MRKJ

Kailie Jett, Jack Nolan, Renny Victoria, Martyna Zubek
CS 347 Software Development Process - Team 18

"We pledge our Honor that we have abided by the Stevens Honor System"
- KJ, JN, RV, MZ

Table of Contents

Section 1: Introduction

Section 1.1: Purpose

Our team is developing a software solution for the Alset “Hug the Lanes” project. IoT will be used to develop the software for a self-driving assistant car that will be cheaper, more efficient, and safer than the current models being produced. Self-driving cars are the future of ensuring the roads are safe to ultimately decrease the number of accidents and deaths. The software is a mission-critical, real-time embedded system that is responsible for assisting drivers and ensuring their safety on the road. The implementation of an array of features and specifications will allow our software to make real-time decisions and react to road condition changes. Every feature is designed with the driver’s safety in mind, as we want to guarantee the safety of all our customers. Throughout the entirety of every drive, data will be collected, allowing the software to recognize the error and adjust accordingly. This will ensure that the software is updated in regards to the driver's road conditions. Ultimately our goal is to successfully develop and deliver a safe self-driving assistant car.

Our team is relying on IoT in our embedded system because it is among the most advanced architectures for self-driving vehicles. The most crucial components of our embedded system are IoT edge devices. These devices will be placed on the exterior of the vehicle, allowing data collection from the surrounding environment. By utilizing this, many features that are not available on standard cars can be implemented. Features that may be performed include, but are not limited to, parking assistance, an interactive dashboard that is real-time updated, enhanced driving experience when the car is not in autonomous mode, and most importantly safety sensing.

Section 1.2: Software Development Process

The software development process that our team will be utilizing is an iterative waterfall process. As this project is large and complex our team needs a planning process that can encompass the complications within the project. This planning model will allow for consistent software development and for an easy editing process. The iterative approach will allow us to continually improve on previous parts of the project as we move through the development process. Utilizing the iterative waterfall process and communicating regularly with our

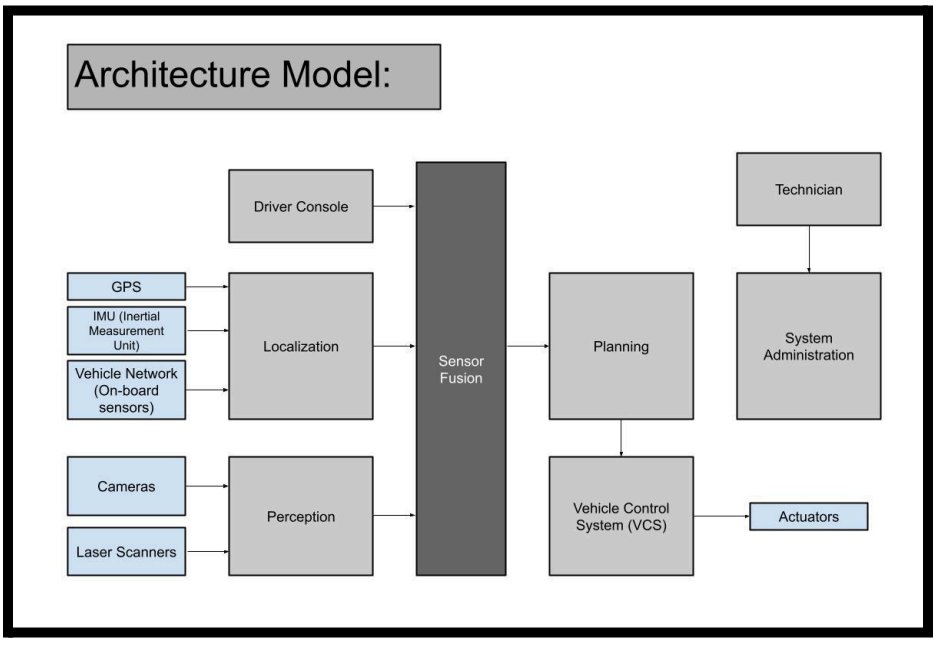
customers, to gain valuable feedback, ensures there will be little to no issues in developing a safe and reliable Alset software.

Section 1.3: Contributions

Our team is composed of members who bring in a variety of personal experiences and knowledge. All members have basic foundational computer science knowledge that has given us a starting basis for the development of self-driving software. All of the programming knowledge and skills will be implemented to develop a quality and safe product. Each member of the team contains diverse strengths and weaknesses that work together efficiently and cohesively.

Section 2: Functional Architecture

The architecture of the car is foundational to the features that allow it to be successfully automated and safe for its user(s). The location sensor, perceptive sensors, sensor fusion module, and the overall Vehicle Control System all allow the car to operate through an exchange of information within the hardware and software. Each feature is used or designed with careful consideration of its role in the overall car, its dependencies on other features, and its effectiveness and efficiency to the system. With the many components of this architecture, many strict requirements will need to be met as well.



Section 2.1: Localization:

Location sensing needs to be extremely accurate to promote safety and is done through a combination of sensors. The GPS communicates via microwaves with global positioning system satellites to receive important geographic information about the car, such as the approximate location, direction, and speed of the car. The GPS directly communicates this information to be localized. Likewise, the inertial measurement unit (IMU) calculates the motion and direction of the car by collecting and processing information about the car's inertia. The vehicle network, which is composed of all the on-board sensors, is the last component of the car that feeds data directly to be localized. The data received from the vehicle network helps to hone in on the location of the car in reference to its local surroundings. This information is all combined and localized creating the approximate location of the vehicle concerning its surroundings. This is then used in combination with the collected information about perception to better assist the driver or help the car autonomously navigate.

Section 2.2: Perception:

Perception is responsible for calculating and processing data about the external environment of the car. The perception component of the self-driving car is directly fed information by two components: cameras and laser scanners. The cameras provide images of the surrounding environment for perception. This information helps perception to detect irregularities and obstructions in the car's environment. Similarly, laser scanners are one of the most crucial components that comprise our self-driving car. We will be utilizing lidar (light detection and ranging) sensors to maximize our car's reliability and safety. Lidar sensors are among the most accurate sensors available today. Lidar sensors work by dispersing light waves out of the exterior of the car in all directions. These waves then reflect off of objects in the car's vicinity and come back to the lidar sensors, which process the information the waves gathered and deliver the data to perception. Lidar sensors are an integral part of our self-driving car because they can generate an accurate map of the entire three-dimensional surroundings of the car.

Section 2.3: Sensor Fusion:

Sensor fusion is the module responsible for the task of processing the large amount of information provided by localization, perception, and the driver. As previously mentioned, localization is responsible for determining the location of the car, while perception is responsible for gathering information about the car's surrounding environment. Sensor fusion is a critical module of the functional architecture of the car because it can combine the data provided by localization and perception to provide a more accurate and complete image of the car's environment. Sensor fusion helps correct potential errors in independent sensors by comparing the data provided by the said sensor to that of the data provided by the other sensors in the car. Sensor fusion increases the reliability and safety of the car by ensuring that potential errors in individual sensors do not affect the car's performance, as sensor fusion can override and correct these errors by using the information given by the other sensors and components that feed information to localization and perception.

Section 2.4: Planning and Communication:

The information that has been collected via the sensors and then fused still needs to be organized to be used. This happens during the planning phase of the architecture. During planning, all data from both the sensors and any data received from the driver is sorted to be developed into a sequence of instructions to be provided to the Vehicle control system. There will be two separate algorithms within the planning step to ensure safety and the priority of the driver's instructions. The initial algorithm will take and process all the data from the sensors to create instructions based on that data. For example, the driver turning on the windshield wipers would result in a windshield wipers on command, or the vehicle sensing a stop sign would result in a stop command. The secondary algorithm would then take all of this parsed data and use it to create a sequence of instructions to send to the Vehicle Control System. To create the best order within the instructions the algorithm will prioritize safety instructions and those based on what the driver has requested. These two algorithms will be required to operate at a very high speed to make sure that the car can keep up with the real-time decisions that are being made. There should also be a very high accuracy rate with the identification of safety issues by the initial algorithm. After all the information has been gathered, sorted, and prioritized it will then be fed to the Vehicle Control System for execution.

Section 2.5: Vehicle Control System:

The Vehicle Control System has a primary function of communicating the sets of instructions received from the planning stage to the different actuators of the vehicle. It also has to make real-time adjustments of those commands to the actuators to ensure that all commands are executed without issue or additional danger to the passengers. For example, the car would receive an instruction to stop, it can not simply halt but instead has to slowly apply braking pressure depending on exactly when the stop needs to occur. This will be a very precision-based system as any errors could lead to the car and passengers being put in harm's way. To properly complete all commands safely, the manufacturers guarantee a very low rate of fault within the hardware which will then make it much easier for us to guarantee a low rate of fault within the control system. Any errors or failures are then stored and reported to the system administration module.

Section 2.6: System Administration:

Technicians for Alset are required to be able to access the logs and general information about the vehicle to run diagnostics and updates which are provided through the system administration module. To properly maintain and protect the data of the vehicles, there will be a secure way to access the information that is password protected. By utilizing a cloud provider, the system administration will take place through data from the car being uploaded and then analyzed for errors or issues. With many cloud providers, interfaces are in place to monitor devices exactly as it would be necessary for the autonomous vehicle. The easy access to the data would also make it so that drivers who have questions about their vehicle would be able to request information and get it back within a short timeframe. The module itself will function by gathering the data collected from the sensors and parsing it to record the logistics of the vehicle. Plus it will gather error information that occurs either in the planning stage or issues that arise from the Vehicle Control System. It will then take all this information and assert if there are any concerning trends via an algorithm. The System Administration Module helps to maintain a safe vehicle for the driver while keeping the software at the most current edition.

Section 2.7: Conclusion

The components of this car system work to collectively ensure proper location sensing, environment perception, sensor fusion, and Vehicle Control System along with the various other aspects of the system. The location sensing, through IMU and GPS, aims to provide reliable navigation. Environment perception allows for safe travel through various environments, and sensor fusion takes the data collected and organizes it to create an image of the surroundings of the car. Organized data is transformed into consumable sequences of instruction that are funneled to the Vehicle Control System. System administration maintains the system and updates it through cloud-based analysis. These various components work harmoniously to optimize the experience and interactions with the car while maintaining safety.

Section 3: Requirements

Section 3.1: Functional Requirements:

To make software that is highly capable of controlling a car, there needs to be precise requirements that are met. The requirements that contribute to making the car autonomous or automatically assisting the driver are considered Functional Requirements. These requirements outline how the car will complete tasks by outlining firstly what conditions need to be met for the car to engage in the task such as object avoidance, turning on the headlights, etc. Then how said behavior will be executed through the architecture from the previous section. These requirements will outline how the software will be structured for the autonomous vehicle to be as sophisticated and simplistic as possible.

Section 3.1.1: Object Avoidance:

- Precondition- The car must be on, there must be a driver, and the location sensors must detect an object that is distance d, 300 feet or less away.
- Postcondition- The car smoothly and safely avoids the object.
- Requirements-
    - The distance d, speed s of the vehicle, location l, height h, and width w of the object are then sent from the sensors to sensor fusion to be parsed into a set of instructions.

- Depending on the location of the object either to the left, right, or taking up the lane the planning module will develop a series of steps to avoid the object
    - If the object is moving, the car will proceed to halt
    - If the object is obstructing the whole lane and there is no lane to change into, the car will proceed to halt
    - If the object is obstructing the whole lane and there is a lane to change into, the car will switch lanes
    - If the object is on the left, and the object does not take up the full lane the car will steer slightly toward the right and slow down for safety practices
    - If the object is on the right, and the object does not take up the full lane the car will steer slightly toward the left and slow down for safety practices
- The driver is also notified via sound that the object is near.
- This set of instructions is then sent to planning to create the most optimized method of execution which is then received by the VCS(Vehicle Control System).
- By delivering the correct instructions to the vehicle actuators, the VCS allows the car to avoid the object.
- All the received data is then passed on to the System administrator.

Section 3.1.2: Self-Driving System Termination:

- Precondition- The car must be on, there must be a driver, and the car must be in motion through the self-driving software.
- Postcondition- The self-driving feature stops and the car switches to control through driver maneuvering.
- Requirements- The system termination physicality is translated into instructions that cause the self-driving software to safely terminate and transition to manual driving, while

maintaining other safety features in case of ineffective transition control to manual driving.

- The button is manually selected by the driver to switch modes from self-driving to driver-based navigation.
- This set of instructions is then sent to planning to be executed, which is then received by the VCS (Vehicle Control System).
- Safety checks are run to make sure the vehicle is in a state that can allow for change in modes.
- The driver is notified of the confirmation of switching modes.
- The system is switched to manual-controlled driving.
- All the received data is then passed on to the System administrator.

Section 3.1.3: Windshield Wipers:

- Precondition- The car must be on, there must be a driver, if sensors detect 70 % of moisture on the windshield under above-freezing conditions, then it will automatically activate until the windshield is cleared.
- Postcondition- The windshield is cleared of moisture and keeps wiping until the user manually shuts off the wipers or the automatic wipers detect that the moisture is gone or minimal.
- Requirements-
    - IoT software activates the windshield sensors.
    - Sensor fusion reads the information sent in from the sensors.
    - If the sensors detect moisture of greater than 70 % on the vehicle windshield, the module will send information to the Vehicle Control System.
    - The Vehicle Control System initiates the activation of the wipers.
    - Once the sensors detect moisture as below the wiping threshold, the wipers should turn off, or if the wipers are manually turned off by the driver.
    - All data from each step is recorded in the System Administration Log File.

Section 3.1.4: Car Alarm:

- Precondition- The panic/alarm button is pressed on the key fob, or the car is parked and one of the following must be met: The car key must be located outside of the vehicle distance d >= 1 feet and entry is attempted, or the theft detection sensors are triggered
- Postcondition- The car alarm sounds quickly and accurately and turns off when it is prompted to.
- Requirements-
    - The sensors collect the distance to the key d, the panic button's state either off or on, the location and repetitions of impact to the car, the location of the whole vehicle with respect to the ground, and the car's gear position.
    - This data is sent to Sensor Fusion to be parsed into instructions and is then sent to the Planning Module.
        - If the Panic/Alarm button is pressed, the Planning Module sends to the VCS to sound the alarm.
        - If the car is in the park:
            - If the distance of the key >= 1 and the door sensor is triggered, the Planning Module sends it to the VCS to sound the alarm.
            - If at least 3 repeated impacts to the same window or door within a minute, the Planning Module sends to the VCS to sound the alarm.
            - The car is lifted from the ground and there is no key in the vehicle, the Planning Module commands the VCS to sound the alarm.
    - The VCS sends the command to turn on the alarm actuators.
    - The alarm sounds until the key fob is pressed or off is selected via the Driver Interface.
    - All system data is reported to the System Administration Log File

Section 3.1.5: Blind Spot Detection:

- Precondition- The car must be on and the driver must be behind the wheel. The car must be in gear position drive or reverse.

- Postcondition- Detection of an object by the blind spot sensors should activate and alert the driver.
- Requirements-
    - IoT software activates the blind spot sensors to be on.
    - Sensor Fusion reads the information sent in from the sensors.
    - If blind spot sensors detect an object within 20 ft from the rear of the vehicle, the Planning Module will send information to the Vehicle Control System.
    - The Vehicle Control System initiates the visual representation of the blind spot to the driver.
    - Once the vehicle is placed into park, the sensors should turn off.
    - All data from each step is recorded in the System Administration Log File.

Section 3.1.6: Automated Parking:

- Precondition- The car must be on and the driver must be behind the wheel. The car must be in gear position drive. Spot nearby within traffic regulations is detected with a area >= c car's area
- Postcondition- Car is parked safely and quickly.
- Requirements-
    - The car is going under 15 mph in drive.
    - Driver activates automated parking
    - Sensor Fusion reads the information received from the object detection sensor and sends it to the Planning Module.
    - The Planning Module decides whether or not to activate parking assistance based on the information from the Sensor Fusion.
        - If area a >= car's area c, a message is displayed asking the driver if he wants to enter parking assistance.
            - If the driver responds yes, then the vehicle will generate instructions to automatically steer, brake, and shift gears.
            - If the driver's response is no, then the parking assistance is turned off.
        - If it is denied, an error message is displayed explaining why.

    - The VCS is then given the instructions generated in planning to send to the Vehicle actuators
    - All data from each step is recorded in the System Administration Log File.

Section 3.1.7: Automatic Headlights:

- Precondition- The car is on, and there is a driver in the driver's seat. At least one of the following is met: there is less than 50 lux of ambient light, or the windshield wipers are on.
- Postcondition- The correct setting of headlights is turned on for the external condition of the car.
- Requirements-
    - The amount of light l, and the moisture content of the windshield m, the distance of the nearest moving objects d, the temperature t are detected by the sensors and sent to Sensor Fusion to be parsed.
    - The Sensor Fusion Module then sends the Planning Module the parsed data for it to decide what headlight mode will be activated.
        - If there is low light (l<50) and no moisture, planning will dictate the normal headlights engage.
        - If there is moisture > 70 and low light (l<50) and temperature t >32 degrees F, the fog lights will be engaged.
        - If there is no object within a readable distance of the sensors and there is no moisture but low light, the high beams or brights will be engaged.
        - If the driver signals for a specific light mode that mode will engage overriding the automatic headlights (ex. Flashing brights for dangers etc).
    - The decision from planning is sent to the VCS which then sends the decision to the headlight actuators.
    - All data from each step is recorded in the System Administration Log File.

Section 3.1.8: Traffic Sign Detection:

- Precondition- The car must be on and the driver must be behind the wheel. The car must be in gear position drive.
- Postcondition- Forwards-facing cameras that are on the windshield are activated
- Requirements-
    - The IoT system must detect different types of signs like traffic lights, yield signs, stop signs, do not enter signs, and speed limit signs.
    - Once the sign is detected, Sensor Fusion determines which part of the Planning Module to enter depending on the sign.
    - The sign will be displayed for the driver's visual.
    - If the driver is in Cruise Control, the driver's path will be altered accordingly.
    - All data from each step is recorded in the System Administration Log File.

Section 3.1.9: Cruise Control:

- Precondition- The car must be on. There must be a driver present. The car's speed must be above 50 mph. The driver must activate cruise control via the car's User Interface.
- Postcondition- The car can maintain a safe speed, acceleration, and deceleration pattern based on traffic conditions.
- Requirements-
    - Sensors read in the command to turn on cruise control, the distance between the vehicle and all nearby objects, and the speed of the car.
    - The information from the sensors is sent to Sensor Fusion to be parsed into instructions and data.
    - These instructions and data are sent to planning to decide if cruise control can be engaged:
        - Cruise control will not engage and there will be a warning message that pops up on the driver's dashboard:
            - If the car's speed is s <50 mph.
            - If there are t > 3 objects, less than 200 ft away.
        - Otherwise, cruise control will be engaged

- If an object < 200 ft in front of the car is slowing down an algorithm will calculate the slowdown necessary to match the speed of said vehicle until its distance > 200ft
    - The car will also detect if any other vehicles attempt to merge via the front sensors, and will adjust speed according to the previous algorithm.

- Cruise control will stay engaged until:
    - The driver turns it off via the interface.
    - The car's speed is s <50 mph or there are t > 3 objects, less than 200 ft away.
- All data from each step is recorded in the System Administration Log File.

Section 3.1.10: Assisted reversing:

- Precondition- The car should be on, there must be a driver present at the wheel, the car must be set to reverse.
- Postcondition- The car successfully reverses until the gear is changed by the driver or by the self-driving software.
- Requirements-
    - The gear shift must be in Reverse, either manually or automatically.
    - Instructions are sent to activate the following prerequisites for assisted reversing:
        - The rear-view camera.
        - Sensors in the anterior of the car.
        - Blind spot detection (See 3.1.5).
        - No objects detected
    - If obstacles are detected while reversing, the car should halt until the sensors indicate that reversal can begin again.
    - The car reverses with assistance until it is stopped by the driver or the self-driving software.

Section 3.1.12: Entering Keyless:

- Precondition- The car is stationary and off.
- Postcondition- Detection of the key whilst touching the door should be activated and the doors should be unlocked.
- Requirements-
    - When the car is turned off, the door sensor should be turned off.
    - Once the key is detected within a foot from the door, the IoT software should initiate door unlocking.
        - If the door handle is touched, the door should open:
            - If the driver touches the driver's door then only the driver's door should unlock.
            - If the driver touches any other door, then all doors should unlock.
    - The Vehicle Control System should log that the car was opened.

Section 3.2: Nonfunctional Requirements:

In addition to the requirements that benefit the functionality of the vehicle, there are requirements that outline the capabilities and constraints that the vehicle must uphold. These requirements are considered nonfunctional and include the requirements for Reliability, Performance, Security, and all Software Updates. These requirements will help ensure that the vehicle can last for years to come, both benefiting the customers and Alset by maintaining high standards for software engineering.

3.2.1: Reliability:

The systems within our car must maintain strict levels of reliability to ensure the safety of our customers.

Requirements:

- The car will not fail more than once in 40 years of operation.
- The system hardware must have a five-nine reliability, meaning that the hardware will have a 99.999% effectiveness.
- The system software must have a five-nine reliability.
- The Sensor Fusion Module must be able to detect and correct all erroneous information that it receives from the Localization Module and Perception Module.

- All of the car's sensors will collect and send data to either the Localization Module or Perception Module every nanosecond.

3.2.2: Performance:

It is imperative that our car maintains strict performance requirements. The car must meet the following performance requirements.

Requirements:

- The duration that it will take for information from the sensors to reach the Vehicle Control System must be no greater than five hundred milliseconds.
- The Localization Module must be able to concurrently process information from all of its corresponding sensors.
- The Perception Module must be able to concurrently process information from all of its corresponding sensors.
- The Sensor Fusion Module must be able to concurrently process and correct mistakes in the information received from the Localization and Perception Modules.
- The Localization Module must be able to concurrently process information from all of its corresponding sensors.

Section 3.2.3: Security:

The autonomous vehicle software being manufactured and updated by our team will have many security policies in place to better protect the user and vehicle.

Requirements:

- Data stored in the log files and System Administration Module will only be accessible by technicians and ourselves.
- The software itself will only be accessible to the team.
    - These will be implemented by using two separate password and username interfaces.
- The car will only be able to start motion when the key is detected within the vehicle itself.

- The software itself must have updates to improve security as the car is used more and if any potential threats are determined.
    - See the next section for more details

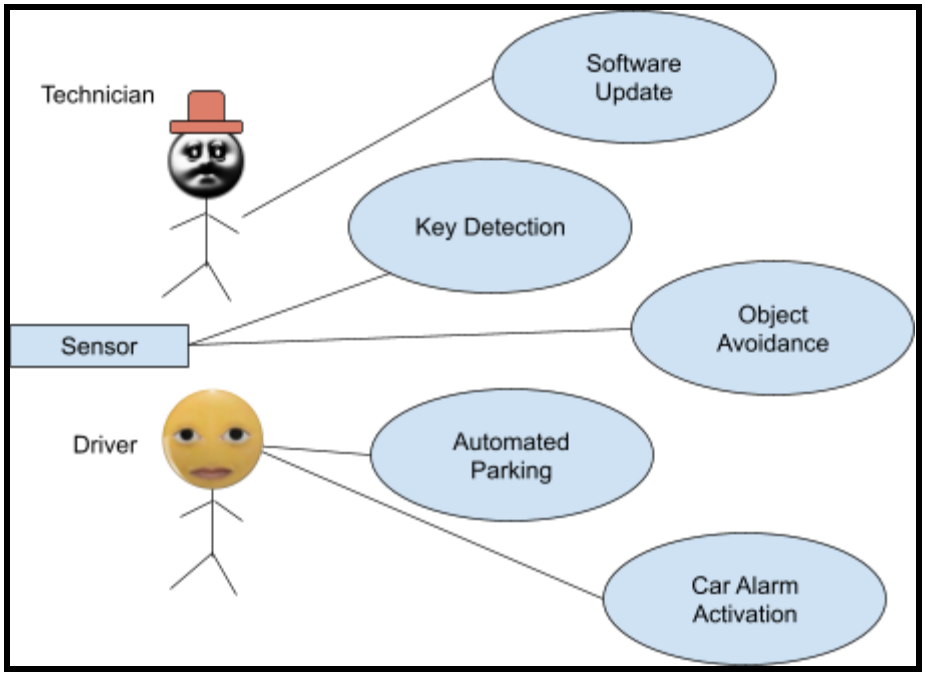
Section 3.2.4: Software Updates:

To ensure safety and security in the self-driving vehicle, routine software updates will be provided to users.

