University of Dhaka

Department of Computer Science and Engineering

CSE 3216: : Software Design Patterns Lab

Assignment

Special Car Company (SCC)

Submitted By:

1. Syed Mumtahin Mahmud, Roll: 50

2. Nazira Jesmin Lina, Roll: 55

Submitted On :

August 21, 2023

Submitted To :

Redwan Ahmed Rizvee

Contents

1 Introduction

In this problem, we are tasked with designing a system for the "Special Car Company (SCC)," which is the world's largest car construction parent company. SCC consists of various child companies that produce different car models. The focus of this problem is to develop an architecture for this system, considering various design principles and patterns. The problem provides us with several contextual elements and constraints that need to be incorporated into the system's design. The goal is to create a robust and flexible system that can handle various aspects of car manufacturing, customization, notifications, and customer interactions.

1.1 Problem Entities:

1. **Car:** The central entity of the system, with attributes such as Engine, Tires, Chassis, AC, Color, Total Price, Body Design, Automated AI, and Seats. Each of these attributes contributes to the car's features and cost.
2. **Engine, Tires, Chassis, AC, Color:** Different components that make up a car, with various options and types available.
3. **Body Design:** Specific to each group of cars (Ferrari, Ford, Toyota, BMW, Chevrolet) and their variants (Racing car, Private Car, SUV, Military Vehicle).
4. **Automated AI:** AI system for automated driving, with variations based on geographic regions.
5. **Seats:** Vary based on car type (Racing car, Private Car, SUV, Military Vehicle).
6. **Decorating Entities:** Customization options for clients, including Customized Rain Shield, Bumper type, Gate Controlling System, and Open Roof System.
7. **Notification System:** Provides Price Change and Car's basic features change notifications to subscribed clients.
8. **Car Services:** Requests for car servicing, washing, and online delivery.
9. **Web-based System:** The central online platform where clients can interact with the SCCś services, including car customization, notifications, and services.
10. **Mobile Application:** A new mobile application is being developed to extend the accessibility of SCCś services to clients.

1.2 Language and Framework:

Python (3.11.4) is used for the implementation purpose. Python is a high-level, interpreted programming language known for its simplicity and readability. It supports object-oriented programming (OOP) principles, allowing us to create and use classes and objects.

2 Model Design pattern:

2.1 Car module -Factory Method Pattern:

In this module,the Factory Method pattern is used to create various car models (RacingCar, PrivateCar, SUVCar, MilitaryCar). Each of these concrete classes inherits from the abstract class Car and overrides the 'prepareCar' method to create specific instances of car components.

2.1.1 Design Rationale:

The Factory Method pattern is suitable for this system as it allows the creation of different car models while encapsulating the creation logic within the concrete subclasses. This approach adheres to the Open-Closed Principle, as new car models can be added without modifying existing code. Each car model's specific configuration is encapsulated within its respective class, making the system more modular and extensible.

2.1.2 Pros and Cons of the Solution:

Pros:

- Supports the addition of new car models without modifying existing code.
- Enhances modularity by encapsulating creation logic within subclasses.
- Supports a consistent interface for creating car instances.

Cons:

- Can lead to a proliferation of subclasses if there are many car models.
- Subclasses might have similar code for creating components, leading to code duplication.

2.1.3 UML Class Diagram:

In the UML diagram, we have the following classes and relationships:
Abstract Classes:

- Car is an abstract class with attributes representing car components. It has abstract methods prepareCar, cost, and getDescription.
- CarPartsFactory, CompanyFactory, AutomationFactory, Seat, AC, Chassis, Engine, and Tire are also abstract classes or interfaces defining various aspects of car creation.

Concrete Subclasses:

- RacingCar, PrivateCar, SUVCar, and MilitaryCar are subclasses of Car, representing specific types of cars.
- Subclasses like HighPoweredAC, LowPoweredAC, Tabular, Backbone, LadderFrame, CC1300, CC1700, CC1800, CC2100, Snow, Spare, Whitewall, and Slick provide concrete implementations for specific car components.

Relations:

- Inheritance relationships exist between the abstract class Car and its concrete subclasses (RacingCar, PrivateCar, SUVCar, MilitaryCar).
- Composition relationships are shown between Car and its attributes (engine, tire, ac, chassis, carColor, seat, bodyDesign, automation).

The diagram would show the inheritance relationships between classes and the composition relationships between Car objects and their decorators.

