CEN 4020
Software Engineering
Team term project: Spring 2023

TEAM INFORMATION
Team name: Grant

USFIDs:
U60617877, U98428845, U91165060, U01798514, U34083131, U58222623, U21933376

E-mails: atashmore@usf.edu, emanueldejes@usf.edu, shehryarz@usf.edu, jbeyrer@usf.edu, maxaubuchon@usf.edu, dariyaalibi@usf.edu, katemakenzie@usf.edu

---

TEAM MEMBERS' CONTRIBUTIONS
Dariya Alibi
- Role: Coder
- Contribution*(s)*: System Testing, Car Sales Menu

Kate Anderson
- Role: Coder
- Contribution*(s)*: writeJson.py, Integration Testing

Emanuel Aseghehey
- Role: Coder
- Contribution(s): All of the code

Alexander Ashmore
- Role: Documentation
- Contribution(s): Project Description, Problem Boundaries, Requirement Analysis, System Design, Program Design, Coding, System Delivery, Maintenance, Abstract, Key Words, Introduction, Project Development

Max Aubuchon
- Role: Documentation
- Contribution(s): Unit Testing, Integration Testing, System Testing

Joseph Beyrer
- Role: Coder
- Contribution(s): Unit Testing, Demo video

Shehryar Zaihd
- Role: Documentation/Coder
- Contribution(s): Integration Testing

TABLE OF CONTENTS

COMPANY INFORMATION

Company Setup

- Name: PigeonBox
- Management: Democratic
- Product: Vehicle Tracking System

Company Visual

ABSTRACT

In this paper, we discuss the challenges faced in designing and implementing software systems. We also highlight the challenges that can arise from a lack of experience in working on a team for large-scale projects. This paper also provides an example of a project development that involved creating a vehicle tracking system for a car dealership. This system is called PigeonBox. We discuss the importance of promoting effective communication, creating a collaborative environment, and providing technical and management support to address the challenges in software system development.

KEY WORDS

Robust; Consistent; Thorough

INTRODUCTION

Designing and implementing a software system has several challenges that can arise during the development process. Developing the system requires careful planning, a thorough understanding of the requirements, and a structured approach to develop a functional, reliable, and scalable software system. These challenges include defining the scope of the software system, choosing appropriate technology, creating a robust architecture, designing a user-friendly interface, testing, and debugging, and maintaining the system after delivery.

Other challenges can result from a lack of experience in working in a team for a larger project. These include communication, collaboration, management, and technical issues. Many of the problems we faced during development were in the category of project management and communication. We suffered from the lack of set deadlines for individual components of this project. We started late and continued to have issues with time management due to other activities in team members' lives. During initial development, we should have set mini-deadlines to make sure we are keeping pace with the final project demonstration deadline and kept each other in check with these deadlines. A rushed environment also leads to miscommunication on project requirements. Better project management would have been necessary to avoid these communication and time management issues. While our team did not face issues with collaboration amongst team members, it did suffer from a lack of consistent collaboration.

It is essential to address these problems by promoting effective communication, creating a collaborative environment, and providing technical and management support.

PROJECT DEVELOPMENT

Our team decided to go with the default project given to us by the professor. This required less designing of the customer requirements since we were already provided specifications. This project entailed us developing a vehicle tracking system for a local car dealership that sells small to midsize sedans. This system should allow users to process customer orders and track the dealerships' vehicle inventory.

The first step in this project is outlining the problem boundaries. This includes determining how much our group can accomplish in the timeline allotted to us and limits that need to be set on the project based on this understanding. Our group ran into issues in starting the project due to a lack of time management skills. We started the project with 5 weeks left in the semester, causing us to put some serious limitations on the project. We opted for what we consider a thorough implementation of minimal requirements.

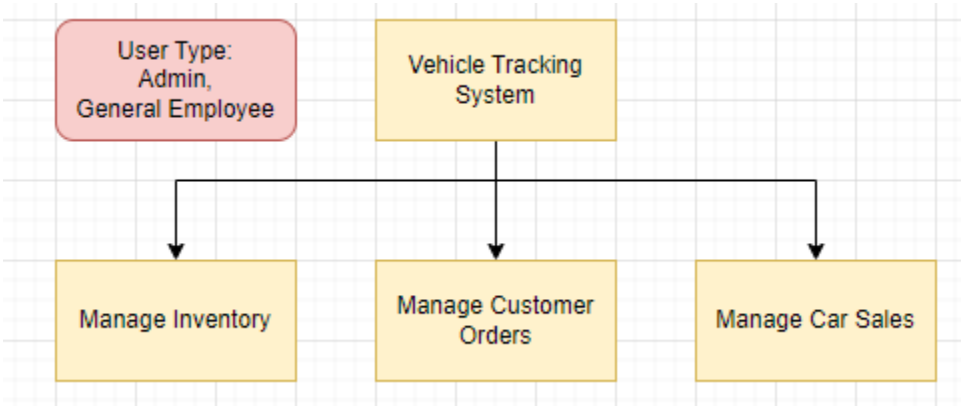
The product will be designed to be a command line interface system. This means that the project is strictly text-based without any images. While an ideal implementation would include images to view cars during the ordering process this would require either preexisting knowledge of how to implement or more time. Our team of 7 had neither so we stuck with a text-based interface since we can still develop the project this way and hit all the requirements. We also limited who can use the system to one interface at a time, meaning only one employee at the dealership can manage the inventory or place customer orders at a time. This is not the best implementation, but implementing a database that multiple users can log in at once will most likely result in failure. No team members have experience in this field, so to avoid missing the deadline we started with the initial goal of only having one person able to work on the system. Our group decided that if time allows we may eventually make it so multiple users can log onto the system, but this is hopeful.

After establishing our constraints on the system, our group discussed who in the dealership will be interacting with the system. To make it easier, we made it so that only employees of the dealership can interact with the system. This means if a customer wants to make an order, they cannot do so alone and must work with an employee to make an order, much like many existing dealerships. This ensures the salesperson can try to get the most out of the customer. We then went over each user's characteristics and tried to understand what they'll need to do. We split the users into two groups: admins and general employees. Admins are users that have more "managerial" permissions while still having general employee functions. While there are several departments in a car dealership, overlap in functions may occur. So for our minimal implementation, we avoided making too many user types that were very niche to only one module like customer service only having access to managing customer information but nothing else in the system.

The next step we undertook was what we assumed the system would be like and its environment. We assumed the system will only be operational at one dealership, used only by dealership staff, the dealership follows Florida law for Florida Dealership Licensing, and so forth. These assumptions are important for establishing requirements for the product and how we would implement it. This section was not done initially and we had to keep coming back to it during the development process as we ran into more problems with logic and how things operated.

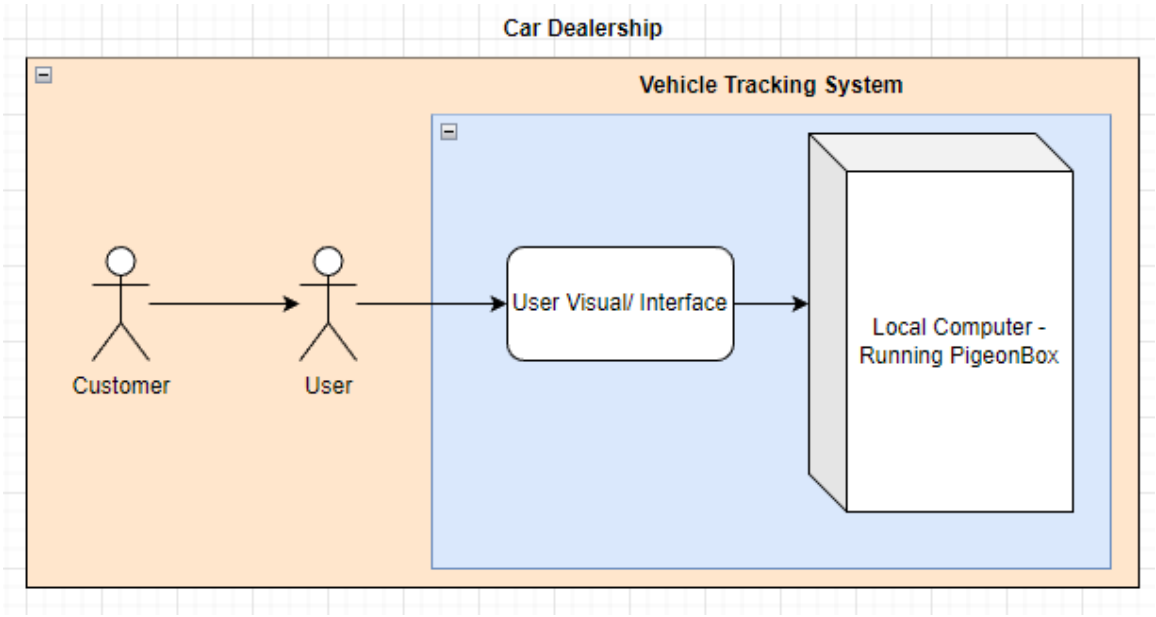
The next step was establishing requirements for the system. We have requirements set by the car dealership, but these are not thorough and to have a complete system we must add our own. The dealership required us to be able to search for cars in the inventory, add and delete cars from the inventory, process customer orders, manage car sales, and maintain a history for each car sold. This only covers the baseline and we had to add other requirements to get the system functioning. This includes managing customer information, employee information, account information for security purposes, modifying car details such as status, and so forth. The functional requirements of the system were discussed and then evaluated based on the priority of implementation. We then moved on to non-functional requirements. This includes the user's interface, product requirements, external requirements, and dealership requirements. The user interface covers a general understanding of how users will view the product like different color values for different messages, or how well the user understands what to do next. Product requirements include establishing how learnable the system is, its performance requirements, and system requirements. The system has very low standards, only requiring Windows 10 or later and 1 GB of RAM. The product will work on MAC but not as an executable and only through an IDE. External requirements deal with how secure the system is. Encrypting the passwords seemed unnecessary for us because the system will be stored locally in the dealership, disconnected from any outside networks. It is only accessed by dealership employees, and these employees can only get their account from an admin who creates it for them, giving them the information afterward.

The next step was system design. This section should be the most thorough in the document to make sure that the requirements are fully understood and the system is fully understood. Lack of development in this area will cause issues down the road with how things should be implemented so, unfortunately, I made sure to create this section thoroughly. We start by rating our understanding of each requirement out of 5, 1 is not understood, and 5 is fully understood. The minimum rating to start implementing is a rating of 3. Each requirement was evaluated by team members and altered until it was at least a score of 3 before we started implementing the system. The below information in the section 'System Design' goes over each requirement we discussed earlier. We discuss the specifications of the requirements thoroughly so we make sure the requirements are fully understood before undertaking implementation.
We then moved on to system decomposition, breaking the system into smaller modules. During this process, we ended up using function decomposition, breaking the system into functional components that worked together and made sense. Essentially we broke up the functions into three main functions and three supporting functions, each with its own function.



The image above shows the system is broken into 'manage inventory', 'manage customer orders', and 'manage car sales'. Manage inventory allows users to view all cars in the inventory, add and remove cars, and make custom orders. Manage customer orders allows users to view details of the orders, process the order to its final stage, and also make an order. Manage car sales allows users to see cars that have the status of Delivered, meaning the order process is complete and it is in the possession of the customer. Supporting functions include managing customers, managing employees, and managing account. General employees have access to all functions in the system excluding adding and removing cars, and managing employees. These are reserved for admins.

Next is developing the system's architecture. We followed an object-oriented design structure so that our product was flexible and maintainable. In the car dealership, the system is stored locally on a computer, there is one user interface, the user interacts on this interface, and the customer must interact through the user to interact with the system, such as making an order. We did not use a database and stored the data as JSON objects in a JSON file.



We used GitHub as our repository so we can easily collaborate and maintain version histories, keeping consistent. We used Python as an interpreter for an easier language model as well. Our team then moved on to understanding each user scenario. This includes each successful use of that function and possible failures of that function. For example, we'll have a scenario for a user logging in successfully and a description of a scenario of users failing to log in. This makes sure we are thorough with implementation, covering all possible scenarios users will find themselves in.

We then went over the use cases of the system. This includes what functions each user has access to, the description of the use case, and the post and pre-conditions of that use case. This is very similar to user scenarios but is more in-depth and includes activity diagrams showing how the user will flow through the system.

The next step of system design is developing the independent components, which in our case ends up being every component because we broke each function into their most simple form. We tried to design the system to have low coupling and high cohesion, but we ended up with both low coupling and cohesion, which isn't the worst case. The goal here is to make the system robust and not reliant on other functions during implementation so a failure of one would not fail another.

The last part of system design is designing the user interface. For our group, we only had to decide on a color scheme that was consistent and easily recognizable with colors and the order of information. This means using red for error messages, blue for user options, white for general text, purple for menu names, and so forth. The goal is to be consistent so users can easily distinguish between types of texts and where to generally look for things to continue using the system.

The next step is program design which is establishing some simple standards and styles for the project. Our team decided to include a header for every file and provide a one-line description for every function for quick browsing. Each function will also have a general description, accessibility, calling examples, and function prototypes in a separate file called documentation_functionDescription.txt. Otherwise, the header block becomes disgustingly large. We then established the data structures we'll end up using, which for us were only dictionaries and lists,

easily used through Python's library. Any significant algorithms used related to checking if cars, employees, or customers already exist in the system.

The next step is coding the project. It was implemented using Python since the main developer was most familiar with this language. General standards and styles were implemented to make sure the code was being implemented properly, including object-oriented design and code reviews. As a document team member, I never participated once in coding the project but I often participated in a code review process as well as general raw testing. These reviews were incredibly important in making sure the project was progressing smoothly and we were staying consistent with the established standards.

After the project is fully functional, we move on to unit testing where we make sure the code is working as intended, find any errors that may pop up, and make sure the code is robust. Python has a testing module that is easily imported, so we used this as a baseline for testing. We made sure to find all bugs and develop some testing examples. This includes placing ourselves in the shoes of the customer, and dealership employees, and trying to encounter errors during any processes they may complete.

Next is integration testing which involves seeing how well the system works as a complete unit. We decided upon sandwich integration which involves testing both top-level and low-level modules. This is possible because of how we designed the system's architecture, limiting the amount of coupling between functions, and increasing reliability. Next is system testing which involves stress testing, volume testing, configuration testing, and time testing. The goal was to make the system reliable enough to work in extreme cases that customers will probably never encounter. For example, as a car dealership, there is a limit to the number of cars that can be held in the system to sell due to space, however, we must still test in cases with an abnormal amount of cars stored on the system. System testing is incredibly important to make sure the system is robust and works consistently for the customer.

Next is system delivery which includes simple guides for installation, users, operators, and programmers. These outline steps on how to use the system for each category of persons. Users should be able to fully understand how to operate the system based on these guides. The final step is maintenance for the project. While this section feels unnecessary for our project because it is hypothetical, this is incredibly important in the real world. The goal of our system was to create something that needed few adaptive maintenance techniques and would only fix short-term problems. In this scenario, we are a small company so long-term maintenance is not ideal for us. It is better to make a robust and reliable system that will work for the car dealership for many years. Initially, we should plan to fix any errors and suggestions from the customers, but we should not have to make any large adaptive adjustments like moving the system fully online, excluding any employee interaction, and being fully interactive with customers. This defeats the whole development process.

This now marks the end of the software development life cycle and our attempt to develop a robust software system. We appreciate the lectures you oversaw and the time spent on this project and thank you for your professorship. By now we have at least an inkling of understanding of this process and what a pain each step is to make sure the system is robust. It is a long and arduous process that requires many steps back to make sure you are being consistent with the policies you placed on yourself.

PROBLEM BOUNDARIES

Project Description

Our company, Pigeon Box, is to develop a system that keeps track of orders and inventory of a car dealership dealing with small and mid-sized sedans. This vehicle tracking system (VTS) should allow the dealership to view and search cars in their inventory, add and remove cars from the inventory, manage customer orders, and maintain the history of cars sold. Priority is placed on the system's ability to manage the cars in the system rather than creating and processing new orders for customers. Since available vehicles are on site at the dealership, customers can view these vehicles in person instead of through the screen.

User Characteristics

There are several types of departments that are involved in using a VTS in a dealership, including sales and marketing, finance, service, HR, customer service, and parts departments. However, there are many forms of overlap between these departments and the functionality of the vehicle tracking system is minimal, so users are split into two types: general employee and admin. Admin users have all the functions of a general employee plus what is viewed as important “management” decisions, such as removing cars from the dealership or keeping track of employee data.

Below are how the users interact with the system:

1. General employee:
    - Can log into their account
    - Can read their personal information
    - Can change their account password
    - Can change their account username
    - Can view cars in the dealership’s inventory
    - Can search for cars in the dealership’s inventory
    - Can view the status of the cars in the inventory
    - Can view the details of available cars
    - Can view the details of ordered cars
    - Can view the details of sold cars
    - Can create a new order
    - Can create a new customer
    - Can view customer details, excluding credit card number
    - Can edit customer details
    - Can create a new backorder
    - Can remove an order
    - Can log out
2. Admin:
    - Can view employee details
    - Can grant admin privilege
    - Can add employees
    - Can remove employees
    - Can add cars to dealership inventory
    - Can remove cars from dealership inventory
    - Can log into their account
    - Can read their personal information
    - Can change their account password
    - Can change their account username
    - Can view cars in the dealership’s inventory
    - Can search for cars in the dealership’s inventory
    - Can view the status of the cars in the inventory
    - Can view the details of available cars

- Can view the details of ordered cars
- Can view the details of sold cars
- Can create a new order
- Can create a new customer
- Can view customer details, excluding credit card number
- Can edit customer details
- Can create a new backorder
- Can remove an order
- Can log out

Assumptions

- It is assumed that the dealership is asking for a vehicle inventory tracking system for only one physical location.
- It is assumed that the software will only be used by dealership staff, therefore validation and security will not be a key point of the software.
- It is assumed that the dealership does not require a way for customers to order cars directly without an employee/sales person. When a customer is making an order, it will be an employee filling out the information like a consultation.
- It is assumed that the function to add and remove cars from the inventory is reserved for people in higher up positions like “manager” or an overseer type. Meaning a general employee cannot delete or add a car into the dealership’s inventory.
- It is assumed that the system will be required to handle different car makes (ie. Toyota, Hyunadi, etc).
- It is assumed that the user logged into the computer is the person handling any orders made during that login.
- It is assumed the dealership only holds small and mid-sized sedans.
- It is assumed that vehicles labeled as available are on-site.
- It is assumed that vehicles labeled as ordered are in the process of being ordered and will be delivered to the customer’s home address in due time.
- It is assumed that when vehicles are labeled as delivered, the car has finished the order process and is in the possession of the customer, delivered at their registered home address.
- It is assumed that cars labeled as backordered are not currently on-site of the dealership.
- It is assumed that the dealership adheres to Florida law for Florida Dealership Licensing and is certified to sell cars.
- It is assumed that users are familiar with the English language.

Constraints

Many if not all the project constraints are based on the initial timeline outlined by the project description, 14 weeks. However, this project is further limited to 5 weeks of implementation as we started working with only 5 weeks remaining before the project demo.

The following constraints are based on this 5 week timeline:

1. Text based interface:
    - No team members have any prior experience with implementing a graphical user interface. Issues that can come up during inexperienced implementation may result in a functionally incomplete project. While it is ideal to have some form of graphical interface to view cars, the product requirements can still be met with a text based interface.
2. No car images
    - As it is a text based interface, users will not be able to visually view selected cars. While it would be nice to view during the ordering process, it is not absolutely necessary as priority is placed on managing the car inventory rather than placing new orders.
3. Data storage
    - Data is stored as JSON objects on a local drive. Implementing the project to run on a database can have issues during implementation in terms of compatibility and data integrity. Our unfamiliarity and time constraint led us to avoid these issues by having the data and program stored on a local device.
4. One user
    - Because the data is stored on a local device, only one user is able to use the VTS at a time. That means only one person in the dealership is able to view orders, process orders, and so forth.

REQUIREMENT ANALYSIS

Functional Requirements

Priority Definitions

- Priority 1 - The requirement is a “must have” outlined by the dealership.
- Priority 2 - The requirement is needed for improved processing or to handle requirements labeled as priority 1.
- Priority 3 - The requirement would be “nice to have” for better functionality.

Req #	Requirement	Summary	Priority
FR_1	Handle multiple account types	The functions of admin and general employee will have slight variations based on what they should be “allowed” to do	2
FR_2	Search for cars in the inventory	Filter the search by criteria like make, model, year, etc and receive a list of cars in the inventory that meet that criteria	1
FR_3	Add vehicles to inventory	Admin can add new cars to the inventory	1
FR_4	Delete vehicles from inventory	Admin can delete existing cars from the inventory.	1
FR_5	Process customer orders	Maintain a list of cars on order, cars on backorder, and cars recently delivered.	1

Req #	Requirement	Summary	Priority
FR_6	Manage car sales	The data of the vehicle sold, the data of the employee who sold the car, and the data of the customer it is sold to	1
FR_7	Maintain a history for each car sold	For each car sold there should be data for initial delivery.	1
FR_8	Edit customer data	The dealership needs to be able to make any changes to customer data if necessary ie: change address, email, credit card number	2
FR_9	View customer data	The dealership should be able to view customer data excluding credit card number	2
FR_10	Edit employee data	Employees should be able to change their own username and password	3
FR_11	View employee data	Employees can only view their own user information while admins can view all employee user information	3
FR_12	Create new customer	New customers should be able to be added into the system	2
FR_13	Delete existing customer	Admin is able to delete existing customers	3
FR_14	Create new employee	New employees should be able to be added into the system	2
FR_15	Delete existing employee	Admin is able to delete existing employees	2
FR_16	Manually change car status	When a car is processed, it goes from available to ordered to delivered once it reaches the customer. The dealership should be able to manually change this status from ordered to delivered.	2
FR_17	Log out and log in	Users should be able to switch login accounts from the main menu	2
FR_18	Different employee function access	Multiple departments are involved in using the system, but only two types of access need to be granted; a general employee and an admin. Only one type of employee access is necessary since crossover can occur but department specific functionality would be ideal	3
FR_19	Different text colors	Since it is text based, different colors will help improve readability for different types of messages	3

Req #	Requirement	Summary	Priority
FR_20	Create customer order	User should be able to process an available car in the inventory as a customer order	1

Non-Functional Requirements

User Interface Design Requirements

- The user interface will be compatible with any Windows OS.
- Initially, once launching the application, users will be prompted to either login or shut down the application. Once the user is prompted and enters their username and password, they will proceed to their appropriate view.
- The system user should be aware of what to do next.
- The screen should be made so the information appears in the same general format.
- Error messages, confirmation messages, and general information messages will use different colors to differentiate from each other.
- Default values for fields and answers to be entered by the user should be specified.
- The user should never get a fatal error.

Product Requirements

- Learnability
    - Admins and general employees should be able to master the system within an hour
    - Users should only need brief training to know where everything is in the system, but the system should be straight forward enough that anything past that is unnecessary
    - User and operating training will be available in this document
    - In the case of an error, a specific and detailed message will notify the user so they understand what went wrong
    - The user will be responsible for their own actions

- Accessibility
    - It is assumed that an existing employee provides newly created employees with their username and password when hired. These employees can then change their username and password once logging in
    - Only admins and general employees (employees of the car dealership) can access the system
    - Customers cannot directly access the system through an account of their own and must access through proxy with a car salesman

- Performance
    - The system is designed to work on a single terminal for the car dealership
    - This terminal can only be accessed by one user at a time
    - Future updates might include the ability to add more terminals to the system so that multiple users can interact at the same time
    - Performance is dependent on the hardware components of the client.

- Efficiency
    - Each operation will be fast and occur in real time
    - Once learning the system, the users should be able to operate any operation within minutes

- Memorability
    - The system should be intuitive so “vaguely remembering” where something is should not be an issue
    - User interfaces have minimal design so that everything that needs to be accessed and seen is accessible

- Satisfaction
    - The system should be easy to use

- System
    - Visual Studio Code 2019 or later
    - Windows 10 or later
    - System will work on Mac through an IDE but the executable will not.
    - 1GB of RAM or more

External Requirements

- Protection
    - The system will validate each input data for special characters and other specific conditions before inserting them into the system
    - Critical actions like removing employees, customers, and vehicles from the system need to be checked that they are intentional and not accidental. A confirmation popup will be used to confirm these types of actions
    - The system will not include a register function on the login screen and new users can only be created by existing admins

- Authorization and Authentication
    - The user authentication will use username and password
    - Authorization will be based on the user type: admin or general employee. Each user will have access to their respective information
    - If the user attempts to log in with the wrong credentials a message will be shown to them and after three attempts the login process will restart

Dealership Requirements

The main purpose of the VTS is to keep track of cars in the local car dealership inventory as well as process customer orders. There is sensitive customer data held in this application, that being email address, home address, and credit card number, so the application should only be accessible to users that have an account (employees of the dealership). Even customers do not have direct access to the application and must order cars via proxy of a car salesman. Therefore, it is assumed that the application is used by only the employees of the dealership and does not have any communication with other systems or networks.

- Availability
    - The application is able to be available 24 hours a day, every day. As long as power is supplied to the dealership, the application should be operable
    - Scheduled maintenance on the system should not affect functionality

- Florida Department of Highway Safety and Motor Vehicles (DHSMV)
    - It is assumed that employees hired by the dealership for sale of cars have submitted their electronic fingerprinting and have been approved by the Florida Department Law Enforcement.
    - It is assumed that employees hired by the dealership for sale of cars have not been found guilty of a felony
    - It is assumed that employees hired by the dealership for sale of cars will renew their Florida Motor Vehicle Dealer License
    - Based on these assumptions that adhere to the DHSMV requirements for dealing cars, the system will not record or update these values under employee data.

SYSTEM DESIGN

Requirement Analysis

Functional Requirement Evaluation

Below we measured the above requirements based on the designers and testers understanding, based on the scale 1 to 5 below:

1. The requirement is not understood, and a design/test can not be implemented.
2. Parts of the requirement are not understood properly and may affect the development of a good design/test.
3. The requirement has elements very different from previous work but is understood and a good design/test can be implemented.
4. The requirement has unexplored elements, but similar to previous designs/tests.
5. Complete understanding of the requirement, no problem developing a design/test.

Req #	Requirement	Summary	Understanding
FR_1	Handle multiple account types	The functions of admin and general employee will have slight variations	4
FR_2	Search for cars in the inventory	Filter the search by criteria like make, model, year, etc and receive a list of cars in the inventory that meet that criteria	5
FR_3	Add vehicles to inventory	Admin can add new cars to the inventory	5
FR_4	Delete vehicles from inventory	Admin can delete existing cars from the inventory.	5
FR_5	Process customer orders	Maintain a list of cars on order, cars on backorder, and cars recently delivered.	3
FR_6	Manage car sales	The data of the vehicle sold, the data of the employee who sold the car, and the data of the customer it is sold to	5
FR_7	Maintain a history for each car sold	For each car sold there should be data for initial delivery.	4
FR_8	Edit customer data	The dealership needs to be able to make any changes to customer data if necessary ie: change address, email, credit card number	5
FR_9	View customer data	The dealership should be able to view customer data excluding credit card number	5
FR_10	Edit employee data	Employees should be able to change their own username and password	5
FR_11	View employee data	Employees can only view their own user information while admins can view all employee user information	5

Req #	Requirement	Summary	Understanding
FR_12	Create new customer	New customers should be able to be added into the system	5
FR_13	Delete existing customer	Admin is able to delete existing customers	4
FR_14	Create new employee	New employees should be able to be added into the system	5
FR_15	Delete existing employee	Admin is able to delete existing employees	5
FR_16	Manually change car status	Car status should be able to be changed to backordered, delivered, ordered, available.	3
FR_17	Log out and log in	Users should be able to switch login accounts from the main menu	5
FR_18	Different employee function access	Multiple departments are involved in using the system, but only two types of access need to be granted; a general employee and an admin. Only one type of employee access is necessary since crossover can occur but department specific functionality would be ideal	5
FR_19	Different text colors	Since it is text based, different colors will help improve readability	4
FR_20	Create customer order	User should be able to process an available car in the inventory as a customer order	4

Functional Requirements Specification

Requirement #	FR_1
Name	Handle multiple account types
Summary	The functions of admin and general employee will have slight variations based on what they should be “allowed” to do in terms of needing a higher level of authority for that “decision.”

Specifications	The difference between admin functions and general employee functions will be based on the concept of admin being a managerial position. Functions that are considered to need a higher level of authority will be given to the admin and restricted to general employees. Admin will have all functions available to general employees plus their exclusive functions. Below are the functions exclusive to admin: Managing Employees: ● Create new employee ○ Creating new employees is exclusive to people in hiring positions, which is typically a managerial position. A general employee should not be allowed to hire new people and give access to the existing system which holds sensitive information such as customer credit card information. ● Delete existing employee ○ Deleting existing employees can result in data issues if performed incorrectly. When processing an order for customers, the name of the user is logged and stored as the person who ordered that car for the customer. If the employee information is deleted, what happens to the car order data that stores the name of the employee that sold the car? Therefore, this function remains in control of admins and should only be used in specific use cases. ● Grant employee with admin privilege ○ Granting admin privilege is restricted to admins. ● View employee data ○ While general employees are allowed to see their own data such as username, admins are allowed to view all employee data in the system. They cannot change this data, but are able to view them. Managing Vehicles in inventory: ● Add new vehicles to inventory ○ Adding new vehicles into the inventory means adding an entirely new vehicle. This includes the make, model, year, mileage, interior, exterior, warranty and so forth. The preexisting vehicles in the inventory can be “added” again by making them a backorder, but general employees are not allowed to add an entirely new vehicle to the system. ● Remove existing vehicle from inventory ○ Removing vehicles means removing a vehicle labeled as ordered, available, or backordered. If a vehicle is removed during the process of making an order for a customer, that order is also removed since the vehicle is removed. Removal means removal from the dealership premises and is no longer available to be sold.

Requirement #	FR_2
Name	Search for cars in the inventory

Summary	Filter the search by criteria like make, model, year, etc and receive a list of cars in the inventory that meet that criteria
Specifications	Users should be able to search the dealerships' inventory by make, model, and year separated by commas. This search must include the car's make, model, and year and will not return a search result without all three. This is the simplest implementation that allows users to search through the inventory. Users will also be able to filter through the inventory for cars with different status. These statuses are Available, Delivered, Ordered, and Backorder. By choosing a filter option, users will only see cars in the inventory that have that status. Future updates may include a more in depth search that allows users to search for any attribute of a vehicle and see if it meets that criteria. For example, a user can search for a Camry with a premium package type.

Requirement #	FR_3
Name	Add vehicles to inventory
Summary	Admin can add new cars to the inventory
Specifications	This function is reserved for admin privileges. Adding a new vehicle means to add a vehicle that was not already in the system's inventory. Vehicles in the dealership work in a way that once the vehicle is placed in the dealership's inventory, that vehicle will either be available, in the process of being ordered, or put to backorder to make it available again. This means the same exact vehicle with the same attributes is brought back into the inventory. This is different from adding a new vehicle into the inventory with its own unique attributes. Each vehicle added needs values for: ● VIN: must be a new integer value not already in the system ● make: must be a string ● model: can be string or number ● year: must be a 4 digit number ● mileage: must be a number ● color: must be a string ● price: must be a number ● engine: must be a string ● transmission: must be a string ● interior: must be a string ● external design: must be a string ● handling: must be a string ● audio: must be a string ● comfort features: must be a string ● package: must be a string ● warranty: must be a string ● maintenance: must be a string

Requirement #	FR_4
Name	Delete vehicles from inventory
Summary	Admin can delete existing cars from the inventory.
Specifications	This function is reserved for admin privileges. Deleting vehicles from inventory removes the vehicle from every known linked location. If a vehicle is removed and a customer has an order placed on it, that means that order is canceled as well. Removing vehicles is a final decision that can involve canceling orders for customers so this requires a confirmation message and checker to see if it is ordered. If it is ordered, the user can still continue to delete the vehicle if they wish to do so.

Requirement #	FR_5
Name	Process customer orders
Summary	Maintain a list of cars on order, cars on backorder, and cars recently delivered.
Specifications	This is different from “creating a customer order.” Processing customer orders means to be able to view and manage the different stages of the ordering process: Available, Ordered, Backorder, Delivered. This requirement is met by allowing users to maintain a list of cars on order, backorder, and recently delivered. Available cars can be seen in the search function.

Requirement #	FR_6
Name	Manage car sales
Summary	The data of the vehicle sold, the data of the employee who sold the car, and the data of the customer it is sold to.
Specifications	Car sales refers to cars that have the status of Ordered or Delivered. Managing car sales means that for each order, the attributes of the vehicle sold such as make, model, year, mileage and so forth is stored. The first and last name of the employee who was logged in during the creation of the order is stored. The customer’s first and last name, address, email address, and credit card information is stored.

Requirement #	FR_7
Name	Maintain a history for each car sold
Summary	For each car sold there should be data for initial delivery.

Specifications	When a car fully completes the order process and transitions from Ordered to Delivered, the date that the car was changed to delivered should be stored as an attribute with that sale. For each car sold, users should be able to see all related information to that order: ● Initial delivery date ● User who made the sale ● Customer who bought the car ● Car attributes: make, model, year, milage, interior design, etc. Future updates may include implementing a maintenance tracker for these vehicles.

Requirement #	FR_8
Name	Edit customer data
Summary	The dealership needs to be able to make any changes to customer data if necessary ie: change address, email address, email, credit card number
Specifications	Users should be able to update the customer's data in case that there was a change in the information: ● home address: These values should have no special characters. ● Email address: This should be a valid email address. ● Credit card number The credit card number is not able to be viewed, but is editable. The credit card number must be 16 digits exact. These values should have no special characters.

Requirement #	FR_9
Name	View customer data
Summary	The dealership should be able to view customer data excluding credit card number
Specifications	Users should be able to view the customer's data below, excluding credit card information: ● First name: These values should have no special characters. ● last name: These values should have no special characters. ● home address: These values should have no special characters. ● email address: This should be a valid email address.

Requirement #	FR_10
Name	Edit employee data
Summary	Employees should be able to change their own username and password

Specifications	Users should be able to edit their own data: ● username ● password Users are not able to have the same username. If they enter a username that is already taken, they will be asked to enter a different username. Passwords will require confirmation to change. The user will be prompted to enter the new password twice. If they do not match, the process restarts. Passwords cannot be changed into the same password, they must be different.

Requirement #	FR_11
Name	View employee data
Summary	Employees can only view their own user information while admins can view all employee user information
Specifications	Employee view: ● Users are able to view their username, first name, password, and date the account was created. Admin views: ● Users are able to view their username, first name, password, and date the account was created. ● Users are able to see all other employees and admins first name, last name, and date joined.

Requirement #	FR_12
Name	Create new customer
Summary	New customers should be able to be added into the system
Specifications	Users should be able to create new customers in two areas. They can directly create new customers while managing customer data, or they will be prompted to create a new customer while making the order if the order is not for an existing customer. Both function in the same way and ask to fill out these details: ● First name: These values should have no special characters. ● Last name: These values should have no special characters. ● home address: These values should have no special characters. ● Email address: This should be a valid email address. ● Credit card number The credit card number is not able to be viewed, but is editable. The credit card number must be 16 digits exact. These values should have no special characters.

Requirement #	FR_13
Name	Delete existing customer
Summary	Admin is able to delete existing customers
Specifications	This function is reserved for admin privileges. The user can delete existing customers in the system. If a customer with car history is deleted, all the related information is also deleted. This means any orders created by the customer are deleted and any cars registered as Delivered to that customer are deleted. The cars ordered by the deleted customer will return to the inventory as Available. Deleting a customer will prompt a confirmation popup.

Requirement #	FR_14
Name	Create new employee
Summary	New employees should be able to be added into the system
Specifications	This function is reserved for admin privileges. The user must fill out these values as a temporary placeholder before giving it to the new employee. The new employee can log on with these credentials and change it themselves. The user has the choice to grant new employees with admin privileges. ● First name: These values should have no special characters. ● Last name: These values should have no special characters. ● username ● password ● Grant admin privileges?

Requirement #	FR_15
Name	Delete existing employee
Summary	Admin is able to delete existing employees

Specifications	This function is reserved for admin privileges. The user can delete existing employees in the system. If an employee has sold any vehicles, that employee account is not able to be deleted. This is because this would cause fatal errors in the system when trying to view the information for those orders or delivered cars and the data of the person who sold the car is not there. Instead, the admin can keep the account, but change the username and password once the employee is fired. If an employee is deleted but has not sold any cars, there are no issues and it can be deleted without worry. Deleting a customer will prompt a confirmation popup.

Requirement #	FR_16
Name	Manually change car status
Summary	Car status should be able to be changed to backordered, delivered, ordered, available.
Specifications	When a car is processed, it goes from Available to Ordered to Delivered once it reaches the customer. The dealership should be able to manually change this status from Ordered to Delivered as a confirmation check. The employees of the dealership will deliver the car to the customer's house and as a quality check, they will manually update the order status themselves. When a car is ordered when its status is Ordered, the user will be prompted to place the car on Backorder. This means to add the same vehicle (make, model, year) to the inventory again. Once the car arrives at the dealership, the employee can then manually change the status from Backordered to Available as a quality check.

Requirement #	FR_17
Name	Log out and log in
Summary	Users should be able to switch login accounts from the main menus
Specifications	From the main menu, users should be able to log out of their account. This will bring us to the startup page where there are options to login or shut down the system. The same user or other users are able to login at this time. When using the system, users are expected to use their own accounts and log out when done.

Requirement #	FR_18
Name	Different employee function access
Summary	Multiple departments are involved in using the system, but only two types of access need to be granted; a general employee and an admin. Only one type of

	employee access is necessary since crossover can occur but department specific functionality would be ideal
Specifications	A car dealership will have several types of departments that are involved in different things. For example, sales and marketing should only have access to the functions of searching the inventory and reviewing current orders. They don't need access to car history. Customer service may have access to customer data so that they can manage and edit this data if they have to, but they don't need to search or order new cars. Splitting access like this to multiple accounts may be included in a future update, but it is not necessary for the system to function properly.

Requirement #	FR_19
Name	Different text colors
Summary	Since it is text based, different colors will help improve readability for different types of messages
Specifications	Error messages, confirmation messages, prompts, and general information messages will use different colors to differentiate from each other. Error messages are red. Confirmation messages will use yellow. General information will use white. User prompts will be blue. Menu names will be purple.

