Abstract Factory Pattern : To Define Car Entities

Pros: Provides a way to create families of related objects, ensures compatibility among components, and allows easy addition of new component types.

Cons: Complexity can increase as the number of components and their variations grow.

Rationale: Abstract Factory Pattern is used to create different car components (Engine, Tire, Chassis, AC, etc.) with the ability to ensure that components of a single family are compatible.

Principle: Depend Upon Abstractions, Not Concrete Classes

Decision: Chose this pattern to manage the creation of interrelated car components while maintaining consistency.

Application: Instantiated different component factories (AsiaCarComponentFactory, USACarComponentFactory) based on geographical locations for creating compatible car components.

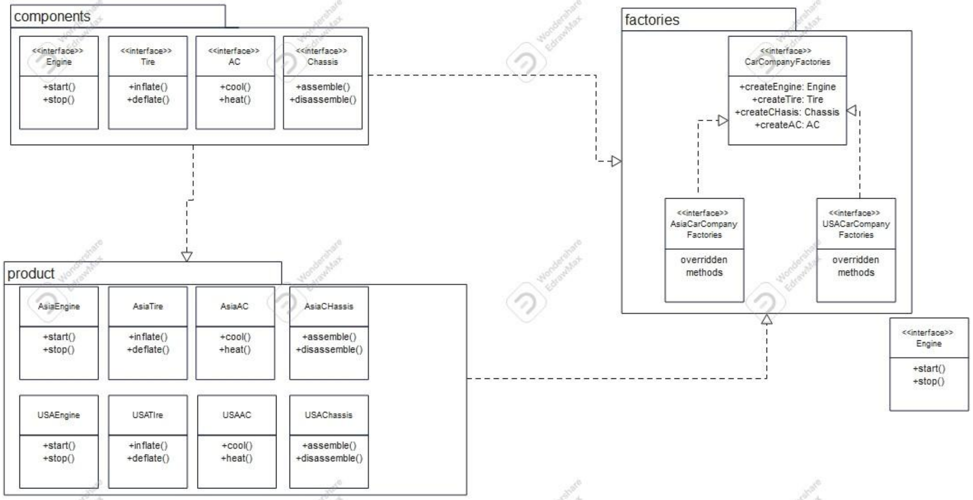

Fig 1: **Abstract Factory Pattern** : To Define Car Entities

Factory Pattern: To Define Car Variants

Pros: Encapsulates object creation, provides flexibility in changing object types, and promotes separation of responsibilities.

Cons: Can lead to increased class hierarchy if not managed properly.

Rationale: Factory Pattern is used to create different car variants (Racing Car, Private Car, SUV, Military Vehicle) for each car group (Ferrari, Ford, Toyota, etc.).

Principle:

- Program to an Interface, Not an Implementation
- Strive for Loosely Coupled Designs
- Classes Should Be Open for Extension but Closed for Modification

Decision: Selected this pattern to manage the creation of car variants while keeping client code less coupled to concrete car variant classes.

Application: Used specific factory classes (FerrariFactory, ToyotaFactory, etc.) to create corresponding car variants.