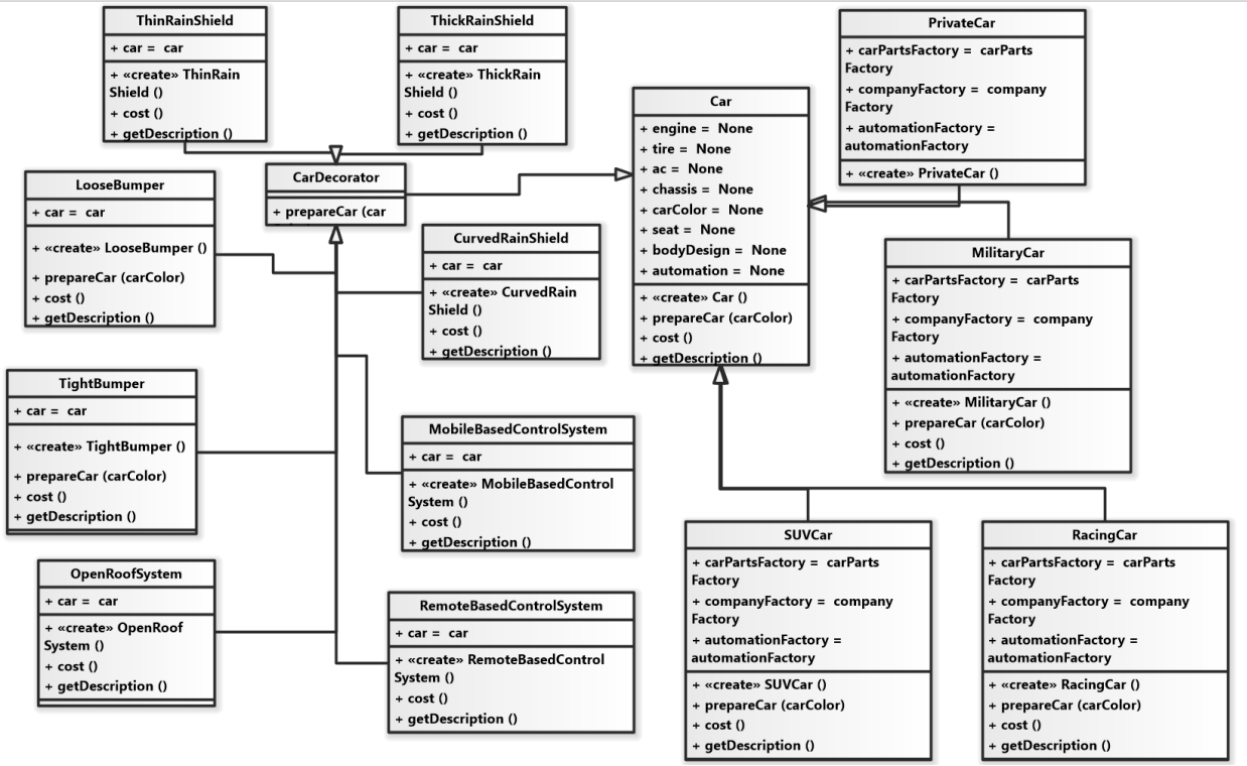

Figure 1: UML Class diagram for Car module and Customization module

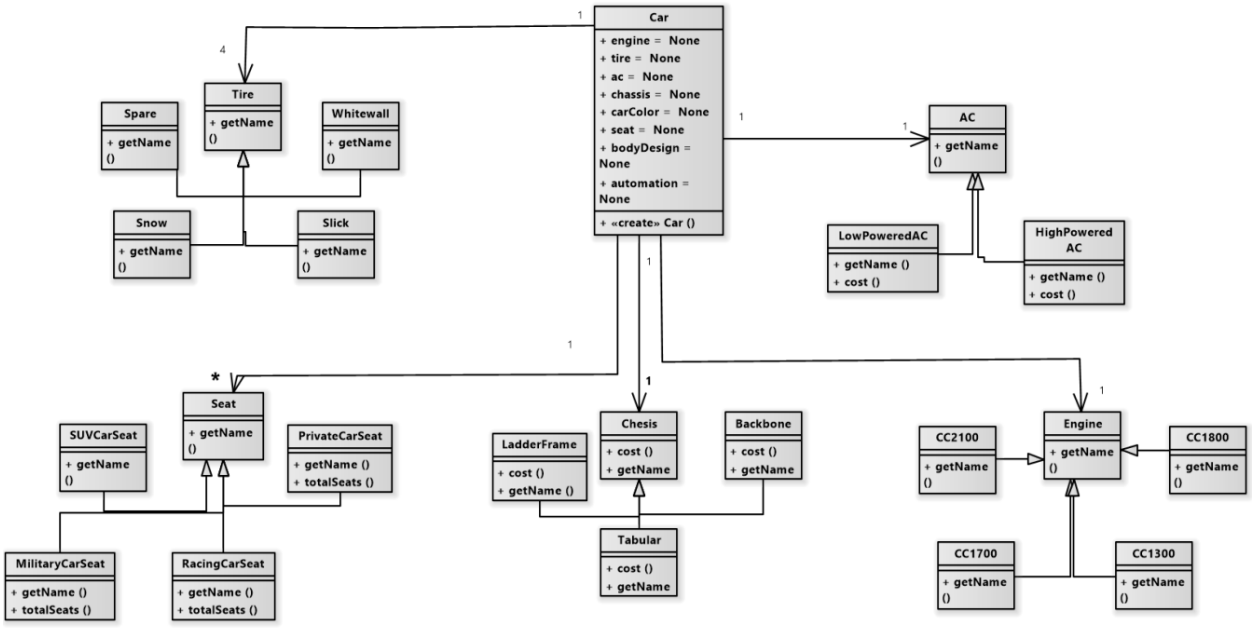

Figure 2: UML Class diagram for Car module

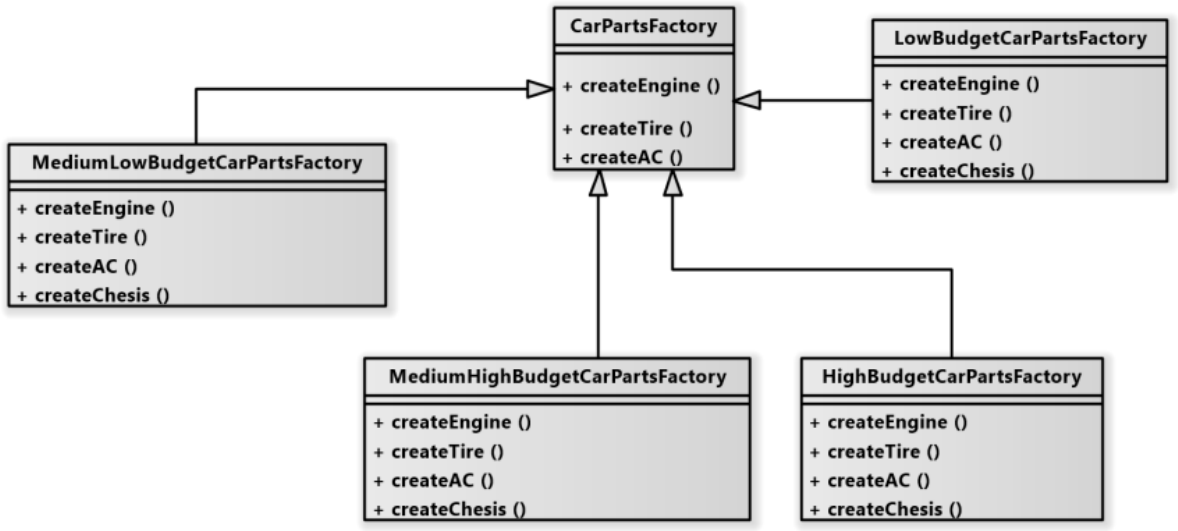

Figure 3: UML Class diagram for Car module

2.2 SCC module-Factory Method Pattern along with Abstract Factory Pattern:

We have used two types of pattern in this module.They are factory method pattern and abstract factory method pattern.

2.2.1 Used pattern:

(a) **Factory Method Pattern:** The SCC class implements the Factory Method pattern. It has an abstract method createCar() which is intended to be overridden by subclasses. The method is used to create different types of cars based on the carType parameter.

(b) **Abstract Factory Pattern:**

- There are three implementations of the Abstract Factory pattern: CompanyFactory, CarPartsFactory, and AutomationFactory.
- CompanyFactory and its subclasses (FordsCompanyFactory, FerrariCompanyFactory, etc.) represent different car company factories, each capable of creating a specific body design.
- CarPartsFactory and its subclasses (LowBudgetCarPartsFactory, MediumLowBudgetCarPartsFactory, etc.) represent different car parts factories based on the budget type.
- AutomationFactory and its subclasses (AsiaAutomationFactory, AmericaAutomationFactory) represent different automation factories.

2.2.2 Design Rationale

(a) **Factory Method Pattern:**

- The Factory Method pattern is used in the SCC class to encapsulate the car creation process. This allows for the creation of different types of cars without modifying the core logic of the class.
- This approach adheres to the Open-Closed Principle by enabling the addition of new car types without modifying existing code.

(b) **Abstract Factory Pattern:**

- The Abstract Factory pattern is employed to provide an interface for creating families of related objects (car parts, body designs, automation systems) without specifying their concrete classes.
- This pattern ensures that related objects are created together, maintaining consistency and avoiding incompatible object combinations.

2.2.3 Pros and Cons of the Solution:

(a) **Factory Method Pattern:**
Pros:

- Supports extensibility by allowing new car types to be added easily.
- Supports extensibility by allowing new car types to be added easily.

- Encapsulates object creation logic, promoting a clean separation of concerns.
- Provides a consistent interface for creating cars across different types.

Cons:

- Requires the implementation of the 'createCar()' method in subclasses, which can lead to code duplication if not managed properly.
- Can lead to an increase in the number of subclasses for different car types.

(b) **Abstract Factory Pattern:**
Pros:

- Enforces the creation of compatible object families, ensuring that all objects work together cohesively.
- Facilitates the creation of objects with a consistent interface, regardless of their concrete class.
- Supports the principle of Dependency Inversion by allowing high-level classes to depend on abstractions (interfaces) rather than concrete classes.

Cons:

- Can lead to a complex class hierarchy, especially when multiple families of objects are involved.
- Can lead to a complex class hierarchy, especially when multiple families of objects are involved.
- Adding new product variations (new car parts, body designs) may require modifying existing factory interfaces and their implementations.

2.2.4 UML Class Diagram:

In the UML diagram, we have the following classes and relationships:

Abstract Classes:

- SCC is an abstract class. It defines methods like orderCar, serviceCar, and onlineOrder, which provide the primary functionalities of the system.
- BodyDesign, CompanyFactory, AutomationFactory, CarPartsFactory, and Automation are also abstract classes. They provide interfaces for creating related objects or families of objects.

Interfaces:

- BodyDesign, CompanyFactory, AutomationFactory, and CarPartsFactory are interfaces that define the contract for creating specific types of objects.
- CarPartsFactory defines methods like createEngine, createTire, createAC, and createChesis, which are implemented by concrete factories.

Subclasses and Relationships:

- Fords, Ferrari, Toyota, BMW, and Chevrolet are subclasses of BodyDesign. Each represents a different car company's body design.

