planning sends a “turn-off” request to Vehicle Control if the vehicle’s speed is equal to 0 and the vehicle is in park.

1.3.1.1.3.7 If Planning sends a “turn-off” request, Vehicle Control transmits the “turn-off” signal to the Stop actuator to stop the engine and to the Display.

1.3.1.1.3.8 If Vehicle Control sends a “turn-off” request, the Display shuts off.

1.3.1.1.3.9 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2 Automated Features

1.3.1.2.1 Lane Assistance for Assisted Driving

Preconditions

- The vehicle is on and the engine is running.
- Cruise control and/or self-driving is activated.
- The vehicle is in motion, so it has a speed greater than 0 mph.
- The sensors have identified that the vehicle has veered less than 6 inches of a lane marker, without the activation of a turn signal, via cameras that can see the sides of the car in relation to the lines that mark the lanes.

Postconditions

- The driver is alerted of this change via a small ping and a small light on the dashboard.
- The car is maneuvered back to the center of the lane and can continue driving as normal.

1.3.1.2.1.1 The cameras identify the vehicle is 6 inches away from the lane marker.

1.3.1.2.1.2 This difference from the center, as identified from the cameras, will be measured and sent to the Sensor Fusion module.

1.3.1.2.1.3 The Sensor Fusion module normalizes this data, and sends it to the Planning module.

1.3.1.2.1.4 The Planning module sends a request to the Vehicle Control module to adjust the steering of the vehicle in order to center the vehicle back in the lane again.

1.3.1.2.1.5 Vehicle Control transmits a message to the Steering actuator to turn the steering wheel the correct amount in order to recenter the vehicle in the lane. The vehicle is in the center of the lane when the cameras on the sides of the car sense that it is equidistant from both of the lane markers, that is, the distance from the left side of the car to the lane marker on the left side equals the distance from the right side of the car to the lane marker on the right side.

1.3.1.2.1.6 Simultaneously, Vehicle Control sends a message to Display to alert the driver of this lane correction via a small ping and lighting up of a small light on the dashboard.

1.3.1.2.1.7 If an obstacle is detected (via Requirements 1.3.1.2.2, 1.3.1.2.6, or 1.3.1.2.7), then the respective requirements are then enacted in order to maintain the safety of the vehicle and the passengers.

1.3.1.2.1.8 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.2 Stop Signs, Traffic Lights, and Automatic Braking

Pre-conditions:

- The vehicle is on and the engine is running.
- Cruise control and/or self-driving is activated.
- The vehicle is in motion so it has a speed greater than 0.
- Object sensors and cameras have identified a stop sign or red traffic light in the path of the vehicle at a distance of 1000 feet or fewer.

Post-conditions:

- The vehicle should slow down gradually until it comes to a complete stop at least 5 feet from the stop sign or traffic light.

1.3.1.2.2.1 Object sensors and cameras identify if there is a stop sign or traffic light 1000 feet or less in the upcoming path of the vehicle.

1.3.1.2.2.2 Cameras and object sensors measure and send the stop sign or red traffic light distance from the vehicle and the vehicle's speed to Sensor Fusion.

1.3.1.2.2.3 Sensor Fusion normalizes and optimizes the distance and speed data and sends it to the Planning module.

1.3.1.2.2.4 The Planning module transmits a brake request to Vehicle Control when the distance is less than or equal to 250 feet with a determined brake intensity which is calculated using an algorithm based on the vehicle's current speed and distance from the stop sign or red traffic light to ensure a safe stop at least 5 feet away from the stop sign or traffic light.

1.3.1.2.2.5 Vehicle Control sends the brake request and brake intensity to the Brake actuators.

1.3.1.2.2.6 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.3 Automated Front Windshield Wipers

Pre-conditions

- The vehicle is on and the engine is running.

- Sensors recognize it is raining/snowing/hailing (i.e. they detect a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield).

Post-conditions:
- Windshield wipers are turned on to the correct speed.

1.3.1.2.3.1 Sensors and cameras capture data of fluid (rain) and solid (snow, hail) precipitation and measure the amount of precipitation on the windshield and the rate at which it is falling.

1.3.1.2.3.2 Sensor Fusion collects the data from the sensors and cameras and, upon normalizing it, sends it to Planning.

1.3.1.2.3.3 Planning confirms the speed at which the precipitation is falling and–if it is falling at a rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes have been detected on the windshield–transmits a request to Vehicle Control to start the windshield wipers at a particular speed appropriate for the weather conditions (either 1 wipe every 2 seconds–if the falling rate of the precipitation is 1-4 per second–1 wipe every second–if the falling rate of the precipitation is 5-10 per second–or 2 wipes every second–if the falling rate of the precipitation is over 10 per second).

1.3.1.2.3.4 Vehicle Control sends the windshield wiper request to the windshield wiper actuators.

1.3.1.2.3.5 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.4 Automated Headlights

Pre-conditions
- The vehicle is on and the engine is running.
- Headlight sensors recognize low light conditions meaning that there is less than 400 lumens per square meter surrounding the vehicle.

Post-conditions:
- Headlights turn on.
- Display turns on night mode.

1.3.1.2.4.1 The Headlight sensors measure the brightness of the environment surrounding the vehicle and send it to Sensor Fusion.

1.3.1.2.4.2 Sensor Fusion optimizes the data about the brightness of the environment and transmits it to Planning.

1.3.1.2.4.3 Planning processes the data and sends the “turn-on headlights” signal to Vehicle Control if the light surrounding the vehicles is less than 400 lumens per square meter.

1.3.1.2.4.4 If Planning sends the “turn-on headlights” request then Vehicle Control sends the request to the Headlight actuator to turn the headlights on.

1.3.1.2.4.5 If Vehicle Control successfully turns on the headlights then it relays that information to the Display.

1.3.1.2.4.6 If Vehicle Control indicates that it turned the headlights on, the Display shows a message to notify the driver the headlights have been turned on if successful. Display also turns on night mode, which dims the brightness and uses darker colors.

1.3.1.2.4.7 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.5 Automatic Door Locking

Preconditions

- The vehicle is on and the engine is running.
- Doors are unlocked.
- The vehicle has reached a speed of 15 mph.

Postconditions

- All doors are locked.

1.3.1.2.5.1 The Vehicle Speed sensors record the speed of the vehicle and send it to Sensor Fusion.

1.3.1.2.5.2 The Door sensors indicate if the vehicle doors are unlocked and send it to Sensor Fusion.

1.3.1.2.5.3 Sensor Fusion processes and normalizes the speed and door data from the sensors and sends it to Planning.

1.3.1.2.5.4 Planning sends a “lock doors” request to Vehicle Control if the speed is greater than or equal to 15 miles per hour and the doors are unlocked.

1.3.1.2.5.5 If Planning sends the “locks doors” request then Vehicle Control transmits a “lock doors” request to the Door Lock actuators.

1.3.1.2.5.6 If Vehicle Control successfully locks the doors then it relays that information to the Display.

1.3.1.2.5.7 If Vehicle Control indicates that the doors were successfully locked, Display notifies the driver of the locking of the doors by a message and a low audible click as the doors lock.

1.3.1.2.5.8 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.6 Speed Control

Preconditions

- The vehicle is on and the engine is running.
- Cruise control and/or self-driving is activated.

- The vehicle is in motion (i.e. it has a speed greater than 0 mph).
- The sensors have identified the posted speed limit (either via GPS or via camera sensors).
- There is at least 300 feet in the front of the car if it needs to accelerate.

Postconditions

- The vehicle should have smoothly adjusted speed according to the posted speed limit it has identified.
- The speedometer should be appropriately adjusted to reflect the vehicle's current speed.

1.3.1.2.6.1 The GPS sensor, if available, sends the posted speed limit to the Sensor Fusion module.

1.3.1.2.6.2 If the GPS sensor is not available, the camera sensors will physically identify the posted speed limit and send it to the Sensor Fusion module.

1.3.1.2.6.3 The Sensor Fusion module will normalize the different speeds that were sent to it either by the GPS or camera sensors and send it to the Planning module.

1.3.1.2.6.4 The Planning module will send a request to Vehicle Control to adjust the speed of the vehicle accordingly, either accelerating or decelerating.

1.3.1.2.6.5 If the request is to accelerate, Vehicle Control will send a request to the Accelerator actuator telling it to accelerate accordingly until it matches the posted speed.

1.3.1.2.6.6 If the request is to decelerate, Vehicle Control will send a request to the Brake actuator, telling it to decelerate accordingly until it matches the posted speed.

1.3.1.2.6.7 Simultaneously, the Vehicle Control module will send a request to the Display module to make an adjustment on the speedometer on the car's dashboard, showing either the current acceleration or the deceleration.

1.3.1.2.6.8 If at any time the car detects an obstacle or another event in which it cannot complete a full, appropriate change in speed, the appropriate requirement is executed (e.g. Requirements 1.3.1.3, 1.3.1.4, or 1.3.1.9).

1.3.1.2.6.9 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.7 Potential Crash Detection and Automatic Braking

Pre-conditions

- The vehicle is on and the engine is running.
- Cruise control and/or self-driving is activated.
- The vehicle is in motion so it has a speed greater than 0.

- There is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

Post-conditions:
- The vehicle should stop safely at least 3 feet from the object.

1.3.1.2.7.1 Front sensors and cameras record and send data on the size and distance of the object from the vehicle to Sensor Fusion.
1.3.1.2.7.2 Vehicle Speed sensors record the speed of the car and send it to Sensor Fusion.
1.3.1.2.7.3 Sensor Fusion normalizes the data of the size and distance of the object as well as the vehicle's speed and sends it to Planning.
1.3.1.2.7.4 Planning transmits a request to Vehicle Control to start braking if the object is both within 100 feet and larger than 8 inches at an intensity calculated using an algorithm to safely stop the vehicle at least 3 feet from the object.
1.3.1.2.7.5 If Planning transmits a request to start braking, Vehicle Control sends the braking request to the Brake actuators.
1.3.1.2.7.6 Vehicle Control alerts the Display if it has successfully applied emergency braking.
1.3.1.2.7.7 If Vehicle Control indicates that it has applied emergency braking, Display warns the driver with a message and audio alert.
1.3.1.2.7.8 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.2.8 Crash Response

Pre-conditions
- The vehicle is on and the engine is running.
- The vehicle crashed/unexpectedly made sudden contact with an object (another vehicle, a pedestrian, a building, a traffic sign, etc. breached the surrounding 1 inch radius of the vehicle and has done so very suddenly–that is, within 3 seconds).
- The vehicle is at a complete stop/is not moving (velocity of 0 mph).
- The airbags have deployed.

Post-conditions:
- The vehicle dials 911.

1.3.1.2.8.1 Sensors and cameras identify that the vehicle made abrupt contact with an object (another vehicle, a pedestrian, a building, a traffic sign, etc. breached the surrounding 1 inch radius of the vehicle and has done so very suddenly–that is, within 3 seconds), the vehicle is at a

complete stop/is not moving (velocity of 0 mph), and the airbags have been deployed.

1.3.1.2.8.2 Sensor Fusion collects the data from the sensors and cameras, and upon normalizing it, sends it to Planning.

1.3.1.2.8.3 Planning confirms that a crash has occurred and transmits a request to Vehicle Control to dial 911.

1.3.1.2.8.4 Vehicle Control sends the call request to the Call actuator.

1.3.1.2.8.5 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.3 Technician Features

1.3.1.3.1 Technician Signing On

Preconditions

- The car is on.
- The engine is not running.
- There is a secure sign-on previously established/stored where the Technician's username and password are safely and securely stored.

Postconditions

- The Technician successfully logged into the system and is able to perform any software updates or fix any faulty part of the system.

1.3.1.3.1.1 Technician connects the laptop to the car via USB.

1.3.1.3.1.2 There is a secure sign-on prompt displayed on the laptop for a username and password.

1.3.1.3.1.3 The Technician successfully enters the username and password.

1.3.1.3.1.4 System Management/Admin authenticates the provided username and password against the stored Technician's username and password.

1.3.1.3.1.5 If both the username and password are correct, System Management/Admin grants the Technician access to the log file.

1.3.1.3.1.6 If the incorrect username or password is entered, System Management/Admin does not grant the Technician access to the log file, and the secure sign-on prompt remains displayed on the laptop.

1.3.1.3.1.7 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.1.3.2 Technician Signing Off

Preconditions

- The car is on.
- The engine is not running.
- The Technician has successfully been authenticated by System Management/Admin and is signed into the system.

Postconditions

- The Technician's actions within the system are "saved" and cannot be altered by another actor.
- While the laptop is still connected to the car's system via USB, the secure sign-on is displayed.

1.3.1.3.2.1 The Technician has completed any actions needed to be implemented within the system.

1.3.1.3.2.2 The Technician submits a "sign off" request to System Management/Admin.

1.3.1.3.2.3 System Management/Admin completes the "sign off" request, and signs off the Technician from the system.

1.3.1.3.2.4 The display on the laptop screen again shows a prompt requesting a username and password.

1.3.1.3.2.5 The Technician disconnects the USB connection between the car and the laptop.

1.3.1.3.2.6 All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.3.2 Non-Functional Requirements

1.3.2.1 Reliability

1.3.2.1.1 There will be no more than one failure for any functional requirement feature software in 45 years of operation.

1.3.2.1.2 The implementation of a redundant system allows for an overall increase in the reliability of the whole vehicle software system. There will be two identical systems, where one is stored in the front of the vehicle and one is stored in the back to reduce the overall risk the system faces. If one fails, the other is there so the functionality and dependability of the vehicle are not harmed.

1.3.2.2 Performance

1.3.2.2.1 All automated features should have a reaction time within 1 millisecond after an event happens to when it makes a decision.

1.3.2.2.2 All data from sensors will be sent from the Sensor Fusion to the Planning module every 1 microsecond to ensure prompt reaction time.

1.3.2.3 Security

1.3.2.3.1 The driver will have a key fob to unlock, lock, and sound the alarm for their vehicle. The key fob will need to be detected and in a certain proximity of 40 feet of the vehicle to allow the driver to gain access to the vehicle.

1.3.2.3.2 The vehicle, when locked, will remain secure and if unwanted access to the vehicle occurs (for example, an attempted break-in), the alarm will sound, alerting the driver.

1.3.2.3.3 The Technician will have access to the vehicle's log files and software, only after inputting a secure username and password that matches the stored username and password.

1.3.2.4 Software Updates

1.3.2.4.1 The Technician will regularly check which modules need updating and ensure that the appropriate modules have the latest software updates installed.

1.3.2.4.2 The Technician will ensure that the car will automatically check for available updates and receive notifications from the manufacturer's servers.

1.3.2.4.3 The Technician is responsible for updating the software and is the only one who has access to the software updates.

1.3.2.4.4 The Technician will verify that the updates are authentic by checking that the cryptographic signature is from the manufacturer.

1.3.2.4.5 The Technician will install the updates on the relevant modules and will roll back to the previous version if there are any errors in the software update.

1.3.2.4.6 System Management/Admin will log the software updates into the log file.

1.4 Requirement Modeling

1.4.1 Use Case Scenarios

Below is a Use Case Diagram depicting the relationship between different actors (Driver, Sensors, and Technician), and the corresponding use case (Starts the Car, Potential Crash Detection, Adverse Driving Conditions, Lane Assistance, and Sign on/off). This diagram shows the relationships between the defined use cases through the actors who invoke them.

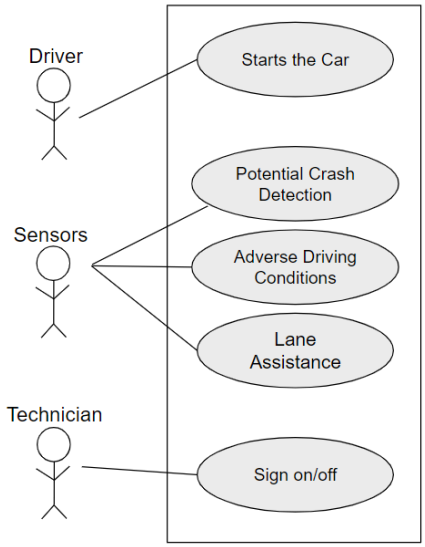


Figure 2: Use Case Diagram

1.4.1.1 Use Case 1: Driver Starts the Car

Precondition: The vehicle is off and the engine is not running, the driver has the key and it is within 2 feet of the vehicle, the driver is pressing the brake, and the driver is sitting in the driver's seat.

Postcondition: The engine is running and the Display lights up.

Trigger: The Driver presses the “start/stop” button.