- Updates will provide improvement to existing features, fix bugs, provide security updates, and/or new functionalities to the vehicle software.
- Updates will be sent through the secure cloud-based system. Users will be notified of existing and pending updates, and will be given options to install them (if the update does not have mandatory/security-dire features).
- When a software update is available, and the driver is detected in the vehicle, the driver will be given the opportunity to begin installing the update.
    - To install the update, the car must remain parked and on during the entirety of the installation process to ensure the software is installed correctly and the driver's safety is guaranteed.
- All updates will be monitored and saved under System Administration to have a log of changes and make sure that the system is up-to-date.

Section 4: Requirement Modeling

4.1 Use Cases:



Use Case 4.1.1 Object avoidance is activated by a sensor

Precondition- The car must be on, there must be a driver, and the location sensors must detect an object that is distance d, 300 feet or less away.

Postcondition- The car smoothly and safely avoids the object.

Trigger: Sensor detects object less than 300 feet away

1. The sensor detects an object less than 300 feet away
2. The information about the object, and its location is parsed in sensor fusion
3. The car speed and the object information are sent to Planning
4. Planning determines if the object needs to be avoided
5. If an object needs to be avoided, Planning sends the request to VCS to maneuver around the object
6. Exception: If the object is too small to necessitate avoidance then Planning does not send anything to VCS
7. Planning sends to VCS to issue object-detected sound
8. Planning sends all related data to the System Admin for logging
9. VCS sends all related data to the System Admin for logging.
10. Other Possible Exceptions to Object Avoidance:
11. Driver-activated brake

Use Case 4.1.2 Key Detection by a sensor

Precondition- Car is off and stationary. A driver is present and the location sensor must detect a key that is distance d, 2 feet or less away from the car.

Postcondition- The car opens and turns on

Trigger: Sensor detects a key less than 2 feet away

1. The sensor detects a key less than 2 feet away
2. The information about the key, and its location is parsed in sensor fusion
3. The door locks and key information is sent to Planning
4. Planning determines if the key is the appropriate one
5. Planning decides to allow the car to unlock
6. Once the car is unlocked and the driver is present inside the car premises, the car has the ability to be turned on
7. If the driver wants to turn the car on, the driver clicks the on button and Planning sends the request to VCS to turn on the car
8. Exception: If the key is no longer in the 2 feet radius of the car then Planning sends to VCS that the key is missing
    a. Planning sends to VCS to issue key-missing sound
    b. Planning sends all related data to the System Admin for logging
    c. VCS sends all related data to the System Admin for logging.
9. Planning sends to VCS to turn the car on
10. Planning sends all related data to the System Admin for logging
11. VCS sends all related data to the System Admin for logging.

Use Case 4.1.3 Software update implementation

Precondition - The Car is off and parked

Postcondition - The Car's software has been updated

Trigger: Technician sends a software update to the Car

1. The Technician accesses System Administration using their login username and password

    a. If the username and password pair exists, then the Technician is granted access to the System Administration
    b. Otherwise, the Technician is denied access to the System Administration
2. The Technician accesses the Car's data files using the Car's ID
3. The Technician calls the getCarData() function to find out what software update(s) to initiate
    a. If there are multiple software updates needed, the Technician will queue them in sequential order
    b. Otherwise, the Technician initiates the sole software update
4. The Technician sends "initiate update" command to the System Administration
5. System Administration initiates the software update process
6. System Administration disables Car driving capabilities via disableCarDriving()
    a. If the Driver turns on the Car while the software is updating, then System Administration sends the "display update in progress" command to the Driver's Console
7. System Administration sends the "update" command and new software code to every Module
8. If an individual Module update is successful it sends "true" to System Administration, otherwise, it sends "false"
9. Once the update has been completed, System Administration logs the data received from every Module
10. The Technician updates the log files in System Administration to indicate that the software update was completed
11. System Administration enables Car driving capabilities via enableCarDriving()

<u>Use Case 4.1.4 Car alarm activation by hitting key fob</u>

Precondition - The car is either on or off, the car must be parked/stationary, and the key fob must be within 100 feet of the car.

Postcondition - The car alarm sounds and keeps sounding until turned off

Trigger - Panic button is hit on the key fob within 100ft radius

1. The driver hits the panic button on the key fob

2. The signal is received by the system to be processed by the Planning Module
3. The Planning Module sends to the VCS to activate the alarm
4. VCS sends instructions to turn on the alarm actuators
5. The alarm audibly sounds until the key fob is pressed again to deactivate the alarm
6. Exception: If the car battery dies, then alarm will not continue to sound
7. Planning sends all related data to the System Admin for logging
8. VCS sends all related data to the System Admin for logging.

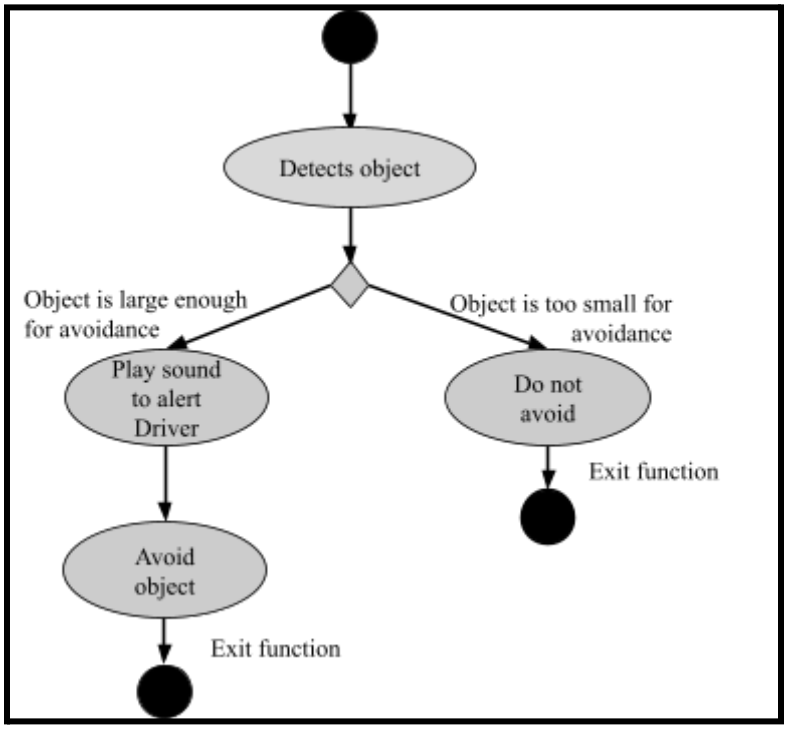
Use Case 4.1.5 Automated Parking Activation by Driver

Precondition- The car must be on and the driver must be behind the wheel. The car must be in gear position drive. Spot nearby within traffic regulations is detected with a area >= c car's area
Postcondition- Car is parked safely and quickly.
Trigger- Driver activates automated parking on Driver Interface

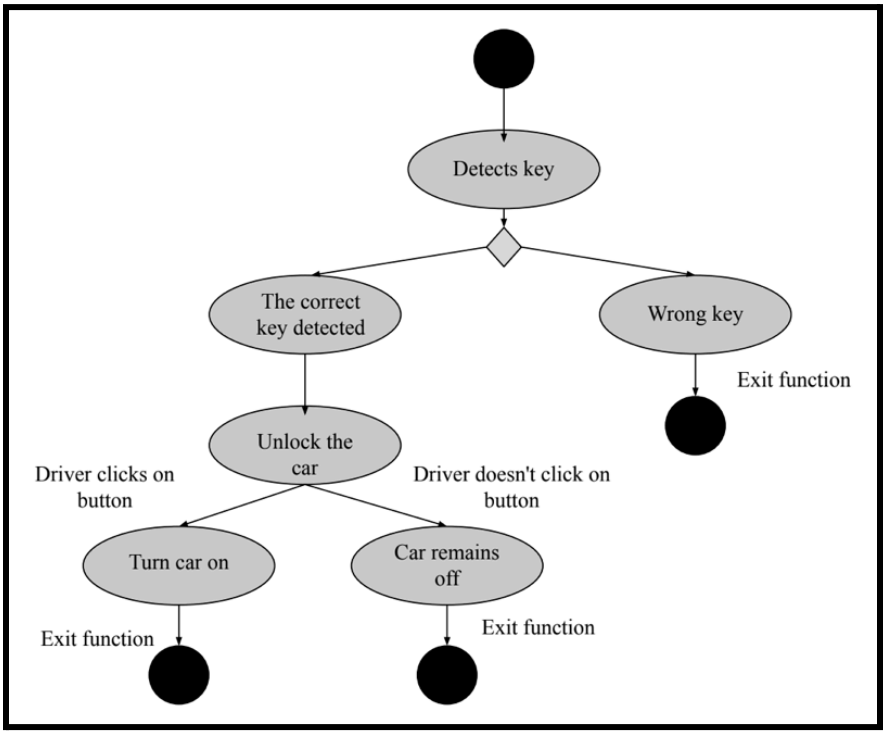
1. Driver activates automated parking on Driver Interface
2. Information about nearby objects is parsed and sent to Sensor Fusion
3. Planning decides if the space near the car is large enough for the car to park
4. If the space is large enough planning sends a set of instructions to the driver interface requesting permission to park
    a. The parking space must be at least 9 ft by 18ft for automated parking to initiate
5. If the driver accepts, Planning generates a set of instructions to park the car which is sent to the VCS
6. Exception: VCS does nothing if there is not enough space to park the car
7. Exception: VCS does nothing if the driver rejects the request to park at the specified location
8. Planning sends all related data to the System Admin for logging
9. VCS sends all related data to the System Admin for logging.
10. Other possible Exceptions:
11. Driver accelerates to > 15 mph

4.2 Activity Diagrams:

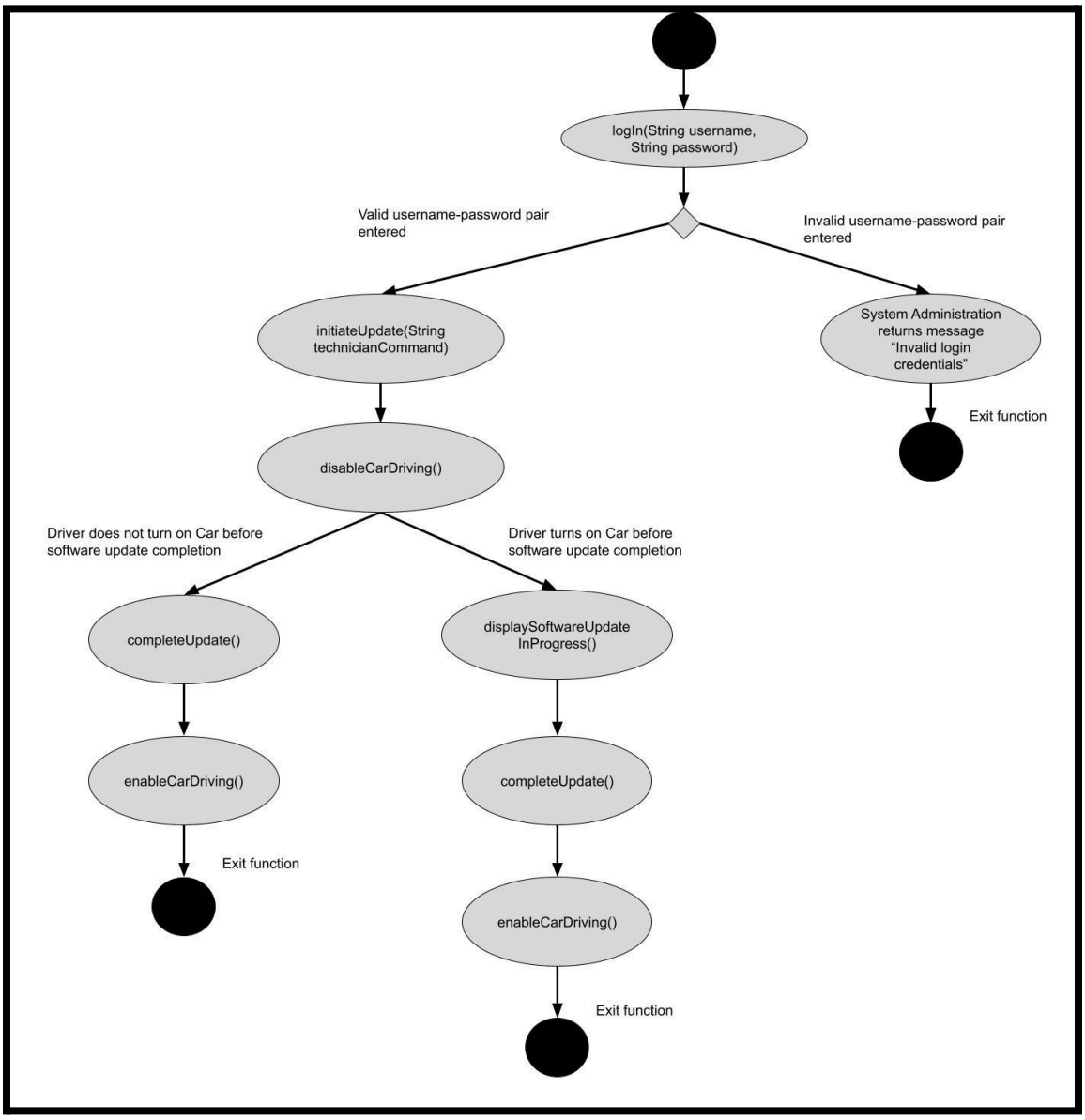
4.2.1 Object avoidance is activated by a sensor



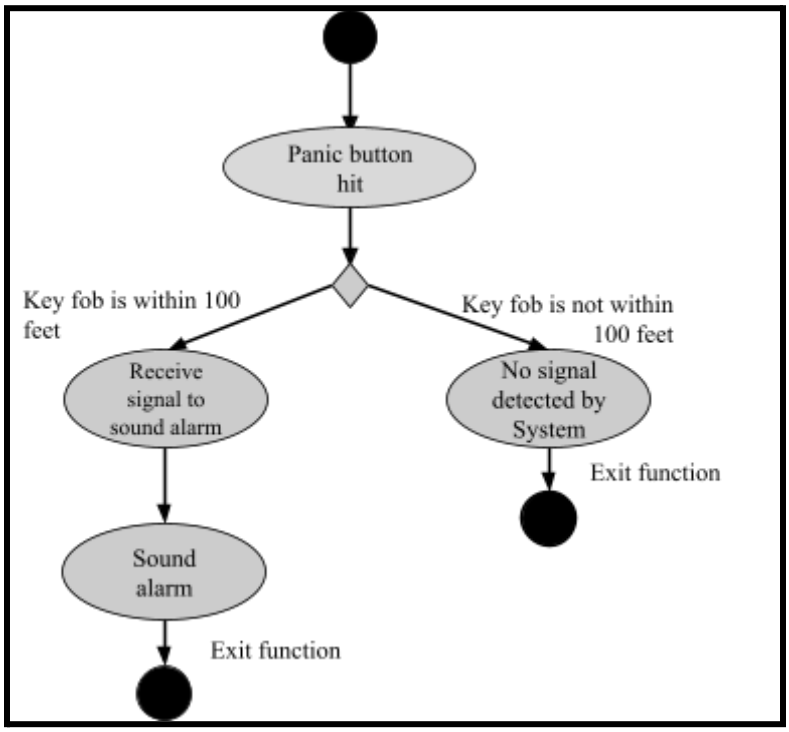
4.2.2 Key Detection is activated by a sensor



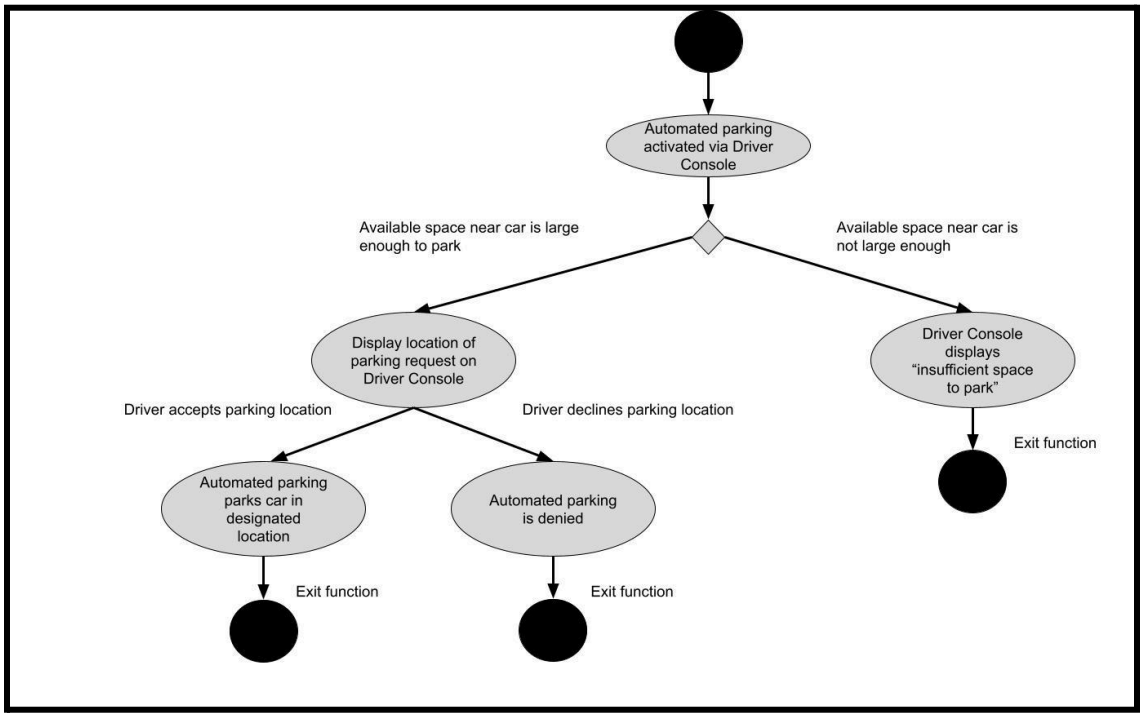
4.2.3 Software update implementation



4.2.4 Car alarm activation by key fob

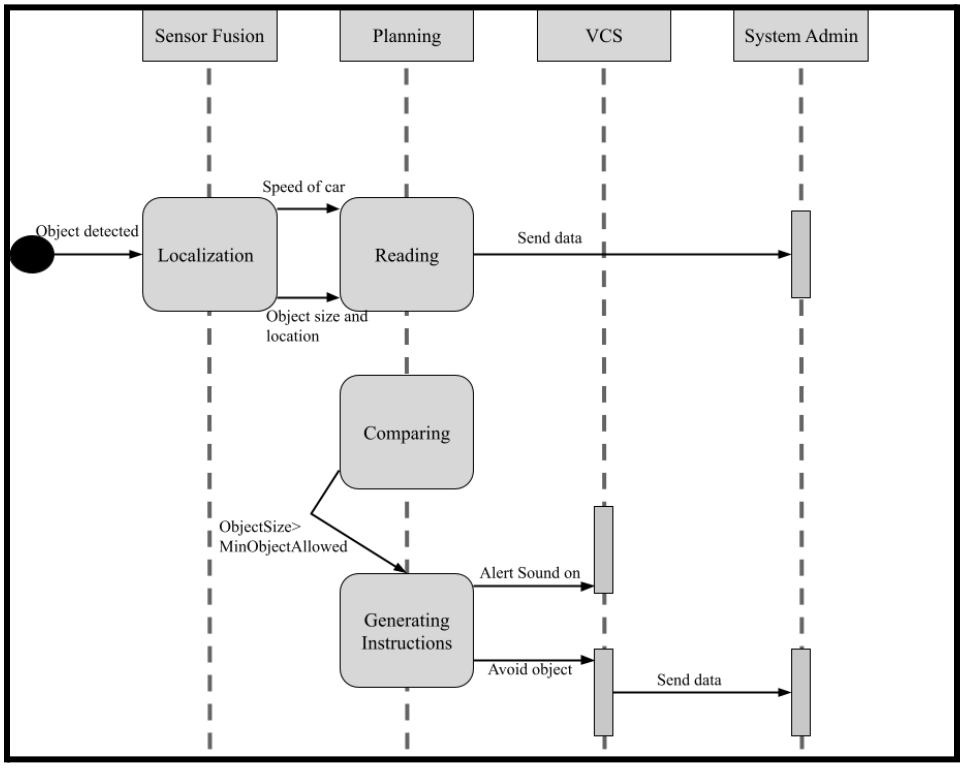


4.2.5 Automated Parking Activation by Driver

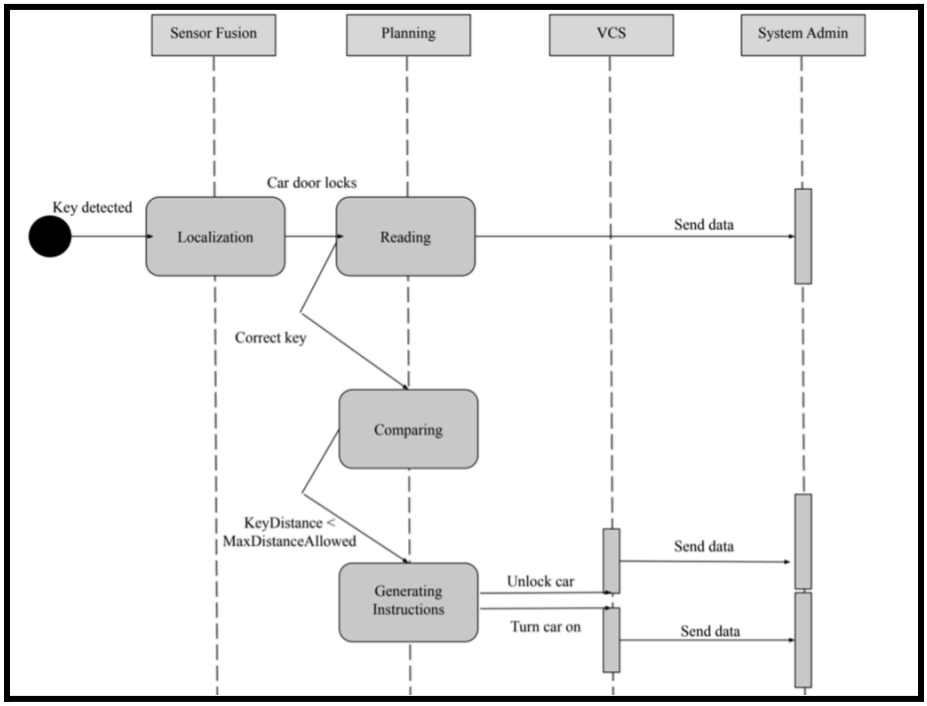


4.3 Sequence Diagrams:

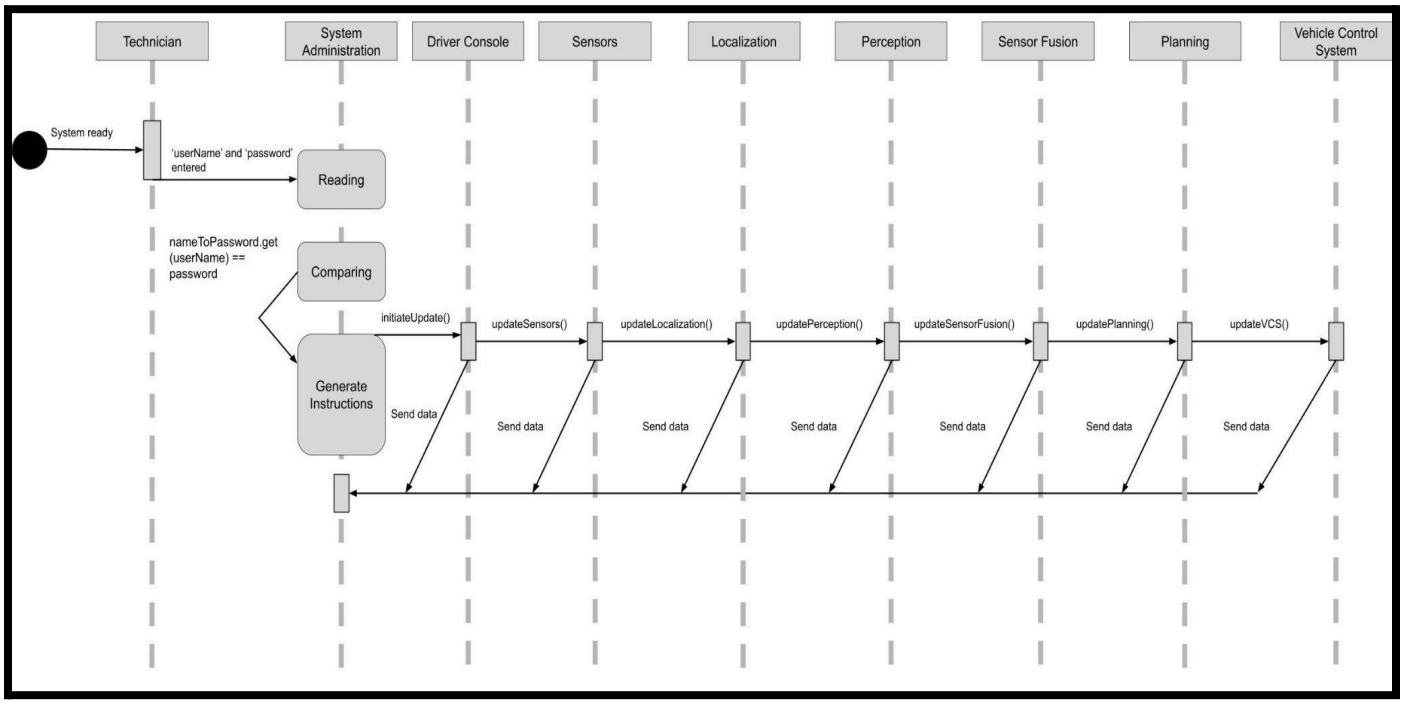
4.3.1 Object avoidance is activated by a sensor