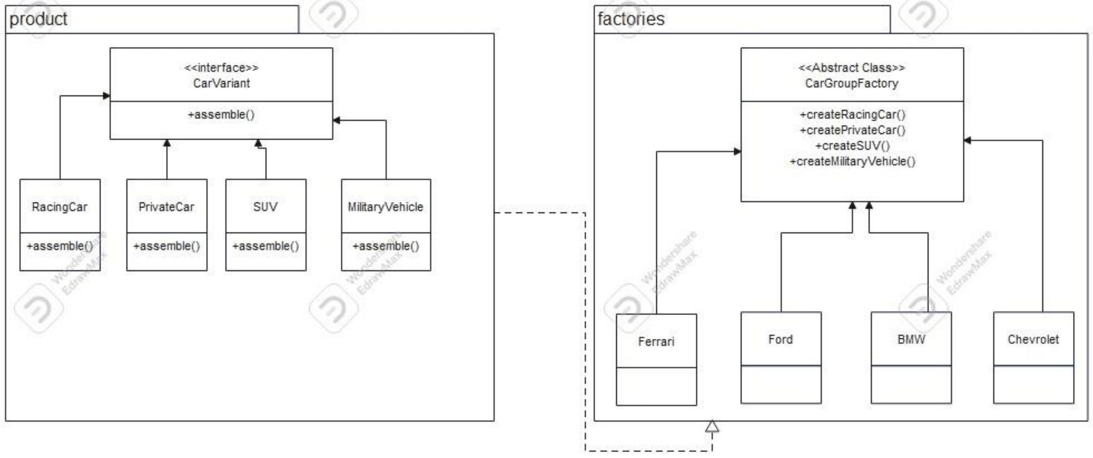

Fig 2: **Factory Pattern:** To Define Car Variants

Decorator Pattern: For Decorations & Customizations

Pros : Allows dynamic addition of responsibilities, promotes open-closed principle, and enables flexible composition of features.

Cons : Can lead to many small classes if overused.

Rationale : Decorator Pattern is used to add customizations like rain shields, bumpers, etc., to cars without modifying the core Car class.

Principle :

- Favor Composition Over Inheritance
- Classes Should Be Open for Extension but Closed for Modification
- Depend Upon Abstractions, Not Concrete Classes

Decision: Chose this pattern to provide clients with the ability to customize cars with various features.

Application: Created decorator classes (CustomizedRainShieldDecorator, BumperDecorator, etc.) to add features to car objects.

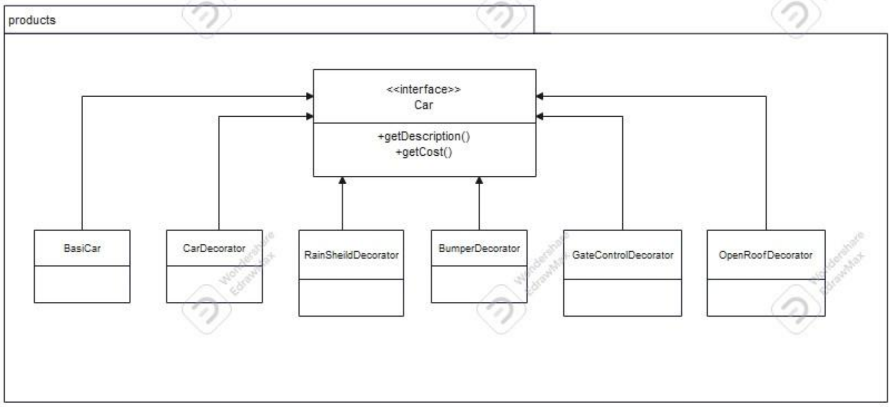

Fig 3: **Decorator Pattern**: For Decorations & Customizations

Observer Pattern: For Notification System

Pros: Supports loose coupling between subjects and observers, enables event-driven behavior, and facilitates multiple updates to observers.

Cons: May lead to performance issues if many observers are present.

Rationale: Observer Pattern is used for the notification system to allow clients to subscribe to Price Change and Feature Change notifications.

Principle:

- Depend Upon Abstractions, Not Concrete Classes
- Strive for Loosely Coupled Designs
- Program to an Interface, Not an Implementation

Decision: Adopted this pattern to provide clients with real-time updates on changes.

Application: Implemented NotificationSubject interface and concrete subject classes for different types of notifications.

