ALSET Software Development

Phase 1

Team Name: D.E.Nu.M
Members: Elizabeth Bruski, Nusiba Zaman, Melissa Ozcan, Diego Ramirez

Section 1: Introduction

Implementing an iterative process, we are planning to improve and develop our model through each cycle of the process. Prioritizing progress is important to us and the iterative process allows us to begin our project and accurately know what to adjust by the end of each round. Each cycle of our process will begin with a group meeting, every week, where we discuss the plan moving forward and the modeling changes that need to be made. On our own we can construct different parts of the software and meet again as a group to deploy the project and communicate how we must tweak and improve our design. This team is prepared to take on this project, as we all have significant experience in the software development processes from our previous core computer science classes such as Data Structures and Algorithms. This will be a mission critical real-time system which is super important since failure of the software to react in time would likely cause damage to the vehicle, its surroundings, and people in it. The success of this whole project relies on our access to IoT and network so we can consistently download and apply updates.

I. Requirements:

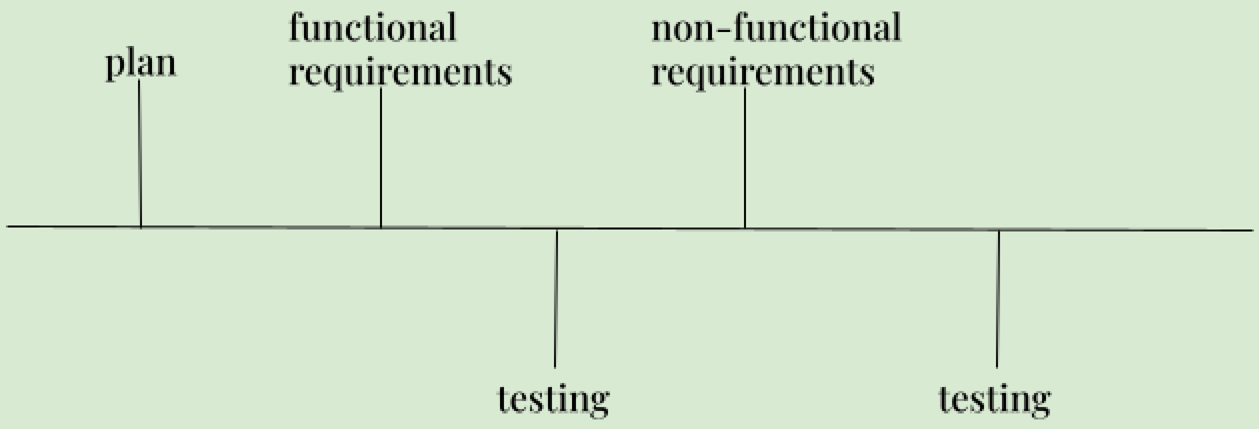
A. Functional Requirements

1. Functional requirements for this D.E.Nu.M software will ensure our vehicles are street-legal and high performing since safety is our utmost priority. We need all vehicles to recognize and follow the lanes in the road, while also navigating roads without lanes, like those in neighborhoods and gravel lanes. To keep these vehicles legal the software must recognize whether there is a person in the driver's seat, and stop accordingly. Once the vehicle begins driving, we need it to sense the color of streetlights, speed limits, and speed of surrounding cars to accelerate and decelerate smoothly and safely. As we continue development we will continue realizing and prioritizing features we need to implement. Since safety is the most important requirement category of any vehicle, we will test all functional driving features before implementing and testing the non-functional ones.
2. The vehicle will have a virtual key, accessed through the corresponding application. This will allow the car to start remotely, without leaving it unlocked, in danger of getting

hijacked. For those who would prefer a physical key we will also offer a key card and a sleek flat bead that can be attached to a watch or bracelet for a practical, easily held key.

B. *Non-Functional Requirements*

1. DENM cars provide more than just intelligent and responsive technology. Bluetooth connection allows users to never be disconnected from their personal lives. Through voice commands users can pick up phone calls, respond to texts, and listen to emails by connection their phone to the car.
2. Our cars also connect to user's personal music streaming platforms to play their favorite music, right where they last left it off on their own devices. We allow easy scrolls through their playlists and favorite albums for easy music changes on the road. Users can also set up a default song to play for new passengers in their car if a song isn't already playing.
3. Our cars are family friendly and comfort is prioritized for our users with options to adjust seats and temperature depending on which user enters the car.
4. DENM also provides a mobile app that allows users to start their cars from their phones. It allows the car to warm up in the winter before the user enters the car to save their time.
5. We will provide eye tracking technology to keep drivers involved and responsible on the road. This allows a safer and more focused driving experience.

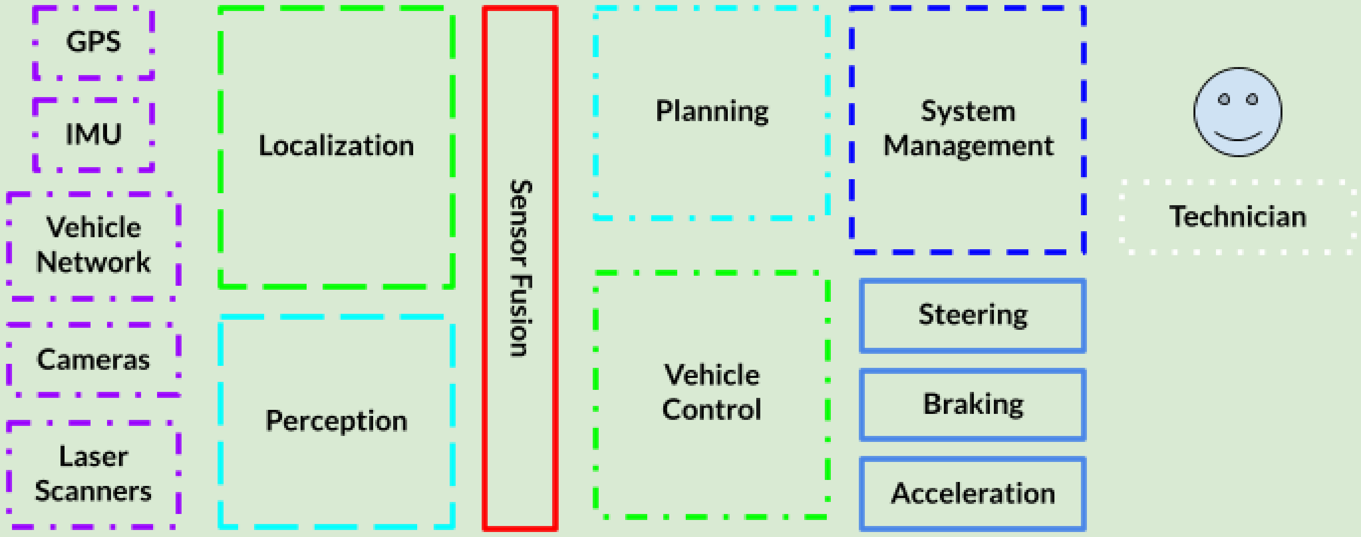


Section 2: Functional Architecture

Functional Architecture explains the skeleton of the car and the driving experience. Our vehicle will be in stage 0-3 of automated operation where we will require a driver to be present in the driver's seat of the vehicle and they will assume legal responsibility for its movement. As we continue developing the software we will roll out updates, which can be downloaded via wifi, requiring the car to have wifi connectivity. These updates will be released on the Alset website and can be accessed through the car's personal secure login. The car itself will also have a secure login system, where Alset-certified technicians can see performance statistics and manually modify certain settings should the car need personalized updates. They will have access to the system administration settings. In order to reduce our cost of change as we continue developing the software we will keep in mind what the consumers would like, so we don't need to reinvent our software, but rather can build on iterations and improve it. Both of these require both wifi network and cloud connectivity to ensure they are up to date.

I. Sensor Data and Environment Perception:

It will include a sensor fusion algorithm, which correlates sensor inputs from cameras, radar, and lidar. This algorithm will be trained on our dataset of roads, streets and objects. The sensors are divided between those that directly work with the vehicle and those that deal with perception.



II. Driver Responsibility

Drivers also take on responsibility in Alset cars. They can control the speeds of the car depending on the speed limit of the area they are driving in. This allows the driver to be actively conscious behind the wheel and active about the travel of the car.

Mentioned as one of our functional requirements, eye tracking will hold the driver responsible to be focused on the road. Our eye tracking technology will be built into the base of the rear view mirror and consistently monitor the gaze of the driver. If their eyes leer off the road for longer than 5 seconds they will be alerted via the speaker system in the car to return their sight back on the road.

Inputs	Outputs
- Cameras - Sensors - localization - vertical & horizontal laser scanner - LiDAR - Vehicle Perception - System Management: google maps	- Automatic Braking - Steering Assistance - Directional Choice (from mapping service) - changing based on traffic and other data - Logging Information - Real Time Imaging - turns on sensors when necessary

III. Features

Our cars include a wide range of features, working together to enhance the driving experience. With advanced GPS systems connected to in-car wifi, routes are constantly analyzed to offer the most efficient path to your destination.

Our vehicle network can connect to multiple users via our DENuM app to account for multiple drivers of your vehicle. With preset settings, users can adjust the car seat, mirrors, and temperature from the app and have the best driving environment for them ready through a click of a button.

Section 3: Requirements

3.1 Functional Requirements

Our functional requirements work together to create a safe and smooth self-driving car experience.

3.1.1 Cruise Control

Pre-Conditions:

- The vehicle is in the drive gear
- The vehicle is moving at or above 1 MPH

Post-Conditions:

- The vehicle continues to move at the speed that it was set to, until it is instructed to stop by input from the vehicle's brake pedal. Speed can also be adjusted by buttons on the driver's console.

This system will allow the driver to set a desired speed, and maintain that speed until the driver deactivates the system or applies the brake pedal.

3.1.2 Lane Keeping Assistance

Pre-Conditions:

- The vehicle is in the drive gear
- The vehicle must have the proper camera sensors to detect when the car is veering outside of the lanes
- The Lane Keep Assist system is activated
- The vehicle is traveling at a speed of at least 35 MPH
- The lane markings are clearly visible

Post-Conditions:

- The vehicle stays in the same lane, unless the driver indicates otherwise by using the turn signal and changing lanes.
- The vehicle maintains a safe distance from other vehicles on the road, as determined by the radar sensor

This system will monitor the vehicle's position within the lane, and provide the driver with visual and/or audible warnings if the vehicle deviates from the lane. If the driver does not correct the deviation, it will automatically apply corrective steering to keep the vehicle within the lane.

3.1.3 Forward Collision Warning

Pre-Conditions:

- The vehicle is in the drive gear
- The FCW system is activated
- The vehicle is moving at a speed of at least 20 MPH

- The camera and radar sensors are installed and functioning properly

Post-Conditions:

- The FCW system may activate the vehicle's brakes or other safety systems to avoid a collision.
- If the driver responds to the warning and takes action to avoid a collision, the FCW system will disengage.
- If a collision cannot be avoided, the FCW system will activate the vehicle's safety systems to minimize the impact and protect the vehicle's occupants.

This system will monitor the distance between the vehicle and objects in front of it, and provide the driver with visual and/or audible warnings if the distance becomes too short to avoid a collision.

3.1.4 Automatic Emergency Braking

If the driver does not respond to the forward collision warning, this system will automatically apply the brakes to avoid or mitigate a collision.

Pre-Conditions:

- Car is accelerating or actively moving
- There is an object in the way of the direction the car is moving in
- The car is using cruise control to regulate following distance to keep in the flow of traffic

Post-Conditions:

The car will auto-brake according to the distance it is away from the reason for braking. Measured in time it would take the car to encounter the object, it will either brake extremely aggressively or lightly.

- If the car is following too closely to the car in front of it, with three following distance options in car-length increments, it will automatically lightly brake

3.1.5 Blind Spot Detection

This system will monitor the blind spots on the sides and rear of the vehicle, and provide the driver with visual and/or audible warnings if there is a vehicle in the blind spot.

Pre-Conditions:

- The vehicle is in the drive gear.
- The BSD system is activated.
- The vehicle is moving at a speed of at least 20 MPH.
- The radar sensors and/or camera sensors are functioning properly.

Post-Conditions:

- The warning will typically appear on the side mirrors or on the dashboard display.

- If the driver responds to the warning and takes action to avoid a collision, the BSD system will disengage.
- If the driver does not respond to the warning and attempts to change lanes or turn, the BSD system may activate the vehicle's safety systems to avoid a collision.
- The BSD system will disengage automatically when the vehicle is no longer in motion, or when the driver turns off the BSD system manually.

3.1.6 Rear Cross Traffic Alert

This system will monitor the area behind the vehicle when the vehicle is in reverse, and provide the driver with visual and/or audible warnings if there is a vehicle or pedestrian approaching from the side.
Pre-Conditions:

- The vehicle is in reverse gear.
- The RCTA system is activated.
- The radar sensors and/or camera sensors are functioning properly.

Post-Conditions:

- The warning will typically appear on the dashboard display or in the rearview mirror.
- If the driver responds to the warning and takes action to avoid a collision, the RCTA system will disengage.
- If the driver does not respond to the warning and continues to reverse, the RCTA system may activate the vehicle's safety systems to avoid a collision.
- The RCTA system will disengage automatically when the vehicle is shifted out of reverse gear, or when the driver turns off the RCTA system manually.

3.1.7 Adaptive Headlights

This system will adjust the direction and intensity of the headlights based on the detected amount of light from daylight
Pre-Conditions:

- The vehicle is in the drive gear.
- The adaptive headlights system is activated.
- The vehicle is traveling at a speed of at least 25 MPH.
- The sensors, such as cameras or light sensors, are working properly.

Post-Conditions:

- If the vehicle is approaching a bend or intersection, the adaptive headlights system may adjust the headlights to provide better visibility of the area.

- If the vehicle is traveling on a highway, the adaptive headlights system may adjust the headlights to provide better illumination of the road ahead and prevent blinding other drivers.
- If the system detects oncoming traffic or vehicles in front of the vehicle, it may automatically adjust the headlights to avoid blinding other drivers and to ensure maximum visibility.
- The adaptive headlights system will disengage automatically when the vehicle is turned off or the driver manually turns off the system.

3.1.8 Traffic Sign Recognition

This system will recognize and interpret traffic signs, and provide the driver with visual and/or audible warnings if the driver is not following the traffic signs.

Pre-Conditions:

- Car is accelerating and actively moving.
- Car is approaching an intersection

Post-Conditions:

- Action based off of light
    a) If red, slow and stop the vehicle.
    b) If yellow, slow the vehicle.
    c) If green, continue at the current acceleration through the intersection.

3.1.9 Driver Monitoring

This system will monitor the driver's behavior, such as eye gaze, head position, and drowsiness, and provide the driver with visual and/or audible warnings if the driver shows signs of distraction or fatigue.

Pre-Conditions:

- Driver is not touching the wheel for more than three seconds, noticed through pressure sensors in the wheel

Post-Conditions:

- The car issues an alert, beeping at the driver to remind them to engage with the road again

3.1.10 Diagnostic and Software Update

The technician interface will allow a qualified technician to log in with a user ID and password, retrieve the log file, perform diagnostics, and update the software modules as necessary.