- FordsCompanyFactory, FerrariCompanyFactory, ToyotaCompanyFactory, BMWCompanyFactory, and ChevroletCompanyFactory are subclasses of CompanyFactory. They create specific body designs based on the company.
- AsiaAutomationFactory and AmericaAutomationFactory are subclasses of AutomationFactory. They create automated AI systems based on different regions.
- AsiaAutomation and AmericaAutomation are subclasses of Automation. They represent different types of automated AI systems.

Relationships:

The abstract class SCC has a composition relationship with interfaces like CarPartsFactory, CompanyFactory, and AutomationFactory. The SCC class has a dependency relationship with BodyDesign, CompanyFactory, AutomationFactory, and CarPartsFactory, using the factory methods to create objects and assemble car parts. The SCC class has a dependency relationship with the WebBackend class, indicating a collaboration for service and delivery requests. The RacingCar, PrivateCar, SUVCar, and MilitaryCar classes are concrete classes that implement the abstract methods of the Car class. The decorator classes have a composition relationship with the Car classes, indicating that they enhance the car objects' functionality. The concrete factory classes like FordsCompanyFactory, AsiaAutomationFactory, etc., provide implementations for the interfaces they inherit from.

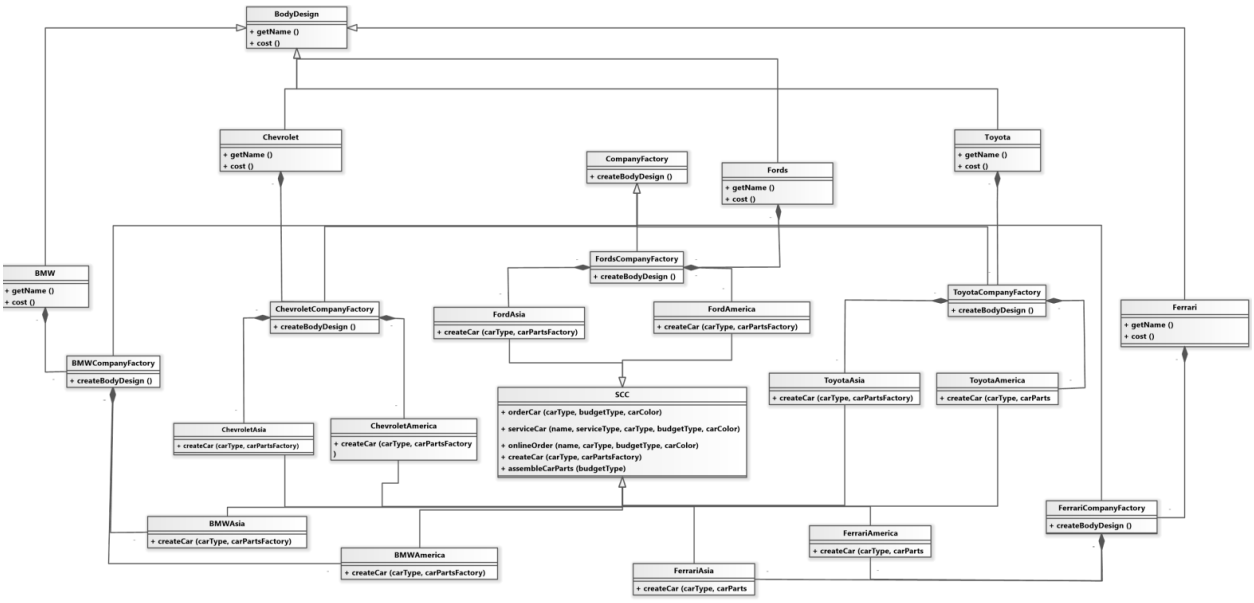

Figure 4: UML Class diagram for SCC module

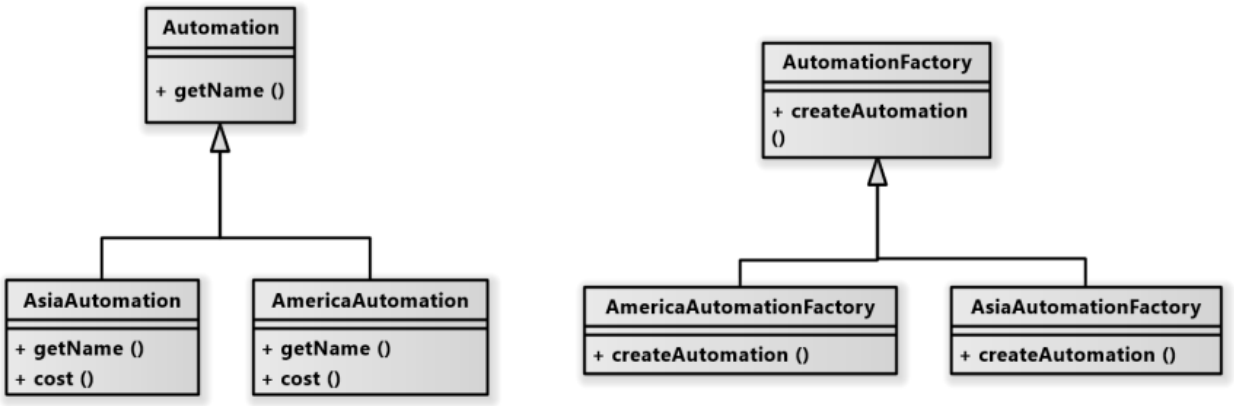

Figure 5: UML Class diagram for SCC module's automation section

2.3 Car Customization module-Decorator Pattern:

The Decorator pattern is used to add additional features or customizations to the base Car objects. The CarDecorator abstract class serves as the base decorator, and different concrete decorator classes (ThickRainShield, ThinRainShield, CurvedRainShield, TightBumper, LooseBumper, MobileBasedControlSystem, RemoteBasedControlSystem, OpenRoofSystem) extend it to provide various customizations.