1. The Driver presses the “start/stop” button.
2. Start/Stop sensor sends a “turn-on” signal to Sensor Fusion.
3. Sensor Fusion normalizes the Start/Stop sensor data and sends an “engine start” request to Planning.
4. Planning decides whether to allow the vehicle to be turned on or not.
5. If turn-on is allowed, Planning transmits a “turn-on” request to Vehicle Control.
6. Exception: If turn-on is not allowed due to an existing condition such as battery issues, the key is not detected, or lack of gas, Planning transmits a “turn-on denied” message to Vehicle Control to send to Display.

7. The Display turns on.
8. All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.4.1.2 Use Case 2: Potential Crash Detection

Precondition: The vehicle is on and the engine is running, cruise control and/or self-driving is activated, the vehicle is in motion so it has a speed greater than 0, and there is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

Postcondition: The vehicle should stop safely at least 3 feet from the object.

Trigger: The front sensors and cameras detect an object larger than 8 inches in the path of the front of the vehicle at a distance of 100 feet or less.

1. The front sensors and cameras record data of an object larger than 8 inches in the path of the front of the vehicle at a distance of 100 feet or less and send it to Sensor Fusion.
2. Sensor Fusion normalizes the data and transmits it to Planning.
3. Planning determines if the object is large enough and close enough to apply the brakes and, if so, determines the intensity of braking using an algorithm to safely stop at least 3 feet from the object.
4. If braking is supposed to be applied, Planning transmits a request to start braking to Vehicle Control and sends a request to Display to display a message that it has detected an object that could cause a potential crash and play an audio alert.
5. Exception: If Planning decides that brakes do not need to be applied anymore such as if the object is no longer within 100 feet of the vehicle's path, or the vehicle moved out of the way, the message will not be visible on the Display and the audio alert will stop.
6. Exception: If Planning never determines that there is an object in the path or that it is not a danger to send a request to brake, then nothing happens.
7. Display displays a message that it has detected an object that could cause a potential crash and plays an audio alert.
8. All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform

1.4.1.3 Use Case 3: Adverse Driving Conditions

Precondition: The vehicle is on and the engine is running, and the sensors recognize it is raining/snowing/hailing (they detect a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35

raindrops/snowflakes on the windshield) or the headlight sensors recognize low light conditions (less than 400 lumens per square meter surrounding the vehicle).

Postcondition: The windshield wipers are turned on to the correct speed and/or the headlights are turned on.

Trigger: The sensors recognize it is raining/snowing/hailing (a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield) and/or that there are low light conditions (less than 400 lumens per square meter surrounding the vehicle).

1. The sensors and cameras record data of rain/snow/hail on the windshield (a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield) and/or of low light conditions (less than 400 lumens per square meter surrounding the vehicle) and sends this data to Sensor Fusion.
2. Sensor Fusion normalizes the data and transmits it to Planning.
3. Planning determines if the conditions require the windshield wipers and/or headlights to be turned on.
4. If windshield wipers and/or headlights are supposed to be turned on, Planning transmits a request to turn either of the two (or both, if necessary) onto Vehicle Control.
5. Exception: If Planning decides that there is no need to turn on the windshield wipers and/or headlights (that is to say, if any of the necessary preconditions are suddenly not met).
6. All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.4.1.4: Use Case 4: Lane Assistance for Assisted Driving

Precondition: The vehicle is on and the vehicle's engine is running. Also, cruise control and/or self-driving mode is activated. The speed of the car is greater than 0 miles per hour.

Postcondition: The vehicle is safely maneuvered back to the center of the lane and the driver is simultaneously alerted of this change via a small ping and a small light on the dashboard.

Trigger: The sensors have identified that the vehicle has shifted less than 6 inches of a lane marker without the activation of a turn signal.

1. The cameras identify that the vehicle is less than 6 inches away from the lane marker.

2. The cameras measure the offset from the center of the lane and send this measurement to the Sensor Fusion module.
3. Sensor Fusion normalizes this data and sends it to the Planning module.
4. The Planning module sends a request to the Vehicle Control module to adjust the steering wheel the correct amount in order to recenter the vehicle in the lane. The vehicle is in the center of the lane when the cameras on the sides of the car sense that the distance from the left side of the car to the lane marker on the left side equals the distance from the right side of the car to the lane marker on the right side.
5. At the same time, Vehicle Control sends a signal to the Display module of the car to alert the Driver of this lane correction through a small ping and the lighting up of a small light on the dashboard.
6. Exception: If the vehicle encounters any obstacles, as sensed through the cameras, then Planning sends a request to the Vehicle Control module to stop the recentering of the car and instead to follow the protocol for Potential Crash Detection and Automatic Braking as detailed in Section 1.3.1.2.7.
7. For every action that is successfully and unsuccessfully performed, each module updates the System Management/Admin log file.

1.4.1.5 Use Case 5: Technician Sign On and Sign Off

Precondition: The car is on and the engine is not running. There is a secure sign-on previously established/stored where the Technician's username and password are safely and securely stored.

Postcondition: The Technician successfully logged into the system and is able to perform any software updates or fix any faulty part of the system. The Technician's actions within the system are "saved" and cannot be altered by another actor. While the laptop is still connected to the car's system via USB, the secure sign-on is displayed.

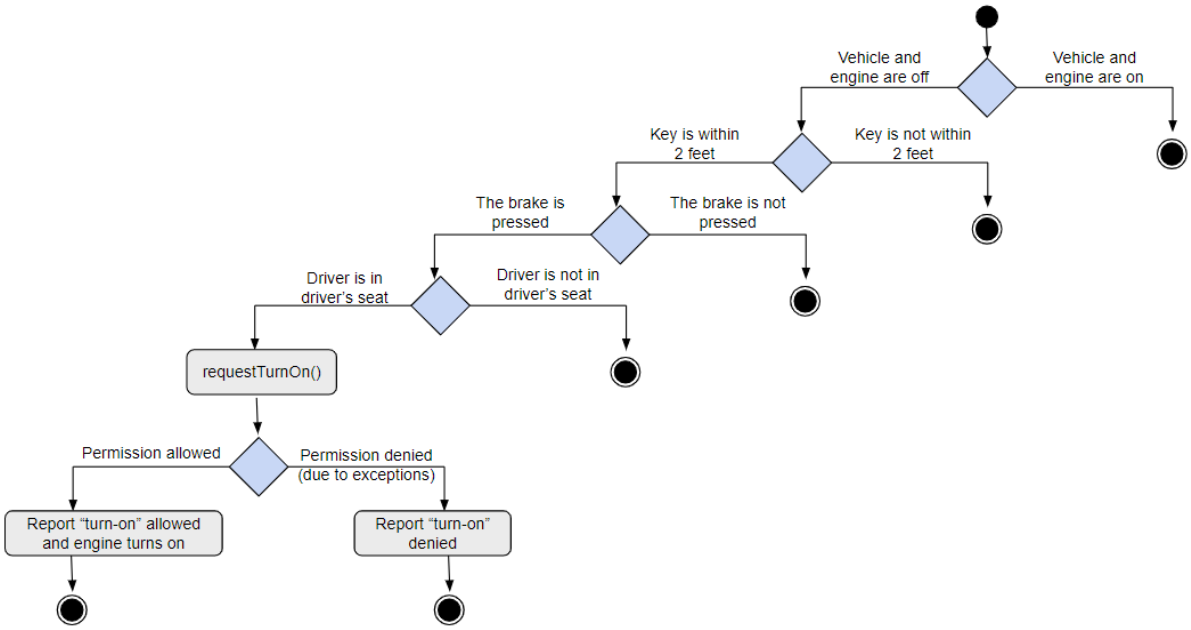
Trigger: The Technician connects their laptop to the car.

1. The Technician signs in with a username and password
2. If the username and password are correct, the Technician gains access to the log file. Otherwise, the request for the username and password will still be displayed.
3. Both successful and failed sign-ins will be recorded in the log file.
4. All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.
5. Before signing off, the Technician must complete any actions that need to be implemented.
6. The Technician submits a sign off request.

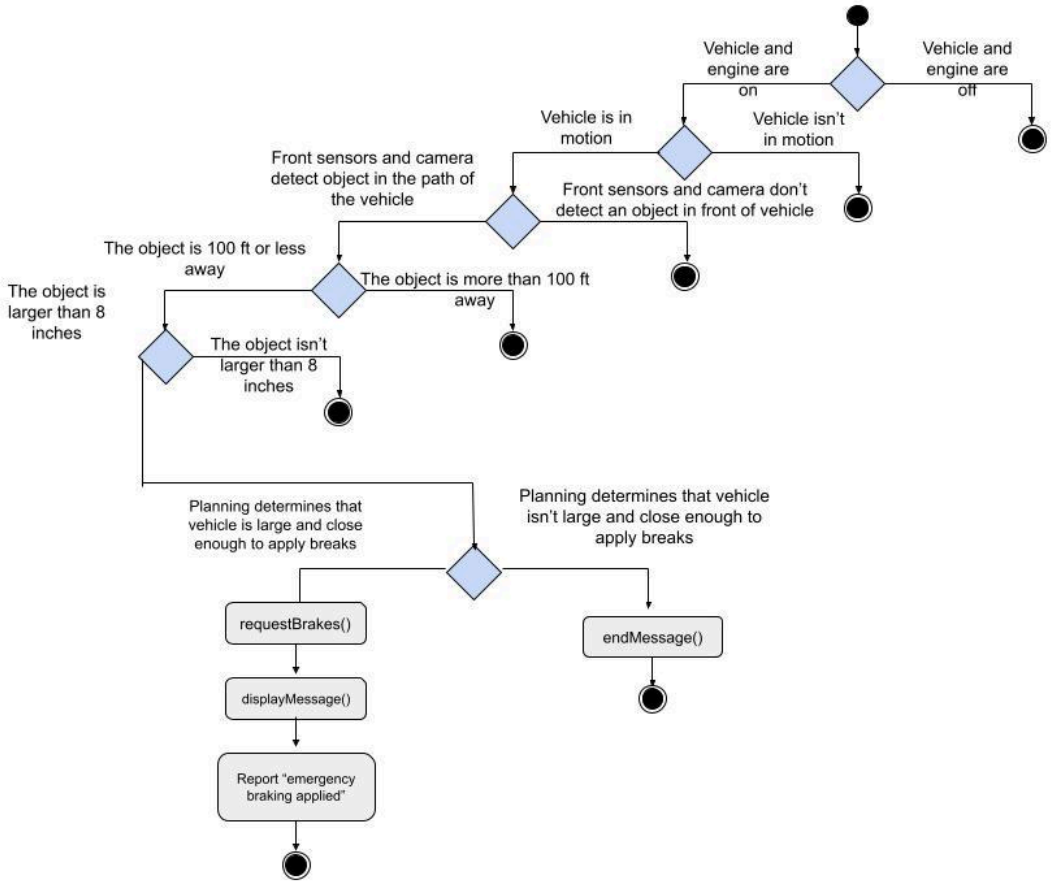
7. System Management/Admin completes the “sign off” request, and signs off the Technician from the system.
8. The display on the laptop screen again shows a prompt requesting a username and password.
9. The Technician disconnects the USB connection between the car and the laptop.
10. All modules update the System Management/Admin log file for every action they successfully and unsuccessfully perform.