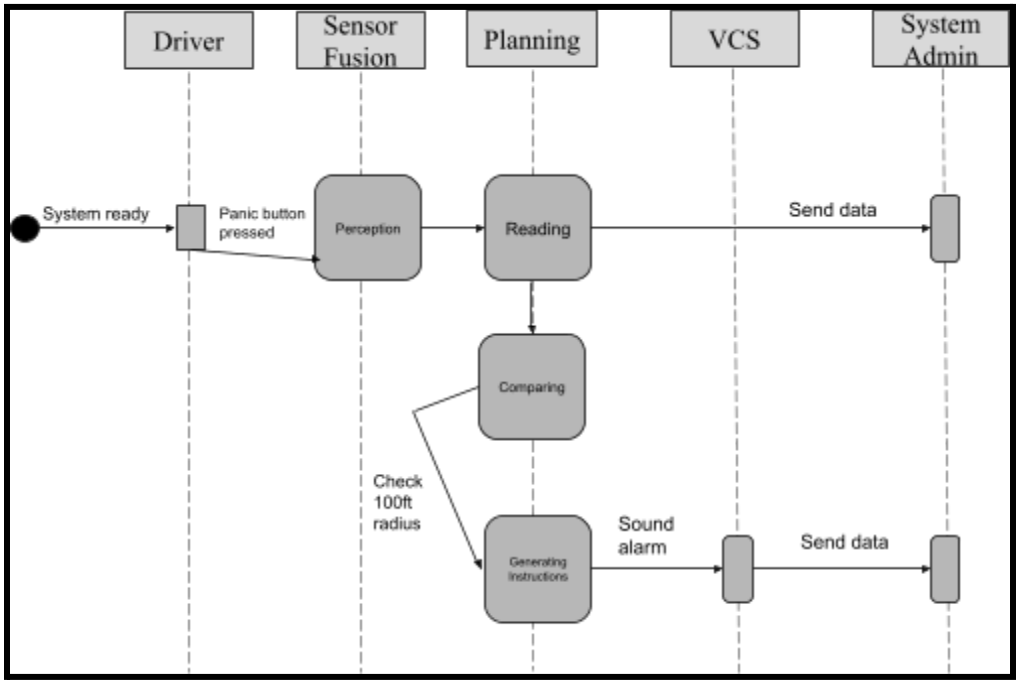
4.3.2 Key Detection is activated by a sensor



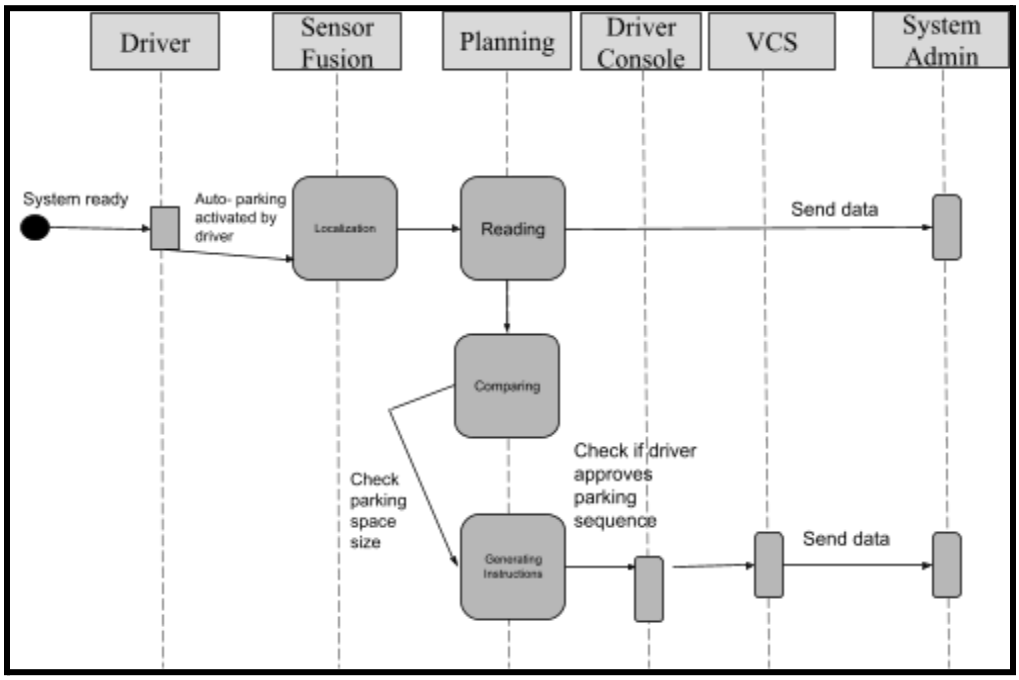
4.3.3 Software update implementation



4.3.4 Car alarm activation by key fob



4.3.5 Automated Parking Activation by Driver

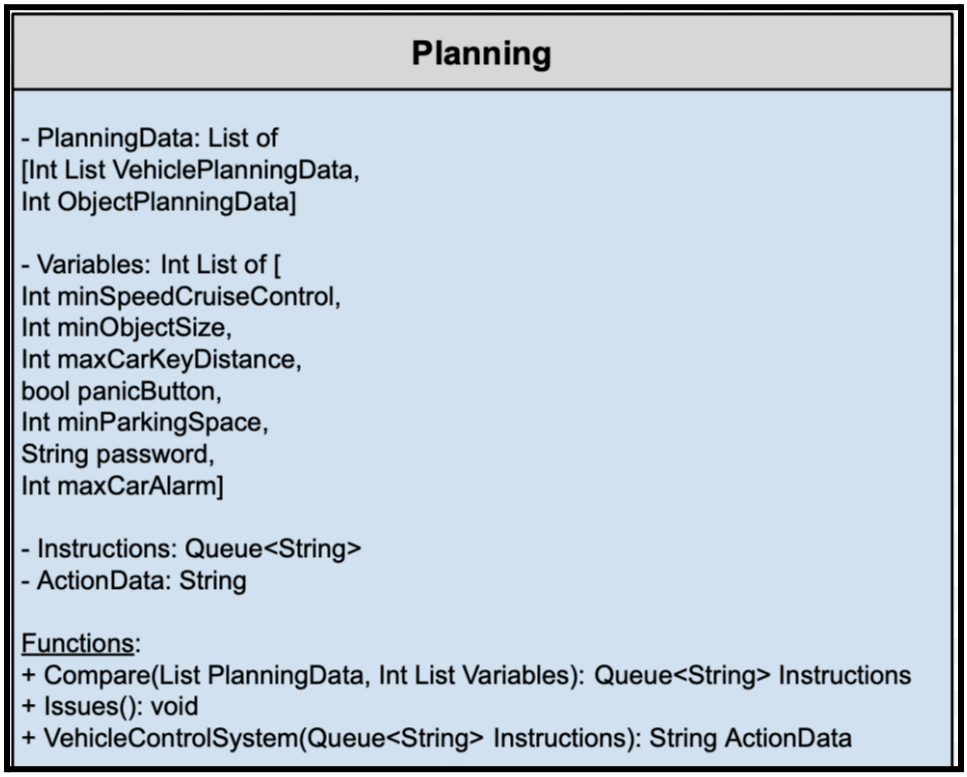


4.4: Classes:

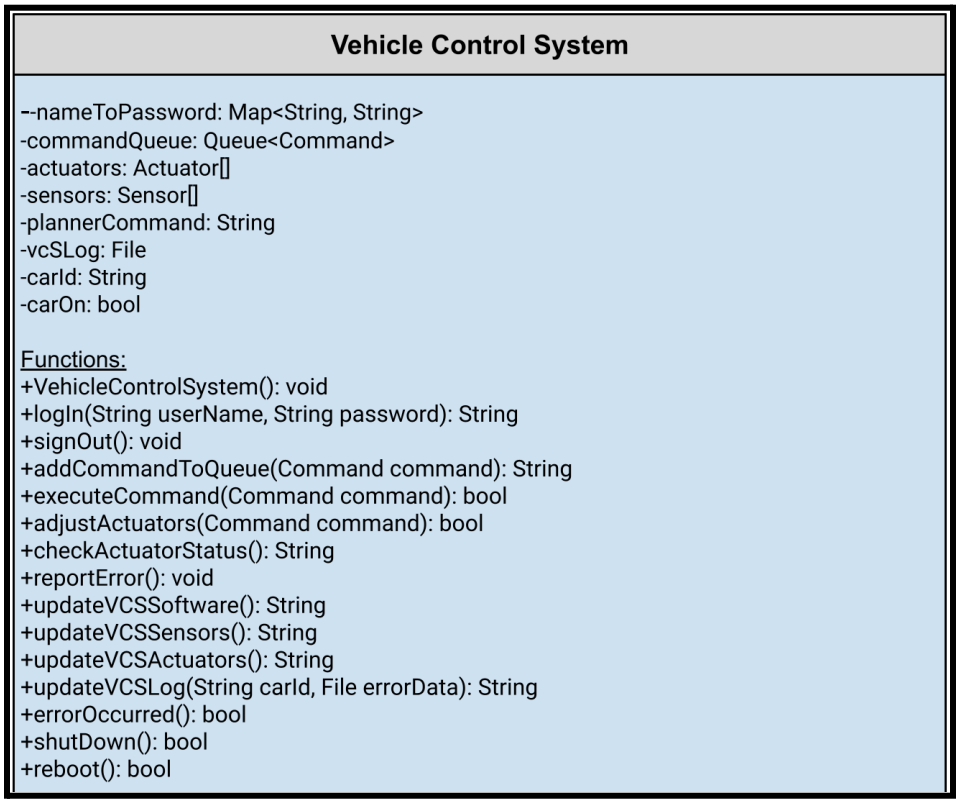
Sensor Fusion-

Sensor Fusion
-SensorData: Int List -PlanningData: List of [-VehiclePlanningData: Int List of [Int speed, Int acceleration, Int Direction, Int Size, Bool DriverAlarm, Bool DriverUnlock], -ObjectPlanningData: Int List of [Int ObjectLocation, Int ObjectSize, Int ObjectSpeed, Int ObjectAcceleration, Int HumidityLevel, Int LightLevel]] -LogFile: File Functions: +SensorFusion(Int List SensorData): List PlanningData +Localization(Int List VehicleData): Int List VehiclePlanningData +Perception(Int List ObjectData): Int List ObjectPlanningData +SplitSensorInfo(Int List SensorInfo): Int List ObjectData, Int List VehicleData +SensorFusionLog(Int List SensorData): File LogFile

Planning-



Vehicle Control System-



System Admin-

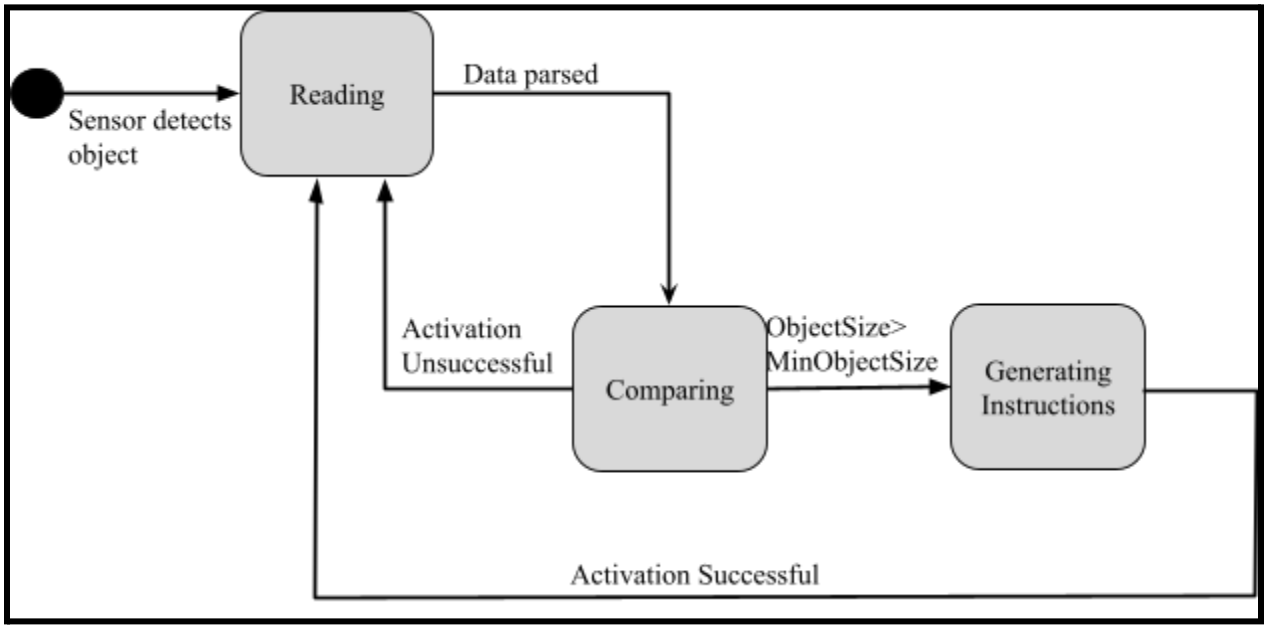
System Administration

-nameToPassword: Map<String, String>
-updateQueue: Queue<Update>
-carDataFile: File
-technicianUsername: String
-technicianCommand: String
-technicianIsLoggedIn: bool
-carId: String
-carOn: bool

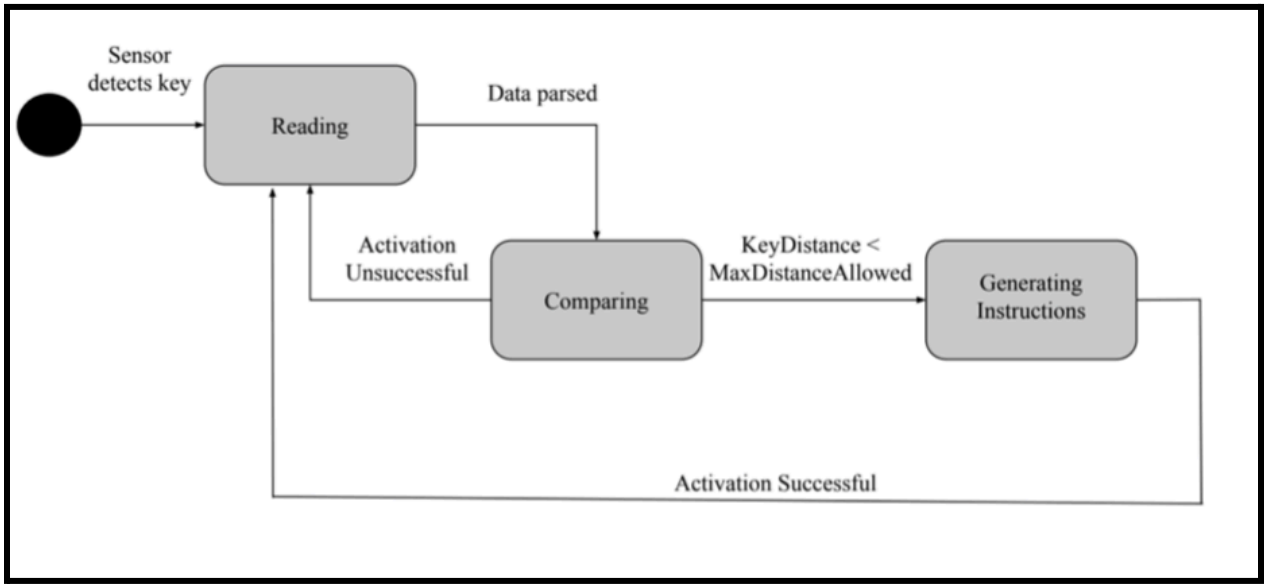
Functions:
+SystemAdministration(): void
+logIn(String userName, String password): String
+signOut(): void
+addUpdateToQueue(Update update): String
+getCarData(): File
+initiateUpdate(String technicianCommand): String
-completeUpdate(): bool
-displaySoftwareUpdateInProgress(): String
-updateDriverConsole(): bool
-updateSensors(): bool
-updateLocalization(): bool
-updatePerception(): bool
-updateSensorFusion(): bool
-updatePlanning(): bool
-updateVCS(): bool
-updateCarLogFile(String carId, File updatedData): String
-updateComplete(): bool
-disableCarDriving(): String
-enableCarDriving(): String

4.5: State Diagrams:

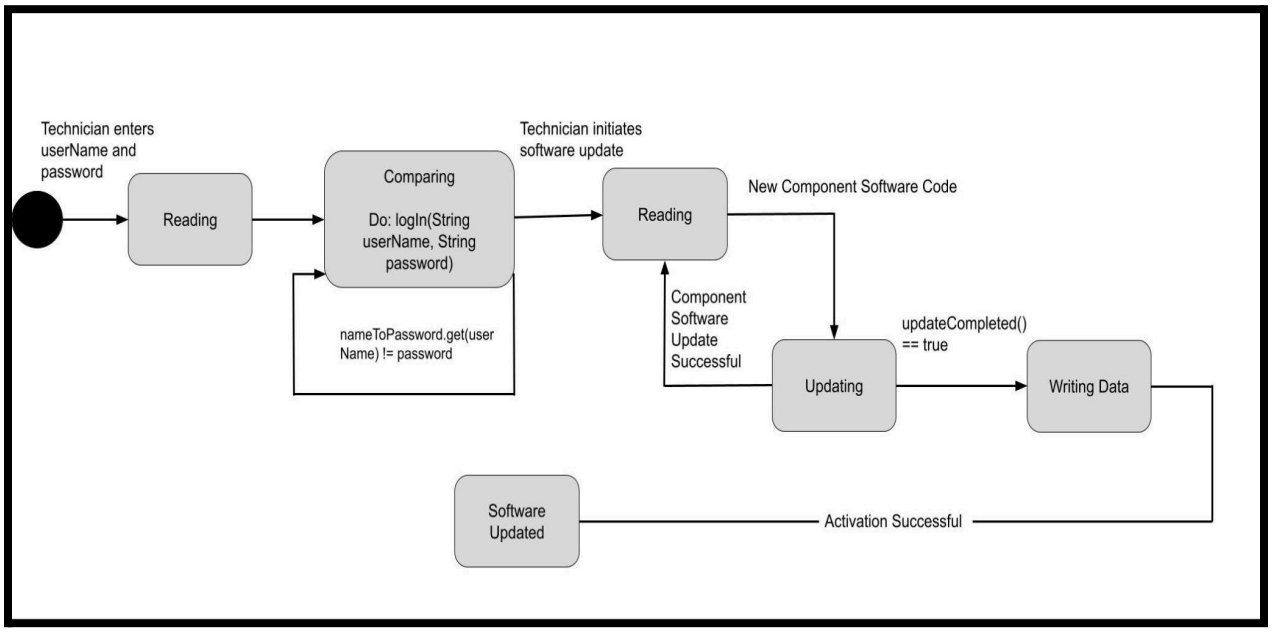
4.5.1 Object Avoidance is activated by a sensor



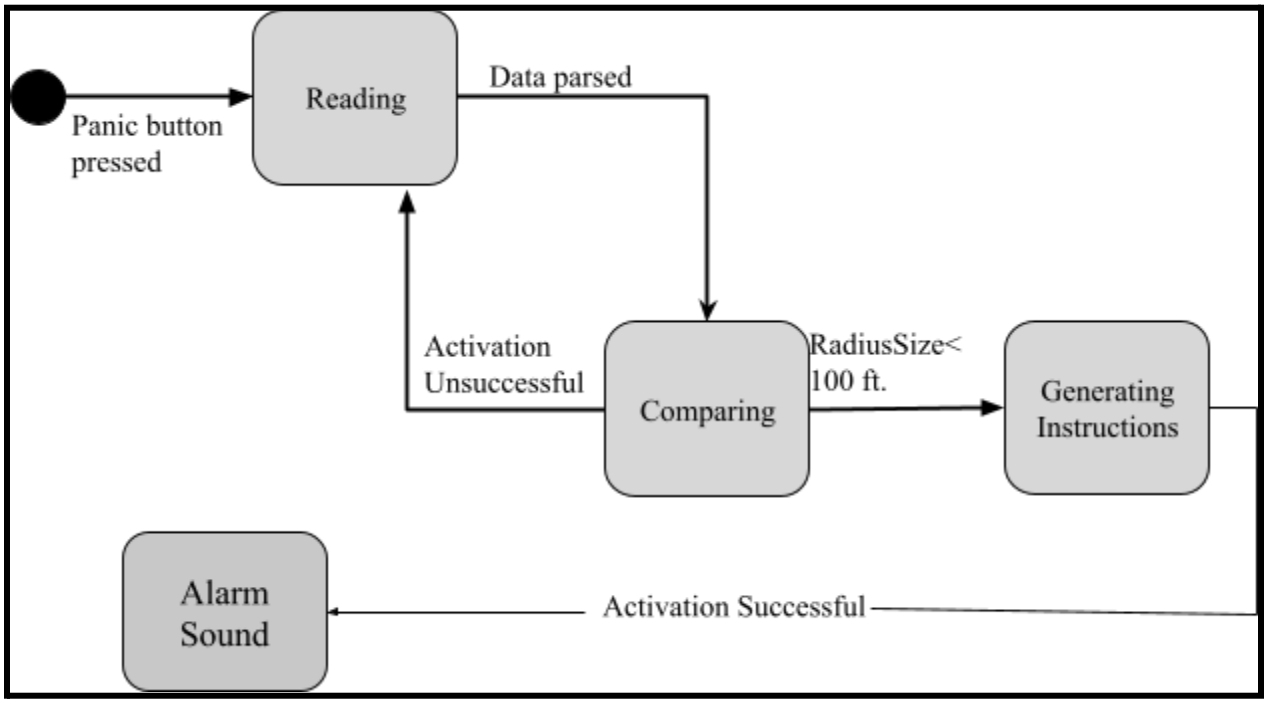
4.5.2 Key Detection is activated by a sensor