Requirement #	FR_20
Name	Create customer order
Summary	User should be able to process an available car in the inventory as a customer order
Specifications	Users should be able process a car labeled as Available as a customer order. This means a customer is buying this car. This can be done either from going through the existing list of cars in the inventory or searching for a specific car. Users need to choose to order a car for a new customer or an existing customer. If it is an existing customer, the user should choose from a list of customers in the system. If it is a new customer, the user must fill out information in a similar format to the "create new customer" requirement. Once this step is done the car's status will change from Available to Ordered and can be found in the Car Sales menu.

System Decomposition

The system can be broken up into the following three main functionalities of managing the system's inventory, customer orders, and managing car orders' history by using functional decomposition. Each of these main functionalities can be further split up into subfunctions that help meet customer requirements. All of these functions must also be supported by supporting functions so that we hit all functional and nonfunctional requirements. Below will be the details of each function and afterwards illustrations to help deepen understanding.

Main Functionalities

Function #	MF_1
Function Name	Manage inventory
User Privilege	Admin, general employee
Summary	The purpose of this module is to satisfy customer functional requirements FR_2, FR_3, FR_4 which entail searching the inventory, adding cars, and removing cars. Users should be able to browse all cars in the system's inventory and filter by their status of Available, Backordered, Ordered, or Delivered. From this module, users should be able to be routed to MF_2, manage customer orders, and add a new order.
Subfunctions	Add vehicle Remove vehicle Search inventory View car details Filter by Add order

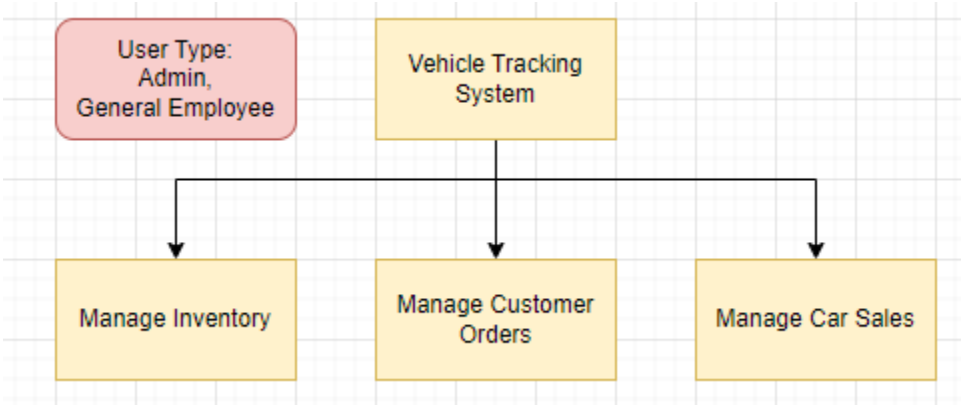
Function #	MF_2
Function Name	Manage customer orders
User Privilege	Admin, general employee
Summary	The purpose of this module is to satisfy customer functional requirements FR_5, FR_20 which entail processing customer orders and creating customer orders. Users should be able to view all ongoing order details, create new orders, and remove orders.
Subfunctions	View order details Remove order Add order

Function #	MF_3
Function Name	Manage car sales
User Privilege	Admin, general employee

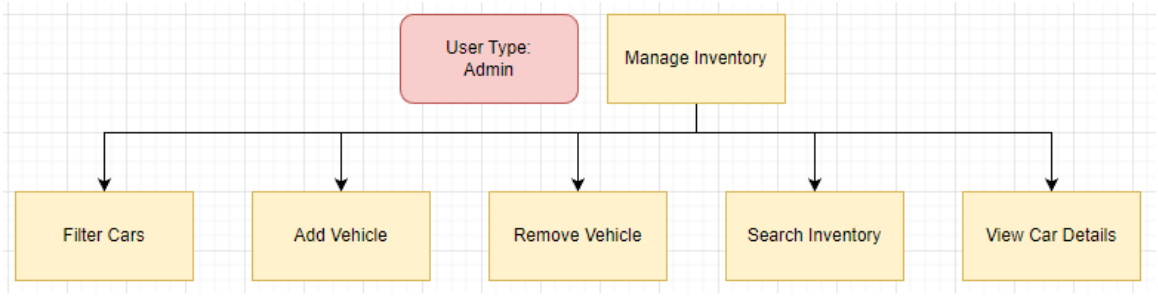
Summary	The purpose of this module is to satisfy customer functional requirements FR_6, FR_7 which are managing car sales and maintaining a history for each car sold. Cars marked as Delivered are considered completely done with the order process and are “sold” at that moment. Users should be able to view all cars marked as delivered and view details of that car sale history.
Subfunctions	View Car History

Main functions diagrams

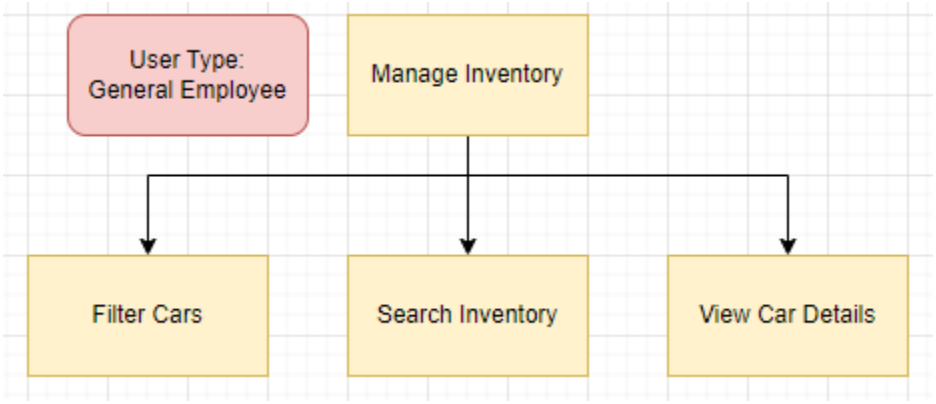
Main functions



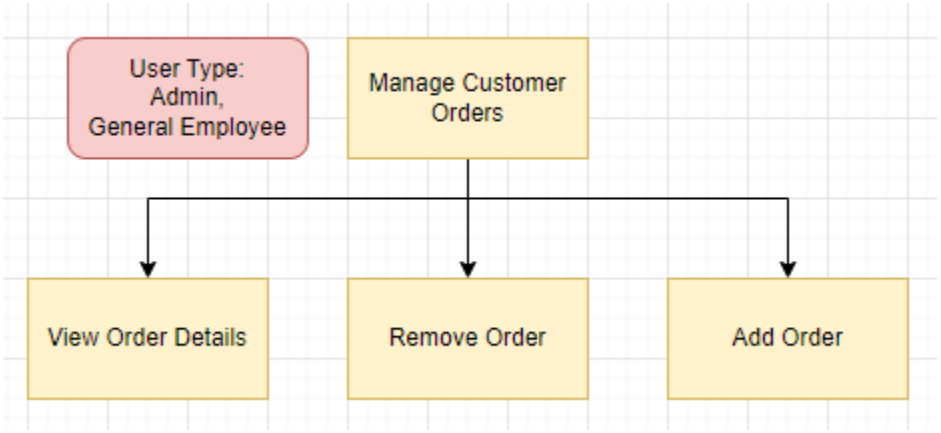
Manage Inventory - Admin



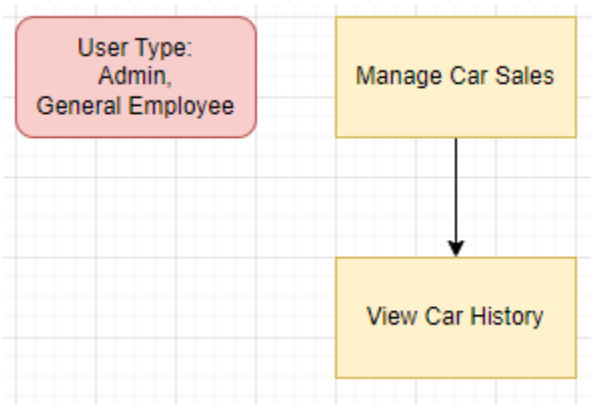
Manage Inventory - General Employee



Manage Customer Orders



Manage Car Sales



Subfunctions

Function #	SF_1
Function Name	Add vehicle
User Privilege	Admin
Summary	Users with admin privileges will be able to add new vehicles into the dealership's inventory by providing details for each of the attributes of the vehicle below. Each vehicle added needs values for: ● VIN: must be a new integer value not already in the system ● make: must be a string ● model: can be string or number ● year: must be a 4 digit number ● mileage: must be a number ● color: must be a string ● price: must be a number ● engine: must be a string ● transmission: must be a string ● interior: must be a string

	• external design: must be a string • handling: must be a string • audio: must be a string • comfort features: must be a string • package: must be a string • warranty: must be a string • maintenance: must be a string
Subfunctions	None

Function #	SF_2
Function Name	Remove vehicle
User Privilege	Admin
Summary	Users with admin privileges will be able to delete existing vehicles from the dealership's inventory. This will require confirmation.
Subfunctions	None

Function #	SF_3
Function Name	Search inventory
User Privilege	Admin, general employee
Summary	The purpose of this module is to browse through the car's inventory or search for cars that match criteria of make, model, year. Currently the prototype will have 50 cars in the inventory, but what if there are over 500. 50 is still small enough to browse, but if more are added in the future, we will want to be able to search quickly if that car is in the inventory.
Subfunctions	None

Function #	SF_4
Function Name	View order details
User Privilege	Admin, general employee
Summary	Users will be able to view the details of any customer order. These details include the salesperson's name, the customer's details (address, email, first name, last name), and the car details. User will be able to change the status of the order to Delivered.
Subfunctions	Change car status

Function #	SF_5
Function Name	Remove order
User Privilege	Admin, general employee
Summary	Users can remove ongoing orders from the inventory, canceling that car order and returning that car to the system's inventory as available. This will require confirmation.
Subfunctions	None

Function #	SF_6
Function Name	Add order
User Privilege	Admin, general employee
Summary	Users can create new orders for new customers or existing customers by selecting a vehicle through the search inventory module..
Subfunctions	None

Function #	SF_7
Function Name	View car history
User Privilege	Admin, general employee
Summary	Users will be able to see all details related to the vehicle with status as Delivered. This includes the sales's person's name, the customer's name and details, the car details, and the initial delivery date when the car arrived at the customer's address.
Subfunctions	None

Function #	SF_8
Function Name	Filter by
User Privilege	Admin, general employee
Summary	Users should be able to sort cars by their status: Available, Ordered, Backorder, Delivered and view them in a list.
Subfunctions	None

Function #	SF_9

Function Name	View car details
User Privilege	Admin, general employee
Summary	Users can select a car to view car details before ordering. Details include all attributes of that car from make and model to price.
Subfunctions	Change car status Change car price Change car mileage Change car warranty plans

Function #	SF_10
Function Name	Change car status
User Privilege	Admin, general employee
Summary	Users can change car status from Ordered to Delivered. Car then moves from car orders to car sales
Subfunctions	None

Function #	SF_11
Function Name	Change car price
User Privilege	Admin, general employee
Summary	Users can change the car's price. Must be a valid integer.
Subfunctions	None

Function #	SF_12
Function Name	Change car mileage
User Privilege	Admin, general employee
Summary	Users can change car mileage. Must be a valid integer.
Subfunctions	None

Function #	SF_13
Function Name	Change car warranty plans
User Privilege	Admin, general employee

Summary	Users can add to the car's current list of warranty plans. Users cannot remove warranties that came with the car.
Subfunctions	None

Supporting Functions

Function #	SUF_1
Function Name	Manage Employees
User Privilege	Admin
Summary	Users will be able to view all employees registered with the car dealership. This includes employees and admins. Users will be able to remove, add, and view details of these accounts.
Subfunctions	Remove employee Add employee View employee details

Function #	SUF_2
Function Name	Manage Customers
User Privilege	Admin, general employee
Summary	Users will be able to view all customers registered with the car dealership. Users will be able to view customer details, add new customers, and remove existing customers from the system.
Subfunctions	Remove customer Add customer View customer details

Function #	SUF_3
Function Name	Manage Account
User Privilege	Admin, general employee
Summary	Users will be able to view their account details and change their username and password.
Subfunctions	Change password Change username View account details

Supporting Sub-functions

Function #	SUSF_1
Function Name	Remove employee
User Privilege	Admin
Summary	Delete employee accounts from the system. If the employee has made any car sales they are not able to be deleted from the system. This will require confirmation.
Subfunctions	None

Function #	SUSF_2
Function Name	Add employee
User Privilege	Admin
Summary	Create new employee accounts to add to the system. User, the admin, will create a temporary username and password for the newly registered employee. Users have the choice to grant user admin privileges here.
Subfunctions	Grant admin privileges

Function #	SUSF_3
Function Name	View employee details
User Privilege	Admin
Summary	Users can view a list of all employees and admins on the system. Users can view the details of these employees, seeing first name, last name, username, and date joined.
Subfunctions	None

Function #	SUSF_4
Function Name	Remove customer
User Privilege	Admin, general employee
Summary	Users can remove existing customers from the system. This requires confirmation.
Subfunctions	None

Function #	SUSF_5

Function Name	Add customer
User Privilege	Admin, general employee
Summary	Users can create new customers on the system. They must fill out: ● First name: These values should have no special characters. ● Last name: These values should have no special characters. ● home address: These values should have no special characters. ● Email address: This should be a valid email address. ● Credit card number: The credit card number must be 16 digits exact.
Subfunctions	None

Function #	SUSF_6
Function Name	View customer details
User Privilege	Admin, general employee
Summary	Users can view the details of customers registered on the system. This includes the customers': ● First name: These values should have no special characters. ● Last name: These values should have no special characters. ● Home address: These values should have no special characters. ● Email address: This should be a valid email address. Users are also able to edit customer details from this point.
Subfunctions	Edit customer details

Function #	SUSF_7
Function Name	Change username
User Privilege	Admin, general employee
Summary	Users are able to change their username into a username not already registered with the system.
Subfunctions	None

Function #	SUSF_8
Function Name	Change password
User Privilege	Admin, general employee
Summary	Users are able to change their password. Users must type their password

	twice, matching, to confirm. This password cannot be the same as the current password.
Subfunctions	None

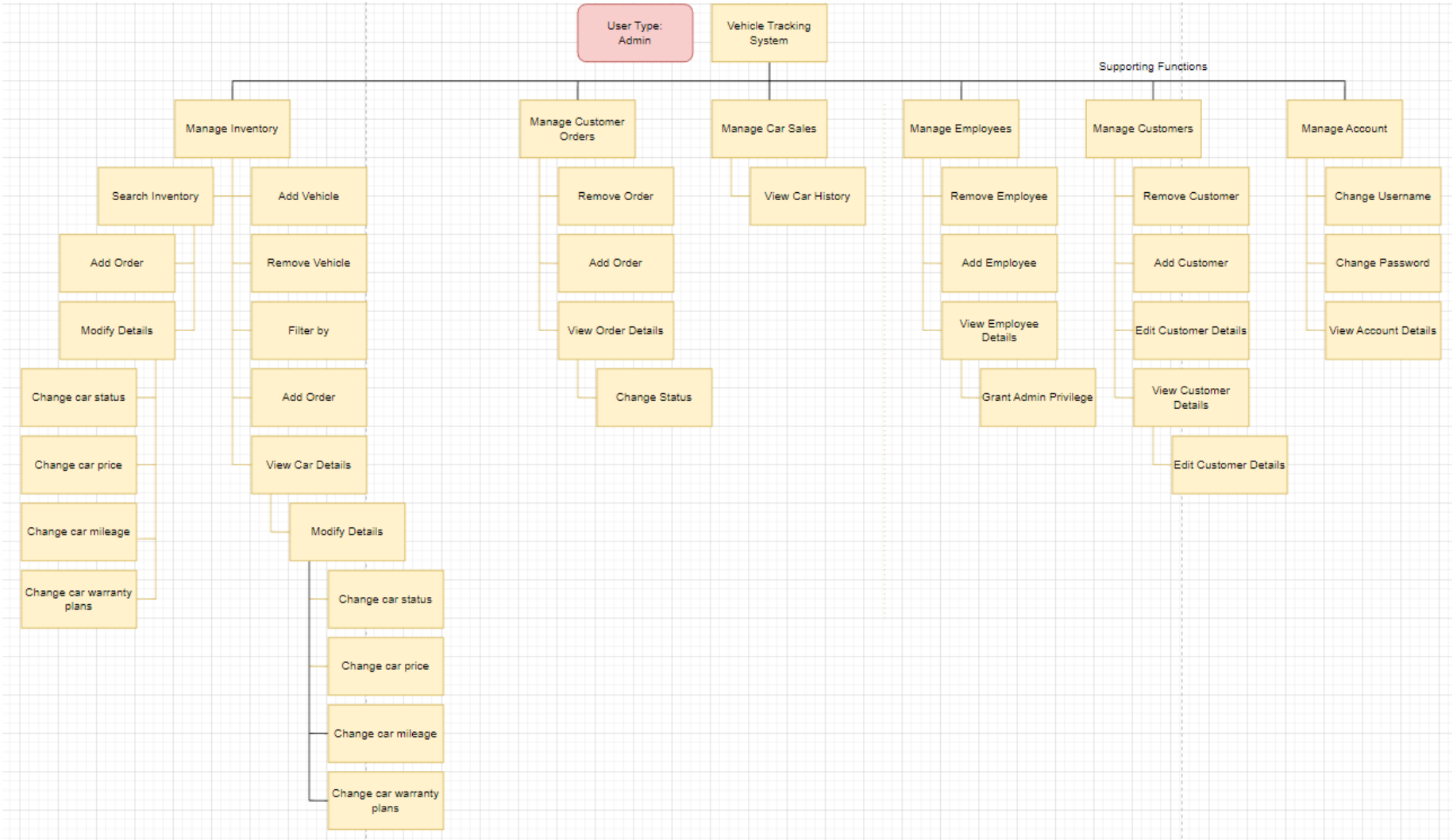
Function #	SUSF_9
Function Name	View account details
User Privilege	Admin, general employee
Summary	Users can see their username, first name, last name, and date joined.
Subfunctions	None

Function #	SUSF_10
Function Name	Grant admin privileges
User Privilege	Admin
Summary	Users can grant admin privileges to general employees.
Subfunctions	None

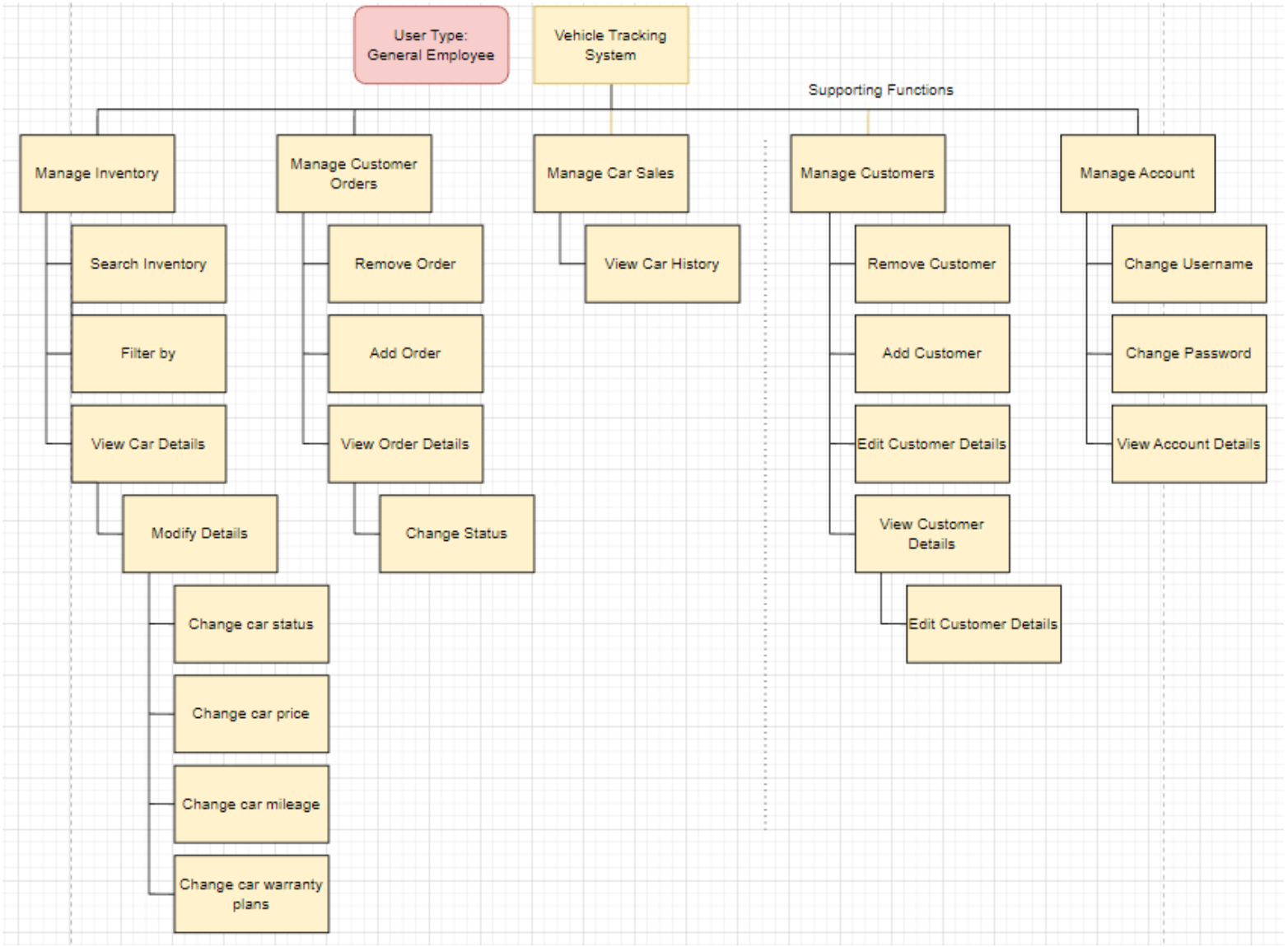
Function #	SUSF_11
Function Name	Edit customer details
User Privilege	Admin, general employee
Summary	Users can edit the customer's details: ● home address: These values should have no special characters. ● Email address: This should be a valid email address. ● Credit card number: The credit card number must be 16 digits exact.
Subfunctions	None

Total System Decomposition Diagrams

Admin



General Employee



System Architecture

Our system follows an object-oriented design architectural structure. The goal is to create a flexible and maintainable design so that future requirement changes are easy to adapt to. Users, the car dealership's employees, have direct access to the system through a monitor which connects to a dealership computer. Customers have access to the system through the car dealership employees. The system is split up into scripts that run PigeonBox, data that store the customer, inventory, orders, and user information, and parsers that alter and read this data.

Repository

Our team used GitHub as our repository to collaborate during this project. Since we have multiple developers working on the project GitHub allows us to work on the same codebase while keeping track of changes, managing conflicts, and ensuring consistency. GitHub also has a function in which developers can create issues, allowing us to track bugs and requirements in the code. Other developers can see these issues and task themselves with solving them.

Interpreter

Developers for this project preferred using Python as an interpreter so we opted for this rather than using other compiled languages. This allows development to proceed at a much faster pace than using something like C. Unfortunately we'll incur some slow code since it is not pre-optimized and increased use of memory. However, PigeonBox is a fairly small program that does not require loading onto a database to use, so there should not be any significant slow response times.

Abstraction

Scripts

1. bcolors.py
    a. Responsible for color messages for better formatting and readability.
2. interface.py
    a. Responsible for viewing certain interfaces like customer details or car details.
3. main.py
    a. Responsible for the majority of the functions users use to interact with the system: pick index, menus, modify details, etc.
4. orders.py
    a. Responsible for making orders working like a database with a one to one relationship.
5. session.py
    a. Responsible for dealing with authorization and session logic.
6. status.py
    a. Responsible for implementing vehicle status: Available, Ordered, BackOrdered, Delivered.
7. users.py
    a. Responsible for creating a user class that can get its values from users.json for Admins and General Employees. Customers are also defined here.
8. vehicles.py
    a. Responsible for creating a car class that can get its values from inventory.json.

Data

This module is used for storing customer details, car details, order details and user details:

1. customers.json
    a. This file stores attributes relating to each customer: email, first name, last name, credit card, home address.

2. inventory.json
    a. This file stores attributes relating to each car in the dealership's inventory. This includes the vehicle's VIN, status, info, performance, design, exterior, handling, comfort, audio, and protection.

3. orders.json
    a. This file stores attributes related to orders: order id, car vehicle identification number, customer who bought the car, user who sold the car, the day the car was ordered, and the day the car was delivered.

4. users.json
    a. Users are either Admins or General Employees. This file will store each user's username, password, first name, last name, date joined, and type being Adminor General Employee.

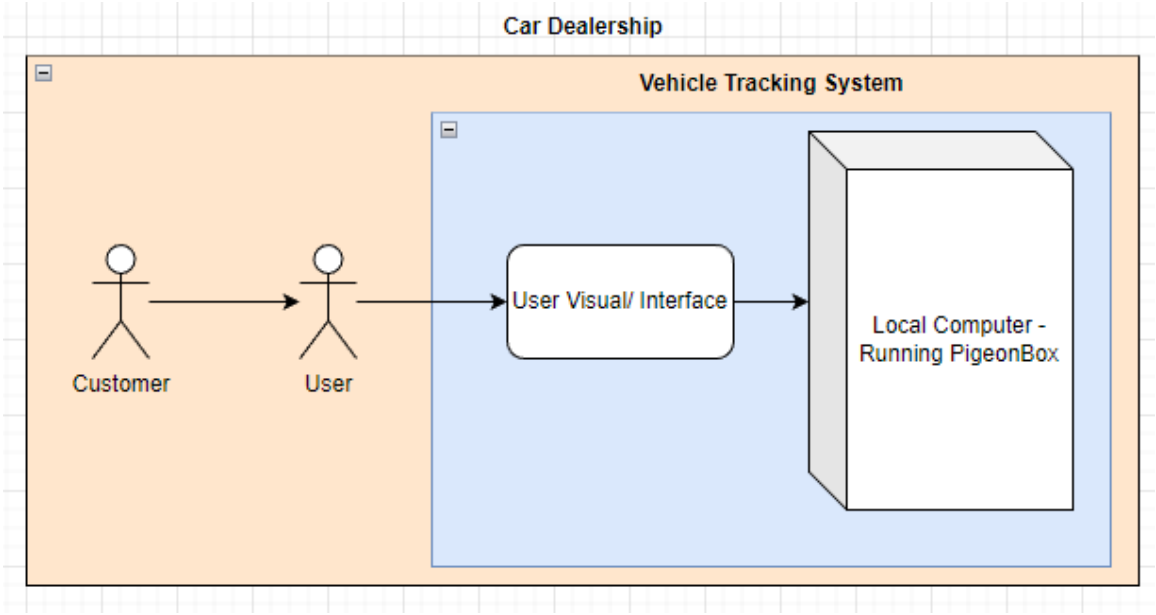
Parsers

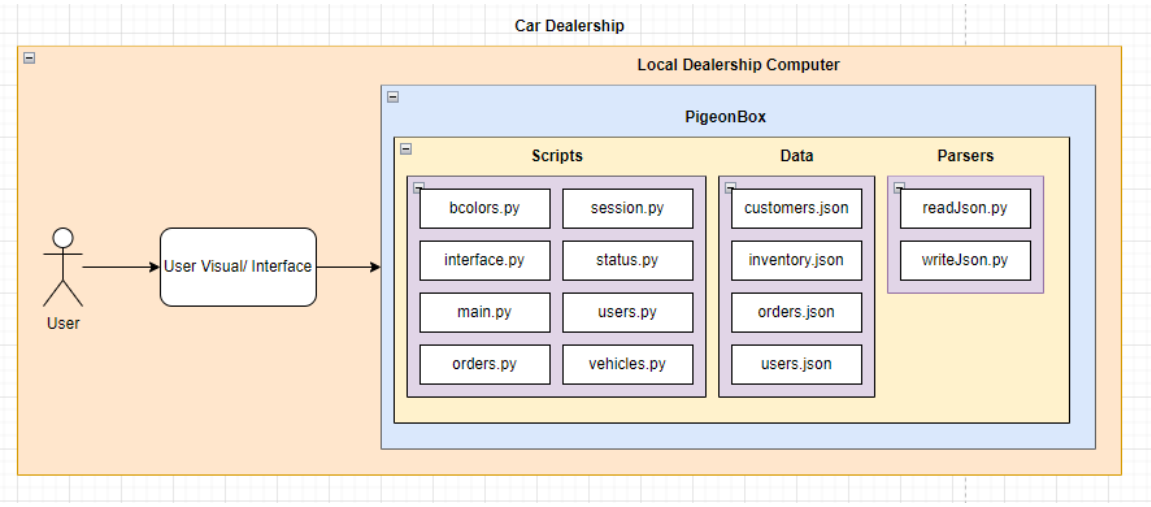
This module is used for reading and writing data stored in json files as json objects.

1. readJson.py
    a. This is responsible for reading the json files: customers, inventory, orders, and users.

2. writeJson.py
    a. This is responsible for changing existing json objects or adding new objects with proper formatting.

Architecture Diagrams





User Scenarios

Below is a list of all possible scenarios users may find themselves in while traversing the system. These include "proper" traversal of the functions in the system as well as error cases. These scenarios cover the functional requirements and follow closely to the system decomposition diagrams above.s

1. Scenario: User fails to login
    a. User enters username.
    b. User enters password.
    c. Either the username, password, or both are incorrect.
    d. User will be prompted to re-enter their data.
    e. User gets three attempts. If the third attempt is incorrect, the program shuts down.

2. Scenario: Admin successful login
    a. User enters username.
    b. User enters password.
    c. Information is correct.
    d. System verifies and authenticates username and password.
    e. User is logged in.
    f. User is logged in as Admin.
    g. User views the main page.

3. Scenario: General employee successful login
    a. User enters username.
    b. User enters password.
    c. Information is correct.
    d. System verifies and authenticates username and password.
    e. User is logged in.
    f. User is logged in as a general employee.
    g. User views the main page.

4. Scenario: User successfully logs out and enters login page
    a. User is on the main page.
    b. User presses 'q' to exit.
    c. User confirms they would like to log in again.
    d. User is brought to the login page.

5. Scenario: User successfully logs out and shuts down program
    a. User is on the main page.
    b. User presses ‘q’ to exit.
    c. User denies they would like to log in again.
    d. User shuts down the system.

6. Scenario: Admin creates new employee
    a. Admin is logged into the system.
    b. Admin goes to Manage Employees Menu.
    c. Admin selects “Add Employee”.
    d. Admin creates employee username.
    e. Admin creates employee password.
    f. Admin creates employee first name.
    g. Admin creates employee last name.
    h. Admin chooses to grant admin privileges to employee.
    i. Employee is added to users.json
    j. Admin is redirected to Manage Employee Menu.
    k. Employee can now be seen in the employee list.

7. Scenario: Admin faces error while creating new employee
    a. Admin is logged into the system.
    b. Admin goes to Manage Employees Menu.
    c. Admin selects “Add Employee”.
    d. Admin creates employee username.
    e. Admin creates employee password.
    f. Admin creates employee first name.
    g. Admin creates employee last name.
    h. Admin chooses whether or not to grant admin privileges to employee.
    i. If the username already exists, Admin gets an error stating the user already exists.
    j. Creating new employee fails.
    k. Admin is redirected to Manage Employee Menu.

8. Scenario: Admin deletes existing employee
    a. Admin is logged into the system.
    b. Admin goes to Manage Employees Menu.
    c. Admin selects “Remove Employee”.
    d. Admin selects employee by index.
    e. Admin confirms selection and deletion.
    f. Employee is removed from users.json and will no longer show up in the employee list.
    g. Admin is redirected to Manage Employee Menu.

9. Scenario: Admin faces error while deleting existing employee
    a. Admin is logged into the system.
    b. Admin goes to Manage Employees Menu.
    c. Admin selects “Remove Employee”.
    d. Admin selects employee by index.
    e. Admin confirms selection and deletion.
    f. Employee has made an order on the system and cannot be deleted.
    g. Admin is redirected to Manage Employee Menu.

10. Scenario: Admin views employee details

a. Admin is logged into the system.
b. Admin goes to Manage Employees Menu.
c. Admin selects “View Employee details”.
d. Admin selects employee by index.
e. Admin views employee details.
f. Admin is redirected to Manage Employee Menu.

11. Scenario: Admin grants employee admin privileges
    a. Admin is logged into the system.
    b. Admin goes to Manage Employees Menu.
    c. Admin selects “View Employee details”.
    d. Admin selects employee by index.
    e. Admin views employee details.
    f. Admin is asked to grant admin privileges.
    g. Admin is redirected to Manage Employee Menu.
    h. The employee with newly granted admin privileges will show up in the admin list.

12. Scenario: Admin adds car to inventory
    a. Admin is logged into the system.
    b. Admin goes to the Car Inventory Menu.
    c. Admin selects “Add/Remove Cars”.
    d. Admin selects “Add car”.
    e. Admin fills in the car's attributes.
    f. Admin selects status of car.
    g. Car is added to inventory.json.
    h. Admin is redirected to Inventory Menu.
    i. The newly added car should show up at the bottom of the inventory list.

13. Scenario: Admin faces error while adding car to inventory
    a. Admin is logged into the system.
    b. Admin goes to the Car Inventory Menu.
    c. Admin selects “Add/Remove Cars”.
    d. Admin selects “Add car”.
    e. Admin fills in the car's attributes.
    f. If the car VIN attribute is the same, error “Car already exists” is thrown and car is not created.
    g. Admin is redirected to Inventory Menu.

14. Scenario: Admin removes car from inventory
    a. Admin is logged into the system.
    b. Admin goes to the Car Inventory Menu.
    c. Admin selects “Add/Remove Cars”.
    d. Admin selects “Remove car”.
    e. Admin selects cars by index.
    f. Admin confirms deletion.
    g. Car is removed from inventory.json.
    h. Admin is redirected to Inventory Menu.
    i. The selected car should no longer show up on the inventory list.

15. Scenario: User changes username
    a. User selects Account settings
    b. User selects “Change username”.

c. User enters a new username that isn't already on the system.
d. Username is changed.
e. User is redirected to the Account settings menu.

16. Scenario: User faces error while changing username
    a. User selects Account settings
    b. User selects "Change username".
    c. User enters a new username that is already on the system.
    d. Error "New username cannot be the same as old username" or "username is already taken".
    e. Username fails to change.
    f. User is redirected to the Account settings menu.

17. Scenario: User changes password
    a. User selects Account settings
    b. User selects "Change password".
    c. User enters a new password.
    d. User retypes new password to confirm.
    e. Password is changed.
    f. User is redirected to the Account settings menu.

18. Scenario: User faces error while changing password
    a. User selects Account settings
    b. User selects "Change password".
    c. User enters a new password.
    d. User retypes new password to confirm.
    e. Password does not match or password is the same as the old password.
    f. User is redirected to the Account settings menu.

19. Scenario: User view's their account details
    a. User selects Account settings.
    b. User selects "View Account Details".
    c. User view account information.
    d. User is redirected to the Account settings menu.

20. Scenario: User adds customer
    a. User selects Manage Customers.
    b. User selects "Add customer".
    c. User enters the customer's first name, last name, email, credit card number, and home address.
    d. Customer is successfully added and is added to customer.json.
    e. User is redirected to the Manage Customer Menu.
    f. The new customer can be found at the bottom of the customer list.

21. Scenario: User faces error while adding customer
    a. User selects Manage Customers.
    b. User selects "Add customer".
    c. User enters the customer's first name, last name, email, credit card number, and home address.
    d. Customer already exists if information is matching.
    e. Customer fails to be added to the system.
    f. User is redirected to the Manage Customer Menu.

22. Scenario: User removes customer
    a. User selects Manage Customers.
    b. User selects “Remove Customer”.
    c. User selects customer by index.
    d. User is asked for confirmation on deleting customer. User selects yes.
    e. Customer is deleted and removed from customers.json.
    f. User is redirected to Manage Customers Menu.

23. Scenario: User views customer details
    a. User selects Manage Customers.
    b. User selects “View Customer details”.
    c. User selects customer by index.
    d. User is redirected to Manage Customers Menu.

24. Scenario: User edits customer details from view customer details
    a. User selects Manage Customers.
    b. User selects “View Customer details”.
    c. User selects customer by index.
    d. User confirms updating customer details.
    e. User selects home address, email address, or card to update.
    f. User enters new information.
    g. User is redirected to Manage Customers Menu.

25. Scenario: User faces error while editing customer details from view customer details
    a. User selects Manage Customers.
    b. User selects “View Customer details”.
    c. User selects customer by index.
    d. User confirms updating customer details.
    e. User selects home address, email address, or card to update.
    f. User enters new information.
    g. If card number is not 16 digits user will be prompted to enter again.
    h. User is redirected to Manage Customers Menu.

26. Scenario: User adds order from Order Menu for existing customer
    a. User selects Customer Orders.
    b. User selects “Add order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User denys order for new customer
    f. User selects customer by index.
    g. Order is made successfully and is added to orders.json.
    h. User is redirected to the Order Menu.
    i. Order can be seen at the bottom of the order list.

27. Scenario: User faces error while adding order from Order Menu for existing customer
    a. User selects Customer Orders.
    b. User selects “Add order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User denys order for new customer
    f. User selects customer by index.

g. User selected order that is already ordered and is not available.
h. New order fails.
i. User is redirected to the Order Menu.

28. Scenario: User adds order from Order Menu for new customer
    a. User selects Customer Orders.
    b. User selects “Add order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User confirms order for new customer
    f. User fills out customer information: first name, last name, home address, email address, credit card number.
    g. New customer is added to system.
    h. Order is made successfully and is added to orders.json.
    i. User is redirected to the Order Menu.
    j. Order can be seen at the bottom of the order list.

29. Scenario: User faces error while adding order from Order Menu for new customer
    a. User selects Customer Orders.
    b. User selects “Add order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User confirms order for new customer
    f. User fills out customer information: first name, last name, home address, email address, credit card number.
    g. New customer is added to system.
    h. User selected order that is already ordered and is not available.
    i. New order fails.
    j. User is redirected to the Order Menu.
    k. Order can be seen at the bottom of the order list.