2.3.1 Design Rationale:

The Decorator pattern is applied to provide a flexible way to add customizations to cars without altering their core structure. This pattern promotes the Single Responsibility Principle by ensuring that each decorator class focuses on a specific customization. It allows for the dynamic addition of new features or changes to existing ones, enhancing the system's flexibility and maintainability.

2.3.2 Pros and Cons of the Solution:

Pros:

- Provides a flexible way to add functionalities to objects without modifying their structure.
- Allows for the dynamic composition of features.
- Enhances the readability and maintainability of the code by separating concerns.

Cons:

- Can lead to the creation of a large number of decorator classes for various customizations.
- May introduce complexity if decorators are nested extensively.

2.3.3 UML Class Diagram:

In Fig.1 we have attached the uml class diagram of customization module which is closely related to the car module

2.4 Notification Module-Singleton pattern along with Observer Pattern:

We have used two pattern in this module.they are Singleton pattern and Observer Pattern.

2.4.1 Used design pattern:

(a) **Singleton Pattern:** The Singleton pattern is used in the NotificationSystem class to ensure that only one instance of the class exists throughout the application's lifecycle. The '_unique_instance' attribute holds the single instance of the class, and the 'get_instance' method is responsible for creating and returning that instance.

(b) **Observer Pattern:**The Observer pattern is used to implement the notification system. Here's how it's applied:

- NotificationSystem' acts as the subject (observable) class that keeps track of subscribers and notifies them when changes occur.
- NotificationSystem' acts as the subject (observable) class that keeps track of subscribers and notifies them when changes occur.
- Subscriber' is an abstract class that defines the methods required for observers (subscribers).
- SubscriberClient' is a concrete observer class that subscribes to the 'NotificationSystem' and receives update notifications when changes occur.

2.4.2 Design Rationale:

(a) **Singleton Pattern:**The Singleton pattern ensures that there is only one instance of the 'NotificationSystem' throughout the application. This is useful for maintaining a single source of truth for managing subscribers and notifications. The singleton approach simplifies access to the notification system and avoids unnecessary multiple instances.

(b) **Observer Pattern:** The Observer pattern enables a loosely coupled communication between the notification system and its subscribers. This ensures that subscribers receive updates without needing to be tightly integrated with the subject. Subscribers can easily subscribe, unsubscribe, and receive updates without impacting other parts of the system.

2.4.3 Pros and Cons of the solution:

(a) **Singleton Pattern:**
Pros:

- Single instance ensures global access.
- Lazy initialization improves memory efficiency.
- Provides centralized resource management.

Cons:

- Introduces global state.
- Testing can be challenging due to global nature.
- Can lead to tight coupling between components.

(b) **Observer Pattern:**
Pros:

- Promotes loose coupling between subject and observers.
- Allows easy addition/removal of observers without affecting others.
- Subscribers receive real-time updates.
- Separates concerns for decoupled architecture.

Cons:

- Adds complexity to the codebase.
- Managing order of notifications can be complex.
- Poor memory management can lead to leaks.

2.4.4 UML Class Diagram:

Here's a description of the UML class diagram elements and their relationships:

Abstract Classes: NotificationSystem is an abstract class, representing the subject of the Observer pattern. It manages subscribers and notifies them of changes.

Concrete Classes: SubscriberClient is a concrete class representing the observers in the Observer pattern. It subscribes to the NotificationSystem to receive updates.

Relationships:

- SubscriberClient‘ has a composition relationship with NotificationSystem. Each SubscriberClient instance holds a reference to the single instance of NotificationSystem.
- NotificationSystem implements the Singleton pattern with a static _unique_instance‘ attribute and a get_instance() method to ensure a single instance of the class.
- NotificationSystem has a ‘subscriber_set‘ attribute, representing a set of subscribers.
- NotificationSystem has an association relationship with SubscriberClient. It registers and removes subscribers from the set.
- NotificationSystem has a dependency relationship with SubscriberClient through the update() method, which is invoked to notify subscribers of changes.