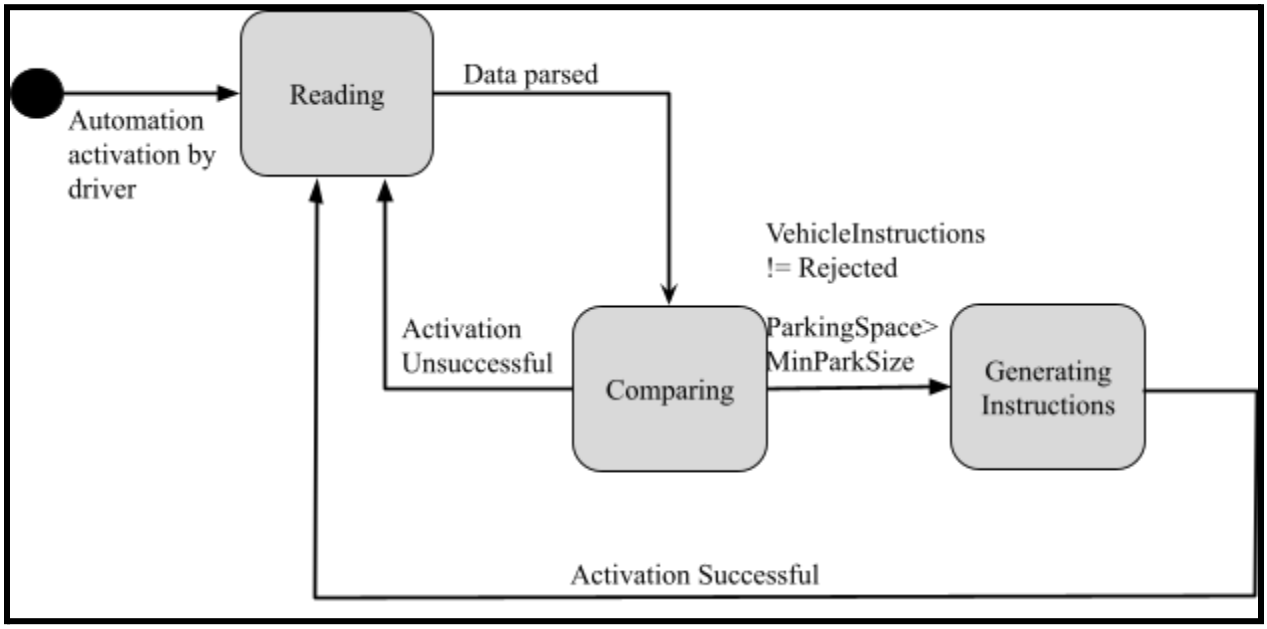
4,5,3 Software update implementation



4.5.4 Car alarm activation by key fob



4.5.5 Automated Parking Activation by Driver



Section 5: Design

5.1 Software Architecture:

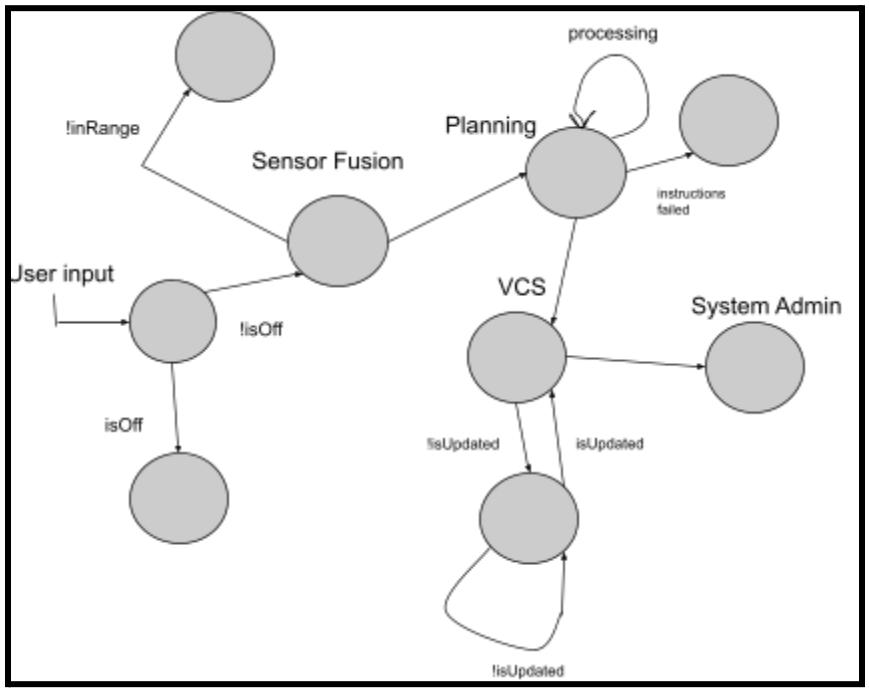
Software Architecture enables a software engineer to analyze the effectiveness of design to meet its requirements, consider the alternatives, and reduce the risks associated with software construction. The Software Architecture is a singular system that oversees and engages with numerous smaller subsystems. These subsystems encompass various elements of the Vehicle such as the Sensors, the Sensor Fusion module, the Localization and Planning modules, and the Driver Console module, among other modules that ensure the cohesive operation of the Vehicle. To understand a software architecture, major software structural elements are broken down and analyzed. The seven different architecture styles were analyzed by describing them, creating a visual flow diagram, and listing their pros and cons, before making the final decision on the optimal architecture; the options are finite state machine, layered architecture, data-center architecture, data flow architecture, call return architecture, object-oriented architecture, and model view controller architecture.

5.1.1 Finite State Machine

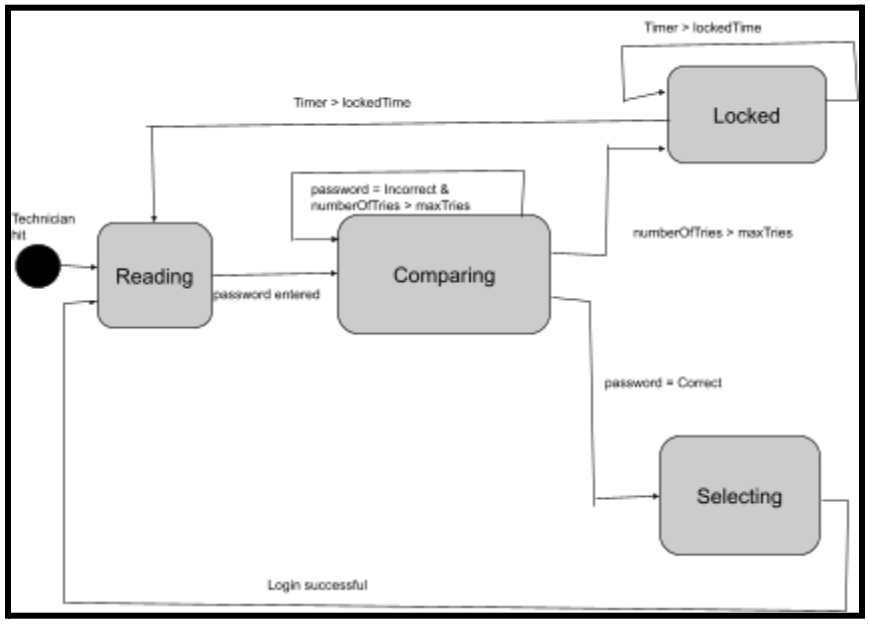
Description-

Finite state machine architecture is a programming architecture that enables the dynamic transition between states, which is contingent upon values derived from user inputs or prior states.

IOT



Technician Login



Pros

- Various checks for validation
- Allows for multiple factors of security
    - E.g. Timer, Locking, etc.
- Can be very streamlined for simple functionality

Cons

- Limited flexibility once set up
- Potential over-complexity from states
- Reworking difficulties
    - Dependent on states and transitions

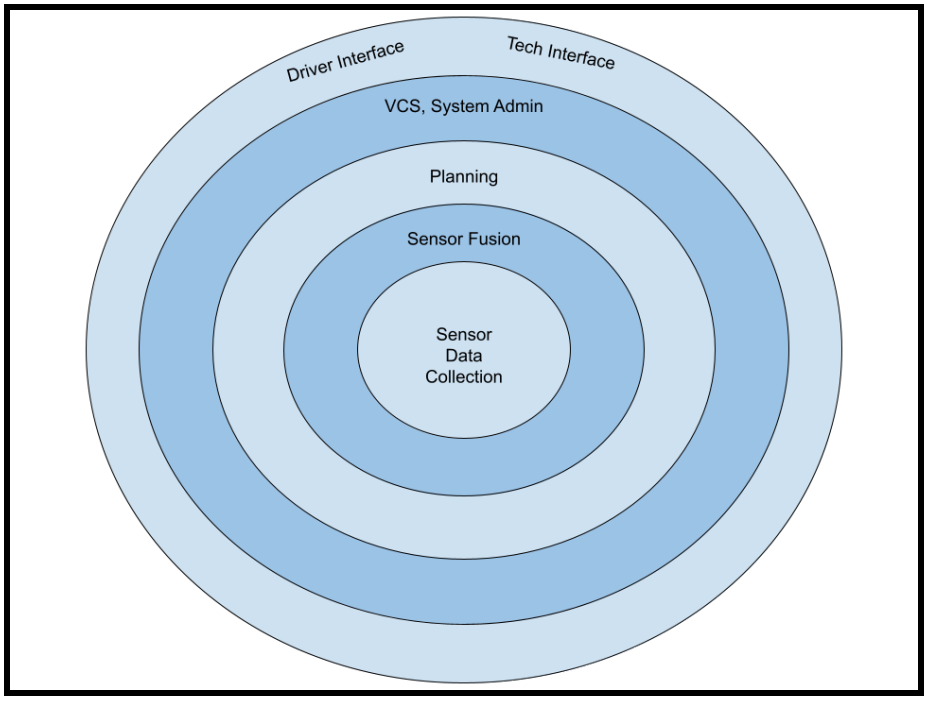
Analysis -

The Finite State Machine for Password Verification can be used in many different aspects of IOT Htl, such as for running software updates that need user verification/authorization. Though there may be some challenges in some aspects of the structure, the basic architecture functionality is sufficient for verifying a password while also checking for other security concerns. The machine, however, cannot be applied to the entire Alset IOT HTL because of the amount of states that would be necessary to implement it. With a complex real-time system such as the IOT, there will be many states necessary to correctly identify the best set of commands for the vehicle. This makes the finite state machine not an optimal choice for the IOT HTL.

5.1.2 Layered Architecture

Description-

Layered Architecture is a structural design pattern that consists of components arranged in horizontal layers, each of which operates independently from the others.



Pros-

- Separation of user-level and machine-level programs
    - Prevents user interference in machine-level programs
- Software is inherently sectioned into different levels
    - Easier to divide up programming tasks

Cons-

- Each layer can not be sectioned easily
- Limited Flexibility
- Less focus on User Interaction implications with the software
    - Focuses on interactions between layers

Analysis-

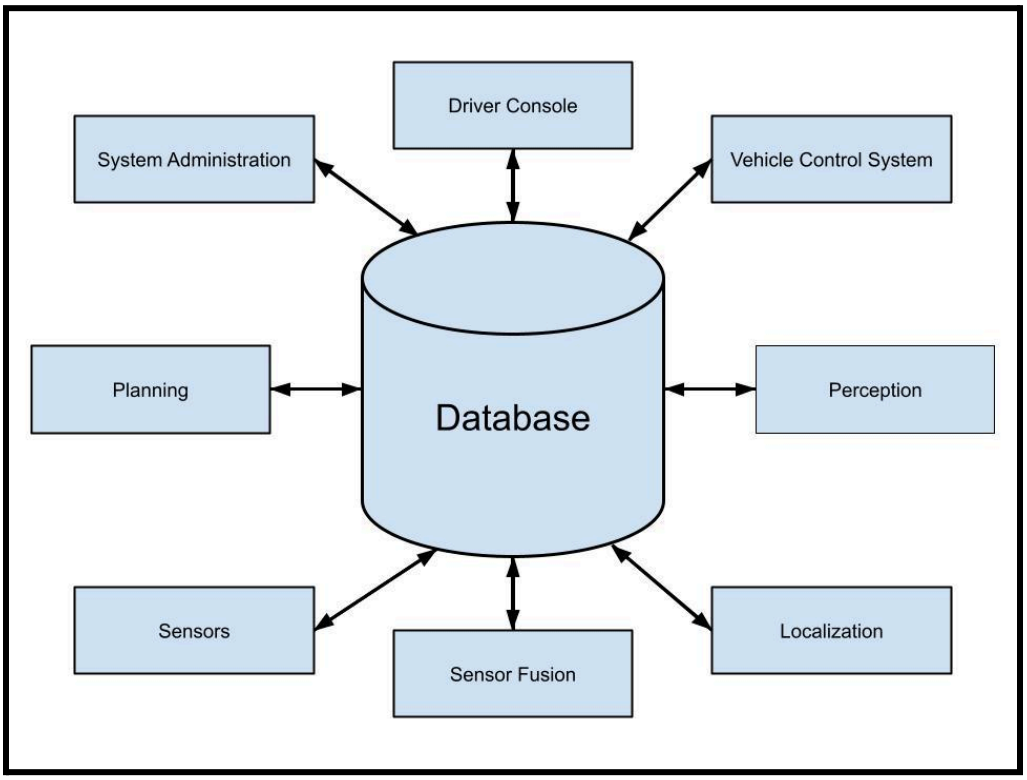
Within the Alset vehicle's IOT, the layered architecture would not be an optimal fit for software architecture. The layered architecture is typically used with something similar to an operating system. As the software implemented for the vehicle does not have more than two layers of access, the layered architecture would end up oversimplifying portions of the architecture while over-emphasizing others. A good example of this would be when designing the code there would be the sensor layer, sensor fusion, and then the rest of the classes and

methods. With those as the only three layers, there would be issues as not much of the actual code was dictated.

5.1.3 - Data-Centered Architecture

Description-

Data-centered architecture uses a centralized data repository to facilitate communication between various components of the project.



Pros-

- The concepts and rules can be kept in a table, instead of stored throughout the program
- It is simple to add to or alter, data without changing the code
- Data is secured

Cons-

- Issues can arise when looking up data in a table while running the program
- The components of the program will be defective if the data table breaks
- Does not meet the efficiency needed for a real-time system

Analysis-

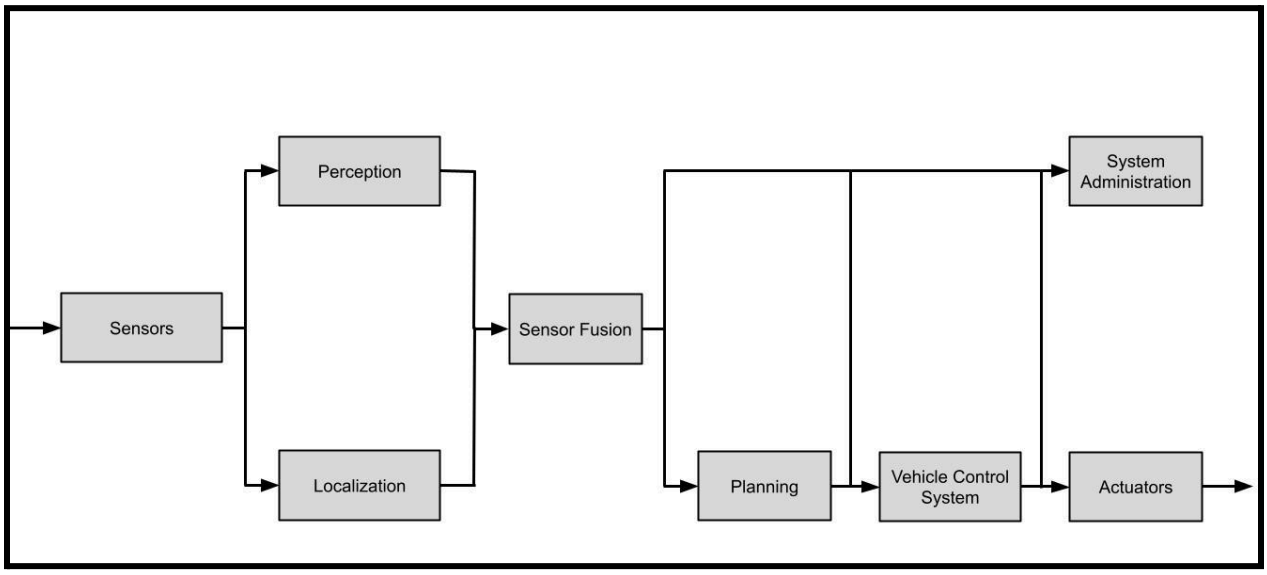
A data-centered architecture is not viable as a software architecture model within the Alset vehicle's IoT. The main reason a data-centered architecture is not viable is that it would

perform poorly as a system that needs to execute in real time. Most of the systems' decisions will rely on real-time data that is sourced from the sensors and cameras. Because of this, a centralized database would be unnecessary as the primary architectural component of our system. Using a centralized database would only slow down the Vehicle's performance, as each module would need to continually access the database to retrieve and store data, which would severely hinder our vehicle's performance.

5.1.4 - Data Flow Architecture

Description-

Data flow architecture uses a sequence of computational components, that utilize pipes and filters, to convert input data into output data.



Pros-

- The development will be quicker
- Smooth transition between design and implementation
- Can be decomposed into subsystems, each of which can operate independently to receive input data and generate output data
- Works well for systems involving sequential processing of data

Cons-

- Data coupling could occur if components are not independent of one another

- Poorly suited for interactive applications because data would need to be processed along the data flow architecture before being output/displayed

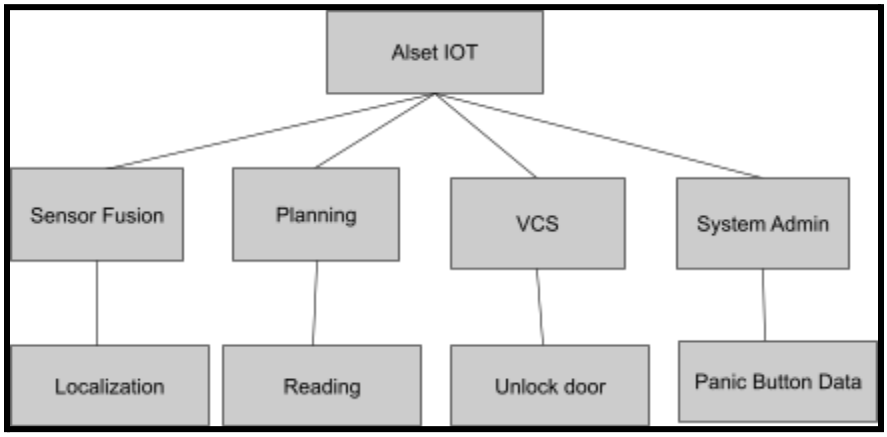
Analysis-

It could be viable to implement a data flow architecture for our Alset IoT vehicle's software. Subdividing components into subsystems can offer benefits like increased flexibility and focused component testing, which would reduce development time. The filter and pipe structure allow for data to be transferred easily which in turn makes this a good architecture for a real-time system. The time needed for data to be processed will be shorter in comparison with other architectures. There are some drawbacks as well. As the focus of this architecture is data, there can be a bit of overhead for transforming data to make it acceptable between filters.

5.1.5 - Call Return Architecture

Description-

Call return architecture partitions the main program into several components, each of which can call upon the other program components as needed.



Pros-

- Simple to iteratively develop code
- Simple to modify code
- The code is modular, which makes it easy to incorporate or remove specific functionalities without disrupting the entire project

Cons-

- Difficult and lengthy to debug code because of high modular interdependency

- Modules may be interdependent, which means that changes to core elements like data structures could necessitate restructuring across the project

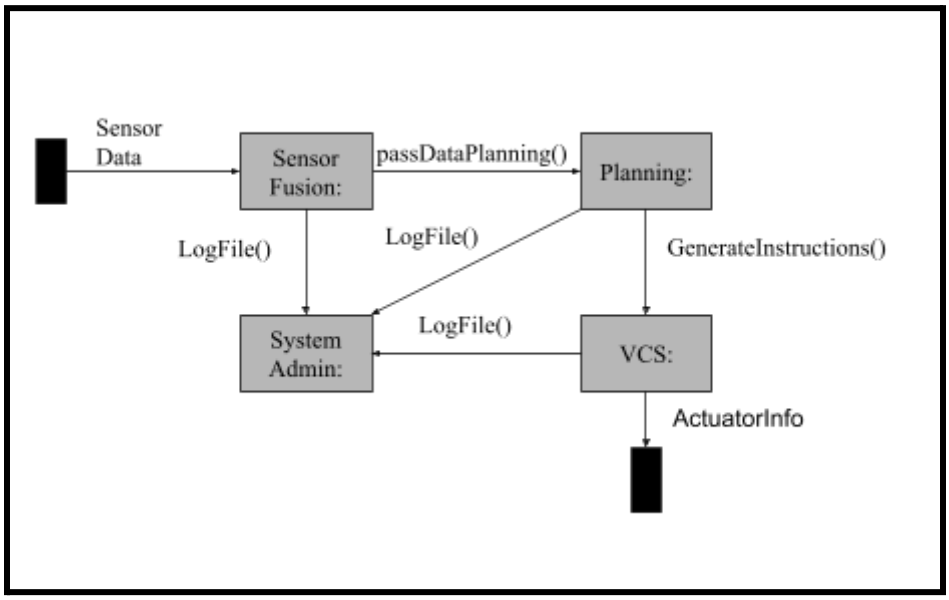
Analysis-

Call return architecture is not suitable for the Alset vehicle's IoT software system. The vehicle is composed of many modules which rely on one another. We could structure our software so that each hardware module corresponds to a software module, which is capable of requesting essential information. However, there are some difficulties with using a call return architecture. The excess of function calls would not be suitable for our real-time system. There are situations where we would have to modify the call return architecture, like if there are modules that want to use a two-way form of communication.

5.1.6 - Object-Oriented Architecture

Description-

Object-oriented architecture employs the allocation of responsibilities to reusable, collaborative objects across a system.