1.4.2 Activity Diagrams

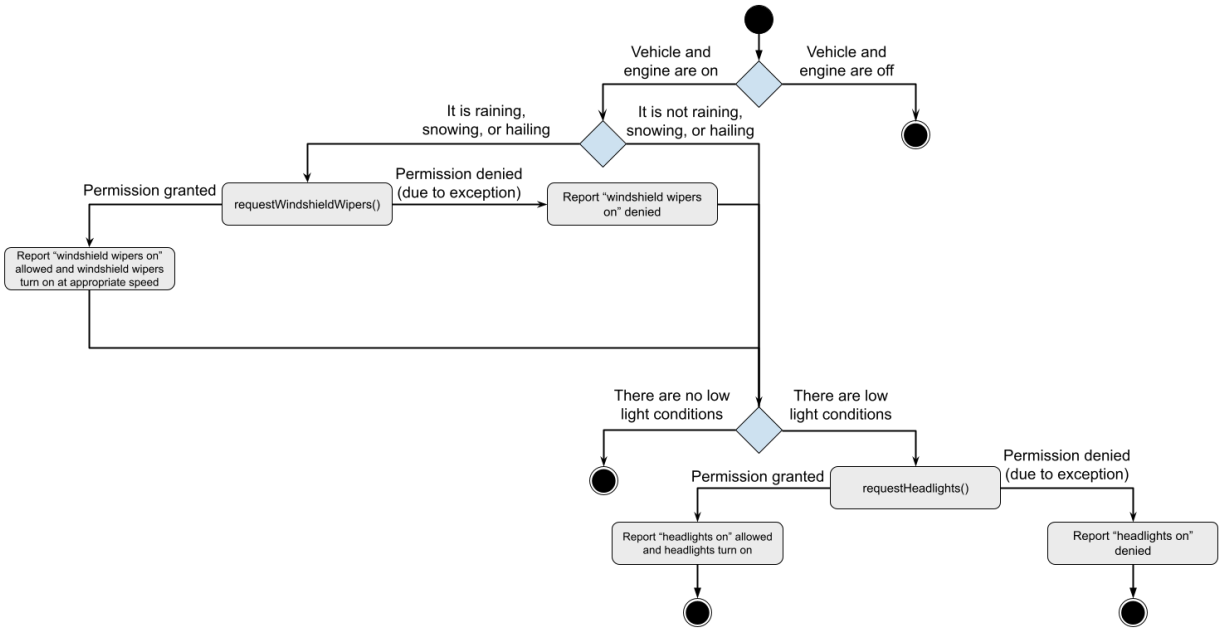
1.4.2.1 Driver Starts the Car



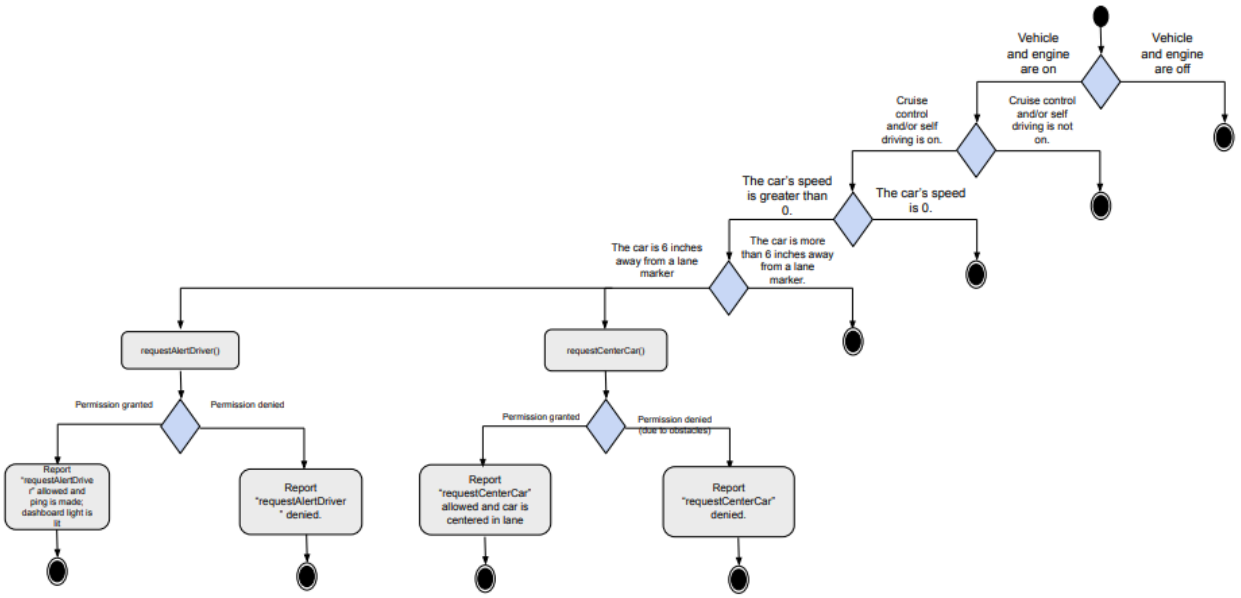
1.4.2.2 Potential Crash Detection



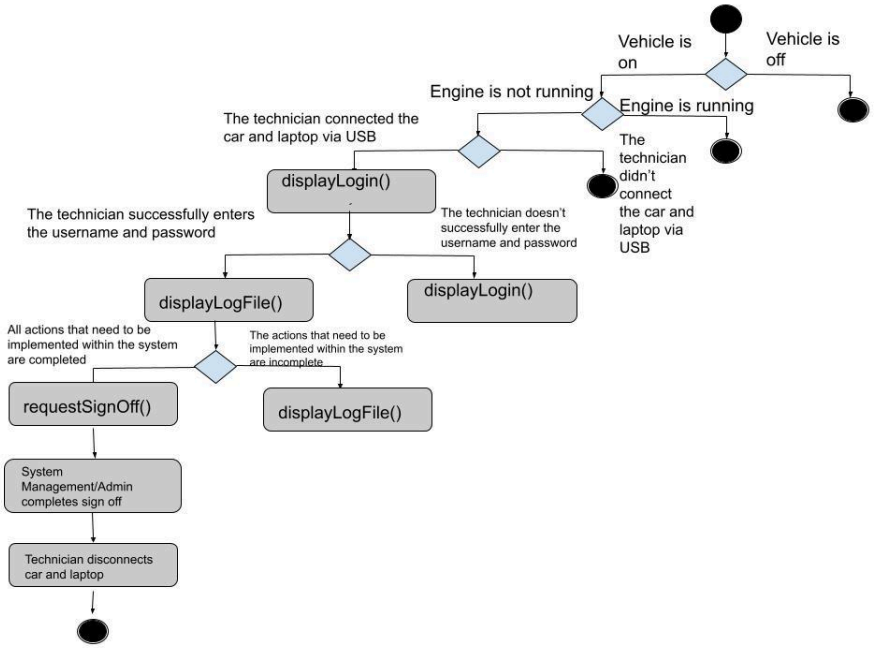
1.4.2.3 Adverse Driving Conditions



1.4.2.4 Lane Assistance for Assisted Driving

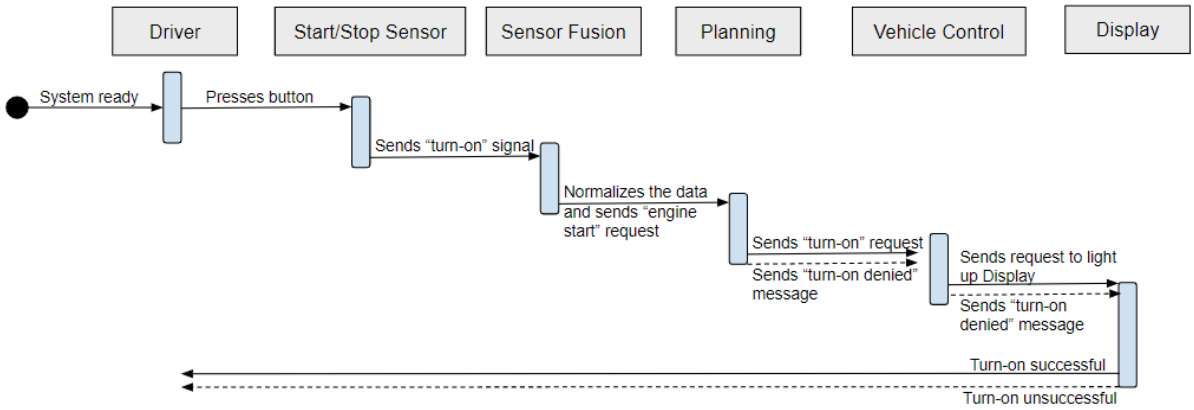


1.4.2.5 Technician Sign On and Sign Off

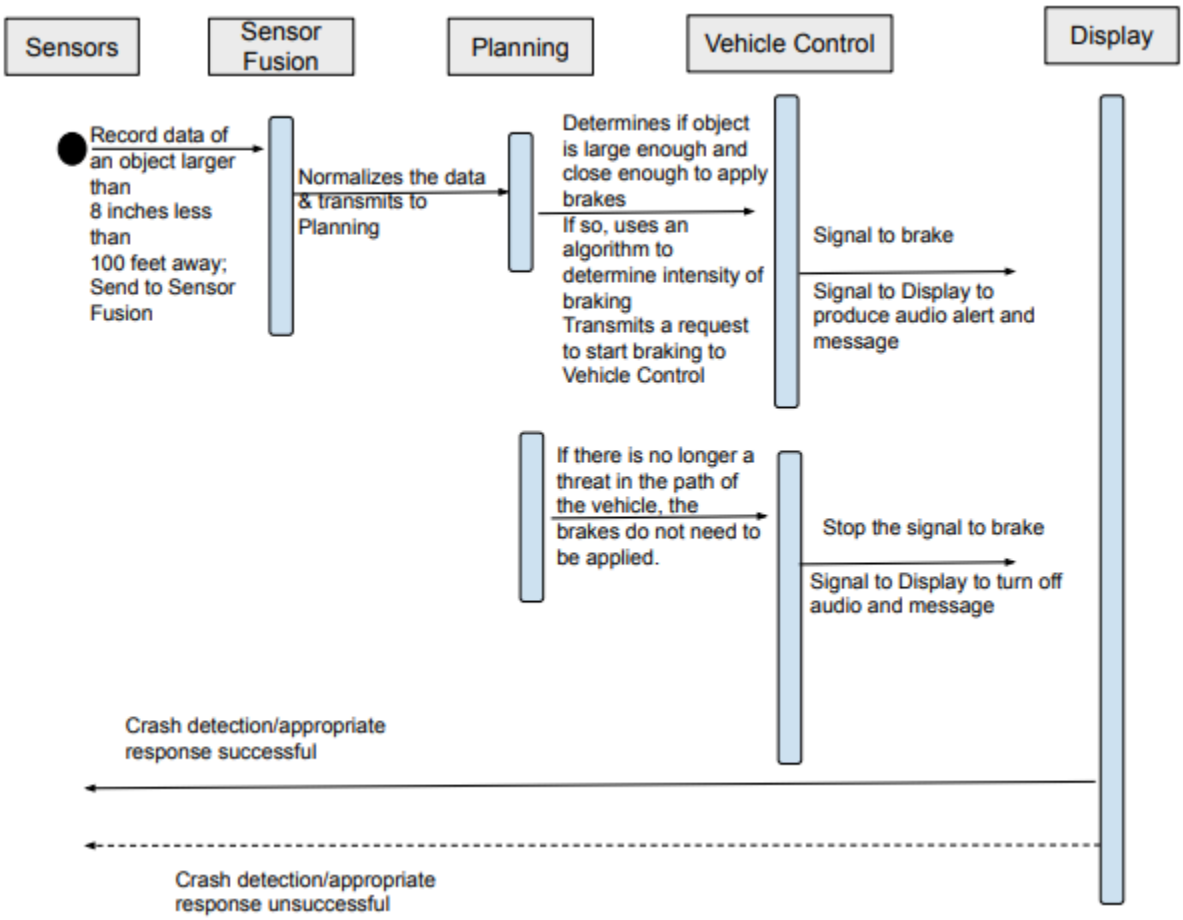


1.4.3 Sequence Diagrams

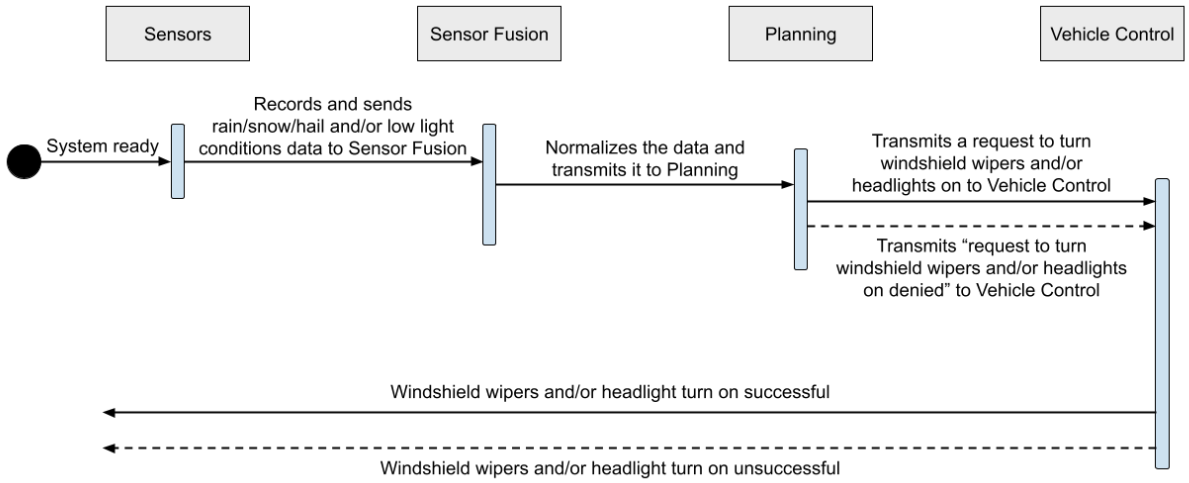
1.4.3.1 Driver Starts the Car



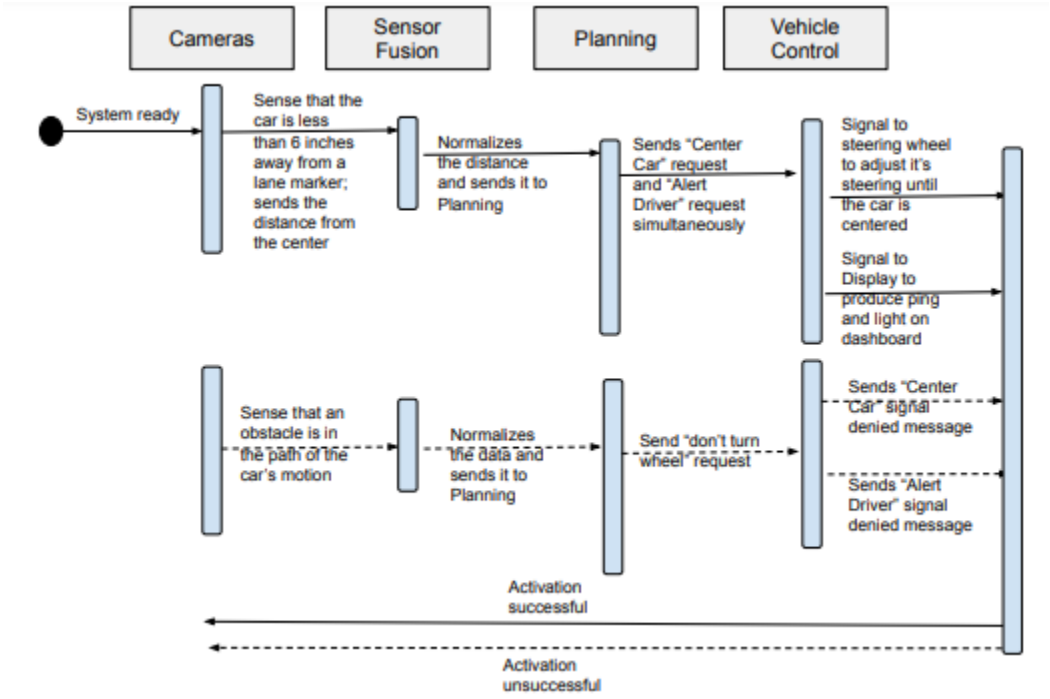
1.4.3.2 Potential Crash Detection



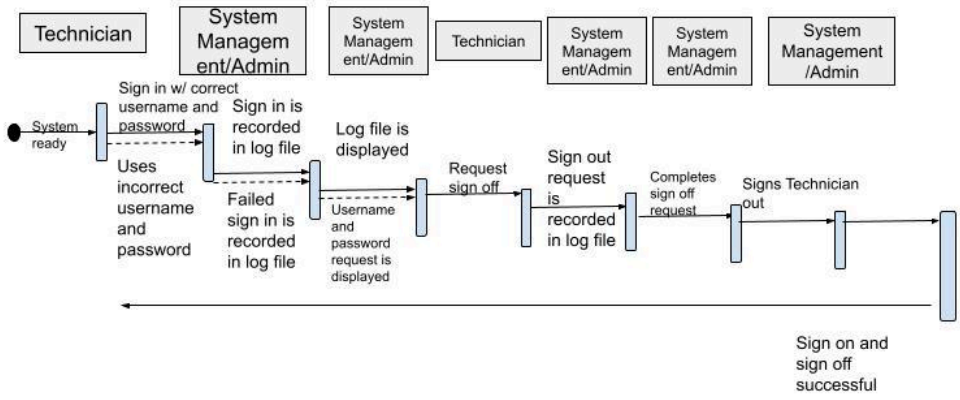
1.4.3.3 Adverse Driving Conditions



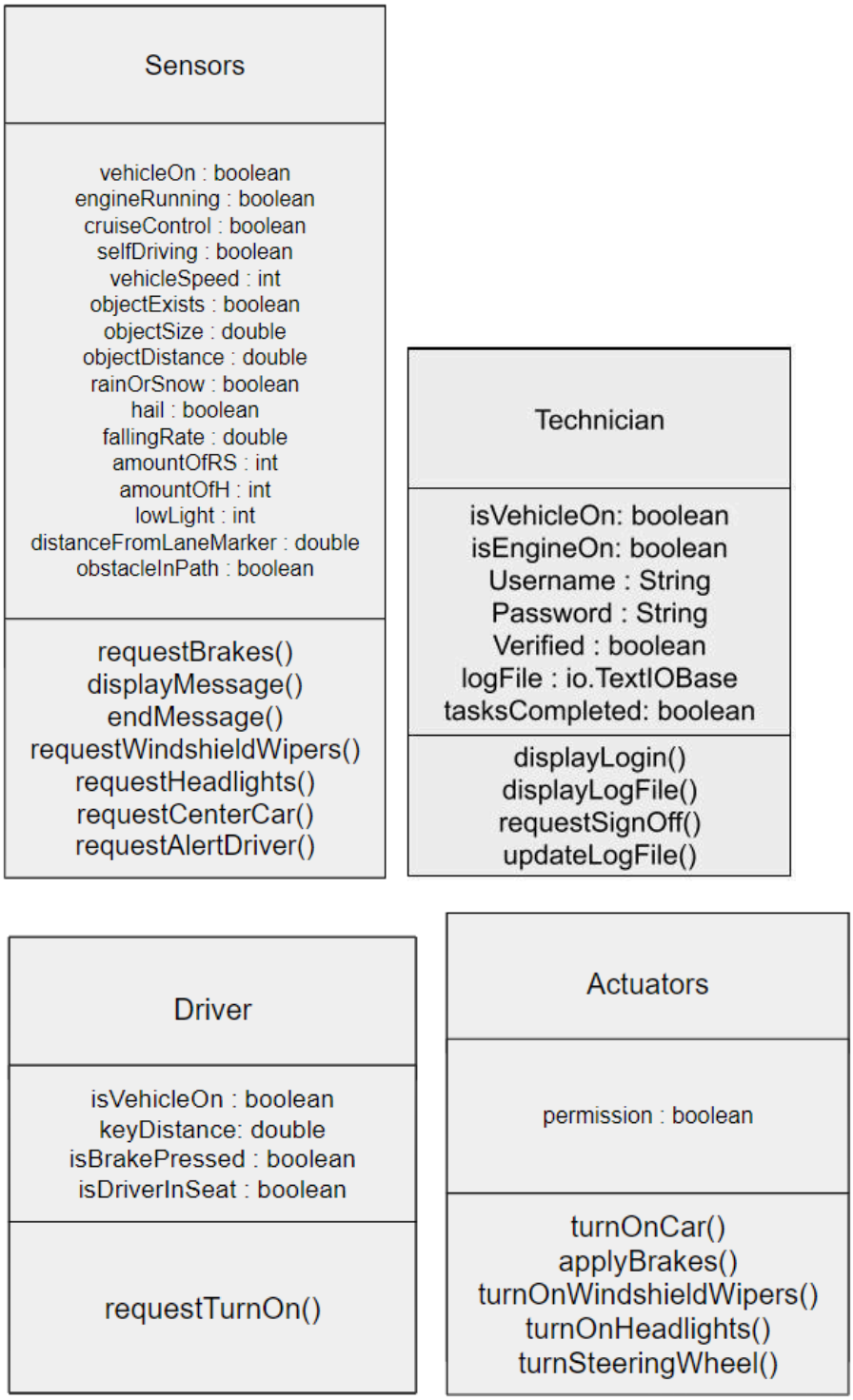
1.4.3.4 Lane Assistance for Assisted Driving