Pre-Conditions:

- Car is connected to wifi and can remain connected to wifi
- Car has pending updates

Post-Conditions:

- Car asks driver for permission to update

- Car downloads software updates from the server

Section 4: Requirement Modeling

4.1: Use Case Scenarios:

4.1.1 Driver activates Lane Keep Assist (LKA)

Pre-Conditions: The vehicle is in the drive gear, the vehicle is moving at above 20 MPH but is not over 90 MPH
Post-Conditions: The vehicle continues to move at the speed that it was set to, until it is instructed to stop by input from the vehicle's brake pedal.

Trigger: The Driver activates the LKA system through the vehicle's onboard computer or by pressing a button on the dashboard.

1. The camera sensors detect the vehicle's position within the lane
2. The Lane Keep Assist system continuously monitors the vehicle's position within the lane
3. If the vehicle deviates from the lane, the Lane Keep Assist system provides the driver with visual and/or audible warnings
4. If the driver does not correct the deviation, the Lane Keep Assist system applies corrective steering to keep the vehicle within the lane
5. If the driver uses the turn signal to indicate a lane change, the Lane Keep Assist system disengages until the vehicle is back within the lane
6. The radar sensor detects other vehicles on the road and maintains a safe distance from them

4.1.2 Object Detection and Collision Prevention

Pre-condition: The Car is on and sensors are on, the Car is moving at a speed greater than 5mph.
Post-condition: The system detects objects and prevents collisions.

Trigger: The Sensor Fusion system receives data from the sensors.

1. The Sensor Fusion system processes the data.
2. The Sensor Fusion system sends the processed data to the Planning system.
3. The Planning system decides if there is a risk of collision.
4. If there is a risk of collision, the Planning system sends a signal to the Vehicle Control System (VCS) to slow down or stop the vehicle.

5. The Driver is notified of the risk of collision.

Exception: If the Planning system cannot determine a safe path to avoid collision, the system sends a signal to the Driver to take control of the vehicle

4.1.3 Software Update

Pre-condition: The Car is connected to a network or a technician device.
Post-condition: The software update is successfully installed.

Trigger: The Technician initiates the software update.

1. The Technician connects the device to the Car.
2. The Technician initiates the software update process.
3. The Planning system receives the update request.
4. The Planning system verifies that the update is valid and compatible with the Car's system.
5. If the update is valid and compatible, the Planning system sends the update to the Sensor Fusion system.
6. The Sensor Fusion system receives the update and installs it on the Car's system.
7. The Sensor Fusion system verifies that the update is successfully installed.
8. The Planning system sends a notification to the Technician that the update is complete.

Exception: If the update is invalid or incompatible, the Planning system sends a notification to the Technician that the update cannot be installed.

4.1.4 System Calibration

Pre-condition: The Car is turned on and sensors are on.
Post-condition: The sensors are calibrated and ready to operate.

Trigger: The Technician initiates the calibration process.

1. The Technician connects the calibration device to the Car.
2. The Calibration system receives the calibration request.
3. The Calibration system retrieves the current calibration data from the Sensor Fusion system.
4. The Calibration system performs the calibration process on the sensors.

5. The Calibration system sends the updated calibration data to the Sensor Fusion system.
6. The Sensor Fusion system receives the updated calibration data and verifies that the sensors are calibrated.
7. The Sensor Fusion system sends a notification to the Technician that the calibration is complete.

Exception: If the calibration fails, the Sensor Fusion system sends an error message to the Calibration system, which sends a notification to the Technician that the calibration failed.

4.1.5 System Shutdown

Pre-condition: The Car is turned on.
Post-condition: The Car is turned off and all systems are shut down.

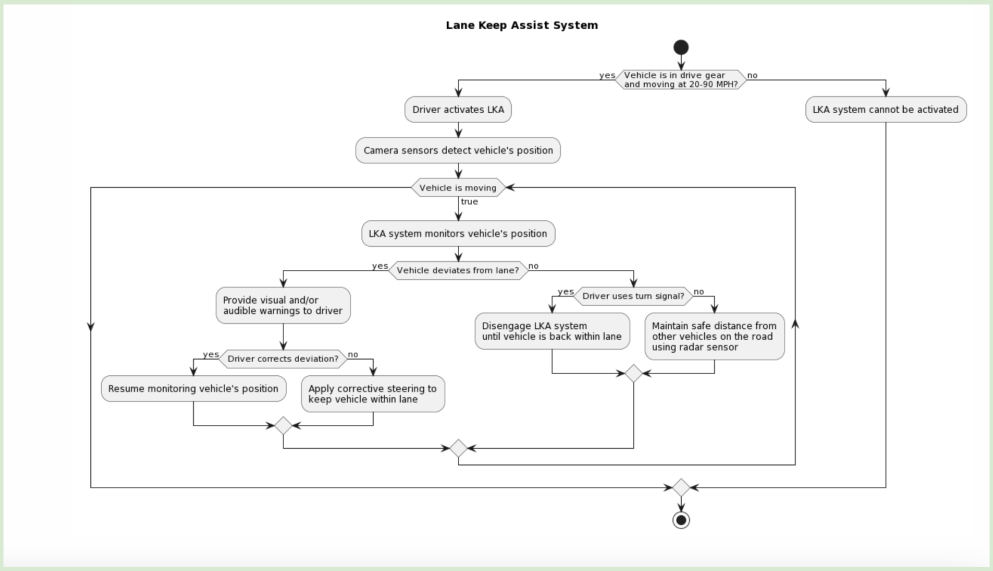
Trigger: The Driver initiates the shutdown process.

1. The Driver presses the shutdown button.
2. The Planning system receives the shutdown request.
3. The Planning system sends a signal to the VCS to turn off the engine.
4. The Sensor Fusion system receives the shutdown signal and sends a signal to all sensors to turn off.
5. The Planning system sends a signal to the Display system to turn off the display.
6. The Display system turns off the display.
7. The Sensor Fusion system sends a notification to the System Admin that the Car has been shut down.

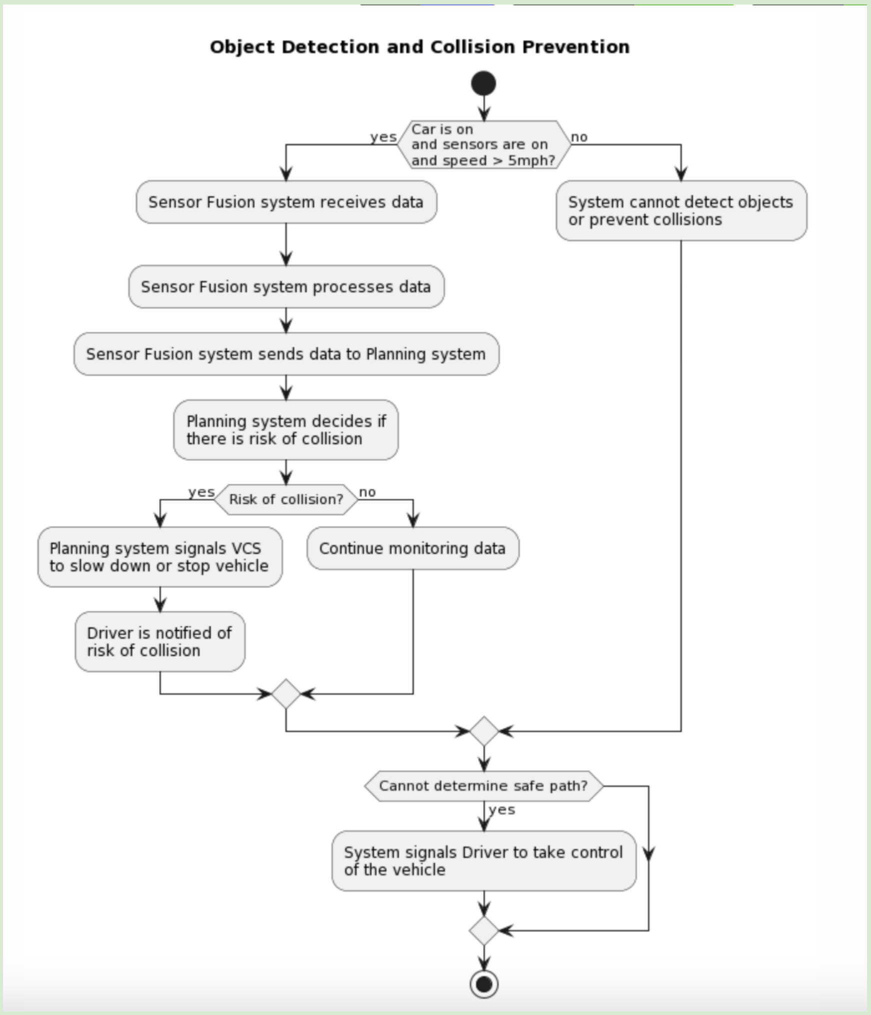
Exception: None.

4.2: Activity Diagrams:

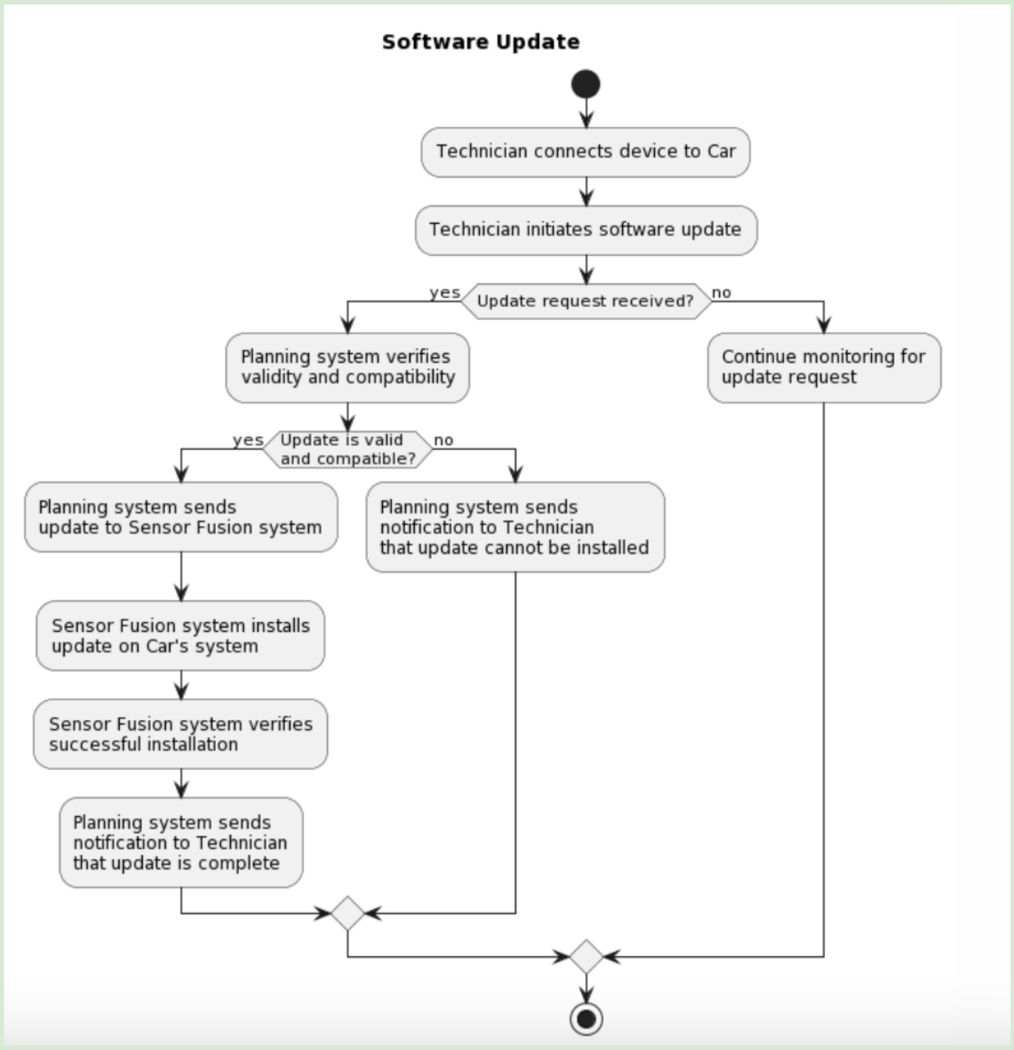
4.2.1 Lane Keep Assist


Lane Keep Assist System
yes
Vehicle is in drive gear and moving at 20-90 MPH?
no
Driver activates LKA
LKA system cannot be activated
Camera sensors detect vehicle's position
Vehicle is moving
true
LKA system monitors vehicle's position
yes
Vehicle deviates from lane?
no
Provide visual and/or audible warnings to driver
yes
Driver uses turn signal?
no
Disengage LKA system until vehicle is back within lane
Maintain safe distance from other vehicles on the road using radar sensor
yes
Driver corrects deviation?
no
Resume monitoring vehicle's position
Apply corrective steering to keep vehicle within lane

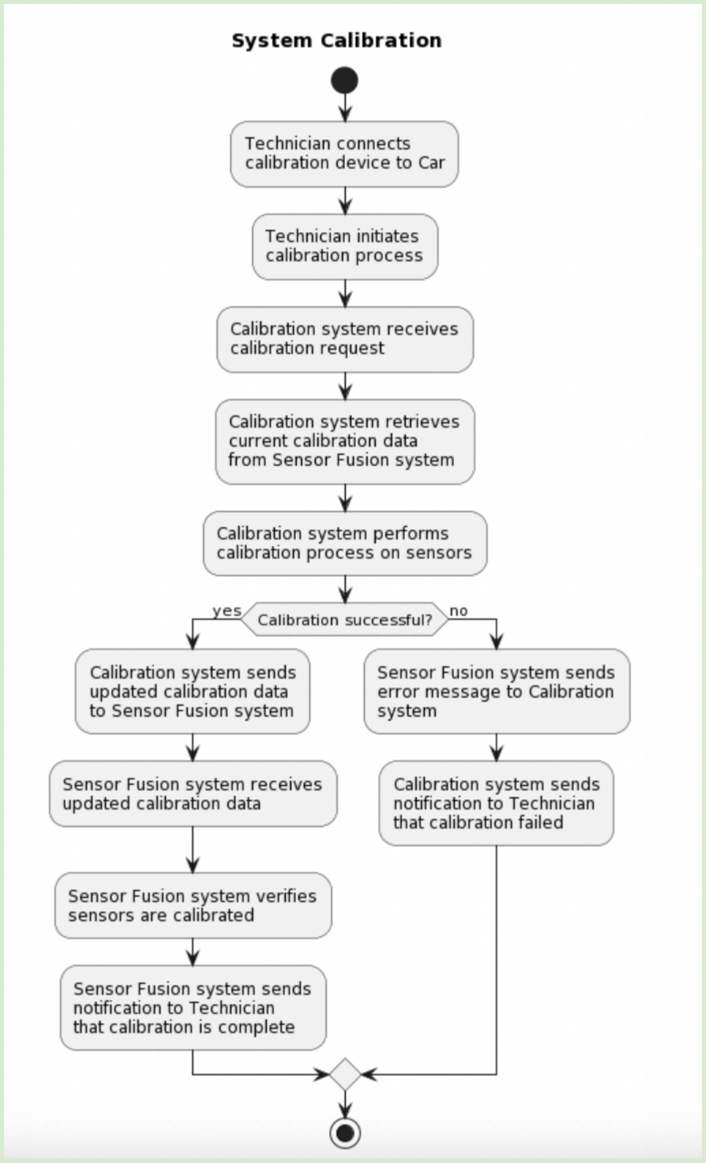
4.2.2 Object Detection and Collision Prevention