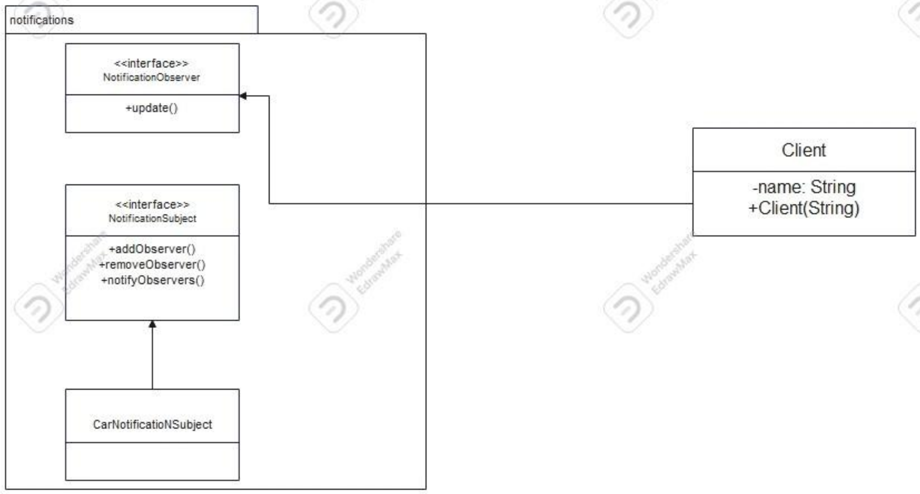

Fig 4: **Observer Pattern**: For Notification System

Command Pattern: For Central Online System

Pros: Encapsulates request details, supports queuing of requests, and allows the implementation of undo functionality.

Cons: Can lead to an increase in the number of command classes.

Rationale: Command Pattern is used to encapsulate requests like servicing, washing, and online delivery for the Central Online System.

Principle:

- Depend Upon Abstractions, Not Concrete Classes
- Strive for Loosely Coupled Designs
- Classes Should Be Open for Extension but Closed for Modification

Decision: Chose this pattern to manage and execute requests in a flexible and decoupled manner.

Application: Created command classes (ServiceCarCommand, WashCarCommand, etc.) to encapsulate request details.

Template Pattern: For Request Processing

Pros: Defines a common process structure, enforces a consistent process flow, and allows customization of specific steps.

Cons: May lead to inflexible designs if not properly designed.

Rationale: Template Pattern is used to define a template for processing different types of requests (approve, perform, etc.).

Principle:

- Identify the Aspects of Your Application That Vary and Separate Them from What Stays the Same
- Strive for Loosely Coupled Designs Between Objects That Interact.
- Program to an Interface, Not an Implementation

Decision: Chose this pattern to provide a standardized process flow for request handling.

Application: Create an abstract class with template methods for different request types.

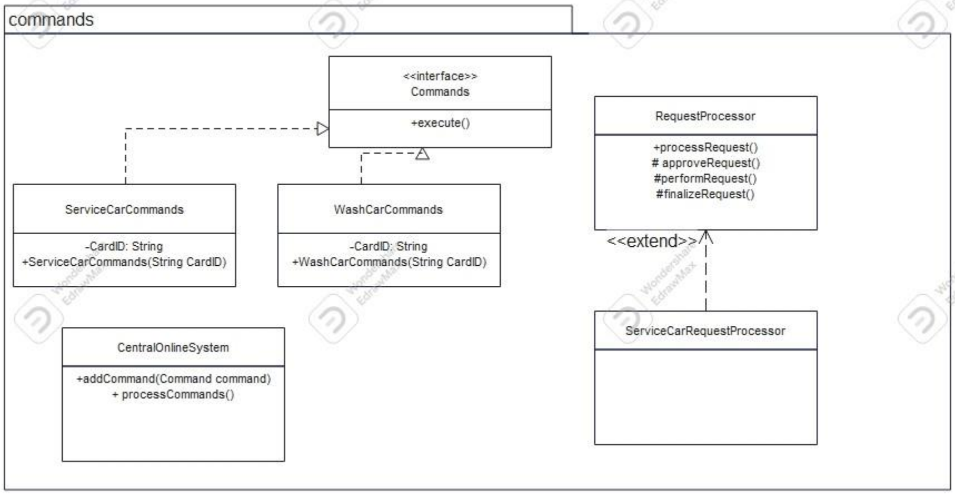

Fig 5: **Command Pattern & Template Pattern**: For Central Online System & Request Processing

Facade and Adapter Patterns: For Web-based & Mobile Applications

***Facade Pattern**:*

Pros: Provides a unified interface, simplifies client interaction, and hides system complexities.

Cons: May become a single point of failure if not designed carefully.