1.4.3.5 Technician Signs On and Off

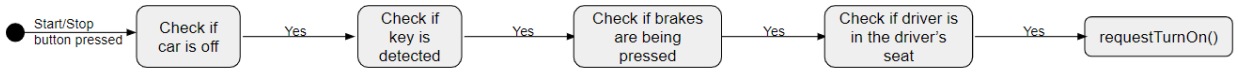


1.4.4 Classes

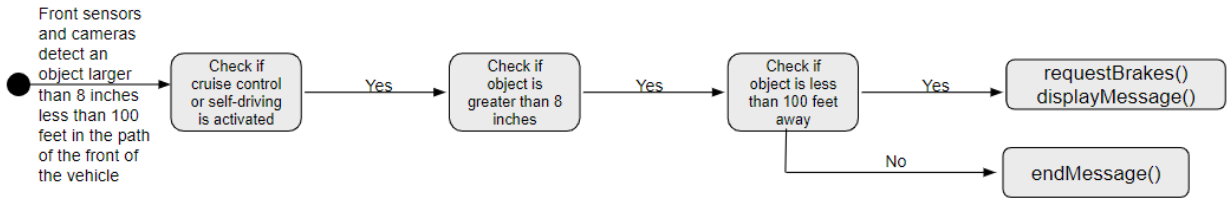

Sensors
vehicleOn : boolean
engineRunning : boolean
cruiseControl : boolean
selfDriving : boolean
vehicleSpeed : int
objectExists : boolean
objectSize : double
objectDistance : double
rainOrSnow : boolean
hail : boolean
fallingRate : double
amountOfRS : int
amountOfH : int
lowLight : int
distanceFromLaneMarker : double
obstacleInPath : boolean
requestBrakes()
displayMessage()
endMessage()
requestWindshieldWipers()
requestHeadlights()
requestCenterCar()
requestAlertDriver()
Technician
isVehicleOn: boolean
isEngineOn: boolean
Username : String
Password : String
Verified : boolean
logFile : io.TextIOBase
tasksCompleted: boolean
displayLogin()
displayLogFile()
requestSignOff()
updateLogFile()
Driver
isVehicleOn : boolean
keyDistance: double
isBrakePressed : boolean
isDriverInSeat : boolean
requestTurnOn()
Actuators
permission : boolean
turnOnCar()
applyBrakes()
turnOnWindshieldWipers()
turnOnHeadlights()
turnSteeringWheel()

1.4.5 State Diagrams

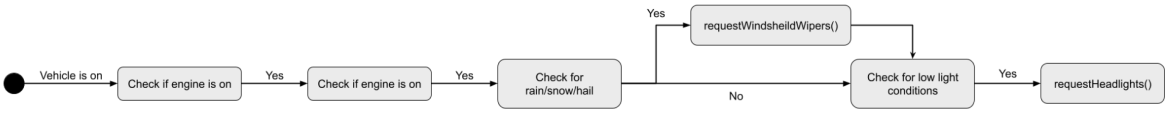
1.4.5.1 Driver Starts the Car



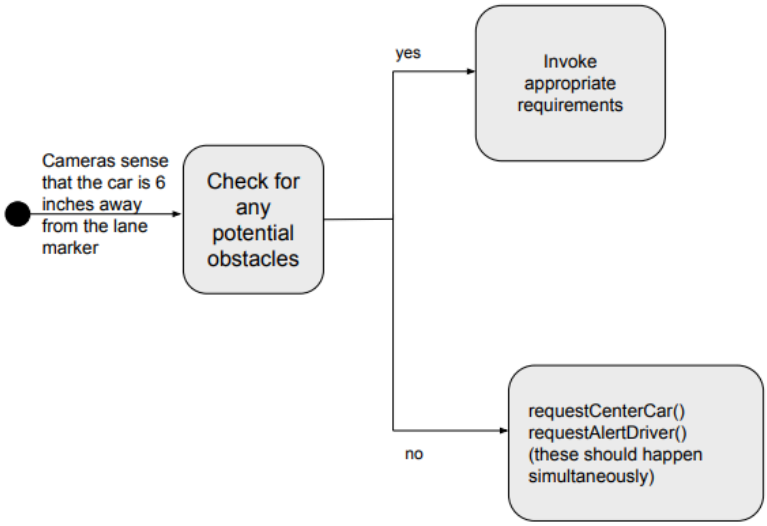
1.4.5.2 Potential Crash Detection



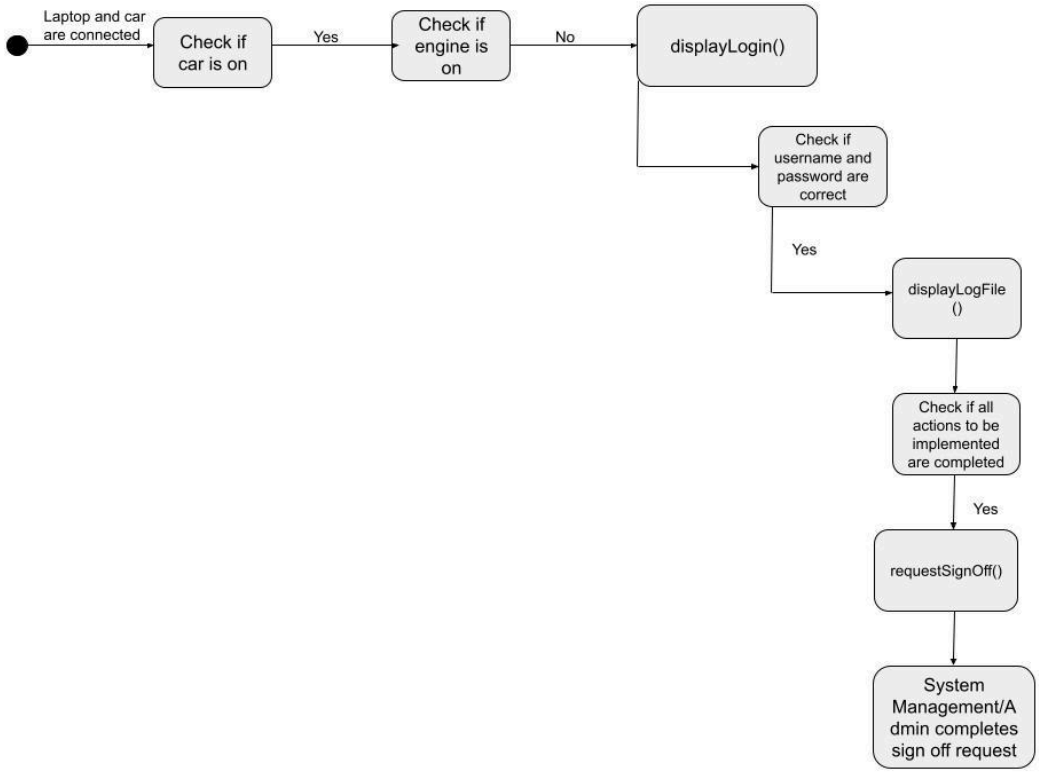
1.4.5.3 Adverse Driving Conditions



1.4.5.4 Lane Assistance for Assisted Driving



1.4.5.5 Technician Signs On and Signs Off



1.5 Design

This section focuses on designing the software architecture, interface, and component-level details for IoT HTL. It includes an evaluation of various software architecture models for their feasibility in IoT HTL, as well as discussing specific pros and cons. Based on this evaluation, the most suitable software architectures are chosen. Also, the interface design (subsection 1.5.2) defines how software elements, hardware elements, and end-users communicate, including both Technician and Driver interfaces. Additionally, the exact specifications for each user interface are given. Finally, in subsection 1.5.3, the component-level design is described, transforming structural elements into detailed procedural descriptions of software components. Overall, this section aims to provide a guide for coding implementation.

1.5.1 Software Architecture

1.5.1.1 Data-Centered Architecture

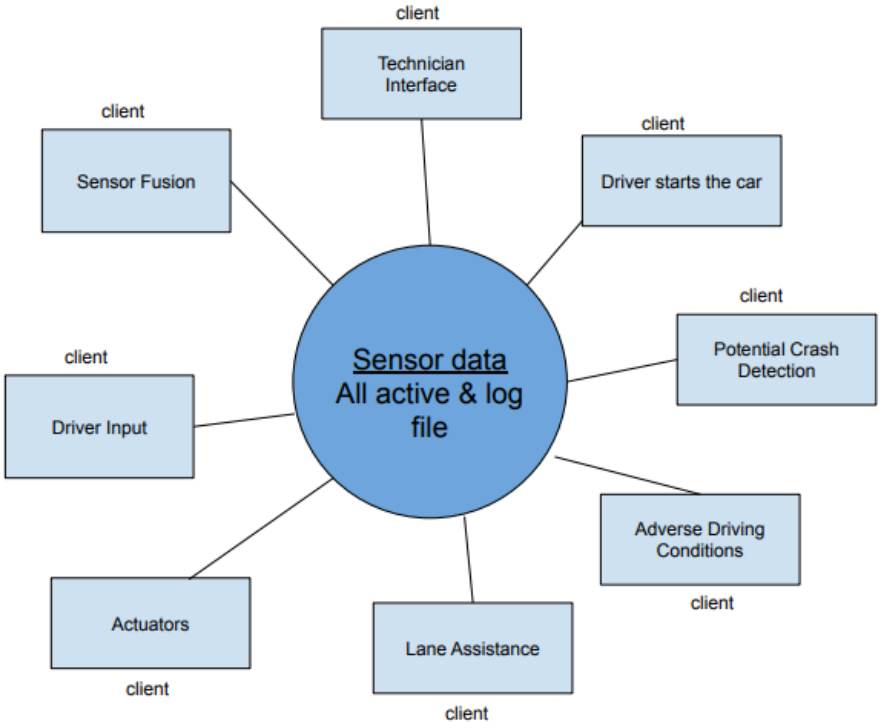
- This architecture is not a good fit for IoT HTL because of its data-centered framework and how the main aspect of the architecture focuses on storing data instead of processing data in an effort to communicate across different fields.

Pros:

- Client software can access the centralized data store and communicate with each other through this shared resource without modifying other client components
- Can easily add more client components to the architecture

Cons:

- Dependency on one single source of data (the centralized data-store)
- Clients require fast retrieval of the data
- Single point of failure
- Not suitable for a real-time system application because of slow queries to the database



1.5.1.2 Data Flow Architecture

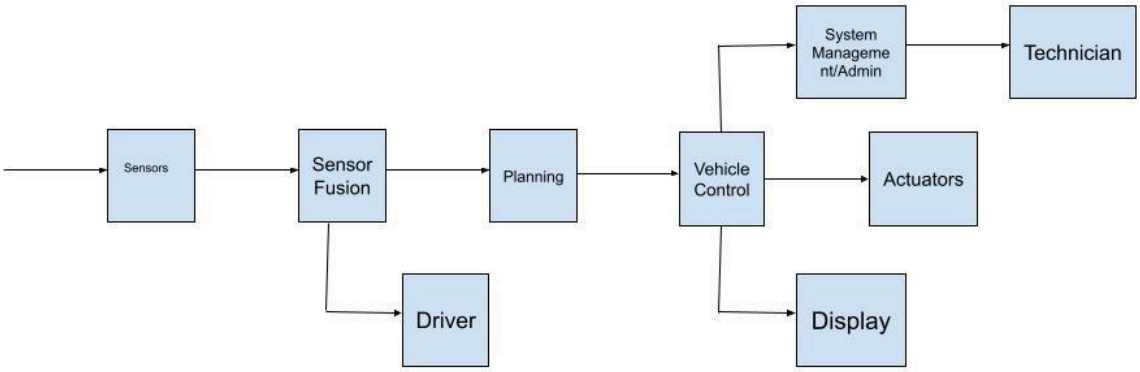
- This architecture is a good fit for the IoT HTL because this architecture is flexible and scalable. It is easy to add more modules as needed and components can be modified as needed without disrupting the entire system. For example, it can be used for the adverse driving conditions use case because the sensor filter can input data into the sensor fusion filter, then planning, then vehicle control, and everything can be recorded in the System Management log file.

Pros:

- Step-by-step checking of conditions based on input simplifies the design behind implementing use cases
- New filters can be added easily, making this architecture flexible
- Each filter working independently of pipes allows filters to be replaced or reconfigured without affecting the overall architecture

Cons:

- The design of this architecture could be very complex if inputs are connected in many different ways
- Tracing the flow of data to identify the source of errors can be time-consuming



1.5.1.3 Call Return Architecture

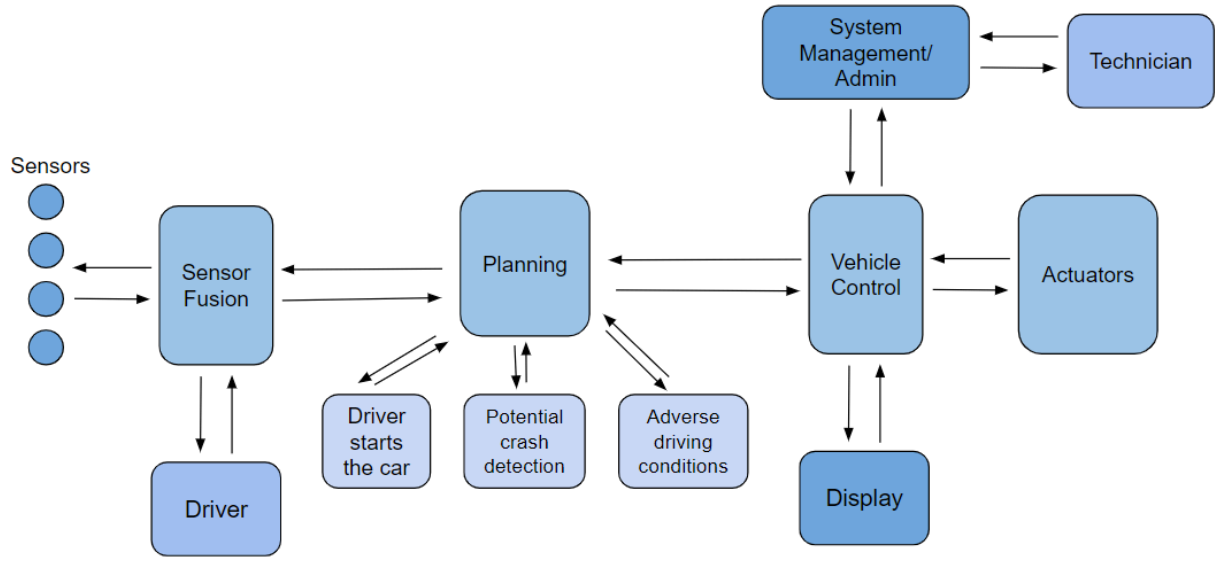
- This architecture provides a clear sequence of how functions call one another and the flow of actions and data returns. However, due to the fact that these use cases are being implemented in a real-time system with a clear flow of data that depends on pushing instead of returning data, this would not be ideal for the IoT HTL. As shown in the diagram below, each module has to call and request the data before receiving it, which makes this architecture inadequate. For instance, Sensor Fusion has to call the sensors and then the sensors return the data in order for Sensor Fusion to receive the data. This is impractical for a real-time system.

Pros:

- Enables code modularity by breaking down programs into smaller subroutines such as Sensor Fusion, Planning, Vehicle Control, etc
- Explicitly shows the control flow by following a clear sequence of function calls and returns
- Easy to modify for simple and small programs
- Easy to understand and visualize

Cons:

- Not suitable for a real-time system because real-time systems require pushing data when it becomes available instead of always having to request it
- Can become complex if functions are interconnected in many different ways
- Hard to scale especially as the number of functions increases and the interconnections between functions become more complex
- If a decision-making portion gets flooded with data it may not respond at the right time which is necessary for a real-time system.