Object Detection and Collision Prevention
Car is on and sensors are on and speed > 5mph?
yes
no
Sensor Fusion system receives data
System cannot detect objects or prevent collisions
Sensor Fusion system processes data
Sensor Fusion system sends data to Planning system
Planning system decides if there is risk of collision
Risk of collision?
yes
no
Planning system signals VCS to slow down or stop vehicle
Continue monitoring data
Driver is notified of risk of collision
Cannot determine safe path?
yes
System signals Driver to take control of the vehicle

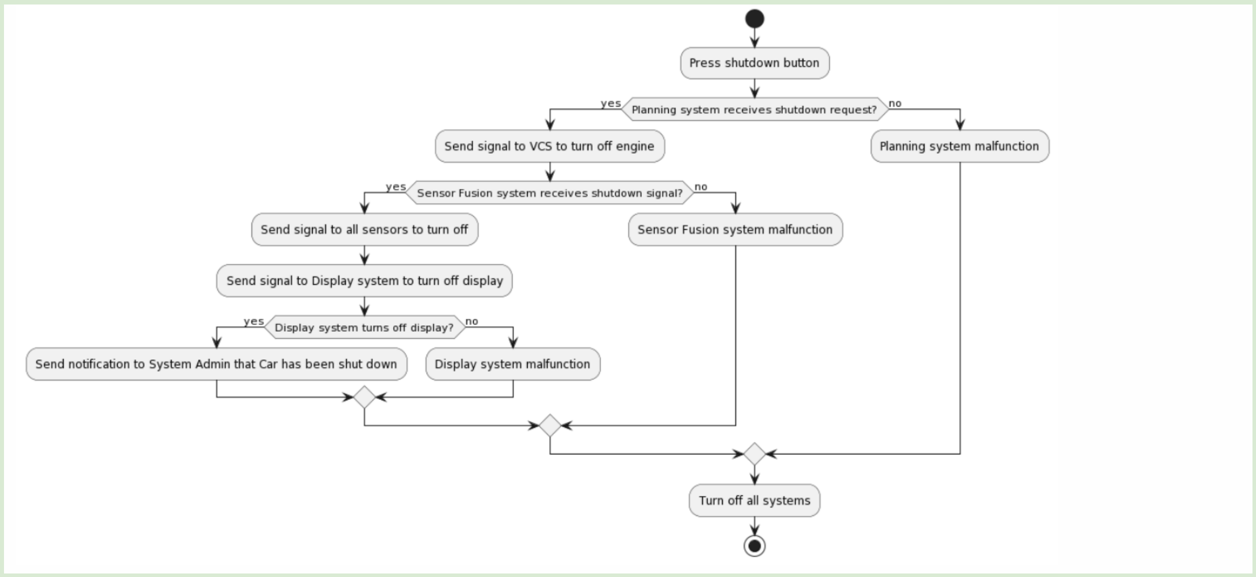
4.2.3 Software Update


Software Update
Technician connects device to Car
Technician initiates software update
Update request received?
yes
no
Planning system verifies validity and compatibility
Continue monitoring for update request
Update is valid and compatible?
yes
no
Planning system sends update to Sensor Fusion system
Planning system sends notification to Technician that update cannot be installed
Sensor Fusion system installs update on Car's system
Sensor Fusion system verifies successful installation
Planning system sends notification to Technician that update is complete

4.2.4 System Calibration

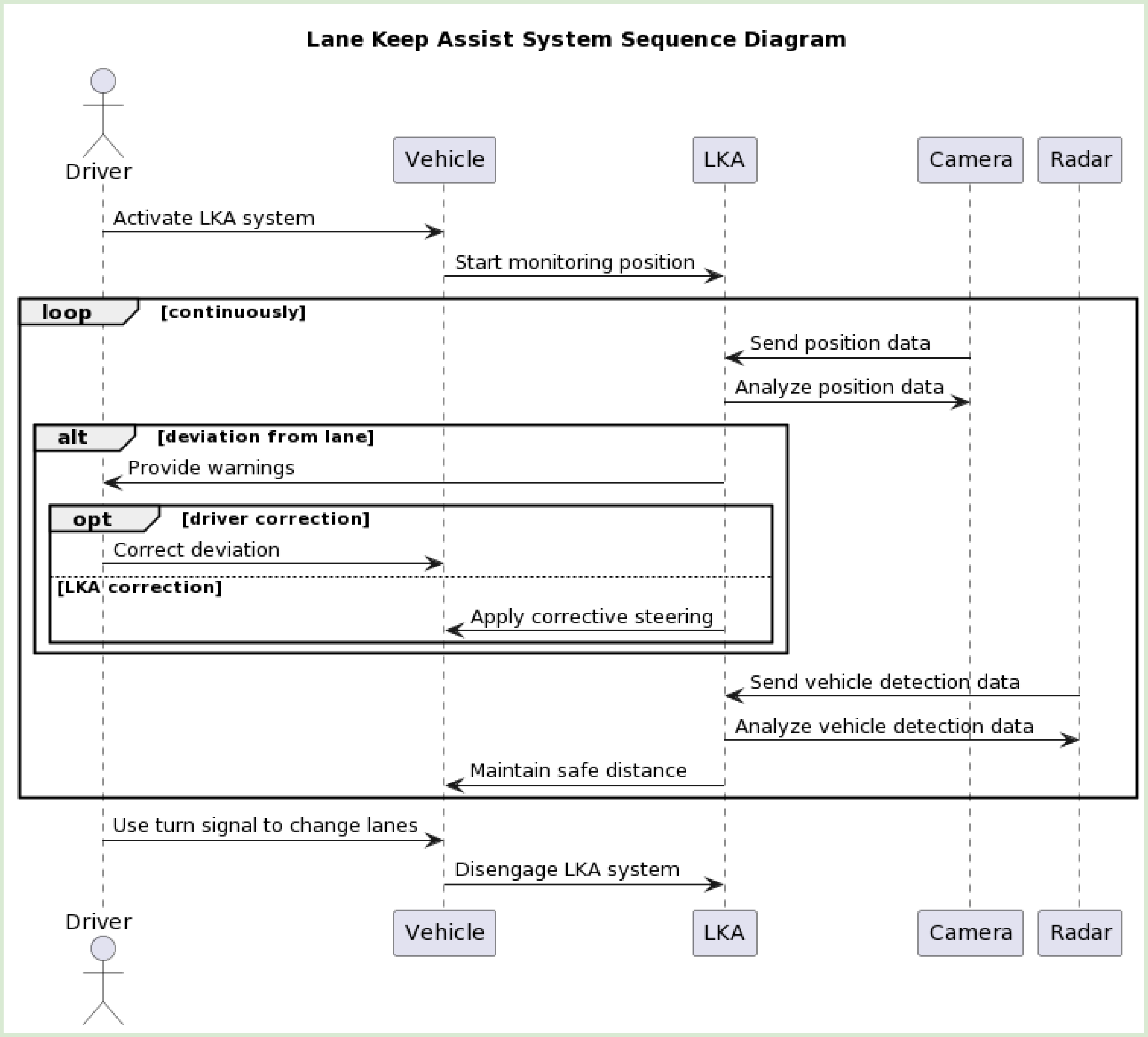

System Calibration
Technician connects calibration device to Car
Technician initiates calibration process
Calibration system receives calibration request
Calibration system retrieves current calibration data from Sensor Fusion system
Calibration system performs calibration process on sensors
Calibration successful?
yes
no
Calibration system sends updated calibration data to Sensor Fusion system
Sensor Fusion system receives updated calibration data
Sensor Fusion system verifies sensors are calibrated
Sensor Fusion system sends notification to Technician that calibration is complete
Sensor Fusion system sends error message to Calibration system
Calibration system sends notification to Technician that calibration failed

4.2.5 System Shutdown

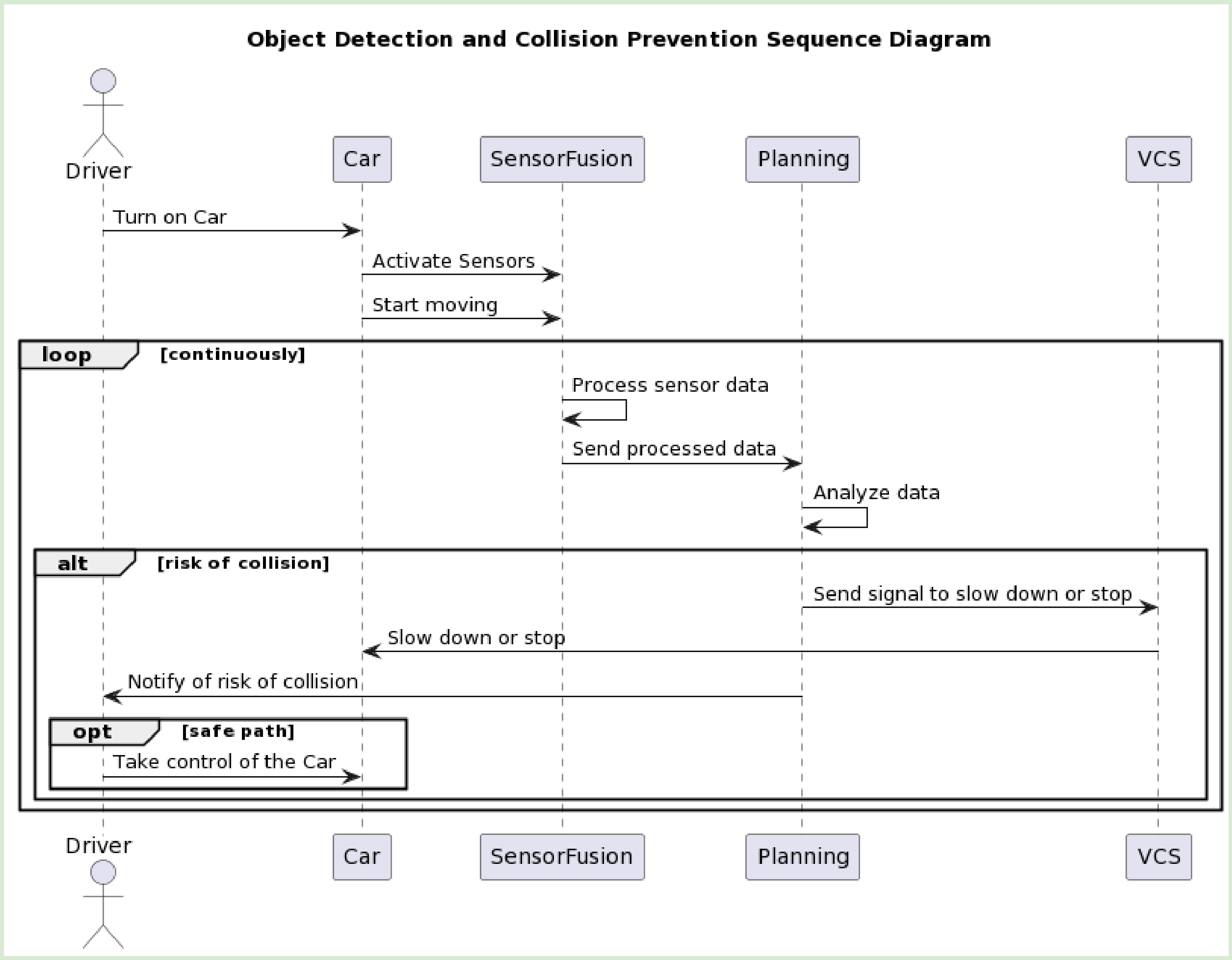

Press shutdown button
Planning system receives shutdown request?
yes
no
Send signal to VCS to turn off engine
Planning system malfunction
Sensor Fusion system receives shutdown signal?
yes
no
Send signal to all sensors to turn off
Sensor Fusion system malfunction
Send signal to Display system to turn off display
Display system turns off display?
yes
no
Send notification to System Admin that Car has been shut down
Display system malfunction
Turn off all systems

4.3: Sequence Diagrams:

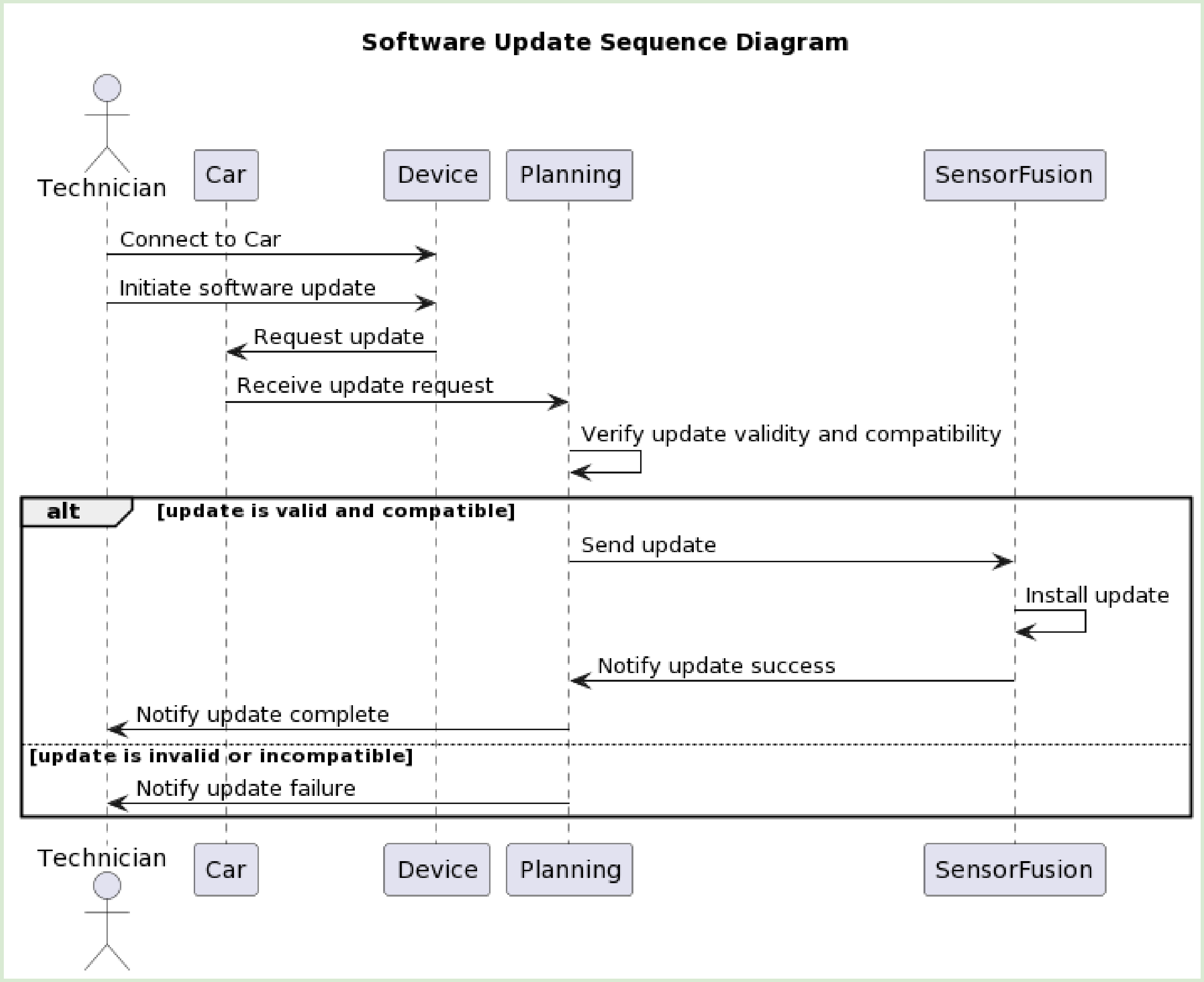
4.3.1 Lane Keep Assist



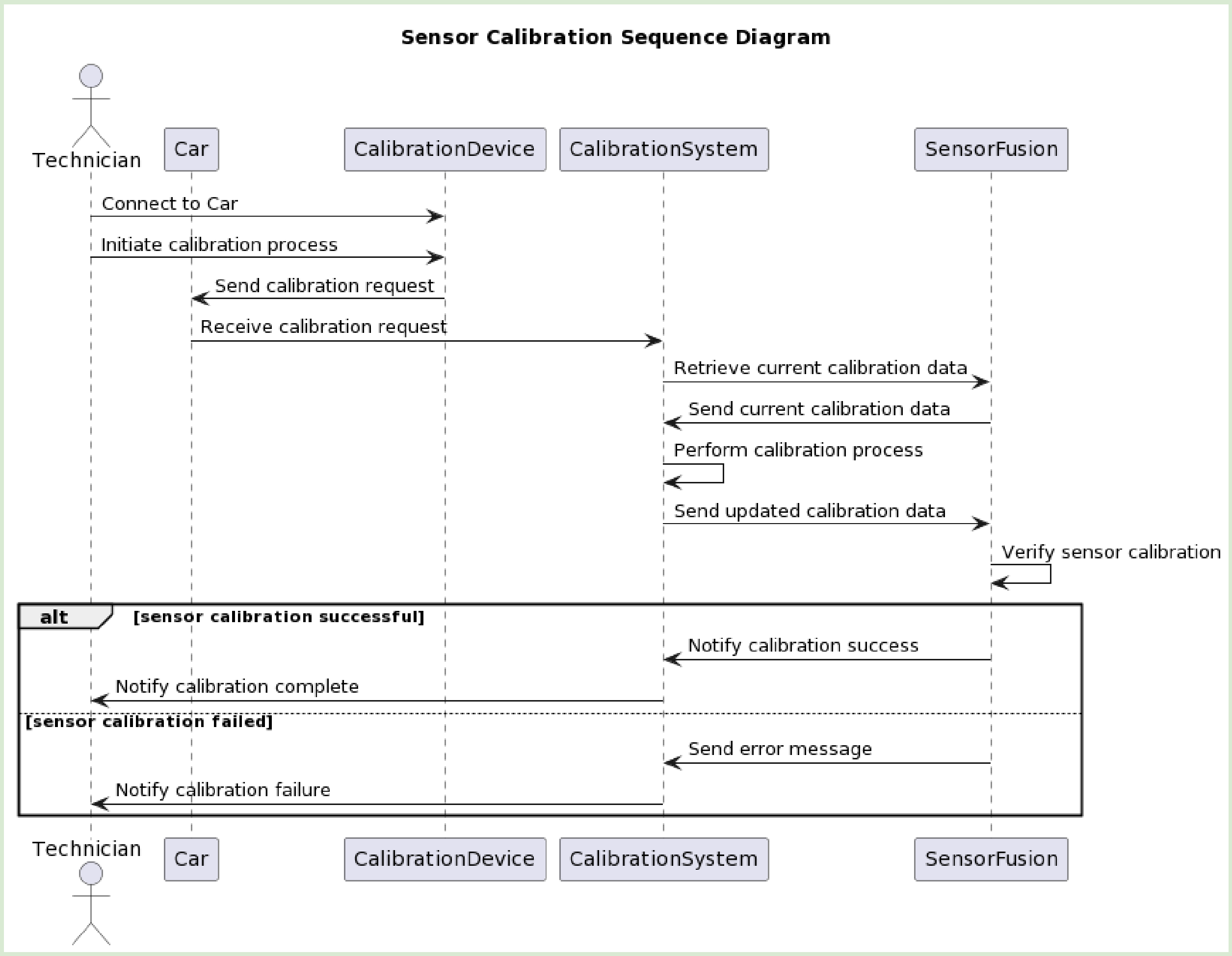
4.3.2 Object Detection and Collision Prevention



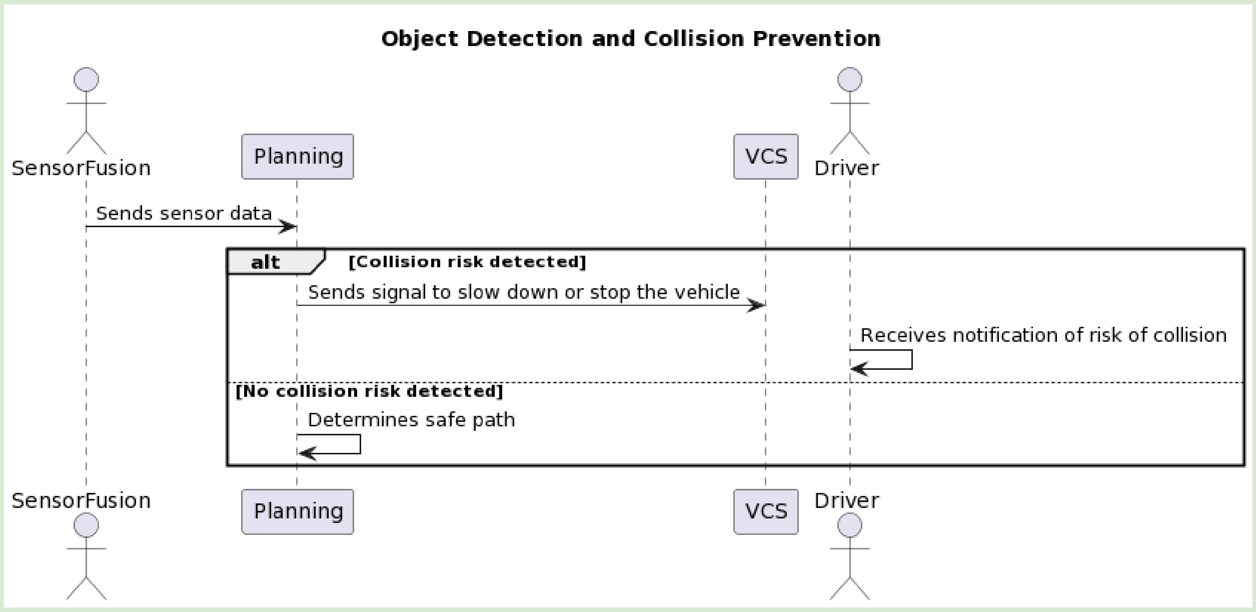
4.3.3 Software Update



4.3.4 System Calibration


Sensor Calibration Sequence Diagram
Technician
Car
CalibrationDevice
CalibrationSystem
SensorFusion
Connect to Car
Initiate calibration process
Send calibration request
Receive calibration request
Retrieve current calibration data
Send current calibration data
Perform calibration process
Send updated calibration data
Verify sensor calibration
alt
[sensor calibration successful]
Notify calibration success
Notify calibration complete
[sensor calibration failed]
Send error message
Notify calibration failure

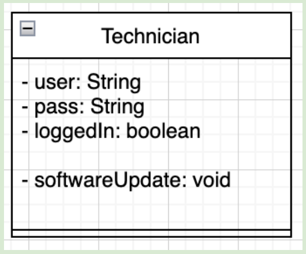
4.3.5 System Shutdown



Object Detection and Collision Prevention
SensorFusion
Planning
VCS
Driver
Sends sensor data
alt
[Collision risk detected]
Sends signal to slow down or stop the vehicle
Receives notification of risk of collision
[No collision risk detected]
Determines safe path
SensorFusion
Planning
VCS
Driver

4.4: Classes:

We will need a car class so we can create an instance each time the car turns on. The systems interfaces and classes will extend this, including startup and shutdown instructions and methods. The real time monitoring will occur via the sensors, which our code will consistently confirm is keeping our car in a safe range. If not, it will automatically call the methods to correct the car's placement on the road.



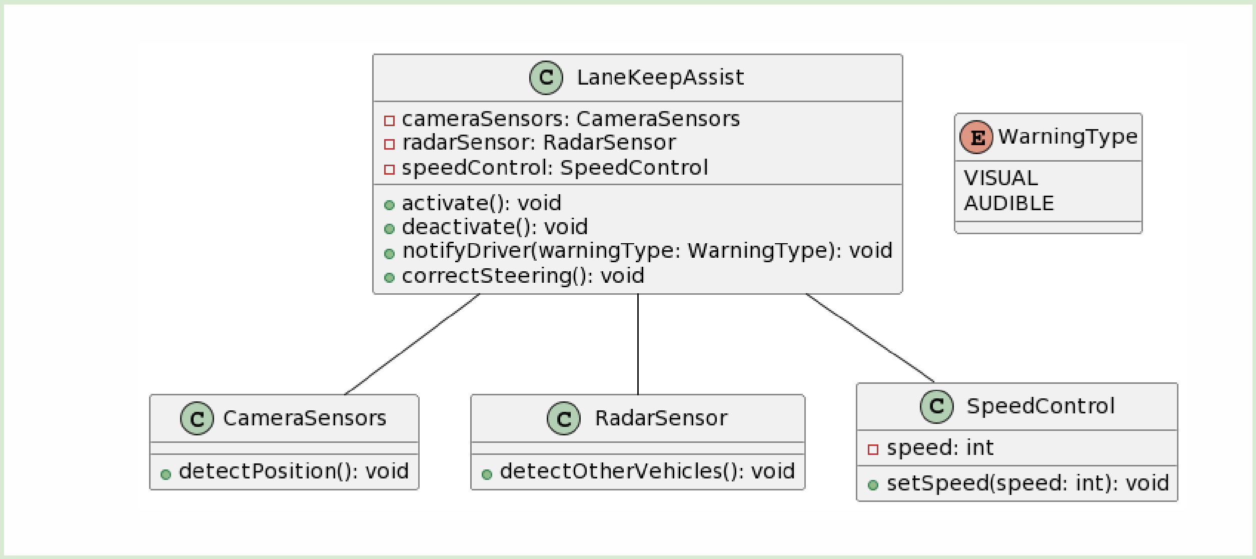
Here is an updated class diagram, for our driver interface:

- Private Members:
    - dashboard: Dashboard
    - navigation: Navigation
    - media: Media
    - settings: Settings
- Public Methods:
    - DriverInterface(): Constructor for the class
    - displayDashboard(): Displays the dashboard

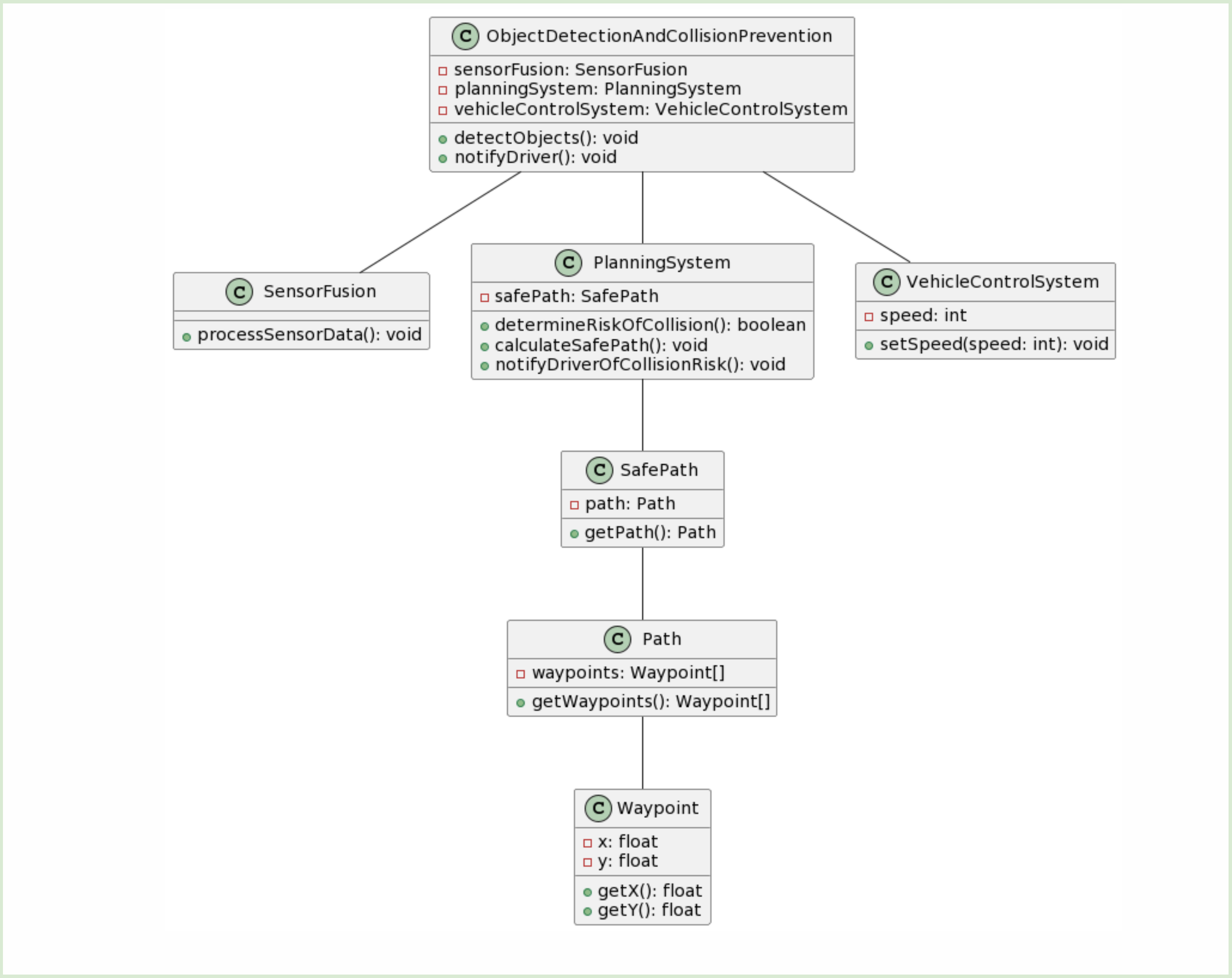
    - setDestination(destination: String): Sets the destination for navigation
    - getDestination(): Returns the current destination
    - playMedia(media: String): Plays the specified media
    - pauseMedia(): Pauses the current media
    - stopMedia(): Stops the current media
    - setSeatPosition(position: String): Sets the seat position
    - getSeatPosition(): Returns the current seat position
    - setClimateControl(climate: String): Sets the climate control settings
    - getClimateControl(): Returns the current climate control settings
    - setDisplayPreferences(display: String): Sets the display preferences
    - getDisplayPreferences(): Returns the current display preferences
- Private Classes:
    - Dashboard:
        - Private Members:
            - speed: int
            - fuelLevel: int
            - batteryLevel: int
        - Public Methods:
            - Dashboard(): Constructor for the class
            - show(): Displays the dashboard information
            - getSpeed(): Returns the current speed
            - setSpeed(speed: int): Sets the current speed
            - getFuelLevel(): Returns the current fuel level
            - setFuelLevel(fuelLevel: int): Sets the current fuel level
            - getBatteryLevel(): Returns the current battery level
            - setBatteryLevel(batteryLevel: int): Sets the current battery level
    - Navigation:
        - Private Members:
            - destination: String
        - Public Methods:
            - Navigation(): Constructor for the class
            - setDestination(destination: String): Sets the destination for navigation
            - getDestination(): Returns the current destination
    - Media:
        - Private Members:
            - isPlaying: boolean
        - Public Methods:
            - Media(): Constructor for the class

