Design Rationale Document for Special Car Company (SCC) System

1. Singleton Pattern for CarFactory, NotificationSystem, ShopFactory, CarGroupFactory, and MobileMiddleware:

- **Issue/Decision:** The Singleton pattern is used for these classes to ensure that only one instance exists throughout the application lifecycle.
- **Rationale**: The Singleton pattern guarantees that these classes have a single point of access, preventing unnecessary multiple instances from being created. This is suitable for instances that are globally accessible and should have consistent behavior across the application.
- **Principle Application:**
    - Depend Upon Abstractions
    - Strive for Loosely Coupled Designs.
- **Backup UML Diagram:** N/A
- **Pros**: Ensures a single instance, global access, and prevents unnecessary memory usage.
- **Cons**: Can introduce tight coupling if not used carefully.

2. Factory Method Pattern for Car Creation:

- **Issue/Decision:** Factory Method pattern is used to create different types of cars based on their variants.
- **Rationale:** The Factory Method pattern encapsulates object creation in a way that allows subclasses to define the type of objects created. This allows for flexibility and scalability when introducing new car variants.
- **Principle Application**:
    - Program to an Interface, Not an Implementation
    - Favor Composition Over Inheritance.
- **Backup UML Diagram:** N/A
- **Pros:** Provides a clear interface for creating objects, supports extensibility, and follows the Open-Closed Principle.
- **Cons:** Requires the creation of additional classes for each car variant.

3. Strategy Pattern for Automated AI:

- **Issue/Decision:** Strategy pattern is used for handling different AI strategies based on geographical regions.
- **Rationale**: The Strategy pattern allows interchangeable algorithms or behaviors while keeping the code organized and maintainable. It enables flexibility in selecting AI strategies based on the region.
- **Principle Application:**
    - Program to an Interface, Not an Implementation
    - Depend Upon Abstractions.
- **Backup UML Diagram**: N/A
- **Pros**: Promotes encapsulation, facilitates easy swapping of algorithms, and supports different AI strategies for different regions.

- **Cons**: Can lead to increased complexity if there are many strategies.

4. Observer Pattern for Notifications:

- **Issue/Decision**: Observer pattern is used to implement the notification system for notifying clients about changes.
- **Rationale**: The Observer pattern defines a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically.
- **Principle Application:**
  - Program to an Interface, Not an Implementation
  - Depend Upon Abstractions
  - Strive for Loosely Coupled Designs.
- **Backup UML Diagram**: N/A
- **Pros**: Provides a decoupled way to notify clients about changes, supports dynamic addition/removal of observers, and follows the Single Responsibility Principle.
- **Co**ns: Can potentially lead to performance issues if there are too many observers.

5. Decorator Pattern for Customizations:

- **Issue/Decision:** Decorator pattern is used for adding customizations to cars.
- **Rationale:** The Decorator pattern allows dynamic addition of responsibilities to objects, without affecting their structure. It supports flexible customization options without the need for subclass explosion.
- **Principle Application**:
  - Program to an Interface, Not an Implementation
  - Favor Composition Over Inheritance
- **Backup UML Diagram:** N/A
- **Pros:** Provides a way to add functionalities dynamically, follows the Open-Closed Principle, and avoids class proliferation.
- ***Cons***: Can lead to complex class hierarchies if overused.

6. Command Pattern for Online System Requests:

- **Issue/Decision:** Command pattern is used to encapsulate and parameterize requests as objects.
- **Rationale:** The Command pattern decouples sender and receiver of a request, allowing for parameterization of clients with different requests, queuing of requests, and logging of commands.
- **Principle Application**:
  - Program to an Interface, Not an Implementation
  - Depend Upon Abstractions
  - Favor Composition Over Inheritance.
- **Backup UML Diagram:** N/A
- **Pros**: Encapsulates requests as objects, supports undo/redo functionality, and promotes loose coupling between sender and receiver.
- **Cons:** Can result in a large number of command classes if not managed properly.

