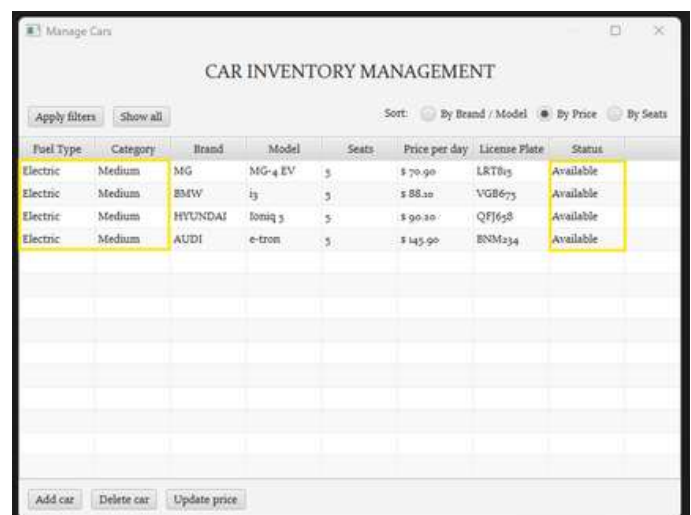
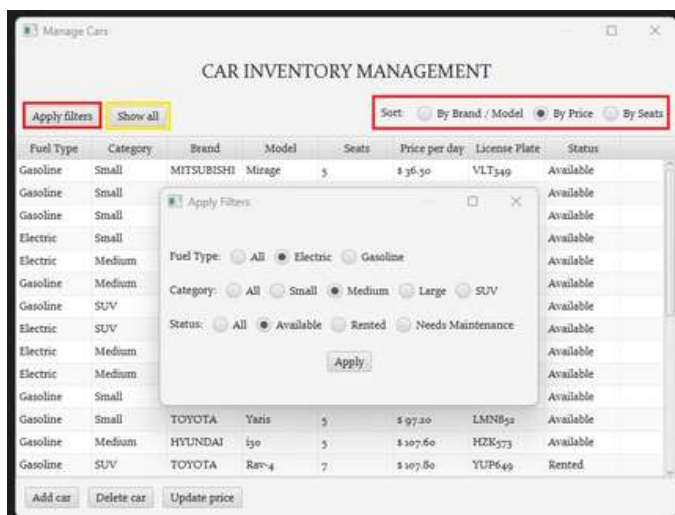


# GUI Design

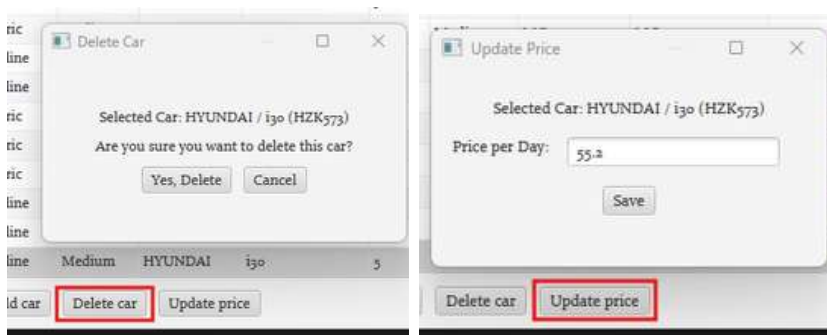


This GUI implements a sophisticated system for managing a car rental business. It is specifically designed for use by **employees of a rental car company**, facilitating four major operational use cases; Manage Cars, Rent Cars, Return Cars, and Repair Cars. The main menu page is designed with four large buttons, each representing a specific use case, allowing users to easily navigate to their desired functionality.

## 1) Manage Cars



In the 'Manage Cars' menu, users can view all car inventory owned by the business in a table view. By clicking the **"Apply Filter"** button in the upper left corner, users can filter the view based on criteria such as fuel type, category, and status. The selected filter settings are saved until users reload the page or choose to show all. By clicking the **"Show All"** button, users can reset any applied filters and view all cars again. Additionally, the **"Sort"** radio buttons in the upper right corner allow users to choose the sorting method for the displayed vehicles.