Pros-

- Objects help keep data secure
- The reuse of objects removes repetitive code
- Easy to modify objects

Cons

- Depending on program size, object-oriented programs may run slower than other programs

Analysis-

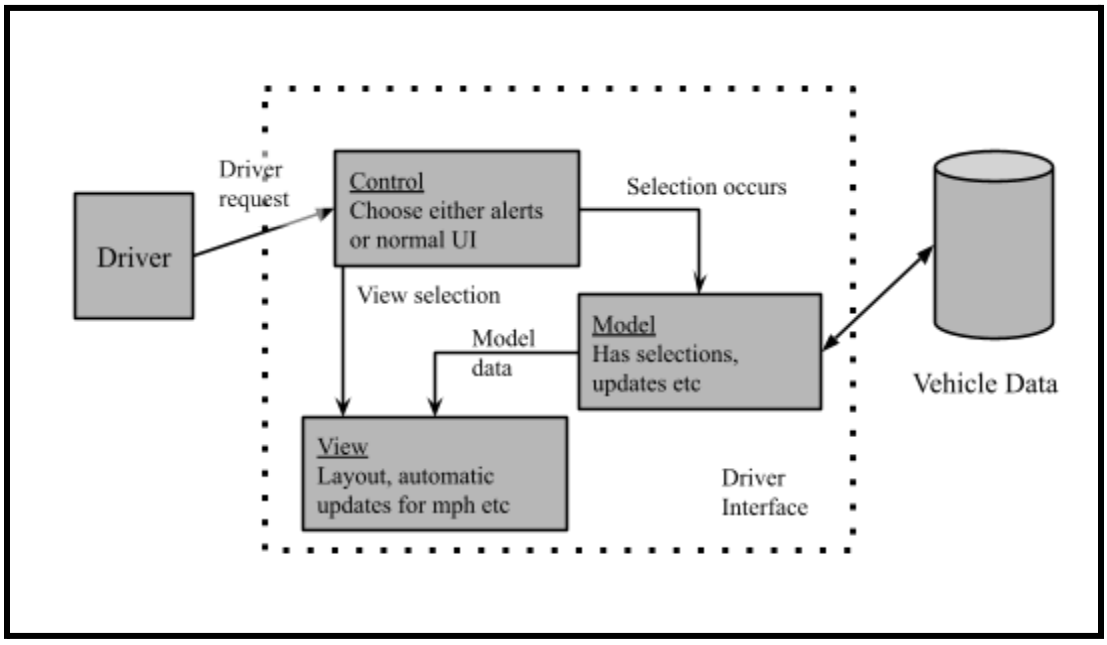
Employing an object-oriented architecture could be viable for the Alset vehicle's IoT system. We could use classes to manage the main components of the vehicle's system, which would allow for seamless communication throughout the system. The methods of each class would do the specific work that has been outlined previously. The main downside would be implementing shared information as object-oriented programming limits the viability of sharing data between classes to add security. Altogether, the object-oriented architecture is a viable option but may not be the most optimal solution for the IOT architecture.

5.1.7 - Model View Controller Architecture

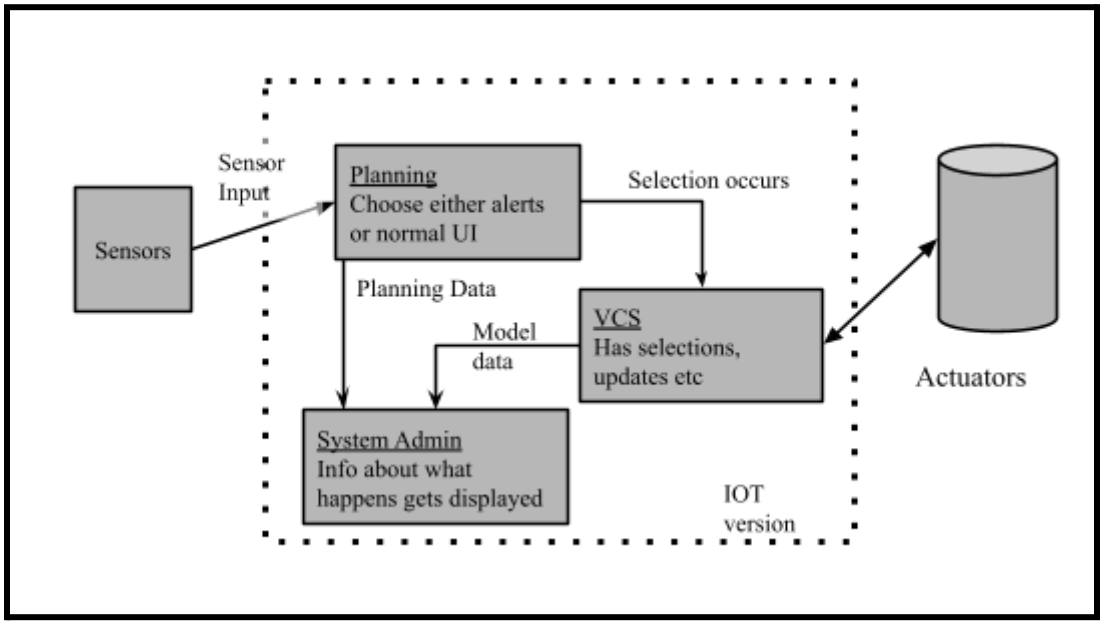
Description-

Model view controller architecture splits the software into three key components: the model, the view, and the controller. The model component manages all data-related logic in the system. The view component manages all the user interface logic. The controller component facilitates the communication between the model and view components and determines how the data is exhibited.

Driver UI version



IOT Version



Pros-

- Can modify one component without altering the other components
- High scalability because of the separated components
- Shortened development time, backend and frontend development can occur simultaneously because of the separation of components

Cons-

- Code can be difficult to understand and test because of the separation of components
- Complex design and separation of components can make it inefficient for smaller projects

Analysis-

The model view architecture is not an optimal software architecture choice because of the uneven distribution of work between the user interface and data components in the Alset vehicle's software. Our software mainly operates without using information from the user interface. The majority of the data processed by our system is from cameras and sensors, and model view architecture does not provide clear guidelines on how to structure the code so that the components compensate for this imbalance. Therefore, model view architecture is not an ideal choice for the Alset vehicle's software architectural design for the IOT portion but the driver interface could be easily deployed as a model view architecture as is depicted above. Also,

the portion of the technician interface that is not the password login could be implemented as a model view architecture.

5.1.8 Conclusion:

In regards to the Technician login, the Finite State Machine architecture is the most optimal. Its straightforwardness with multiple security factors and validation checks ensures that only the technician can access the software from the outside. This architecture will ensure that proper verification and authorization are needed to proceed with proper updates to the software. Although this architecture is optimal for the Technician login, the Object-Oriented architecture is more optimal for the IoT HTL. The Object-Oriented architecture will allow for reusability of objects to prevent repetitive code and easily modify the objects. This will simplify the structure of the code and allow uninterrupted communication throughout the system. The data flow would have also been an option for the IoT HTL, but it's insufficient since the software would have to backtrack before the appropriate display. In regards to the driver interface and non-login technician interface of the UI, it should follow the Model View Controller architecture. Since this architecture is split into three components, you can modify one without modifying the rest and it shortens the time of development. Although this is a complex design for smaller projects, for a big project like this it's most favorable.

5.2 Interface Design:

5.2.1 Driver Interface:

The Driver interface will consist of buttons and alerts. It will also display key information about the vehicle such as speed, temperature, and what features are enabled which are as follows. The Driver interface itself will receive data from the user and occasionally the planning module will send it requests for information. It will be designed in a way that is not overwhelming to the user with most features simply being an icon that is on or off. More important features will be larger, and less important will be smaller.

Features:

- Turn on/off button
- Speedometer
- Temperature Monitor

- Headlight Status
- Cruise Control Status
- Automated Parking Allowance Alert
- Windshield Wiper Status.

5.2.2 Technician Interface:

The Technician Interface will consist of buttons and access points for logging in to implement software updates or repair features. The interface will receive input from the Technician, such as the username and password for authorization, or after logging in, the interface will receive commands to deploy updates or to fix certain features that may be malfunctioning.

Features:

- Login Page
- Software Update Button
- Vehicle Outline Map
- Log files Access
- Notification/Alert Inbox

5.3 Component Level Design:

Component-level design is integral to the success of every software project. In software architectural design, component-level design is the process where the system is broken down into smaller, independent subsystems. In this section, we focused on two goals: breaking down the modules of our Vehicle's architecture into individual components and creating interfaces between these components. We have broken down our architecture into unique components, which are each defined in their class. Our team is using a class-based component-level design for our software architecture. Using class-based component-level design allows us to create reusable, organized classes for each module in our software.

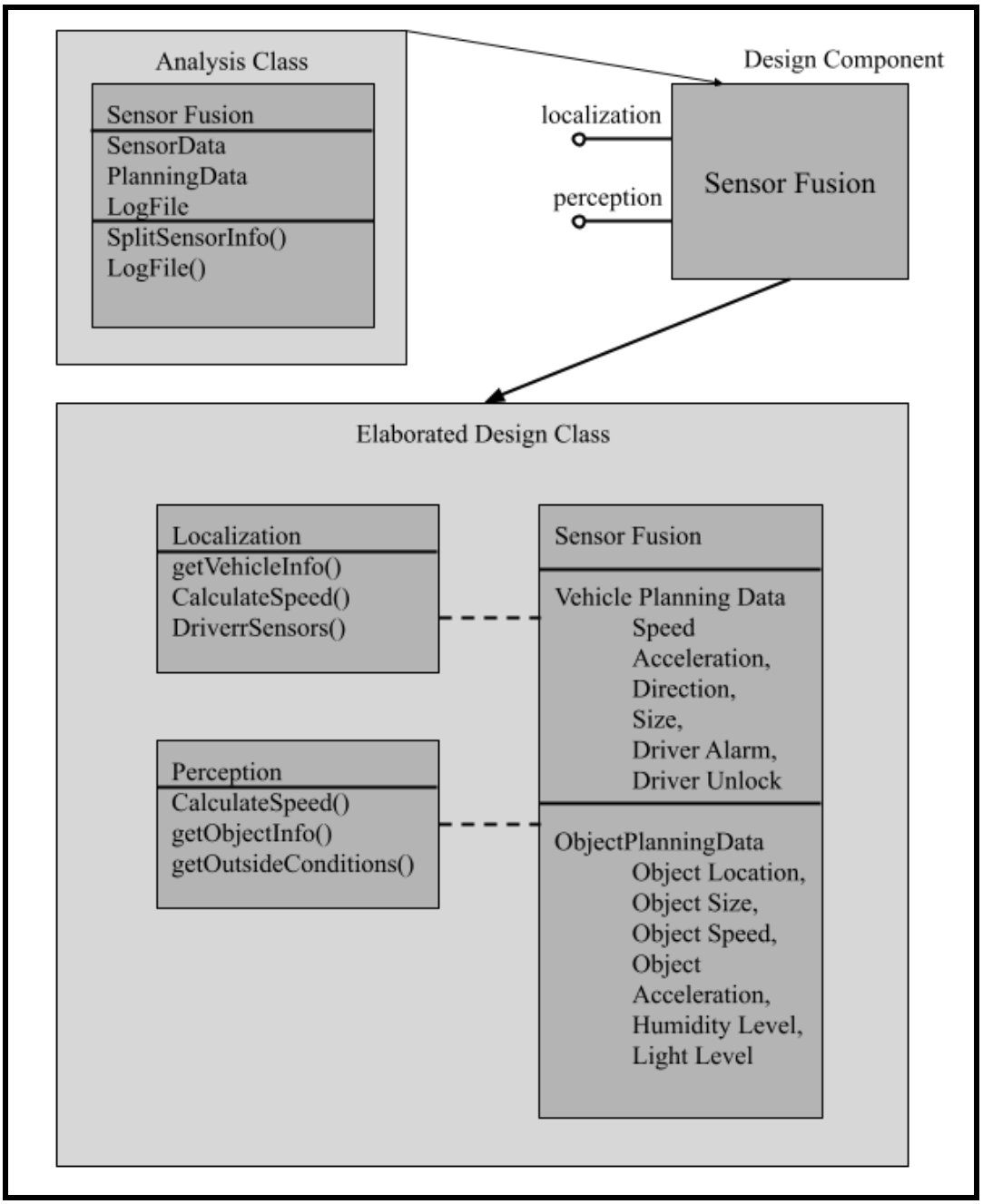


Diagram of the Sensor Fusion component class. This diagram shows the functions of Sensor Fusion, as well as the interface between the Sensor Fusion, Localization, and Perception classes.

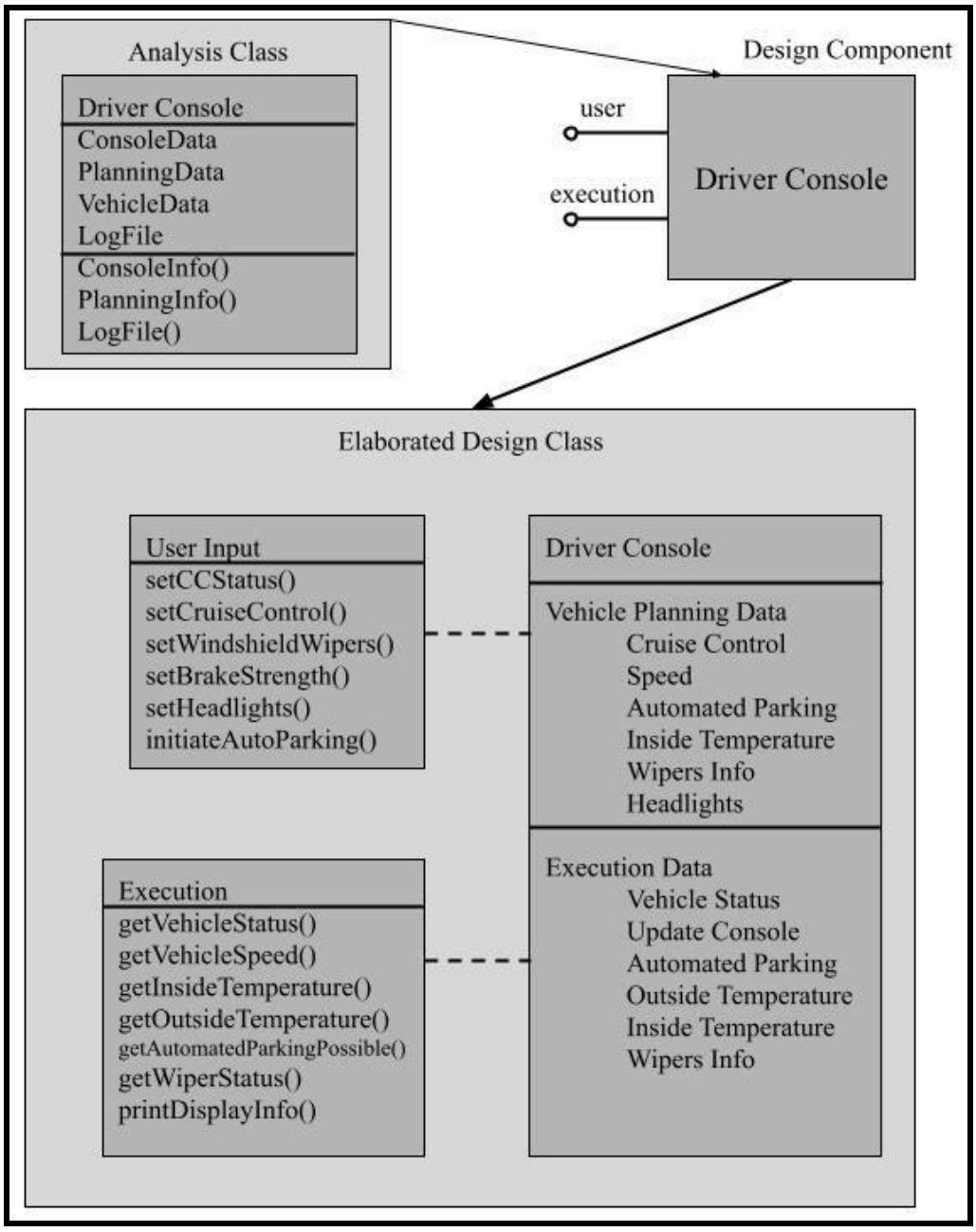


Diagram of the Driver Console component class. This diagram shows the functions of the Driver Console and details how it interacts with input from the Driver and the Planning component.

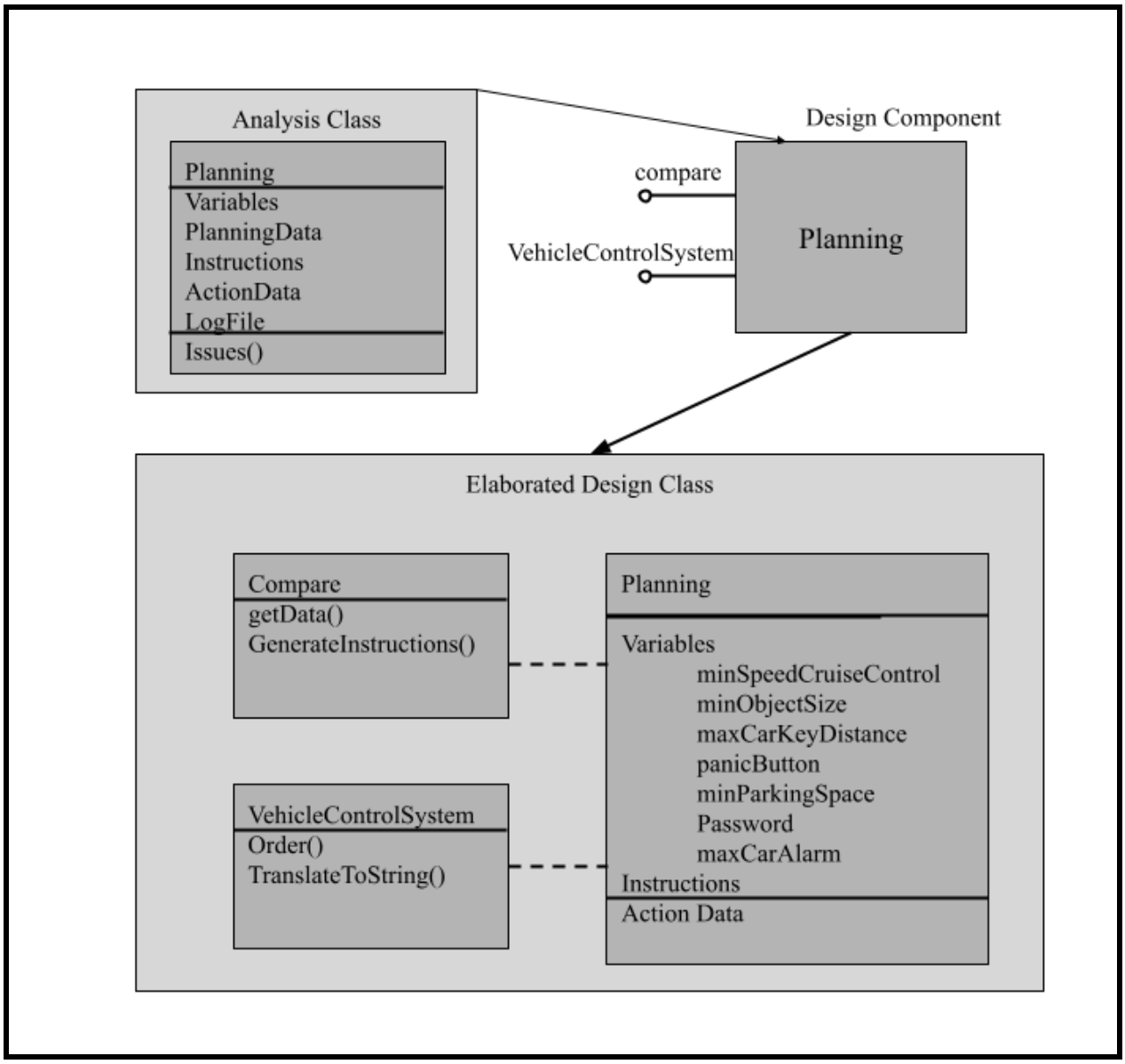


Diagram of the Planning component class. This diagram shows how Planning uses data to generate instructions, as well as the ways in which it interfaces with the VCS class.

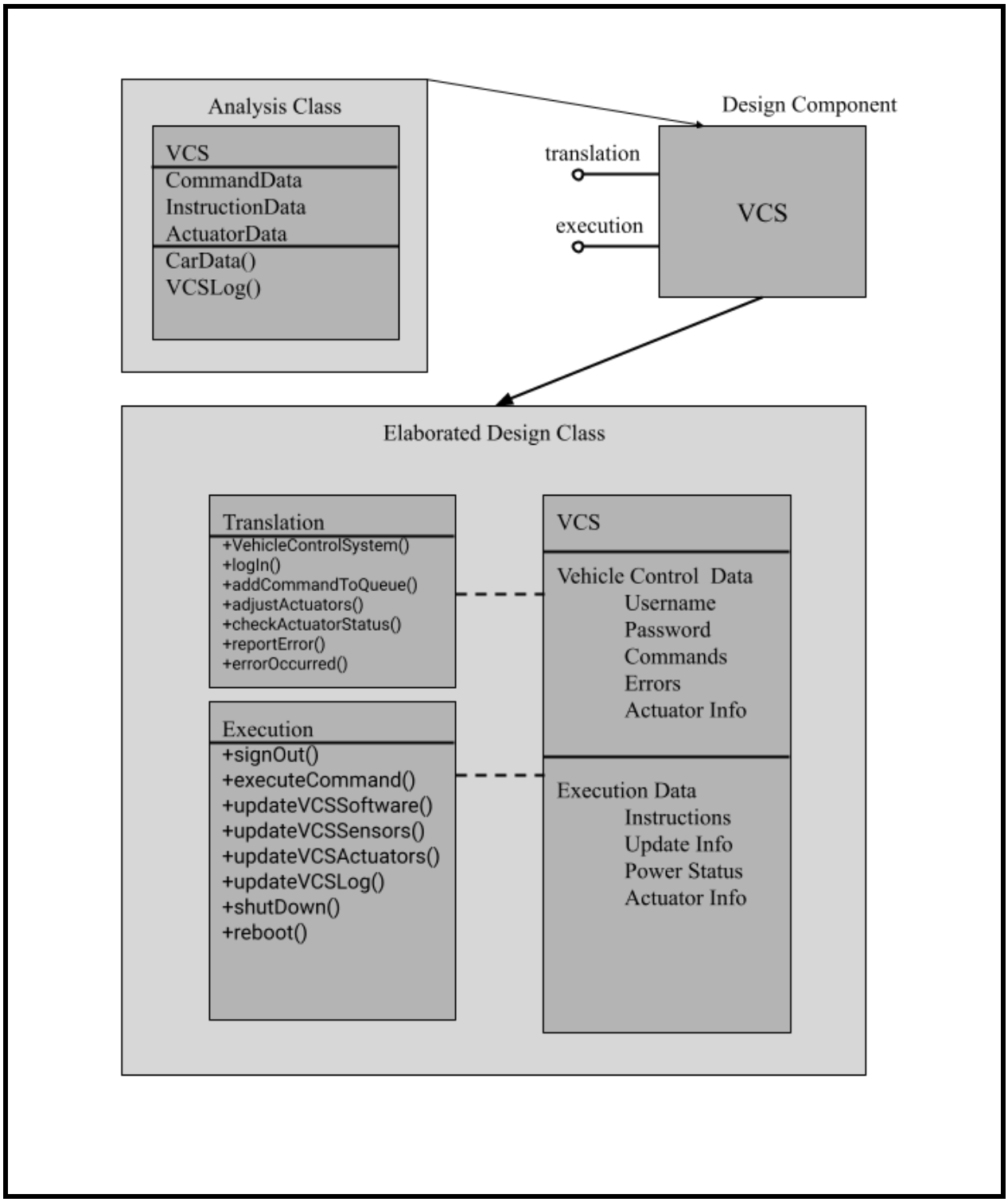


Diagram of the Vehicle Control System component class. This diagram shows the data and functions associated with VCS. It also highlights the data translation and execution processes that are associated with the VCS class.