1.5.1.4 Object-Oriented Architecture

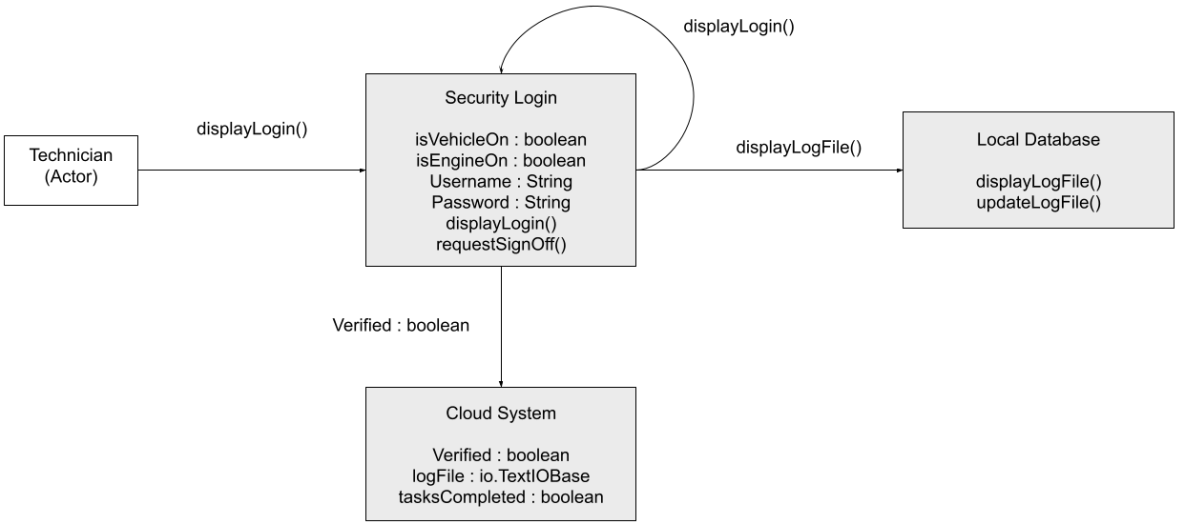
- Object-Oriented Architecture proves to be a good fit for IoT HTL because it is easily designable to fit the many requirements of the use cases as described in Section 1.4. It splits up the different processes to be implemented into different sections, allowing for smaller tasks and goals to be completed while implementing the code. It also clearly defines the direct communication between classes and objects that are integrated within the IoT HTL. This would likely prove to be beneficial in the case of the technician, where a login is required and many different data fields and methods are used according to whether or not a login attempt is successful, whether the technician is authorized to make changes to the log file, etc.

Pros:

- Appears to hold potential in the implementation of previously-defined use cases, especially those that take similar actions (could benefit from reusable code).
- Compartmentalizes different processes through the use of objects, allowing for the overall project to be separated into easier-to-tackle and easier-to-track components.
- Use of objects allows for data that isn't being modified (i.e. data that does not belong to the object being accessed) to be protected from modification.
- Flow appears to lend itself to the implementation of a real-time system.

Cons:

- Use of objects may not be ideal when dealing with use cases that require the modification of various variables belonging to different objects.
- Objects may complicate implementation when it comes to the interconnected nature of some of the procedures.



1.5.1.5 Layered Architecture

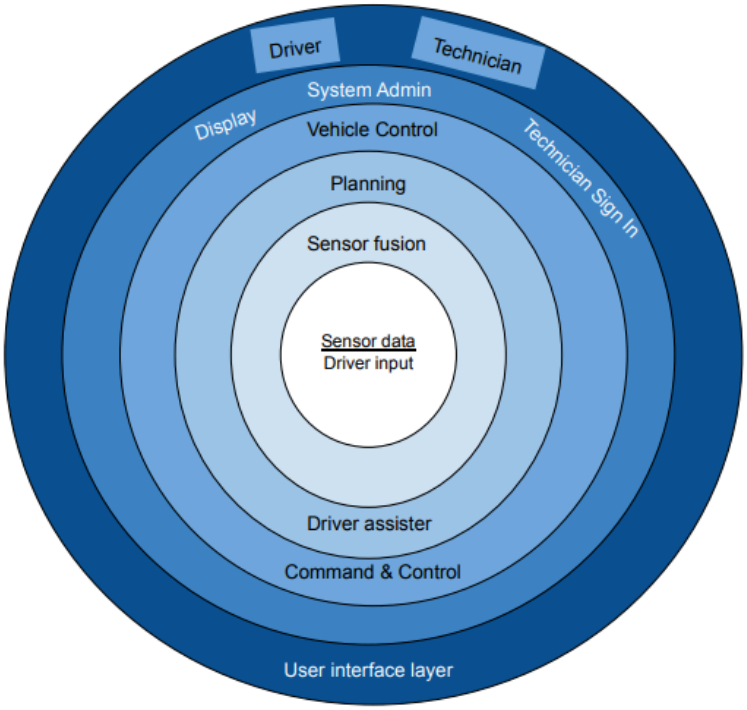
- This architecture would not be an adequate fit for IoT HTL because there is not a core set of operations that are being interfaced from. Although a user and technician interface is a prominent aspect of IoT HTL, utilizing a layered architecture would not provide the communication between different fields that are needed.

Pros:

- The layers are independent of each other
- Dependencies between layers are minimized
- Easy to debug because the layers have strict limits to what is inside, so it is easier to find where something might be going wrong
- Easy to add security features among layers

Cons:

- Lack of communication/dependency between layers
- Timing is an issue because the layers are independent of each other
- Not supportive of a mission-critical real-time system because data is accumulated in the center and must be sent outward to each layer



1.5.1.6 Model View Controller Architecture

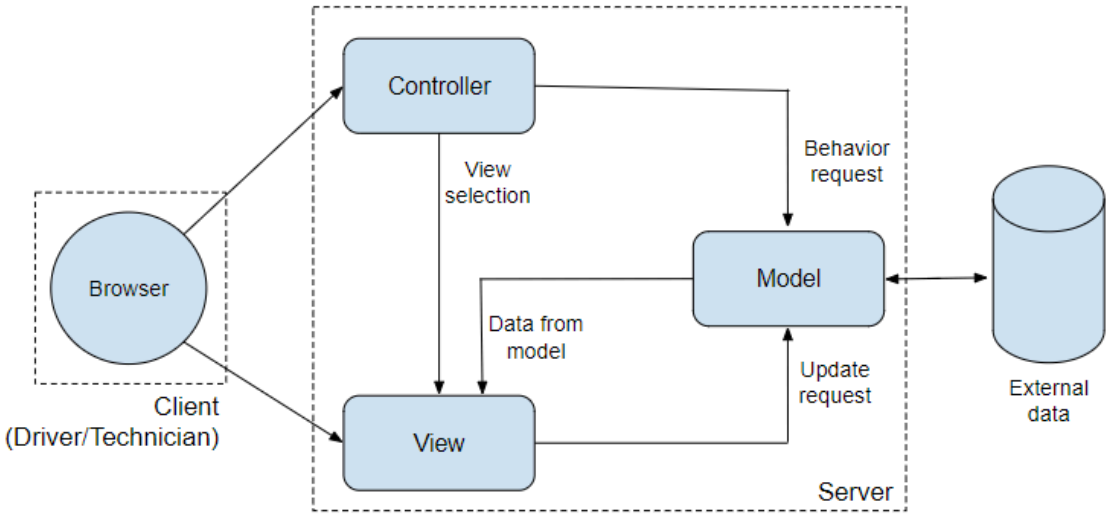
- This architecture is not a fit for the data and real-time aspect of this project because it is better suited for organizing large-size web applications and projects in that manner instead of self and assisted-driving software. However, it can be helpful for the Technician and Driver Display and with the presentation of the data and information. As the diagram demonstrates, the Driver and Technician can interact with the Display interface to view the data about the condition of the vehicle and invoke actions by pressing buttons on the screen.

Pros:

- Well-suited for presenting and displaying data and information
- Easy to maintain, scale, modify, and test because of the clear distinctions between each component
- Develops different view components to gain a better understanding of how the software is used
- Useful to represent the part of the vehicle that the user interacts with and sees

Cons:

- Increases complexity and overhead of code because of having to create and manage three components possibly ending up with more classes and functions than needed
- Not suitable for real-time systems



1.5.1.7 Finite State Machine Architecture

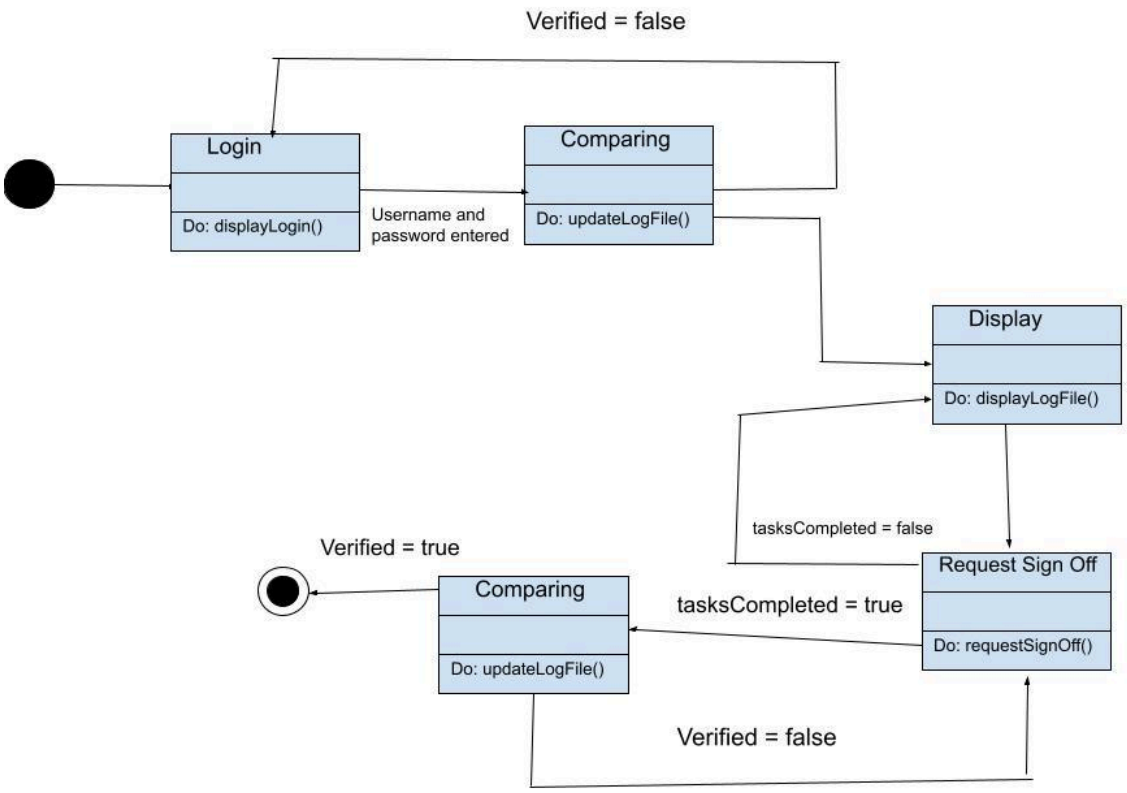
- This architecture would be a fit for IoT HTL because the states can work as conditions and generate the correct output depending on the input. This architecture simplifies verifying conditions in use cases, especially for the Technician.

Pros:

- States that represent the different conditions can be used to implement the use cases
- Easy to verify conditions with inputs
- Suitable for real-time systems that need a high standard of reliability and fault tolerance

Cons:

- Could become difficult to manage with a large number of states
- Time-consuming to implement if the state transition logic is complex



Overall, after considering the pros and cons of each type of software architecture, Object-Oriented architecture is ideal for implementing the use cases and the Finite State Machine architecture is the most reasonable choice for the Technician features.

1.5.2 Interface Design

Driver Interface

- A display on the dashboard that shows any warning lights and the status of sensors and other vital components of the vehicle by default
- A part of the display that shows the driver the speed at which the vehicle is going, the vehicle's route/directions (when cruise control/self-driving is on and/or the driver is en route somewhere)
- A section of the display that allows the driver to adjust settings related to the display (such as icon and text size, day/night mode, etc.) and manually turn things such as cruise control/self-driving, windshield wipers, and headlights on or off.

  Class:
  - Driver

  Data:
  - currentSpeed : int
  - setDestination : string

- directions : string
- inSelfDriving : boolean
- inAssistedDriving : boolean
- inManual : boolean
- cruiseControl : boolean
- needSoftwareUpdate : boolean
- windshielfWipersOn : boolean
- headlightsOn : boolean
- messageAlerts : string
- pingAlert : boolean

Functions:

- turnOnCruiseControl()
- turnOnSelfDriving()
- turnOnAssistedDriving()
- turnOnManualDriving()
- turnOnWindshieldWipers()
- turnOnHeadlights()
- setDestination()

Technician Interface

- Sign-in page that leads to a directory with access to the log file and any other information regarding the vehicle
- After successfully signing in, the Technician has access to diagnostic and software tools to implement any software updates and fixes to the vehicle. Also, the Technician has access to oversee and monitor the actuators
- A dashboard that shows any warning lights as well as the status of sensors and other vital components of the vehicle

Class:

- Technician

Data:

- Username : string
- Password : string
- Verified : boolean
- logFile : io.TextOBase
- tasksCompleted : boolean
- vehicleOn : boolean

Functions:

- displayLogin()
- displayLogFile()
- requestSignOff()
- updateLogFile()

1.5.3 Component-Level Design

1.5.3.1 Sensor Fusion, Planning, Vehicle Control, and Driver Actions Component

Data:

- vehicleOn : boolean
- cruiseControl : boolean
- selfDriving : boolean
- vehicleSpeed : int
- objectExists : boolean
- objectSize : double
- objectDistance : double
- rainOrSnow : boolean
- hail : boolean
- fallingRate : boolean
- amountofRS : int
- amountofH : int
- lowLight : boolean
- distanceFromLaneMarker : int
- obstacleInPath : boolean
- keyDistance : int
- isBrakePressed : boolean
- isDriverInSeat : boolean

Functions:

- requestBrakes()
- displayMessage()
- endMessage()
- requestWindshieldWipers()
- requestHeadlights()
- requestCenterCar()
- requestAlertDriver()
- requestTurnOn()
- turnOnCar()
- turnSteeringWheel()
- applyBrakes()
- turnOnWindshieldWipers()
- turnOnHeadlights()

Interactions/Triggers

- Driver input by pressing the brakes or gas as well as pressing buttons
- The vehicle makes decisions and performs actions deemed necessary

1.5.3.2 System Admin/Management and Technician Interface

Data:

- username : string
- password : string
- verified : boolean
- logFile : io.TextOBase
- tasksCompleted : boolean
- vehicleOn : boolean

Functions:

- displayLogin()
- displayLogFile()
- requestSignOff()
- updateLogFile()

Interactions/Triggers

- Technician plugging in their laptop via USB
- Buttons and text boxes for user input
- The Technician can see a sign-in screen and after successful authentication, has access to a page that allows them to use diagnostic and software tools

1.5.3.3 Driver Display

Data:

- currentSpeed : int
- setDestination : string
- directions : string
- inSelfDriving : boolean
- inAssistedDriving : boolean
- inManual : boolean
- cruiseControl : boolean
- needSoftwareUpdate : boolean
- windshielfWipersOn : boolean
- headlightsOn : boolean
- messageAlerts : string
- pingAlert : boolean

Functions:

- turnOnCruiseControl()
- turnOnSelfDriving()
- turnOnAssistedDriving()
- turnOnManualDriving()
- turnOnWindshieldWipers()
- turnOnHeadlights()
- setDestination()

Interactions/Triggers

- Buttons, toggle switches, and sliders for driver input
- Displays the status of different features of the car indicating if they are on or off as well as the current speed of the vehicle
- Displays any potential alerts that the driver should be aware of

1.6 Project Code

Sensor and Sensor Fusion Components:

This component reads from a file that is passed as an argument which contains all the data accumulated by the sensors. The order of the sensor data in the input file is as follows so each corresponding line in the file contains the data for that attribute:

1. *isVehicleOn
2. *isEngineOn
3. *isBrakePressed
4. *isDriverInSeat
5. *inPark
6. *cruiseControl
7. *selfDriving
8. *objectExists
9. *isStopSignTrafficLight
10. *rainOrSnowStatus
11. *hailStatus
12. *doorLockStatus
13. *obstacleInPath
14. *crashDetected
15. *airbagsDeploymentStatus
16. *isStartButtonPressed
17. *isRemoteStartButtonPressed
18. *isTurnSignalOn
19. *isComputerConnected
20. distanceFromLaneMarker
21. lightStatus
22. keyDistance
23. vehicleSpeed
24. stopSignTrafficLightDistance
25. amountOfRS
26. amountOfH
27. objectSize
28. objectDistance
29. speedLimit
30. fallingRate

‘ * ‘ indicates that the attribute is a boolean but will be represented as an integer in the data file.

If there are multiple entries for one type of data, if it is a boolean only the last entry is recorded, and if it is an integer the average is calculated and recorded. If there is only one entry, then that is recorded. All this raw data is put into an integer array which then Sensor Fusion iterates through and checks if each data entry is valid within the context of what it is measuring. If there is an error, a -1 is returned and assigned to the attribute to indicate that there is no valid data value. If a boolean is invalid it returns false. Also, in both cases, a message is printed to indicate

the absence of a valid value of data. Sensor Fusion also provides getter methods that Planning can call to retrieve the data for each attribute to use when testing conditionals.