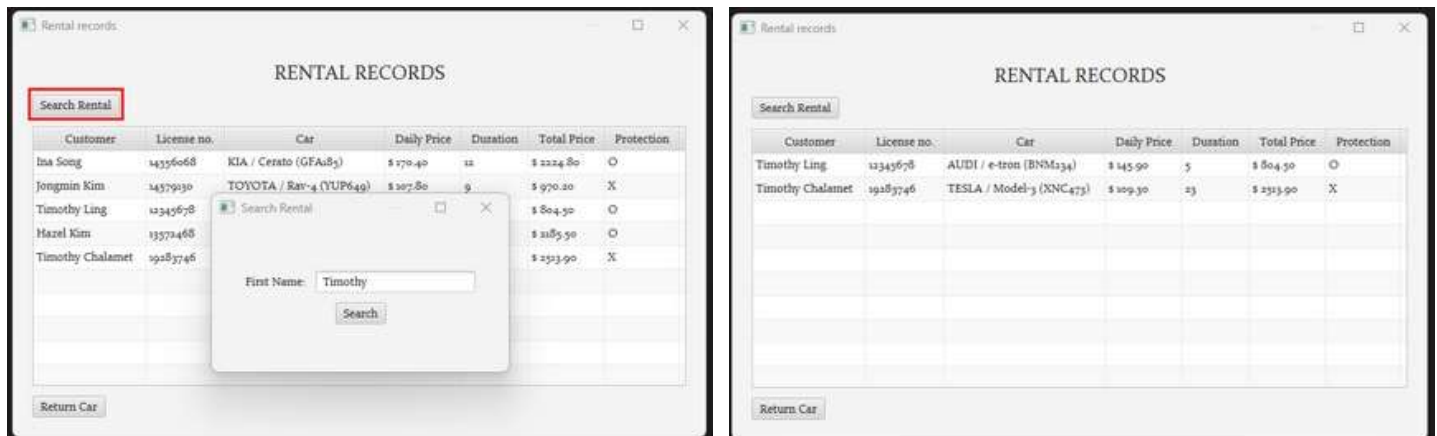
After selecting a specific car from the list, users can delete the car by clicking the **"Delete Car"** button, which will take them to a confirmation page before the deletion is finalized. Additionally, by clicking the **"Update Price"** button, users can change the price of the selected vehicle.

When users click the **"Add Car"** button, the add car form appears, allowing them to fill in the required fields and submit the form to add a new car. Each field is implemented to accept only the correct format(3 alphabets + 3 numbers for license plate, only alphabets for brand, and only numbers for seats and price), ensuring data accuracy. In addition, if users leave the required field blank or enter an incomplete value, the submission will not go through, and an error message will appear next to the field.

## 2) Rent Cars

In the 'Rent Cars' menu, users can rent a car by completing a form that consists of three stages: User Information, Car Selection, and Rental Information. The form is designed with checkboxes, and formatted text fields. If any required input is missing or invalid, an error message is displayed to guide the user, ensuring data accuracy and user-friendly experience. When users click the **"Select Car"** button, a car selection window appears, allowing them to choose their desired car through filtering and sorting options. The price information is instantly updated and displayed at the bottom of the form based on the user's selection.

