**Abstract Factory:** The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. In your context, you could have a ‘VehicleFactory’ interface with a method like ‘createVehicle()’. Then, for each specific type of vehicle (like Car, Truck, or Motorcycle), you would have a concrete factory implementing ‘VehicleFactory’. Each concrete factory would create its specific type of vehicle.

interface VehicleFactory {

Vehicle createVehicle();

}

class CarFactory implements VehicleFactory {

@Override

public Vehicle createVehicle() {

return new Car();

}

}

**Decorator:** The Decorator pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. In this case, you could create decorators like ‘VehicleColorDecorator’ and ‘VehicleStatusDecorator’ that would add color and status to a Vehicle, respectively.

abstract class VehicleDecorator extends Vehicle {

protected Vehicle decoratedVehicle;

public VehicleDecorator(Vehicle decoratedVehicle) {

this.decoratedVehicle = decoratedVehicle;

}

}

class VehicleColorDecorator extends VehicleDecorator {

private String color;

public VehicleColorDecorator(Vehicle decoratedVehicle, String color) {

super(decoratedVehicle);

this.color = color;

}

@Override

public String getColor() {

return this.color;

}

// Other methods...

}

**Observer:** The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In your case, the ‘Vehicle’ objects could be the subjects. When a ‘Vehicle's’ status changes, it notifies all registered observers, which could be GUI components.

class Vehicle {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(this);

}

}

}

**Singleton:** The Singleton Pattern ensures a class only has one instance and provides a global point of access to it. To create a single instance of your ‘MainScreen’, you would provide a public method to get the instance of the class and prevent any other instantiation by making the constructor private.

class MainScreen {

private static MainScreen instance;

private MainScreen() {}

public static MainScreen getInstance() {

if (instance == null) {

instance = new MainScreen();

}

return instance;

}

}

**Memento:** The Memento Pattern provides the ability to restore an object to its previous state. create a ‘Memento’ class that encapsulates the state of a ‘Vehicle’ and a ‘Caretaker’ class that can hold and manage up to three Memento objects.

class Memento {

private VehicleState state;

public Memento(VehicleState state) {

this.state = state;

}

public VehicleState getState() {

return state;

}

}

**Thread Pool**: Thread Pool Pattern is where a number of threads are created to perform a number of tasks, which are usually organized in a queue. The number of threads can be managed by the system, usually set at runtime. In this context, you would create a thread pool of seven threads, each representing an attendant for a test drive. These threads would then be allocated to tasks as requests for test drives come in.

ExecutorService executor = Executors.newFixedThreadPool(7);

for (TestDriveRequest request : requests) {

executor.execute(new TestDriveTask(request));

}