Answer sheet : 1

Question 1. Explain the Singleton Pattern. What are its key characteristics, and in which scenarios is it most appropriately used?

Answer:

Singleton Pattern Overview:

The Singleton Pattern is a design pattern that ensures a class has only one instance throughout the lifecycle of an application. This single instance is globally accessible, meaning that any part of the program can access this instance. It is particularly useful when exactly one object is needed to coordinate actions across the system.

Key Characteristics:

1. Single Instance:
   - The primary goal of the Singleton Pattern is to ensure that a class has only one instance. This is achieved by controlling the instantiation process and providing a global point of access to the instance.

2. Global Access:
   - The Singleton instance is accessible globally, meaning that any part of the application can access this single instance using a static method (usually getInstance()).

3. Lazy Initialization (Optional):
   - The instance is created only when it is needed (the first time it is requested). This is known as lazy initialization. It can save resources if the instance is not needed right away.

4. Thread Safety (Optional):
   - In multi-threaded environments, special care must be taken to ensure that the Singleton instance is created only once, even if multiple threads try to access it simultaneously. This can be achieved using synchronization or other thread-safety mechanisms.

5. Private Constructor:

- The constructor of the Singleton class is made private to prevent direct instantiation from outside the class. This ensures that the only way to get the instance is through the provided static method.

Example Implementation in Java:

```
public class Singleton {
    private static Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Thread-Safe Singleton Example:

In a multi-threaded environment, you might need to ensure that only one thread can create the instance at a time:

```
public class Singleton {
    private static Singleton instance = null;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Double-Checked Locking for Performance Optimization:

To reduce the overhead of synchronization, you can use double-checked locking:

```
public class Singleton {
    private static volatile Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Scenarios Where Singleton Pattern is Most Appropriate:

1. Configuration Management:
   - When an application needs a centralized configuration manager that should be accessed by various parts of the application. The Singleton ensures that all parts of the application use the same configuration instance.

2. Resource Management:
   - For managing shared resources such as connection pools, logging services, or caches. Singleton ensures that only one instance manages these resources, preventing issues like inconsistent states or resource leaks.

3. Logging:
   - Often used for logging purposes where having a single logger instance throughout the application helps maintain a consistent logging strategy.

4. Access Control:
   - In scenarios where access to a resource or service must be controlled or restricted to a single instance, such as license management or feature toggling in a software application.

5. Application State:
   - To maintain a global state or a central registry of objects, where multiple parts of the application need to share and access this state.

Drawbacks and Considerations:

1. Global State:
   - Singletons introduce a global state into the application, which can make testing and debugging more difficult. Overuse of Singletons can lead to code that is tightly coupled and hard to maintain.

2. Difficulty in Testing:
   - Because Singletons are globally accessible and their state persists, they can complicate unit testing. Test cases might interfere with each other if they share a Singleton instance.

3. Concurrency Issues:
   - If not implemented carefully, Singletons can lead to issues in multi-threaded environments, such as race conditions.

4. Inflexibility:
   - Singletons can be less flexible compared to other patterns because they restrict instantiation and enforce a single point of access. This can make it difficult to extend or modify the Singleton class.

Conclusion:

The Singleton Pattern is a powerful tool in situations where a single instance of a class is necessary for coordinating actions across an application. However, it should be used judiciously, with careful consideration of potential drawbacks like global state and testing challenges. Proper implementation, especially in multi-threaded environments, is crucial to avoid common pitfalls.

Question 2. Compare and contrast the Adapter Pattern and the Facade Pattern. Provide an example scenario where each would be used.

Answer:

The Adapter Pattern and Facade Pattern are both structural design patterns, but they serve different purposes.

Adapter Pattern:

The Adapter Pattern is used to convert the interface of a class into one that the client expects, allowing incompatible interfaces to work together. It acts as a bridge between two objects. For instance, if you have a third-party library with a different interface than your system expects, you can use an adapter to integrate it.

Example: Suppose you're developing a payment system that expects a PaymentInterface. You want to integrate a third-party payment gateway with a different interface. The Adapter would implement PaymentInterface and internally translate the calls to the third-party system's API.

Facade Pattern:

The Facade Pattern is used to simplify interactions with complex subsystems by providing a unified, easy-to-use interface. It's mainly for reducing the complexity of interacting with a subsystem that has multiple interfaces or classes.

Example: In a home automation system, a HomeFacade could be designed to manage subsystems like Lighting, Security, and TemperatureControl. Instead of interacting with each subsystem separately, a single HomeFacade object simplifies operations, like turning everything off when leaving home.

In summary, the Adapter focuses on compatibility between interfaces, while the Facade focuses on simplifying complex systems.

Question 3. Describe the Observer Pattern. How does it support decoupling in an object-oriented system? Give an example of its implementation.

Answer:

The Observer Pattern is a behavioral design pattern in object-oriented programming that defines a one-to-many dependency between objects. In this pattern, observers (or listeners) are automatically notified and updated when the subject (or observable) they are monitoring changes state.

Decoupling Support:
The Observer Pattern supports decoupling by promoting loose coupling between the subject and its observers. The subject does not need to know the concrete class or implementation details of the observers; it only knows that they implement a common interface. This allows the subject and observers to vary independently, making the system more flexible and easier to extend or modify.

Example:
Consider a news feed system where users (observers) subscribe to specific topics (subject). When new articles are published, the news feed (subject) notifies all its subscribers (observers) of the update.

java
Copy code

```
// Subject (Observable)
class NewsFeed {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String article) {
        for (Observer o : observers) {
            o.update(article);
        }
    }

    public void publish(String article) {
        notifyObservers(article);
    }
```