30. Scenario: User adds order from inventory menu for existing customer
    a. User selects Car Inventory Menu.
    b. User selects “Make a customer order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User denys order for new customer
    f. User selects customer by index.
    g. Order is made successfully and is added to orders.json.
    h. User is redirected to the Car Inventory Menu.

31. Scenario: User faces error while adding order from inventory menu for existing customer
    a. User selects Car Inventory Menu.
    b. User selects “Make a customer order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User denys order for new customer
    f. User selects customer by index.
    g. User selected order that is already ordered and is not available.
    h. New order fails.
    i. User is redirected to the Car Inventory Menu.

32. Scenario: User adds order from inventory menu for new customer
    a. User selects Car Inventory Menu.
    b. User selects “Make a customer order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User confirms order for new customer
    f. User fills out customer information: first name, last name, home address, email address, credit card number.
    g. New customer is added to the system.
    h. Order is made successfully and is added to orders.json.
    i. User is redirected to the Car Inventory Menu.

33. Scenario: User faces error while adding order from inventory menu for new customer
    a. User selects Car Inventory Menu.
    b. User selects “Make a customer order”.
    c. User selects vehicle by index.
    d. User confirms order.
    e. User confirms order for new customer
    f. User fills out customer information: first name, last name, home address, email address, credit card number.
    g. New customer is added to the system.
    h. User selected order that is already ordered and is not available.
    i. New order fails.
    j. User is redirected to the Car Inventory Menu.

34. Scenario: User removes order
    a. User selects Customer Orders Menu
    b. User selects “Remove order”.
    c. User selects order by index.
    d. User confirms removal.
    e. Order is successfully removed from orders.json.
    f. User is redirected to the Customer Orders Menu.
    g. The order can no longer be seen from customer orders list.
    .
35. Scenario: User views order details
    a. User selects Customer Orders Menu
    b. User selects “View order details”.
    c. User selects order by index.
    d. User is redirected to the Customer Orders Menu.

36. Scenario: User changes order status from Ordered to Delivered
    a. User selects Customer Orders Menu
    b. User selects “View order details”.
    c. User selects order by index.
    d. User confirms changing order status.
    e. User selects status by index for “Available”.
    f. User is redirected to the Customer Orders Menu.
    g. The order is no longer viewed in Customer Orders Menu and can be found in Car Sales Menu.

37. Scenario: User views filtered cars
    a. User selects Car Inventory Menu.

b. User selects “Filter by”.
c. User selects filter type by index number.
d. User will see list of cars that meet that status.
e. User is redirected to Inventory Menu.’

38. Scenario: User searches for car by make, model, year
    a. User selects Car Inventory Menu.
    b. User selects “Search”.
    c. User enters model, make, and year separated by commas for desired car.
    d. Car details are returned if car is in the inventory.
    e. User is redirected to Car Inventory Menu.

39. Scenario: User faces error while searching for car by make, model, year
    a. User selects Car Inventory Menu.
    b. User selects “Search”.
    c. User enters model, make, and year separated by commas for desired car.
    d. User did not correctly enter model, make, year.
    e. User fails to make search.
    f. User is redirected to Car Inventory Menu.

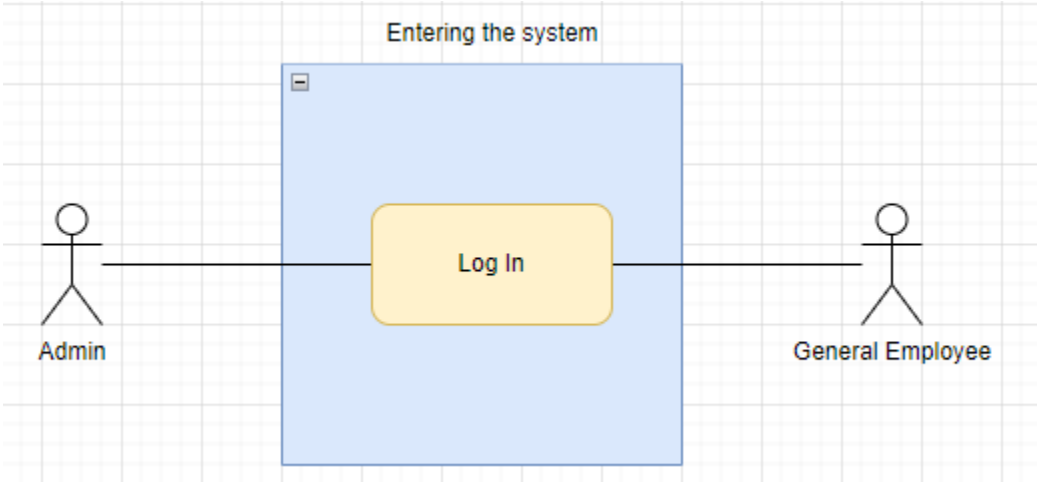
40. Scenario: User views car details
    a. User selects Car Inventory Menu.
    b. User selects “view car details”.
    c. User selects car by index.
    d. User is returned car details.
    e. User denies modify details.
    f. User is redirected to the Car Inventory Menu.

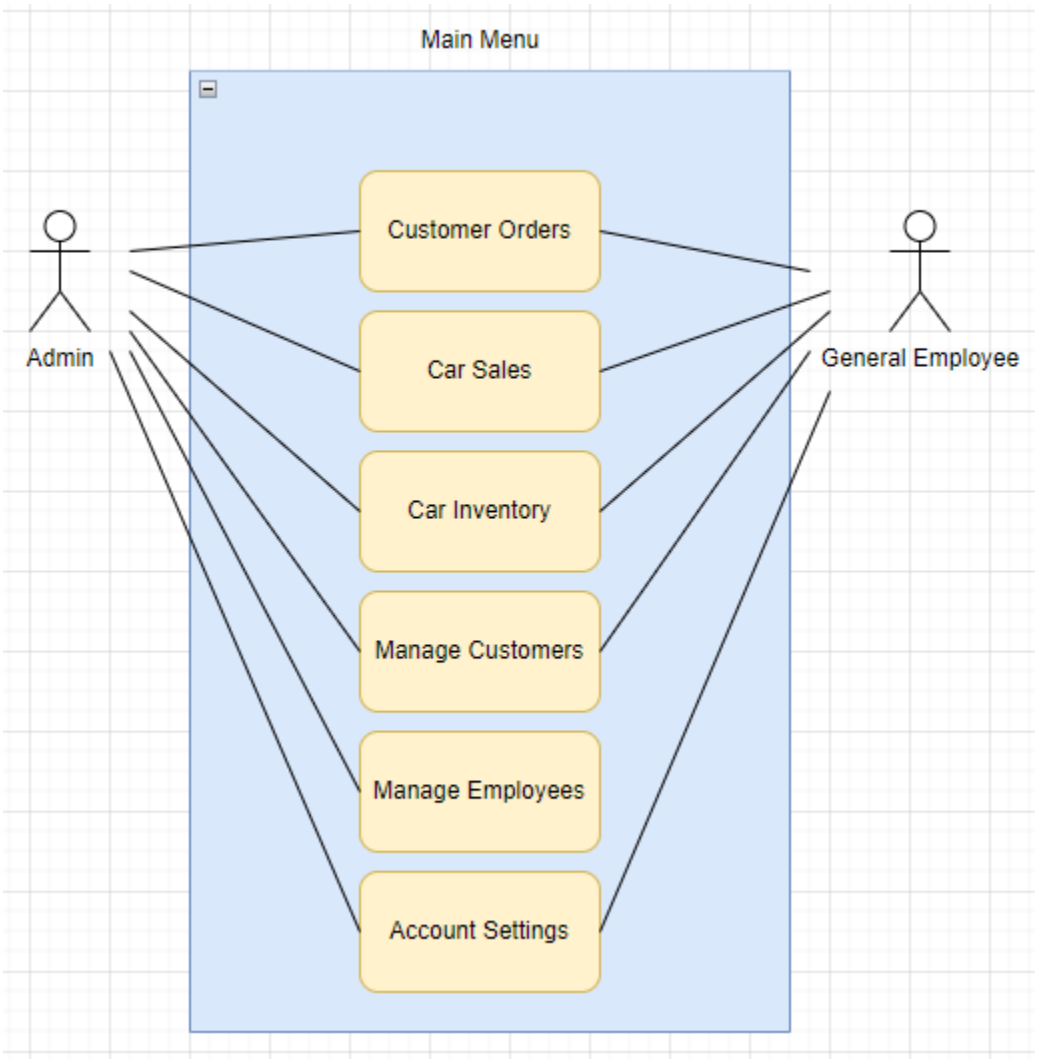
41. Scenario: User modifies car details
    a. User selects Car Inventory Menu.
    b. User selects “view car details”.
    c. User selects the car by index.
    d. User is returned car details.
    e. User confirms modify details.
    f. User selects detail to modify by index: price, warranty plans, mileage, status.
    g. User enters new detail.
    h. Detail is modified successfully.
    i. User is redirected to Car Inventory Menu.

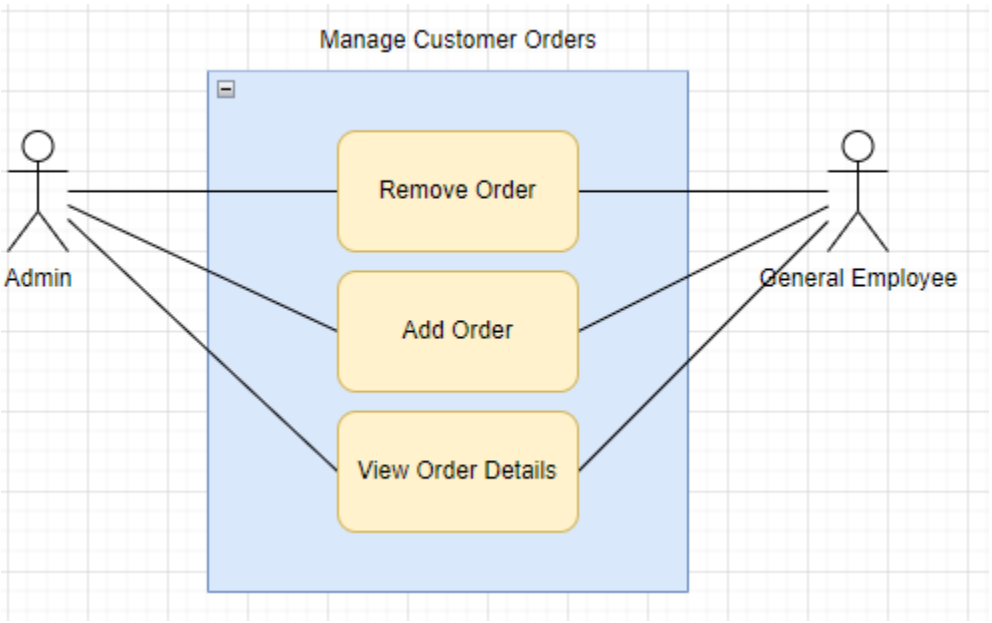
42. Scenario: User faces error while modifying car details
    a. User selects Car Inventory Menu.
    b. User selects “view car details”.
    c. User selects car by index.
    d. User is returned car details.
    e. User confirms modify details.
    f. User selects detail to modify by index: price, warranty plans, mileage, status.
    g. User enters new detail.
    h. Detail entered does not match input validation and user is prompted to reenter data.
    i. Detail is modified successfully.
    j. User is redirected to Car Inventory Menu.

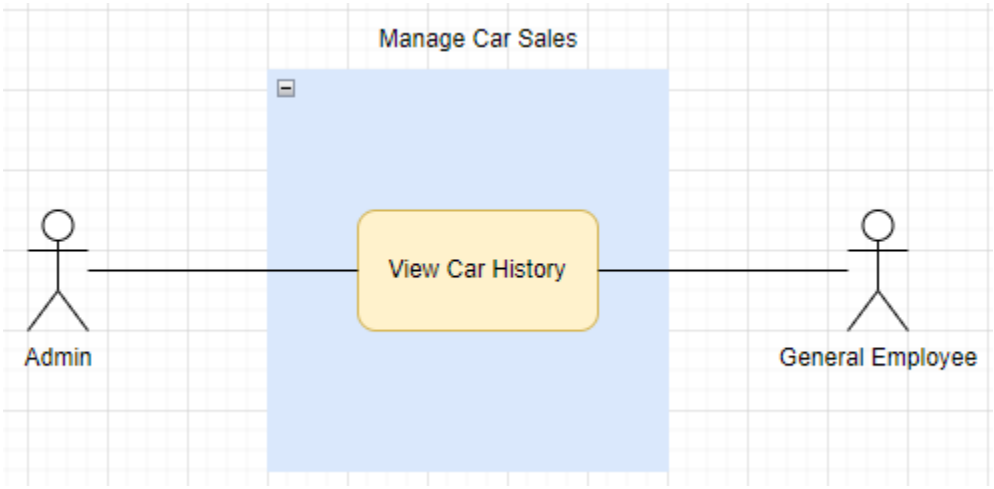
43. Scenario: User views car history
    a. User selects Car Sales Menu.
    b. User selects “View Sale details”.
    c. User selects sale by index.
    d. User is returned car sale details.
    e. User is redirected to the Car Sales Menu.

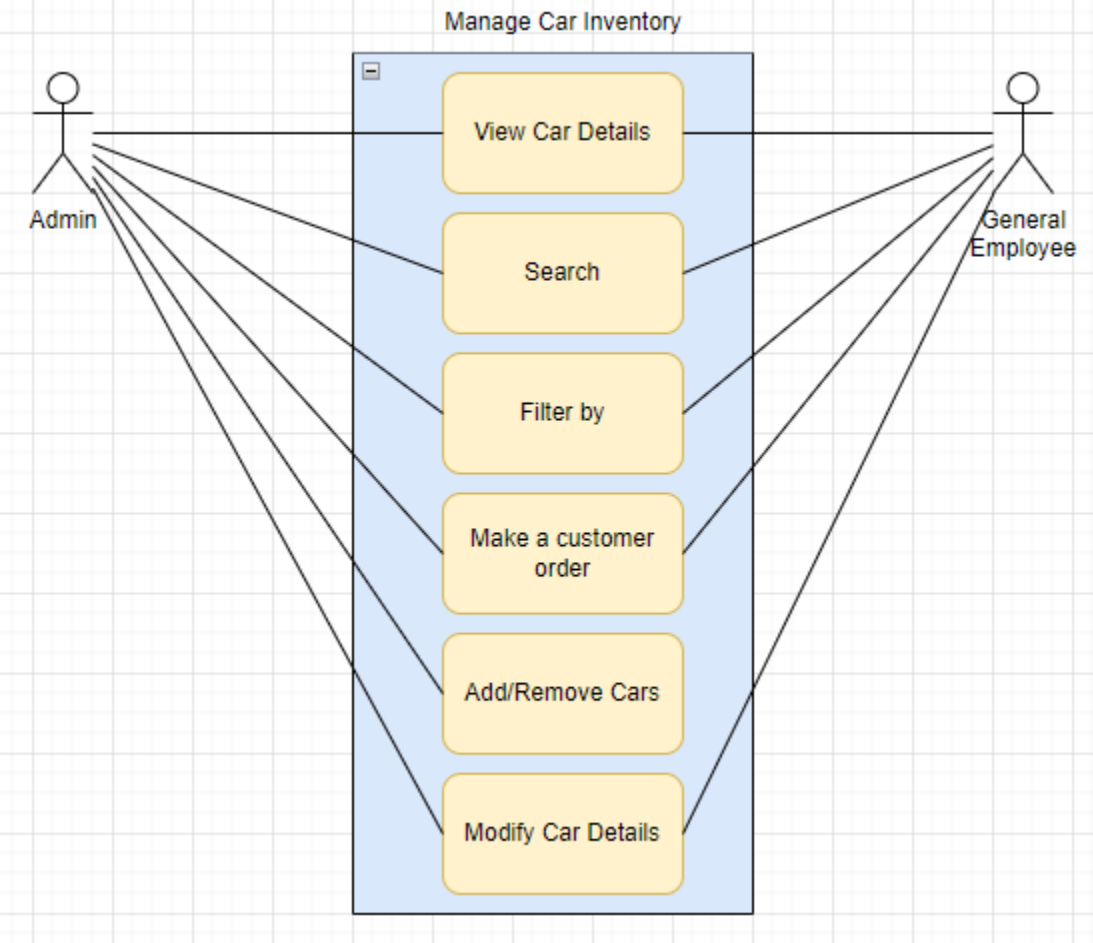
Use Case Diagrams

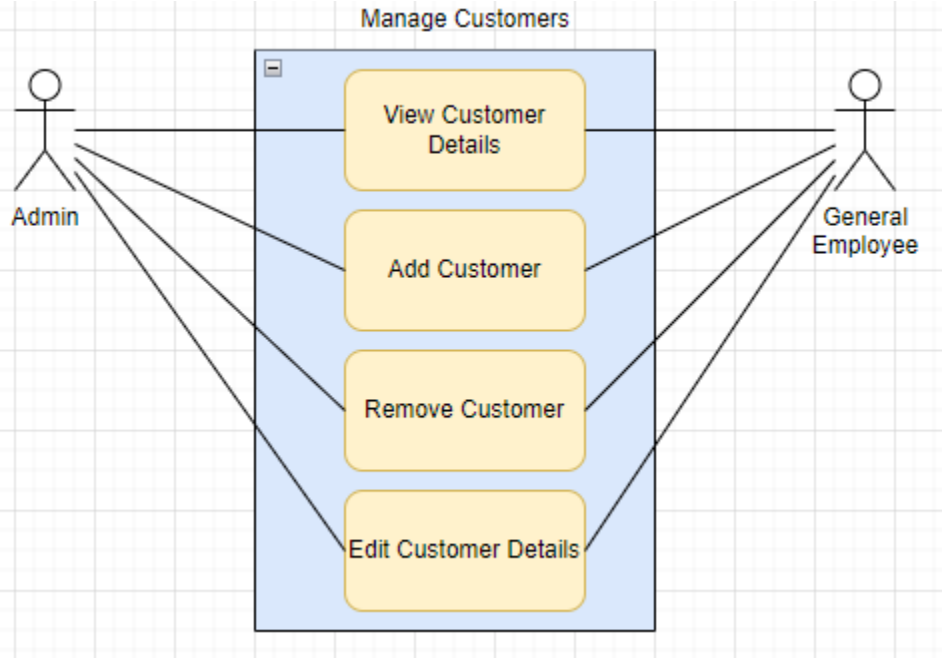


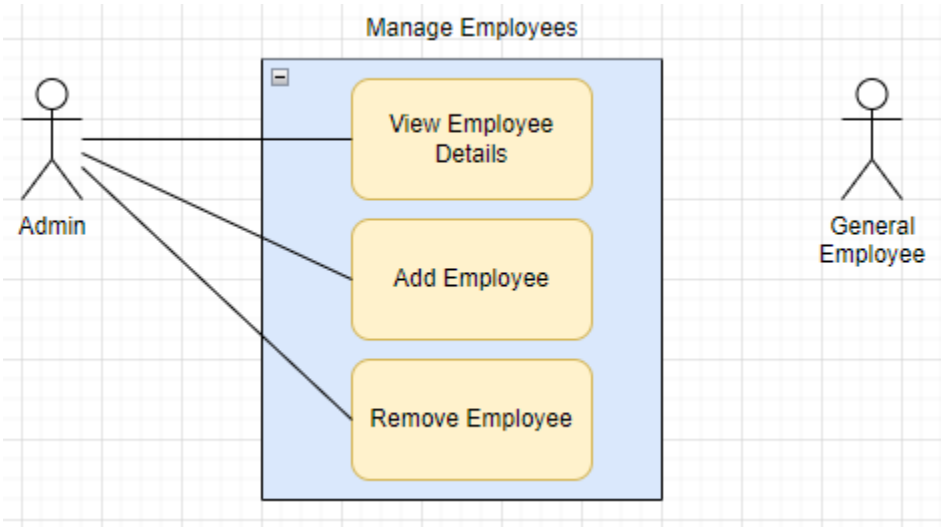

Main Menu
Customer Orders
Car Sales
Car Inventory
Manage Customers
Manage Employees
Account Settings
Admin
General Employee

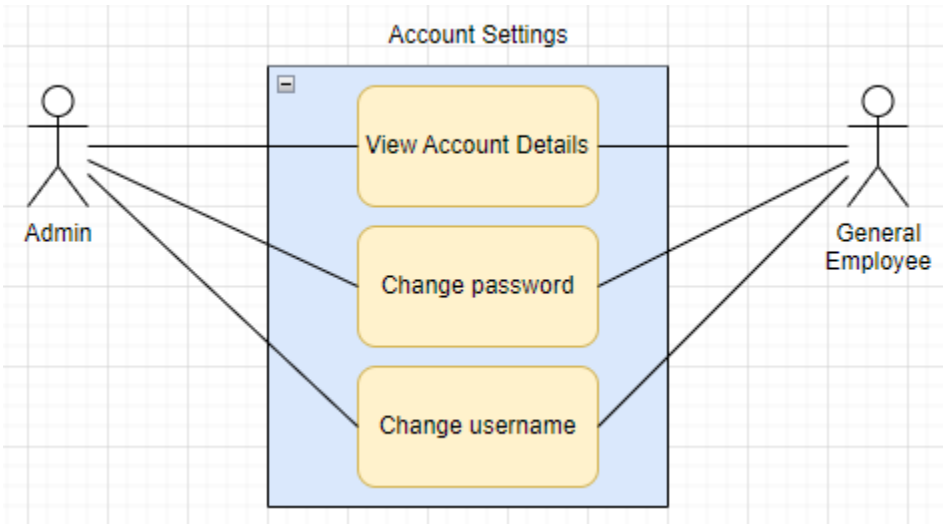

Manage Customer Orders
Remove Order
Add Order
View Order Details
Admin
General Employee


Manage Car Sales
View Car History
Admin
General Employee


Manage Car Inventory
View Car Details
Search
Filter by
Make a customer order
Add/Remove Cars
Modify Car Details
Admin
General Employee


Manage Customers
View Customer Details
Add Customer
Remove Customer
Edit Customer Details
Admin
General Employee

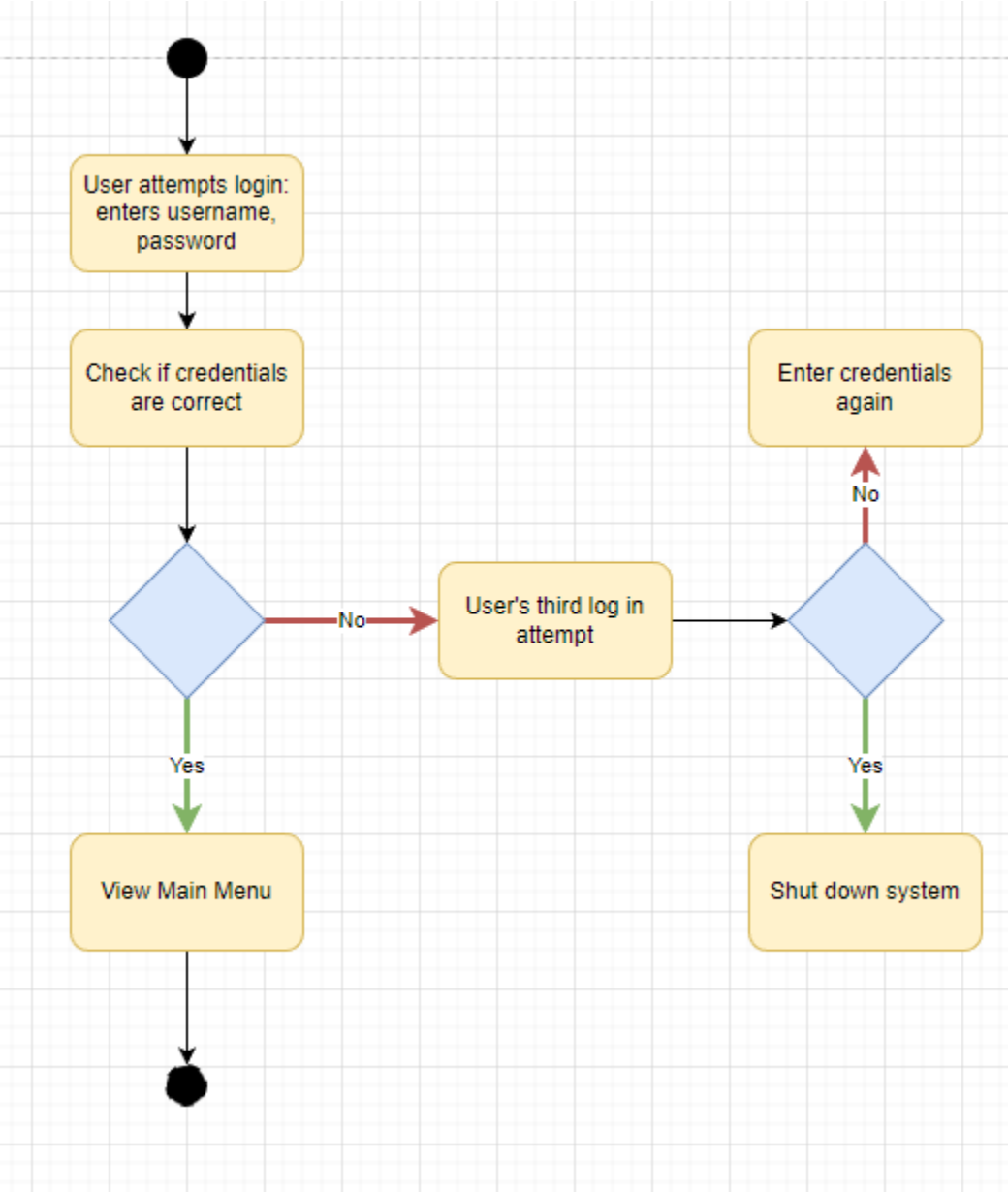




Use cases/activity diagrams

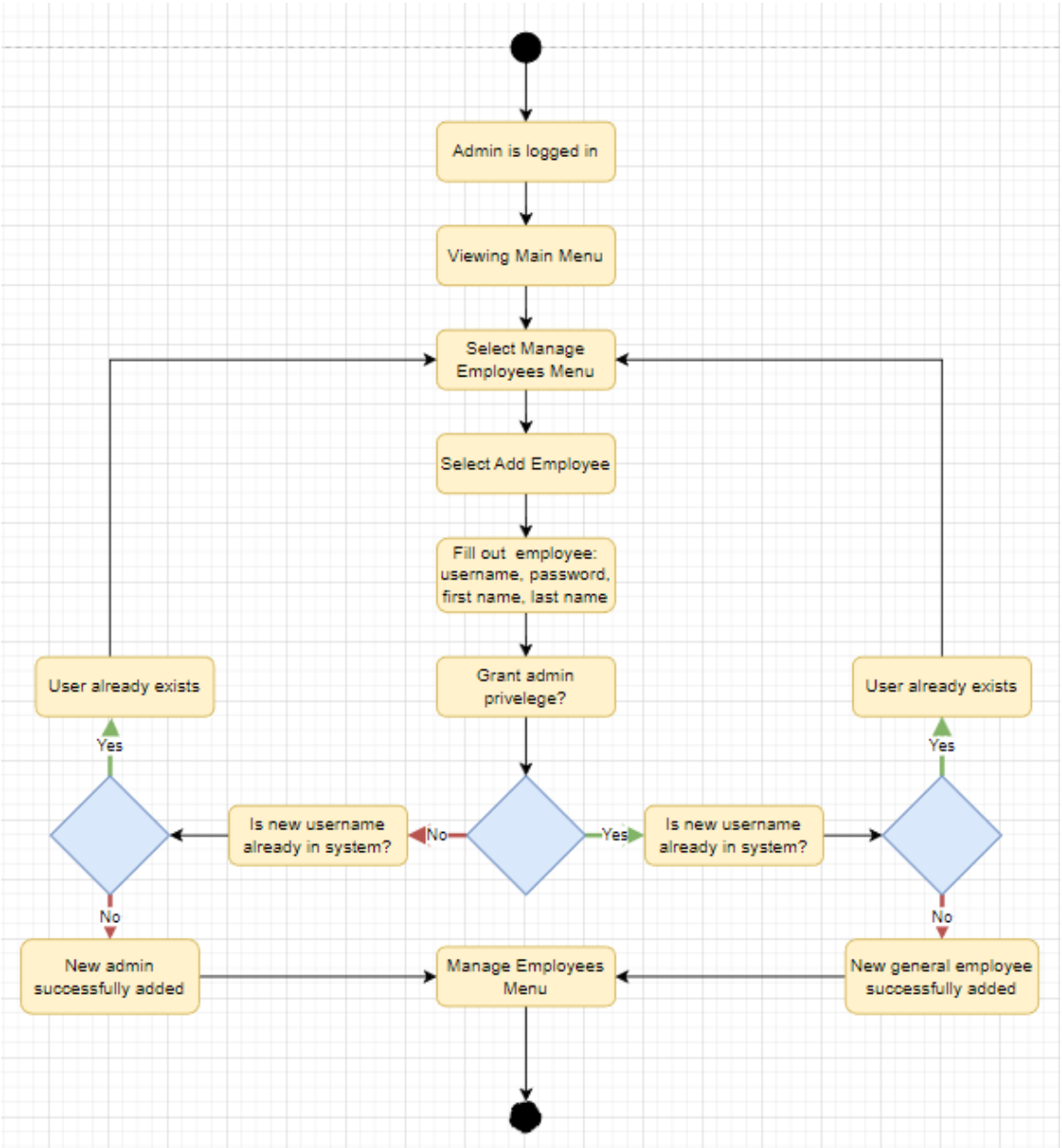
Use Case Number	1
User Case Name	User successful login
Overview	The car dealership's employees get access to the system by providing their personal credentials: username and password.
Actor(s)	Admin, General Employee
Description	To enter the system the user needs to provide their own username and password correctly.
Pre-condition	Each user should have been provided an account before logging into the system for the first time.
Alternative	None

Post condition	The user has access to the system.

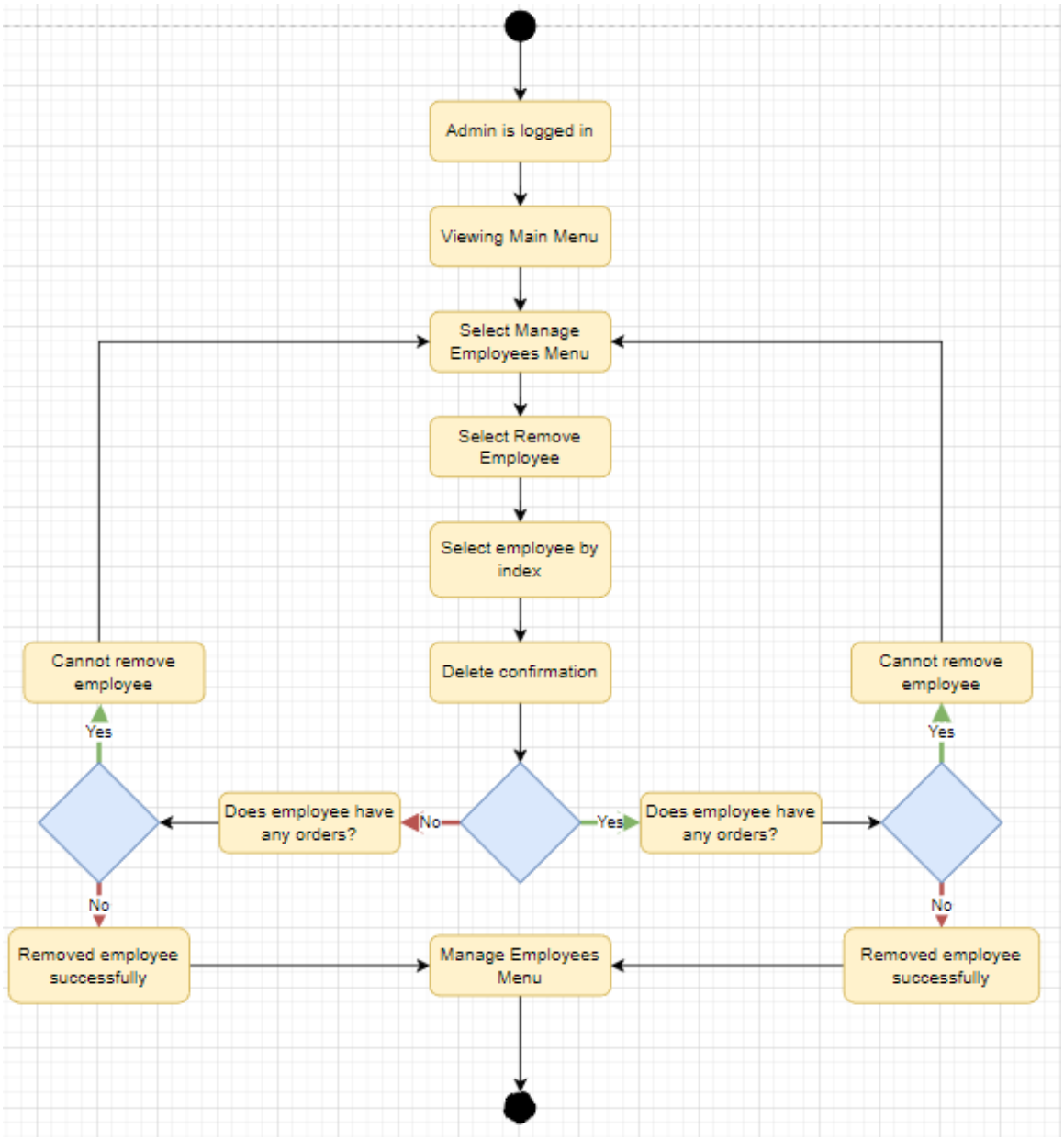


Use Case Number	2
User Case Name	Admin creates new employee
Overview	If an employee needs to be added to the system, an admin needs to register and provide that employee with a profile in the system.
Actor(s)	Admin

Description	Admin users are responsible for creating new profiles for new employees in the system. Employees cannot create their own account to the system directly from the login for security purposes.
Pre-condition	The employee needs an account in the system to have access to the VTS functions.
Alternative	The system will warn the admin if any fields are not filled correctly.
Post condition	The new employee is part of the system and can access it when the admin provides them their credentials.

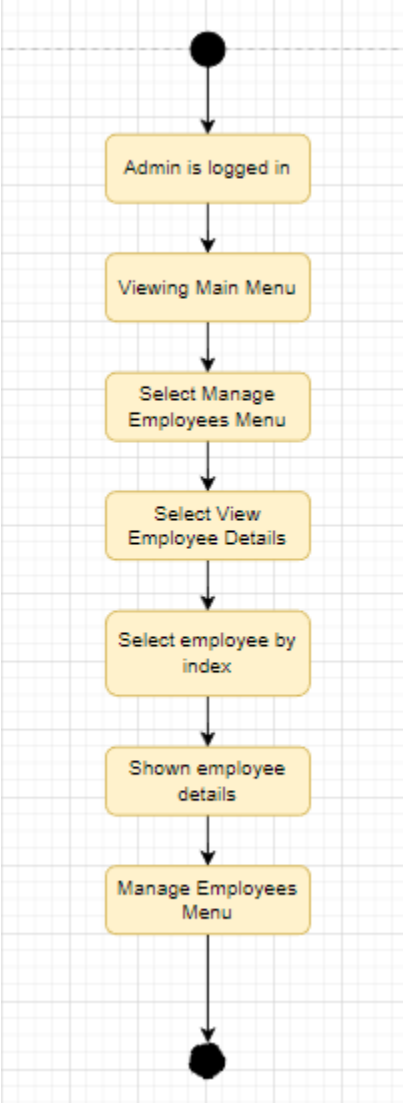


Use Case Number	3
User Case Name	Admin deletes existing employee
Overview	If an employee no longer works at the dealership an admin must be able to delete the employee's account.s
Actor(s)	Admin
Description	Employees must be removed from the system if they no longer work at the dealership. This is done by the remove employee function, however employees that made any car sales cannot be removed or errors will occur. In this case the admin should effectively shut down the employees' account by changing their username and password so that their name is still registered under the order details, but they no longer have access to the system.
Pre-condition	The dealership management decides the employee will not be part of the dealership anymore.
Alternative	Employee made a car order and cannot be deleted. Admin should instead edit employee details, changing username and password.
Post condition	Employee is removed from the system.



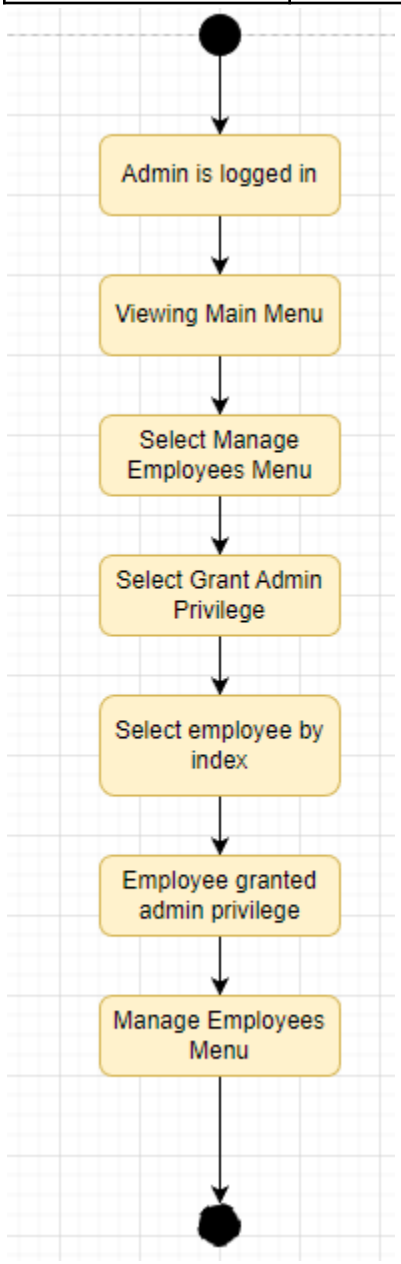
Use Case Number	4
User Case Name	Admin views employee details
Overview	Admin chooses to look at employees' detailed information.
Actor(s)	Admin
Description	In the case that an admin wants to view detailed information of their employees' or other employees, they can. This includes their first name, last name, and date joined.

Pre-condition	Must be logged in as admin.
Alternative	None.
Post condition	Admin views personal information of employee.