Rationale: Facade Pattern is used to create a simplified interface (MobileAppFacade) for both web-based and mobile applications.

Principle:

- Strive for Loosely Coupled Designs Between Objects That Interact
- Program to an Interface, Not an Implementation
- Depend Upon Abstractions, Not Concrete Classes

Decision: Selected this pattern to ensure ease of use and a consistent interaction experience for clients.

Application: Designed a facade to abstract away the complexities of the system.

Adapter Pattern:

Pros: Allows different interfaces to work together, adapts existing functionality to new requirements, and promotes code reusability.

Cons: Introduces an additional layer that could impact performance if not optimized.

Rationale: Adapter Pattern is used to make the existing web-based system compatible with the new mobile application.

Principle:

- Strive for Loosely Coupled Designs Between Objects That Interact
- Program to an Interface, Not an Implementation
- Depend Upon Abstractions, Not Concrete Classes

Decision: We chose this pattern to ensure seamless integration between the two different interfaces.

Application: Created an adapter to adapt the web-based system's interface to match the mobile application's requirements.

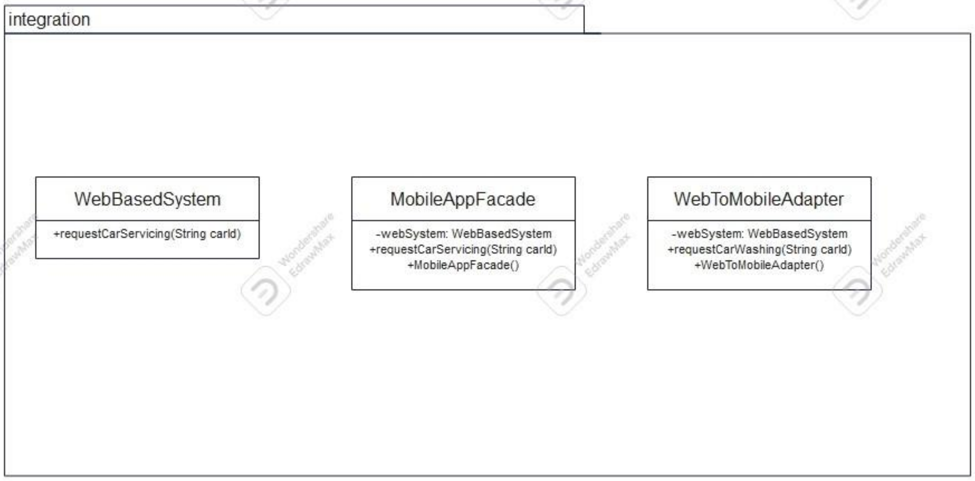
integration
WebBasedSystem
+requestCarServicing(String carId)
MobileAppFacade
-webSystem: WebBasedSystem
+requestCarServicing(String carId)
+MobileAppFacade()
WebToMobileAdapter
-webSystem: WebBasedSystem
+requestCarWashing(String carId)
+WebToMobileAdapter()

Fig 6: **Facade and Adapter Patterns**: For Web-based & Mobile Applications

Adapter Pattern: For Automated AI

Pros: Allows different interfaces (AI systems) to work together, adapts the existing AI functionality to the system's requirements.

Cons: Can introduce complexity in managing adapters and ensuring compatibility.

Rationale: Adapter Pattern is used to incorporate the existing AI systems (with their specific interfaces) into the system.

Principle:

- Strive for Loosely Coupled Designs Between Objects That Interact
- Program to an Interface, Not an Implementation
- Depend Upon Abstractions, Not Concrete Classes

Decision: Chose this pattern to ensure that the AI systems can seamlessly interact with the rest of the system.

Application: Created adapters to bridge the gap between the AI systems' interfaces and the system's requirements.