7. Mobile Middleware using Singleton:
- **Issue/Decision: S**ingleton pattern is used for the MobileMiddleware to ensure a single instance.
- **Rationale**: The Singleton pattern ensures a single instance of the middleware that bridges the gap between the mobile app and the web-based system. This prevents multiple instances causing inconsistencies.
- **Principle Application:**
    - Depend Upon Abstractions
    - Strive for Loosely Coupled Designs
- **Backup UML Diagram**: N/A
- **Pros:** Ensures one instance, provides global access, and prevents multiple instances.
- **Cons:** Can lead to tight coupling and decreased testability if not used carefully.

The design rationale document outlines the design decisions made during the implementation of the SCC system using various design patterns. Each pattern was selected based on its suitability to the specific requirements and design goals. The use of these patterns enhances modularity, flexibility, and maintainability of the system while adhering to key software design principles. Careful consideration of the pros and cons of each pattern helped in achieving an effective and well-structured design for the SCC system.

The design of the system for Special Car Company (SCC) using the specified patterns.

1. Singleton Pattern for CarFactory:
Implement the Singleton pattern for the CarFactory, ensuring that there's only one instance responsible for creating different types of cars.

```
public class CarFactory {
    private static CarFactory instance;

    private CarFactory() {
        // Private constructor to prevent instantiation
    }

    public static CarFactory getInstance() {
        if (instance == null) {
            instance = new CarFactory();
        }
        return instance;
    }
    public Car createCar(CarType type) {
            switch (type) {
                case RACING:
                    return new RacingCar();
                case PRIVATE:
                    return new PrivateCar();
                case SUV:
                    return new SUVCar();
                case MILITARY:
                    return new MilitaryCar();
                default:
                    throw new IllegalArgumentException("Invalid car type: " + type);
            }
   }

}
```

2. Factory Method Pattern for Car Creation:
Utilize the Factory Method pattern within the CarFactory to create different types of cars (Racing, Private, SUV, Military). Each car variant creation is encapsulated in a separate factory method.

```
// Abstract base class for different types of cars
public abstract class Car {
    private String type;

    public Car(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }

    // Define common methods and attributes for all types of cars
    // You can add more methods and attributes here
}

// Concrete subclass of Car
class RacingCar extends Car {
    public RacingCar() {
        super("Racing Car");
        // Implement specific behavior and attributes for a racing car
    }

    // Additional methods and attributes for RacingCar
}

// Concrete subclass of Car
class PrivateCar extends Car {
    public PrivateCar() {
        super("Private Car");
        // Implement specific behavior and attributes for a private car
    }

    // Additional methods and attributes for PrivateCar
}

// Concrete subclass of Car
class SUV extends Car {
    public SUV() {
        super("SUV");
        // Implement specific behavior and attributes for an SUV
    }

    // Additional methods and attributes for SUV
}

// Concrete subclass of Car
class MilitaryVehicle extends Car {
    public MilitaryVehicle() {
        super("Military Vehicle");
        // Implement specific behavior and attributes for a military vehicle
    }

    // Additional methods and attributes for MilitaryVehicle
}

// Abstract factory class for creating cars
public abstract class CarFactory {
    public abstract Car createCar();
}

// Concrete factory subclass for creating RacingCars
```

```
class RacingCarFactory extends CarFactory {
    public Car createCar() {
        return new RacingCar();
    }
}

// Concrete factory subclass for creating PrivateCars
class PrivateCarFactory extends CarFactory {
    public Car createCar() {
        return new PrivateCar();
    }
}

// Concrete factory subclass for creating SUVs
class SUVFactory extends CarFactory {
    public Car createCar() {
        return new SUV();
    }
}

// Concrete factory subclass for creating MilitaryVehicles
class MilitaryVehicleFactory extends CarFactory {
    public Car createCar() {
        return new MilitaryVehicle();
    }
}
```