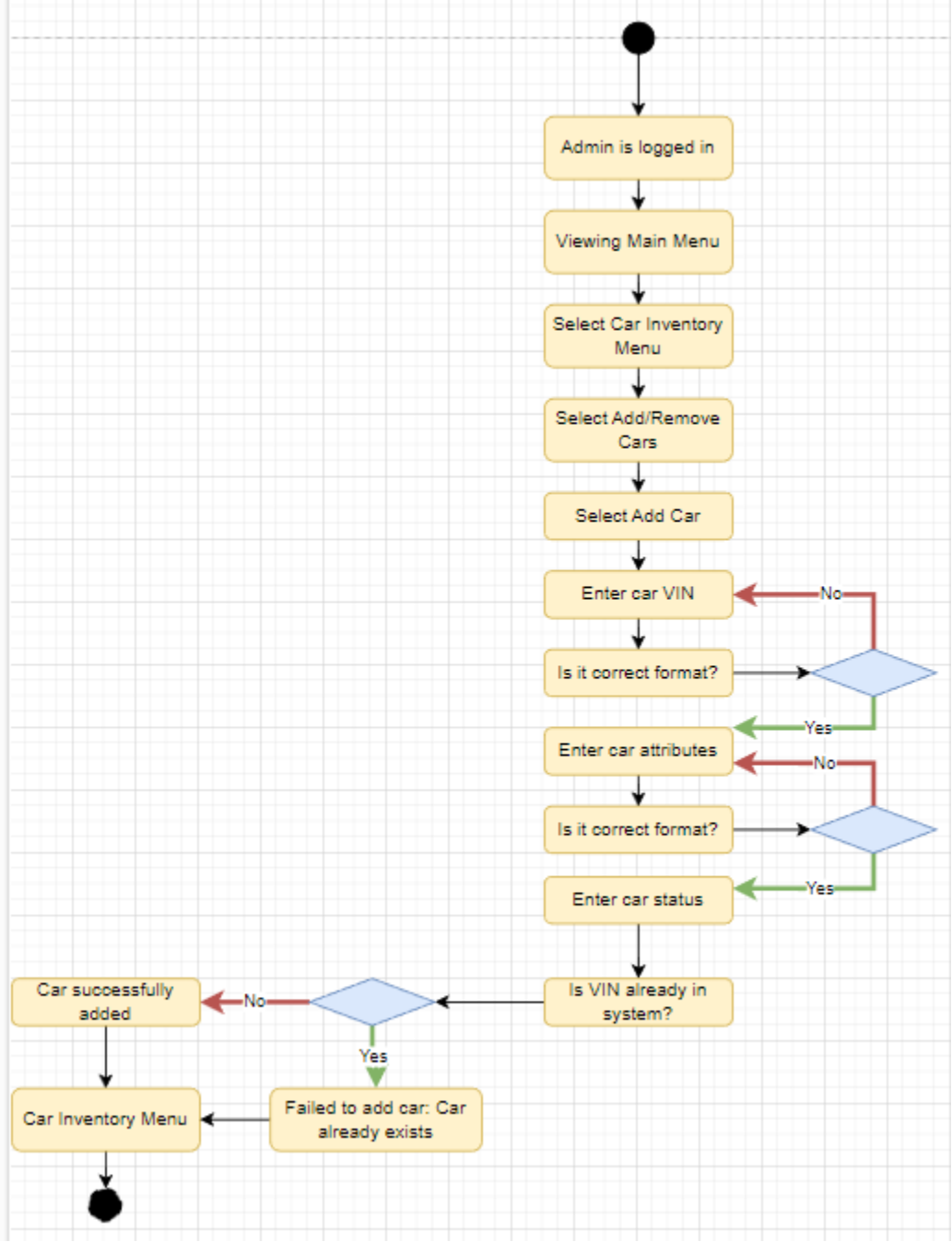
Use Case Number	5
User Case Name	Admin grants employee admin privileges
Overview	Admins grant existing employees admin privileges.
Actor(s)	Admin
Description	In the case of a promotion in the dealership to a higher position, rather than creating a new account, an existing employee can grant an existing employee admin privileges.

Pre-condition	Employee is already part of the system. User must be logged in as an admin.
Alternative	When creating a new employee, you can give them admin privileges.
Post condition	Employee is now have admin privileges.



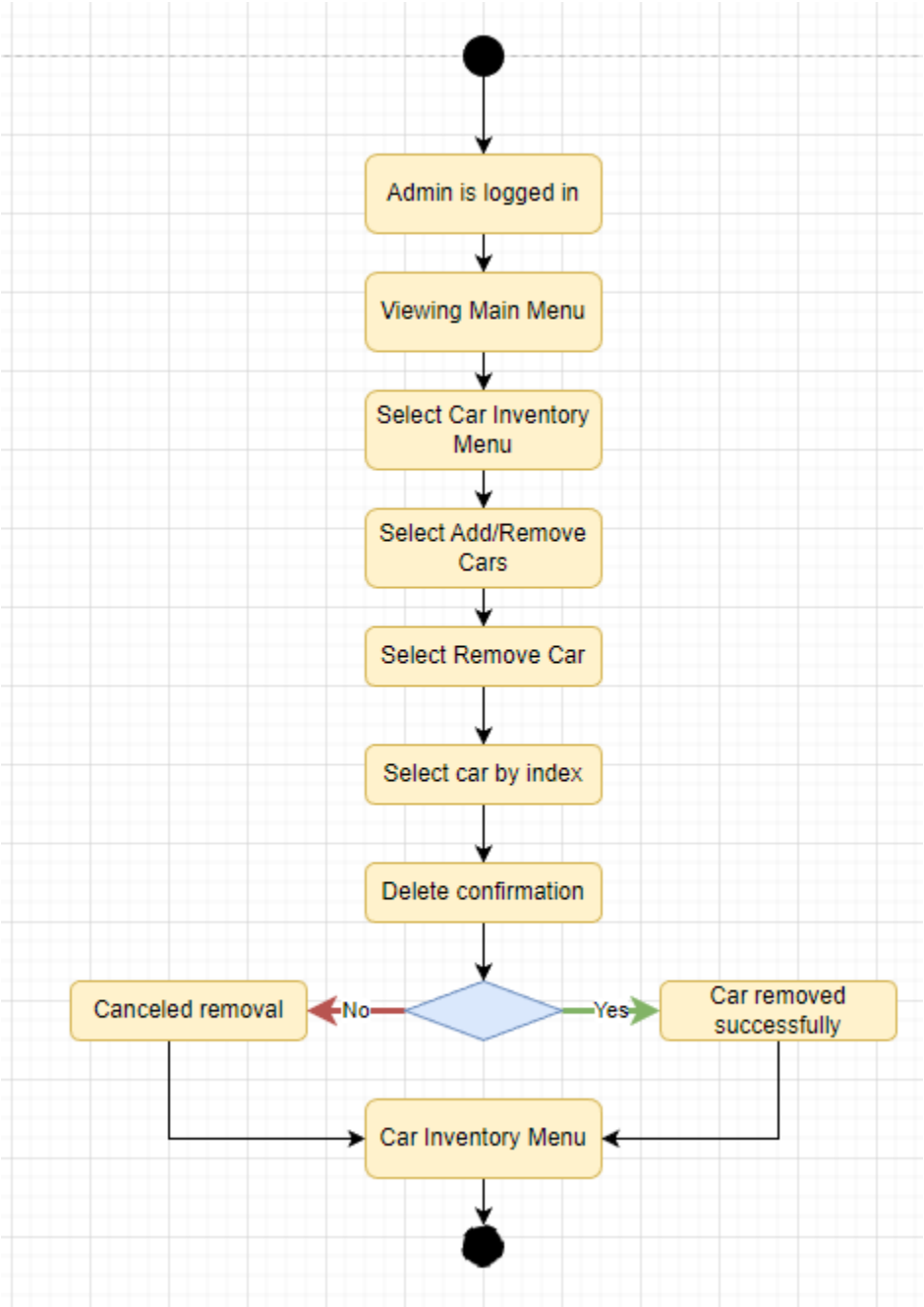
Use Case Number	6
User Case Name	Admin adds car to inventory
Overview	Admin adds new vehicle for purchase and management.

Actor(s)	Admin
Description	A new vehicle needs to be properly added to the system before it is available for employees to sell the car. This requires admins to fill out all attributes of the car.
Pre-condition	Must be logged in as an admin. Car's identification number (VIN) should not be in the system.
Alternative	Cannot add vehicle unless fields are filled in properly. If the vehicle attributes (VIN) are the same as an existing vehicle, it is the same vehicle and cannot be added.
Post condition	Vehicle is added to the dealership inventory and is available to be ordered.

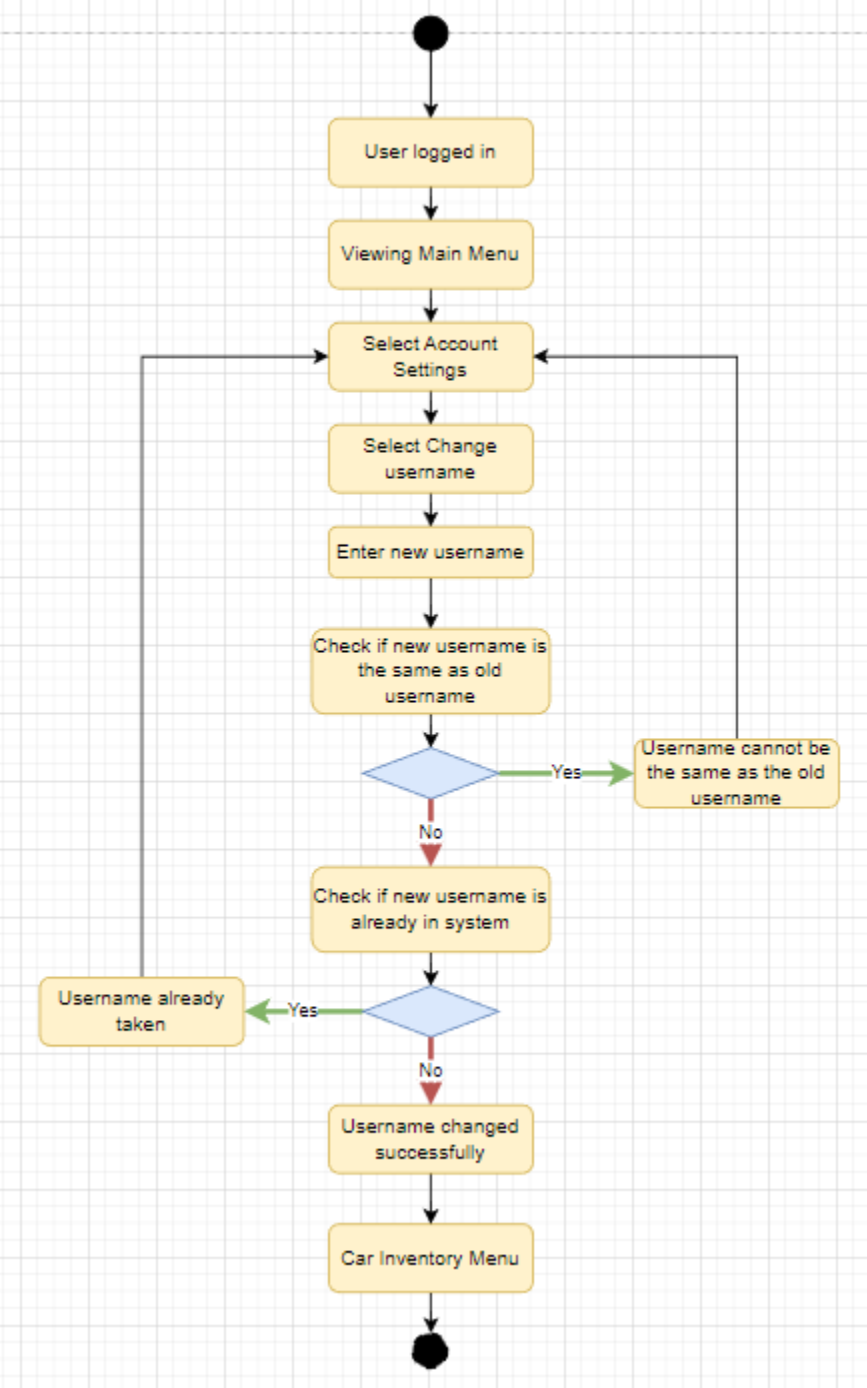


Use Case Number	7
User Case Name	Admin removes car from inventory
Overview	Admin removes car from dealership inventory.
Actor(s)	Admin
Description	Management decides that the vehicle will no longer be sold at the

	dealership and must be removed from the inventory. Reasons can be a variety of issues: can't sell it due to demand, call back, etc.
Pre-condition	User must be logged in as an admin. Vehicle must be registered in the system.
Alternative	None.
Post condition	Vehicle is removed from the dealership's inventory.

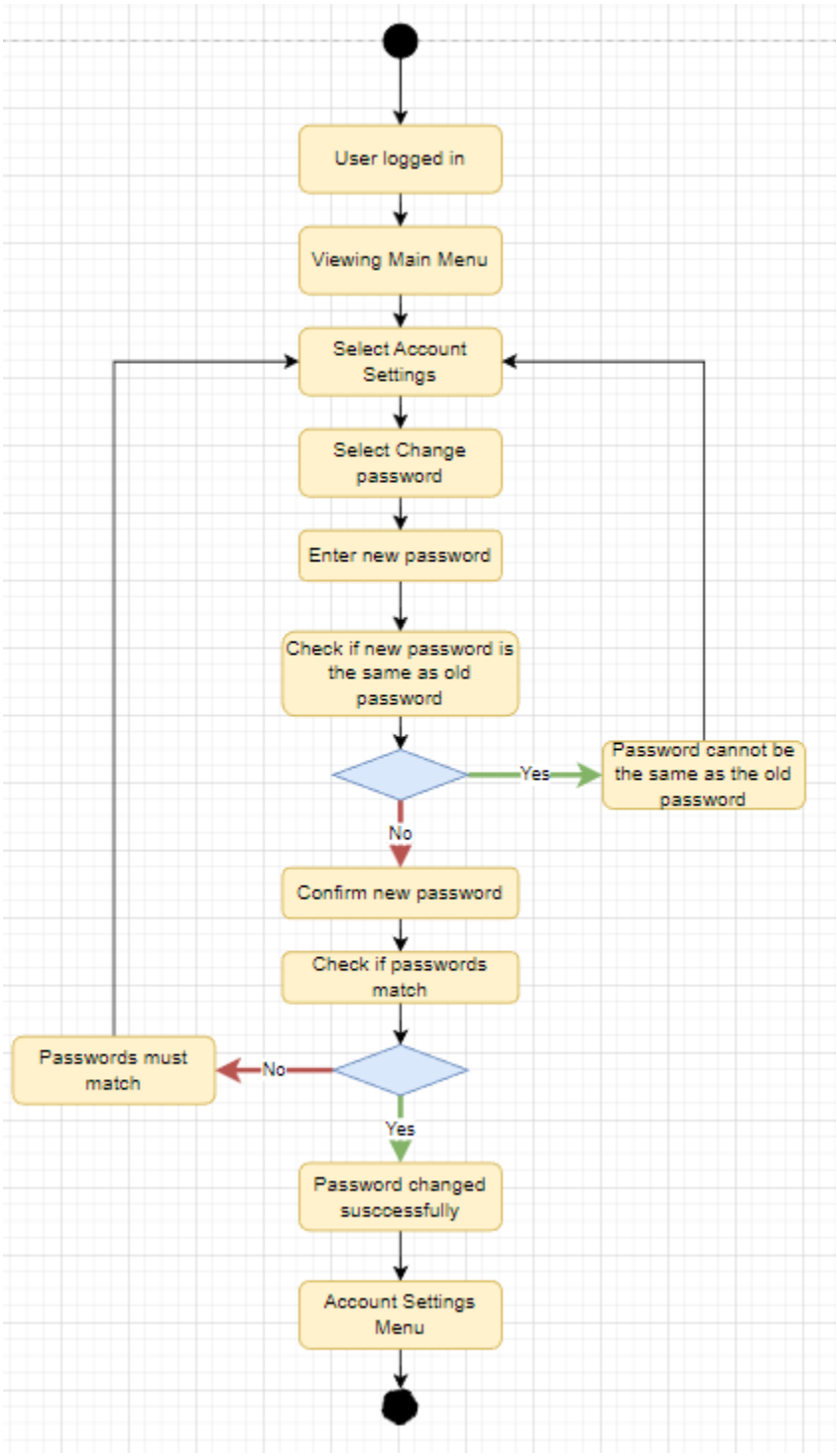


Use Case Number	8
User Case Name	User changes username
Overview	User is able to change their username for their login credentials.
Actor(s)	Admin, general employee
Description	When user accounts are first created, they are made by an admin and given to that user. At this point the user can login with premade credentials and if they wish they may change their username credential to something they want.
Pre-condition	Admin must create the user account and user must be part of the system.
Alternative	New usernames cannot be the same as the old username and it cannot already by registered in the system.
Post condition	User's username is changed.



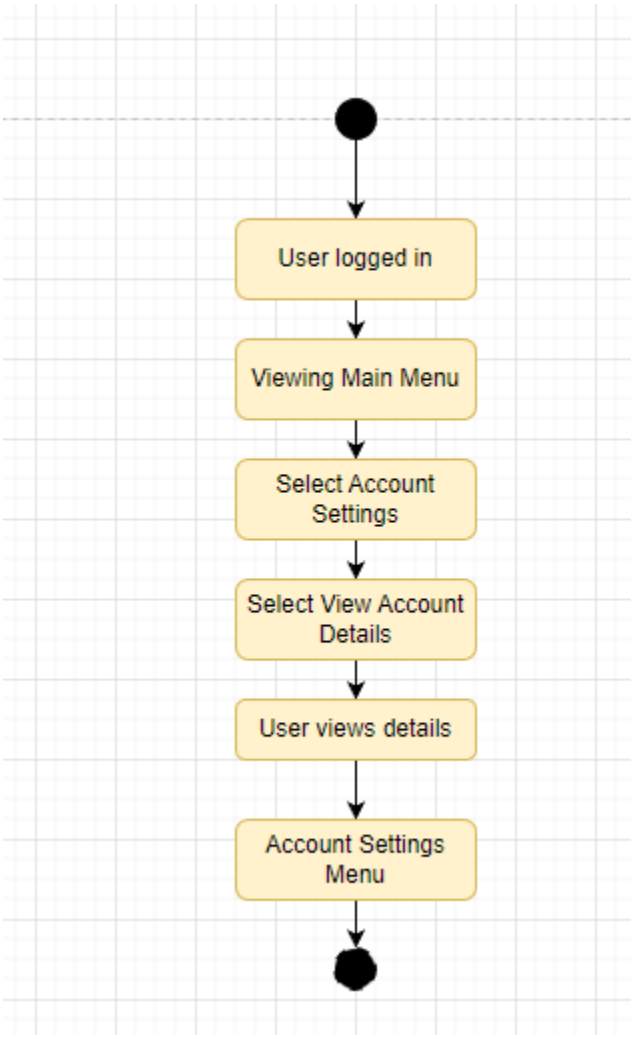
Use Case Number	9
User Case Name	User changes password
Overview	User is able to change their password for their login credentials.
Actor(s)	Admin, general employee

Description	When user accounts are first created, they are made by an admin and given to that user. At this point the user can login with premade credentials and if they wish they may change their password credential to something they want.
Pre-condition	Admin must create the user account and user must be part of the system.
Alternative	New password cannot be the same as the old password and password confirmation must match.
Post condition	User's password is changed.



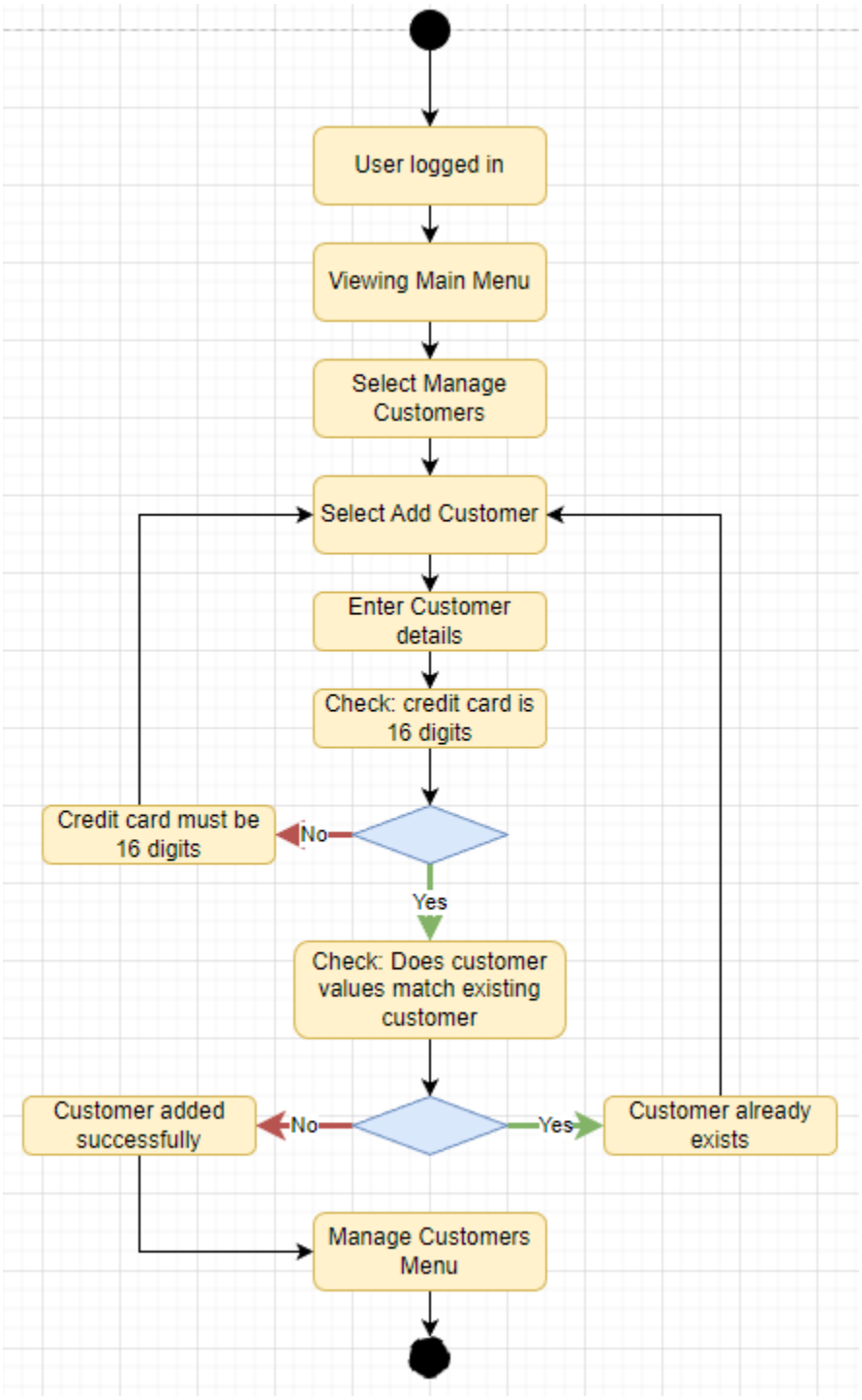
Use Case Number	10
User Case Name	User view's their account details
Overview	Users can view their own account details.

Actor(s)	Admin, General Employee
Description	Users can view their first name, last name, username, and date joined.
Pre-condition	User must be logged in.
Alternative	None.
Post condition	User views account details.



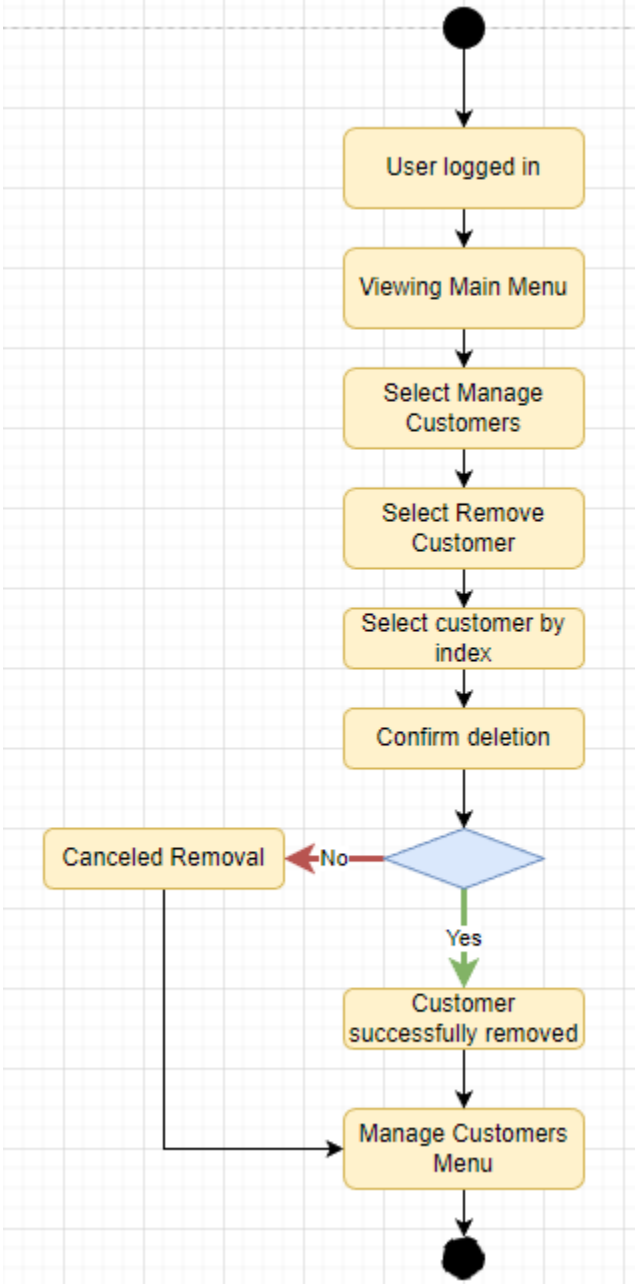
Use Case Number	11
User Case Name	User adds customer
Overview	Users can add new customers to the system.
Actor(s)	Admin, General Employee
Description	Customers cannot directly create their accounts and must do so through

	an employee of the car dealership. The employee will add all customer information here so that the employee may order a car for the customer if they so wish.
Pre-condition	User must be logged in. Customer must not be part of system.
Alternative	Customer cannot be added if already in the system.
Post condition	New customer is added to the system.

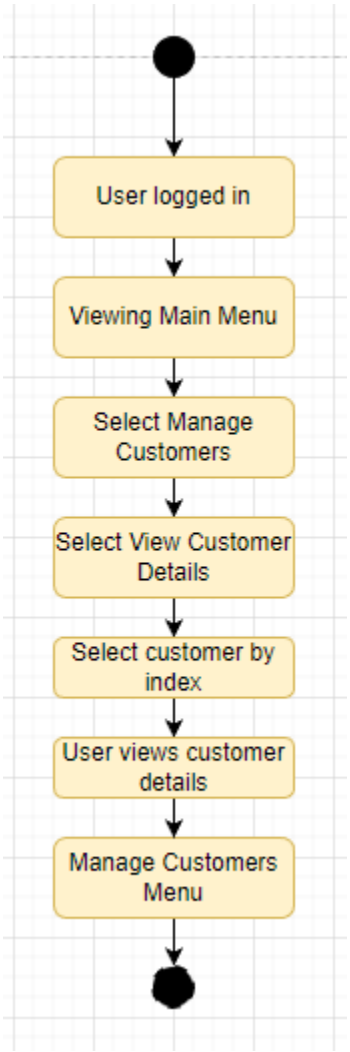


Use Case Number	12
User Case Name	User removes customer
Overview	If needed, customers can be removed from the system.

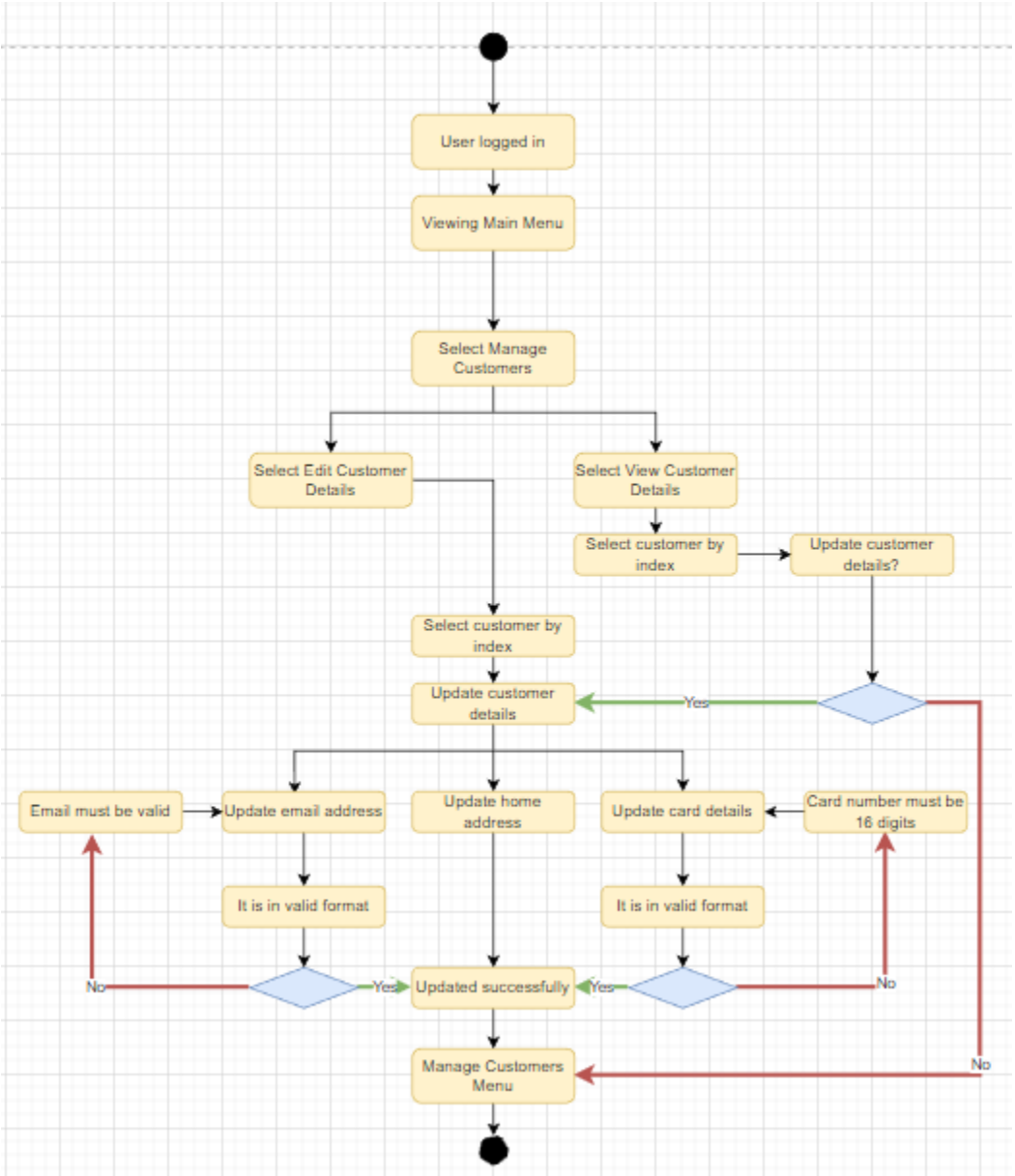
Actor(s)	Admin, General Employee
Description	If customer wishes to be removed from the system, or the customer signs up and does not make any purchases for over a year users are able to remove customers from the system.
Pre-condition	Customer must be registered in the system.
Alternative	None.
Post condition	Customer is removed from the system.



Use Case Number	13
User Case Name	User views customer details
Overview	User can view details of the customer.
User(s)	Admin, General Employee
Description	For checking customer data, users can view customer details: first name, last name, home address, email. If these need to be checked and updated, user can first view them.
Pre-condition	Customer must be registered on the system.
Alternative	None.
Post condition	User can view customer details.



Use Case Number	14
User Case Name	User edits customer details
Overview	User can edit an existing customer's details
Actor(s)	Admin, General Employee
Description	In the case that a customer has updated their home address, email, or credit card number, users can edit this information to get it up to date.
Pre-condition	Customer must be registered in the system.
Alternative	Error if new registered data is inputted incorrectly.
Post condition	Customer's data is updated.

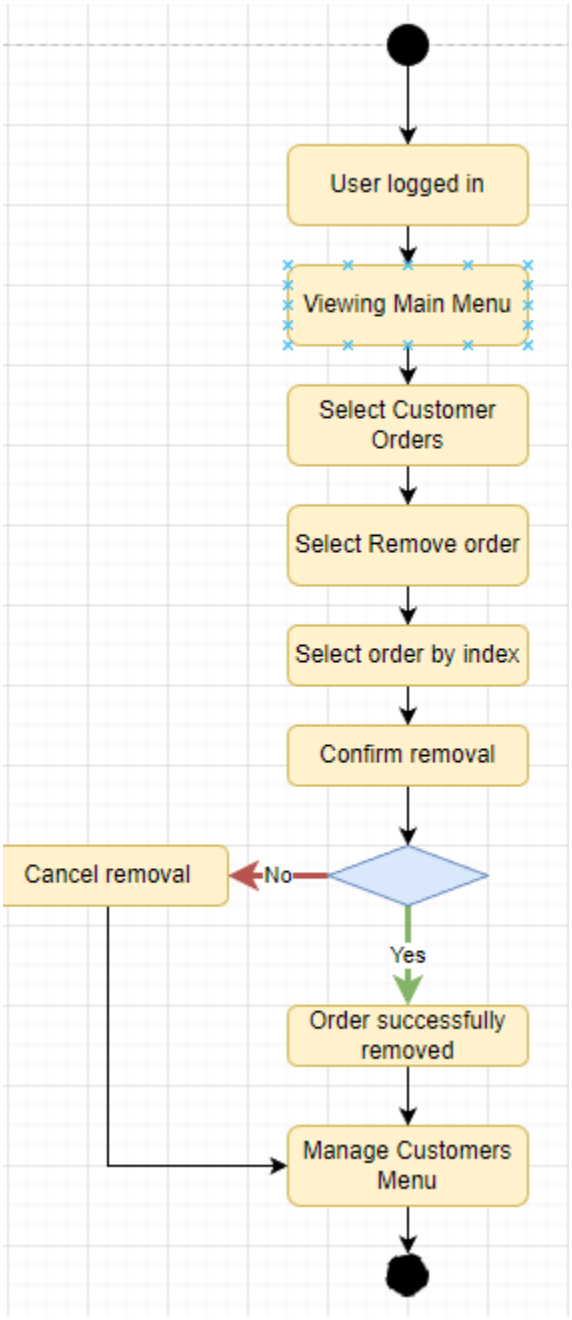


Use Case Number	15
User Case Name	User adds order
Overview	User can make order for customer.
Actor(s)	Admin, General Employee
Description	Customers will be with dealership employee during this process. User/dealership employee will create a new order for the customer. If the customer is not already registered, they will go through the Add Customer process first then continue with the add order process. The order will show up in the Customer Orders Menu and its status will change from

	Available to Ordered.
Pre-condition	Selected car must have status of Available. Customer must be registered in system before ordering car.
Alternative	If the car status is not available, the order cancels.
Post condition	The order will be linked with that customer and can be found in the Customer Orders Menu.

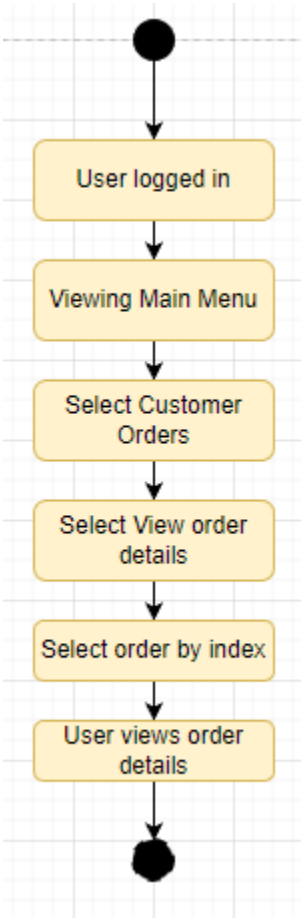
Use Case Number	16

User Case Name	User removes order
Overview	If needed, users can remove orders that have not completed the order process.
Actor(s)	Admin, General Employee
Description	Customers who want to cancel their order can request employees of the dealership to do so. Employees can then cancel these orders. The car will return to the available car list and are again able to be purchased by other customers.
Pre-condition	Car must have status Ordered.
Alternative	None.
Post condition	Car order is removed from the system.



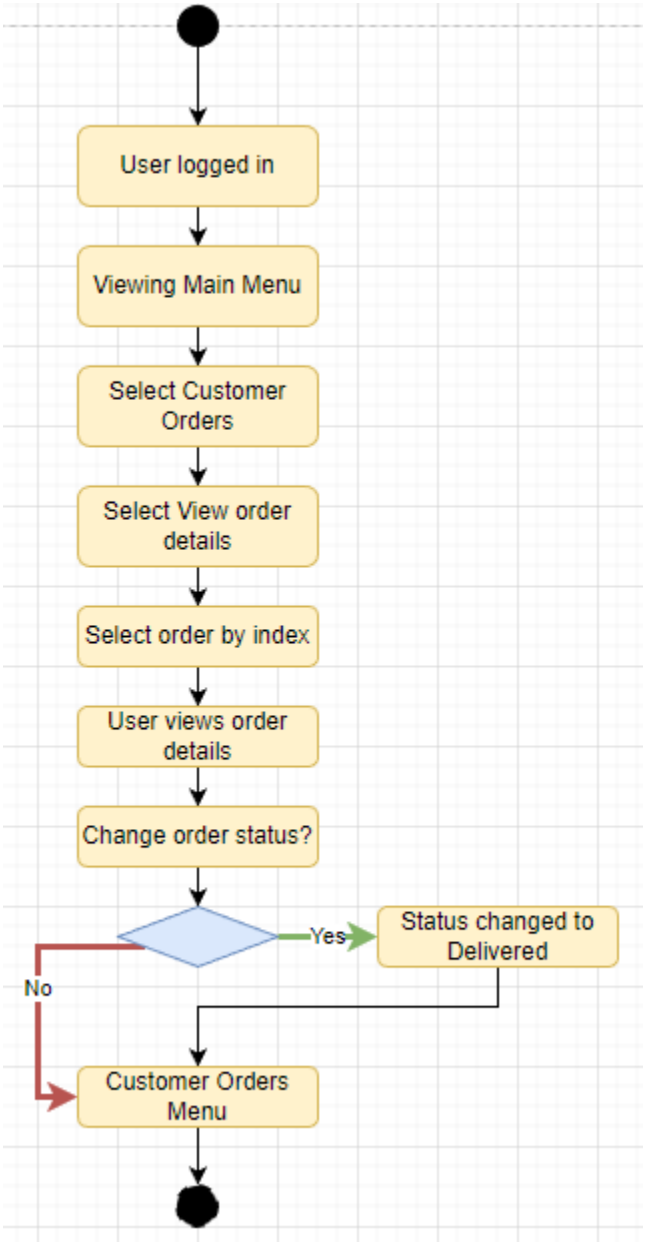
Use Case Number	17
User Case Name	User views order details
Overview	Users can view details of customer orders
Actor(s)	Admin, General Employee
Description	Users can view who sold the car, the customer who bought the car, and car details such as make, model, year, mileage, interior and so forth. From

	this point, the user can manually change the status from Ordered to Delivered if the order process has been completed and the car is delivered to the customer's residence.
Pre-condition	Order status must be Ordered. Customer order must be registered in system.
Alternative	None.
Post condition	User views order details.