Here is an example of what the input file looks like:

```
* 1
* 1
* 1 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 1
* 0
* 0
* 0
4
100 200
300
0
-1
0
0
0
-1
-1
-1
```

```
import javax.print.attribute.IntegerSyntax;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class SensorFusion {
    private boolean isVehicleOn, isEngineOn, isBrakePressed, isDriverInSeat, inPark,
cruiseControl, selfDriving, objectExists;
    private boolean detectedStopSignTrafficLight, rainOrSnowStatus, hailStatus,
doorLockStatus, obstacleInPath;
    private boolean crashDetected, airbagsDeploymentStatus;
    private boolean isStartButtonPressed, isRemoteStartButtonPressed, isTurnSignalOn,
isComputerConnected;
    private int lightStatus, vehicleSpeed, amountOfRS, amountOfH, speedLimit;
    private double distanceFromLaneMarker, keyDistance, stopSignTrafficLightDistance,
objectSize, objectDistance, fallingRate;
    private double wipesPerSec;
    private boolean headlightsStatus, call911;
    private int displayMode;

    public SensorFusion(String filename){
        int [] data = new int[30]; //stores normalized sensor data
        int i = 0;
        try {
```

```
            // Read data from file
            File dataFile = new File(filename);
            Scanner myReader = new Scanner(dataFile);
            while (myReader.hasNextLine()) {
                String line = myReader.nextLine();
                String[] sensorData = line.split(" ");
                if (sensorData.length == 1) {
                    // If there is only one value for that type of data then put it into the
array
                    data[i] = Integer.valueOf(sensorData[0]);
                } else {
                    if (sensorData[0].equals("*")) {
                        // '*' indicates it is a boolean so take only the last value
                        data[i] = Integer.valueOf(sensorData[sensorData.length - 1]);
                    } else {
                        // If not a boolean, take the average of the data
                        int average = 0;
                        for (String value : sensorData) {
                            average += Integer.valueOf(value);
                        }
                        average = average / sensorData.length;
                        data[i] = average;
                    }
                }
                i++;
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
        }

        isVehicleOn = validateBoolean(data[0], 0);
        isEngineOn = validateBoolean(data[1], 1);
        isBrakePressed = validateBoolean(data[2],2);
        isDriverInSeat = validateBoolean(data[3],3);
        inPark = validateBoolean(data[4],4);
        cruiseControl = validateBoolean(data[5],5);
        selfDriving = validateBoolean(data[6],6);
        objectExists = validateBoolean(data[7],7);
        detectedStopSignTrafficLight = validateBoolean(data[8],8);
        rainOrSnowStatus = validateBoolean(data[9],9);
        hailStatus = validateBoolean(data[10],10);
        doorLockStatus = validateBoolean(data[11],11);
        obstacleInPath = validateBoolean(data[12],12);
        crashDetected = validateBoolean(data[13],13);
        airbagsDeploymentStatus = validateBoolean(data[14],14);
        isStartButtonPressed = validateBoolean(data[15],15);
        isRemoteStartButtonPressed = validateBoolean(data[16],16);
        isTurnSignalOn = validateBoolean(data[17],17);
        isComputerConnected = validateBoolean(data[18],18);
```

```
        distanceFromLaneMarker = validateNum(data[19],19);
        lightStatus = validateNum(data[20],20);
        keyDistance = validateNum(data[21],21);
        vehicleSpeed = validateNum(data[22],22);
        stopSignTrafficLightDistance = validateNum(data[23],23);
        amountOfRS = validateNum(data[24],24);
        amountOfH = validateNum(data[25],25);
        objectSize = validateNum(data[26],26);
        objectDistance = validateNum(data[27],27);
        speedLimit = validateNum(data[28], 28);
        fallingRate = validateNum(data[29], 29);
    }
    private boolean validateBoolean(int dataValue, int i){
        //assign correct value, true or false, to the boolean
        if(dataValue == 1){
            return true;
        }else if(dataValue == 0){
            return false;
        }else{
            System.out.println("No valid data entry: " + (i + 1));
            return false;
        }
    }
    private int validateNum(int datavalue, int i){
        if(datavalue < 0){
            //return -1 to signify an error or if there is no value for that data type
            System.out.println("No valid data entry: " + (i + 1));
            return -1;
        }else{
            return datavalue;
        }
    }
    //get methods that can be called from other classes to retrieve the data
    public boolean getVehicleStatus(){return isVehicleOn;}
    public boolean getEngineStatus(){return isEngineOn;}
    public boolean getBrakePressed(){return isBrakePressed;}
    public boolean getDriverInSeat(){return isDriverInSeat;}
    public boolean getInPark(){return inPark;}
    public boolean getCruiseControl(){return cruiseControl;}
    public boolean getSelfDriving(){return selfDriving;}
    public boolean getObjectExists(){return objectExists;}
    public boolean getStopSignOrTrafficLightDetected(){return detectedStopSignTrafficLight;}
    public boolean getRainOrSnowStatus(){return rainOrSnowStatus;}
    public boolean getHailStatus(){return hailStatus;}
    public boolean getDoorLockStatus(){return doorLockStatus;}
    public boolean getObstacleInPath(){return obstacleInPath;}
    public boolean getCrashDetected(){return crashDetected;}
    public boolean getAirbagsDeploymentStatus(){return airbagsDeploymentStatus;}
    public boolean getStartButtonPressed() {return isStartButtonPressed;}
    public boolean getRemoteStartButtonPressed() {return isRemoteStartButtonPressed;}
```

```
    public boolean getTurnSignalStatus() {return isTurnSignalOn;}
    public boolean getComputerConnectionStatus(){return isComputerConnected;}

    public double getDistanceFromLaneMarker(){return distanceFromLaneMarker;}
    public int getLightStatus(){return lightStatus;}
    public double getKeyDistance(){return keyDistance;}
    public int getVehicleSpeed(){return vehicleSpeed;}
    public double getStopSignTrafficLightDistance(){return stopSignTrafficLightDistance;}
    public int getAmountOfRS(){return amountOfRS;}
    public int getAmountOfH(){return amountOfH;}
    public double getObjectSize(){return objectSize;}
    public double getObjectDistance(){return objectDistance;}
    public int getSpeedLimit(){return speedLimit;}
    public double getFallingRate(){return fallingRate;}

    //set methods that can be called from other classes to update the data
    public void setVehicleStatus(boolean newValue){ isVehicleOn = newValue;}
    public void setEngineStatus(boolean newValue){ isEngineOn = newValue;}

    public void setBrakePressed(boolean newValue){ isBrakePressed = newValue;}
    public void setInPark(boolean newValue){inPark = newValue;}
    public void setCruiseControl(boolean newValue){cruiseControl = newValue;}
    public void setSelfDriving(boolean newValue){selfDriving = newValue;}
    public void setDoorLockStatus(boolean newValue){doorLockStatus = newValue;}
    public void setTurnSignalStatus(boolean newValue) {isTurnSignalOn = newValue;}
    public void setComputerConnectionStatus(boolean newValue){isComputerConnected = newValue;}
    public void setVehicleSpeed(int newValue){vehicleSpeed = newValue;}
    public void setWipesPerSec(double newValue){wipesPerSec = newValue;}
    public void setHeadlightsStatus(boolean newValue){headlightsStatus = newValue;}
    public void setDisplayMode(int newValue){displayMode = newValue;}
    public void set911Call(boolean newValue){call911 = newValue;}
    public void setDistanceFromLaneMarker(double newValue){distanceFromLaneMarker = newValue;}
}
```

Planning and Vehicle Control Components:

```
public class Planning {
   public SensorFusion sensFus;
   // Class Attributes
   private boolean turnOnCar;
   private boolean turnOnCarRemote;
   private boolean turnOffCar;
   private boolean gradualBrakingStop;
   private boolean gradualBrakingEmergency;
   private boolean applyBrakes;
   private boolean turnOnWindshieldWipers;
   private boolean turnOnHeadlights;
   private boolean lockDoors;
   private boolean turnSteeringWheel;
   private boolean slowDown;
   private boolean speedUp;
   private boolean dial911;
   private double wipesPerSec;

   // Default Constructor
```

```
    public Planning(String filename) {
        sensFus = new SensorFusion(filename);
        mainPlan();
    }

    /**
     * Function to interpret sensor data and inform VCS about
     * what to do.
     * @return void
     */
    private void mainPlan() {
        // If Vehicle is not on
        if(!sensFus.getVehicleStatus()) {
            // 1.3.1.1.1 Driver Starts the Car
            if(sensFus.getStartButtonPressed()) {
                if(sensFus.getKeyDistance() >= 0 && sensFus.getKeyDistance() <= 1.5 &&
sensFus.getDriverInSeat() && sensFus.getBrakePressed()) {
                    turnOnCar = true;
                }
            } else if(sensFus.getRemoteStartButtonPressed()) {
                // 1.3.1.1.2 Remote Start
                if(sensFus.getKeyDistance() >=0 && sensFus.getKeyDistance() <= 15) {
                    turnOnCarRemote = true;
                }
            }
        } else {
            // If Vehicle is on


            // If Driver presses start button and Vehicle is on
            if(sensFus.getStartButtonPressed()) {
                // 1.3.1.1.3 Driver Turns Off the Car
                if(sensFus.getVehicleSpeed() == 0 && sensFus.getInPark()) {
                    turnOffCar = true;
                }
            } else if((sensFus.getCruiseControl() || sensFus.getSelfDriving()) &&
sensFus.getVehicleSpeed() > 0 && (sensFus.getDistanceFromLaneMarker() >= 0 &&
sensFus.getDistanceFromLaneMarker() < 6 && !sensFus.getTurnSignalStatus())) {   //dist from lane
marker >= 0 error handling
                // 1.3.1.2.1 Lane Assistance for Assisted Driving
                turnSteeringWheel = true;
            } else if((sensFus.getCruiseControl() || sensFus.getSelfDriving()) &&
sensFus.getVehicleSpeed() > 0 && sensFus.getStopSignOrTrafficLightDetected() &&
                    sensFus.getStopSignTrafficLightDistance() >= 0 &&
sensFus.getStopSignTrafficLightDistance() <= 1000) {  //StopSignTrafficLightDistance >= 0 error
handling
                // 1.3.1.2.2 Stop Signs, Traffic Lights, and Automatic Braking
                /*
                 * The Planning module transmits a brake request to Vehicle Control
                 * when the distance is less than or equal to 250 feet with a determined
                 * brake intensity which is calculated using an algorithm based on the
                 * vehicle's current speed and distance from the stop sign or red traffic
                 * light to ensure a safe stop at least 5 feet away from the stop sign or
                 * traffic light.
                 */
                gradualBrakingStop = true;
            }

            //Use case -- Adverse Driving Conditions
            if(sensFus.getRainOrSnowStatus() || sensFus.getHailStatus()) {
                // 1.3.1.2.3 Automated Front Windshield Wipers
                if(sensFus.getAmountOfRS() > 35 || sensFus.getAmountOfH() > 4) {
```

```
                    turnOnWindshieldWipers = true;
                    if(sensFus.getFallingRate() >= 1 && sensFus.getFallingRate() <= 4) {
//error handling (like getFallingRate >= 0) here
                        wipesPerSec = 0.5;
                    } else if(sensFus.getFallingRate() >= 5 && sensFus.getFallingRate() <= 10) {
//error handling (like getFallingRate >= 0) here
                        wipesPerSec = 1;
                    } else if(sensFus.getFallingRate() > 10) {
                        wipesPerSec = 2;
                    }
                }
            }

            if(sensFus.getLightStatus() >= 0 && sensFus.getLightStatus() < 400) { //getLightStatus
>= 0 error handling
                // 1.3.1.2.4 Automated Headlights
                turnOnHeadlights = true;
            }

            if(!sensFus.getDoorLockStatus() && sensFus.getVehicleSpeed() >= 15) {
                // 1.3.1.2.5 Automatic Door Locking
                lockDoors = true;
            }

            if(sensFus.getSelfDriving() && sensFus.getVehicleSpeed() > 0) {
                // 1.3.1.2.6 Speed Control
                if(sensFus.getSpeedLimit() > sensFus.getVehicleSpeed() && sensFus.getSpeedLimit()
>=0 && sensFus.getObjectDistance() >= 300) { //Added in the check for validity //getSpeedLimit >=
0 error handling
                    speedUp = true;
                }
                if(sensFus.getSpeedLimit() < sensFus.getVehicleSpeed() && sensFus.getSpeedLimit()
>=0) { //getSpeedLimit >= 0 error handling
                    slowDown = true;
                }
            }
            if((sensFus.getCruiseControl() || sensFus.getSelfDriving()) &&
sensFus.getVehicleSpeed() > 0 && sensFus.getObjectExists() && sensFus.getObjectSize() > 8 &&
sensFus.getObjectDistance() >= 0 && sensFus.getObjectDistance() <= 100) { //getVehicleSpeed,
getObjectDistance >= 0 error handling
                // 1.3.1.2.7 Potential Crash Detection and Automatic Braking
                gradualBrakingEmergency = true;
            }

            if(sensFus.getCrashDetected() && sensFus.getVehicleSpeed() == 0 &&
sensFus.getAirbagsDeploymentStatus()) {
                // 1.3.1.2.8 Crash Response
                dial911 = true;
            }
        }
    }

    public boolean getTurnOnCar() {
        return turnOnCar;
    }

    public boolean getTurnOnCarRemote() {
        return turnOnCarRemote;
    }

    public boolean getTurnOffCar() {
        return turnOffCar;
```

```
    }

    public boolean getGradualBrakingEmergency() {
        return gradualBrakingEmergency;
    }

    public boolean getGradualBrakingStop() {
        return gradualBrakingStop;
    }

    public boolean getApplyBrakes() {
        return applyBrakes;
    }

    public boolean getTurnOnWindshieldWipers() {
        return turnOnWindshieldWipers;
    }

    public double getWipesPerSec() {
        return wipesPerSec;
    }

    public boolean getTurnOnHeadlights() {
        return turnOnHeadlights;
    }

    public boolean getLockDoors() {
        return lockDoors;
    }

    public boolean getSlowDown() {
        return slowDown;
    }
    public boolean getSpeedUp() {
        return speedUp;
    }

    public boolean getTurnSteeringWheel() {
        return turnSteeringWheel;
    }

    public boolean getDial911() {
        return dial911;
    }
}

public class VehicleControl {

    // Class Attributes
    public Planning plan;

    // Default Constructor
    public VehicleControl(String filename) {
        plan = new Planning(filename);
        mainVC();
    }

    private void mainVC() {
        if(plan.getTurnOnCar()) {
```

```
    // 1.3.1.1.1 Driver Starts the Car
    if(turnOnCar()) {
        System.out.println("Vehicle Status: On");
    } else {
        System.out.println("ERROR: Vehicle unable to turn on.");
    }
} else if(plan.getTurnOnCarRemote()) {
    // 1.3.1.1.2 Remote Start
    if(turnOnCarRemote()) {
        System.out.println("Vehicle Status: On");
    } else {
        System.out.println("ERROR: Vehicle unable to turn on remotely.");
    }
} else if(plan.getTurnOffCar()) {
    // 1.3.1.1.3 Driver Turns Off the Car
    if(turnOffCar()) {
        System.out.println("Vehicle Status: Off");
    } else {
        System.out.println("ERROR: Vehicle unable to turn off.");
    }
}

if(plan.getGradualBrakingStop()) {
    // 1.3.1.2.2 Stop Signs, Traffic Lights, and Automatic Braking
    if (gradualBrakingStop()) {
        System.out.println("Brakes successfully applied! (stop)");
    } else {
        System.out.println("ERROR: Brakes not applied.");
    }
}

if(plan.getGradualBrakingEmergency()){
    // 1.3.1.2.7 Potential Crash Detection and Automatic Braking
    if(gradualBrakingEmergency()) {
        System.out.println("Brakes successfully applied! (emergency)");
    } else {
        System.out.println("ERROR: Brakes not applied.");
    }
}

if(plan.getTurnOnWindshieldWipers()) {
    // 1.3.1.2.3 Automated Front Windshield Wipers
    if(turnOnWindshieldWipers()) {
        System.out.println("Windshield wipers successfully turned on!");
    } else {
        System.out.println("ERROR: Windshield wipers not turned on.");
    }
}

if(plan.getTurnOnHeadlights()) {
    // 1.3.1.2.4 Automated Headlights
    if(turnOnHeadlights()) {
```

```
            System.out.println("Headlights successfully turned on!");
        } else {
            System.out.println("ERROR: Headlights not turned on.");
        }
    }

    if(plan.getLockDoors()) {
        // 1.3.1.2.5 Automatic Door Locking
        if(lockDoors()) {
            System.out.println("Doors successfully locked!");
        } else {
            System.out.println("ERROR: Doors not locked.");
        }
    }


    if(plan.getTurnSteeringWheel()) {
        // 1.3.1.2.1 Lane Assistance for Assisted Driving
        if(turnSteeringWheel()) {
            System.out.println("Steering wheel successfully turned!");
        } else {
            System.out.println("ERROR: Steering wheel not turned.");
        }
    }

    if(plan.getSpeedUp()) {
        // 1.3.1.2.6 Speed Control
        if(speedUp()) {
            System.out.println("Vehicle successfully sped up!");
            System.out.println("Velocity: " + plan.sensFus.getVehicleSpeed());
        } else {
            System.out.println("Vehicle speed increase unsuccessful.");
            System.out.println("Velocity: " + plan.sensFus.getVehicleSpeed());
        }
    }

    if(plan.getSlowDown()) {
        // 1.3.1.2.6 Speed Control
        if(slowDown()) {
            System.out.println("Vehicle successfully slowed down!");
            System.out.println("Velocity: " + plan.sensFus.getVehicleSpeed());
        } else {
            System.out.println("Vehicle speed decrease unsuccessful.");
            System.out.println("Velocity: " + plan.sensFus.getVehicleSpeed());
        }
    }

    if(plan.getDial911()) {
        // 1.3.1.2.8 Crash Response
        if(dial911()) {
            System.out.println("Vehicle successfully dialed 911! Help is on the way!");
        } else {
```

```
            System.out.println("Vehicle unsuccessful in dialing 911.");
        }
    }
}

/**
 * Function called by Planning when preconditions for
 * turning on car are met.
 * Returns true if car turns on successfully.
 * Returns false if car is not successfully turned on.
 * @return boolean
 */
private boolean turnOnCar() {
    // 1.3.1.1.1 Driver Starts the Car
    plan.sensFus.setVehicleStatus(true); // Vehicle is now on!
    return true;
}

/**
 * Function called by Planning when preconditions for
 * turning on car remotely are met.
 * Returns true if car turns on successfully.
 * Returns false if car is not successfully turned on remotely.
 * @return boolean
 */
private boolean turnOnCarRemote() {
    // 1.3.1.1.2 Remote Start
    plan.sensFus.setVehicleStatus(true); // Vehicle is now on!
    // Vehicle heater or A/C turns on depending on settings the driver last left
    // the heater or A/C on.
    // Display and all buttons light up.
    return true;
}

/**
 * Function called by Planning when preconditions for
 * turning off car are met.
 * Returns true if car turns off successfully.
 * Returns false if car is not successfully turned off.
 * @return boolean
 */
private boolean turnOffCar() {
    // 1.3.1.1.3 Driver Turns Off the Car
    plan.sensFus.setVehicleStatus(false); // Vehicle is now off!
    // Display shuts off
    return true;
}

/**
 * Function called by Planning when preconditions for
 * applying brakes are met.
 * Returns true if car applies brakes successfully.
```

```
     * Returns false if car does not successfully apply brakes.
     * @return boolean
     */
    private boolean applyBrakes() {
        plan.sensFus.setBrakePressed(true);     //turning brakes on
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * applying brakes gradually are met.
     * Returns true if car applies brakes gradually successfully.
     * Returns false if car does not successfully gradually apply brakes.
     * @return boolean
     */
    private boolean gradualBrakingEmergency() {  //calls applyBrakes, check if works by showing
speed
        // 1.3.1.2.7 Potential Crash Detection and Automatic Braking
        /*
         * Planning transmits a request to Vehicle Control to start braking
         * if the object is both within 100 feet and larger than 8 inches at an
         * intensity calculated using an algorithm to safely stop the vehicle at
         * least 3 feet from the object.
         */

        //object distance crash - within 100, stop less than 3 feet, objectsize > 8 inches

        while (plan.sensFus.getObjectDistance() >= 3 && plan.sensFus.getVehicleSpeed() > 0){
            plan.sensFus.setVehicleSpeed(plan.sensFus.getVehicleSpeed() - 10); //braking
intensity
        }

        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * applying brakes gradually are met. (Stop signs, traffic lights, automatic braking)
     * Returns true if car applies brakes gradually successfully.
     * Returns false if car does not successfully gradually apply brakes.
     * @return boolean
     */
    private boolean gradualBrakingStop() {  //calls applyBrakes, check if works by showing speed
        // To be implemented
        // 1.3.1.2.2 Stop Signs, Traffic Lights, and Automatic Braking

        //object distance stop - 250 ft away, will stop less than 5 feet

        while (plan.sensFus.getObjectDistance() >= 5 && plan.sensFus.getVehicleSpeed() > 0){
            plan.sensFus.setVehicleSpeed(plan.sensFus.getVehicleSpeed() - 3); //braking intensity
        }
        return true;
```

```
    }


    /**
     * Function called by Planning when preconditions for
     * turning on windshield wipers are met.
     * Returns true if car turns on windshield wipers successfully.
     * Returns false if car does not successfully turn on windshield wipers.
     * @return boolean
     */
    private boolean turnOnWindshieldWipers() {
        // 1.3.1.2.3 Automated Front Windshield Wipers
        plan.sensFus.setWipesPerSec(plan.getWipesPerSec()); // Add setWipesPerSec to SensorFusion
class
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * turning on headlights are met.
     * Returns true if car turns on headlights successfully.
     * Returns false if car does not successfully turn on headlights.
     * @return boolean
     */
    private boolean turnOnHeadlights() {
        // 1.3.1.2.4 Automated Headlights
        plan.sensFus.setHeadlightsStatus(true); // Add setHeadlightsStatus to SensorFusion class
        plan.sensFus.setDisplayMode(1); // Add setDisplayMode to SensorFusion class; 1 indicates
night mode, 0 indicates day mode
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * turning on headlights are met.
     * Returns true if car turns on headlights successfully.
     * Returns false if car does not successfully turn on headlights.
     * @return boolean
     */
    private boolean lockDoors() {
        // 1.3.1.2.5 Automatic Door Locking
        plan.sensFus.setDoorLockStatus(true); // Add setHeadlightsStatus to SensorFusion class
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * speeding up are met.
     * Returns true if car speeds up successfully.
     * Returns false if car does not successfully speed up.
     * @return boolean
     */
```

```
    private boolean speedUp() {
        // 1.3.1.2.6 Speed Control
        int currSpeed = plan.sensFus.getVehicleSpeed();
        while (currSpeed < plan.sensFus.getSpeedLimit()){
            //Increase vehicle speed by 1
            int speedToUpdate = plan.sensFus.getVehicleSpeed();
            speedToUpdate = speedToUpdate + 1;
            plan.sensFus.setVehicleSpeed(speedToUpdate);
            currSpeed = speedToUpdate;
        }
        plan.sensFus.setVehicleSpeed(currSpeed);       //setting it to current speed (should be
increased now)
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * slowing down are met.
     * Returns true if car slows down successfully.
     * Returns false if car does not successfully slows down.
     * @return boolean
     */
    private boolean slowDown() {
        // 1.3.1.2.6 Speed Control
        int currSpeed = plan.sensFus.getVehicleSpeed();
        while (currSpeed > plan.sensFus.getSpeedLimit()){
            //Decrease vehicle speed by 1
            int speedToUpdate = plan.sensFus.getVehicleSpeed();
            speedToUpdate = speedToUpdate - 1;
            plan.sensFus.setVehicleSpeed(speedToUpdate);
            currSpeed = speedToUpdate;
        }
        plan.sensFus.setVehicleSpeed(currSpeed);       //setting it to current speed (should be
decreased now)
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * turning steering wheel are met.
     * Returns true if car turns steering wheel successfully.
     * Returns false if car does not successfully turns steering wheel.
     * @return boolean
     */
    private boolean turnSteeringWheel() {
        // 1.3.1.2.1 Lane Assistance for Assisted Driving
        double dist = plan.sensFus.getDistanceFromLaneMarker();
        //~3 feet needed on each side of car to be centered
        //rn 6 inches
        //make it 3 feet!
        double distToMove = 36 - dist;
        plan.sensFus.setDistanceFromLaneMarker(dist + distToMove);  //now centered!
```

```
        return true;
    }

    /**
     * Function called by Planning when preconditions for
     * dialing 911 are met.
     * Returns true if car dials 911 successfully.
     * Returns false if car does not successfully dial 911.
     * @return boolean
     */
    private boolean dial911() {
        // 1.3.1.2.8 Crash Response
        plan.sensFus.set911Call(true); // Add set911Call to SensorFusion class
        return true;
    }
}
```