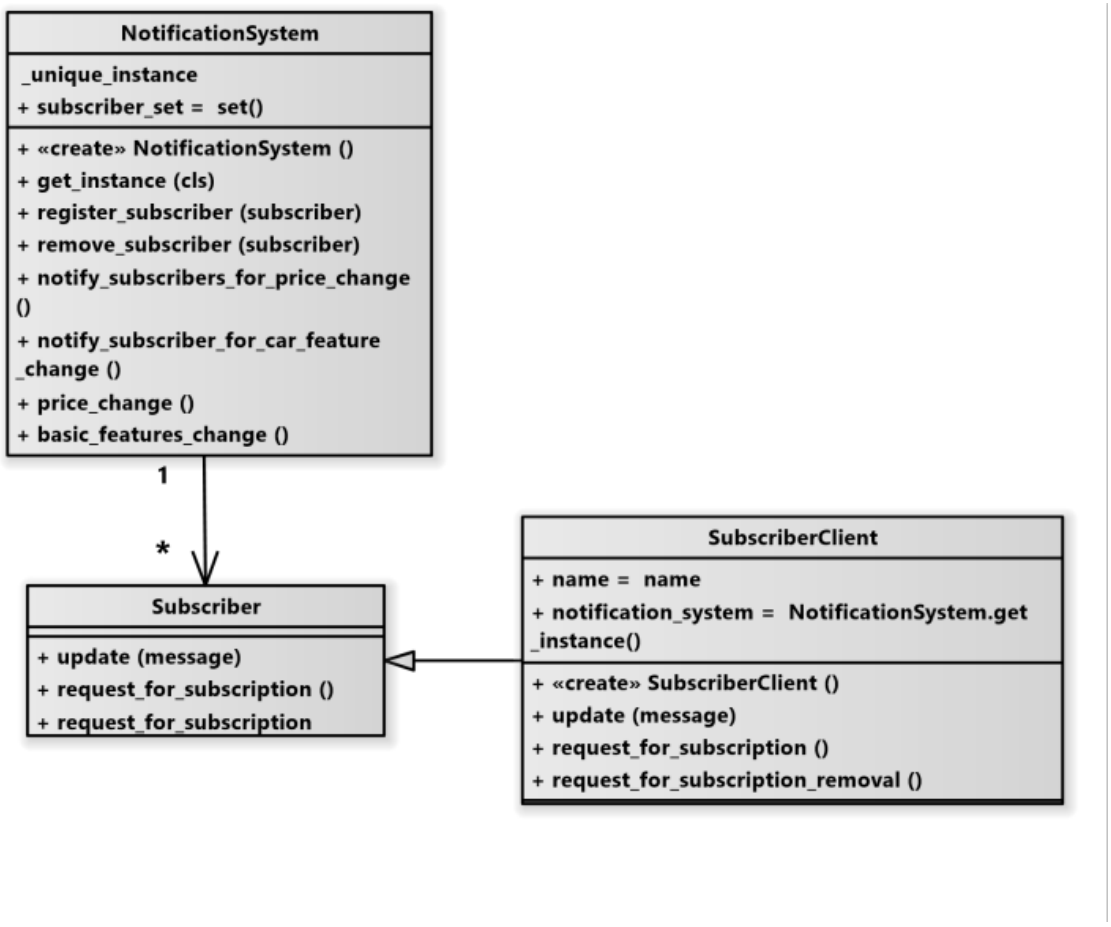

Figure 6: UML Class diagram for Notification module

2.5 Central Online system module-Command pattern:

In this module we have used Command pattern.

2.5.1 Command Pattern:

The Command Pattern is a behavioral design pattern that turns a request into a stand-alone object. In this code, the CarService classes (CarWashingService and CarServicingService) are implementing

the Command Pattern. These classes encapsulate the actions (washing and servicing) as commands, and the WebBackend class acts as the invoker of these commands.

2.5.2 Design Rationale:

1. **Separation of Concerns:** The code separates different concerns into different classes. The WebBackend class is responsible for handling service requests and deliveries, while the CarService hierarchy handles the execution of specific services, and the OnlineDeliveryService handles car deliveries.

2. **Abstraction and Encapsulation:** The CarService class and its subclasses encapsulate the behavior of car services, abstracting away the specific details of each service.

3. **Interface-based Programming:** The CarService class defines an abstract method execute(), which enforces that subclasses must implement this method.

4. **Polymorphism:** Polymorphism is used through the abstract execute() method, which allows different implementations to be used interchangeably in the WebBackend class.

2.5.3 Pros and Cons of the solution:

Pros:

- Decoupling: The Command Pattern decouples the WebBackend class from the specific service execution classes (CarWashingService and CarServicingService).

- Extensibility: New service execution classes can be added without modifying the WebBackend class.

- Centralized Control: The WebBackend class becomes a centralized point of control for service execution.

- Logging and Auditing: The pattern allows for easy logging and auditing of service requests and executions.

- Undo/Redo and History: The Command Pattern supports undo and redo operations and can maintain a history of executed commands.

Cons:

- Complexity: The Command Pattern introduces complexity due to the creation of command classes and their interactions.

- Overhead: For a small-scale application, the Command Pattern might introduce unnecessary overhead.

- Learning Curve: Developers unfamiliar with the Command Pattern might find it challenging to understand the added abstraction.

2.5.4 UML class Diagram:

Abstract Class: We have used SCC abstract class as the middleware as we want to show the car info in the Central online system.

- SCC (Super Car Company) is an abstract class. It is denoted by the use of the keyword 'ABC' (Abstract Base Class) in its definition.
- It has abstract methods (createCar), which are indicated by the @abstractmethod decorator.
- It also has concrete methods (orderCar, serviceCar, onlineOrder, assembleCarParts).

Concrete Classes:

- WebBackend is a concrete class responsible for handling service requests and deliveries.
- CarService is an abstract class with abstract method execute, defining the template for car services.
- CarWashingService and CarServicingService' are concrete subclasses of CarService, implementing specific car services.
- OnlineDeliveryService is a concrete class for delivering cars.