3. Strategy Pattern for Automated AI:
Apply the Strategy pattern for the AutomatedAI system. Create a strategy interface for AI and implement different AI strategies for Asia and USA. Cars can then have a reference to an AI strategy.

```
// Interface for Automated AI
public interface AutomatedAI {
    void drive();
}

// Implementation for Asia-based Automated AI
public class AsiaBasedAutomatedAI implements AutomatedAI {
    public void drive() {
        // Dummy logic for AI driving in Asia
        System.out.println("Automated AI is driving in Asia.");
    }
}

// Implementation for USA-based Automated AI
public class USABasedAutomatedAI implements AutomatedAI {
    public void drive() {
        // Dummy logic for AI driving in the USA
        System.out.println("Automated AI is driving in the USA.");
    }
}
```

4. Observer Pattern for Notifications:
Use the Observer pattern for the notification system. Create a subject class that allows clients to subscribe and unsubscribe. Notify all subscribed clients when there are price changes or feature updates.

```
// Observer interface for receiving notifications
public interface Observer {
    void update(String message);
}

// Subject interface for managing observers and sending notifications
public interface Subject {
    void subscribe(Observer observer);
    void unsubscribe(Observer observer);
    void notifyObservers(String message);
}

// Concrete implementation of the Observer interface (Client)
public class Client implements Observer {
    private String name;

    public Client(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received a notification: " + message);
    }
}

// Concrete implementation of the Subject interface (NotificationSystem)
public class NotificationSystem implements Subject {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer observer) {
        observers.add(observer);
        System.out.println("Observer " + observer + " subscribed.");
    }

    public void unsubscribe(Observer observer) {
        observers.remove(observer);
        System.out.println("Observer " + observer + " unsubscribed.");
    }

    public void notifyObservers(String message) {
        System.out.println("Sending notification to all observers:");
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

5. Decorator Pattern for Customizations:
Implement the Decorator pattern for customizing cars. Define a base Car class and create decorator classes for customizations like rain shields, bumpers, gate controlling systems, and open roof systems.

```
// Interface for Car
public interface Car {
    double getPrice();
}

// Concrete base car class
public class BaseCar implements Car {
    public double getPrice() {
        // Return base car price
        return 20000.0; // Placeholder value
    }
```

```
}

// Abstract decorator class
public abstract class CarDecorator implements Car {
    protected Car decoratedCar;

    public CarDecorator(Car car) {
        this.decoratedCar = car;
    }

    public abstract double getPrice(); // Override in concrete decorators
}

// Concrete decorator class for CustomizedRainShield
public class CustomizedRainShieldDecorator extends CarDecorator {
    public CustomizedRainShieldDecorator(Car car) {
        super(car);
    }

    public double getPrice() {
        // Calculate price with customized rain shield
        return decoratedCar.getPrice() + 500.0; // Placeholder value
    }
}

// Concrete decorator class for Bumper
public class BumperDecorator extends CarDecorator {
    public BumperDecorator(Car car) {
        super(car);
    }

    public double getPrice() {
        // Calculate price with customized bumper
        return decoratedCar.getPrice() + 300.0; // Placeholder value
    }
}

// Concrete decorator class for Gate Controlling System
public class GateControllingSystemDecorator extends CarDecorator {
    public GateControllingSystemDecorator(Car car) {
        super(car);
    }

    public double getPrice() {
        // Calculate price with gate controlling system
        return decoratedCar.getPrice() + 1000.0; // Placeholder value
    }
}

// Concrete decorator class for Open Roof System
public class OpenRoofSystemDecorator extends CarDecorator {
    public OpenRoofSystemDecorator(Car car) {
        super(car);
    }

    public double getPrice() {
        // Calculate price with open roof system
        return decoratedCar.getPrice() + 800.0; // Placeholder value
    }
}
```

6. Command Pattern for Online System:
Apply the Command pattern for the online system's requests. Define command objects for car servicing, washing, and online delivery. Each command encapsulates the request and can be executed by a receiver.