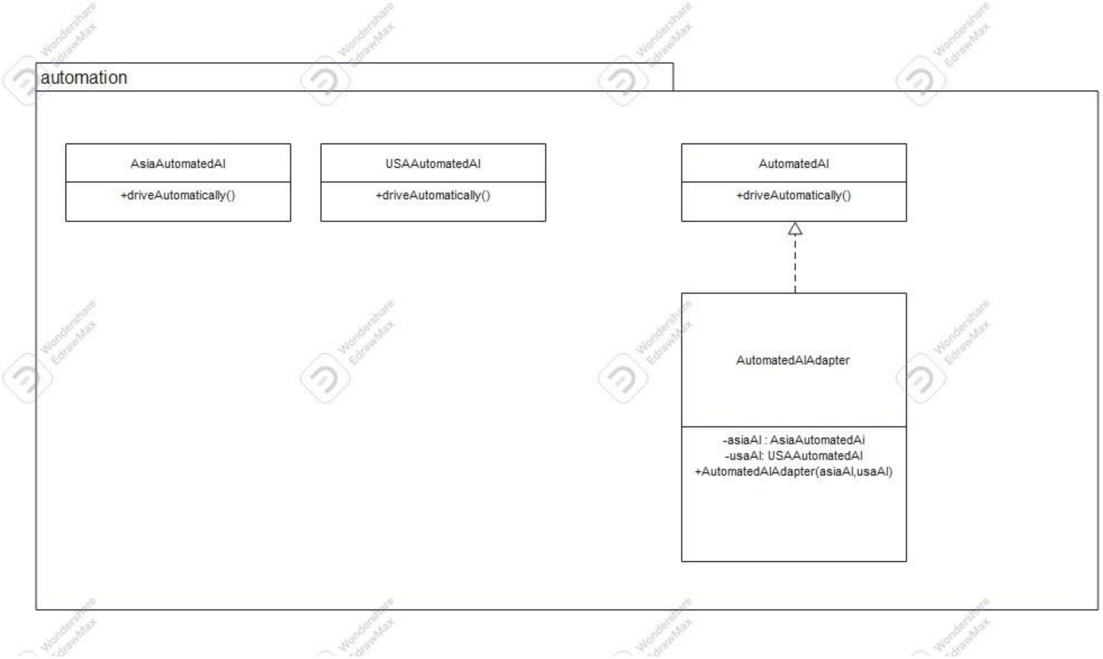

Fig 7: **Adapter Pattern:** For Automated AI

The SCC Package Hierarchy

```
SCC
├── components
│   ├── AC.java
│   ├── Chassis.java
│   ├── Engine.java
│   ├── Tire.java

├── factories
│   ├── AsiaCarComponentFactory.java
│   ├── CarComponentFactory.java
│   ├── CarGroupFactory.java
│   ├── FerrariFactory.java
│   └── FordFactory.java
│  ├── BMWFactory.java
│   └── ChevroletFactory.java
```

```
├── products
│   ├── AsiaAC.java
│   ├── AsiaChassis.java
│   ├── AsiaEngine.java
│   ├── AsiaTire.java
│   ├── CarVariant.java
│   ├── RacingCar.java
│   ├── PrivateCar.java
│   ├── SUV.java
│   ├── MilitaryVehicle.java
│   ├── Car.java
│   ├── BasicCar.java
│   ├── CarDecorator.java
│   ├── RainShieldDecorator.java
│   ├── BumperDecorator.java
│   ├── GateControlDecorator.java
│   ├── OpenRoofDecorator.java
.
├── notifications
│   ├── NotificationType.java
│   ├── NotificationObserver.java
│   ├── NotificationSubject.java
│   ├── CarNotificationSubject.java

├── commands
│   ├── Command.java
│   ├── ServiceCarCommand.java
│   ├── WashCarCommand.java
│   ├── OnlineDeliveryCommand.java
│   ├── CentralOnlineSystem.java
│   ├── RequestProcessor.java
│   ├── ServiceCarRequestProcessor.java

├── integration
│   ├── WebBasedSystem.java
│   ├── MobileAppFacade.java
│   ├── WebToMobileAdapter.java

├── automation
│   ├── AsiaAutomatedAI.java
│   ├── USAAutomatedAI.java
│   ├── AutomatedAI.java
│    ├── AutomatedAIAdapter.java
│
└── Main.java
└── Client.java
```

Link To UML Diagram Folder: URL Diagrams