Relationships:

- SCC is related to CarPartsFactory through the method assembleCarParts which we have discussed earlier.
- SCC is related to WebBackend through methods serviceCar and onlineOrder.
- SCC is related to Car through the method createCar.
- WebBackend has association relationships with CarWashingService, CarServicingService, and OnlineDeliveryService, indicating that it uses these services to fulfill its responsibilities.
- CarService has a dependency relationship with SubscriberClient through the update method.
- CarWashingService and CarServicingService have aggregation relationships with CarType indicating that they use car types.

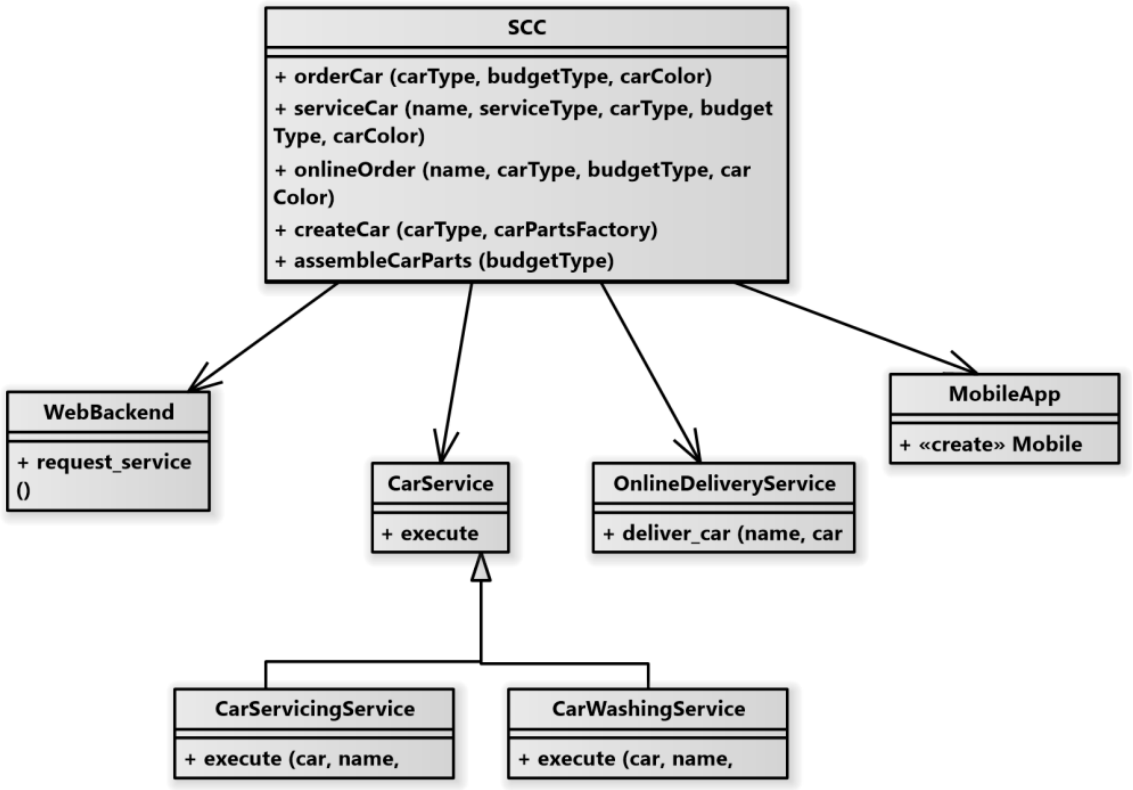

Figure 7: UML Class diagram for Central online system module

2.6 Mobile App module-Adapter Pattern:

the Adapter pattern is used in a way that simplifies the interaction between a mobile application and the existing SCC class, allowing seamless ordering of cars or services. It provides a streamlined interface for mobile users while leveraging the existing functionality of the SCC class.

2.6.1 Adapter Pattern:

The MobileApp class acts as an adapter to the SCC class. It provides a simplified interface for ordering cars or services via a mobile app. It adapts the different methods of the SCC class for the mobile app's usage.

2.6.2 Design Rationale:

- The adapter pattern is used in the MobileApp class to adapt the methods of the SCC class for use in a mobile application context.
- It allows existing classes to be reused in different contexts without modifying their code.
- The adapter provides a simplified interface tailored for mobile app users.

2.6.3 Pros and Cons of the solution:

Pros:

- the Adapter pattern allows the reuse of existing classes with incompatible interfaces in new contexts. In this case, the 'MobileApp' class acts as an adapter, enabling the use of methods from the SCC class in the context of a mobile application.
- The Adapter pattern facilitates the integration of different systems or components that have varying interfaces. The MobileApp class provides a unified interface that abstracts the complexities of the underlying SCC class methods.
- Adapting existing classes via an adapter minimizes the need to modify or refactor the existing codebase. This is particularly useful when you want to use existing functionality in new scenarios without altering the original code.
- Any changes or updates needed for adapting the existing class are localized to the adapter. This prevents the need to modify the core functionality of the SCC class, maintaining stability and minimizing regression risks.