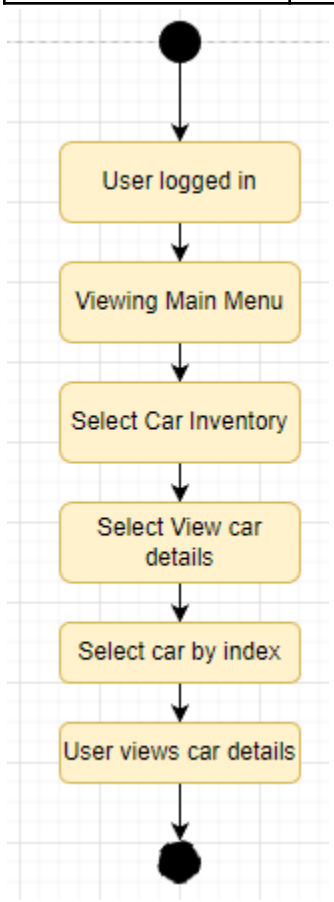
Use Case Number	18
User Case Name	User changes order status from Ordered to Delivered
Overview	User can change order status once the car is delivered to customer.
Actor(s)	Admin, General Employee
Description	The order process ends once the car reaches the customer's home address. Once this occurs, employees will manually change the order status from Ordered to Delivered. The order will no longer show up in the Customer Orders Menu.

Pre-condition	Order status must be Ordered. Customer order must be registered in system. Car must be delivered to customer home address.
Alternative	None.
Post condition	Customer order's status will be Delivered. Order will no longer show up in Customer Order Menu.



Use Case Number	19
User Case Name	User views car details

Overview	User can view details of cars in inventory.
Actor(s)	Admin, General Employee
Description	Users are able to view all attributes of car in the dealership's inventory. These are make, model, year, price, mileage, interior, paint, etc. Viewing car details is a preliminary step before modifying these details or making a car order.
Pre-condition	User must be logged in. Car must be registered in the inventory.
Alternative	None.
Post condition	User views car's details.

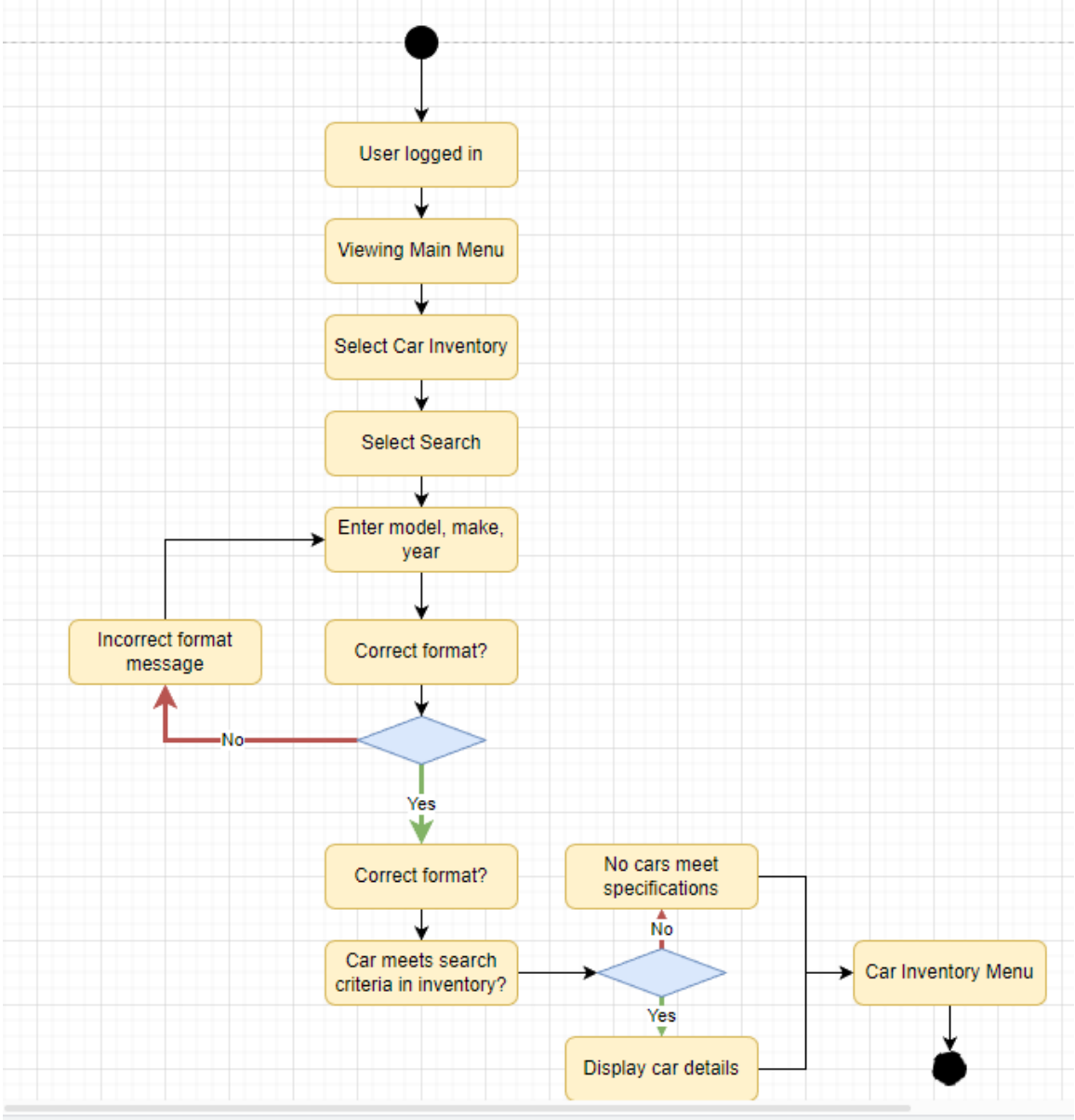


Use Case Number	20
User Case Name	User views filtered cars
Overview	User can view cars by their status.
Actor(s)	Admin, General Employee

Description	Users can view cars by their status: Delivered, Backordered, Ordered, Available. Dealership employees are able to see all vehicles in the inventory this way or can filter the inventory by their status for faster browsing.
Pre-condition	User must be logged in.
Alternative	There may be no cars of that status in the inventory.
Post condition	User will view list of cars with that filter status.

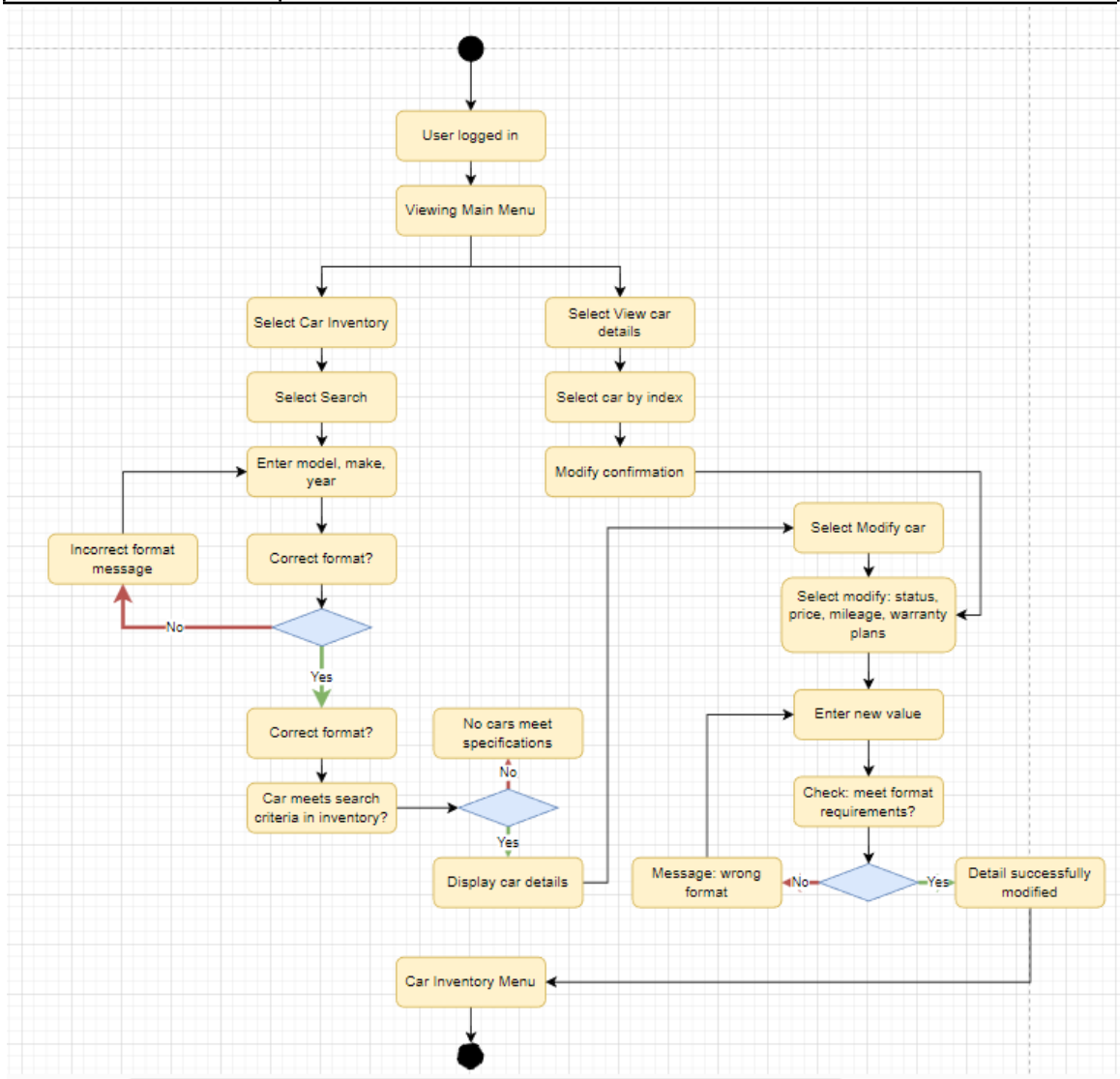
Use Case Number	21

User Case Name	User searches for car by make, model, year
Overview	Users can search the dealership inventory by car's make, model, and year.
Actor(s)	Admin, General Employee
Description	If users want to check if a specific car is in their inventory they can search by make, model, year. If a car that matches that information is in the dealership inventory, its details will show up.
Pre-condition	User must be logged in. User must know what car they want to search for. Desired car must be in inventory to show up.
Alternative	If the search doesn't match the name of the car exactly in the inventory it won't show up. Search must be in the correct format or it will result in an error.
Post condition	Cars that meet the search criteria will show up.



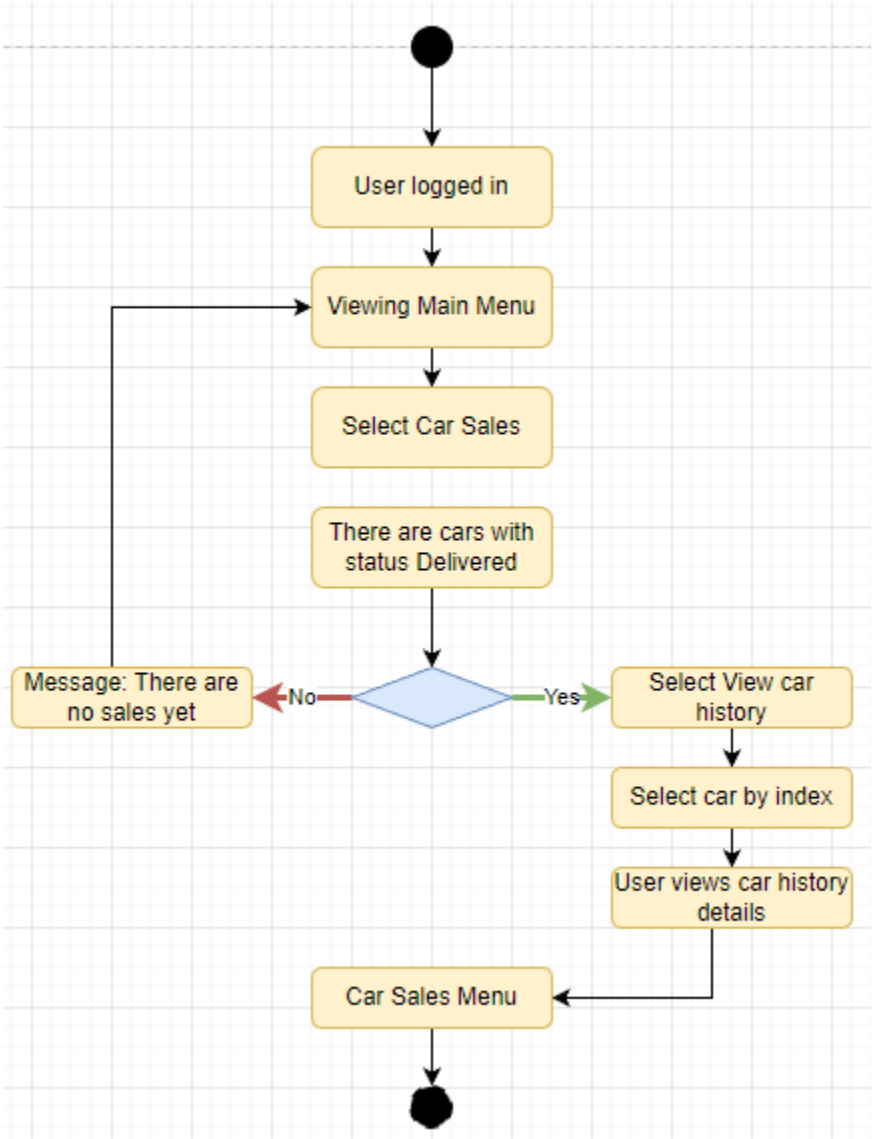
Use Case Number	22
User Case Name	User modifies car details
Overview	User can modify car details: status, price, warranty plans, mileage.
Actor(s)	Admin, General Employee
Description	If the dealership needs to change the car's attributes they can. If mileage increases, they can change this. If the value of the car decreases,they can lower it. These functions need to be editable to function properly for the dealership.

Pre-condition	User is logged in. Car is registered in the car's inventory. Car is not status Delivered.
Alternative	Format needs to be right in modified values.
Post condition	Car details are successfully modified by the user.

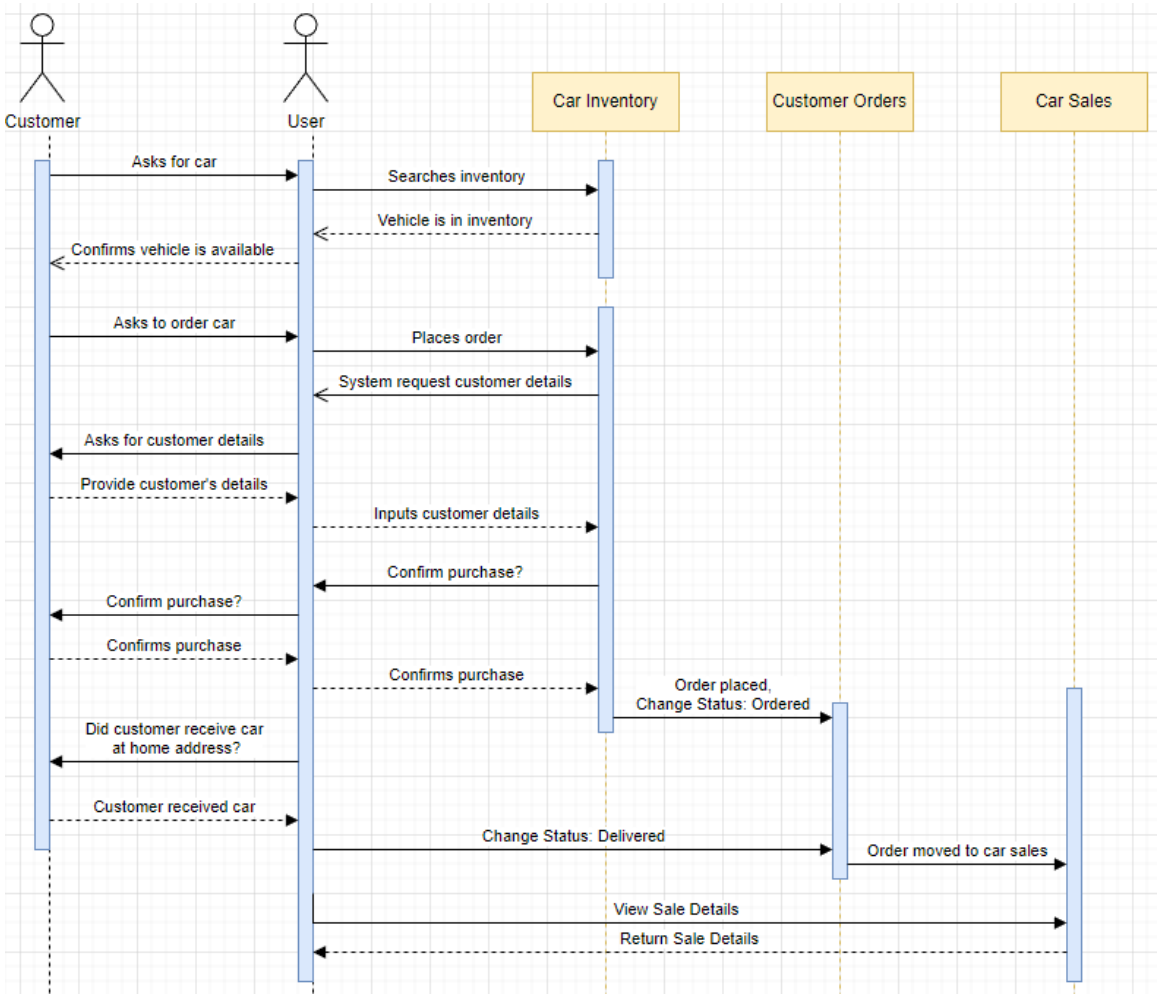


Use Case Number	23
User Case Name	User views car history
Overview	Users can view details of cars with status Delivered.
Actor(s)	Admin, General Employee

Description	Orders that go from status of Ordered to Delivered arrive in the Car Sales Menu. Here users can view any information related to that sale from the salesperson, customer, car details, and delivery date.
Pre-condition	User must be logged in. Car status must be Delivered. Car must initially be Ordered, then processed to Delivered.
Alternative	None.
Post condition	User can view details of car sale.



Sequence Diagram - customer order process



Independent Components

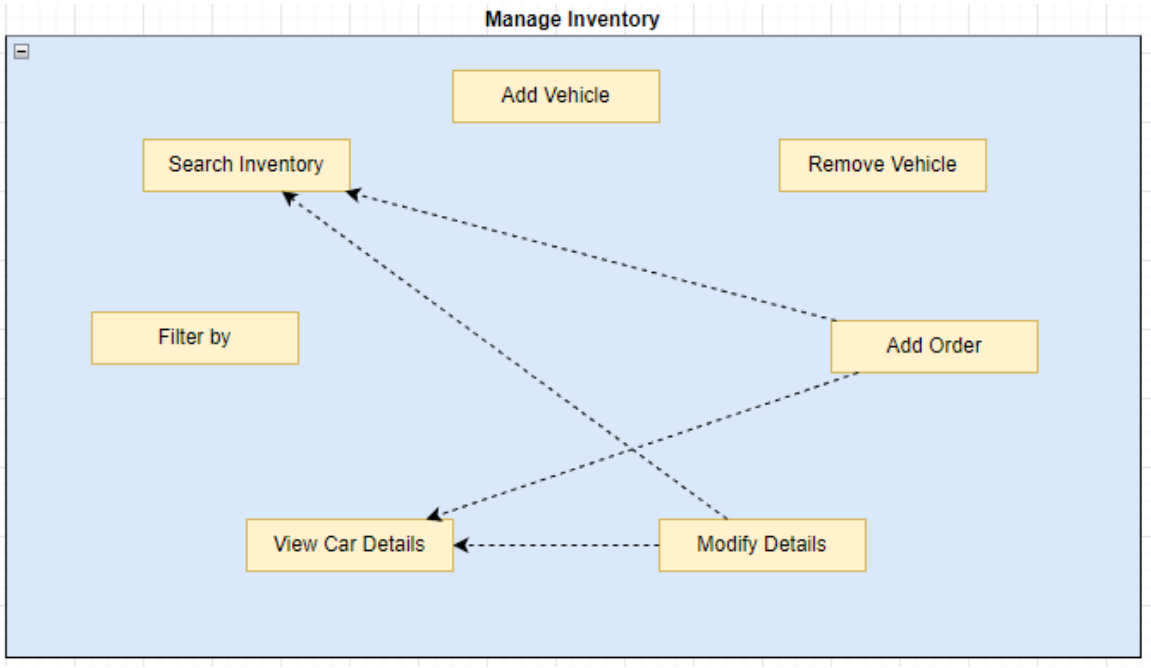
Coupling

With how the modules are split, the main goal is to limit the amount of coupling that would occur in our system. Most of the problems that may occur will be from writing to the json files, which is unavoidable for our purposes. Besides that, we can limit the amount of coupling to just control types to improve dependence.

<u>Manage Inventory Module</u>

Coupling type(s):

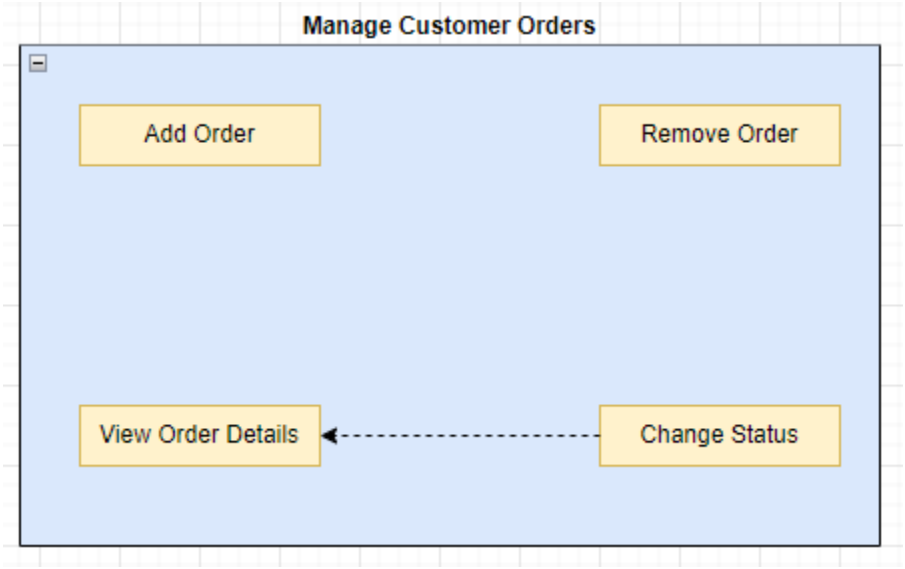
- Control coupling: Add Order - Search Inventory
- Control coupling: Add Order - View Car Details
- Control coupling: Modify Details - Search Inventory
- Control coupling: Modify Details - View Car Details



Manage Customer Orders Module

Coupling type(s):

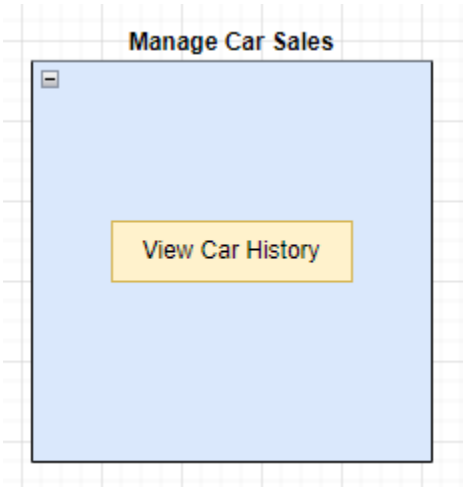
- Control coupling: Change Status - View Order Details



Manage Car Sales Module

Coupling type(s):

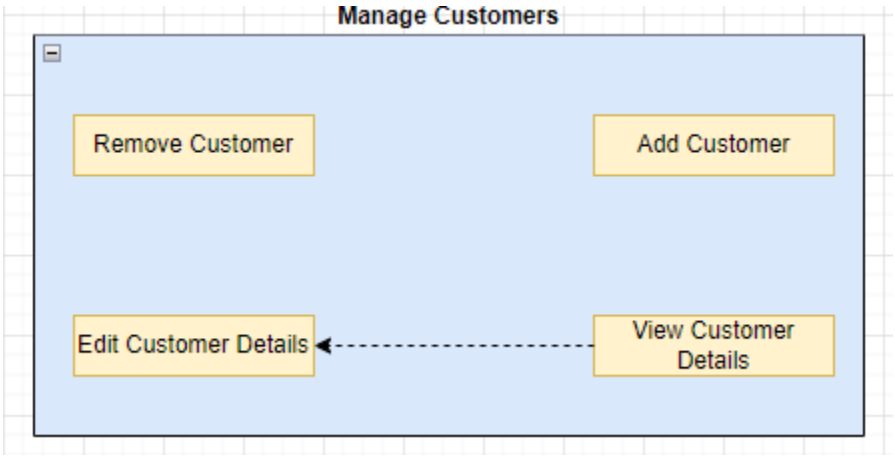
- No direct coupling



Manage Customers Module

Coupling type(s):

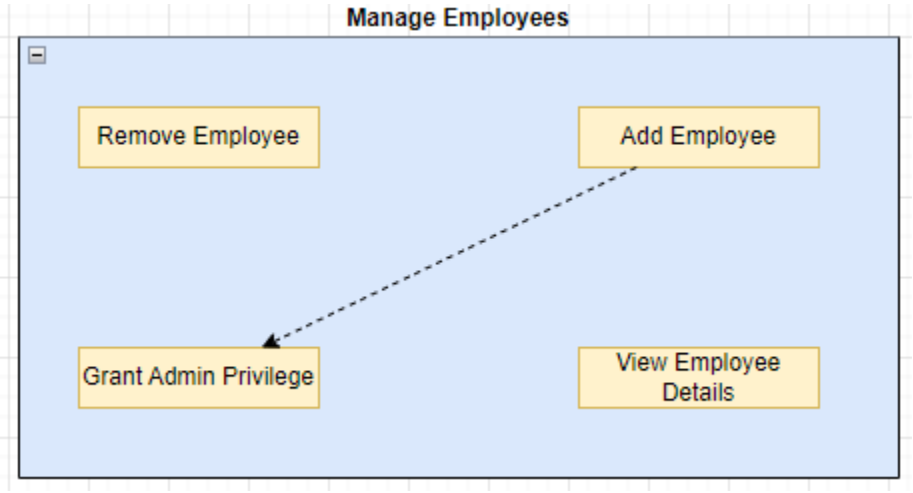
- Control coupling: View Customer Details - Edit Customer Details



Manage Employees Module

Coupling type(s):

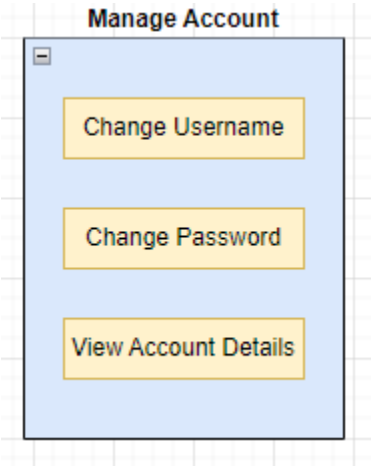
- Control coupling: Add Employee - Grant Admin Privilege



Manage Account Module

Coupling type(s):

- No direct coupling

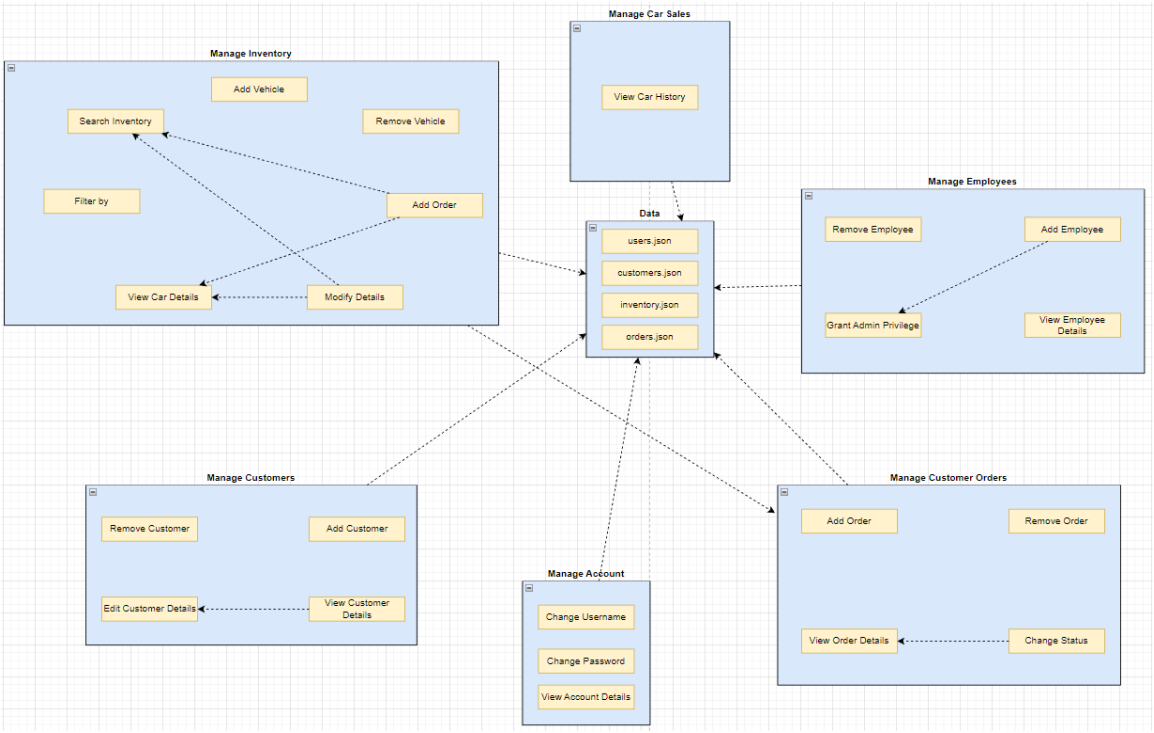


Total Vehicle Tracking System Module

External coupling occurs between the json files storing the data and the main function modules. This is because we are writing to these json files whether that be adding order, modifying order, adding customers, adding employees, and so forth. These modifications can affect formatting if not properly accounted for, so improper formatting when adding a new car may cause a fault when trying to view car details.

Coupling type(s):

- Control Coupling: Manage Inventory - Manage Customer Orders
- External Coupling: Manage Inventory - Data
- External Coupling: Manage Car Sales - Data
- External Coupling: Manage Employees - Data
- External Coupling: Manage Customer Orders - Data
- External Coupling: Manage Account - Data
- External Coupling: Manage Customers - Data



Cohesion

The system's goal is to have high cohesion in conjunction with low coupling. This is achieved by splitting the modules into managing the dealership's inventory, customer orders, customer sales, employees, customers, and user's account. The main functions being the inventory, customer orders, and customer sales. Unfortunately, we have limited potential for high cohesion and most of the modules have low cohesion. This is not the worst case though since we have low cohesion and low coupling. High cohesion will be accomplished when applicable.

Manage Inventory: logical cohesion. All elements are related to managing car inventory but do not attempt to accomplish a specific task.

- Add Vehicle
- Search Inventory
- Remove Vehicle
- Filter By
- Add Order
- View Car Details
- Modify Car Details

Manage Car Sales: no cohesion since one submodule.

- View Car History

Manage Customer Orders: logical cohesion. All elements are related to managing customer orders but do not attempt to accomplish a specific task.

- Add Order
- Remove Order
- View Order Details
- Change Status

Manage Customers: logical cohesion.All elements are related to managing customers but do not attempt to accomplish a specific task.

- Remove Customer
- Add Customer
- Edit Customer Details
- View Customer Details

Manage Employees: logical cohesion. There is functional cohesion between Add Employee and Grant Admin Privilege, working on the task of creating a new user for the system.

- Remove Employee
- Add Employee
- Grant Admin Privilege
- View Employee Details

Manage Account: logical cohesion. All elements are related to managing the user's account but do not attempt to accomplish a specific task.

- Change Username
- Change Password
- View Account Details

User Interface

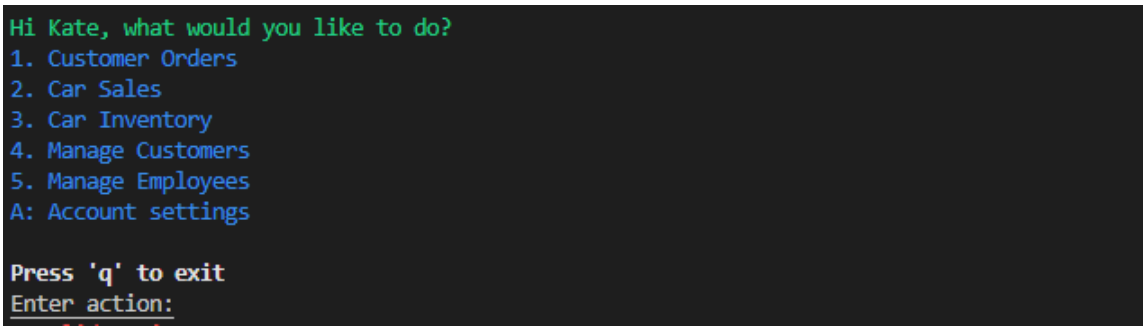
The type of text will dictate the text's formatting. Important information will be bold like "press 'q' to exit". Invalid messages or error messages will have a red color. Success messages will have a green color. Action messages, options users can choose from to use the function, will have a blue color. Warning messages like confirmation messages will be yellow. Purple messages are for menu names and continuing to the next page messages. All interfaces will follow these color styles.