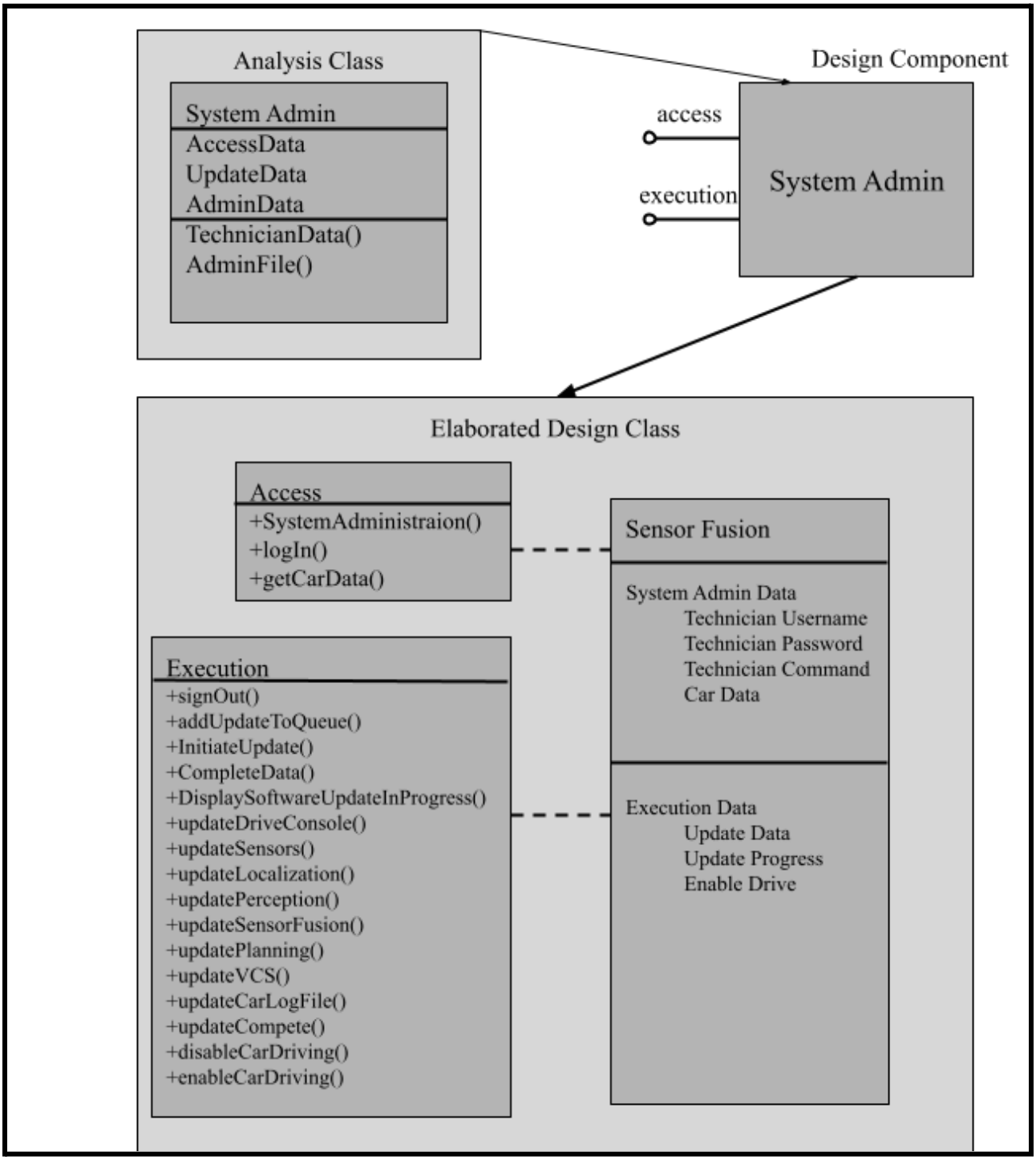


Diagram of the System Administration component class. This diagram shows the data and functions associated with the system administration. It shows how the system admin first accesses the system and then executes the necessary update actions.

6 Project Code:

6.1 Sensor Fusion Module:

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.util.ArrayList;

public class Sensor {
   public ArrayList<String> readFile(String filename){
      ArrayList<String> data = new ArrayList<>();
      ArrayList<String> logFile = new ArrayList<>();
      try {
         BufferedReader request_read = new BufferedReader(new
FileReader(System.getProperty("user.dir")+"\\request.txt"));
         String L = request_read.readLine();
         while(L != null) {
            if (L.startsWith("Cruise Control ON")) {
               data.add("Driver: Cruise Control");
               logFile.add("Driver: Cruise Control");
            } else if(L.startsWith("Cruise Control OFF")){
               data.add("Driver: Cruise Control OFF");
               logFile.add("Driver: Cruise Control OFF");
            }
            else if (L.startsWith("Automated Parking")) {
               data.add("Driver: Automated Parking");
               logFile.add("Driver: Automated Parking");
            }else if (L.startsWith("Self Driving Termination")) {
               data.add("Driver: Self Driving Termination");
               logFile.add("Driver: Self driving termination activation");
            }else if (L.startsWith("Assisted Reversing ON")) {
               data.add("Driver: Assisted Reversing");
               logFile.add("Driver: Assisted Reversing");
            }
            else if (L.startsWith("Assisted Reversing OFF")) {
               data.add("Driver: Assisted Reversing OFF");
               logFile.add("Driver: Assisted Reversing OFF");
            }
            else if (L.startsWith("Headlights Short")) {

               data.add("Driver: Headlights Short");
               logFile.add("Driver: Headlights Short");
            }
            else if (L.startsWith("Headlights Long")) {

               data.add("Driver: Headlights Long");
               logFile.add("Driver: Headlights Long");
            }else if (L.startsWith("Headlights Off")) {

               data.add("Driver: Headlights Long");
               logFile.add("Driver: Headlights Long");
            }
            L = request_read.readLine();
         }
```

```
        request_read.close();
        BufferedReader reader = new BufferedReader(new FileReader(filename));

        String line = reader.readLine();
        while (line != null) {
           if (line.startsWith("Headlights Shorts: ")) {
              String[] parts = line.split(": ");
              int light = getNumber(parts[1]);
              if (light >= 0) {
//                 System.out.println("Invalid headlights short activation: " + light);
                 logFile.add("Invalid headlights short activation: " + light);
              } else {
                 data.add("Headlights Short: " + parts[1]);
                 logFile.add("Headlights short activation: " + parts[1]);
              }
           } else if (line.startsWith("Headlights Long: ")) {
              String[] parts = line.split(": ");
              int light = getNumber(parts[1]);
              if (light >= 0) {
//                 System.out.println("Invalid headlights long activation: " + light);
                 logFile.add("Invalid headlights long activation: " + light);
              } else {
                 data.add("Headlights Long: " + parts[1]);
                 logFile.add("Headlights short activation: " + parts[1]);

              }}
           else if (line.startsWith("Humidity: ")) {
              String[] parts = line.split(": ");
              int humidity = getNumber(parts[1]);
              if (humidity >= 0) {
                 data.add("Humidity: " + humidity);
                 logFile.add("Humidity reading: " + humidity);
              } else {
//                 System.out.println("Invalid negative humidity value: " + humidity);
                 logFile.add("Invalid humidity reading: " + humidity);
              }
           } else if (line.startsWith("Speed: ")) {
              String[] parts = line.split(": ");
              int speed = getNumber(parts[1]);
              if (speed >= 0) {
                 data.add("Speed: " + speed);
                 logFile.add("Speed reading: " + speed);

              } else {
//                 System.out.println("Invalid negative speed value: " + speed);
                 logFile.add("Invalid speed reading: " + speed);
              }
           } else if (line.startsWith("Acceleration: ")) {
              String[] parts = line.split(": ");
              int acel = getNumber(parts[1]);
              if (acel >= 0) {
                 data.add("Acceleration: " + acel);
                 logFile.add("Acceleration reading: " + acel);
              } else {
//                 System.out.println("Invalid negative acceleration value: " + acel);
                 logFile.add("Invalid acceleration reading: " + acel);
```

```
            }
        }else if (line.startsWith("Object Distance: ")) {
            String[] parts = line.split(": ");
            int dist = getNumber(parts[1]);
            if (dist >= 0) {
                data.add("Object Distance: " + dist);
                logFile.add("Distance reading: " + dist);

            } else {
//                  System.out.println("Invalid negative size value: " + dist);
                logFile.add("Invalid distance reading: " + dist);
            }
        }else if (line.startsWith("Number of Objects: ")) {
            String[] parts = line.split(": ");
            int numOfObjects = getNumber(parts[1]);
            if (numOfObjects >= 0) {
                System.out.println(numOfObjects);
                data.add("Number of Objects: " + numOfObjects);
                logFile.add("Number of objects reading: " + numOfObjects);
            } else {
//                  System.out.println("Invalid number of objects value: " + numOfObjects);
                logFile.add("Invalid number of objects reading: " + numOfObjects);
            }
        }else if (line.startsWith("Object Width: ")) {
            String[] parts = line.split(": ");
            int width = getNumber(parts[1]);
            if (width >= 0) {
                data.add("Object Width: " + width);
                logFile.add("Object width reading: " + width);

            } else {
//                  System.out.println("Invalid negative size value: " + width);
                logFile.add("Invalid object width reading: " + width);

            }
        }else if (line.startsWith("Object Height: ")) {
            String[] parts = line.split(": ");
            int height = getNumber(parts[1]);
            if (height >= 0) {
                data.add("Object Height: " + height);
                logFile.add("Object height reading: " + height);

            } else {
//                  System.out.println("Invalid negative size value: " + height);
                logFile.add("Invalid object height reading: " + height);

            }
        }else if (line.startsWith("Object Length: ")) {
            String[] parts = line.split(": ");
            int length = getNumber(parts[1]);
            if (length >= 0) {
                data.add("Object Length: " + length);
                logFile.add("Object width length: " + length);

            } else {
//                  System.out.println("Invalid negative size value: " + length);
```

```
                logFile.add("Invalid object length reading: " + length);

            }
        }else if (line.startsWith("LightLevel: ")) {
            String[] parts = line.split(": ");
            int light = getNumber(parts[1]);
            if (light >= 0) {
                data.add("LightLevel: " + light);
                logFile.add("Light level reading: " + light);

            } else {
//                System.out.println("Invalid negative light value: " + light);
                logFile.add("Invalid light level  reading: " + light);

            }
        } else if (line.startsWith("Lanes On The Right: ")) {
            String[] parts = line.split(": ");
            int lanes = getNumber(parts[1]);
            if (lanes >= 0) {
                data.add("Lanes On The Right: " + lanes);
                logFile.add("Lanes On The Right Reading: " + lanes);
            } else {
                logFile.add("Invalid lanes on the right reading: " + lanes);
            }
        }else if (line.startsWith("Lanes On The Left: ")) {
            String[] parts = line.split(": ");
            int lanes = getNumber(parts[1]);
            if (lanes >= 0) {
                data.add("Lanes On The Left: " + lanes);
                logFile.add("Lanes On The Left Reading: " + lanes);
            } else {
                logFile.add("Invalid lanes on the left reading: " + lanes);
            }
        } else if (line.startsWith("Front Right Blind Spot: ")) {
            if (line.equalsIgnoreCase("Front Right Blind Spot: True")) {
                data.add("Front Right Blind Spot: True");
                logFile.add("Front Right Blind Spot Reading: True");
            } else if (line.equalsIgnoreCase("Front Right Blind Spot: False")) {
                data.add("Front Right Blind Spot: False");
                logFile.add("Front Right Blind Spot: False");
            } else {
                logFile.add("Invalid Front Right Blind Spot Reading");
            }
        } else if (line.startsWith("Front Left Blind Spot: ")) {
            if (line.equalsIgnoreCase("Front Left Blind Spot: True")) {
                data.add("Front Left Blind Spot: True");
                logFile.add("Front Left Blind Spot Reading: True");
            } else if (line.equalsIgnoreCase("Front Left Blind Spot: False")) {
                data.add("Front Left Blind Spot: False");
                logFile.add("Front Left Blind Spot: False");
            } else {
                logFile.add("Invalid Front Left Blind Spot Reading");
            }
        } else if (line.startsWith("Back Right Blind Spot: ")) {
            if (line.equalsIgnoreCase("Back Right Blind Spot: True")) {
                data.add("Back Right Blind Spot: True");
```

```
                logFile.add("Back Right Blind Spot Reading: True");
            } else if (line.equalsIgnoreCase("Back Right Blind Spot: False")) {
                data.add("Back Right Blind Spot: False");
                logFile.add("Back Right Blind Spot: False");
            } else {
                logFile.add("Invalid Back Right Blind Spot Reading");
            }
        } else if (line.startsWith("Back Left Blind Spot: ")) {
            if (line.equalsIgnoreCase("Back Left Blind Spot: True")) {
                data.add("Back Left Blind Spot: True");
                logFile.add("Back Left Blind Spot Reading: True");
            } else if (line.equalsIgnoreCase("Back Left Blind Spot: False")) {
                data.add("Back Left Blind Spot: False");
                logFile.add("Back Left Blind Spot: False");
            } else {
                logFile.add("Invalid Back Left Blind Spot Reading");
            }
        }else if (line.startsWith("Location: ")) {
            String[] parts = line.split(": ");
            if (line.equalsIgnoreCase("Location: Ahead") || line.equalsIgnoreCase("Location: Left") ||
line.equalsIgnoreCase("Location: Right")) {
                data.add(line);
                logFile.add("Location reading: " + line);
            } else {
//                System.out.println("Invalid negative acceleration value: " + loc);
                logFile.add("Invalid location reading: " + line);
            }
        }else if (line.startsWith("Key: ")) {
            String[] parts = line.split(": ");
            int key = getNumber(parts[1]);
            if (key <= 0) {
//                System.out.println("Invalid key detection: " + key);
                logFile.add("Invalid key detection: " + key);
            } else {
                data.add("Key: " + parts[1]);
                logFile.add("Key detection: " + parts[1]);

            }

        }else if (line.startsWith("Door Touched: ")) {
            String[] parts = line.split(": ");
            String door = parts[1];
            System.out.println(door);
            if (door.equals("driver\n") || door.equals("other\n")) {
//                System.out.println("Invalid door detection: " + door);
                data.add("Door Touched: " + parts[1]);
                logFile.add("Door detection: " + parts[1]);
            } else {
                logFile.add("Invalid door detection : " + door);
            }
        }else if (line.startsWith("Traffic Light: ")) {
            String[] parts = line.split(": ");
            int light = getNumber(parts[1]);
            if (light >= 0) {
//                System.out.println("Invalid traffic light detection: " + light);
                logFile.add("Invalid traffic light color reading: " + light);
```

```
            } else {
                data.add("Traffic Light: " + parts[1]);
                logFile.add("Traffic light color reading: " + parts[1]);
            }
        } else if (line.startsWith("Driver Detected")) {
            data.add("Driver Detected");
        } else if (line.startsWith("Traffic Light Distance: ")) {
            String[] parts = line.split(": ");
            int lightDistance = getNumber(parts[1]);
            if (lightDistance < 0) {
//                System.out.println("Invalid traffic light distance: " + lightDistance);
                logFile.add("Invalid traffic light distance reading: " + lightDistance);
            } else {
                data.add("Traffic Light Distance: " + parts[1]);
                logFile.add("Traffic light distance reading: " + parts[1]);
            }
        }else if (line.startsWith("Car: ON")) {
//                System.out.println("Invalid traffic light detection: " + light);
            data.add("CAR ON: ");
            logFile.add("CAR: ON");

        }else if (line.startsWith("Car: OFF")) {
//                System.out.println("Invalid traffic light detection: " + light);
            data.add("CAR OFF: ");
            logFile.add("CAR: OFF");

        }else if (line.startsWith("Car Gear: Reverse")) {

//                System.out.println("Invalid traffic light detection: " + light);
            data.add("CAR GEAR: Reverse");
            logFile.add("CAR GEAR: Reverse");
        }
        else if (line.startsWith("Panic Button: True")) {
//                System.out.println("Invalid traffic light detection: " + light);
            data.add("Panic Button: On");
            logFile.add("Panic Button: Activated");

        }else if (line.startsWith("Parking Space: ")) {
            String[] parts = line.split(": ");
            int parkingSpace = getNumber(parts[1]);
//                System.out.println("Invalid traffic light detection: " + light);
            data.add("Parking Space: "+parkingSpace);
            logFile.add("Parking Space Size: "+parkingSpace+"sq ft");

        }else if (line.startsWith("Parking Space Direction: ")) {
            String[] parts = line.split(": ");
            String parkingDir= parts[1];
//                System.out.println("Invalid traffic light detection: " + light);
            data.add("Parking Space Direction: "+parkingDir);
            logFile.add("Parking Space Direction: "+parkingDir);

        }

        line = reader.readLine();
    }
    reader.close();
```

```
    }catch(FileNotFoundException e) {
        System.out.println(e);
    }catch(IOException e) {
        System.out.println(e);
    }
    writeLogFile(logFile);
    return data;
}
private void writeLogFile(ArrayList<String> dataList) {
    try {
        File myObj = new File("log.txt");
        FileWriter fw = new FileWriter(myObj);

        BufferedWriter bw = new BufferedWriter(fw);

        for (String data : dataList) {
            bw.write(data);
            bw.newLine();
        }

        bw.close();
        copyToLogDirectory(myObj);

    } catch (IOException e) {

        e.printStackTrace();
    }

}
private void copyToLogDirectory(File file) throws IOException {
    // Check if 'log' directory exists, create if not
    File logDirectory = new File("log");
    if (!logDirectory.exists()) {
        if (logDirectory.mkdir()) {
            System.out.println("Directory 'log' created successfully.");
        } else {
            System.out.println("Failed to create directory 'log'.");
            return; // Exit method if directory creation fails
        }
    }
    // Copy file to 'log' directory if it exists
    if (logDirectory.exists()) {
        Files.copy(file.toPath(), Path.of("log", "SENSORlog.txt"), StandardCopyOption.REPLACE_EXISTING);
        System.out.println(file.getName() + " copied to 'log' directory.");
    }
}
private static int getNumber(String str) {
    int num = 0;
    try {
        num = Integer.parseInt(str);
        return num;
    }catch(NumberFormatException e) {
        return -1;
    }
}
private static void printDataList(ArrayList<String> dataList) {
```