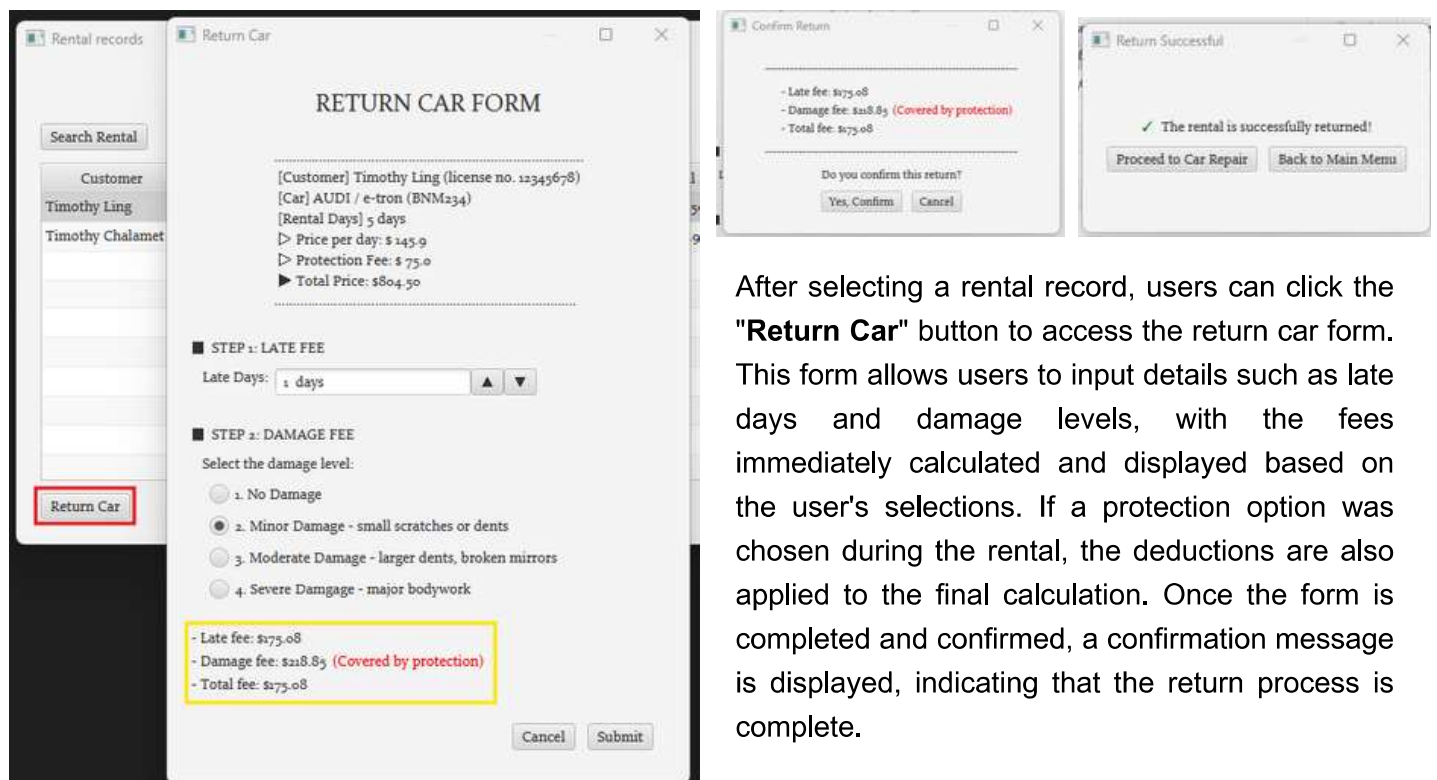
### 3) Return Cars



**RENTAL RECORDS**

Customer	License no.	Car	Daily Price	Duration	Total Price	Protection
Ina Song	14356068	KIA / Cerato (GFA85)	\$ 170.40	12	\$ 2224.80	O
Jongmin Kim	14379130	TOYOTA / Rav-4 (YUP649)	\$ 107.80	9	\$ 970.20	X
Timothy Ling	12345678	AUDI / e-tron (BNM234)	\$ 145.90	5	\$ 804.50	O
Hazel Kim	13572468		\$ 1185.90			
Timothy Chalamet	19283746	TESLA / Model-3 (XNC473)	\$ 109.90	23	\$ 2513.90	X

In the 'Return Cars' menu, rental records containing customer, car, and rental information are displayed in a table view. By clicking the **"Search Cars"** button in the upper left corner, a search window appears, allowing users to easily retrieve rental records by entering the customer's first name.