System Admin and Technician Components:

These components read from the log file, which contains the Technician's credentials, statuses on tasks from other components the Technician needs to fix and software updates. The log file will primarily contain two types of data, strings and boolean values. The boolean values will appear as "*0" for false and "*1" for true. Along with the string variables, the statuses of tasks and software updates will be documented as "complete" or "incomplete." The statuses of tasks and software updates will be sorted in reverse chronological order, with the tasks section specifying which component the Technician needs to do the task for. The input file will contain the username and password entered by the Technician (entry variable), and vehichleOn, isEngineOn.

Example of input file:
*1
*0
"John Doe"
"Sandwich1234"
Example of log file (in order of variables after input information excluding filename, String[] LogFile, and logFileScanner):
*1
*0
"John Doe"
"Sandwich1234"
"Software Updates: Status
Incomplete
Complete
Complete
…"
"Tasks: Status
Incomplete
Complete

Complete
…”

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.FileWriter;
public class SystemAdmin {
   String username;
   String password;
   String entryUser;
   String entryPassword;
   boolean verified;
   boolean isEngineOn; //might change vechicleOn to include the "in"
   String[] LogFile; //contains name of log file
   boolean tasksCompleted; //how to test if tasks are completed?
   boolean vehicleOn; //technically a requirement
   private Scanner logFileScanner;
   boolean isComputerConnected;

   public LogFile(String filename) { //if "incomplete" is found for a task status, mark
tasksComplete as false
       File dataFile = new File(filename);
       try {
           Scanner myReader = new Scanner(dataFile);
           while (myReader.hasNextLine()) {
               String line = myReader.nextLine();
               String[] logEntry = line.split(" ");
               // Assuming your format is "task complete/incomplete"
              if (logEntry.length >= 2 && logEntry[1].trim().equals("incomplete")) {
                   // Do something with the incomplete task
                  tasksCompleted = false;
               }
               else{
                   tasksCompleted = true;
               }
           }
           myReader.close();
       } catch (FileNotFoundException e) {
           System.out.println("Creating file: " + filename);
           File myObj = new File(filename);
           e.printStackTrace();
       }
   }

   public void displayLogin(){
       Scanner input = new Scanner(System.in);
       System.out.println("Enter Username: ");
       entryUser = input.next();
       System.out.println("Enter Password: ");
       entryPassword = input.next();
       verified = entryUser.equals(username) && entryPassword.equals(password);
       while(!(verified)){
           System.out.println("Incorrect Username or Password");
           System.out.println("Enter Username: ");
           entryUser = input.next();
           System.out.println("Enter Password: ");
           entryPassword = input.next();
       }
       if (verified == true){
           displayLogFile();
```

```
        }
    }
    public void displayLogFile() {
        try {
            logFileScanner = new Scanner(new File(LogFile));
            while (logFileScanner.hasNextLine()) {
                System.out.println(logFileScanner.nextLine());
            }
        } catch (FileNotFoundException e) {
            System.out.println("Log file not found.");
            e.printStackTrace();
        }
    }

    public void requestSignOff(){
        if(tasksCompleted){
            System.out.println("Signing out");
            if (logFileScanner != null){
                logFileScanner.close();
            }
        }
        else{
            System.out.println("Must complete tasks");
        }
    }
    public void updateLogFile(boolean verified){ //implement something to get date later
        FileWriter myWriter = new FileWriter(LogFile);
        if (verified == true){
            myWriter.write("Recent successful login.");
            myWriter.close();
        }
        else{
            myWriter.write("Recent failed login.");
            myWriter.close();
        }
    }


    if (isEngineOn == false && vehicleOn == true && isComputerConnected == true){ //allow
Technician to log in once vehicle is on and engine is off
        displayLogin();
    }
}
```

1.7 Testing

1.7.1 Validation Testing - Requirement Testing

1.7.1.1 Driver Starts the Car (Requirement 1.3.1.1.1)

Precondition: The vehicle is off, the engine is not running, the key is within 2 feet of the vehicle, the Driver is pressing the brake, and the Driver is sitting in the driver's seat.

Input/Trigger: Start/Stop button is pressed.

TEST 1: The input data file has valid values:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
1     //Key is 1 foot away
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

TEST 2: The input data has an invalid key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
-12   //Invalid key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is not turned on
Output:
Test passed!

TEST 3: The input data has the boundary key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
2     //Boundary key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

1.7.1.2 Remote Start (Requirement 1.3.1.1.2)

Precondition: The vehicle is off, the engine is not running, and the key is within 15 feet of the vehicle.

Input/Trigger: The Driver hits the button that is responsible for remote start on the key fob.

TEST 1: The input data file has the valid values:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Remote start button is pressed
* 0
* 0
0
0
10   //Key is 10 feet away
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

TEST 2: The input data file has an invalid key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Remote start button is pressed
* 0
* 0
0
0
-3   //Invalid key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is not turned on
Output:
Test passed!

TEST 3: The input data file has boundary key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Remote start button is pressed
* 0
* 0
0
0
15   //Boundary key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

1.7.1.3 Driver Turns off the Car (Requirement 1.3.1.1.3)

Precondition: The vehicle is on, the engine is running, the speed is 0, and the vehicle is in park.

Input/Trigger: Start/Stop button is pressed.

TEST 1: The input data file has the valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 1   //Vehicle is in park
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
-1
0
0    //Speed is 0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned off

Output: "Vehicle Status: Off"

Test passed!

TEST 2: The input data file has the value that the car is not in park:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0   //Vehicle is not in park
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
-1
0
0    //Speed is 0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is not turned off

Output:

Test passed!

TEST 3: The input data file has invalid speed:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 1   //Vehicle is in park
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
-1
0
-10  //Speed is invalid
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is not turned off

Output:

Test passed!

1.7.1.4 Lane Assistance for Assisted Driving (Requirement 1.3.1.2.1)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, speed is greater than 0 mph, and sensors have identified that the vehicle has veered less than 6 inches of a lane marker.

Input/Trigger: Cameras and sensors detect the vehicle less than 6 inches of a lane marker without activation of a turn signal.

TEST 1: The input data file has the valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
4     //4 inches from lane marker
-1
0
15   //Speed is 15 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is turned

Output: "Steering wheel successfully turned!"

Test passed!

TEST 2: The input data file has the distance from lane marker greater than 6:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
10   //10 inches from lane marker
-1
0
15   //Speed is 15 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is not turned

Output:

Test passed!

TEST 3: The input data file has the boundary distance from the lane marker that does not cause the steering wheel to be turned:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
6     //6 inches from lane marker
-1
0
15   //Speed is 15 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is not turned

Output:

Test passed!

1.7.1.5 Stop Signs, Traffic Lights, and Automatic Braking (Requirement 1.3.1.2.2)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, the vehicle's speed is greater than 0, and object sensors and cameras have identified a stop sign or red traffic light in the path of the vehicle at a distance of 1000 feet or fewer.

Input/Trigger: Object sensors and cameras have identified a stop sign or red traffic light in the path of the vehicle at a distance of 1000 feet or fewer.

TEST 1: The input data file has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 1   //Stop sign/red traffic light detected
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
100 //Stop sign/red traffic light is 100 feet away
-1
-1
-1
-1
-1
-1
```

Result: Brakes are applied
Output: "Brakes successfully applied! (stop)"
Test passed!

TEST 2: The input data file has a stop sign/traffic light distance value greater than 1000:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 1   //Stop sign/red traffic light detected
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
1200          //Stop sign/red traffic light is 1200 feet away
-1
-1
-1
-1
-1
-1
```

Result: Brakes are not applied
Output:
Test passed!

TEST 3: The input data file has the boundary stop sign/traffic light value that causes the brakes to be applied:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 1   //Stop sign/red traffic light detected
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
1000          //Stop sign/red traffic light is 1000 feet away
-1
-1
-1
-1
-1
-1
```

Result: Brakes are applied
Output: "Brakes successfully applied! (stop)"
Test passed!