```
        System.out.println("Sensor Data List:");
        for (String data : dataList) {
            System.out.println(data);
        }
    }
}
```

6.2 Planning Module:

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.StandardCopyOption;
import java.util.List;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.LinkedList;
import java.util.Queue;
import java.nio.file.Files;
import java.nio.file.Path;

public class Planning {
    //Variables
    private int numberOfObjectsDetected;
    private int moisture;
    private int speed;
    private int ParkingSpace;
    private int temp;
    private int lightLevel;
    private int minObjectDistance;
    private int minParkingSpace;
    private int lanesOnRight;
    private int lanesOnLeft;
    private double trafficLightDistance;
    private double objectHeight;
    private double objectWidth;
    private double objectDistance;
    private boolean objectAvoided;
    private boolean driverDetected;
    private boolean cruiseControlOn;
    private boolean driverHeadlightCommand;
    private boolean wipersAreOn;
    private boolean keyDetected;
    private boolean carIsOn; // TODO INITIALIZE THIS BASED OFF OF INPUT
    private boolean carAlarmIsOn;
    private boolean panicButtonPressed;
    private boolean driverDoorTouched;
    private boolean otherDoorTouched;
    private boolean blindSpotsChecked;
    private boolean objectInFrontRightBlindSpot;
    private boolean objectInFrontLeftBlindSpot;
    private boolean objectInBackRightBlindSpot;
    private boolean objectInBackLeftBlindSpot;
    private String location;
```

```
private String parkingDir;
private String carGear;
private String[] data;
private ArrayDeque<String> instructions1;
private ArrayDeque<String> instructions2;
private ArrayDeque<String> instructions3;
// Methods
// Constructor - receives array of strings from sensor fusion
// Input format: ["humdity: 70"];
public Planning(ArrayList<String> data) {
   // Initializing all variables
   numberOfObjectsDetected = -1;
   lightLevel = 50; // 50 lux ambient light
   moisture = -100;
   temp = -100; // -100 indicates that the temperature has not been detected yet
   minObjectDistance = 300; // if object is closer than 300 ft, avoid it
   minParkingSpace = 162; // need at least 162 square feet to park : 9 ft * 18 ft
   trafficLightDistance = -1;
   objectHeight = 0;
   ParkingSpace=0;
   objectWidth = 0;
   objectDistance = 0;
   lanesOnRight = 0;
   lanesOnLeft = 0;
   objectAvoided = false;
   carIsOn = false;
   driverDetected = false;
   cruiseControlOn = false;
   driverHeadlightCommand = false;
   wipersAreOn = false;
   keyDetected = false;
   carAlarmIsOn = false;
   panicButtonPressed = false;
   objectInFrontRightBlindSpot = false;
   objectInFrontLeftBlindSpot = false;
   objectInBackRightBlindSpot = false;
   objectInBackLeftBlindSpot = false;
   blindSpotsChecked = false;
   location = "";
   carGear="drive";
   instructions1 = new ArrayDeque<>(); // safety instructions
   instructions2 = new ArrayDeque<>(); // driver request instructions
   instructions3 = new ArrayDeque<>(); // otSher instructions
   this.data = new String[data.size()];
   this.data = data.toArray(this.data);
   // Remove all the spaces from input
   for (int i = 0; i < this.data.length; i++) {
      this.data[i] = this.data[i].replaceAll("\\s", "");
   }
   // Find out if car is on and if someone is in the car
   for (int i = 0; i < this.data.length; i++) {
      String command = this.data[i].split("[:]")[0];
      if (command.equalsIgnoreCase("CARON")) {
         carIsOn = true;
      }
      if (command.equalsIgnoreCase("DriverDetected")) {
```

```
            driverDetected = true;
        }
    }
    generateInstructions();
}
/**
 * Takes a String[] of data and generates instructions
 */
private void generateInstructions() {
    for (int i = 0; i < data.length; i++) {
        String command = data[i].split("[:]")[0];
        if (carIsOnAndDriverDetected() && !objectAvoided) {
          blindSpotDetection();
          blindSpotsChecked = true;
          avoidObject();
          objectAvoided = true;
        }
        if(command.equals("CARON")){
            carIsOn=true;
            instructions1.add("Car Status: ON");
        }
        if(command.equals("CAROFF")){
            carIsOn=false;
            instructions1.add("Car Status: OFF");
        }
        if(command.equals("Speed")){
            speed=Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
            instructions1.add("Car Speed: "+speed);
        }
        if(command.equals("ParkingSpace")){
            ParkingSpace=Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
            instructions1.add("Parking Space: "+ParkingSpace);
        }
        if(command.equals("ParkingSpaceDirection")){
            parkingDir=data[i].substring(data[i].lastIndexOf(":") + 1);
            instructions1.add("Parking Space Direction: "+parkingDir);
        }
        if(command.equals("Driver")){
            String request= data[i].split("[:]")[1];
            // CRUISE CONTROL
            if(request.equals("CruiseControl")){
                // Find out if car is on, get speed, and get number of objects
                for (int j = 0; j < data.length; j++) {
                    String input = data[j].split("[:]")[0];
                    if (input.equalsIgnoreCase("CARON")) {
                        carIsOn = true;
                    }
                    if (input.equalsIgnoreCase("Speed")) {
                        speed = Integer.parseInt(data[j].substring(data[j].lastIndexOf(":") + 1));
                        instructions2.add("Car Speed: "+speed);
                    }
                    if (input.equalsIgnoreCase("NumberOfObjects")) {
                        numberOfObjectsDetected = Integer.parseInt(data[j].substring(data[j].lastIndexOf(":") + 1));
                    }
                }
                // If car is on and speed >= 50 mph, activate cruise control
```

```
            if(carIsOn && speed >= 50 && numberOfObjectsDetected < 3){
                //success added
                instructions1.add("Request CC: Success");
                cruiseControlOn = true;
            }else{
                //fail
                instructions1.add("Request CC: Fail");
            }
        }
        if(request.equals("CruiseControlOFF")){
            instructions1.add("TURN OFF CC");
        }
        // ASSISTED REVERSING
        if(request.equals("AssistedReversing")){
            //decide to activate assisted reversing
            for (int j = 0; j < data.length; j++) {
                String input = data[j].split("[:]")[0];
                if (input.equalsIgnoreCase("NumberOfObjects")) {
                    numberOfObjectsDetected = Integer.parseInt(data[j].substring(data[j].lastIndexOf(":") + 1));
                }
                if (input.equalsIgnoreCase("CARON")) {
                    carIsOn = true;
                }
                if(input.equalsIgnoreCase("CARGEAR")){
                    carGear=data[j].substring(data[j].lastIndexOf(":") + 1);
                    instructions1.add("Car Gear: Reverse");
                }
            }
            if(carIsOn && carGear.equals("Reverse") && numberOfObjectsDetected==0){
                //success added
                instructions1.add("Request Assisted Reverse: Success");
            }else{
                //fail
                instructions1.add("Request Assisted Reverse: Fail");
            }
        }
        if(request.equals("AssistedReversingOFF")){
            instructions1.add("Request Assisted Reverse: OFF");
        }
        if(request.equals("SelfDrivingTermination")){
            //turn off self driving
            instructions1.add("Self Driving Terminate");
        }
        // AUTOMATED PARKING
        if(request.equals("AutomatedParking")){
            //decide to activate automated parking
            if(carIsOn
&&(speed<=15)&&(ParkingSpace>minParkingSpace)&&((parkingDir=="left")||(parkingDir=="right"))){
                //success added
                instructions1.add("Request Automated Parking: Success");
            }else{
                //fail
                instructions1.add("Request Automated Parking: Fail");
            }
        }
        if (request.equals("HeadlightsShort") || request.equals("HeadlightsLong")) {
```

```
            // If the driver has sent a headlight request, cancel any automatic headlight instructions
            if (request.equals("HeadlightsShort")) {
                instructions1.add("TURN ON HEADLIGHTS: SHORT");
            } else if(request.equals("HeadlightsLong")){
                instructions1.add("TURN ON HEADLIGHTS: LONG");
            }else{
                instructions1.add("TURN OFF: HEADLIGHTS");
            }
            driverHeadlightCommand = true;
            System.out.println("Driver headlight command: " + driverHeadlightCommand);
        }
    }
    if (command.equals("LightLevel")) {
        int light = Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
        // Loop through data until moisture and temp are found
        boolean found1, found2;
        if (temp == -100) {
            found1 = false;
        } else {
            found1 = true;
        }
        if (moisture == -100) {
            found2 = false;
        } else {
            found2 = true;
        }
        int j = 0;
        while (!found1 || !found2) {
            if (j == data.length) {
                j = 0;
            }
            String type = data[j].split("[:]")[0];
            if (type.equalsIgnoreCase("Temperature")) {
                found1 = true;
                temp = Integer.parseInt(data[j].substring(data[j].lastIndexOf(":") + 1));
                instructions2.add("Temperature: "+temp);
            }
            if (type.equalsIgnoreCase("Moisture")) {
                found2 = true;
                moisture = Integer.parseInt(data[j].substring(data[j].lastIndexOf(":") + 1));
            }
            j++;
        }
        // Now that the moisture level and temp are known, check the 3 cases
        if (objectDetected() && light < lightLevel && moisture == 0) { // Case 1: light < 50 and no moisture
            instructions1.add("HEADLIGHTS: NORMAL");
        }
        if (moisture > 70 && light < lightLevel && temp > 32) { // Case 2: moisture > 70 and low light and temp
> 32
            instructions1.add("HEADLIGHTS: FOG LIGHTS");
        }
        if (!objectDetected() && moisture == 0 && light < lightLevel) {
            instructions1.add("HEADLIGHTS: HIGH BEAMS");
        }
        temp = -100;
        light = -100;
```

```
    moisture = -100;
}
if (command.equals("Humidity")) {
    // Getting the humidity value and storing it as an 'int'
    int humidity = Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
    // If humidity >= 70%, add windshield wiper command to instructions
    if (humidity >= 70) {
        instructions1.add("WINDSHIELD WIPERS: ON");
        wipersAreOn = true;
    }
    if (humidity < 70 && wipersAreOn){
        instructions1.add("WINDSHIELD WIPERS: OFF");
        wipersAreOn = false;
    }
}
if (command.equals("Key")) { // All key based actions
    String keyValue = data[i].substring(data[i].lastIndexOf(":") + 1);
    if (keyValue.equalsIgnoreCase("no")) {
        keyDetected = false;
    } else {
        keyDetected = true;
    }
    // Determine if panic button has been pressed and if door is touched
    for (int j = 0; j < data.length; j++) {
        if (data[j].split("[ ]")[0].equals("PanicButton")) {
            String panic = data[j].substring(data[j].lastIndexOf(":") + 1);
            if (panic.equalsIgnoreCase("yes")) {
                panicButtonPressed = true;
            } else {
                panicButtonPressed = false;
            }
        }
        if (data[j].split("[:]")[0].equals("DoorTouched")) {
            String touched = data[j].substring(data[j].lastIndexOf(":") + 1);
            if (touched.equalsIgnoreCase("driver")) {
                driverDoorTouched = true;
            }
            if (touched.equalsIgnoreCase("other")){
                otherDoorTouched = true;
            }
        }
    }
    // If panic button pressed, sound alarm
    if (panicButtonPressed) {
        instructions1.add("CAR ALARM: ON");
        carAlarmIsOn = true;
    }
    // If the key is detected, check which door is touched before sending command
    if (keyDetected && !panicButtonPressed) {
        if (!otherDoorTouched) {
            instructions1.add("UNLOCK DRIVER DOOR");
            carAlarmIsOn = false;
        }
        if (otherDoorTouched) {
            instructions1.add("UNLOCKED ALL DOORS");
            carAlarmIsOn = false;
```

```
                }
            }
            // Set both door touched variables to 'false'
            driverDoorTouched = false;
            otherDoorTouched = false;
        }
        if (command.equalsIgnoreCase("TrafficLight") && driverDetected && carIsOn) {
            // Get the traffic light distance and determine if someone is in vehicle
            for (int j = 0; j < data.length; j++) {
                String comm = data[j].split("[:]")[0];
                if (comm.equalsIgnoreCase("TrafficLightDistance")) {
                    trafficLightDistance = Double.parseDouble(data[j].substring(data[j].lastIndexOf(":") + 1));
                }
            }
            String trafficLightColor = data[i].substring(data[i].lastIndexOf(":") + 1);
            if (trafficLightColor.equalsIgnoreCase("Green")) {
                instructions3.add("TRAFFIC LIGHT GREEN");
            }
            // Slow down if light is yellow and it is >= 100 ft away
            if (trafficLightColor.equalsIgnoreCase("Yellow") && trafficLightDistance >= 100) {
                // If CC is on, turn it off
                if (cruiseControlOn) {
                    instructions1.add("TURN OFF CC");
                }
                instructions1.add("TRAFFIC LIGHT YELLOW: REDUCE SPEED TO STOP WITHIN " +
trafficLightDistance + " FEET");
            }
            if (trafficLightColor.equalsIgnoreCase("Yellow") && trafficLightDistance < 100) {
                instructions1.add("TRAFFIC LIGHT YELLOW: MAINTAIN SPEED");
            }
            if (trafficLightColor.equalsIgnoreCase("Red")) {
                // If CC is on, turn it off
                if (cruiseControlOn) {
                    instructions1.add("TURN OFF CC");
                }
                instructions1.add("TRAFFIC LIGHT RED: SLOW TO STOP BEFORE INTERSECTION");
            }
        }
    }
  }
  /**
   * Checks 'data' to see if there is an object detected by the sensors
   * @return True if object is detected, False otherwise
   */
  private boolean objectDetected() {
    // Loop to get object height, width, and distance
    for (int i = 0; i < data.length; i++) {
        String command = data[i].split("[:]")[0];
        if (command.equalsIgnoreCase("ObjectDistance")) {
            objectDistance = Double.parseDouble(data[i].substring(data[i].lastIndexOf(":") + 1));
        }
        if (command.equalsIgnoreCase("ObjectHeight")) {
            objectHeight = Double.parseDouble(data[i].substring(data[i].lastIndexOf(":") + 1));
        }
        if (command.equalsIgnoreCase("ObjectWidth")) {
            objectWidth = Double.parseDouble(data[i].substring(data[i].lastIndexOf(":") + 1));
```

```
        }
        if (command.equalsIgnoreCase("Location")) {
                location = data[i].substring(data[i].lastIndexOf(":") + 1);
        }
        if (command.equalsIgnoreCase("LanesOnTheRight")) {
                lanesOnRight = Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
        }
        if (command.equalsIgnoreCase("LanesOnTheLeft")) {
                lanesOnLeft = Integer.parseInt(data[i].substring(data[i].lastIndexOf(":") + 1));
        }
    }
    // Check if object is not detected
    if (objectHeight == 0 || objectWidth == 0 || objectDistance == 0) {
        return false;
    }
    return true;
}
/**
 * Detects if an object is in the car's blind spot
 * @return True if there is an object in the blind spot, else false
 */
private boolean blindSpotDetection() {
        // If the car is not on or the driver is not detected, return false
        if (!carIsOnAndDriverDetected()) {
                return false;
        }
        // Find the condition of all 4 blind spots
        for (int i = 0; i < data.length; i++) {
                String command = data[i];
                if (command.equalsIgnoreCase("FrontRightBlindSpot:True")) {
                        objectInFrontRightBlindSpot = true;
                }
                if (command.equalsIgnoreCase("FrontLeftBlindSpot:True")) {
                        objectInFrontLeftBlindSpot = true;
                }
                if (command.equalsIgnoreCase("BackRightBlindSpot:True")) {
                        objectInBackRightBlindSpot = true;
                }
                if (command.equalsIgnoreCase("BackLeftBlindSpot:True")) {
                        objectInBackLeftBlindSpot = true;
                }
        }
        // Check all four blind spots for objects, send command if object is there
        if (objectInFrontRightBlindSpot) {
                instructions2.add("OBJECT DETECTED IN FRONT RIGHT BLIND SPOT");
        }
        if (objectInFrontLeftBlindSpot) {
                instructions2.add("OBJECT DETECTED IN LEFT RIGHT BLIND SPOT");
        }
        if (objectInBackRightBlindSpot) {
                instructions2.add("OBJECT DETECTED IN BACK RIGHT BLIND SPOT");
        }
        if (objectInBackLeftBlindSpot) {
                instructions2.add("OBJECT DETECTED IN BACK LEFT BLIND SPOT");
        }
```

```
        return (objectInFrontRightBlindSpot || objectInFrontLeftBlindSpot || objectInBackRightBlindSpot ||
objectInBackLeftBlindSpot);
    }
    /**
     * Determines how to avoid object that is detected
     * @return True if object was avoided, else false
     */
    private boolean avoidObject() {
        // Return 'false' if object is not detected or the car is not on or driver is not detected
        if (!objectDetected() || !carIsOn || !driverDetected) {
            return false;
        }
        // If object is detected, check if it needs to be avoided
        // Avoid object if it is <= 300 ft away, and its height or width is greater than 0.5 ft
        if (objectDistance <= minObjectDistance && (objectHeight >= 0.5 || objectWidth >= 0.5)) {
            // If the object is obstructing the whole lane and there is no lane to move over into on the left or right
            if (location.equalsIgnoreCase("Ahead") && (lanesOnRight == 0 && lanesOnLeft == 0)) {
                    instructions1.add("OBJECT DETECTED: SLOW DOWN TO A STOP");
                    instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                    return true;
            }
            // If object is ahead and there is a lane to merge into
            if (location.equalsIgnoreCase("Ahead") && (lanesOnRight > 0 || lanesOnLeft > 0)) {
                    // Merge to the right
                    if (lanesOnRight > 0 && !objectInBackRightBlindSpot && !objectInBackRightBlindSpot) {
                            instructions1.add("OBJECT DETECTED: MERGE INTO RIGHT LANE");
                            instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                            return true;
                    }
                    // Merge to the left
                    if (lanesOnLeft > 0 && !objectInBackLeftBlindSpot && !objectInBackLeftBlindSpot) {
                            instructions1.add("OBJECT DETECTED: MERGE INTO LEFT LANE");
                            instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                            return true;
                    }
                    // If they cannot merge because object is in blind spot
                    instructions1.add("OBJECTED DETECTED: SLOW DOWN TO A STOP");
                    instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                    return true;
            }
            // If the object is on the left and it does not take up the full lane
            if (location.equalsIgnoreCase("Left") && !objectInBackRightBlindSpot &&
!objectInFrontRightBlindSpot) {
                    instructions1.add("OBJECT DETECTED: MOVE 0.5 FEET TO THE RIGHT AND REDUCE
SPEED BY 30%");
                    instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                    return true;
            } else if (location.equalsIgnoreCase("Right") && !objectInBackLeftBlindSpot &&
!objectInFrontLeftBlindSpot) {
                    instructions1.add("OBJECT DETECTED: MOVE 0.5 FEET TO THE LEFT AND REDUCE
SPEED BY 30%");
                    instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
                    return true;
            } else {
                    instructions1.add("OBJECT DETECTED: SLOW DOWN TO A STOP");
                    instructions1.add("SOUND: ALERT DRIVER THAT OBJECT DETECTED");
```

```
            }
        }
        return false;
    }
    /**
     * Determines if car is on and driver is detected
     * @return True if both are true, else false
     */
    private boolean carIsOnAndDriverDetected() {
            for (int i = 0; i < data.length; i++) {
                    String command = data[i].split("[:]")[0];
                    if (command.equalsIgnoreCase("CARON")) {
                            carIsOn = true;
                    }
                    if (command.equalsIgnoreCase("DriverDetected")) {
                            driverDetected = true;
                    }

            }
            return (carIsOn && driverDetected);
    }

    /**
     * Sends all of the queued instructions to VCS, clears the instruction queue and sends all info to logs
     * @return instructions queue
     */
    public Queue<String> getAllInstructions() {
        File myObj = new File("planningLog.txt");
        Queue<String> ret = new ArrayDeque<>();
        try {
            FileWriter fw = new FileWriter(myObj);
            BufferedWriter bw = new BufferedWriter(fw);
            String single_instruction;
            while (!instructions1.isEmpty()) {
                // If driver has implemented a headlight command, do not send automatic ones
                if (driverHeadlightCommand && (instructions1.peekFirst().equalsIgnoreCase("headlights: normal") ||
instructions1.peekFirst().equalsIgnoreCase("headlights: fog lights") ||
instructions1.peekFirst().equalsIgnoreCase("headlights: high beams"))) {
                    instructions1.poll();
                }
                if (instructions1.isEmpty()) {
                    break;
                }
                single_instruction = instructions1.poll();
                ret.add(single_instruction);
                bw.write(single_instruction+"\n");
            }
            while (!instructions2.isEmpty()) {
                single_instruction = instructions2.poll();
                ret.add(single_instruction);
                bw.write(single_instruction+"\n");
            }
            while (!instructions3.isEmpty()) {
                single_instruction = instructions3.poll();
                ret.add(single_instruction);
                bw.write(single_instruction+"\n");
```

```
            }
            bw.close();
            copyToLogDirectory(myObj);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return ret;
    }
    /**
     * Sends one instruction to VCS, and removes that instruction from the queue
     * @return top priority instruction
     */
    public String getSingleInstruction() {
        if (!instructions1.isEmpty()) {
            // If driver has implemented a headlight command, do not send automatic ones
            if (driverHeadlightCommand && (instructions1.peekFirst().equalsIgnoreCase("headlights: normal") ||
instructions1.peekFirst().equalsIgnoreCase("headlights: fog lights") ||
instructions1.peekFirst().equalsIgnoreCase("headlights: high beams"))) {
                instructions1.poll();
            }
            return instructions1.poll();
        }
        if (!instructions2.isEmpty()) {
            return instructions2.poll();
        }
        if (!instructions3.isEmpty()) {
            return instructions3.poll();
        }
        return null;
    }
    // Sends an instruction to the logs
    public void updateLogInfo(String singleInstruction) {
        try {
            File myObj = new File("planningLog.txt");
            FileWriter fw = new FileWriter(myObj);

            BufferedWriter bw = new BufferedWriter(fw);
            bw.write(singleInstruction);
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
    private void copyToLogDirectory(File file) throws IOException {
        // Check if 'log' directory exists, create if not
        File logDirectory = new File("log");
        if (!logDirectory.exists()) {
            if (logDirectory.mkdir()) {
                System.out.println("Directory 'log' created successfully.");
            } else {
                System.out.println("Failed to create directory 'log'.");
                return; // Exit method if directory creation fails
            }
        }
        // Copy file to 'log' directory if it exists
        if (logDirectory.exists()) {
```

```
            Files.copy(file.toPath(), Path.of("log", "PLANNINGlog.txt"),
StandardCopyOption.REPLACE_EXISTING);
            System.out.println(file.getName() + " copied to 'log' directory.");
        }
    }

}
```

6.3 Vehicle Control System Module:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.text.DecimalFormat;
import java.util.*;

public class VCS {
    private Queue<String> QueuedInstructions;
    public VCS(Queue<String> QueuedInstructions) {
        this.QueuedInstructions = QueuedInstructions;
    }
    public void parseInstructions() {
        String[] instructions= queueToArray(QueuedInstructions);
        Integer mph=0;
        String direction="left";
        try {
            File myObj = new File("actuators.txt");
            if (myObj.createNewFile()) {
                System.out.println("File created: " + myObj.getName());
            }
            FileWriter myWriter = new FileWriter("actuators.txt");
            for(int i=0; i< instructions.length;i++){
                if(instructions[i].equals("WINDSHIELD WIPERS: ON")){
                    myWriter.write("Wiper STATUS: ON\n");}
                if(instructions[i].equals("Wipers OFF")){
                    myWriter.write("WINDSHIELD WIPERS: OFF\n");}
                if(instructions[i].equals("Car Status: ON")){
                    myWriter.write("CAR: ON\n");;}
                if(instructions[i].equals("Car Status: OFF")){
                    myWriter.write("CAR: OFF\n");}
                if(instructions[i].contains("Car Speed: ")){
                    mph=Integer.valueOf(instructions[i].substring(instructions[i].length()-2));
                    myWriter.write("CURRENT SPEED: "+mph+"\n");
                }
                if(instructions[i].contains("Temperature: ")){
                    Integer temp=Integer.valueOf(instructions[i].substring(instructions[i].length()-2));
                    myWriter.write("CURRENT TEMPERATURE: "+temp+"\n");
                }
                if(instructions[i].contains("Parking Space Direction: ")){
                    int colonIndex = instructions[i].indexOf(':');
                    direction=instructions[i].substring(colonIndex + 1).trim();
                    myWriter.write("PARK DIRECTION: "+direction+"\n");
                }
                if(instructions[i].equals("UNLOCK CAR")){
                    myWriter.write("UNLOCK DOORS\n");
```

```
            }
            if(instructions[i].contains("TURN ON HEADLIGHTS: SHORT")||instructions[i].contains("TURN ON
HEADLIGHTS: LONG")){
               if(instructions[i].contains("SHORT")){
                  myWriter.write("TURN ON: NORMAL HEADLIGHTS\n");}
               else{
                  myWriter.write("TURN ON: BRIGHT HEADLIGHTS\n");
               }
            }
            if(instructions[i].equals("TURN OFF: HEADLIGHTS")){
               myWriter.write("TURN OFF: HEADLIGHTS\n");
            }
            if(instructions[i].equals("CAR ALARM: ON")){
               myWriter.write("TURN ON: CAR ALARM\n");
               myWriter.write("LOCK DOORS\n");
            }
            if(instructions[i].equals("CAR ALARM: OFF")){
               myWriter.write("TURN OFF: CAR ALARM\n");
            }
            if(instructions[i].equals("UNLOCK DRIVER DOOR")){
               myWriter.write("UNLOCK: DRIVER DOOR\n");
               myWriter.write("CAR ALARM: OFF\n");
            }
            if(instructions[i].equals("UNLOCKED ALL DOORS")){
               myWriter.write("UNLOCK: ALL DOORS\n");
               myWriter.write("CAR ALARM: OFF\n");
            }
            if(instructions[i].contains("Request CC: Success")){
               myWriter.write("Cruise Control= Engaged\n");
               myWriter.write("SET SPEED for CC: "+mph+"\n");
            }
            if(instructions[i].contains("Request CC: Fail")){
               myWriter.write("Cruise Control Attempt= Fail\n");
            }
            if(instructions[i].contains("Request Automated Parking: Success")){
               myWriter.write("Adjust Steering to : "+ direction);
               myWriter.write("\nReverse 180 ft\n");
               myWriter.write("Adjust Steering opposite of: "+ direction);
               myWriter.write("\nForward 1ft\n");
               myWriter.write("Engage Park\n");
            }
            if(instructions[i].contains("Request Automated Parking: Fail")){
               myWriter.write("Automated Park request= Fail\n");
            }
            if(instructions[i].contains("Request Assisted Reverse: Success")){
               myWriter.write("Reverse 10 ft\n");
            }
            if(instructions[i].contains("Request Assisted Reverse: OFF")){
               myWriter.write("Assisted Reversse Request= off\n");
            }if(instructions[i].contains("Request Assisted Reverse: Fail")){
               myWriter.write("Assisted Reversse Request= Fail\n");
            }if(instructions[i].contains("TRAFFIC LIGHT RED: SLOW TO STOP BEFORE
INTERSECTION")||instructions[i].contains("OBJECT DETECTED: SLOW DOWN TO A STOP")){
               if(instructions[i].contains("OBJECT DETECTED")){
                  myWriter.write("Object was Detected\n");
               }
```

```
            myWriter.write("Apply Brake 10 percent\n");
            myWriter.write("Apply Brake 20 percent\n");
            myWriter.write("Apply Brake 50 percent\n");
        }if(instructions[i].contains("TRAFFIC LIGHT GREEN")){
            myWriter.write("Detected Green Light\n");
            myWriter.write("Maintain Speed\n");
        }if(instructions[i].contains("TRAFFIC LIGHT YELLOW: MAINTAIN SPEED")){
            myWriter.write("Detected Yellow Light\n");
            myWriter.write("MSG: Distance To Light To Low To Stop\n");
            myWriter.write("Maintain Speed\n");
        }
        if(instructions[i].contains("TRAFFIC LIGHT YELLOW: REDUCE SPEED")){
            myWriter.write("Detected Yellow Light\n");
            int feetIndex = instructions[i].indexOf("FEET");
            // Extract the substring containing the distance value
            String distanceSubstring = instructions[i].substring(instructions[i].lastIndexOf(" ", feetIndex - 2) + 1,
feetIndex).trim();
            // Parse the distance value to an integer
            int trafficLightDistance = Integer.parseInt(distanceSubstring);
            DecimalFormat df = new DecimalFormat("#.##");
            myWriter.write("MSG: Distance to light is "+trafficLightDistance+"\n");
            myWriter.write("Apply Brake "+df.format((100-(trafficLightDistance/100))*0.1)+"percent\n");
            myWriter.write("Apply Brake "+df.format((100-(trafficLightDistance/100))*0.2)+"percent\n");
            myWriter.write("Apply Brake "+df.format((100-(trafficLightDistance/100))*0.5)+"percent\n");
            myWriter.write("Apply Brake "+df.format((100-(trafficLightDistance/100))*0.8)+"percent\n");
            myWriter.write("Apply Brake 100 percent\n");
        }if(instructions[i].contains("Self Driving Terminate")){
            myWriter.write("TURN OFF: Self Driving\n");
        }
        if((instructions[i].contains("BLIND SPOT")&&instructions[i].contains("OBJECT
DETECTED"))||instructions[i].contains("SOUND: ALERT DRIVER")){
            myWriter.write("Alarm: Sound\n");
            String lor="";
            String frontback="";
            if(instructions[i].contains("LEFT")){
                lor="left";
            }else{
                lor="right";
            }
            if(instructions[i].contains("FRONT")){
                frontback="front";
            }else{
                frontback="back";
            }
            if(instructions[i].contains("BLIND SPOT")){
                myWriter.write("Object in: "+frontback+" "+lor+" Blind Spot\n");
            }
        }
        if(instructions[i].contains("OBJECT DETECTED: MERGE")){
            String merge="";
            if (instructions[i].contains("LEFT")) {
                merge="left";
            }else{
                merge="right";
            }
            myWriter.write("Turn on: "+merge+" blinker\n");
```