        - play(media: String): Plays the specified media
        - pause(): Pauses the current media
        - stop(): Stops the current media
        - isPlaying(): Returns whether media is currently playing
- Settings:
    - Private Members:
        - seatPosition: String
        - climateControl: String
        - displayPreferences: String
    - Public Methods:
        - Settings(): Constructor for the class
        - setSeatPosition(position: String): Sets the seat position
        - getSeatPosition(): Returns the current seat position
        - setClimateControl(climate: String): Sets the climate control settings
        - getClimateControl(): Returns the current climate control settings
        - setDisplayPreferences(display: String): Sets the display preferences
        - getDisplayPreferences(): Returns the current display preferences

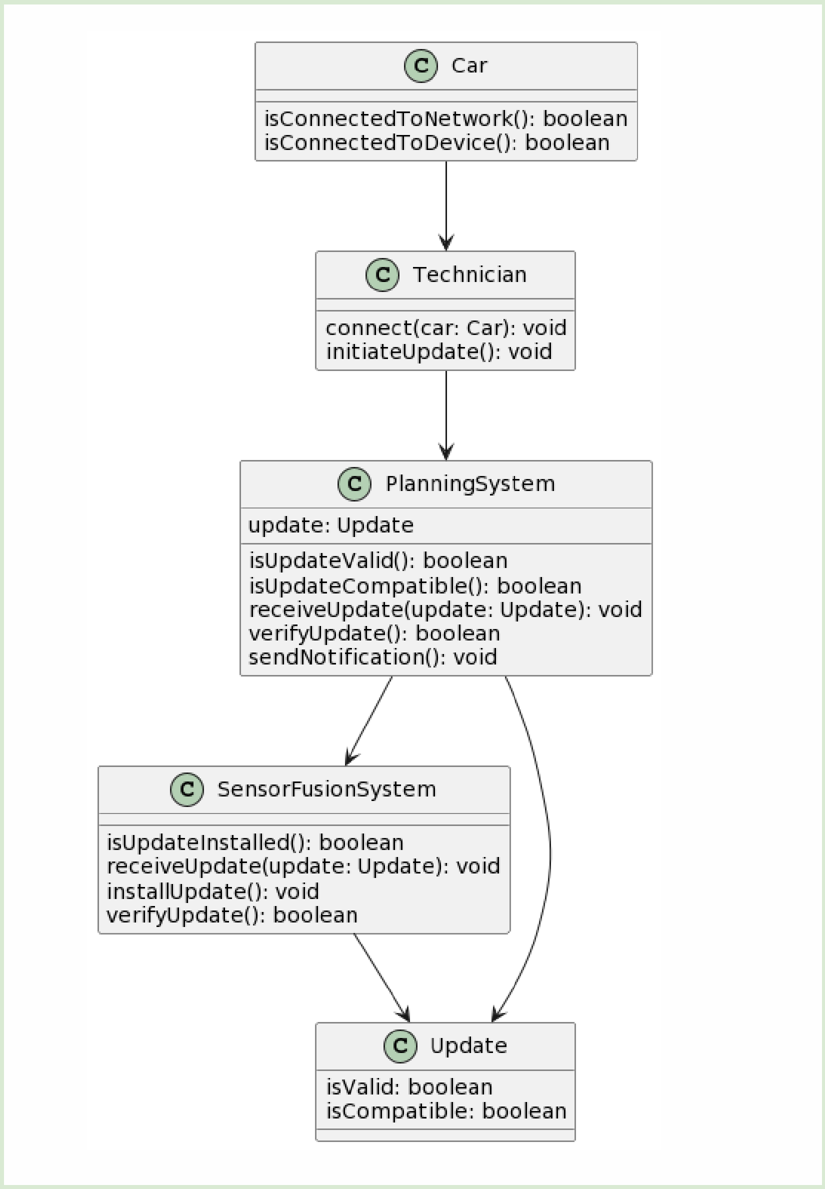
4.4.1 Lane Keep Assist



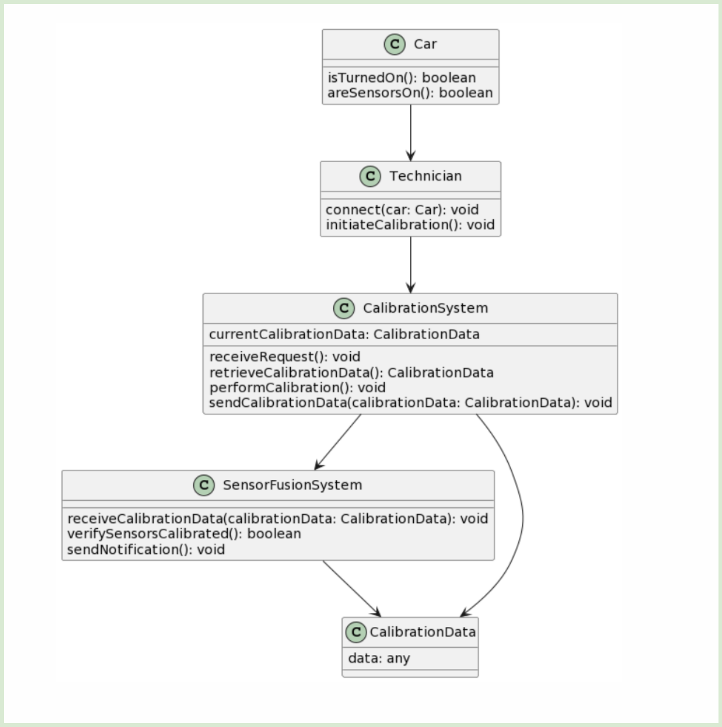
4.4.2 Object Detection and Collision Prevention


ObjectDetectionAndCollisionPrevention
sensorFusion: SensorFusion
planningSystem: PlanningSystem
vehicleControlSystem: VehicleControlSystem
detectObjects(): void
notifyDriver(): void
SensorFusion
processSensorData(): void
PlanningSystem
safePath: SafePath
determineRiskOfCollision(): boolean
calculateSafePath(): void
notifyDriverOfCollisionRisk(): void
VehicleControlSystem
speed: int
setSpeed(speed: int): void
SafePath
path: Path
getPath(): Path
Path
waypoints: Waypoint[]
getWaypoints(): Waypoint[]
Waypoint
x: float
y: float
getX(): float
getY(): float

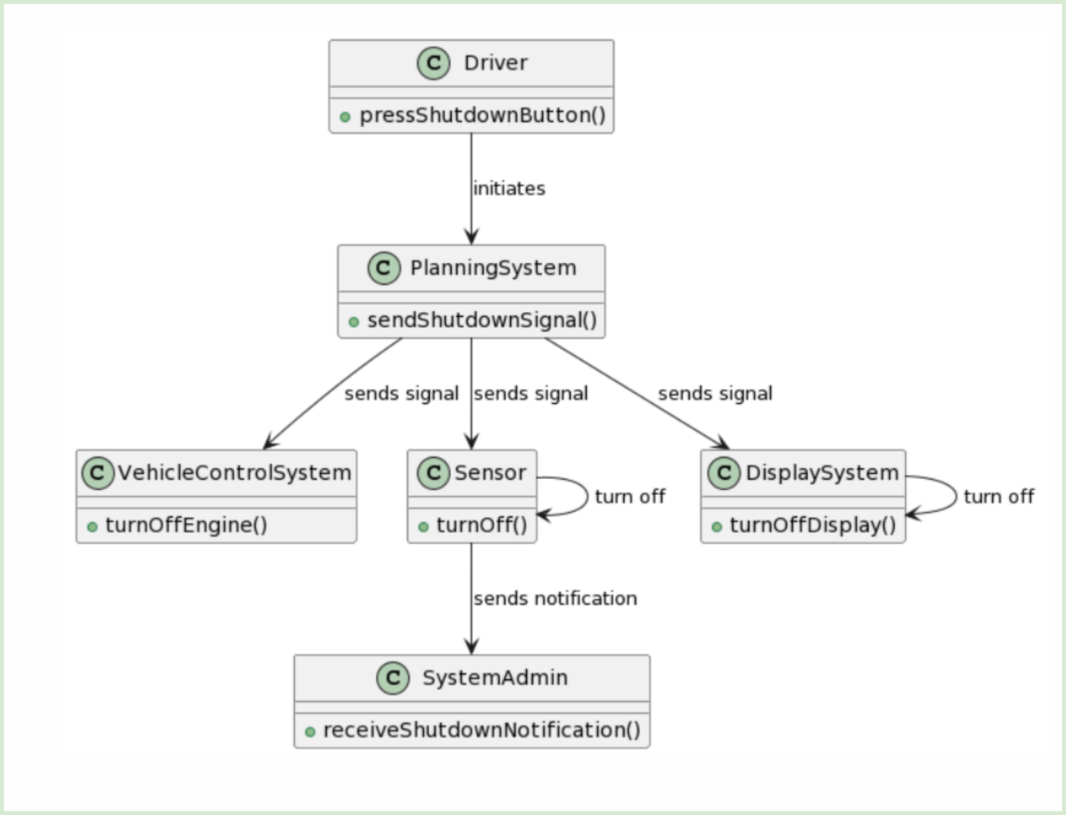
4.4.3 Software Update



4.4.4 System Calibration


Car
isTurnedOn(): boolean
areSensorsOn(): boolean
Technician
connect(car: Car): void
initiateCalibration(): void
CalibrationSystem
currentCalibrationData: CalibrationData
receiveRequest(): void
retrieveCalibrationData(): CalibrationData
performCalibration(): void
sendCalibrationData(calibrationData: CalibrationData): void
SensorFusionSystem
receiveCalibrationData(calibrationData: CalibrationData): void
verifySensorsCalibrated(): boolean
sendNotification(): void
CalibrationData
data: any

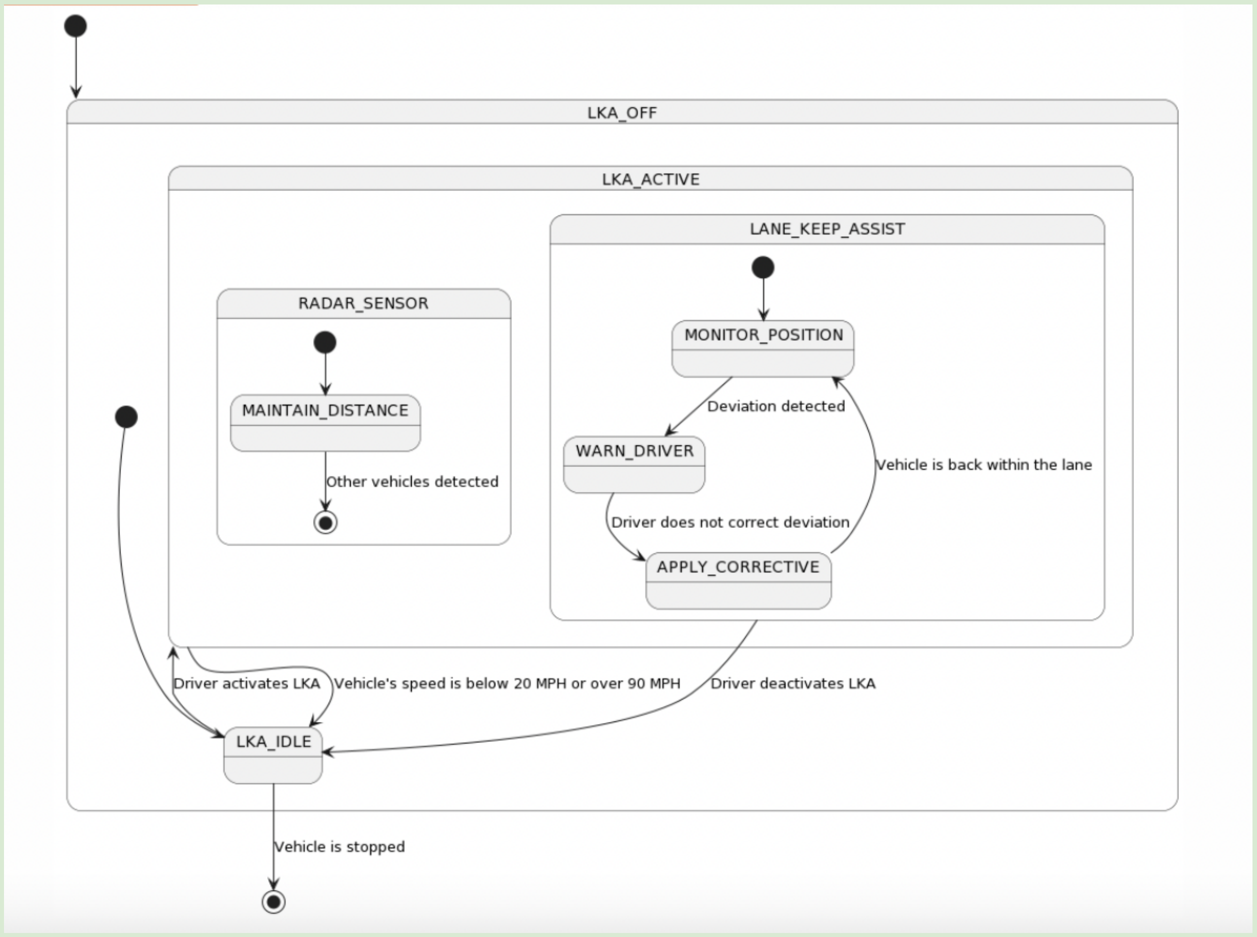
4.4.5 System Shutdown



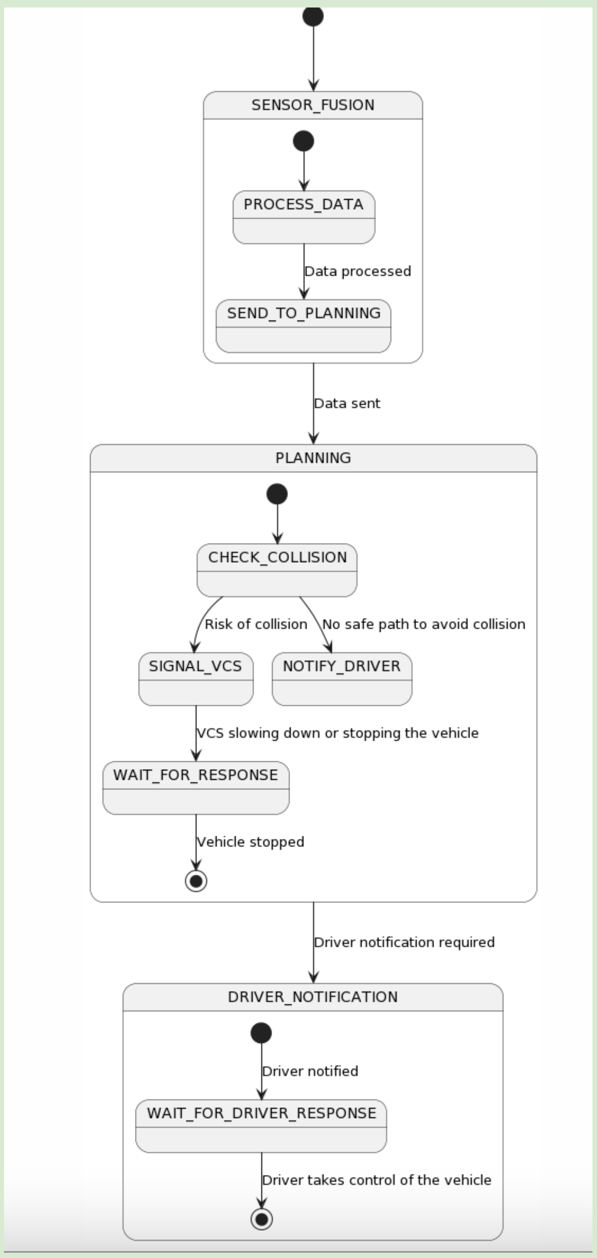
4.5: State Diagrams:

The greater structure of our classes will require a class for the sensors, sensor fusion, planning, display, and vcs. We have the technician and prompting for a password as a state.