```
// Interface for Command
public interface Command {
    void execute();
}

// Concrete command class for car servicing
public class CarServicingCommand implements Command {
    public void execute() {
        // Logic for car servicing
        System.out.println("Car servicing command executed.");
    }
}

// Concrete command class for car washing
public class CarWashingCommand implements Command {
    public void execute() {
        // Logic for car washing
        System.out.println("Car washing command executed.");
    }
}

// Concrete command class for online delivery
public class OnlineDeliveryCommand implements Command {
    public void execute() {
        // Logic for online delivery
        System.out.println("Online delivery command executed.");
    }
}
```

7. Singleton Pattern for NotificationSystem:
Implement the Singleton pattern for the NotificationSystem, ensuring there's only one instance responsible for managing notifications.

```
public class NotificationSystem {
    private static NotificationSystem instance;

    private NotificationSystem() {
        // Private constructor to prevent instantiation
    }

    public static NotificationSystem getInstance() {
        if (instance == null) {
            instance = new NotificationSystem();
        }
        return instance;
    }

    // Other methods for notification management
}
```

8. Singleton Pattern for OnlineSystem:
Use the Singleton pattern for the OnlineSystem to ensure a single instance handling online requests.

```
public class OnlineSystem {
    private static OnlineSystem instance;
    private OnlineSystem() {
        // Private constructor to prevent instantiation
    }

    public static OnlineSystem getInstance() {
        if (instance == null) {
            instance = new OnlineSystem();
        }
        return instance;
    }

    // Other methods for online requests handling
}
```

9. Singleton Pattern for MobileMiddleware:
Implement the Singleton pattern for the MobileMiddleware, providing a single instance for bridging the gap between the mobile app and the web-based system.

```
public class MobileMiddleware {
    private static MobileMiddleware instance;

    private MobileMiddleware() {
        // Private constructor to prevent instantiation
    }

    public static MobileMiddleware getInstance() {
        if (instance == null) {
            instance = new MobileMiddleware();
        }
        return instance;
    }

    // Other methods for mobile app interaction with the web-based system
}
```

10. Singleton Pattern for ShopFactory:
Apply the Singleton pattern for the ShopFactory, ensuring there's only one instance responsible for creating different types of shops.

```
public class ShopFactory {
    private static ShopFactory instance;
    private ShopFactory() {
        // Private constructor to prevent instantiation
    }

    public static ShopFactory getInstance() {
        if (instance == null) {
            instance = new ShopFactory();
        }
        return instance;
    }

    public Shop createShop(ShopType type) {
            switch (type) {
                case ASIA_CENTRIC:
                    return new AsiaCentricShop();
                case USA_CENTRIC:
                    return new USACentricShop();
                default:
                    throw new IllegalArgumentException("Invalid shop type: " + type);
            }
    }

}
```

11. Singleton Pattern for CarGroupFactory:
Implement the Singleton pattern for the CarGroupFactory, providing a single instance responsible for creating different car groups.

```
public class CarGroupFactory {
    private static CarGroupFactory instance;

    private CarGroupFactory() {
        // Private constructor to prevent instantiation
    }

    public static CarGroupFactory getInstance() {
        if (instance == null) {
            instance = new CarGroupFactory();
        }
        return instance;
    }
    public CarGroup createCarGroup(CarGroupType type) {
            switch (type) {
              case FERRARI:
                return new FerrariCarGroup();
              case FORD:
                return new FordCarGroup();
              case TOYOTA:
                return new ToyotaCarGroup();
              // Add cases for other car group types
              default:
                throw new IllegalArgumentException("Invalid car group type: " + type);
            }
   }


}
```

//double check the requirements? ;-;

DESIGN PRINCIPLES in ppt

1. identify the aspects of your application that vary and separate them from what stays the same
2. program to an interface not an implementation
3. favor composition over inheritance
4. strive for loosely coupled designs between objects that interact
5. classes should be open for extension but closed for modification
6. depend upon abstractions, do not depend upon concrete classes

DESIGN PATTERNS for scc:

- **singleton**
- **command**
- **strategy**
- **observer**
- **factory & abstract factory**
- **decorator**

//add the further requirements sir mentioned in the class ;-;