```
                myWriter.write("Adjust Steering: "+merge+" direction\n");
                myWriter.write("Adjust Speed: Traffic Conditions\n");
            }
            if(instructions[i].contains("OBJECT DETECTED: MOVE 0.5 FEET")){
                String merge="";
                if (instructions[i].contains("LEFT")) {
                    merge="left";
                }else{
                    merge="right";
                }
                myWriter.write("Adjust Steering: "+merge+" direction for 0.5 Ft\n");
                myWriter.write("Adjust Speed: 30% Slower\n");
            }
        }
        myWriter.close();
        copyToLogDirectory(myObj);
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
private void copyToLogDirectory(File file) throws IOException {
    // Check if 'log' directory exists, create if not
    File logDirectory = new File("log");
    if (!logDirectory.exists()) {
        if (logDirectory.mkdir()) {
            System.out.println("Directory 'log' created successfully.");
        } else {
            System.out.println("Failed to create directory 'log'.");
            return; // Exit method if directory creation fails
        }
    }
    // Copy file to 'log' directory if it exists
    if (logDirectory.exists()) {
        Files.copy(file.toPath(), Path.of("log", "VCSlog.txt"), StandardCopyOption.REPLACE_EXISTING);
        System.out.println(file.getName() + " copied to 'log' directory.");
    }
}
public static String[] queueToArray(Queue<String> queue) {
    // Create an array with the same size as the queue
    String[] array = new String[queue.size()];
    // Iterate over the queue and fill the array
    int index = 0;
    for (String str : queue) {
        array[index++] = str;
    }
    return array;
}
}
```

6.4 Driver Interface:

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

public class DriverUI {
   private Queue<String> QueuedInstructions;
   public DriverUI(Queue<String> QueuedInstructions) {
      this.QueuedInstructions = QueuedInstructions;
   }
   public void generateUI() {
      String[] instructions= queueToArray(QueuedInstructions);
      String WiperStatus="off";
      String CarStatus="off";
      String Headlights="off";
      String RequestCC="";
      String CC="off";
      String Reverse="off";
      String RequestPark="";
      String RequestReverse = "";
      Integer ObjectDetected=0;
      Integer BlindSpot=0;
      Integer mph=0;
      Integer temperature=0;
      for(int i=0; i< instructions.length;i++){
         if(instructions[i].contains("WINDSHIELD WIPERS: ON")){
            WiperStatus="on";}
         if(instructions[i].equals("WINDSHIELD WIPERS: OFF")){
            WiperStatus="off";}
         if(instructions[i].equals("Car Status: ON")){
            CarStatus="on";}
         if(instructions[i].equals("Car Status: OFF")){
            CarStatus="off";}
         if(instructions[i].contains("Car Speed: ")){
            mph=Integer.valueOf(instructions[i].substring(instructions[i].length()-2));
         }
         if(instructions[i].contains("Temperature: ")){
            temperature=Integer.valueOf(instructions[i].substring(instructions[i].length()-2));
         }
         if(instructions[i].contains("TURN ON HEADLIGHTS: SHORT")||instructions[i].contains("TURN ON
HEADLIGHTS: LONG")){
            Headlights="on";
         }
         if(instructions[i].contains("TURN OFF: HEADLIGHTS")){
            Headlights="off";
         }
         if(instructions[i].contains("Request CC: Success")){
            RequestCC="Success";
            CC="on";
         }
         if(instructions[i].contains("Request CC: Fail")){
            RequestCC="Fail";
            CC="off";
         }
         if(instructions[i].contains("Request Automated Parking: Success")){
```

```
            RequestPark="Success";
        }
        if(instructions[i].contains("Request Automated Parking: Fail")){
            RequestPark="Fail";
        }
        if(instructions[i].contains("Request Assisted Reverse: Success")){
            RequestReverse="Success";
            Reverse="on";
        }
        if(instructions[i].contains("Request Assisted Reverse: OFF")){
            RequestReverse="Success";
            Reverse="off";
        }
        if(instructions[i].contains("Request Assisted Reverse: Fail")){
            RequestReverse="Fail";
        }
        if(instructions[i].contains("TURN OFF CC")||instructions[i].contains("Self Driving Terminate")){
            CC="off";
        }
        if(instructions[i].contains("SOUND: ALERT DRIVER")){
            ObjectDetected=1;
        }
        if(instructions[i].contains("BLIND SPOT")&&instructions[i].contains("OBJECT DETECTED")){
            BlindSpot=1;
        }
    }
        JFrame f = new JFrame("A JFrame");
        f.setSize(500, 500);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final JTextArea textArea = new JTextArea(10, 40);
        JPanel buttonPanel = new JPanel(new GridLayout(2, 2));
        f.getContentPane().add(BorderLayout.CENTER, textArea);
        final JLabel label = new JLabel("DriverUI");
        label.setHorizontalAlignment(SwingConstants.CENTER);
        f.getContentPane().add(BorderLayout.NORTH, label);
        JPanel leftPanel = new JPanel();
        leftPanel.setLayout(new BorderLayout());
        //add values from instructions to left panel of statuses
        String[] sampleData = {"Car Status: ", "MPH: ", "Temperature: ", "Windshield Wiper: ", "Headlights: ","CC:
", "Assisted Reverse: "};
        if(CarStatus=="on"){
            sampleData[0]=sampleData[0]+"on";
            sampleData[1]=sampleData[1]+mph;
            sampleData[2]=sampleData[2]+temperature;
            sampleData[3]=sampleData[3]+WiperStatus;
            sampleData[4]=sampleData[4]+Headlights;
            sampleData[5]=sampleData[5]+CC;
            sampleData[6]=sampleData[6]+Reverse;

        }else{
            sampleData[0]=sampleData[0]+"off";
            sampleData[1]=sampleData[1]+"0";
            sampleData[2]=sampleData[2]+temperature;
            sampleData[3]=sampleData[3]+"off";
            sampleData[4]=sampleData[4]+"off";
            sampleData[5]=sampleData[5]+CC;
```

```
            sampleData[6]=sampleData[6]+Reverse;
        }
      // Add the JList to the left panel
        JList<String> list = new JList<>(sampleData);
        JScrollPane scrollPane = new JScrollPane(list);
        leftPanel.add(scrollPane, BorderLayout.CENTER);
      // Add the left panel to the JFrame on the WEST side
        f.getContentPane().add(leftPanel, BorderLayout.WEST);
        final JButton request_cruise_control = new JButton("Request Cruise Control");
        final JButton request_automated_park = new JButton("Request Automated Park");
        final JButton request_selfdrive_termination = new JButton("Request Self Driving Termination");
        final JButton request_assisted_reverse = new JButton("Request Assisted Reverse");
        final JButton request_headlights= new JButton("Request Headlights");
        JPanel headlightsPanel = new JPanel();
        headlightsPanel.add(request_headlights);
        buttonPanel.add(request_cruise_control);
        buttonPanel.add(request_automated_park);
        buttonPanel.add(request_selfdrive_termination);
        buttonPanel.add(request_assisted_reverse);
        headlightsPanel.add(request_headlights);
        JPanel southPanel = new JPanel(new BorderLayout());
        southPanel.add(buttonPanel, BorderLayout.CENTER);
        southPanel.add(headlightsPanel, BorderLayout.SOUTH);
        f.getContentPane().add(BorderLayout.SOUTH, southPanel);
//cruise control request button
        request_cruise_control.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                Object[] options = {"On",
                        "Off"};
                int n = JOptionPane.showOptionDialog(f,
                        "Would you like to request "
                                + "cruise control?",
                        "Cruise Control",
                        JOptionPane.YES_NO_CANCEL_OPTION,
                        JOptionPane.QUESTION_MESSAGE,
                        null,
                        options,
                        options[1]);
                if (n == JOptionPane.YES_OPTION) {
                    writeRequestToFile("Cruise Control ON");
                }
                if (n == JOptionPane.NO_OPTION) {
                    writeRequestToFile("Cruise Control OFF");
                }
            }
        });
//assisted park request button
        request_automated_park.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                Object[] options = {"Yes", "No"};
                int n = JOptionPane.showOptionDialog(f,
                        "Would you like to request " + "automated parking?",
                        "Automated Park",
                        JOptionPane.YES_NO_CANCEL_OPTION,
```

```
                JOptionPane.QUESTION_MESSAGE,
                null,
                options,
                options[1]);
            if (n == JOptionPane.YES_OPTION) {
                writeRequestToFile("Automated Parking");
            }
        }
    });
//self drive termination button
 request_selfdrive_termination.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        Object[] options = {"Yes", "No"};
        int n = JOptionPane.showOptionDialog(f,
                "Would you like to request " + "self driving termination?",
                "Self Driving Termination",
                JOptionPane.YES_NO_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                null,
                options,
                options[1]);
        if (n == JOptionPane.YES_OPTION) {
            writeRequestToFile("Self Driving Termination");

        }
    }
 });
 //Assisted Reverse request button
 request_assisted_reverse.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        Object[] options = {"On", "Off"};
        int n = JOptionPane.showOptionDialog(f,
                "Would you like to request " + "assisted reversing?",
                "Assisted Reverse",
                JOptionPane.YES_NO_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                null,
                options,
                options[1]);
        if (n == JOptionPane.YES_OPTION) {
            writeRequestToFile("Assisted Reversing ON");
        }if(n==JOptionPane.NO_OPTION){
            writeRequestToFile("Assisted Reversing OFF");
        }
    }
 });
 //Headlights request button
 request_headlights.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        Object[] options = {"Normal", "Brights","Off"};
        int n = JOptionPane.showOptionDialog(f,
                "What type of headlights would you like to turn on?",
                "Headlights",
```

```
                JOptionPane.YES_NO_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                null,
                options,
                options[1]);
            if (n == JOptionPane.YES_OPTION) {
                writeRequestToFile("Headlights Short");
            }else if(n == JOptionPane.NO_OPTION){
                writeRequestToFile("Headlights Long");
            }else if (n == JOptionPane.CANCEL_OPTION) {
                writeRequestToFile("Headlights Off");
            }
        }
    });
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    if(ObjectDetected==1){
        showAlert("Object Near By, Take Precautions!","Object Detected");
    }
    if(BlindSpot==1){
        showAlert("Object in Blind Spot, Take Precautions!","Object Detected");
    }
        //alerts for driver
    String finalRequestCC = RequestCC;
    String finalAssistedRev= RequestReverse;
    String finalAutomatedPark= RequestPark;
    SwingUtilities.invokeLater(() -> { //Alerts driver about requests success or failure
    if(finalRequestCC.equals("Success")){
        JOptionPane.showMessageDialog(f, "Cruise Control Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
    }
    if(finalRequestCC.equals("Fail")){
        JOptionPane.showMessageDialog(f, "Cruise Control NOT Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
    }
    if(finalAssistedRev.equals("Success")){
            JOptionPane.showMessageDialog(f, "Assisted Reverse Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
        }
    if(finalAssistedRev.equals("Fail")){
            JOptionPane.showMessageDialog(f, "Assisted Reverse NOT Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
        }
    if(finalAutomatedPark.equals("Success")){
            JOptionPane.showMessageDialog(f, "Automated Parking Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
        }
    if(finalAutomatedPark.equals("Fail")){
            JOptionPane.showMessageDialog(f, "Automated Parking NOT Activated!", "Alert",
JOptionPane.WARNING_MESSAGE);
        }
    });
    try{new FileWriter("request.txt", false).close();}catch(IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
```

```
    }
    //helper for request done by driver
  private void writeRequestToFile(String request) {
    try {
      FileWriter writer = new FileWriter("request.txt", true);
      writer.write(request + "\n");
      writer.close();
      System.out.println("Request added to file: " + request);
    } catch (IOException ex) {
      System.out.println("An error occurred while writing to file.");
      ex.printStackTrace();
    }
  }
  public static String[] queueToArray(Queue<String> queue) {
    // Create an array with the same size as the queue
    String[] array = new String[queue.size()];
    // Iterate over the queue and fill the array
    int index = 0;
    for (String str : queue) {
      array[index++] = str;
    }
    return array;
  }
  public static void showAlert(String message, String title) {
    JDialog dialog = new JDialog();
    dialog.setTitle(title);
    dialog.setModal(false); // Non-modal, so it won't block the UI
    dialog.setSize(300, 100);
    // Create a JLabel to display the message
    JLabel label = new JLabel(message, SwingConstants.CENTER);
    label.setFont(new Font("Arial", Font.PLAIN, 14));
    label.setForeground(Color.BLACK);
    dialog.add(label);
    // Set the location of the dialog to the center of the screen
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    int x = (screenSize.width - dialog.getWidth()) / 2;
    int y = (screenSize.height - dialog.getHeight()) / 2;
    dialog.setLocation(x, y);
    // Make the dialog visible
    dialog.setVisible(true);
    // Set a timer to close the dialog after 3 seconds (3000 milliseconds)
    int delay = 3000;
    ActionListener taskPerformer = new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        dialog.dispose(); // Close the dialog
      }
    };
    new Timer(delay, taskPerformer).start();
  }
  }
```

6.5 System Administration/ Technician Interface Module:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
import java.io.BufferedReader;
import java.io.FileReader;

public class SystemAdmin {
   public static class Login extends JFrame implements ActionListener {
      private JTextField username;
      private JPasswordField password;
      private JButton button;
      // Login Box
      public Login() {
         setTitle("Technician Login");
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         setSize(300, 150);
         setLayout(new GridLayout(3, 2));
         JLabel usernameField = new JLabel("Username:");
         add(usernameField);
         username = new JTextField();
         add(username);
         JLabel passwordField = new JLabel("Password:");
         add(passwordField);
         password = new JPasswordField();
         add(password);
         button = new JButton("Login");
         button.addActionListener(this);
         add(button);
         setVisible(true);
      }

      public void actionPerformed(ActionEvent e) {
         String user = username.getText();
         String pwd = new String(password.getPassword());
         if (user.equals("Joe") && pwd.equals("Shmoe")) {
            openTechnicianOverview();
         } else {
            JOptionPane.showMessageDialog(this, "Username or Password not correct");
         }
      }

      private void openTechnicianOverview() {
         dispose();
         JFrame overviewFrame = new JFrame("Technician Overview");
         overviewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         overviewFrame.setLayout(new BoxLayout(overviewFrame.getContentPane(), BoxLayout.Y_AXIS)); // Set
Y_AXIS layout
         JLabel welcomeLabel = new JLabel("Welcome! This is the Technician Overview page.");
         overviewFrame.add(welcomeLabel);
         displayLogFile(overviewFrame, "log/VCSlog.txt", "VCS Log");
         displayLogFile(overviewFrame, "log/PLANNINGlog.txt", "PLANNING Log");
         displayLogFile(overviewFrame, "log/SENSORlog.txt", "SENSOR Log");
         // Add buttons panel
         JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
         // Software Update
         JButton updateButton = new JButton("Software Update");
         updateButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                performSoftwareUpdate();
            }
        });
        buttonPanel.add(updateButton);
        // Logout
        JButton logoutButton = new JButton("Logout");
        logoutButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                logoutAndShowMessage();
            }
        });
        buttonPanel.add(logoutButton);
        overviewFrame.add(buttonPanel);
        overviewFrame.pack();
        overviewFrame.setLocationRelativeTo(null);
        overviewFrame.setVisible(true);
    }
    private void displayLogFile(JFrame frame, String filePath, String logName) {
        JPanel logPanel = new JPanel(new BorderLayout());
        JTextArea logTextArea = new JTextArea();
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                logTextArea.append(line + "\n");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            logTextArea.setText("Error reading log file: " + logName);
        }
        JScrollPane scrollPane = new JScrollPane(logTextArea);
        logPanel.add(new JLabel(logName), BorderLayout.NORTH);
        logPanel.add(scrollPane, BorderLayout.CENTER);

        frame.add(logPanel);
        frame.revalidate();
        frame.repaint();
    }
    private void performSoftwareUpdate() {
        JOptionPane updateDialog = new JOptionPane("Software Update running...",
JOptionPane.INFORMATION_MESSAGE);
        JDialog dialog = updateDialog.createDialog("Software Update");
        dialog.setModal(false);
        dialog.setVisible(true);
        Timer timer = new Timer(5000, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                updateDialog.setMessage("Software Updated!");
                ((Timer) e.getSource()).stop();
            }
        });
        timer.setRepeats(false);
        timer.start();
    }
    private void logoutAndShowMessage() {
        JOptionPane.showMessageDialog(null, "Logged out");
        System.exit(0);
    }
```

```
    }
    public void main() {
        SwingUtilities.invokeLater(Login::new);
    }
}
```

6.6 Main Module:

```
import java.util.ArrayList;
import java.util.Queue;
public class main {
        public static void main(String[] args) {
            //Test Case 1= Windshield Wiper
            //Test Case 2= Key Detection
            //Test Case 3= Traffic Light
            //Test Case 4= Car Alarm
            //Test Case 5= Cruise Control
            //Test Case 6= Automated Parking
            //Test Case 7= Assisted Reversing
            //Test Case 8= Object Avoidance
            //Test Case 9= Blind Spot Detection
            Sensor sensor = new Sensor();
            ArrayList<String> data = sensor.readFile(System.getProperty("user.dir")+"\\TestCase2.txt");
            Planning planning= new Planning(data);
            Queue<String> instructions= planning.getAllInstructions();
            VCS vcs= new VCS(instructions);
            vcs.parseInstructions();
            DriverUI driverUI= new DriverUI(instructions);
            driverUI.generateUI();
            SystemAdmin systemAdmin = new SystemAdmin();
            systemAdmin.main();
        }
}
```

7. Testing:

7.1 Requirement Testing:

7.1.1 Windshield Wiper Activation:

- Precondition: Car is on, humidity>=70
- Postcondition: Windshield wipers are turned on
- Input: *TestCase1.txt*

  Humidity: 80

  Car: ON

  Speed: 20

- Output: Driver UI displays Windshield Wipers as on,

  Actuator.txt

  Wiper STATUS: ON

7.1.2 Headlight Activation:

- Precondition: Car is on, driver requests low beam
- Postcondition: Low beam headlights are turned on
- Input: *Request1.txt*

  Car: ON

  Headlights short: on
- Output: Driver UI displays low beam headlights as on,

  Actuators.txt

  TURN ON: NORMAL HEADLIGHTS

7.1.3 Traffic Light Detection:

- Precondition: Car is on, driver detected in seat, traffic light is detected
- Postcondition: Car responds correctly to light
- Input: *TestCase3.txt*

  Car: ON

  Driver Detected

  Speed: 55

  Traffic Light: Red

  Traffic Light Distance: 150
- Output: *Actuators.txt*

  Apply Brake 10 percent

Apply Brake 20 percent

Apply Brake 50 percent

7.1.4 Entering Keyless:

- Precondition: key distance<=3
- Postcondition: Car is unlocked
- Input: *TestCase2.txt*

  Car: ON

  Key: 2

  Door Touched: driver
- Output: *Actuator.txt*

  UNLOCK: DRIVER DOOR

  CAR ALARM: OFF

7.1.5 Car Alarm Activation:

- Precondition: key distance<=3
- Postcondition: Car Alarm sounds, car doors lock
- Input: *TestCase4.txt*

  Car: ON

  Key: 2

  Panic Button: True
- Output: *Actuator.txt*

  TURN ON: CAR ALARM

  LOCK DOORS

7.1.6 Object Avoidance:

- Precondition: Car is on, driver is detected
- Postcondition: Object is Avoided
- Input: *TestCase8.txt*

  Car: ON

  Driver Detected

  Speed: 40

  Car Gear: Drive

  Number of Objects: 1

  Object Distance: 100

  Object Width: 4

  Object Height: 8

  Location: Ahead

  Lanes On The Left: 1

  Lanes On The Right: 0
- Output: Driver UI alerts driver of object. Alarm notification sound goes off,

  Actuator.txt

  Turn on: left blinker

  Adjust Steering: left direction

  Adjust Speed: Traffic Conditions

  Alarm: Sound

7.1.7 Self Driving System Termination:

- Precondition: Car is on
- Postcondition: Driver is in control of car
- Input: *Request.txt*

  Self Driving Termination
- Output: Driver UI updates with any self driving systems that were turned off such as cruise control.

  Actuators.txt

  TURN OFF: Self Driving

7.1.8 Blind Spot Detection:

- Precondition: Car is on, driver is detected
- Postcondition: Driver is notified of object in blind spot
- Input: *TestCase9.txt*

  Car: ON

  Driver Detected

  Speed: 40

  Car Gear: Drive

  Front Right Blind Spot: True

  Front Left Blind Spot: False

  Back Right Blind Spot: True

  Back Left Blind Spot: False

  Number of Objects: 1

  Object Distance: 100

Object Width: 4

Object Height: 8

- Output: Driver UI alerts driver of object. Alarm notification sound goes off

  Actuators.txt

  Object was Detected

  Apply Brake 10

  Apply Brake 20

  Apply Brake 50

  Alarm: Sound

  Object in: front right Blind Spot

  Alarm: Sound

  Object in: back right Blind Spot

7.1.9 Automated Parking:

- Precondition: Car is on, speed is less than 10 mph
- Postcondition: Car is parked
- Input: *TestCase6.txt*

  Car: ON

  Speed: 10

  Parking Space: 200

  Parking Space Direction: left

  Request.txt

  Automated Parking

- Output: *Actuators.txt*

Adjust Steering to : left

Reverse 180 ft

Adjust Steering opposite of: left

Forward 1ft

Engage Park

7.1.10 Cruise Control:

- Precondition: Car is on, speed is greater than 50
- Postcondition: Car safely engages cruise control
- Input: *TestCase5.txt*

  Car: ON

  Speed: 60

  Number of Objects: 1

  Request.txt

  Cruise Control ON
- Output: Driver UI displays CC as on,

  Actuators.txt

  Cruise Control= Engaged

  SET SPEED for CC: 60

7.1.11 Assisted Reversing:

- Precondition: Car is on, car gear is set to reverse, speed is less than 15mph
- Postcondition: Car safely reverses
- Input: *TestCase7.txt*

  Car: ON

Speed: 10

Car Gear: Reverse

Number of Objects: 0

Request.txt

Assisted Reversing ON

- Output: Driver UI displays Assisted Reversing as on,

Actuators.txt

Reverse 10 ft

7.2 Use Case Testing:

7.2.1 Entering Keyless:

- Precondition: key distance<=3
- Postcondition: Car is unlocked
- Input: *TestCase2.txt*

  Car: ON

  Key: 2

  Door Touched: driver
- Output: *Actuator.txt*

  UNLOCK: DRIVER DOOR

  CAR ALARM: OFF

7.2.2 Car Alarm Activation:

- Precondition: key distance<=3
- Postcondition: Car Alarm sounds, car doors lock
- Input: *TestCase4.txt*

Car: ON

Key: 2

Panic Button: True

- Output: *Actuator.txt*

  TURN ON: CAR ALARM

  LOCK DOORS

7.2.3 Software Update Implementation:

- Precondition: Technician logged in
- Postcondition: Software updated
- Input:

  username: “Joe”

  password: “Shmoe”

  Software Update Button: True

- Output: Dialog Box
  - “Software update running…”
  - “Software updated”

7.2.4 Object avoidance is activated by a sensor:

- Precondition: Car is on, driver is detected
- Postcondition: Object is Avoided
- Input: *TestCase8.txt*

  Car: ON

  Driver Detected

  Speed: 40

Car Gear: Drive

Number of Objects: 1

Object Distance: 100

Object Width: 4

Object Height: 8

Location: Ahead

Lanes On The Left: 1

Lanes On The Right: 0

- Output: Driver UI alerts driver of object. Alarm notification sound goes off,

  Actuator.txt

  Turn on: left blinker

  Adjust Steering: left direction

  Adjust Speed: Traffic Conditions

  Alarm: Sound

7.2.5 Automated Parking Activation by Driver:

- Precondition: Car is on, speed is less than 10 mph
- Postcondition: Car is parked
- Input: *TestCase6.txt*

  Car: ON

  Speed: 10

  Parking Space: 200

  Parking Space Direction: left

  Request.txt

Automated Parking

- Output: *Actuators.txt*

  Adjust Steering to : left

  Reverse 180 ft

  Adjust Steering opposite of: left

  Forward 1ft

  Engage Park