**RETURN CAR FORM**

[Customer] Timothy Ling (license no. 12345678)  
 [Car] AUDI / e-tron (BNM234)  
 [Rental Days] 5 days  
 ▶ Price per day: \$ 145.9  
 ▶ Protection Fee: \$ 75.0  
 ▶ Total Price: \$804.50

**STEP 1: LATE FEE**  
 Late Days: 1 days

**STEP 2: DAMAGE FEE**  
 Select the damage level:

- ☐ 1. No Damage
- ☒ 2. Minor Damage - small scratches or dents
- ☐ 3. Moderate Damage - larger dents, broken mirrors
- ☐ 4. Severe Damage - major bodywork

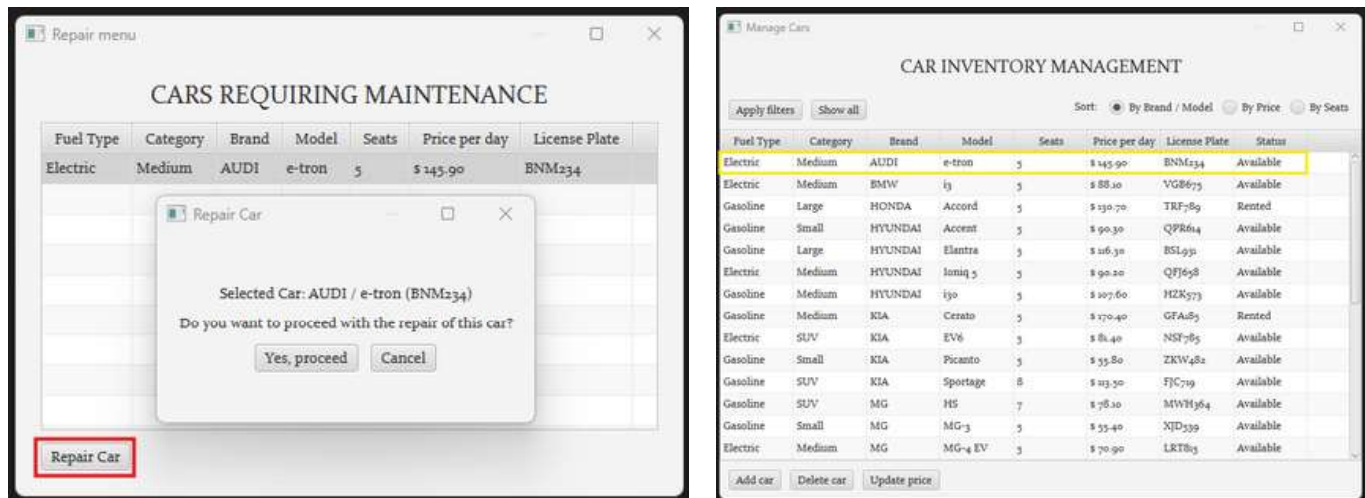
- Late fee: \$175.08  
 - Damage fee: \$218.85 (Covered by protection)  
 - Total fee: \$175.08

Do you confirm this return?  
 Yes, Confirm Cancel

✓ The rental is successfully returned!  
 Proceed to Car Repair Back to Main Menu

After selecting a rental record, users can click the **"Return Car"** button to access the return car form. This form allows users to input details such as late days and damage levels, with the fees immediately calculated and displayed based on the user's selections. If a protection option was chosen during the rental, the deductions are also applied to the final calculation. Once the form is completed and confirmed, a confirmation message is displayed, indicating that the return process is complete.

## 4) Repair Cars



In the 'Repair Cars' menu, cars that have been returned with recorded damage are displayed in a table view. After selecting a vehicle that requires repair, users can click the **"Repair Car"** button, which, after confirmation, updates the car's status to "available" once the repair is completed. If there are no cars needing repair, an error message is displayed indicating that there are no vehicles to repair, and the table view will not be shown.

# Explanation

## 1) Model

In the Car Rental Business, the Model is responsible for managing the core data and business logic. It maintains three primary observable lists: inventory, customisedInventory, and rentals. The 'inventory' list contains the full set of cars available in the system, while the 'customisedInventory' is a subset of this list that can be filtered or sorted according to user-defined criteria. By using 'customisedInventory', the system ensures that any filtering or sorting operations do not directly alter the original inventory data, preserving the integrity of the actual car stock. The 'rentals' list tracks the active rental records. The Model also initializes the application with a predefined set of cars in the inventory and a few rental records in the rentals list. These initial records are created in the Model's constructor, ensuring that the system has data to operate on from the start. The Model ensures that any changes to the data are properly validated and reflected across the system.