1.7.1.6 Automated Front Windshield Wipers (Requirement 1.3.1.2.3)

Precondition: The vehicle is on, the engine is running, and sensors recognize it is raining/snowing/hailing (i.e. they detect a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield).

Input/Trigger: Sensors recognize it is raining/snowing/hailing (i.e. they detect a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield).

TEST 1: The input data file has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //It is hailing
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
0
0
-1
6     //6 hailstones on the windshield
-1
-1
-1
8     //8 hailstones per second
```

Result: Windshield wipers are turned on

Output: "Windshield wipers successfully turned on!"

Test passed!

TEST 2: The input data file has the boundary value of hailstones on the windshield so the wipers are not turned on:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //It is hailing
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
0
0
-1
4     //4 hailstones on the windshield
-1
-1
-1
8     //8 hailstones per second
```

Result: Windshield wipers are not turned on

Output:

Test passed!

TEST 3: The input data file has the invalid value of hailstones on the windshield so the wipers are not turned on:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //It is hailing
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
0
0
-1
-4    //-4 hailstones on the windshield
-1
-1
-1
8     //8 hailstones per second
```

Result: Windshield wipers are not turned on

Output:

Test passed!

1.7.1.7 Automated Headlights (Requirement 1.3.1.2.4)

Precondition: The vehicle is on, the engine is running, and there is less than 400 lumens per square meter surrounding the vehicle.

Input/Trigger: Headlight sensors recognize low light conditions meaning that there is less than 400 lumens per square meter surrounding the vehicle.

TEST 1: The input data file has valid values:

```
* 1          //Vehicle is on
* 1          //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
100         //100 lumens per
square meter
0
0
0
-1
-1
-1
-1
-1
-1
```

Result: Headlights are turned on

Output: "Headlights successfully turned on!"

Test passed!

TEST 2: The input data file has boundary lumens per square meter so headlights do not turn on:

```
* 1          //Vehicle is on
* 1          //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
400         //400 lumens per
square meter
0
0
0
-1
-1
-1
-1
-1
-1
```

Result: Headlights do not turn on

Output:

Test passed!

TEST 3: The input data file has invalid boundary lumens per square meter so headlights do not turn on:

```
* 1          //Vehicle is on
* 1          //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-100        //-100 lumens per
square meter
0
0
0
-1
-1
-1
-1
-1
-1
```

Result: Headlights do not turn on

Output:

Test passed!

1.7.1.8 Automatic Door Locking (Requirement 1.3.1.2.5)

Precondition: The vehicle is on, the engine is running, the doors are unlocked, and the vehicle speed is greater than or equal to 15 mph.

Input/Trigger: The vehicle reaches a speed greater than 15 mph while doors are unlocked.

TEST 1: The input data has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0   //Doors are unlocked
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
20   //Speed is 20 mph
0
-1
-1
-1
-1
-1
-1
```

Result: Doors are locked

Output: "Doors successfully locked!"

Test passed!

TEST 2: The input data has the boundary value of speed so doors get locked:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0   //Doors are unlocked
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
0
-1
-1
-1
-1
-1
-1
```

Result: Doors are locked

Output: "Doors successfully locked!"

Test passed!

TEST 3: The input data has too low of a value for speed so doors do not get locked:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0   //Doors are unlocked
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
5    //Speed is 5 mph
0
-1
-1
-1
-1
-1
-1
```

Result: Doors are not locked

Output:

Test passed!

1.7.1.9 Speed Control (Requirement 1.3.1.2.6)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, the vehicle's speed greater than 0 mph, the sensors have identified the posted speed limit (either via GPS or via camera sensors), and there is reasonable space (about 300 feet on the front of the car.

Input/Trigger: Sensors identify a posted speed limit, there is at least 300 feet in front of the vehicle, and the current vehicle's speed is less than the speed limit.

TEST 1: The input data has valid values to make the vehicle speed up:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
10   //Current speed is 10 mph
0
-1
-1
-1
400 //There is 400 feet in front of the car
15   //Speed limit is 15 mph
-1
```

Result: Vehicle speeds up until it reaches the speed limit
Output: "Vehicle successfully sped up! Velocity: 15"
Test passed!

TEST 2: The input data has valid values to make the vehicle slow down:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
20   //Current speed is 20 mph
0
-1
-1
-1
-1
15   //Speed limit is 15 mph
-1
```

Result: Vehicle slows down until it reaches the speed limit
Output: "Vehicle successfully slowed down! Velocity: 15"
Test passed!

TEST 3: The input data has speed limit equal to the vehicle's current speed:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Current speed is 15 mph
0
-1
-1
-1
-1
15   //Speed limit is 15 mph
-1
```

Result: The vehicle does not change speed
Output:
Test passed!

1.7.1.10 Potential Crash Detection and Automatic Braking (Requirement 1.3.1.2.7)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, vehicle speed is greater than 0, and there is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

Input/Trigger: There is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

TEST 1: The input data has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
0
-1
-1
10   //There is an object of 10 inches
50   //The object is 50 feet away
-1
-1
```

Result: Brakes are applied

Output: "Brakes successfully applied! (emergency)"

Test passed!

TEST 2: The input data has the boundary object distance so braking is still applied:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
15   //Speed is 15 mph
0
-1
-1
10   //There is an object of 10 inches
100 //The object is 100 feet away
-1
-1
```

Result: Brakes are applied

Output: "Brakes successfully applied! (emergency)"

Test passed!

TEST 3: The input data has invalid object distance so braking is not applied:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
-15  //Speed is -15 mph
0
-1
-1
10   //There is an object of 10 inches
100 //The object is 100 feet away
-1
-1
```

Result: Brakes are not applied

Output:

Test passed!

1.7.1.11 Crash Response (Requirement 1.3.1.2.8)

Precondition: The vehicle is on, the engine is running, the vehicle crashed/unexpectedly made sudden contact with an object (another vehicle, a pedestrian, a building, a traffic sign, etc. breached the surrounding 1 inch radius of the vehicle and has done so very suddenly–that is, within 3 seconds), the vehicle speed is 0, and the airbags have deployed.

Input/Trigger: The vehicle crashed/unexpectedly made sudden contact with an object (another vehicle, a pedestrian, a building, a traffic sign, etc. breached the surrounding 1 inch radius of the vehicle and has done so very suddenly–that is, within 3 seconds).

TEST 1: The input data has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Crash detected
* 1   //Airbags deployed
* 0
* 0
* 0
* 0
-1
-1
0
0     //Speed is 0
0
-1
-1
-1
-1
-1
-1
```

Result: Vehicle calls 911

Output: "Vehicle successfully dialed 911! Help is on the way!"

Test passed!

TEST 2: The input data has invalid speed value:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Crash detected
* 1   //Airbags deployed
* 0
* 0
* 0
* 0
-1
-1
0
-20  //Speed is invalid value
0
-1
-1
-1
-1
-1
-1
```

Result: 911 is not called

Output:

Test passed!

1.7.1.12 Technician Signing On (Requirement 1.3.1.3.1)

Precondition: The car is on and the engine isn't running. The Technician has successfully connected to the system admin.

Input/Trigger: Username, password, vehichleOn, isEngineOn, isComputerConnected

TEST 1:
The input data file has the valid values:
*1 //vehicle is on
*0 //engine is off
*1 //computer is connected
"John Doe" //compared with information stored in log file
"Sandwich1234" //compared with information stored in log file
Result: displayLogFile() is called
Output: Log file
Test passed!

TEST 2:
*1 // vehicle is on
*1 //engine is on
*1 //computer is connected

Result:
Log in page will not be displayed
Test passed!

TEST 3:
The input data file has the valid values:
*1 //vehicle is on
*0 //engine is off
"Jane Doe" //compared with information in log file
"Sandwich2468" //compared with information in log file
Result: displayLogin() continues to be called
Output: Incorrect Username or Password
Test passed!

1.7.1.13 Technician Signing Off (Requirement 1.3.1.3.2)

Precondition: The car is on and the engine isn't running. The Technician has successfully connected to the system admin. The username and password have successfully been verified and there are no tasks to complete.

Input/Trigger: Username, password, vehichleOn, isEngineOn, requestSignOff()

TEST 1:
The input data file has the valid values:
*1 //vehicle is on
*0 //vehicle is off

*1 //computer is connected
“John Doe” //compared with information stored in log file
“Sandwich1234” //compared with information stored in log file
requestSignOff() //check status of tasks/software updates in log file

The log file contains the following:

*1 //vehicle on
*0 //engine off
*1 //computer is connected
*1 // verified
*1 // tasks completed

“John Doe”
“Sandwich1234”
“Tasks: Status
Complete
Complete
Complete
…”

Data sent from other components
Result: Technician is signed out
Output: “Signing out”
Test passed!

TEST 2:
The input data file has the valid values:
*1 //vehicle on
*0 //engine off
*1 //computer is connected
“John Doe”
“Sandwich1234”

The log file contains the following:
*1 //vehicle on
*0 //engine off
*1 //computer is connected
*1 // verified
*1 // tasksCompleted
“John Doe”
“Sandwich1234”

“Tasks: Status
Complete

Complete
Incomplete
…”
Data sent from other components
Result: The Technician is prevented from signing out.
Output: “Must complete tasks”
Test passed!

1.7.2 Scenario-based Testing - Use Case Testing

1.7.2.1 Driver Starts the Car (Use Case 1.4.1.1)

Precondition: The vehicle is off, the engine is not running, the key is within 2 feet of the vehicle, the Driver is pressing the brake, and the Driver is sitting in the driver's seat.

Input/Trigger: Start/Stop button is pressed.

TEST 1: The input data file has the valid values:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
0     //Key is 0 feet away
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

TEST 2: The input data has an invalid key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
-24   //Invalid key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is not turned on
Output:
Test passed!

TEST 3: The input data has the boundary key distance:

```
* 0   //Vehicle is off
* 0   //Engine is off
* 1   //Brake is being pressed
* 1   //Driver is in driver's seat
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //Start/Stop button is pressed
* 0
* 0
* 0
0
0
2     //Boundary key distance
0
-1
-1
-1
-1
-1
-1
-1
```

Result: The engine is turned on and the Display lights up
Output: "Vehicle Status: On"
Test passed!

1.7.2.2 Potential Crash Detection (Use Case 1.4.1.2)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, vehicle speed is greater than 0, and there is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

Input/Trigger: There is an object (car, sign, pedestrian, or any slow/non-moving object) larger than 8 inches detected in the path of the front of the vehicle at a distance of 100 feet or less.

TEST 1: The input data has valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
25   //Speed is 25 mph
0
-1
-1
12   //There is an object of 12 inches
50   //The object is 50 feet away
-1
-1
```

Result: Brakes are applied

Output: "Brakes successfully applied!"

Test passed!

TEST 2: The input data has the boundary object distance so braking is still applied:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
25   //Speed is 25 mph
0
-1
-1
12   //There is an object of 12 inches
100 //The object is 100 feet away
-1
-1
```

Result: Brakes are applied

Output: "Brakes successfully applied!"

Test passed!

TEST 3: The input data has invalid object distance so braking is not applied:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 1   //Self-driving is on
* 1   //Object is detected
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0
-1
-1
0
-25  //Speed is -25 mph
0
-1
-1
10   //There is an object of 10 inches
100 //The object is 100 feet away
-1
-1
```

Result: Brakes are not applied

Output:

Test passed!

1.7.2.3 Adverse Driving Conditions (Use Case 1.4.1.3)

Precondition: The vehicle is on, the engine is running, and the sensors recognize it is raining/snowing/hailing (they detect a falling rate of at least 1 raindrop/snowflake/hailstone per second, or more than 4 hailstones or 35 raindrops/snowflakes on the windshield) or the headlight sensors recognize low light conditions (less than 400 lumens per square meter surrounding the vehicle).

Input/Trigger: Sensors recognize that it is raining, snowing, or hailing meeting the conditions listed above or headlight sensors recognize low light conditions.

TEST 1: The input data has valid values:

```
* 1   //Vehicle  is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //It is hailing
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
200 //200 lumens per square meter
0
0
0
-1
6     //6 hailstones on the windshield
-1
-1
-1
8     //8 hailstones per second
```

Result: Windshield wipers and headlights are turned on

Output: “Windshield wipers successfully turned on!

Headlights successfully turned on!”

Test passed!

TEST 2: The input data has invalid values so neither the headlights or windshield wipers turn on:

```
* 1   //Vehicle  is on
* 1   //Engine is on
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 1   //It is hailing
* 0
* 0
* 0
* 0
* 0
* 0
* 0
* 0
-1
500 //200 lumens per square meter
0
0
0
-1
2     //2 hailstones on the windshield
-1
-1
-1
8     //8 hailstones per second
```

Result: Windshield wipers and headlights are not turned on

Output:

Test passed!

1.7.2.4 Lane Assistance for Assisted Driving (Use Case 1.4.1.4)

Precondition: The vehicle is on, the engine is running, cruise control and/or self-driving is activated, speed is greater than 0 mph, and sensors have identified that the vehicle has veered less than 6 inches of a lane marker.

Input/Trigger: Cameras and sensors detect the vehicle less than 6 inches of a lane marker without activation of a turn signal.

TEST 1: The input data file has the valid values:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
2     //2 inches from lane marker
-1
0
30   //Speed is 30 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is turned

Output: "Steering wheel successfully turned!"

Test passed!

TEST 2: The input data file has the distance from lane marker greater than 6:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
7     //7 inches from lane marker
-1
0
15   //Speed is 15 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is not turned

Output:

Test passed!

TEST 3:

The input data file has the boundary distance from the lane marker that does not cause the steering wheel to be turned:

```
* 1   //Vehicle is on
* 1   //Engine is on
* 0
* 0
* 0
* 1   //Cruise control is on
* 0
* 0
* 0
* 0
* 0
* 1
* 0
* 0
* 0
* 0
* 0
* 0   //Turn signal is off
* 0
6     //6 inches from lane marker
-1
0
30   //Speed is 30 mph
-1
-1
-1
-1
-1
-1
-1
```

Result: The steering wheel is not turned

Output:

Test passed!

1.7.2.5 Technician Sign On and Sign Off (Use Case 1.4.1.5)

Precondition: The car is on and the engine isn't running. The Technician has successfully connected to the system admin. The username and password have successfully been verified and there are no tasks to complete.

Input/Trigger: Username, password, vehichleOn, isEngineOn, requestSignOff()

TEST 1:
The input data file has the valid values:
*1 //Vehicle is on
*0 //Engine is off
*1 //computer is connected
"John Doe"
"Sandwich1234"
requestSignOff()

The log file has the following values:
*1 //vehicle on
*0 //engine off
*1 //computer is connected
*1 //verified
*1 // tasks completed
"John Doe"
"Sandwich1234"
"Software Updates: Status
Complete
Complete
Complete
…"

"Tasks: Status
Driver Complete
Planning Complete
Display Complete
…"
Result: Technician is signed out
Output: **"**Signing out"
Test passed!

TEST 2:
The input data file has the valid values:
*1 //the vehicle is on
*0 //the engine is off
"Jon Doe"

“Sandwich1234”

The log file has the following values:
*1 //vehicle on
*0 //engine off
*1 //computer is connected
*0 //verified false
*1 // tasks completed
“John Doe” //correct username and password stored in log file
“Sandwich1234”
“Software Updates: Status
Complete
Complete
Complete
…”

“Tasks: Status
Driver Complete
Planning Complete
Display Complete
…”
Result: Login page continues to be displayed
Output: Incorrect Username or Password
Test passed!

TEST 3:
The input data file has the valid values:
*0 //the vehicle is off
*0 //the engine is off
*0 //computer is disconnected

Result: The Technician is unable to connect their laptop to the system.
Output: None
Test passed!

TEST 4:
The input data file has the valid values:
*1 //the vehicle is off
*0 //the engine is off
*1 //computer is connected
“John Doe”
“Sandwich1234”
requestSignOff()
The log file has the following values:

*1 //vehicle on
*0 //engine off
*1 //computer is connected
*1 //verified
*0 // tasks completed is false
“John Doe” //correct username and password stored in log file
“Sandwich1234”
“Software Updates: Status
Incomplete
Complete
Complete
…”

“Tasks: Status
Complete
Complete
Incomplete
…”
“Tasks: Status
Driver Complete
Planning Complete
Display Complete
…”

Result: Technician is unable to log off
Output: “Must complete tasks”
Test passed!