Main function menus

The interfaces below are explained in depth in their own sections. These are the main module menus below for quick access:

Login Page

Main Menu



Customer Orders Menu

```
Order Menu
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action:
```

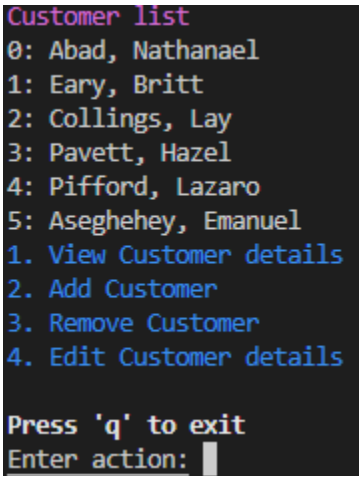
Car Sales Menu

```
Car Sales Menu

Delivered Cars:
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro

What would you like to do?

1. View sale details

Press 'q' to exit
Enter action:
```

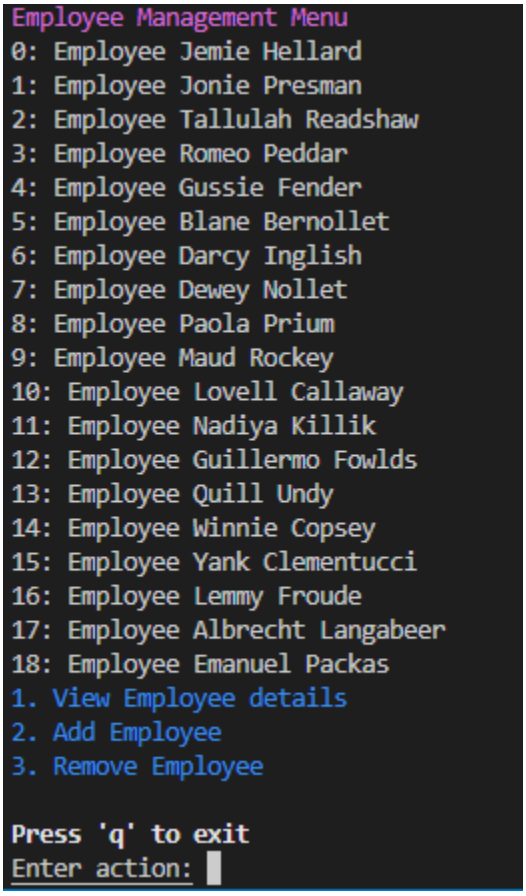
Car Inventory Menu

```
Inventory Menu
0: 1994 1500 Chevrolet $172,965.00 Status.AVAILABLE
1: 1987 Century Buick $149,805.00 Status.AVAILABLE
2: 2006 F350 Ford $184,781.00 Status.AVAILABLE
3: 2012 Crosstour Honda $78,700.00 Status.ORDERED
4: 2001 S80 Volvo $182,934.00 Status.ORDERED
5: 2006 A8 Audi $157,727.00 Status.DELIVERED
6: 2011 Range Rover Land Rover $190,839.00 Status.ORDERED
7: 1993 Trans Sport Pontiac $37,995.00 Status.AVAILABLE
8: 1997 Ram Van 1500 Dodge $75,620.00 Status.ORDERED
9: 1956 Corvette Chevrolet $99,672.00 Status.AVAILABLE
10: 2000 Echo Toyota $181,801.00 Status.AVAILABLE
11: 1999 Impreza Subaru $191,478.00 Status.AVAILABLE
12: 1992 Suburban 2500 GMC $21,262.00 Status.AVAILABLE
13: 2009 Colorado Chevrolet $133,215.00 Status.AVAILABLE
14: 2009 Savana 2500 GMC $69,003.00 Status.AVAILABLE
15: 1995 VS Commodore Holden $97,612.00 Status.AVAILABLE
16: 2002 Optima Kia $55,901.00 Status.AVAILABLE
17: 2012 Patriot Jeep $127,583.00 Status.AVAILABLE
18: 1972 Thunderbird Ford $15,696.00 Status.AVAILABLE
19: 2009 M-Class Mercedes-Benz $118,667.00 Status.AVAILABLE
20: 2003 Yukon GMC $35,646.00 Status.AVAILABLE
21: 1992 164 Alfa Romeo $172,583.00 Status.AVAILABLE
22: 1996 Millenia Mazda $64,063.00 Status.AVAILABLE
23: 1995 Esteem Suzuki $100,522.00 Status.AVAILABLE
24: 1989 E-Series Ford $18,209.00 Status.AVAILABLE
25: 1994 Crown Victoria Ford $35,398.00 Status.AVAILABLE
26: 2002 Sportage Kia $63,014.00 Status.AVAILABLE
27: 1984 Grand Marquis Mercury $165,472.00 Status.AVAILABLE
28: 2001 Leganza Daewoo $197,844.00 Status.AVAILABLE
29: 2006 Element Honda $76,873.00 Status.AVAILABLE
30: 1997 SL-Class Mercedes-Benz $165,560.00 Status.AVAILABLE
31: 2009 Cooper Clubman MINI $183,430.00 Status.AVAILABLE
32: 2001 Tiburon Hyundai $86,164.00 Status.AVAILABLE
33: 2010 Commander Jeep $100,402.00 Status.AVAILABLE
34: 2009 Camry Toyota $26,096.00 Status.AVAILABLE
35: 2006 Continental GT Bentley $98,656.00 Status.AVAILABLE
36: 1988 RX-7 Mazda $159,604.00 Status.AVAILABLE
37: 2008 RDX Acura $5,847.00 Status.AVAILABLE
38: 2008 Ram Dodge $148,258.00 Status.AVAILABLE
39: 1992 MX-5 Mazda $176,840.00 Status.AVAILABLE
40: 2003 Grand Caravan Dodge $46,516.00 Status.AVAILABLE
41: 1998 Prizm Chevrolet $7,910.00 Status.AVAILABLE
42: 1993 Rally Wagon 1500 GMC $27,861.00 Status.AVAILABLE
43: 1995 Previa Toyota $56,935.00 Status.AVAILABLE
44: 2007 207 Peugeot $135,478.00 Status.AVAILABLE
45: 2005 Legacy Subaru $86,802.00 Status.AVAILABLE
46: 2009 Borrego Kia $64,791.00 Status.AVAILABLE
47: 1992 Ram Van B250 Dodge $62,573.00 Status.AVAILABLE
48: 2003 XC90 Volvo $64,008.00 Status.AVAILABLE
49: 2010 Rogue Nissan $197,771.00 Status.ORDERED
50: 2012 Z Tesla $6,789,837,298,341.00 Status.ORDERED
0. View car details
1. Search
2. Filter by
3. Make a customer order
4. Add/Remove Cars

Press 'q' to exit
Enter action:
```

Manage Customers Menu



Manage Employees Menu



Account settings Menu

```
Account settings
1. Change password
2. Change username
3. View Account Details

Press 'q' to exit
Enter action:
```

Startup Interface

Login Page

Correct login credentials will allow user access to the system.

```
Login page

Welcome to PigeonBox
Enter username: test
Enter password: test
```

Login Page - Incorrect Credentials

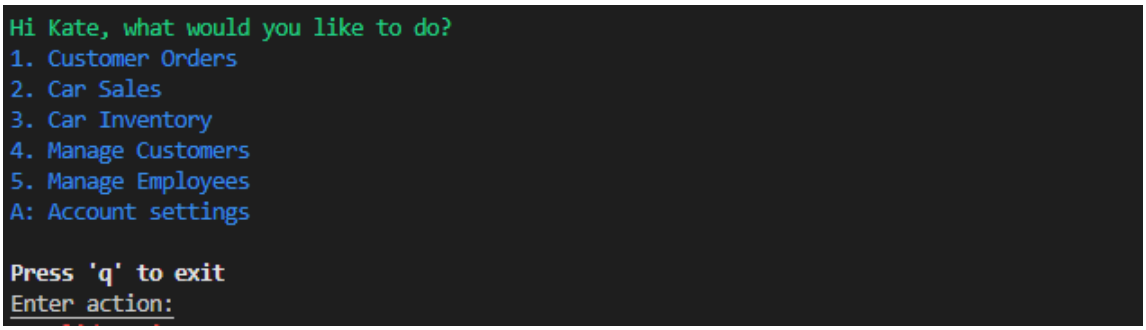
If username and password do not match an existing user account the system will prompt re-entry. The user gets three attempts before the system force shuts down.

```
Login page

Welcome to PigeonBox
Enter username:
Enter password:

Attempt 2
Enter username:
Enter password:

Attempt 3
Enter username:
Enter password:

Failed all 3 attempts, sorry
```

Main Menu Interface

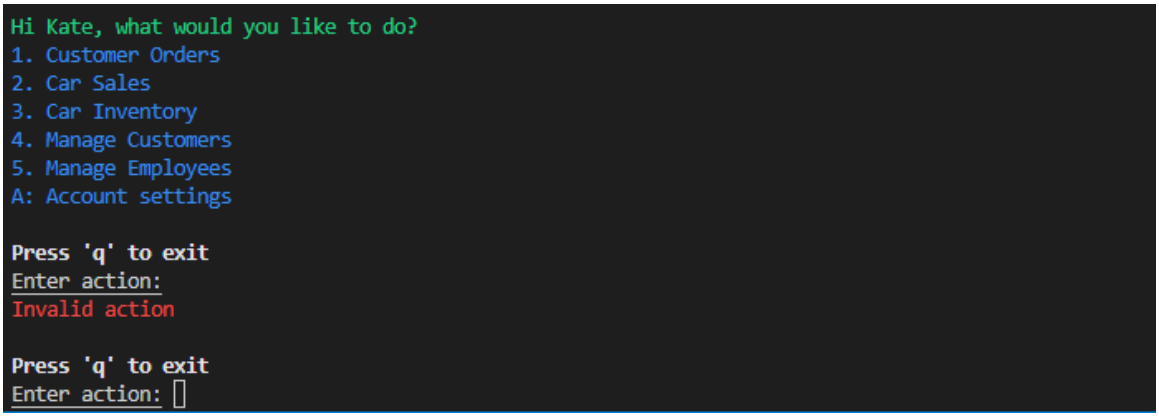
Main Menu

The main menu will display Customer Orders, Car Sales, Car Inventory, Manage Customers, Manage Employees, and Account Settings. Users will have the option to press 'q' to exit the system, prompting the user to either shut down the system or log back in.



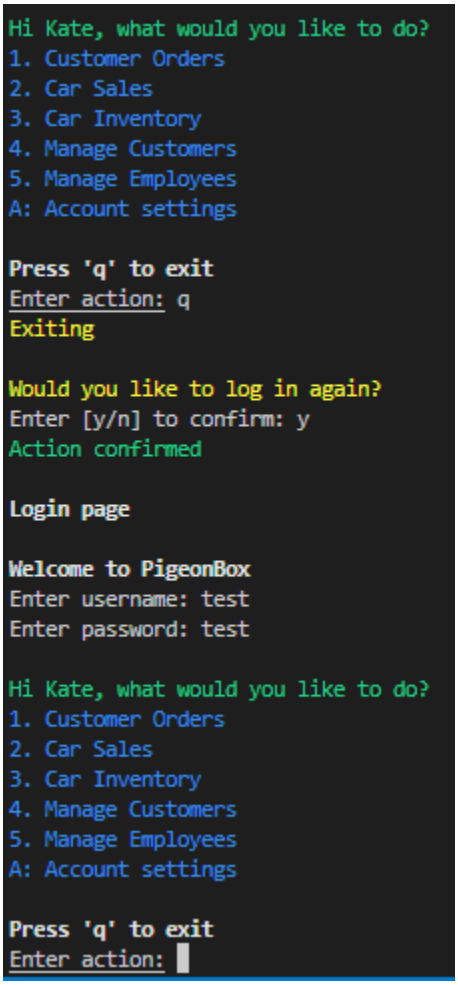
Main Menu - Invalid Action

If the action is not one of the indexes to choose from or q to exit, the action is invalid and won't be accepted.



System Log out/ Log back in

Users when attempting to exit the system will be asked if they wish to shut down the system or log back in. Confirming log back in will bring users back to the login page.



System Shutdown

Instead of logging back in, the user will confirm no, shutting down PigeonBox.

```
Login page

Welcome to PigeonBox
Enter username: test
Enter password: test

Hi Kate, what would you like to do?
1. Customer Orders
2. Car Sales
3. Car Inventory
4. Manage Customers
5. Manage Employees
A: Account settings

Press 'q' to exit
Enter action: q
Exitting

[False, False, False, False]
Would you like to log in again?
Enter [y/n] to confirm: n
PS H:\USF\2022-2023\USF Spring2023\SoftwareEngineering\VTS\vehicle-tracking-system-main\src>
```

Customer Order Interface

Customer Order Menu

Users will have a view of all orders in the system. These orders will show the order number, the user who sold the car, the car make and model, and the customer's first and last name. Users can have the option of adding a new order, removing orders, viewing order details or going back to the Main Menu.

```
Order Menu
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action:
```

Customer Order Menu - No Orders

In the case that there are no orders on the system, there will be no list of current orders.

```
Order Menu
No data to display
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action:
```

Add Order

Select Add Order will prompt the user to select from a list of cars in the dealership's inventory. This is chosen by index value. Users will also have to confirm this order and if it is for a new customer. If it is for an existing customer, a list of all customers on the system will be shown to choose from. If it is a new customer, the user will go through the add customer process.

```
47: 1992 Ram Van B250 Dodge $62,573.00 Status.AVAILABLE
48: 2003 XC90 Volvo $64,008.00 Status.AVAILABLE
49: 2010 Rogue Nissan $197,771.00 Status.ORDERED
50: 2012 Z Tesla $6,789,837,298,341.00 Status.ORDERED

Pick index from the dislayed list above
Enter index [0-50] OR enter 'q' to exit: 47

Picked: 1992 Ram Van B250 Dodge $62,573.00 Status.AVAILABLE

You are about to order this car: 1992 Ram Van B250 Dodge $62,573.00 Status.AVAILABLE

Are you sure you want to proceed?
Enter [y/n] to confirm: y
Action confirmed
Is this order for a new customer?
Enter [y/n] to confirm: n
0: Abad, Nathanael
1: Eary, Britt
2: Collings, Lay
3: Pavett, Hazel
4: Pifford, Lazaro
5: Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit: 5

Picked: Aseghehey, Emanuel
Order #8 Made by Admin Kate Anderson: Dodge Ram Van B250 for Aseghehey, Emanuel

Press any key to continue
```

Add Order - Failure

This is similar to Add Order's interface but will show an error message if the order fails the process.

```
Picked: Eary, Britt
Failed to make order. This car has already been ordered by someone else.

Press any key to continue
```

Remove Order

When selecting remove order, the user will see a list of all orders in the system and choose this by typing in its index value. Users will see which order is picked and will have a confirmation message to delete the order. Typing 'y' will delete the order and 'n' will cancel this process.

```
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
6: Order #8 Made by Admin Kate Anderson: Dodge Ram Van B250 for Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-6] OR enter 'q' to exit: 6

Picked: Order #8 Made by Admin Kate Anderson: Dodge Ram Van B250 for Aseghehey, Emanuel
Are you sure you want to remove this order: Order #8 Made by Admin Kate Anderson: Dodge Ram Van B250 for Aseghehey, Emanuel?
Enter [y/n] to confirm: y
Action confirmed

Press any key to continue
```

View Order Details

Selecting View order details will reprint the list of orders and ask users to pick an index. Choosing a correct index will display the details of that car order: customer details, employee details, car details.

```
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action: 3
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit: 0

Picked: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael

Car:
2001 S80 Volvo $182,934.00 Status.ORDERED
JTJHY7AX1E4512122 with Sports package
        Performance
        Engine: Electric, Transmission: Manual
        Mileage: 5207 miles
        Design
        Interior design: ['Power Mirrors', 'Power Locks', 'Side Airbags', 'Universal Garage Door Opener']
        Exterior design: [{'paint': 'Pearlescent', 'extra': ['Daytime Running Lights', 'Power Hatch/Deck Lidis', 'ABS Brakes']}]
        Extras
        Comfort: ['Cruise Control', 'Bluetooth Technology', 'Sunroof(s)', 'Adjustable seats', 'Leather seats']
        Entertainment ['AM/FM Stereo', 'Auxiliary Audio Input', 'Satellite Radio Ready']
        Protection plans
        Maintenance 10-year Oil Change maintenance
        Warranty plans ['30 day money back guarantee (up to 1500 miles)', 'Unlimited Theft Protection', 'Unlimited Tire & Wheel Protection']

Sales by: Employee Jemie Hellard

Customer
Email: aiannetti0@umich.edu
Address: 7 Texas Terrace
List of all orders: [2001 S80 Volvo $182,934.00 Status.ORDERED]

Would you like to change the order status?
Enter [y/n] to confirm:
```

View Order Details - Invalid Index

If the index chosen to view details is invalid, the user will be notified with a red error message and prompted to select a correct index. Users also have the choice to cancel the action by pressing q.

```
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit: 6
Invalid index! Must be greater than 0 AND smaller than maximum length

Press any key to continue
0: Order #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: Order #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: Order #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: Order #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: Order #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit:
```

Change Order Status - Delivered

Once the dealership confirms the car has arrived at the customer's residence, a user will change the order status from Ordered to Delivered.

```
Picked: Order #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt

Car:
2006 A8 Audi $157,727.00 Status.ORDERED
1D7RE5GK1BS514059 with Economic package
        Performance
        Engine: 4 Cylinder, Transmission: Automatic
        Mileage: 2670 miles
        Design
        Interior design: ['Power Windows', 'Power Mirrors', 'Side Airbags', 'Power Seat(s)']
        Exterior design: [{'paint': 'Pearlescent', 'extra': ['ABS Brakes', 'Turbo Charged Engine']}]
        Extras
        Comfort: ['Lane Departure Warning', 'Navigation System', 'Front Seat Heaters', 'Adjustable seats']
        Entertainment ['SiriusXM Trial Available', 'Meridian Audio', 'Satellite Radio Ready']
        Protection plans
        Maintenance 10-year Oil Change maintenance
        Warranty plans ['Unlimited Tire & Wheel Protection', 'Unlimited Windshield Protection']

Sales by: Employee Jemie Hellard

Customer
Email: wohalloran1@epa.gov
Address: 296 Talmadge Point
List of all orders: [2006 A8 Audi $157,727.00 Status.ORDERED]

Would you like to change the order status?
Enter [y/n] to confirm: y
Action confirmed
0: Available
1: Ordered
2: BackOrder
3: Delivered

Press 'q' to exit
Pick a status: 3
2006 A8 Audi $157,727.00 Status.DELIVERED
```

Car Sales Interface

Car Sales Menu

Users can view a list of all customer orders that have been fully processed. This means the car has the status of delivered and is in final possession of the customer. All processed orders are considered sales. This page will be used in future updates for managing these cars and keeping their history. Future updates may include maintenance systems and future scheduling for the car owners. At launch, the only available option is to view the order's details: customer information, salesperson information, car information, and the delivery date.

```
Car Sales Menu

Delivered Cars:
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro

What would you like to do?

1. View sale details

Press 'q' to exit
Enter action:
```

View Car History

Users will select the sale's details by choosing from an index of choices. Here the user can view the car details, employee details, customer details, and delivery date of the car. Future updates may include maintenance checks.

```
Picked: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael

Car:
2001 S80 Volvo $182,934.00 Status.DELIVERED
JTJHY7AX1E4512122 with Sports package
        Performance
        Engine: Electric, Transmission: Manual
        Mileage: 5207 miles
        Design
        Interior design: ['Power Mirrors', 'Power Locks', 'Side Airbags', 'Universal Garage Door Opener']
        Exterior design: [{'paint': 'Pearlescent', 'extra': ['Daytime Running Lights', 'Power Hatch/Deck Lidis', 'ABS Brakes']}]
        Extras
        Comfort: ['Cruise Control', 'Bluetooth Technology', 'Sunroof(s)', 'Adjustable seats', 'Leather seats']
        Entertainment ['AM/FM Stereo', 'Auxiliary Audio Input', 'Satellite Radio Ready']
        Protection plans
        Maintenance 10-year Oil Change maintenance
        Warranty plans ['30 day money back guarantee (up to 1500 miles)', 'Unlimited Theft Protection', 'Unlimited Tire & Wheel Protection']

Sales by: Employee Jemie Hellard

Customer
Email: aiannetti0@umich.edu
Address: 7 Texas Terrace
List of all orders: [2001 S80 Volvo $182,934.00 Status.DELIVERED]

Delivery date: 2023-04-16

Press any key to continue
```

Car Inventory Interface

Car Inventory Menu

The Car Inventory Menu will display a list of all cars and their status in the dealerships' inventory. The car's price, year, make, and model will be displayed with a corresponding index value. These cars are organized in the order they were added into the system. Users will have the option to View car details, Search, Filter by, Make a customer order, Add/Remove cars, or 'q' to exit.

```
Inventory Menu
0: 1994 1500 Chevrolet $172,965.00 Status.AVAILABLE
1: 1987 Century Buick $149,805.00 Status.AVAILABLE
2: 2006 F350 Ford $184,781.00 Status.AVAILABLE
3: 2012 Crosstour Honda $78,700.00 Status.ORDERED
4: 2001 S80 Volvo $182,934.00 Status.ORDERED
5: 2006 A8 Audi $157,727.00 Status.DELIVERED
6: 2011 Range Rover Land Rover $190,839.00 Status.ORDERED
7: 1993 Trans Sport Pontiac $37,995.00 Status.AVAILABLE
8: 1997 Ram Van 1500 Dodge $75,620.00 Status.ORDERED
9: 1956 Corvette Chevrolet $99,672.00 Status.AVAILABLE
10: 2000 Echo Toyota $181,801.00 Status.AVAILABLE
11: 1999 Impreza Subaru $191,478.00 Status.AVAILABLE
12: 1992 Suburban 2500 GMC $21,262.00 Status.AVAILABLE
13: 2009 Colorado Chevrolet $133,215.00 Status.AVAILABLE
14: 2009 Savana 2500 GMC $69,003.00 Status.AVAILABLE
15: 1995 VS Commodore Holden $97,612.00 Status.AVAILABLE
16: 2002 Optima Kia $55,901.00 Status.AVAILABLE
17: 2012 Patriot Jeep $127,583.00 Status.AVAILABLE
18: 1972 Thunderbird Ford $15,696.00 Status.AVAILABLE
19: 2009 M-Class Mercedes-Benz $118,667.00 Status.AVAILABLE
20: 2003 Yukon GMC $35,646.00 Status.AVAILABLE
21: 1992 164 Alfa Romeo $172,583.00 Status.AVAILABLE
22: 1996 Millenia Mazda $64,063.00 Status.AVAILABLE
23: 1995 Esteem Suzuki $100,522.00 Status.AVAILABLE
24: 1989 E-Series Ford $18,209.00 Status.AVAILABLE
25: 1994 Crown Victoria Ford $35,398.00 Status.AVAILABLE
26: 2002 Sportage Kia $63,014.00 Status.AVAILABLE
27: 1984 Grand Marquis Mercury $165,472.00 Status.AVAILABLE
28: 2001 Leganza Daewoo $197,844.00 Status.AVAILABLE
29: 2006 Element Honda $76,873.00 Status.AVAILABLE
30: 1997 SL-Class Mercedes-Benz $165,560.00 Status.AVAILABLE
31: 2009 Cooper Clubman MINI $183,430.00 Status.AVAILABLE
32: 2001 Tiburon Hyundai $86,164.00 Status.AVAILABLE
33: 2010 Commander Jeep $100,402.00 Status.AVAILABLE
34: 2009 Camry Toyota $26,096.00 Status.AVAILABLE
35: 2006 Continental GT Bentley $98,656.00 Status.AVAILABLE
36: 1988 RX-7 Mazda $159,604.00 Status.AVAILABLE
37: 2008 RDX Acura $5,847.00 Status.AVAILABLE
38: 2008 Ram Dodge $148,258.00 Status.AVAILABLE
39: 1992 MX-5 Mazda $176,840.00 Status.AVAILABLE
40: 2003 Grand Caravan Dodge $46,516.00 Status.AVAILABLE
41: 1998 Prizm Chevrolet $7,910.00 Status.AVAILABLE
42: 1993 Rally Wagon 1500 GMC $27,861.00 Status.AVAILABLE
43: 1995 Previa Toyota $56,935.00 Status.AVAILABLE
44: 2007 207 Peugeot $135,478.00 Status.AVAILABLE
45: 2005 Legacy Subaru $86,802.00 Status.AVAILABLE
46: 2009 Borrego Kia $64,791.00 Status.AVAILABLE
47: 1992 Ram Van B250 Dodge $62,573.00 Status.AVAILABLE
48: 2003 XC90 Volvo $64,008.00 Status.AVAILABLE
49: 2010 Rogue Nissan $197,771.00 Status.ORDERED
50: 2012 Z Tesla $6,789,837,298,341.00 Status.ORDERED
0. View car details
1. Search
2. Filter by
3. Make a customer order
4. Add/Remove Cars

Press 'q' to exit
Enter action:
```

View Car Details

Selecting view car details, the system will display the list of all available cars again for users to choose an index. Once this occurs users will be able to see all car details.

```
Pick index from the dislayed list above
Enter index [0-50] OR enter 'q' to exit: 1

Picked: 1987 Century Buick $149,805.00 Status.AVAILABLE

1987 Century Buick $149,805.00 Status.AVAILABLE
4F2CY0C73AK949059 with Sports package
        Performance
        Engine: V8, Transmission: Manual
        Mileage: 7799 miles
        Design
        Interior design: ['Side Airbags', 'Power Locks', 'Power Windows', 'Power Seat(s)', 'Power Mirrors']
        Exterior design: [{'paint': 'Matte', 'extra': ['Power Hatch/Deck Lidis', 'Alloy Wheels', 'Daytime Running Lights', 'Turbo Charged Engine']}]
        Extras
        Comfort: ['Memory Seats', 'Lane Departure Warning', 'Rear View Camera', 'Sunroof(s)']
        Entertainment ['Auxiliary Audio Input', 'SiriusXM Trial Available', 'Meridian Audio']
        Protection plans
        Maintenance 3-year/36.000-mile scheduled maintenance
        Warranty plans ['2-year Dent Protection', 'Unlimited Theft Protection', 'Unlimited Windshield Protection']

Do you want to modify its details (price, warranty plans, mileage, or status)?
Enter [y/n] to confirm:
```

Modify Details

When modifying car details, users can choose to change the car status, price, mileage, and warranty plans.

```
Do you want to modify its details (price, warranty plans, mileage, or status)?
Enter [y/n] to confirm: y
Action confirmed
1987 Century Buick $149,805.00 Status.AVAILABLE
1. Change car status
2. Change car price
3. Change car mileage
4. Change car warranty plans

Press 'q' to exit
Enter action:
```

Search

A successful search will result in the car's details showing up with the option to order the car or modify its details.

```
Search car in inventory
Enter model, make and year separated by commas
Z,Tesla,2012

Car details:
2012 Z Tesla $6,789,837,298,341.00 Status.ORDERED
jhkbewgu3 with sports package
        Performance
        Engine: 4-cylinder, Transmission: Hybrid
        Mileage: 1023 miles
        Design
        Interior design: ['nice']
        Exterior design: [{'paint': 'translucent', 'extra': ['cheap']}]
        Extras
        Comfort: ['no comfort']
        Entertainment ['bluetooth']
        Protection plans
        Maintenance no maint
        Warranty plans ['no warr']
1. Order car
2. Modify car (status, price, mileage, warranty plans)

Press 'q' to exit
Enter action:
```

Search - No Match

A failed search where a car does not match the search criteria will result in a red message saying no match.

```
Search car in inventory
Enter model, make and year separated by commas
test,test,1

No car match :(

Press any key to continue
```

Filter by

Users are able to filter by available, ordered, backordered, or delivered here. If any cars match that status, a list similar to the Inventory Menu will show up. If no cars match that status, a message saying “No data to display” will appear.

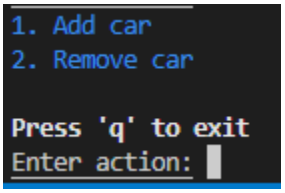
```
Filter by Status:
1. Available
2. Ordered
3. Backorder
4. Delivered

Press 'q' to exit
Enter action:
```

Make a Customer Order

This has the same interface as Add Order in Customer Orders Menu. For a detailed image, please refer to Add Order in the Customer Orders Interface section.

Add/Remove Cars

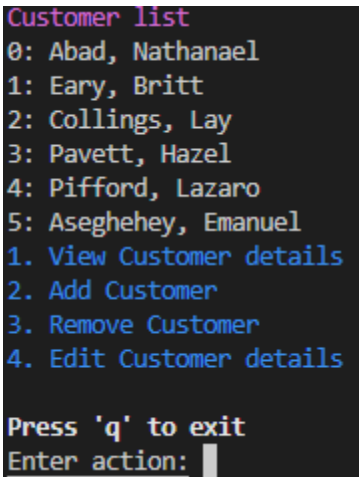
This menu is simple and allows users to add or remove cars. Adding a car will prompt the user to fill in all attributes of that car and removing a car will prompt the user to select from a list of all available cars in the inventory.



Manage Customers Interface

Manage Customers Menu

Users will see a list of all customers in the system and have the option to view customer details, add customers, remove customers, edit customer details, or exit.



View Customer Details

Users will see a list of all customers and after choosing a customer to view, they will see all details relating to that customer.

```
0: Abad, Nathanael
1: Eary, Britt
2: Collings, Lay
3: Pavett, Hazel
4: Pifford, Lazaro
5: Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit: 1

Picked: Eary, Britt

Email: wohalloran1@epa.gov
Address: 296 Talmadge Point
List of all orders: [2006 A8 Audi $157,727.00 Status.DELIVERED]
Would you like to update customer details?
Enter [y/n] to confirm:
```

Add Customer

Users will enter customer details and if the value entered doesn't match they will get a red error message.

```
Enter customer details


Press 'q' to exit
Enter First Name: testname
testname

Press 'q' to exit
Enter Last Name: testname
testname

Press 'q' to exit
Enter email address: test@gmail.com
test@gmail.com
Credit Card #: 0123456789123456

Press 'q' to exit
Enter Home address: testaddress
testaddress

Added testname, testname with success

Press any key to continue
```

Remove Customer

Users will choose from a list of all customers and will be prompted to confirm deletion.

```
0: Abad, Nathanael
1: Eary, Britt
2: Collings, Lay
3: Pavett, Hazel
4: Pifford, Lazaro
5: Aseghehey, Emanuel
6: testname, testname

Pick index from the dislayed list above
Enter index [0-6] OR enter 'q' to exit: 6

Picked: testname, testname

Are you sure you want to delete testname, testname
Enter [y/n] to confirm: y
Action confirmed
Removed customer successfully

Press any key to continue
```

Edit Customer Details

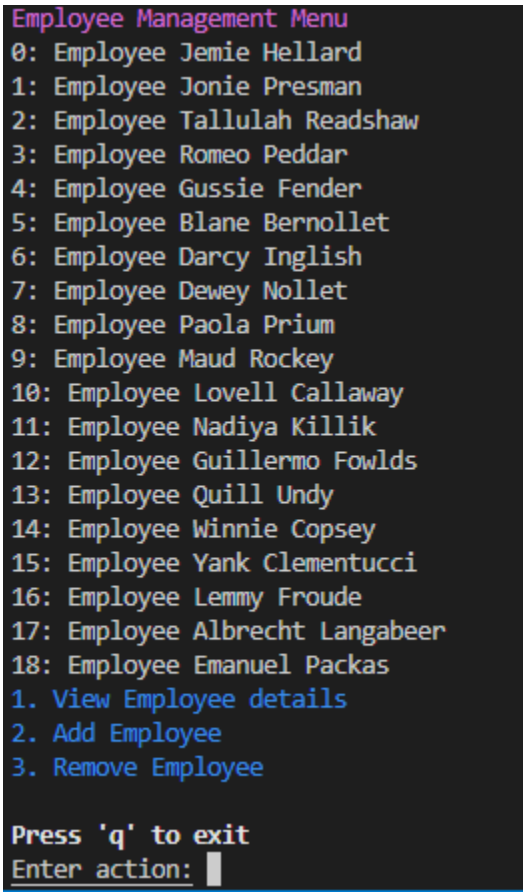
Users will choose from a list of all customers in the system and can edit the customer's home address, email address, or card number.

```
0: Abad, Nathanael
1: Eary, Britt
2: Collings, Lay
3: Pavett, Hazel
4: Pifford, Lazaro
5: Aseghehey, Emanuel

Pick index from the dislayed list above
Enter index [0-5] OR enter 'q' to exit: 0

Picked: Abad, Nathanael
1. Update home address
2. Update email address
3. Update card details

Press 'q' to exit
Enter action:
```

Manage Employees Interface

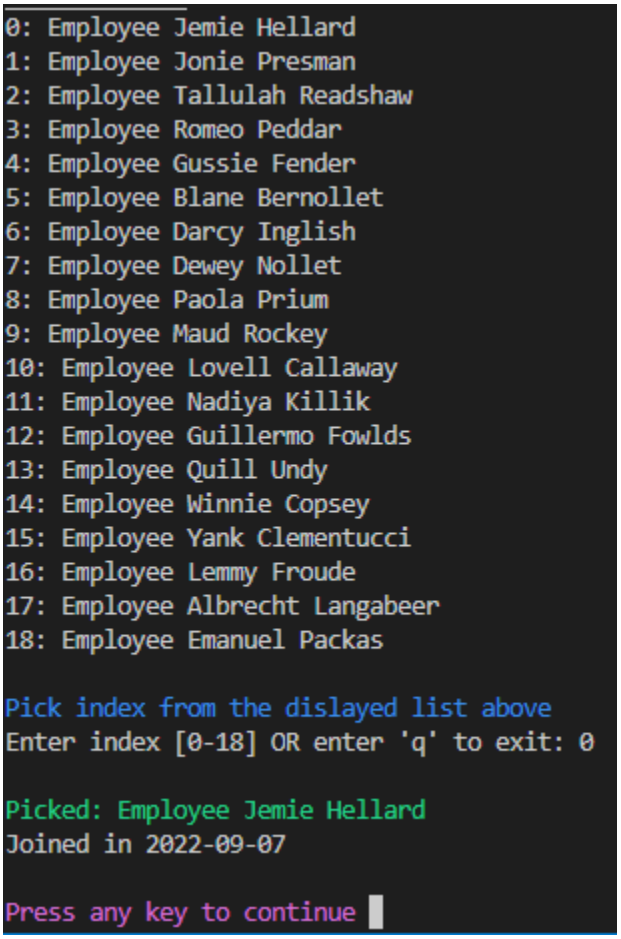
Manage Employees Menu

Users will see a list of all employees registered and can view employee details, add employee, remove employee, or exit.



View Employee Details

Users can see a list of all employees to choose from and the system will return the employee picked and its details.



Add Employee

Users will have green confirmation for entered values. Red error messages will appear if values don't match input criteria. Confirmation will appear yellow.

```
Press 'q' to exit
Enter Enter username: testname
testname

Press 'q' to exit
Enter Enter password for new user testname: testpass
testpass

Press 'q' to exit
Enter Enter first name: testname
testname

Press 'q' to exit
Enter Enter last name: testname
testname
Do you wish to grant Admin priviledges to testname?
Enter [y/n] to confirm: n
User ('testname', 'testname') successfully added

Press any key to continue
```

Remove Employee

Users will see a list of all current employees and can choose from index to delete. Users will confirm deletion.

```
0: Employee Jemie Hellard
1: Employee Jonie Presman
2: Employee Tallulah Readshaw
3: Employee Romeo Peddar
4: Employee Gussie Fender
5: Employee Blane Bernollet
6: Employee Darcy Inglish
7: Employee Dewey Nollet
8: Employee Paola Prium
9: Employee Maud Rockey
10: Employee Lovell Callaway
11: Employee Nadiya Killik
12: Employee Guillermo Fowlds
13: Employee Quill Undy
14: Employee Winnie Copsey
15: Employee Yank Clementucci
16: Employee Lemmy Froude
17: Employee Albrecht Langabeer
18: Employee Emanuel Packas
19: Employee testname testname

Pick index from the dislayed list above
Enter index [0-19] OR enter 'q' to exit: 13

Picked: Employee Quill Undy

Are you sure you want to delete Employee Quill Undy
Enter [y/n] to confirm: y
Action confirmed
Removed employee successfully

Press any key to continue
```

Account Settings Interface

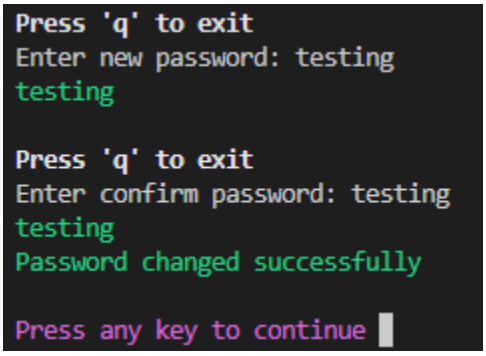
Account Settings Menu

Users can change password, username ,view account details, or exit.

```
Account settings
1. Change password
2. Change username
3. View Account Details

Press 'q' to exit
Enter action:
```

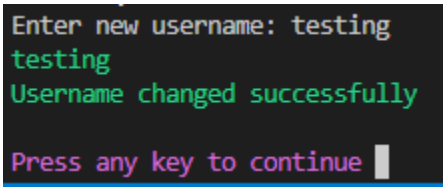
Change Password

Changing password will require a second typing of the new password for confirmation. If successful like below, it will appear like the image. If unsuccessful users will see a red error message.



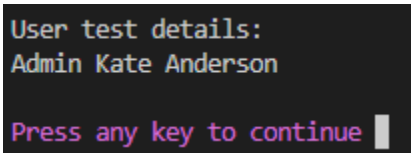
Change Username

Changing username is very straightforward as you can see below.



View Account Details

Users can view their username, first name, and last name.



PROGRAM DESIGN

Standards and Styles

Header Block Formatting

Each script should have a header block at the top of the file. Each script should follow the outline for the header block discussed below:

- What component is called?
  - Name of component
- Who wrote the component?
  - Block that identifies the writer with their email address.
  - Maintenance and test teams can contact the writer with questions or comments.
- Where does the component fit in the general system?
  - Purpose of the functions and how it fits in the whole design
- When was the component written or revised?
  - For fault correction, the requirements change and grow.
  - Keep track of revisions, keep a log of changes and who made them.
- Why does the component exist?
  - General purpose of the code and its functionality.
- Who uses its data structures and algorithms?
  - Names of major data structures and variables.

- Brief description of logic flow algorithms and error handling.

Header Block Formatting Example

```
################################################################################################################
'''
[replace text and delete in bracket]
PROGRAM [name]:  [purpose of code and function. brief]

PROGRAMMER: [firstName] [lastName] [email]

VERSION 1: written [day] [month] 2023 by [firstInitial]. [lastName]
REVISION [revision# ex: 1.1]: [day] [month] 2023 by [firstInitial]. [lastName] to [purpose of revision]


PURPOSE:
[general purpose of code and each functionality. thorough description]

DATA STRUCTURES:
[major data structures and variables]
[ex: variable LENGTH - integer]

ALGORITHM:
[brief description of logic flow]

ERROR HANDLING:
[brief description error handling]
'''
################################################################################################################
```

Comment's Standards

Comments should follow the standards discussed below:

- Comment structuring should have the comment one line above the described code.
- Variable names and statement labels should be commented on.
- Major activities should be commented on as phrases.

Each function should contain the following:

- a one-line description for fast browsing(found in function as docstring).
- a general description(found in documentation_functionDescription.txt).
- typical calling examples(found in documentation_functionDescription.txt).
- accessibility(found in documentation_functionDescription.txt).
- function prototype(found in documentation_functionDescription.txt).
- proper code structuring.
- proper commenting on each important code fragment.

Data Structures

The most significant data structures used for the system are simple lists and dictionaries. These are used to store user's input, mainly for adding vehicles into the system which contain long lists of strings. These are implemented already through python as built in structures.