## 2) View

The View is responsible for managing the user interface and facilitating interaction between the user and the application. It presents the system's data in a visual format through various forms, tables, buttons, and multiple windows, allowing users to interact with different aspects of the application. The View captures user inputs and validates these inputs to ensure they meet the required format. If a user inputs invalid data, the View provides immediate feedback by displaying error messages, guiding the user to correct their input. This validation ensures that the data being sent to the Controller for processing is accurate and within the expected parameters. Additionally, the View dynamically updates to reflect changes in the Model, such as when a new car is added or the rented car is returned, ensuring the user always sees the most current information. The View also manages the presentation and navigation of different windows within the application, thereby structuring user interaction in a clear and organized manner.

## 3) Controller

The Controller acts as the intermediary between the user interface (View) and the underlying data (Model). It handles user inputs from the View, processes them, and updates the Model accordingly. For instance, when a user applies a filter or sorts the car inventory, the Controller interprets these commands, retrieves the necessary data from the Model, and instructs the View to update the display. The Controller ensures that the application logic is executed correctly and that the Model and View remain in sync throughout the application's lifecycle.

## 4) Flow of MVC pattern (Rent Cars)

### 1. Initiate form and validate input(User ↔ View):

The user initiates the process by selecting the "Rent Cars" option from the main menu. The View responds by displaying the rental form. The user fills in the various fields of the form, and the View validates the inputs, prompting the user with error messages if the inputs are invalid or incomplete.

### 2. Display car list(View → Model):

The view retrieves the full list of available cars directly from the Model using 'updateTableView()' method and displays it in a table view.



### 3. Apply filter or sort(View → Controller → Model):

When the user choose to filter or sort the car list, the View captures these choices and sends them to the Controller. The Controller processes these options using '**handleSortBy()**' and '**applyFilters**' methods, which updates the customisedInventory list based on the selected criteria by calling '**sortBy()**' and '**filterCars()**' methods. The returned list from these methods then update the View to reflect the filtered or sorted car options.

```
Button showAllBtn = new Button(text:"Show all");
showAllBtn.setOnAction(e -> updateTableView(controller.applyFilters(fuelType:"All", category:"All", status:"All")));
ToggleGroup sortGroup = new ToggleGroup();
RadioButton ByBrandBtn = new RadioButton(text:"By Brand / Model");
ByBrandBtn.setToggleGroup(sortGroup);
ByBrandBtn.setOnAction(e -> updateTableView(controller.handleSortBy(Comparator.comparing(Car::getBrand).thenComparing
```

**View**

```
public ObservableList<Car> handleSortBy(Comparator<Car> comparator) {
    currentComparator = comparator;
    model.sortBy(comparator);
    return model.customisedInventoryProperty();
}

public ObservableList<Car> applyFilters(String fuelType, String category, String status) {
    this.currentFuelTypeFilter = fuelType;
    this.currentCategoryFilter = category;
    this.currentStatusFilter = status;

    model.filterCars(fuelType, category, status);
    model.sortBy(currentComparator);

    return model.customisedInventoryProperty();
}
```

**Controller**

```
public void sortBy(Comparator<Car> comparator) {
    if (comparator != null) {
        Collections.sort(customisedInventory, comparator);
    }
}

public void filterCars(String fuelType, String category, String status) {
    ObservableList<Car> filteredList = FXCollections.observableArrayList();
    for (Car car : inventory) {
        boolean matchesFuelType = fuelType.equals(anObject:"All") || car.getFuelType().toString().equals(fuelType);
        boolean matchesCategory = category.equals(anObject:"All") || car.getCategory().toString().equals(category);
        boolean matchesStatus = status.equals(anObject:"All") || car.getStatus().toString().equals(status);

        if (matchesFuelType && matchesCategory && matchesStatus) {
            filteredList.add(car);
        }
    }
    this.customisedInventory = filteredList;
}
```

**Model**

### 4. Submit form and create rental (View → Controller → Model):

After selecting a car and completing the form, the user submits the rental form. The View creates a Rental object with all entered data, and send it to the Controller by calling '**addRental()**' method. The controller changes the status of the car and updates the rentals list in the Model.