**1. Singleton Pattern**

* **Applicable to:** The Garage class.
* **Reasoning:** In most car rental systems, there is only one "main" garage or a central repository for all cars. The Singleton pattern ensures that there is only ever one instance of the Garage class, which provides a single, global point of access to the car inventory and contracts. This prevents issues with multiple, uncoordinated garage objects.

**2. Strategy Pattern**

* **Applicable to:** The LeaseContract class and a potential PriceCalculation service.
* **Reasoning:** The way the totalPrice is calculated could change based on different strategies (e.g., standard daily rate, weekend rate, long-term rental discount, promotional rate). Instead of using a large if/else block, you could use the Strategy pattern.
  + **Context:** LeaseContract
  + **Strategy Interface:** PricingStrategy (with a method like calculatePrice(duration, dailyRate))
  + **Concrete Strategies:** StandardPricingStrategy, WeekendPricingStrategy, PromotionalPricingStrategy
  + This makes it easy to add new pricing rules without modifying the LeaseContract class itself.

**3. Factory Method Pattern**

* **Applicable to:** The process of creating Car objects.
* **Reasoning:** The creation of Car objects might involve different steps based on their CarCategory (e.g., a "TRUCK" might have different initialization steps than a "SEDAN").
  + **Creator Interface:** CarFactory (with a createCar() method)
  + **Concrete Creators:** SedanFactory, TruckFactory, SuvFactory
  + This pattern delegates the creation of objects to subclasses, making the system more flexible if different car types have specialized construction logic.

**4. Observer Pattern**

* **Applicable to:** Not explicitly shown, but could be used to notify other parts of the system when a contract's status changes.
* **Reasoning:** When the status of a LeaseContract changes (e.g., from "active" to "completed"), other parts of the system might need to be updated. For example:
  + The Garage object might need to mark the car as "available" again.
  + The user's profile might need to update a list of "current rentals."
  + The Observer pattern allows these objects to "subscribe" to changes in the LeaseContract status, so they are automatically notified and can react accordingly.

**5. Facade Pattern**

* **Applicable to:** The process of creating a LeaseContract.
* **Reasoning:** The user-facing action of "renting a car" involves several sub-tasks: checking the user's age, verifying car availability, creating the contract, and calculating the price. A Facade class could provide a simple, unified interface to this complex subsystem.
  + A class like RentalServiceFacade could have a single public method rentCar(user, car, startDate, endDate) that orchestrates all these internal steps. This hides the complexity and provides a cleaner API for the user interface.