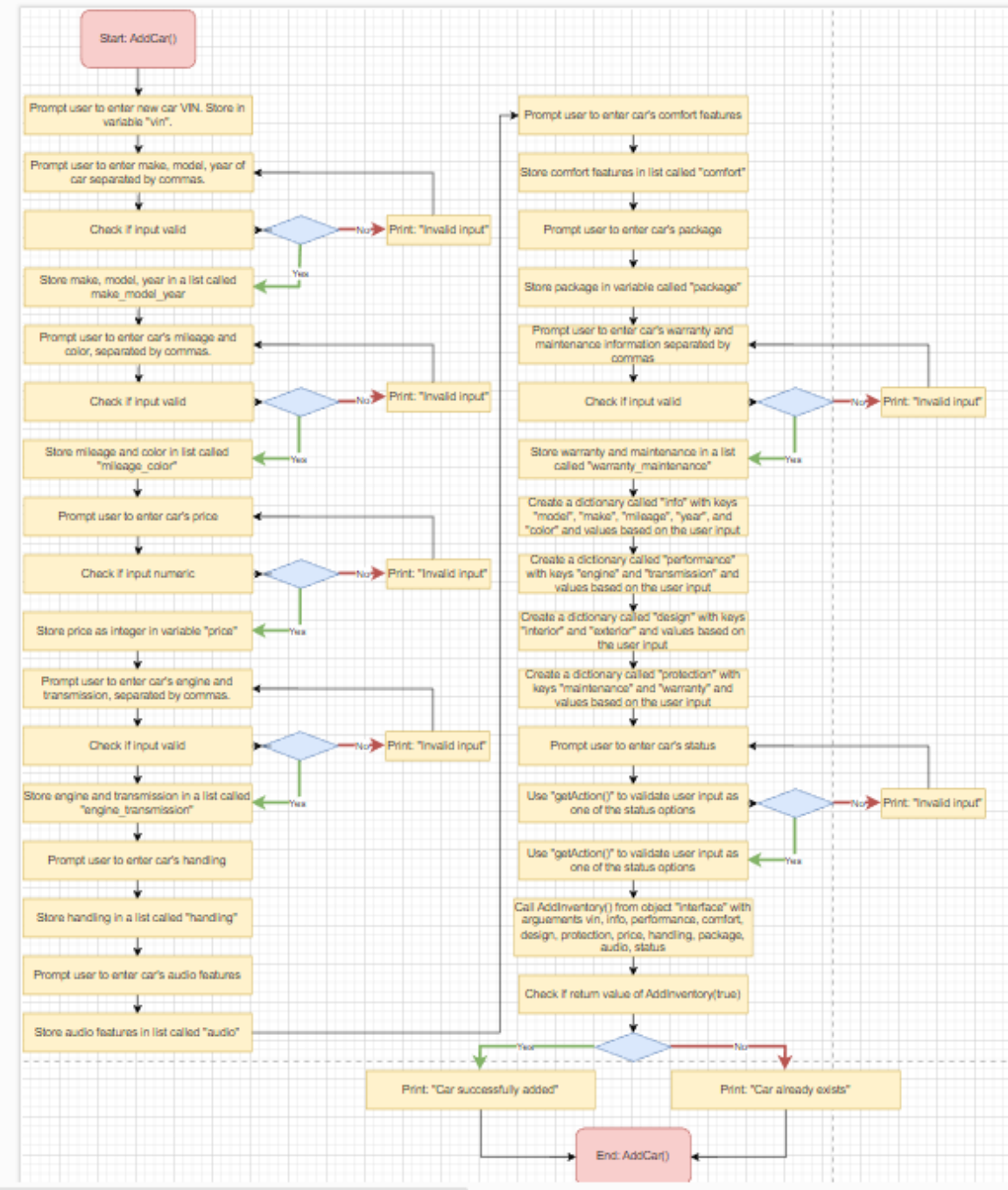
Significant Algorithms

AddCar()

Description

The AddCar() function is designed to add a new car to an inventory system. The algorithm begins by prompting the user for the car's Vehicle Identification Number (VIN), which is a unique identifier for the car. The function then uses a loop to prompt the user to input the make, model, and year of the car, which are separated by commas. The input is then validated to ensure that it is in the correct format and that the year is a number. The next loop prompts the user to input the car's mileage and color, which are also separated by commas. The input is then validated to ensure that the mileage is a number and the color is a string. The user is then prompted to input the car's price, which is also validated to ensure that it is a numeric value. The function then prompts the user to input the car's engine and transmission, which are separated by commas. The input is validated to ensure that it is in the correct format. The user is then prompted to input the car's interior and external design, which are also separated by commas. The input is validated to ensure that it is in the correct format. The function then prompts the user to input the car's paint color, handling features, audio system, comfort features, and package. The input for paint color is validated to ensure that it is a string. The user is then prompted to input the car's warranty and maintenance information, which are separated by commas. The input is validated to ensure that it is in the correct format. The function then creates dictionaries for the car's information, performance, design, and protection. These dictionaries are used to store the various features of the car. Finally, the function prompts the user to input the car's status, which must be one of the following: "ordered", "available", "backorder", or "delivered". The input is validated to ensure that it is one of these values. The function then adds the car to the inventory system using an interface method, and outputs a success message if the car was added successfully or an error message if the car already exists in the inventory.

Flowchart


Start: AddCar()
Prompt user to enter new car VIN. Store in variable "vin".
Prompt user to enter make, model, year of car separated by commas.
Check if input valid
No
Print: "Invalid input"
Yes
Store make, model, year in a list called make_model_year
Prompt user to enter car's mileage and color, separated by commas.
Check if input valid
No
Print: "Invalid input"
Yes
Store mileage and color in list called "mileage_color"
Prompt user to enter car's price
Check if input numeric
No
Print: "Invalid input"
Yes
Store price as integer in variable "price"
Prompt user to enter car's engine and transmission, separated by commas.
Check if input valid
No
Print: "Invalid input"
Yes
Store engine and transmission in a list called "engine_transmission"
Prompt user to enter car's handling
Store handling in a list called "handling"
Prompt user to enter car's audio features
Store audio features in list called "audio"
Prompt user to enter car's comfort features
Store comfort features in list called "comfort"
Prompt user to enter car's package
Store package in variable called "package"
Prompt user to enter car's warranty and maintenance information separated by commas
Check if input valid
No
Print: "Invalid input"
Yes
Store warranty and maintenance in a list called "warranty_maintenance"
Create a dictionary called "info" with keys "model", "make", "mileage", "year", and "color" and values based on the user input
Create a dictionary called "performance" with keys "engine" and "transmission" and values based on the user input
Create a dictionary called "design" with keys "interior" and "exterior" and values based on the user input
Create a dictionary called "protection" with keys "maintenance" and "warranty" and values based on the user input
Prompt user to enter car's status
Use "getAction()" to validate user input as one of the status options
No
Print: "Invalid input"
Yes
Use "getAction()" to validate user input as one of the status options
Call AddInventory() from object "interface" with arguements vin, info, performance, comfort, design, protection, price, handling, package, audio, status
Check if return value of AddInventory(true)
Yes
No
Print: "Car successfully added"
Print: "Car already exists"
End: AddCar()

Pseudocode

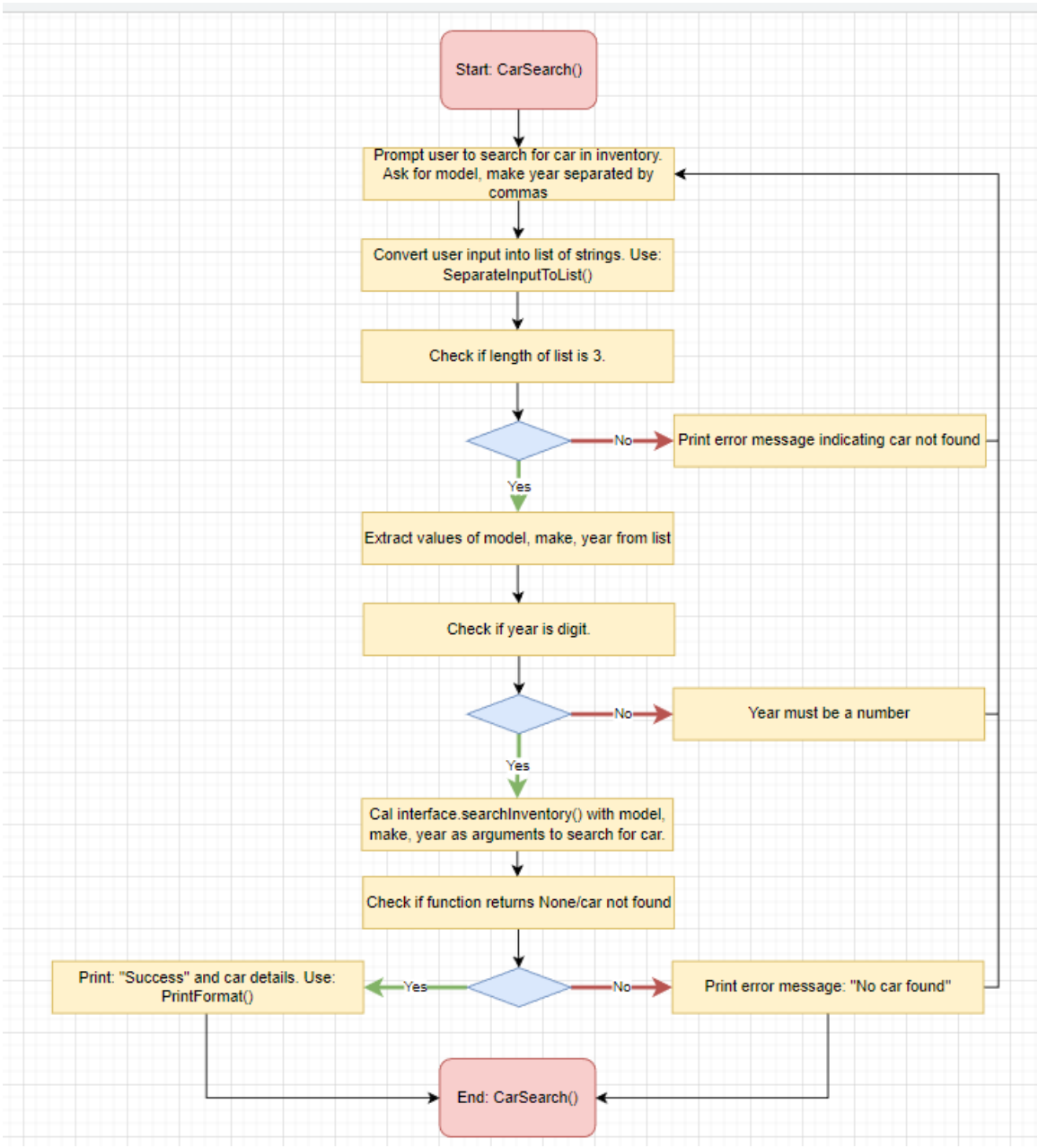
```
function AddCar():
    vin = getUserInput("Enter new car's VIN: ")
    make_model_year = None
    while True:
        make_model_year = getUserInput("Enter make, model, year (Separated by commas): ").split(",")
        if length(make_model_year) != 3 or not isNumeric(make_model_year[2]):
            continue
        break
    mileage_color = None
    while True:
        mileage_color = getUserInput("Enter mileage, color (Separated by commas): ").split(",")
        if length(mileage_color) != 2 or not isNumeric(mileage_color[0]) or not isAlpha(mileage_color[1]):
            continue
        break
    price = getNumericInput("Enter price: ")
    engine_transmission = None
    while True:
        engine_transmission = getUserInput("Enter engine, transmission (Separated by commas): ").split(",")
        if length(engine_transmission) != 2:
            continue
        break
    interior_external_design = None
    while True:
        interior_external_design = getUserInput("Enter interior, external design (Separated by commas): ").split(",")
        if length(interior_external_design) != 2:
            continue
        break
    paint = getUserInput("Enter paint: ")
    handling = [getUserInput("Enter handling: ")]
    audio = [getUserInput("Enter audio: ")]
    comfort = [getUserInput("Enter comfort features: ")]
    package = getUserInput("Enter package: ")
    warranty_maintenance = None
    while True:
        warranty_maintenance = getUserInput("Enter warranty, maintenance (Separated by commas): ").split(",")
        if length(warranty_maintenance) != 2:
            continue
        break
    info = {"model": make_model_year[1],
            "make": make_model_year[0],
            "mileage": int(mileage_color[0]),
            "year": int(make_model_year[2]),
            "color": mileage_color[1]}
    performance = {"engine": engine_transmission[0],
                   "transmission": engine_transmission[1]}
    design = {"interior": [interior_external_design[0]],
              "exterior": [{"paint": paint,"extra": [interior_external_design[1]]}]}
    protection = {"maintenance": warranty_maintenance[1], "warranty": [warranty_maintenance[0]]}
    status = getValidInput(["ordered", "available", "backorder", "delivered"], "Enter status: ")
    if not status:
        return
    add = addInventory(vin, info=info, performance=performance,comfort=comfort, design=design,
                       protection=protection, price=price, handling=handling, package=package, entertainment=audio, status=status)
    if add:
        PrintFormat("Success", "Car successfully added")
    else:
        PrintFormat("Invalid", "Car already exists")
```

CarSearch()

Description

The CarSearch() function is a simple algorithm that allows users to search for a car in the inventory based on the car's make, model, and year. The user is prompted to enter these three values separated by commas. If the user enters invalid or incomplete data, the function will return an error message. Once the user input is validated, the function will call the searchInventory() function from the interface module, passing in the make, model, and year parameters. If a match is found in the inventory, the function will return the car's details, formatted as a string. If no match is found, the function will return an error message. It performs basic input validation and uses a helper function from the interface module to search for the car in the inventory. If a match is found, it returns the car details to the user.

Flowchart


Start: CarSearch()
Prompt user to search for car in inventory. Ask for model, make year separated by commas
Convert user input into list of strings. Use: SeparateInputToList()
Check if length of list is 3.
No
Print error message indicating car not found
Yes
Extract values of model, make, year from list
Check if year is digit.
No
Year must be a number
Yes
Cal interface.searchInventory() with model, make, year as arguments to search for car.
Check if function returns None/car not found
Print: "Success" and car details. Use: PrintFormat()
Yes
No
Print error message: "No car found"
End: CarSearch()

Pseudocode

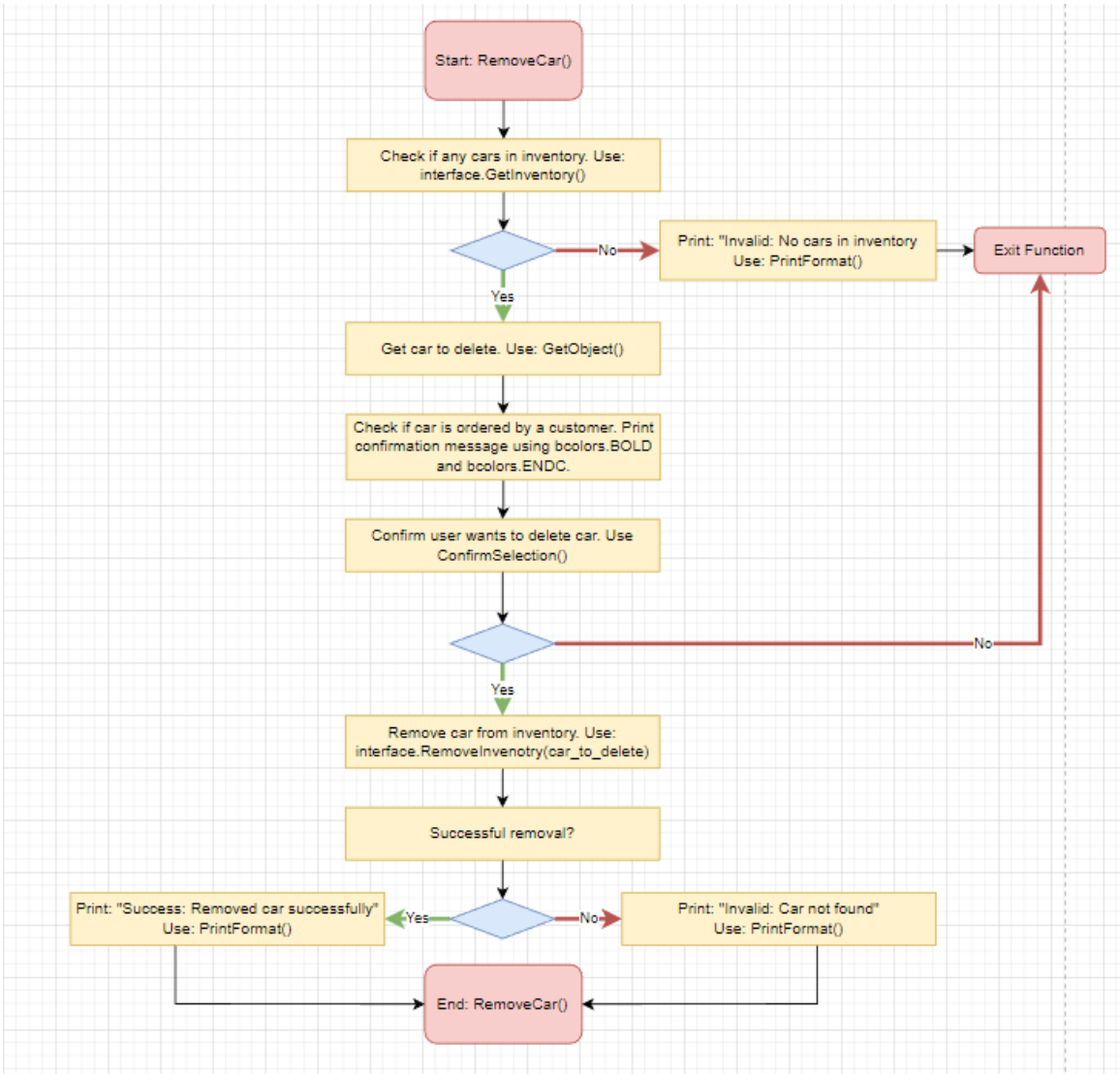
```
def CarSearch():
    # Get search criteria from user
    model, make, year = input("Enter model, make and year separated by commas: ").split(',')
    year = year.strip()

    # Validate year is numeric
    if not year.isnumeric():
        print("Invalid: Year must be a number!")
        return

    # Search inventory for car
    car = interface.searchInventory(model.strip(), make.strip(), int(year))

    # Check if car was found
    if not car:
        print(f"No match found for {year} {make} {model}.")
        return

    # Display car details
    print(f"Car details:\n{car.getDetails()}")
    return car
```

RemoveCar()

Description

The RemoveCar() function is designed to remove a car from the inventory. The function first checks if there are cars in the inventory. If the inventory is empty, the function will display an error message and return. If there are cars in the inventory, the function will prompt the user to select the car to delete. After the user selects a car, the function will add a confirmation message to ensure that the user wants to delete the selected car. The confirmation message will include a bold text notification if the selected car has been ordered. If the user confirms that they want to delete the selected car, the function will remove the car from the inventory. If the car is successfully removed from the inventory, the function will display a success message. If the car is not found in the inventory, the function will display an error message.

Flowchart


Start: RemoveCar()
Check if any cars in inventory. Use: interface.GetInventory()
No
Print: "Invalid: No cars in inventory Use: PrintFormat()
Exit Function
Yes
Get car to delete. Use: GetObject()
Check if car is ordered by a customer. Print confirmation message using bcolors.BOLD and bcolors.ENDC.
Confirm user wants to delete car. Use ConfirmSelection()
No
Yes
Remove car from inventory. Use: interface.RemoveInvenotry(car_to_delete)
Successful removal?
Print: "Success: Removed car successfully" Use: PrintFormat()
Yes
No
Print: "Invalid: Car not found" Use: PrintFormat()
End: RemoveCar()

Pseudocode

```
def RemoveCar():
    if interface.GetInventory() is empty:
        PrintFormat("Invalid", "No cars in inventory")
        return

    car_to_delete = GetObject(interface.GetInventory())

    confirm_msg = f"Are you sure you want to delete {car_to_delete}"
    if interface.isCarOrdered(car_to_delete):
        confirm_msg += f" {bcolors.BOLD}it has been ordered{bcolors.ENDC}"

    if not ConfirmSelection(msg=confirm_msg):
        return

    rem = interface.RemoveInventory(car_to_delete)
    if rem:
        PrintFormat("Success", "Removed car successfully")
    else:
        PrintFormat("Invalid", "Car not found")
```

CODING

Programing Language

Python is our selected programming language for the system. Our main developer is most familiar with this language, making it the most logical choice to make sure the project runs smoothly. Some other reason we chose python are listed below:

- Rapid Development: Python is known for its ease of use and readability, which means our developers can write code more quickly and efficiently. This helps with rapid prototyping and iterative development.
- Large Standard Library: Python comes with a large standard library that includes modules for many common tasks, such as string manipulation, file I/O, and networking. This means we can focus on writing application-specific code rather than low-level details. In our system we mainly use lists and dictionaries.
- Testing and Debugging: Python has a built-in testing framework and a debugger, which makes it easy to write and debug code. These frameworks will be used during unit testing, integration testing, and system testing.

Standards and Styles

The development team should follow the program design standards describing the programming standards. This includes proper commenting and following the header block structure for each file.

The system will also focus on these standards:

- Object-Oriented Design: This emphasizes the use of objects to model real-world entities. It can help improve code organization and maintainability.
- Defensive Programming: This will emphasize error checking and handling to ensure that the code is robust and resistant to failure.
- Code Reviews: Developers will review each other's code to ensure that it meets the required standards and is of high quality. This should help identify issues early and improve the overall quality of the system.
- Clean Code: This is a set of guidelines for writing readable, maintainable, and efficient code.

    - Use Meaningful Names: Use descriptive and meaningful names for variables, functions, and classes. Avoid using short or cryptic names that can be confusing.
    - Keep Functions Small: Keep functions small and focused. A function should do one thing and do it well.
    - Eliminate Duplication: Avoid duplicating code. Use functions and classes to encapsulate common functionality and reuse code wherever possible.
    - Use Consistent Formatting: Use consistent formatting throughout the codebase. There should only be one indent between functions, two indents between classes and avoid whitespace. Each function should have a docstring comment.
- Use camelCase for variable and method names.
- Use PascalCase for class and interface names
- Use descriptive names for variables, methods, and classes.
- Use comments to document code, including method parameters and return values.
- Follow SOLID principles to ensure maintainability and extensibility of the codebase.
    - Single Responsibility Principle (SRP): A class should have only one reason to change. It should have only one responsibility or job to do. This means that a class should have only one reason to be modified.
    - Open/Closed Principle (OCP): A software entity should be open for extension but closed for modification. This means that a class should be designed in a way that allows new functionality to be added without changing the existing code.
    - Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base types. This means that a subclass should be able to be used in place of its parent class without causing any unexpected behavior.
    - Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. This means that a class should not be forced to implement methods or interfaces that it does not need.
    - Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. This means that classes should depend on abstractions rather than concrete implementations.

Code Frames

The system is split up into separate python scripts depending on their functionality. Each frame follows the coding guidelines above, especially object- oriented design, to create functions that are easily used and maintained.

The scripts and their purposes are:

- readJson.py
    - The code consists of four functions that load data from JSON files in order to create objects for a vehicle inventory, users, orders, and customers.
- writeJson.py
    - The general purpose of this code is to provide a function called writeJson that writes data to a JSON file based on the type of object passed in the data parameter.
    - It determines the appropriate file path and serializes the data before writing it to the file.
    - This function is used to save various data types such as inventory, orders, customers, and users to their respective JSON files in the program.
- bcolors.py
    - The purpose of the code is to define a class bcolors that contains various color codes that can be used for printing colored text in the console.
- interface.py

- contains classes and functions to manage the car dealership's interface, allowing employees and admins to view, manage, and update car inventory, customers, orders, and user accounts.

- main.py
  - Manages user accounts and provides an interface for users to interact with their account. The “meat” of the system.
  - Works as a command line interface.
- orders.py
  - The code defines a class called "Order" representing an order made by a buyer for a car, containing information such as the buyer, car, salesperson, date of purchase, and order ID, with various methods to manipulate and retrieve information about the order.
- session.py
  - Handle session management and authentication of users.
- status.py
  - The purpose of this code is to provide a convenient way to represent and manipulate the status of cars in a car sales inventory system, and to allow for easy conversion between string representations and enum values.
- users.py
  - This code defines a User class that represents a user with basic information like username, password, and date joined.
  - It defines two classes Admin and Employee, both inherit from the User class and define their own functionalities.
  - It defines a Customer class with attributes first and last name, card, email, address, and orders.
- vehicles.py
  - The purpose of the class is to provide a blueprint for creating car objects and defining their properties and behaviors.

readJson.py

```
from PigeonBox import vehicles, users, orders
import json

INVENTORY_PATH = "data/inventory.json"
USERS_PATH = "data/users.json"
ORDERS_PATH = "data/orders.json"
CUSTOMER_PATH = "data/customers.json"

def LoadInventory():
    """
    Loads the car inventory data from a JSON file and creates a list of Car
objects.
    Returns:
        List of vehicles.Car objects
    """
    pass

def LoadUsers():
    """
```

```
    Loads user data from a JSON file and creates Employee and Admin objects
accordingly.
    Returns:
        Tuple of two lists: employees and admins
    """
    pass

def LoadOrders():
    """
    Loads a list of orders from a JSON file.
    Returns:
        List of orders
    """
    pass

def LoadCustomers():
    """
    Loads and returns a list of Customer objects from a JSON file.
    Returns:
        List of users.Customer objects
    """
    pass
```

writeJson.py

```
from PigeonBox import vehicles, users, orders
from parsers.readJson import INVENTORY_PATH, ORDERS_PATH, CUSTOMER_PATH,
USERS_PATH
import json
INDENTATION = 2

def writeJson(data):
    """
    This function writes data to a json filePath, it will determine what
kind of data is being loaded and write to
    the appopriate json filePath with the proper formatting
    """
    pass
```

bcolors.py

```
class bcolors:
    """Define a class containing various colors to format output text"""

    HEADER = '\033[95m'
```

```
    OKBLUE = '\033[94m'
    OKCYAN = '\033[96m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'


def PrintFormat(color_status: str, print_info: str) -> None:
    """
    Print formatted text with color and style options

    :param color_status: A string representing the color option
    :param print_info: A string to be printed
    :return: None
    """
    # Define a dictionary with color options
    color_options = {"Invalid": bcolors.FAIL,
                     "Success": bcolors.OKGREEN,
                     "Action": bcolors.OKBLUE,
                     "Important": bcolors.BOLD,
                     "Warning": bcolors.WARNING,
                     "Purple": bcolors.HEADER,
                     "Cyan": bcolors.HEADER}

    # Set the color based on the color status parameter
    color_status = color_options[color_status]

    # Print the text with the specified color
    print(f"{color_status}{print_info}{bcolors.ENDC}")
```

interface.py

```
from parsers import *
from PigeonBox import session, status, orders, users, vehicles


class InterfaceObjects():
    """A class for managing the interface objects and mapping functions."""

    def __init__(self) -> None:
        """ Initializes an instance of a class with inventory, customers,
employees, admins, and users."""
```

```
        self.inventory = readJson.LoadInventory()
        self.customers = readJson.LoadCustomers()
        self.employees = session.Session().ReturnEmployees()
        self.admins = session.Session().ReturnAdmins()
        self.__users__ = self.employees + self.admins # helper


    '''mapper functions'''
    def usernameToUser(self, username):
        """Converts a given username to a corresponding user object if it
exists in the system."""
        pass

    def vinToCar(self, vin):
        """This function returns a Car object based on its Vehicle
Identification Number (VIN)."""
        pass

    def emailToCustomer(self, emailAddress):
        """Get customer object by email address."""
        pass

    def getCustomerList(self):
        """Returns the list of all customers."""
        pass

    def getEmployeeList(self):
        """Returns a list of all employees."""
        pass

    def isEmployee(self, user):
        """Checks whether the given user is an employee or not."""
        pass

    def ViewUsers(self):
        """Returns the list of all users in the system."""
        pass

    def vinExists(self, vin):
        """Check if a VIN (Vehicle Identification Number) exists in the
inventory."""
        pass

    def inInventory(self, car):
        """Check if a car is in the inventory."""
        pass
```

```
    def ViewByStatus(self, statusToViewBy):
        """Returns a list of vehicles filtered by their status."""
        pass

    def GetInventory(self):
        """Get the inventory of the car dealership."""
        pass

    def searchInventory(self, model, make, year):
        """ Search inventory by car model, make, and year and return car
object if found."""
        pass

    def ViewAvailableInventory(self):
        """Returns a list of available vehicles in the inventory."""
        pass

    def UserExists(self, checkUser):
        """ Checks if a user exists in the list of users..Useful for when we
want to add or remove an employee"""
        pass

    def UsernameExists(self, username):
        """Checks if a username already exists in the list of users."""
        pass


class Interface(InterfaceObjects):
    """Interface is the class that contains all the methods for managing the
system's car, customer, users, and orders."""

    def __init__(self) -> None:
        """Initializes an object of the class and loads orders from a JSON
file."""
        super().__init__()
        # loads orders from json
        self.ordersDict = readJson.LoadOrders() # have to pass customers and
car functionality to correctly add to buyers list and such
        self.orders = []
        for order in self.ordersDict:
            self.orders.append(orders.Order(id=order['id'],
car=self.vinToCar(order['carVin']),
buyer=self.emailToCustomer(order['buyer']),
employee=self.usernameToUser(order['soldBy']),
dateBought=order['dateBought']))
```

```
        # 0: inv, 1: order, 2: users, 3: customer
        self.isObjListUpdated = [False] * 4
        # loading customer orders if there's any
        for order in self.orders:
            currentCustomer = order.getUser()
            for customer in self.customers:
                if customer == currentCustomer:
                    customer.orders.append(order)

    def changeCarStatus(self, car, status):
        """Change the status of a car and mark the inventory list as
updated."""
        pass

    def changeCarPrice(self, car, newPrice):
        """Update the price of a car and set a flag for the inventory list
as updated."""
        pass

    def changeCarMileage(self, car, newMileage):
        """ Update mileage of a car object and set isObjListUpdated flag to
True."""
        pass

    def changeCustomerEmail(self, customer, newEmail):
        """Update the email of a given customer object."""
        pass

    def changeCustomerCard(self, customer, newCard):
        """Change a customer's credit card information"""
        pass

    def changeCustomerAddress(self, customer, newAddress):
        """Updates the address of a customer and sets a flag to indicate
that the customer list has been updated."""
        pass

    def changeUserPassword(self, user, newPassword):
        """Changes the password of an admin or employee user."""
        pass

    def changeUserUsername(self, user, newUsername):
        """A function to change the username of an admin or an employee in a
car dealership system."""
        pass
```

```
    def viewOrders(self):
        """Returns the list of orders."""
        pass

    def getOrderslist(self):
        """Returns the list of orders."""
        pass

    def isCarOrdered(self, car) -> bool:
        """Check if a car has been ordered by a customer."""
        pass

    def doesOrderExist(self, checkOrder) -> bool:
        """Checks if a car has been ordered."""
        pass

    def MakeOrder(self, customer, vehicle, seller):
        """Creates an order object for a given customer, vehicle, and seller
and adds it to the order list."""
        pass

    def UndoOrder(self, order):
        """Remove an order from the orders list and update the car
status."""
        pass

    def emailExists(self, email):
        """Checks if an email exists in the customer list."""
        pass

    def isCustomer(self, newCustomer):
        """
        Check if a given customer exists in the list of customers.
        """
        pass

    def AddCustomer(self, first, last, card, email, address):
        """
        Adds a new customer to the list of customers.
        """
        pass

    def RemoveCustomer(self, customer):
        """
        Removes a given customer and their orders from the system, and sets
any ordered car that has not been delivered to backorder status.
```

```
        """
        pass

    def LogOut(self):
        """
        Saves changes to JSON files and logs the user out of the system.
        """
        pass

class AdminInterface(Interface):
    """ Admin can do everything an employee can do and MORE, hence why it
has its own class, which inherits all the methods from Interface
        but an admin can add and remove employees, and add and remove cars
from inventory """

    def AddAdmin(self, username, passwd, fname, lname, dateJoined=None):
        """Add a new admin user to the system."""
        pass

    def AddEmployee(self, username, passwd, fname, lname, dateJoined=None)
-> None:
        """Adds a new employee user to the system."""
        pass

    def ordersMadeByUser(self, user):
        """Returns a list of orders made by a given user."""
        pass

    def RemoveUser(self, userToRemove) -> bool:
        """Removes a user from the system and all orders made by that
user."""
        pass

    def AddInventory(self, vin, info, performance, design, handling,
comfort, entertainment, protection, package, price, status=None) -> bool:
        """  Add a new car to the inventory if it doesn't already exist."""
        pass

    def RemoveInventory(self, car) -> bool:
        """ checks if car exists, if it does, checks the status of the car,
if it is ordered then it also removes the order
        then proceeds by removing the car entirely from the system"""
        pass
```

main.py

```
from PigeonBox.interface import *
from PigeonBox.session import Auth
from PigeonBox.bcolors import *

''' Command line interface '''
def displayData(data):
    """Display an array of data along with their index."""
    pass

define StallUntilUserInput():
    """ Stalls the program and waits for the user to press Enter to continue
execution."""
    pass

def isEmpty(arr):
    """Checks if an array is empty and displays a warning message if it
is."""
    pass

def validatePassword():
    """Function to validate a new password entered by the user, with
confirmation."""
    pass

''' User account details'''
def ChangePasswordMenu():
    """A function to change the password of a user through an interface."""
    pass

def validateUsername():
    """Validates user input for a new username and checks if it is already
taken or the same as the old username."""
    pass

def ChangeUsernameMenu():
    """Changes the username for a user by taking input from the user."""
    pass

''' Checkers and validators '''
define ConfirmSelection(response = {"y", "yes", "n", "no"}, msg="") -> bool:
    """A function to confirm a user's selection through a prompt, returning
a boolean value."""
    pass

def ValidateUserInput(action="action", isNum=False, isEmail=False):
    """Validates user input based on certain conditions like being a number
```

```
or an email address."""
    pass

''' General helper functions '''
def getAction(validSet={"1", "2", "3"}, msg="Enter action:"):
    """
    A function to get user input for a valid action from a set of options.
    """
    pass

def PickIndex(arr):
    """
    Displays an array of elements and prompts the user to pick an index to
select an element.
    Will display elements in array and ask user to pick one, will return the
index of the element picked
    Useful for when choosing an element to view or remove. Like when
removing a car, this will point to its index in the list
    and the user will be able to remove it by index.
    """
    pass

def SeparateInputToList(inpt):
    """
    Takes in a string and returns an array of the string separated by commas
    """
    pass

def GetObject(objectList):
    """
    Selects and returns an object once user chooses it from a list (calls on
PickIndex)
    """
    pass

def updateCarStatus(car):
    """
    Update the status of a car in the inventory.
    """
    pass

''' helper menus'''
def AddEmployee():
    """
    Function to add a new employee with username, password, first name, and
last name
```

```
    and grant admin privileges if desired.
    """
    pass

def RemoveEmployeeMenu():
    """
    'Menu' for removing an employee, will ask user to pick from a list of
employees
    and then confirm the selection. If the user confirms, the employee will
be removed.
    """
    pass

def displayStatusOptions():
    """
    Displays the available status options for an inventory and prompts the
user to choose one.
    """
    pass

def CarSearch():
    """
    A function that searches for a car in the inventory based on the model,
make, and year entered by the user.
    """
    pass

def filterByMenu():
    """
    Displays a menu to filter cars by their status and displays the filtered
data.
    """
    pass

define modifyInventoryMenu():
    """
    Allows an administrator to add or remove a car from the inventory.
    """
    pass


''' Customer Management helper functions'''
def AddCustomer():
    """
    Adds a new customer with their personal and credit card details to the
system.
```

```
    """
    pass

def DeleteCustomerMenu(customerToDelete):
    """Deletes a customer from the system."""
    pass

def Login():
    """ This function takes care of the login page,
      it will ask for username and password and will return the user object
if the user is authenticated"""
    pass

def modifyCarMenu(car):
    """Modifies car status, price, mileage, and warranty plans."""
    pass

def SearchCarMenu(car=None):
    """Function to handle searching for cars in the inventory and providing
actions for the search result."""
    pass

define InventoryMenu():
    """
    Display an inventory menu with options to view car details, search,
filter, make customer orders, and modify inventory (admin-only).
    """
    pass

def AddCar():
    """
    Will get user input for a new car and add it to the inventory
    """
    pass

def RemoveCar():
    """Function to remove a car from the inventory."""
    pass

def addOrderMenu(carToOrder, customers):
    """ Allows the user to add a new order to the system by selecting a car
and customer."""
    pass

def OrderMenu():
    """Prints the order menu and provides options to add, remove, or view
```

```
orders."""
    pass

define ManageEmployeesMenu():
    """
    Displays employee management menu and provides options to view, add or
remove employees.
    """
    pass

def validateCreditCard():
    """
    Validates a credit card number.
    """
    pass

define modifyCustomerDetails(customer=None):
    """
    Modify details of a customer.
    """
    pass

define ManageCustomersMenu():
    """
    Displays a menu for managing customers and prompts for actions to
perform.
    """
    pass

def CarSalesMenu():
    """
    Displays a menu for car sales and prompts for actions to perform.
    """
    pass

def AccountSettingsMenu():
    """
    Displays the menu for changing account details.
    """
    pass

def menu():
    """
    Main menu that the users will first interact with. Calls on other menus.
    """
    pass
```

orders.py

```
from datetime import datetime
from PigeonBox.bcolors import bcolors

class Order():
    """ Class represents orders made, works like a database one-to-one
relationship with a buyer and a car"""

    def __init__(self, id, car=None, buyer=None, dateBought=None,
employee=None) -> None:
        """
        Initializes an object of the Sale class with specified attributes
including car, buyer, sales employee, and date of sale.
        """
        pass

    def getUser(self):
        """Returns the ID of the user who bought the car."""
        pass

    def getCar(self):
        """Returns the car associated with the order object."""
        pass

    def getSeller(self):
        """Returns the employee who sold the car in the order."""
        pass

    def getDate(self):
        """Returns the date when the car was bought."""
        pass

    def getId(self):
        """Returns the ID of a car order."""
        pass

    def getDateJoined(self):
        """Returns the date the user joined in the format "YYYY-MM-DD"."""
        pass

    def to_dict(self):
        """Convert the Order object to a dictionary format."""
        pass

    def serialize(order):
        """Serializes an Order object to a JSON-compatible format."""
```

```
        pass

    def RemoveOrder(self):
        """Removes an order and sets the car status to available."""
        pass

    def orderDetails(self):
        """Generates a string containing the details of the order."""
        pass

    def __str__(self):
        """Returns a string representation of the order object."""
        pass

    def __eq__(self, value) -> bool:
        """Check equality of two Order objects."""
        pass

    def __repr__(self) -> str:
        """Returns the string representation of the car object."""
        pass
```

session.py

```
from parsers import readJson, writeJson

class Session:
    """handles the session management and authentication of users."""

    def __init__(self) -> None:
        """ Initializes an object of the class with two private attributes
containing employee and admin data from a JSON file."""
        self.__employees__, self.__admins__ = readJson.LoadUsers()

    def ReturnEmployees(self):
        """Return the list of employees."""
        pass

    def ReturnAdmins(self):
        """Returns a list of administrator users."""
        pass


class Auth(Session):
    """Extends the Session class to add authentication functionality."""
```

```
    def __init__(self) -> None:
        """Initializes an object of a class and creates a combined list of
employees and admins as a list of users."""
        super().__init__()
        # Combines the lists of employees and admins into a single list of
users
        self.__users__ = self.__employees__ + self.__admins__

    def Authenticate(self, username, password):
        """Authenticate a user with their username and password."""
        pass
```