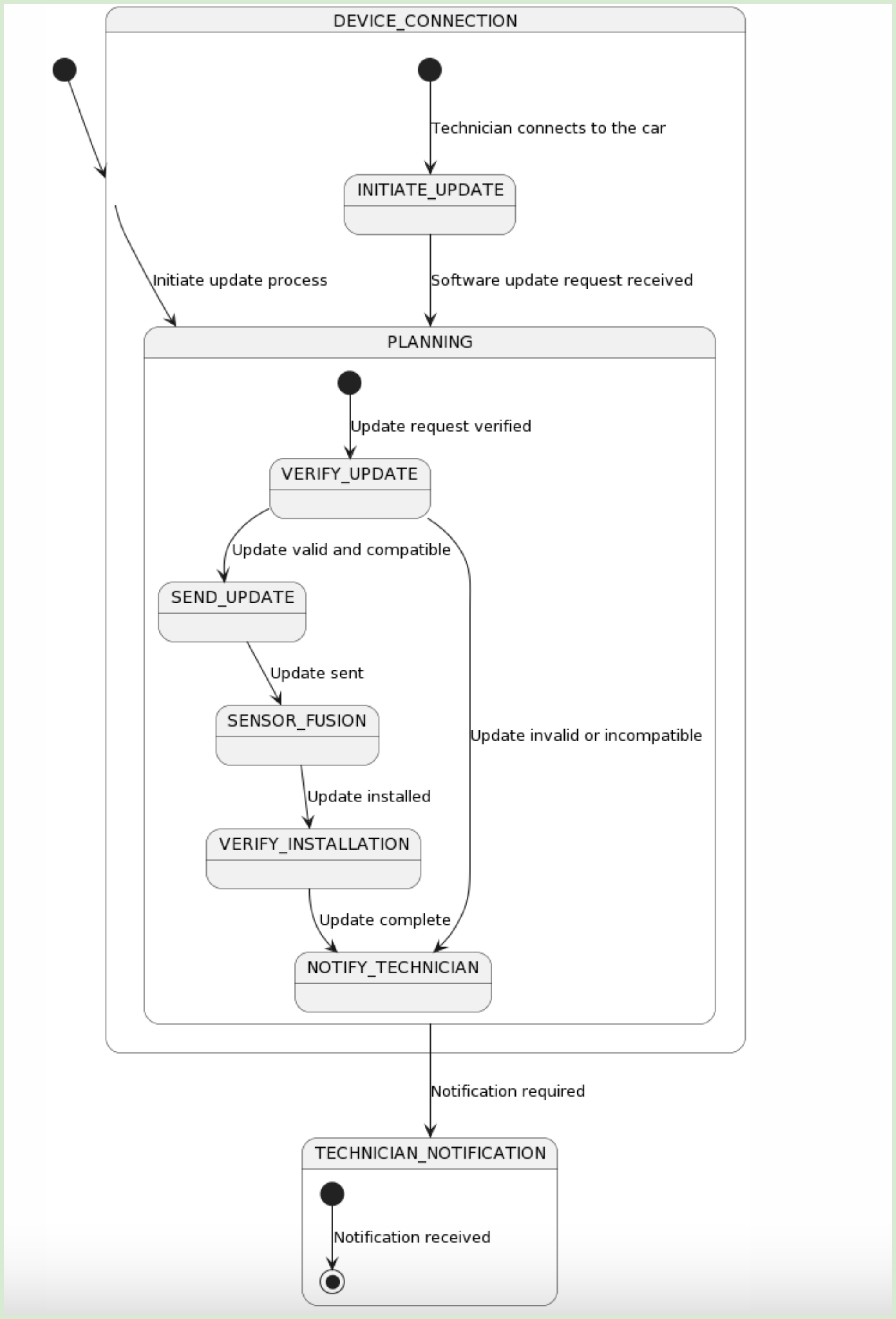
4.5.1 Lane Keep Assist



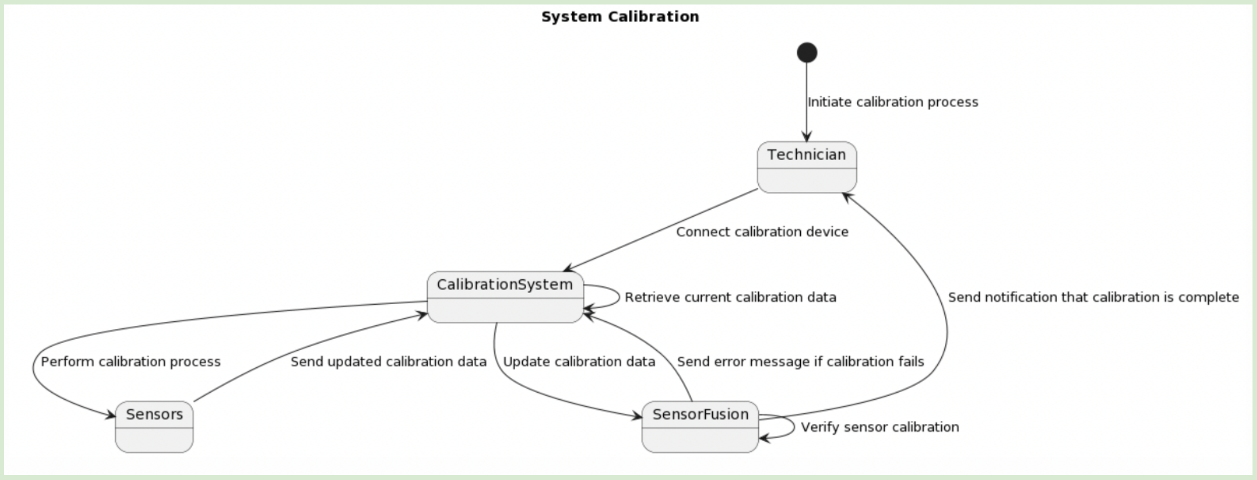
4.5.2 Object Detection and Collision Prevention



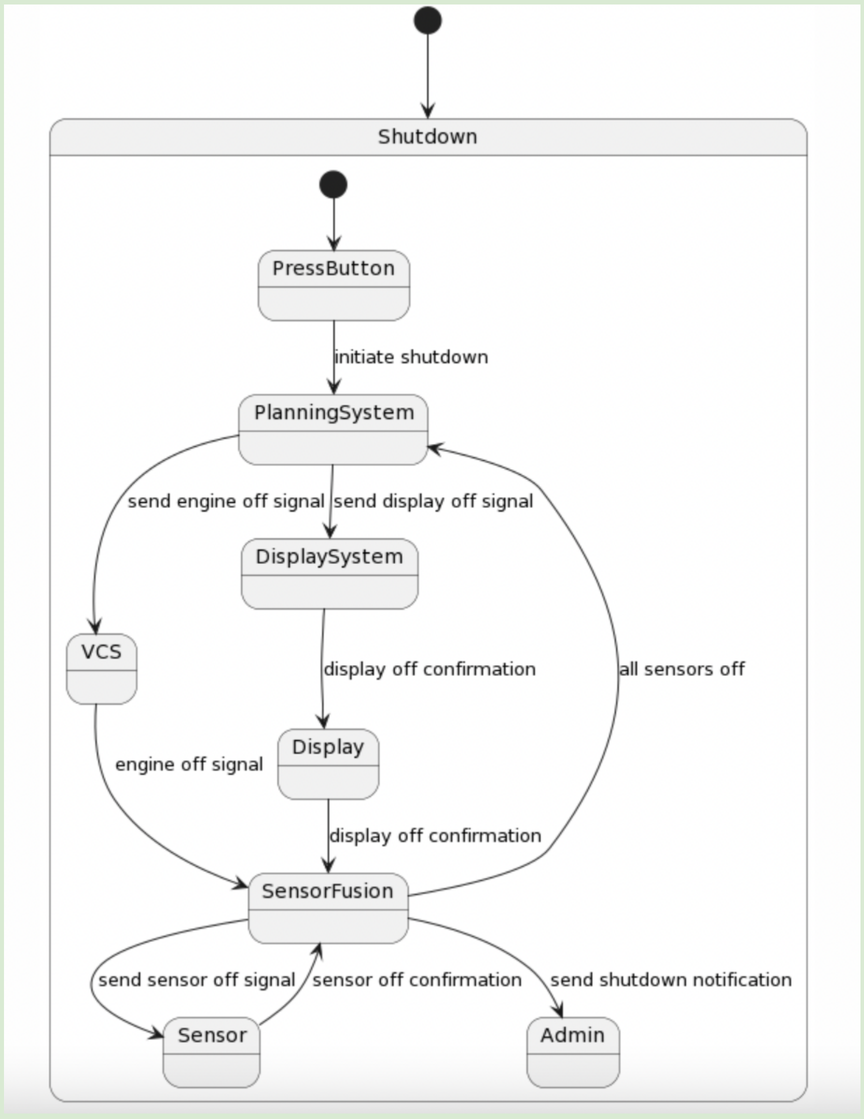
4.5.3 Software Update



4.5.4 System Calibration


System Calibration
Initiate calibration process
Technician
Connect calibration device
CalibrationSystem
Retrieve current calibration data
Send notification that calibration is complete
Perform calibration process
Send updated calibration data
Update calibration data
Send error message if calibration fails
Sensors
SensorFusion
Verify sensor calibration

4.5.5 System Shutdown



Section 5: Design

5.1 Software Architecture

The following software architecture models are considered:

5.1.1 Monolithic Architecture

This architecture is characterized by a single executable file that contains all components of the system. This architecture is not feasible for IoT HTL as it lacks scalability, flexibility, and reliability. The pros of this architecture are that it is easy to develop and test, but the cons are that it is difficult to maintain and update.

5.1.2 Client-Server Architecture

This architecture is characterized by a client that requests services from a server. This architecture is feasible for IoT HTL as it supports scalability, flexibility, and reliability. The pros of this architecture are that it is easy to maintain and update, but the cons are that it is complex to develop and test.

5.1.3 Microservices Architecture

This architecture is characterized by small, independent services that communicate with each other using APIs. This architecture is feasible for IoT HTL as it supports scalability, flexibility, and reliability. The pros of this architecture are that it is easy to maintain and update, but the cons are that it is complex to develop and test.

5.1.4 Event-Driven Architecture

This architecture is characterized by events that trigger actions. This architecture is feasible for IoT HTL as it supports scalability, flexibility, and reliability. The pros of this architecture are that it is easy to maintain and update, but the cons are that it is complex to develop and test.

5.1.5 Service-Oriented Architecture

This architecture is characterized by loosely-coupled, reusable services that communicate with each other using protocols. This architecture is feasible for IoT HTL as it supports scalability, flexibility, and reliability. The pros of this architecture are that it is easy to maintain and update, but the cons are that it is complex to develop and test.

5.1.6 Choice of Software Architecture

Based on the above pros and cons, we have chosen the Service-Oriented Architecture for the IoT HTL. This architecture supports the scalability, flexibility, and reliability that we require for the system. It also allows for easy maintenance and updates, which is essential for the success of the project.

5.2 Interface Design

In this subsection, we will define how software elements, hardware elements, and end-users communicate. We will consider both Technician and Driver interfaces and specify exactly what will appear in each of the user interfaces with names, class, data, etc.

5.2.1 Technician Interface

The Technician Interface will allow Alset-certified technicians to see performance statistics and manually modify certain settings should the car need personalized updates. They will have access to the system administration settings. The interface will have the following components:

- Dashboard: This will show an overview of the system's performance statistics, such as CPU usage, memory usage, and network usage.
- Settings: This will allow the technician to modify certain settings of the system, such as network settings, security settings, and performance settings.
- Logs: This will show a log of all system events, such as errors, warnings, and user actions.

5.2.2 Driver Interface

The Driver Interface will allow the driver to interact with the car's software and view vital information about the car. It will also allow the driver to modify certain settings for their personalized experience. The interface will have the following components:

- Dashboard: This will show an overview of the car's performance statistics, such as speed, fuel level, and battery level.
- Navigation: This will allow the driver to input a destination and get directions to that location.
- Media: This will allow the driver to control the car's media system, such as the radio or music player.
- Settings: This will allow the driver to modify certain settings of their personalized experience, such as seat position, climate control, and display preferences.

5.3 Component–level design

In this subsection, we will describe the detailed design of each software component identified in the software architecture. We will define the inputs, outputs, and operations of each component.

5.3.1 User Interface Component

The User Interface component provides a graphical user interface for technicians and drivers to interact with the system. It receives input from the user and sends output to the appropriate component. Its operations include displaying information to the user, receiving user input, and sending commands to other components.

Inputs: User input from mouse, keyboard, or touchscreen.
Outputs: Graphical user interface for display to the user.
Operations:

Display information to the user
Receive user input
Send commands to other components

5.3.2 Data Management Component

The Data Management component is responsible for storing and retrieving data used by the system. It receives requests for data from other components and returns the requested data. Its operations include storing data, retrieving data, and modifying data. Accessing and writing to the log file are both important parts of data management. The input file for the vehicle is all input-based, with a variety of sensors mentioned.

Inputs: Requests for data from other components.
Outputs: Requested data.
Operations:

Store data
Retrieve data
Modify data

5.3.3 Control Component

The Control component is responsible for controlling the car's functions. It receives commands from the User Interface component and sends signals to the appropriate hardware components. Its operations include receiving commands, interpreting commands, and sending signals.

Our sensors will measure data, like how far the car is from the car in front of it, then pass this information to Sensor Fusion. Should there be discrepancies in the measurements, Sensor Fusion will automatically find their average and use that as the data input, before acting accordingly.