Cons:

- The Adapter pattern adds an extra layer of indirection between the client code and the adapted class. This can slightly affect performance and introduce a level of complexity, especially in cases where the number of adapters increases.
- Depending on the differences between the original class and the required interface, the adapter itself can become complex. This complexity can make the code harder to understand and maintain.
- The introduction of adapters can introduce some overhead in terms of memory usage and processing time, although these overheads are usually minimal in most applications.
- If the original class's interface changes, it can affect the adapters that depend on it. When adapting multiple classes, any change in the core classes might require updates in multiple adapters.

2.6.4 UML Class Diagram:

MobileApp: This class represents a mobile application. It interacts with the SCC class to order cars or services. It can be seen as an adapter that provides a mobile-friendly interface to the existing system.The MobileApp class interacts with the SCC class to order cars or services. It serves as an adapter to provide a simplified interface for mobile users.Here SCC class is working as a middleware.

The UML diagram for this module is given in the Fig.7.

3 Some snaps of the demo file:

3.1 Demo for 1-3 no point:

```
_____________________________________________________________________________
Private Car Description
        Company Name : Fords
        Automated by : America
Internal Components:
        Engine : 1300CC
        Tire type : Snow
        Chesis : Tabular
External Components:
        AC : Low Powered
        Total Seats : 5

Total cost of the car is: 86
_____________________________________________________________________________
Private Car Description
        Company Name : Fords
        Automated by : America
Internal Components:
        Engine : 1300CC
        Tire type : Snow
        Chesis : Tabular
External Components:
        AC : Low Powered
        Total Seats : 5
customized Decorator: ThickRainShield

Total cost of the car is: 88
_____________________________________________________________________________
```

Figure 8: Demo of module 1-3 (SCC)

3.2 Demo for 4 no point:

```
Hello Ironman, we have an update for you
Some cars price has been changed. Please visit our website to see the updates...


Hello Batman, we have an update for you
Some cars price has been changed. Please visit our website to see the updates...


Hello Spiderman, we have an update for you
Some cars price has been changed. Please visit our website to see the updates...


Hello Batman, we have an update for you
Some cars basic features have been changed. Please visit our website to see the updates...


Hello Spiderman, we have an update for you
Some cars basic features have been changed. Please visit our website to see the updates...
```

Figure 9: Demo of module 4 (Notification)

3.3 Demo for 5-6 no point:

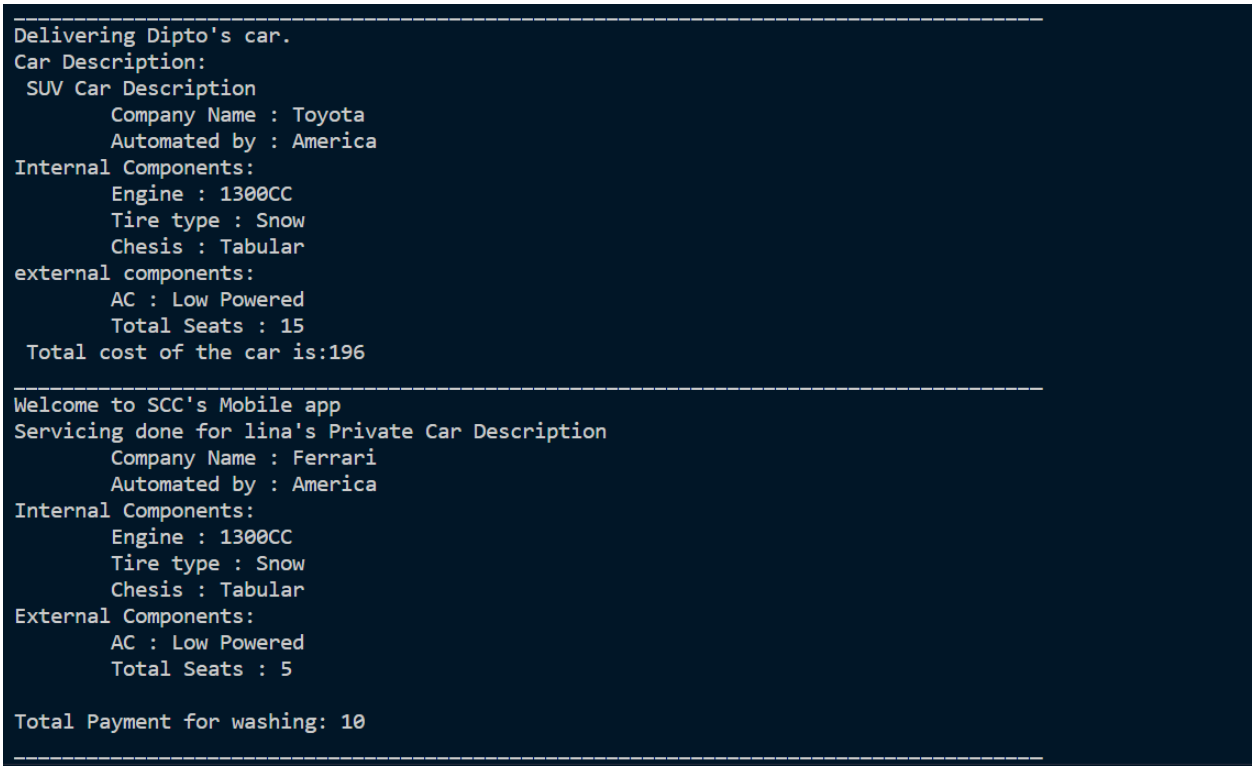

Figure 10: Demo of module 5-6(Central online App)