users.py

```
from datetime import datetime

class User():
    """A class that represents a user with basic information like username,
password, and date joined."""

    def __init__(self, username='', password='', first_name='',
last_name='', date_joined=None) -> None:
        """Initializes a User object with a username, password, first name,
last name, and date joined."""
        self.username = username
        self.password = password
        self.first_name = first_name
        self.last_name = last_name
        # validate date. if date is not given, set it to today's date.
        if not date_joined:
            date_joined = datetime.today()
        else:
            date_joined = datetime.strptime(date_joined, "%Y-%m-%d")
        self.date_joined = date_joined

    def getFirstName(self):
        """Returns the first name of the user."""
        pass

    def getLastName(self):
        """Returns the last name of a user."""
        pass

    def UpdatePassword(self, newpassword):
        """Updates the password of the user."""
        pass
```

```
    def UpdateUserName(self, newusername):
        """Updates the username attribute of a user object with a new
username."""
        pass

    def getUsername(self):
        """Returns the username of a user."""
        pass

    def getPassword(self):
        """Returns the password of a User instance."""
        pass

    def serialize(user):
        """A function to serialize a User object to a JSON-serializable
format."""
        pass

    def __eq__(self, __o):
        """Determines whether two users are equal by comparing their
usernames."""
        pass

    def __str__(self):
        """Returns a string representation of a User object."""
        pass

    def __repr__(self) -> str:
        """Return a string representation of a User object."""
        pass



class Admin(User):
    """class inherits from the User class and defines an Admin user type.An
Admin user can delete or add inventory, add or delete employees."""

    def __init__(self, username='', password='', first_name='',
last_name='', date_joined=None, categoryType="Admin") -> None:
        """Initializes an instance of the Admin class, inheriting from the
User class and setting the category type to "Admin"."""
        super().__init__(username, password, first_name, last_name,
date_joined)
        self.categoryType = categoryType
```

```
    def getCategory(self):
        """Returns the category type of an object."""
        pass

    def __str__(self):
        """Returns a string representation of an Admin instance."""
        pass

    def getDetails(self):
        """Get user details."""
        pass

    def to_dict(self):
        """Returns a dictionary with user information."""
        pass

class Employee(User):
    def __init__(self, username='', password='', first_name='',
last_name='', date_joined=None, categoryType="Employee") -> None:
        """Initializes an Employee object with a specified username,
password, first and last name, date joined, and category type."""
        super().__init__(username, password, first_name, last_name,
date_joined)
        self.categoryType = categoryType

    def __str__(self):
        """Returns a string representation of the Employee object including
category, first name, and last name."""
        pass

    def getDetails(self):
        """Returns the details of the employee's joining date in a formatted
string."""
        pass

    def getCategory(self):
        """Returns the category type of the user."""
        pass

    def to_dict(self):
        """Returns a dictionary representation of the user object."""
        pass


class Customer:
    def __init__(self, first, last, card, email, address) -> None:
```

```
        """Constructor for a Customer class with attributes such as first
and last name, card, email, address and an empty list for orders."""
        self.firstName = first
        self.lastName = last
        self.card = card
        self.email = email
        self.address = address
        self.orders = []

    def getFirstName(self):
        """Returns the first name of a customer."""
        pass

    def getLastName(self):
        """Get the last name of a customer."""
        pass

    def getEmail(self):
        """Returns the email of a customer."""
        pass

    def setCard(self, new_card):
        """Sets the value of the card attribute for the instance of the
class."""
        pass

    def setAddress(self, new_address):
        """Method to set a new address for a user profile."""
        pass

    def setEmail(self, new_email):
        """Setter function to update the email address of a user."""
        pass

    def getDetails(self):
        """Returns a string containing the user's email, address, and list
of orders."""
        pass

    def to_dict(self):
        """Returns the dictionary representation of the customer object."""
        pass

    def serialize(customer):
        """Function that serializes a Customer object into a JSON
serializable dictionary."""
```

```
        pass

    def __str__(self) -> str:
        """This method returns a string representation of the object
instance."""
        pass

    def __repr__(self) -> str:
        """Returns a string representation of the customer object."""
        pass

    def __eq__(self, value) -> bool:
        """Check if two Customer objects are equal by comparing their email
addresses."""
        pass
```

status.py

```
from enum import Enum

class Status(Enum):
    """ Enum class that defines the status of a car, and two functions that
convert
        between the string representation of a status and its corresponding
enum value."""

    AVAILABLE = 1   # When a car is added to the inventory and is available
for purchase
    ORDERED = 2     # When a customer orders a car and it is not yet
delivered
    BACKORDER = 3   # when car is out of stock -- could be a customer
ordered, then got removed, set car     to backorder status rather than
available
    DELIVERED = 4   # when a car is delivered to a customer #TODO: add this
functionality in the car sales menu

STATUS_DICT = { "available": Status.AVAILABLE,
                "ordered": Status.ORDERED,
                "backorder": Status.BACKORDER,
                "delivered": Status.DELIVERED}

def strToStatus(status: str):
    """
    Convert a string representation of car status to its corresponding enum
```

```
value.

    Args:
    status (str): A string representing the status of a car

    Returns:
    Status: The corresponding Status enum value
    """
    pass

def StatusToStr(status: Status):
    """
    Converts a Status enum value to its string representation.

    Args:
    status (Status): The Status enum value

    Returns:
    str: The string representation of the given Status enum value
    """
    pass
```

vehicles.py

```
import locale
import PigeonBox.status as st
from PigeonBox.bcolors import bcolors


class Car():
    """Class that represents a car with various attributes and methods."""

    def __init__(self, vin='', info={}, performance={}, design={},
handling=[], comfort=[], entertainment=[], protection={}, package='',
status=None, price=0):
        """ A class constructor that initializes an object with various
attributes such as vehicle information, performance, design, handling,
comfort, entertainment, protection, package, status, and price."""
        self.vin = vin
        self.info = info
        self.performance = performance
        self.design = design
        self.handling = handling
        self.comfort = comfort
        self.entertainment = entertainment
        self.protection = protection
```

```
        self.package = package
        # Set default status to AVAILABLE if not provided
        if status is None:
            status = st.Status.AVAILABLE
        # Convert status string to Status enum if a string is provided
        if isinstance(status, str):
            status = st.strToStatus(status)
        self.status = status
        self.price = price

    def UpdateMileage(self, newMileage):
        """Update the mileage of a vehicle object."""
        pass

    def UpdateWarranty(self, newWarranty):
        """Update the warranty information of a car object."""
        pass

    def getVin(self):
        """Returns the vehicle identification number (VIN) of a vehicle
object."""
        pass

    def getStatus(self):
        """Returns the current status of a vehicle object."""
        pass

    def getStatusStr(self):
        """A method to get the string representation of the car's status"""
        pass

    def getCarInfo(self):
        """A method to get the string representation of the car's status."""
        pass

    def SetStatus(self, updated_status):
        """Sets the status of a car object either by passing in a Status
enum or a string representation of a status."""
        pass

    def to_dict(self):
        """"Convert object data to a dictionary format."""
        pass

    def serialize(car):
        """JSON serialization of Car object"""
```

```
        pass

    def isAvailable(self):
        """Checks if the car is available."""
        pass

    def UpdatePrice(self, newprice):
        """"Update the price of a Car object."""
        pass

    def getDetails(self):
        """Returns the details of a car object in a formatted string."""
        pass

    def __eq__(self, __o: object) -> bool:
        """Compares two instances of the Car class for equality based on
their VINs."""
        pass

    def __str__(self) -> str:
        """Returns a string representation of a car object including its
year, make, model, price, and status."""
        pass

    def __repr__(self) -> str:
        """Returns a string representation of the car object including its
model and make."""
        pass
```

UNIT TESTING

After coding was completed, the next step was to test to make sure the code worked as intended. While testing was done during the coding process, it is important to make sure that everything works at the end of coding as there are often small errors that might slip through while coding but will cause major problems for the project as a whole. The first step in testing the code was to take individual units of the code and to test them all separately. This is unit testing. The unit testing process can be generally broken up into seven steps.

Setting up test requirements and procedures

The first step of the unit testing process is to set up test requirements and procedures. This means that there should be a clear set of processes that should be able to be run in the program without failure. For our project, this entailed each of the functions described in the system design section of this document.

Performing code inspections

The next step of unit testing involves performing code inspections, meaning having other people look through the source code and understand what is going on and seeing if there are any obvious errors that the programmer may have missed, or if there are any sections of code that might be redundant or

slow the program down. For our project this involved having multiple people look at each section of code so that it was never only one person who knew how the code worked.

Finding and removing bugs

This step is probably the most obvious of them all, being that you must find and remove bugs. By looking through the code and running different test scenarios, finding bugs is almost guaranteed in any project. In our case, when someone would find a bug in the program, they would add it to our GitHub project so that the others could know about it. Then someone would see the list of bugs that we have left and work on removing them from the code.

Performing step/walk-through

The step/walk-through testing is another important step, as this is where you run the code as if you are someone who is just booting it up for the first time to use it. This is where you start the code from the base state, and attempt to achieve different things that the code is meant to accomplish. For us this meant starting up the program and attempting to do things such as adding and removing cars from the inventory, viewing customer orders, or changing the user's username all from the start of the program.

Test Environment and Setup

To carry out the unit tests, the Pytest library and framework were used to create a stream-lined testing environment. The Pytest library allows simple test case verification through the use of assertions. Furthermore, the Pytest library offered access to mocker, a component that allows the emulation of class instances and user inputs to ease the verification of intended behavior within individual functions without having to ensure proper integration, which was verified through integration tests discussed below.

Test Scenarios

For unit testing, each function written in a program represents a testing scenario that requires a test case to be written. Each function of the Vehicle Tracking System was met with a corresponding test case. Below, a few example implementations of unit tests for this program can be seen. In these cases, it is important to note that these tests are meant to cover only the explicit functionality of the given function without relying on any integrated or system performance.

```
def test_displayData(capsys):
    # create a list of test data
    data = ["apple", "banana", "cherry"]

    # capture the printed output of displayData
    displayData(data)
    captured = capsys.readouterr()
    # assert that the printed output is correct
    assert captured.out == "0: apple\n1: banana\n2: cherry\n"
```

Tests the functionality of displayData()

```python
def test_isEmpty():
    # test with an empty list
    assert isEmpty([]) == True

    # test with a non-empty list
    assert isEmpty([1, 2, 3]) == False
```

Tests if an input list is empty or not

```python
def test_ValidateUserInput(mocker):
    # mock the input function to return "1"
    mocker.patch.object(builtins, 'input', return_value="1")

    # test with isNum=True
    assert ValidateUserInput(isNum=True) == 1

    # mock the input function to return "user@example"
    mocker.patch.object(builtins, 'input', return_value="user@example.com")

    # test with isEmail=True
    assert ValidateUserInput(isEmail=True) == "user@example.com"


    # mock the input function to return "badinput", "", "2"
    mocker.patch.object(builtins, 'input', side_effect=["Invalid", "", "2"])

    # test with isNum=True
    assert ValidateUserInput(isNum=False) == 'Invalid'
```

Tests that a users input is acceptable under the conditions of ValidateUserInput()

```python
def test_PickIndex(mocker):
    # test with an empty list
    mocker.patch.object(builtins, 'input', return_value='q')
    assert PickIndex([]) == None

    # test with a non-empty list
    mocker.patch.object(builtins, 'input', side_effect=['a', '2', '1'])
    assert PickIndex(['apple', 'banana', 'cherry']) == 1

    # test with an index out of bounds
    mocker.patch.object(builtins, 'input', side_effect=['5', '-1', 'q'])
    assert PickIndex(['apple', 'banana', 'cherry']) == None

    # test with a non-numeric input
    mocker.patch.object(builtins, 'input', side_effect=['one', '0', 'q'])
    assert PickIndex(['apple', 'banana', 'cherry']) == None

    # test with cancelling
    mocker.patch.object(builtins, 'input', return_value='q')
    assert PickIndex(['apple', 'banana', 'cherry']) == None
```

Tests that when an index is picked through PickIndex(), the correct index is returned based on the user input.

Conducting various tests

Various different types of tests are conducted during unit testing, including:

- Statement tests
- Branch tests
- Path tests
- all-uses tests

These different tests allow us to make sure our code is functional in many different scenarios, some of which we may not have thought of without performing these types of tests.

Proving code correctness

The final step in the unit testing process is proving code correctness. This involves formally reasoning that there is no way the code could fail. For our project this meant going through each of our different files and looking at each function individually to make sure they all would work no matter what situation they were put in.

After performing all of the steps listed above, unit testing was completed. This is an important part of the programming process as it allows the developer to make sure the code is fully functional before putting it all together.

INTEGRATION TESTING

After testing each of the individual units, they must be integrated into a complete system so that it can be seen how well they work together to make a final system that achieves what was set out to be accomplished while designing the project. During integration testing, some significant issues must be overcome in order to complete the system.

Integration Type

Sandwich integration, also known as hybrid integration, is a combination of the two previously mentioned integration techniques. It involves integrating both top-level modules and lower-level modules into the system to make sure that they work together as intended. This method is what was used in our case as we felt that it was the most likely to accurately represent the final project and we did not have too many different modules that would make it difficult to perform sandwich integration.

Test Environment and Setup

The tests were conducted using the Python unittest library, which provides a framework for creating and running test cases. The test environment was set up to include all necessary dependencies and modules, such as the PigeonBox module and related classes. The Interface class, along with related classes such as Car, User, and Order were thoroughly tested to ensure proper integration and functionality.

Test Scenarios

The following scenarios were tested to ensure the correct integration of the Interface class with other components:

- Adding a car to the inventory:
  Test the AddInventory() method to ensure that a new car is added to the inventory correctly and updates the inventory list.

- Modifying a car in the inventory:
  Test the UpdateMileage(), UpdateWarranty(), and UpdatePrice() methods to ensure that changes made to a car's attributes are accurately reflected in the inventory.

- Deleting a car from the inventory:
  Test the RemoveInventory() method to ensure that a car is removed from the inventory and all associated orders are updated accordingly.

- Adding a customer:

Test the AddCustomer() method to ensure that a new customer is added correctly and updates the customer list.

- Managing employees and administrators:
  Test the AddEmployee(), AddAdmin(), and RemoveUser() methods to ensure that employees and administrators are added and removed correctly and the user list is updated accordingly.

- Handling orders:
  Test the MakeOrder() and UndoOrder() methods to ensure that orders are created and removed correctly, and that the order list is updated accordingly.

Test Results
All integration tests were executed successfully, and the program demonstrated correct interaction with other components. The tests confirmed that the Interface class effectively manages inventory, customers, employees, and orders as expected.

Confidence Indicators
Confidence indicators tell the programmer when they are set to stop testing the code. Without a confidence indicator, the programmer would either be giving a wild guess as to when the code is tested thoroughly enough, or they would keep testing forever, neither of which is a good option. By establishing a confidence indicator using statistics, the programmer can have an exact percentage chance that the code will work as intended.

Once the confidence level produced by the testing matches or exceeds the confidence indicator, integration testing is complete. There is still a chance that some bugs might have slipped through even after all of this testing, but by using these different tools and methods, the program is likely to perform as intended.

Conclusion
The integration tests for the Interface class have demonstrated that the program is functioning correctly and interacting properly with other components within the car dealership management system. As a result, the program can be considered reliable and robust in managing the various aspects of the system. Future development efforts should continue to maintain and enhance the integration of the Interface class with other components, ensuring the overall system's stability and performance.

SYSTEM TESTING
System testing is the final stage of development before the system is delivered to the customers. As such, it is one of the most critical states of software engineering. Before moving on to delivering the system, the following tasks must be performed:

Testbed Environment
Testbed environments are the situations that are used when testing the system to ensure it functions properly. This means having all of the hardware, software, operating system, and network configuration set to be tested. It is important to ensure the testbed environment is a similar environment to where the product will actually be used when it is released. This allows for accurate testing of the system in a realistic situation.

Testing Individual Functions

Before performing specific types of testing, it is important to test the individual functionalities of the program. So going through the whole system and ensuring that each of its functions are working as intended before moving on to more time consuming tests.

General Tests

There are specific types of tests that should be used for a majority of projects as they provide a good indicator of how well the project will work in a variety of different circumstances. These different tests include the following:

Stress Testing

Stress testing involves performing excessively more processes than would normally be expected of the program to ensure that the program will not break down if too many functions are called at once. It is useful in figuring out how the program responds to crashes or failures and how well it can fix itself if something like that happens. In order to implement stress testing, we decided to generate a large number of specific entries with multiple filters and sorting options and check if the system is able to handle them without crashing or slowing down. As the result, the program can handle a variety of inputs and configurations but slightly slows down if given too many inputs at the same time.

Volume testing

Volume testing involves placing a large amount of data into the system to ensure it runs properly even with an abundance of data. We decided to generate a large dataset of users, customers, inventory, and orders, and test the system's performance when handling a large amount of data. Furthermore, for the volume test, we created multiple test cases with large data sizes such as a large number of users, customers, inventory, and orders to check the system's scalability. As the result, the code did not encounter performance issues, it resulted well in terms of response time.

Configuration testing

Configuration testing involves testing the system on various computer systems. For instance using a computer with a different operating system than what it was developed on, or using it on hardware that is not as powerful as what it was developed on. This is an important testing step as it is impossible to know what the customer will be using the program on so it is important to make sure it runs well on at least the most commonly used hardware and software. Since we decided to test the system on different operating systems to ensure compatibility, stability, and functionality, the result of the configuration testing showed that the system is compatible and performs well under different configurations. During the process, we encountered issues that were resolved to ensure that the system is more versatile and reliable across different environments.

Time testing

Time testing is used to test the speed of the system on typical datasets. This involves taking time measurements on how fast the system can process different tasks and data when given a situation that is likely to come up when the system is being used. Therefore, we conducted time testing by measuring the time it takes for the system to execute specific functions or processes. The total time is calculated, and an assertion is made that it is less than 100 seconds. Tests were completed in 0.011 seconds, which shows that the functions can handle user inputs and complete their tasks quickly. The tests have passed the timing assertion that the total time taken to execute the functions should be less than 100 seconds. Therefore, we can conclude that the performance of the functions is satisfactory for the tested scenarios.

Quality testing

Quality testing ensures that the system is reliable and maintainable. Making sure the system does not break down in any common scenarios, and if it does the problem can be easily fixed is done in quality testing. Therefore, we created test cases to ensure that the system is reliable and produces accurate results. We tested the system's maintainability by introducing intentional bugs and checking if they are detected and fixed by the system. As well as testing the system's security by attempting to exploit vulnerabilities and checking if the system is able to detect and prevent them.

Perform acceptance testing

Acceptance testing is the process that allows the programmer to ensure that the program meets the end users' approval. It involves testing with actual potential customers and receiving their feedback on how the system might be improved or if there are any issues with the system. Generally there are three different acceptance testing methods, being:

Pilot testing

Pilot testing is an initial smaller test of the system that does not include all of the features that will be available in the final project. This is useful in that it can provide feedback on any structural flaws that might come up with the project before implementing the entire design. To create and run pilot testing, we installed a system that mimics a real use-case scenario, specifically a car dealership. The system as a customer, admin, or user was used to ensure that all functionality works as expected and that any bugs or issues are caught before the system is deployed in a live environment.

Alpha testing

Alpha testing is the testing that is performed in-house. It is performed by people specifically meant to test the system and any issues that are found during alpha testing are more easily fixed than those found in beta testing. Overall, it included various types of testing, such as unit testing, integration testing, and system testing.

Beta testing

Beta testing is performed by real users of the software in a real environment. This allows users to give feedback on the program before it is released fully. Beta testing allows for a much larger pool of testers as the program can be distributed to people to see how it will function in their situations. This provides useful feedback from the people who will actually be using the program. Since the system is being tested from the user, customer, and admin perspectives. It has all the necessary outcomes and processes for ordering cars. Testing the system with the users helped to identify issues that were not caught during pilot testing or alpha testing.

Customer site installation

Performing installations in some of the actual places where the program will be used allows for the developer to understand exactly how the system will be used by the customers and how easy it is to set up and install.

Documenting Tests

For each task that was presented, the tests should always be documented. Without documentation, there is a chance that a test that was already performed might be done again. There is also a chance that any bugs that were found could be forgotten without documentation. Properly documenting tests is an important part of the development process as it allows anyone working on the project to see what has been tested and what issues might have occurred during testing.

After performing system testing, the program is ready for delivery, it has been properly tested in many different environments and feedback has been received to ensure a quality product has been developed.

SYSTEM DELIVERY

System installation

PigeonBox can be run as an executable or through an IDE for programming and operator purposes. For the car dealership employees installing the system for the first time, they should install the system while following the guide labeled “Simple Installation” and programmers/operators who wish to work on the system’s logic should follow the “Advanced Installation” guide.

Simple Installation Guide

Simple installation will only work for the Windows operating system. If you are running Mac, please refer to the advanced installation guide.

1. Download from GitHub
2. Extract files from zip folder
3. Open file
4. Run main.exe to launch PigeonBox
5. PigeonBox should launch as a terminal interface.

Advanced Installation Guide

Use this installation guide for programmers

1. Open Visual Studio Code.
2. Make sure to install python onto your system. You can either search up a tutorial or follow the one below. Skip to step 8 if you already have VSCode installed with python.
3. Go to the extensions tab and type in python.
4. Install the first extension that pops up.
5. Click “File” at the top of VS Code..
6. Click “open folder” from the dropdown.
7. Select the folder containing PigeonBox and enter a folder named src.
8. Open a new terminal.
9. Type in the terminal: pip install colorama
10. To run the program type in the terminal: python3 -m PigeonBox.main
11. If you already had VS Code installed with an older version of python and get an error from running the above command type: python -m PigeonBox.main
12. PigeonBox should run in the terminal from VS Code.

System Demonstration

Initial installation and demonstration of the product will be performed by one of our team members for the local car dealership. The goal of this demonstration aims at showcasing the functionality of PigeonBox and how it can be used efficiently. A video of the product’s demonstration can be found on youtube from the link at the bottom of the document.

Below is an overview of how such a demonstration will occur:

1. Introduction:
   The presenter would introduce themselves and the purpose of the demonstration. They might provide an overview of the CLI car management system and its main features.

2. Navigating the System:
   The presenter would demonstrate how to navigate through the CLI car management system using the command-line interface. This would include showing how to access different menus, view and edit car information, and perform searches.

3. User types:
   The presenter will explain the differences between the two user types General Employee and Admin and what different functions they have access to.
4. Adding a New Car:
   The presenter would demonstrate how to add a new car to the system. They would walk through the process of entering information about the car, such as its make, model, year, and price.

5. Editing Existing Car Information:
   The presenter would show how to edit the information of an existing car, such as updating its mileage or price.

6. Searching for Cars:
   The presenter would demonstrate how to search for cars in the system based on different criteria, such as make, model, and year.

7. Conclusion:
   The presenter would conclude the demonstration by summarizing the main features and benefits of PigeonBox. They might also answer any questions from the dealership's employees..

User Guide

Introduction

PigeonBox is a software application that allows you to manage your vehicles and process customer orders using a command line interface. This guide provides instructions on how to operate PigeonBox effectively.

System Overview

While using PigeonBox, users will be able to add and delete cars from the inventory, process customer orders, view car sale details, search through the inventory, and manage employee and customer information.

Users can be split into two categories: General Employee and Admin. Admins have all the functions of General Employees plus the authority to add and remove cars from the inventory and manage employees like viewing their details and adding them to the system. You may think of Admins as the managers of the car dealership that have functions that require a certain level of authority to perform.

Depending on the user type the system is broken up into different menus. Once logging in, General Employees will be able to view Car Inventory, Customer Orders, Car Sales, Manage Customer, and Account Settings. Admins will have these menu options plus Manage Employees.

Below is each menu and their function:

- Car Inventory
  Users will be able to search the dealerships' inventory for vehicles in this menu. Users have the option to search cars by make, model, and year, filter by their status, make a customer order, or view a car's details. From this menu, you will also have access to edit a vehicle's details like price, mileage, and packages if needed. Only users with admin privileges will be able to add and remove cars from the dealership's inventory in this menu.

- Customer Orders
  Users can manage customer orders from this menu. This includes viewing order details, making an order, or changing the order status from ordered to delivered, marking the end of the order process. Users should only do this once the car reaches the customer's home address. After changing the order status, the order will be marked as a sale and will no longer show up in this menu.

- Car Sales
  This menu is for all sales or fully processed customer orders made in the dealership. This means cars that have come into the possession of the customer can be viewed here along with all relevant details.

- Manage Customers
  Here users will be able to manage customers. This includes adding new customers, removing customers, updating their information, and viewing their information.

- Manage Employees
  Only admins have access to this menu. Here they'll be able to manage employees on the system, as well as add new employees to the system. Admins will be able to grant admin privileges from this menu.

- Account Settings
  All users have access to this menu. Here they'll be able to update their username and password. New employees of the system are given their default account username and password from an existing Admin and can change their credentials to something they want here.

Traversing the system

Users will traverse the system by either pressing 'q' to back out of a function , select an option by its index value, or typing to fill in values requested by the system.

Below are some examples:

- Exiting a function
  Below the user exits the Order Menu by pressing 'q'. Hitting enter is unnecessary. Selecting 'q' will either end a process or go back a page depending on the situation.

```
Order Menu
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: ID: #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: ID: #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: ID: #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action: q
Exiting


Hi Kate, what would you like to do?
1. Customer Orders
2. Car Sales
3. Car Inventory
4. Manage Customers
5. Manage Employees
A: Account settings

Press 'q' to exit
Enter action:
```

- Selecting by index
  Below the user selects 'Customer Orders' by typing in 1, its associated index. These indexes will be on the right of every option. The user then presses enter to submit the action.

```
Hi Kate, what would you like to do?
1. Customer Orders
2. Car Sales
3. Car Inventory
4. Manage Customers
5. Manage Employees
A: Account settings

Press 'q' to exit
Enter action: 1
Order Menu
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: ID: #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: ID: #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: ID: #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action:
```

- Filling in values
  Below the user selected 'change username' prompting the user to type in data. For situations for user input, the user will type in the terminal and press 'enter' to submit.

```
Press 'q' to exit
Enter action: A
Account settings
1. Change password
2. Change username
3. View Account Details

Press 'q' to exit
Enter action: 2

Press 'q' to exit
Enter new username: usernameExample
usernameExample
Username changed successfully
```

Conclusion

This guide provides an overview of PiegonBox and how to traverse the system. By following the instructions in this guide, you can manage the vehicles and customer orders effectively.

Operator Guide

Introduction

PigeonBox is a software application that allows you to manage your vehicles and process customer orders using a command line interface. This guide provides instructions on how to operate PigeonBox effectively and troubleshoot any issues.

Installation

To troubleshoot the system, it is recommended that the operator follows the 'Advanced Installation Guide' in 'System Installation'. This goes over different modules that need installation and the preferred IDE of PigeonBox.

System Overview

While using PigeonBox, users will be able to add and delete cars from the inventory, process customer orders, view car sale details, search through the inventory, and manage employee and customer information.

Users can be split into two categories: General Employee and Admin. Admins have all the functions of General Employees plus the authority to add and remove cars from the inventory and manage employees like viewing their details and adding them to the system. You may think of Admins as the managers of the car dealership that have functions that require a certain level of authority to perform.

Depending on the user type the system is broken up into different menus. Once logging in, General Employees will be able to view Car Inventory, Customer Orders, Car Sales, Manage Customer, and Account Settings. Admins will have these menu options plus Manage Employees.

Below is each menu and their function:

- Car Inventory
  Users will be able to search the dealerships' inventory for vehicles in this menu. Users have the option to search cars by make, model, and year, filter by their status, make a customer order, or view a car's details. From this menu, you will also have access to edit a vehicle's details like price, mileage, and packages if needed. Only users with admin privileges will be able to add and remove cars from the dealership's inventory in this menu.

- Customer Orders
  Users can manage customer orders from this menu. This includes viewing order details, making an order, or changing the order status from ordered to delivered, marking the end of the order process. Users should only do this once the car reaches the customer's home address. After changing the order status, the order will be marked as a sale and will no longer show up in this menu.

- Car Sales
  This menu is for all sales or fully processed customer orders made in the dealership. This means cars that have come into the possession of the customer can be viewed here along with all relevant details.

- Manage Customers
  Here users will be able to manage customers. This includes adding new customers, removing customers, updating their information, and viewing their information.

- Manage Employees

Only admins have access to this menu. Here they'll be able to manage employees on the system, as well as add new employees to the system. Admins will be able to grant admin privileges from this menu.

- Account Settings
  All users have access to this menu. Here they'll be able to update their username and password. New employees of the system are given their default account username and password from an existing Admin and can change their credentials to something they want here.

Traversing the system

Users will traverse the system by either pressing 'q' to back out of a function , select an option by its index value, or typing to fill in values requested by the system.

Below are some examples:

- Exiting a function
  Below the user exits the Order Menu by pressing 'q'. Hitting enter is unnecessary. Selecting 'q' will either end a process or go back a page depending on the situation.

```
Order Menu
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: ID: #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: ID: #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: ID: #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action: q
Exiting


Hi Kate, what would you like to do?
1. Customer Orders
2. Car Sales
3. Car Inventory
4. Manage Customers
5. Manage Employees
A: Account settings

Press 'q' to exit
Enter action:
```

- Selecting by index
  Below the user selects 'Customer Orders' by typing in 1, its associated index. These indexes will be on the right of every option. The user then presses enter to submit the action.

```
Hi Kate, what would you like to do?
1. Customer Orders
2. Car Sales
3. Car Inventory
4. Manage Customers
5. Manage Employees
A: Account settings

Press 'q' to exit
Enter action: 1
Order Menu
0: ID: #2 Made by Employee Jemie Hellard: Volvo S80 for Abad, Nathanael
1: ID: #3 Made by Employee Jemie Hellard: Audi A8 for Eary, Britt
2: ID: #4 Made by Employee Jemie Hellard: Land Rover Range Rover for Pifford, Lazaro
3: ID: #5 Made by Employee Jemie Hellard: Dodge Ram Van 1500 for Collings, Lay
4: ID: #6 Made by Employee Jemie Hellard: Honda Crosstour for Pavett, Hazel
5: ID: #7 Made by Admin Kate Anderson: Nissan Rogue for Aseghehey, Emanuel
What would you like to do?
1. Add order
2. Remove order
3. View order details

Press 'q' to exit
Enter action:
```

- Filling in values
  Below the user selected ‘change username’ prompting the user to type in data. For situations for user input, the user will type in the terminal and press ‘enter’ to submit.

```
Press 'q' to exit
Enter action: A
Account settings
1. Change password
2. Change username
3. View Account Details

Press 'q' to exit
Enter action: 2

Press 'q' to exit
Enter new username: usernameExample
usernameExample
Username changed successfully
```

Error Handling

If an error occurs during the operation of PigeonBox, you will receive an error message on the command line interface. Make sure to read the error message carefully to understand the issue and take necessary action.

To test the system, make sure to run the command ‘pip install pytest-mock’ to install the modules necessary for testing. Run testing by entering the command: ‘pytest’. To make it run silently enter ‘pytest -p’.

Conclusion

This guide provides an overview of PiegonBox, how to traverse the system, and how to troubleshoot any errors that may occur.. By following the instructions in this guide, you can manage the vehicles and customer orders effectively.

Tutorial(s)

Use cases and their activity diagrams in the ‘System Delivery’ give a nice overview of tasks users may encounter and the steps to complete these tasks. There is also a video link at the bottom of this document that will show how to perform each task the system is capable of. This will cover managing employees and customers, creating customer orders, viewing vehicles in the inventory, and more.

Programmer Guide

Introduction

The PigeonBox software system is a CLI(command line interface) that allows users to manage a car dealerships’ inventory and process customer orders. This guide provides technical documentation for developers who will be working on the PigeonBox software system.

System Architecture

PigeonBox’s architecture is built on a one-tier architecture that consists of the client’s interface and the local file system. Clients, the dealership’s employees, directly access the system through a computer interface in the car dealership. This interface is connected to a local file system inside the dealership. The only users who interact with the system are the dealership’s employees, categorized into Admins and General Employees with different privileges. Customers who wish to order cars from the dealership interact with the system by proxy of the dealership employees.

The system is built on Python and stores all data in their specific JSON files. For example, user data is stored in users.json, order data is stored in orders.json, cars are stored in inventory.json and so on. The system is split into modules of data, parsers and PigeonBox. The data module includes the data of the system stored as JSON objects. The parsers module reads and writes to these objects. The PigeonBox module contains the application logic that processes user input.

For diagrams and further details on the system’s architecture, please refer to the document’s System Design section, System Architecture.

Code Conventions

All code for PigeonBox should adhere to the following conventions:

- Use camelCase for variable and method names.
- Use PascalCase for class and interface names
- Use descriptive names for variables, methods, and classes.
- Use comments to document code, including method parameters and return values.
- Follow SOLID principles to ensure maintainability and extensibility of the codebase.

System Requirements

To develop the PigeonBox system, you will need the following software and hardware:

- Visual Studio Code 2019 or later
- Windows 10 or later
- 1GB of RAM or more

Getting Started

To get started with the system, follow these steps:

1. Clone the repository from the project's GitHub page.
2. Open the file on VS Code. Follow the steps from 'Advanced Installation Guide' in the 'System Installation' section.
3. Run the application.

Code Structure

The code for PigeonBox is organized into the following folders:

- PigeonBox: Contains code that runs application logic.
- data: Contains json files that store the system's data.
- parsers: Contains scripts that read and write to the json files.

Scripts in each folder are split based on their functionaility.

Code Sample

Here are some code samples that demonstrate how to perform common tasks in the PigeonBox system:

Pick from index:

```
def PickIndex(arr):
    """  Displays an array of elements and prompts the user to pick an index
to select an element./
        Will display elements in array and ask user to pick one, will return
the index of the element picked
        Useful for when choosing an element to view or remove. Like when
removing a car, this will point to its index in the list
        and the user will be able to remove it by index."""
    arrLength = len(arr) - 1 # length to give user bounds to pick from
    while True:
        displayData(arr) # displays with (index: element) pair
        PrintFormat('Action', '\nPick index from the dislayed list above')
        index = input(f"Enter index [0-{arrLength}] OR enter 'q' to exit: ")
        if index == 'q':
            PrintFormat("Warning", "Exiting\n")
            return # Exiting

        # validation
        if not index.isdigit():
            PrintFormat('Invalid',f"Invalid index! Must be a number")
            StallUntilUserInput()
            continue

        index = int(index)
        if index < 0 or index > arrLength:
            PrintFormat('Invalid',f"Invalid index! Must be greater than 0
AND smaller than maximum length")
```

```
            StallUntilUserInput()
            continue

        return index
```

Get user action:

```
def getAction(validSet={"1", "2", "3"}, msg="Enter action:"):
    """A function to get user input for a valid action from a set of
options."""
    PrintFormat("Important", "\nPress 'q' to exit")
    action = input(f"{bcolors.UNDERLINE}{msg}{bcolors.ENDC} ").lower()
    if action == "q":
        PrintFormat("Warning", "Exiting\n")
        return
    if action not in validSet:
        PrintFormat("Invalid", "Invalid action")
        return getAction(validSet, msg)
    return action
```

Validate user input:

```
def ValidateUserInput(action="action", isNum=False, isEmail=False):
    """Validates user input based on certain conditions like being a number
or an email address."""
    PrintFormat("Important", "\nPress 'q' to exit")
    while True:
        userInput = input(f"Enter {action}: ")
        if userInput.lower() == 'q':
            PrintFormat("Warning", "Exiting\n")
            return
        if not userInput:
            PrintFormat("Invalid", "Bad input")
            continue
        if isNum and (not userInput.isdigit()):
            PrintFormat("Invalid", "Must be a number")
            continue
        if isEmail and ("@" not in userInput or "." not in userInput):
            PrintFormat("Invalid", "Must be a valid email address")
            continue

        PrintFormat("Success", userInput)
        return int(userInput) if isNum else userInput
```

Conclusion

This guide provides an overview of the technical details for developing the PigeonBox system. By following the conventions and guidelines outlined in this guide, developers can ensure that the codebase is maintainable and extensible, and that the software system is robust and performs well.

MAINTENANCE

The goal of maintenance in PigeonBox is to only deploy maintenance for a few years. Initial maintenance should be adaptive to work out the kinks on the software as adaptive maintenance so that it is more easily used by the car dealership's employees. Once these features are worked out, most features should be unnecessary to add to the system. The system is designed as an E-System to accommodate changes to the car dealership industry. The process should remain primarily the same, but if the dealership turns into more of a storage facility that sells cars online to customers, we need to be able to implement that in the system by undergoing changes.

Corrective Maintenance

Any user that encounters an error should inform the operator of that system. They will contact the development team and the team will review that error, which may be a typo in the code. The team will have multiple reviewers for each document, whether it be a script or product documentation. The development team will also use the version control system on GitHub to track changes to files and allow easy rollback if errors occur.

Adaptive Maintenance

New features to the product require documentation to this document and in the code. For example, a future update may include an addition to the Car Sale Menu where maintenance can keep track of cars that need it. All documentation should be kept up to date with the latest changes in PigeonBox. This makes sure the document reflects the current state of the system and prevents misunderstandings.

Perfective Maintenance

The system will incorporate usability testing, which involves testing software with users to identify areas that can be improved to make PigeonBox more usable and intuitive. This will be done during the first launch at the car dealership. We will also perform code refactoring so that the structure and organization of the code is more efficient, readable, and maintainable. To make sure documentation stays consistent, PigeonBox's team is expected to follow style guides described in 'Program Design' and 'Coding' sections.

Documentation guidelines are as followed:

- Keep language concise and simple.
- Do not make deep nested bullet points.
- Do not make unnecessarily long paragraphs.
- If bullet points easily convey the information, use them.
- Create a diagram/illustration if it helps readers understand the concept.

Preventative Maintenance

We cover preventative maintenance by making our documentation standards a priority. This includes standards for documenting code and processes to make sure that the documentation is clear, complete, and up to date. Any changes made to the code should be documented in the software documentation and they should follow the existing coding style and standards.

PigeonBox's team will also perform periodic reviews to identify areas that could be improved or updated. This will happen around once a month.

IMPORTANT LINKS

GitHub Repository: https://github.com/aseghehey/vehicle-tracking-system.git
Product demonstration/tutorial video:
https://drive.google.com/file/d/1eWkdzJPx_8bhhCGgYYpI9F4RXaTdlAy7/view