Inputs: Commands from the User Interface component.
Outputs: Signals to hardware components.
Operations:

Receive commands
Interpret commands
Send signals

5.3.4 Communication Component

The Communication component is responsible for communicating with other devices and systems, such as GPS or remote servers. It receives requests for communication from other components and sends communication signals. Its operations include sending and receiving data over various communication protocols.

Inputs: Requests for communication from other components.
Outputs: Communication signals.
Operations:

Send data over communication protocols
Receive data over communication protocols

5.3.5 Security Component

The Security component is responsible for ensuring the security and integrity of the system. It receives requests for security checks from other components and returns security statuses. Its operations include performing security checks, logging security events, and alerting administrators of security breaches.

Inputs: Requests for security checks from other components.
Outputs: Security statuses.
Operations:

Perform security checks
Log security events
Alert administrators of security breaches

5.3.6 Network Component

The Network component is responsible for managing network connections and traffic. It receives requests for network access from other components and returns network statuses. Its operations include managing network connections, monitoring network traffic, and optimizing network performance.

Inputs: Requests for network access from other components.
Outputs: Network statuses.
Operations:
- Manage network connections
- Monitor network traffic
- Optimize network performance

Overall, this component-level design provides a detailed description of each component's inputs, outputs, and operations. With this information, developers can implement each component and ensure that they interact with each other according to the software architecture.

5.4 - Chosen Architecture

Based on the pros and cons mentioned in the document, the chosen software architecture for the IoT HTL is Service-Oriented Architecture. This architecture supports scalability, flexibility, and reliability, which are necessary requirements for the project. It also allows for easy maintenance and updates, making it the ideal choice for the system.

The interface design includes the Technician Interface and Driver Interface. The Technician Interface will have a dashboard, settings, and logs components, while the Driver Interface will have a dashboard, navigation, media, and settings components.

The component-level design includes the User Interface Component, Data Management Component, and Control Component. The User Interface Component provides a graphical interface for users to interact with the system, while the Data Management Component is responsible for storing and retrieving data used by the system. Finally, the Control Component is responsible for controlling the car's functions, receiving commands from the User Interface Component, and sending signals to the appropriate hardware components.

Section 6: Project Code

6.1 Car

```java
package src;

public class CarInterface {
    private boolean doorsClosed;
    private boolean goingAnywhere;
    private boolean seatbeltsSecure;
    private boolean alarm;

    // getters and setters
    public boolean isDoorsClosed() {
        return doorsClosed;
    }
    public void setDoorsClosed(boolean doorsClosed) {
        this.doorsClosed = doorsClosed;
    }
    public boolean isGoingAnywhere() {
        return goingAnywhere;
    }
    public void setGoingAnywhere(boolean goingAnywhere) {
        this.goingAnywhere = goingAnywhere;
    }
    public boolean isSeatbeltsSecure() {
        return seatbeltsSecure;
    }
    public void setSeatbeltsSecure(boolean seatbeltsSecure) {
        this.seatbeltsSecure = seatbeltsSecure;
    }
    public boolean hasAlarm() {
        return alarm;
    }
    public void setAlarm(boolean alarm) {
        this.alarm = alarm;
    }
}
```

The code above defines a `CarInterface` class that serves as a blueprint for creating car objects with specific properties and functionalities. Each car object has properties such as doorsClosed, goingAnywhere, seatbeltsSecure and alarm. The class also includes methods for getting and setting these properties. It primarily is a template for creating car modules.

6.2 Driver

```java
package src;

public class DriverInterface {

    private Dashboard dashboard;
    private Navigation navigation;
    private Media media;
    private Settings settings;

    public DriverInterface() {
        this.dashboard = new Dashboard();
        this.navigation = new Navigation();
        this.media = new Media();
        this.settings = new Settings();
    }

    public void displayDashboard() {
        this.dashboard.show();
    }

    public void setDestination(String destination) {
        this.navigation.setDestination(destination);
    }

    public String getDestination() {
        return this.navigation.getDestination();
    }

    public void playMedia(String media) {
        this.media.play(media);
    }

    public void pauseMedia() {
        this.media.pause();
    }

    public void stopMedia() {
        this.media.stop();
    }

    public void setSeatPosition(String position) {
        this.settings.setSeatPosition(position);
    }
    public String getSeatPosition() {
        return this.settings.getSeatPosition();
    }

    public void setClimateControl(String climate) {
        this.settings.setClimateControl(climate);
    }

    public String getClimateControl() {
        return this.settings.getClimateControl();
    }

    public void setDisplayPreferences(String display) {
        this.settings.setDisplayPreferences(display);
    }

    public String getDisplayPreferences() {
        return this.settings.getDisplayPreferences();
    }

    private class Dashboard {

        private int speed;
        private int fuelLevel;
        private int batteryLevel;

        public Dashboard() {
            this.speed = 0;
            this.fuelLevel = 100;
            this.batteryLevel = 100;
        }

        public void show() {
            System.out.println("Speed: " + this.speed + " mph");
            System.out.println("Fuel Level: " + this.fuelLevel + "%");
            System.out.println("Battery Level: " + this.batteryLevel + "%");
        }

        public int getSpeed() {
            return this.speed;
        }

        public void setSpeed(int speed) {
            this.speed = speed;
        }

        public int getFuelLevel() {
            return this.fuelLevel;
        }

        public void setFuelLevel(int fuelLevel) {
            this.fuelLevel = fuelLevel;
        }

        public int getBatteryLevel() {
            return this.batteryLevel;
        }

        public void setBatteryLevel(int batteryLevel) {
            this.batteryLevel = batteryLevel;
        }
    }

    private class Navigation {

        private String destination;

        public Navigation() {
            this.destination = "";
        }

        public void setDestination(String destination) {
            this.destination = destination;
        }

        public String getDestination() {
            return this.destination;
        }
    }

    private class Media {

        private boolean isPlaying;

        public Media() {
            this.isPlaying = false;
        }

        public void play(String media) {
            System.out.println("Playing " + media);
            this.isPlaying = true;
        }

        public void pause() {
            System.out.println("Media paused");
            this.isPlaying = false;
        }

        public void stop() {
            System.out.println("Media stopped");
            this.isPlaying = false;
        }

        public boolean isPlaying() {
            return this.isPlaying;
        }
    }

    private class Settings {

        private String seatPosition;
        private String climateControl;
        private String displayPreferences;

        public Settings() {
            this.seatPosition = "Default";
            this.climateControl = "Default";
            this.displayPreferences = "Default";
        }

        public void setSeatPosition(String position) {
            this.seatPosition = position;
        }

        public String getSeatPosition() {
            return this.seatPosition;
        }

        public void setClimateControl(String climate) {
            this.climateControl = climate;
        }

        public String getClimateControl() {
            return this.climateControl;
        }

        public void setDisplayPreferences(String display) {
            this.displayPreferences = display;
        }

        public String getDisplayPreferences() {
            return this.displayPreferences;
        }
    }
```

```
}
```

The code above defines a `DriverInterface` class that serves as a blueprint for creating driver objects with specific properties and functionalities. Each driver object has properties such as dashboard, navigation, media and settings. The class also includes methods for getting and setting these properties. It primarily is a template for creating driver modules.

6.3 Technician

The Technician code allows the technician to see all the important behind-the-scene statistics present about the vehicle, like its memory usage, CPU usage, and network usage. These are stored in logs, also, which corporate data analysts can use to continue optimizing the ALSET software.

```
package src;

import java.util.ArrayList;
import java.util.List;

public class TechnicianInterface {

    private Dashboard dashboard;
    private Settings settings;
    private Logs logs;

    public TechnicianInterface() {
        this.dashboard = new Dashboard();
        this.settings = new Settings();
        this.logs = new Logs();
    }

    public void openDashboard() {
        this.dashboard.show();
    }

    public void openSettings() {
        this.settings.show();
    }

    public void openLogs() {
        this.logs.show();
    }
    public int getDashboardCpuUsage() {
        return this.dashboard.getCpuUsage();
    }

    public void setDashboardCpuUsage(int cpuUsage) {
        this.dashboard.setCpuUsage(cpuUsage);
    }

    public int getDashboardMemoryUsage() {
        return this.dashboard.getMemoryUsage();
    }

    public void setDashboardMemoryUsage(int memoryUsage) {
        this.dashboard.setMemoryUsage(memoryUsage);
    }

    public int getDashboardNetworkUsage() {
        return this.dashboard.getNetworkUsage();
    }

    public void setDashboardNetworkUsage(int networkUsage) {
        this.dashboard.setNetworkUsage(networkUsage);
    }

    public String getSettingsNetworkSettings() {
        return this.settings.getNetworkSettings();
    }

    public void setSettingsNetworkSettings(String networkSettings) {
        this.settings.setNetworkSettings(networkSettings);
    }

    public String getSettingsSecuritySettings() {
        return this.settings.getSecuritySettings();
    }

    public void setSettingsSecuritySettings(String securitySettings) {
        this.settings.setSecuritySettings(securitySettings);
    }

    public String getSettingsPerformanceSettings() {
        return this.settings.getPerformanceSettings();
    }

    public void setSettingsPerformanceSettings(String performanceSettings) {
        this.settings.setPerformanceSettings(performanceSettings);
    }
```

```
public void addLogEvent(String event) {
    this.logs.addEvent(event);
}

public List<String> getLogEvents() {
    return this.logs.getSystemEvents();
}

private class Dashboard {

    private int cpuUsage;
    private int memoryUsage;
    private int networkUsage;

    public Dashboard() {
        this.cpuUsage = 0;
        this.memoryUsage = 0;
        this.networkUsage = 0;
    }

    public void show() {
        System.out.println("CPU usage: " + this.cpuUsage);
        System.out.println("Memory usage: " + this.memoryUsage);
        System.out.println("Network usage: " + this.networkUsage);
    }

    public int getCpuUsage() {
        return this.cpuUsage;
    }

    public void setCpuUsage(int cpuUsage) {
        this.cpuUsage = cpuUsage;
    }

    public int getMemoryUsage() {
        return this.memoryUsage;
    }

    public void setMemoryUsage(int memoryUsage) {
        this.memoryUsage = memoryUsage;
    }

    public int getNetworkUsage() {
        return this.networkUsage;
    }
```

```
        public void setNetworkUsage(int networkUsage) {
            this.networkUsage = networkUsage;
        }
    }

    private class Settings {

        private String networkSettings;
        private String securitySettings;
        private String performanceSettings;

        public Settings() {
            this.networkSettings = "default";
            this.securitySettings = "default";
            this.performanceSettings = "default";
        }

        public void show() {
            System.out.println("Network settings: " + this.networkSettings);
            System.out.println("Security settings: " + this.securitySettings);
            System.out.println("Performance settings: " + this.performanceSettings);
        }

        public String getNetworkSettings() {
            return this.networkSettings;
        }

        public void setNetworkSettings(String networkSettings) {
            this.networkSettings = networkSettings;
        }

        public String getSecuritySettings() {
            return this.securitySettings;
        }

        public void setSecuritySettings(String securitySettings) {
            this.securitySettings = securitySettings;
        }

        public String getPerformanceSettings() {
            return this.performanceSettings;
        }

        public void setPerformanceSettings(String performanceSettings) {
            this.performanceSettings = performanceSettings;
        }
    }
    private class Logs {

        private List<String> systemEvents;

        public Logs() {
            this.systemEvents = new ArrayList<>();
        }

        public void show() {
            for (String event : this.systemEvents) {
                System.out.println(event);
            }
        }

        public void addEvent(String event) {
            this.systemEvents.add(event);
        }

        public List<String> getSystemEvents() {
            return this.systemEvents;
        }
    }
}
```

The code above defines a `TechnicianInterface` class that serves as a blueprint for creating technician objects with specific properties and functionalities. Each technician object has properties such as dashboard, settings and logs. The class also includes methods for getting and setting these properties. It primarily is a template for creating technician modules.

6.4 Log

```
package src;

import java.util.Date;

public class LogInterface {
    private Date timestamp;
    private boolean motion;
    private boolean seatbeltsOn;
    private double pressure;
    private int lightLevels;
    private int traffic;
    private double speed;
    private double distance;
    private String location;
    private String activity;
    private String destination;
    private int temperature;

    public LogInterface(boolean motion, boolean seatbeltsOn, double 
pressure, int lightLevels, int traffic, double speed, double distance, 
String location, String activity, String destination, int temperature) {
        this.timestamp = new Date();
        this.motion = motion;
        this.seatbeltsOn = seatbeltsOn;
        this.pressure = pressure;
        this.lightLevels = lightLevels;
        this.traffic = traffic;
        this.speed = speed;
        this.distance = distance;
        this.location = location;
        this.activity = activity;
        this.destination = destination;
        this.temperature = temperature;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public boolean isMotion() {
        return motion;
    }

    public boolean isSeatbeltsOn() {
        return seatbeltsOn;
    }

    public double getPressure() {
        return pressure;
    }

    public int getLightLevels() {
        return lightLevels;
    }

    public int getTraffic() {
        return traffic;
    }

    public double getSpeed() {
        return speed;
    }

    public double getDistance() {
        return distance;
    }

    public String getLocation() {
        return location;
    }

    public String getActivity() {
        return activity;
    }

    public String getDestination() {
        return destination;
    }

    public int getTemperature() {
        return temperature;
    }
}
```

The code above defines a `LogInterface` class that serves as a blueprint for creating log objects with specific properties and functionalities. Each log object has properties such as timestamp, motion, seatbeltsOn, pressure, lightLevels, traffic, speed, distance, location, activity, destination and temperature. The class also includes methods for getting these properties. It primarily is a template for creating log modules.

6.5 SensorFusion

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

// uses the instance class which has the data
// we assume if there is data in there the car is on

public class SensorFusion {
    /* these are the normalized values that planning will need to access */
    double speed;
    double speedlimit;
    double nfd; // this is normalized following distance
    List<Double> data;
    double technician;
    double lightColor; // 0: green, 1: yellow, 3: red, -1: no light
    double lightsOn;
    double nlight;
    double[] followingDistance; // list of following distances, in meters
    double[] light; // light sensor

    public void main(String[] args) throws InterruptedException {
        SensorFusion sf = new SensorFusion("./data.csv");
    }

    /*
     * constructor of sensor fusion, takes a file path as the data to analyze, must
     * be in proper format!!
     */
    public SensorFusion(String filePath) throws InterruptedException {
        this.data = new ArrayList<>();
        try {
            // Create a FileReader object to read the file
            FileReader reader = new FileReader(filePath);

            // Create a BufferedReader object to read the file line by line
            BufferedReader bufferedReader = new BufferedReader(reader);
            bufferedReader.readLine(); // these are the titles, we do not care!

            // Read the file line by line and parse the data
            String line;
            int i = 0;
            /* this loop iterates through the whole data file */
            while ((line = bufferedReader.readLine()) != null) {

                String[] row = line.split(",");
                readData(row); // this sets the variables properly
                /* normalize all the variables */
                double nfd = normalize(followingDistance);
                double nlight = normalize(light);
                /* call planning so the current statistics can be adequately adjusted */
                planning();

                Thread.sleep(2000); // sleep for two seconds

            }
            // close file reader and buffer reader
            bufferedReader.close();
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    public void planning(){
        if (nfd < 2){ // if the following distance is not enough then slow down the speed
            speed --; // decrease speed
        }
        // turn the lights on if the ambient light is too low
        if (nlight < .3 ){
            lightsOn = 1;
            //decelerate;
        }
        if (lightColor == 2 || lightColor ==1){
            speed --;
            //decelerate;

        }


    }

    public void readData(String[] row) {

        this.speed = data.get(0);
        this.speedlimit = data.get(1);
        this.followingDistance[0] = data.get(2);
        this.followingDistance[1] = data.get(3);
        this.followingDistance[2] = data.get(4);
        this.light[0] = data.get(5);
        this.light[1] = data.get(6);
        this.light[2] = data.get(7);
        this.technician = data.get(8);
        this.lightsOn = data.get(9);

    }

    /* normalizes the data from the sensors, given in array */
    public double normalize(double[] sensors) {
        double total = 0;
        if (sensors.length == 0) { // this is if the sensor doesn't have data
            return -1;
        }
        for (double j : sensors) {
            if (j != -1) { // this sensor is not returning an error & unfunctional
                total += j; // add the values
            }
        }
        double avg = total / sensors.length;
        return avg;
    }

    // if (speed != 0 && techinician == 0){
    // System.out.println("technician cannot be accessed while the car is moving!");
    // }

}
```

The SensorFusion file accepts the path of a file as the car’s current and surrounding statistics. It iterates that file line-by-line until it reaches the end of the drive, represented through the end of the file, with each line representing a drive’s instance and variables each moment. This simulates the way actual sensors would read and report data to the vehicle, even though we don’t have access to those physically. For simulation’s sake, we created a .csv file that has a variety of values for multiple sensors measuring the same thing, like following distance and ambient light. These values are arranged in arrays of doubles that sensor fusion normalizes before sending them to Planning (calling the planning method). Each sensor has an array of doubles representing the variety of values read, each with its own, built-in units. The normalize function shown below takes the average of all the accurate values, the Normalized Data, which excludes any potential erroneous values, that are negative

```
   /* normalizes the data from the sensors, given in array */
   public double normalize(double[] sensors) {
       double total = 0;
       if (sensors.length == 0) { // this is if the sensor doesn't have data
           return -1;
       }
       for (double j : sensors) {
           if (!(j < 0) { // this sensor is not returning an error & unfunctional
               total += j; // add the values
           }
       }
       double avg = total / sensors.length;
       return avg;
   }
```

In planning we compare values like the Normalized Following Distance with what our safe limit is. If the car is too close to the one in front of it, for example, the vehicle will decelerate. Likewise if there is a red light or stop-sign registered, the car will decelerate before coming to a complete stop. The below example shows how the vehicle regulates its speed.

```
// if the light is green and the car is not going fast enough it accelerates
       else if (lightColor == 1 && speed < speedlimit + ACCEPTABLE_SPEED){
           // should accelerate to bring the car up to the speed limit
           accelerate(speedlimit-speed + ACCEPTABLE_SPEED);
       }
```

Planning takes these normalized values, then tells the Vehicle Control System (VCS) what values to manipulate, accordingly. We use methods like *accelerate()* and *decelerate()* to simulate the VCS changing vehicle attributes. However, these checks come after we confirm there are no vehicles close in front of ours, ensuring the proper following distance remains without a chance of an accident. Methods like these represent the VCS, changing states of the vehicle.

```
public void accelerate(double increase) { // increases car speed
       speed += increase;
   }
```

```
public void decelerate(double decrease) { // decreases car speed
    speed -= decrease;
}

public void turnOnHeadlights() { // turns on the headlights
    lightsOn = 0;
}
```

6.6 Display

```
package src;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import java.awt.CardLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class DisplayInterface extends JFrame {

    private DriverInterface driver;
    private JPanel mainPanel;
    private JPanel loginPanel;
    private JPanel updatePanel;

    public DisplayInterface(DriverInterface driver) {
        this.driver = driver;
        setTitle("Display");
        setSize(300, 200);

        mainPanel = new JPanel(new CardLayout());

        // Create login panel
        loginPanel = new JPanel();
        loginPanel.add(new JLabel("Username:"));
        JTextField usernameField = new JTextField(10);
        loginPanel.add(usernameField);
        loginPanel.add(new JLabel("Password:"));
        JTextField passwordField = new JTextField(10);
        loginPanel.add(passwordField);

        // Add submit button and action listener
        JButton submitButton = new JButton("Submit");
        submitButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                CardLayout cardLayout = (CardLayout) mainPanel.getLayout();
                cardLayout.next(mainPanel);
            }
        });
        loginPanel.add(submitButton);

        // Create update panel
        updatePanel = new JPanel();
        updatePanel.add(new JLabel("Update"));

        // Add panels to main panel with CardLayout
        mainPanel.add(loginPanel, "login");
        mainPanel.add(updatePanel, "update");
        add(mainPanel);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        DriverInterface driver = new DriverInterface();
        DisplayInterface display = new DisplayInterface(driver);
    }
```

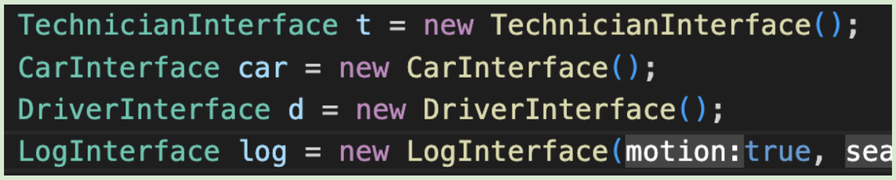
The code above defines a `DisplayInterface` class that extends the `JFrame` class and serves as a blueprint for creating display objects with specific properties and functionalities. Each display object has properties such as driver, mainPanel, loginPanel and updatePanel. The class also includes methods for getting these properties. It primarily is a template for creating display modules.

6.7 Main

The main function is where we create instances of other objects, as mentioned above, where we are able to then test that our software is functioning properly, as is necessary in the next section. After instantiating the interfaces, we manipulate the variables to ensure they are responding properly.

Section 7: Testing

We test our classes to ensure the software will function properly under a variety of conditions. These include erroneous inputs from sensors, indicating that they're unfunctional, and a variety of other edge cases that will test the capacity of our functionality. We instantiate the Interfaces in our Main.java file, then begin adjusting all values of our variables to ensure they are changing efficiently and accurately. Instances to test are declared in main, as follows.

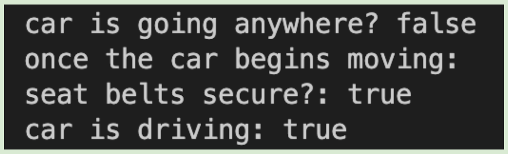


```
TechnicianInterface t = new TechnicianInterface();
CarInterface car = new CarInterface();
DriverInterface d = new DriverInterface();
LogInterface log = new LogInterface(motion:true, sea
```

We do not flesh out every test completed in this file, because that would be both redundant and boringly extensive. Rather we establish initial cases, then demonstrate the way attributes change as the vehicle moves.

7.1 Car

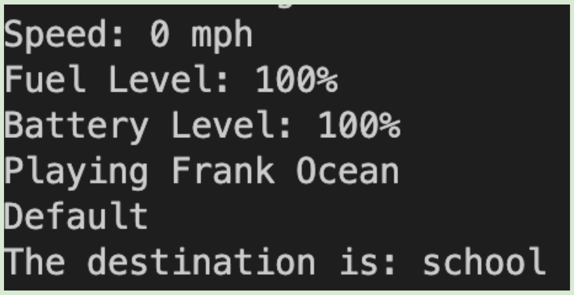
Before the car begins moving, it will accurately print the booleans related to its status. Once the seatbelts are fastened and the vehicle is moving, this interface will print that it is "true" that the vehicle is in motion.



```
car is going anywhere? false
once the car begins moving:
seat belts secure?: true
car is driving: true
```

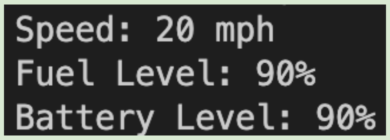
7.2 Driver

When first instantiated and asked to display the dashboard, the vehicle will accurately show the battery and fuel as fully charged, and reflect that the car is still. The cars are initially set to have all values set to false. Once the car begins moving, these values change, leading to this output and explanation. We can update what media the car is playing, as well as the destination. Just calling "d.displayDashboard()" prints current statistics.



```
Speed: 0 mph
Fuel Level: 100%
Battery Level: 100%
Playing Frank Ocean
Default
The destination is: school
```

The fuel level and battery level then drop as the speed increases, and are displayed as we continue manipulating them.

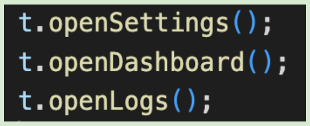


```
Speed: 20 mph
Fuel Level: 90%
Battery Level: 90%
```

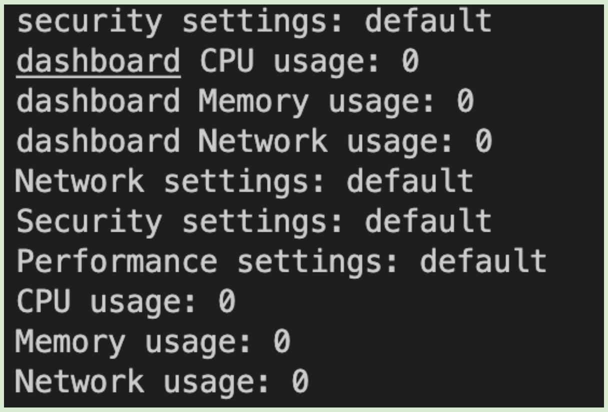
7.3 Technician

We test the Technician section of our software is working properly by

Initially, when the car is not yet running, there should not be any CPU, Memory, or Network usage. All other settings should remain default until they are changed by a user, technician, or driving process. When the technician asks the vehicle to display these values, by calling the methods below, accurate output is displayed.

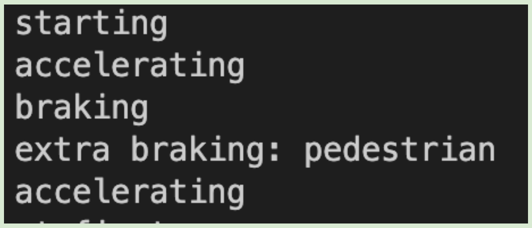


```
t.openSettings();
t.openDashboard();
t.openLogs();
```



```
security settings: default
dashboard CPU usage: 0
dashboard Memory usage: 0
dashboard Network usage: 0
Network settings: default
Security settings: default
Performance settings: default
CPU usage: 0
Memory usage: 0
Network usage: 0
```

Once the vehicle begins moving and recording its actions, these values change and are accurately printed, again,
by the openLogs() method. Below is what the method shows after the car begins driving and encountering obstacles.



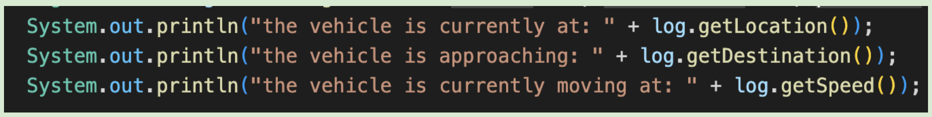
```
starting
accelerating
braking
extra braking: pedestrian
accelerating
```

7.4 Log

When given this constructor for a new LogInterface instance, the software will return the proper values.

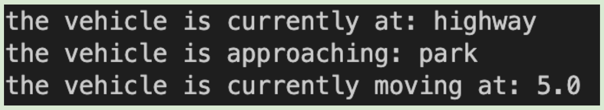
```
LogInterface log = new LogInterface(true, true, 1, 25, 4, 5, 3, "highway", "play",
    "park", 70);
```

This code asks the system to print values that represent the state of certain values in the log, which it accurately does.



```
System.out.println("the vehicle is currently at: " + log.getLocation());
System.out.println("the vehicle is approaching: " + log.getDestination());
System.out.println("the vehicle is currently moving at: " + log.getSpeed());
```

The lines printed to the output indicate that the software is accurately processing the input given to it.

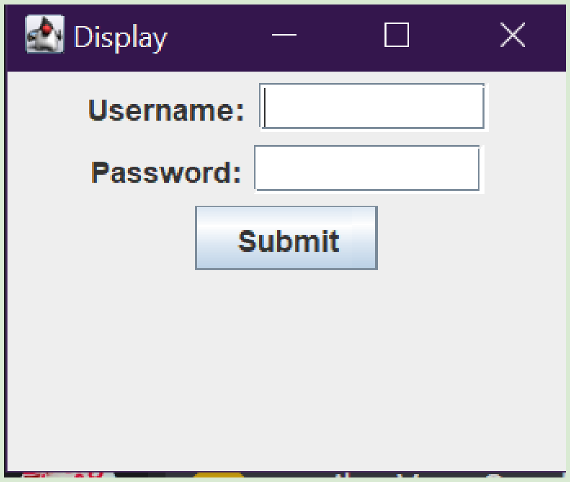


```
the vehicle is currently at: highway
the vehicle is approaching: park
the vehicle is currently moving at: 5.0
```

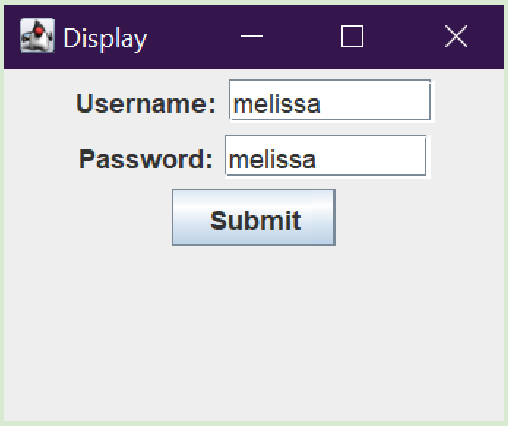
7.6 Display

The DisplayInterface code was tested by compiling and executing the `DisplayInterface` class in a Java development environment. The GUI was then interacted with by filling in the username and password text fields, and clicking on the "Submit" button. This caused the login panel to be hidden and the update panel to be displayed, with the text "Update" in a label. Screenshots were taken at different states of the application.

The screenshot below shows the initial state of the login panel with empty text fields:



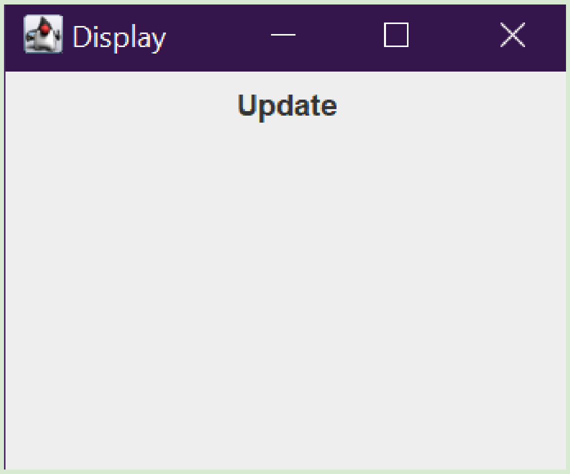
The screenshot below shows the login panel with the text fields filled in:



Username: melissa

Password: melissa

This screenshot shows the update panel with the text "Update":



The screenshots serve as evidence that the code functions as intended, allowing users to enter login information and displaying the update panel after the submit button is clicked. In later phases of development, as we have multiple software iterations, this panel may offer more specific update options and details.