```
}

// Observer interface
interface Observer {
    void update(String article);
}

// Concrete Observer
class User implements Observer {
    private String name;

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

    public void update(String article) {
        System.out.println(name + " received article: " + article);
    }
}
```

In this implementation, the NewsFeed (subject) notifies all User objects (observers) when a new article is published, without tightly coupling them to each other.

Question 4. What is the difference between the Factory Method Pattern and the Abstract Factory Pattern? Provide a use case for each.

Answer:

The Factory Method Pattern and the Abstract Factory Pattern are both creational design patterns, but they differ in scope and purpose.

1. Factory Method Pattern:

The Factory Method Pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. This pattern is used when the exact type of object to be created isn't known until runtime, and it delegates the instantiation responsibility to subclasses.

Use Case: If you have an application that generates different types of reports (e.g., PDFReport, ExcelReport), a factory method can be used to instantiate the appropriate report type depending on user input.

java
Copy code

```
abstract class ReportFactory {
    abstract Report createReport();
}

class PDFReportFactory extends ReportFactory {
    Report createReport() {
        return new PDFReport();
    }
}

class ExcelReportFactory extends ReportFactory {
    Report createReport() {
        return new ExcelReport();
    }
}
```

2. Abstract Factory Pattern:

The Abstract Factory Pattern is used to create families of related or dependent objects without specifying their concrete classes. It involves creating a factory that can return different types of objects, all of which share some common theme or characteristics.

Use Case: In a GUI toolkit, you may want to create themed widgets (buttons, text boxes, etc.). Using an abstract factory, you can create a MacWidgetFactory or WindowsWidgetFactory, where each factory produces consistent components for the respective operating system theme.

java
Copy code

```
interface WidgetFactory {
    Button createButton();
```

```
    TextBox createTextBox();
}

class MacWidgetFactory implements WidgetFactory {
    public Button createButton() {
        return new MacButton();
    }

    public TextBox createTextBox() {
        return new MacTextBox();
    }
}

class WindowsWidgetFactory implements WidgetFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public TextBox createTextBox() {
        return new WindowsTextBox();
    }
}
```

Key Differences:

Factory Method is focused on delegating the instantiation of a single product class, allowing subclasses to choose the concrete type.

Abstract Factory is used to create families of related objects (e.g., buttons, textboxes) without specifying the concrete classes.

Question 5. How does the Strategy Pattern enable flexible design? Illustrate with an example where it can replace conditional logic.

Answer:

The Strategy Pattern enables flexible design by allowing an algorithm's behavior to be selected at runtime, without altering the client code. Instead of using conditionals (if-else or switch statements), the Strategy Pattern encapsulates each algorithm in its own class and lets the client choose the desired algorithm dynamically.

How It Works:

Context: The object that uses the strategy (or algorithm).

Strategy Interface: Defines the common interface for all supported algorithms.

Concrete Strategies: Different implementations of the strategy interface, representing different algorithms.

Benefits:

Eliminates Conditional Logic: Instead of hardcoding various conditions for different behaviors, the Strategy Pattern allows for selecting behavior dynamically, leading to more maintainable and extendable code.

Open/Closed Principle: New strategies can be added without modifying the existing code, thus adhering to the Open/Closed Principle.

Example Scenario: Replacing Conditional Logic

Consider a scenario where a shopping cart offers multiple payment methods like Credit Card, PayPal, and Bitcoin. Without the Strategy Pattern, you'd likely implement this with conditionals:

java

Copy code

```
if (paymentType.equals("CreditCard")) {
    processCreditCard();
} else if (paymentType.equals("PayPal")) {
    processPayPal();
} else if (paymentType.equals("Bitcoin")) {
    processBitcoin();
}
```

Using the Strategy Pattern, you can eliminate this conditional logic by encapsulating each payment method into its own class:

java

Copy code

```
// Strategy Interface
interface PaymentStrategy {
    void pay(int amount);
}
```

```
// Concrete Strategy: Credit Card
class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

// Concrete Strategy: PayPal
class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}

// Context: Shopping Cart
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

Usage:

Now, instead of using conditionals to determine the payment method, the client simply selects the strategy:

java

Copy code

```
ShoppingCart cart = new ShoppingCart();
cart.setPaymentStrategy(new PayPalPayment());  // Use PayPal
```

cart.checkout(100);
This approach increases flexibility and simplifies code maintenance, as adding new payment methods only requires creating a new strategy without modifying the existing logic.

Question 6. Discuss the Command Pattern. How does it encapsulate a request as an object, and what are the benefits of this encapsulation?

Answer:
The Command Pattern is a behavioral design pattern that encapsulates a request or action as an object, allowing for flexible request handling, queuing, logging, and undo operations. By encapsulating a request in a command object, you can decouple the client (which invokes the request) from the receiver (which processes the request), promoting more flexible and maintainable code.

How the Command Pattern Works:
Command Interface: Defines a method (often execute()) that encapsulates the action to be performed.

Concrete Command: Implements the command interface and binds an action (or request) with a receiver. It calls the appropriate method(s) on the receiver.
Receiver: The object that knows how to perform the actual operation (e.g., turn on a light, process an order).
Invoker: Triggers the command (e.g., button press), but does not need to know details of the command execution.
Client: The client creates concrete command objects and sets up the invoker with these commands.

Example:
Let’s say you’re building a home automation system where a button on a remote control can turn on or off different devices, such as a light or fan.

java
Copy code
```
// Command Interface
interface Command {
    void execute();
```

```
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}

// Concrete Command for turning on the light
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

// Invoker (e.g., Remote control)
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
```

```
        command.execute();
    }
}
```

In this example, the LightOnCommand encapsulates the request to turn on the light. The RemoteControl (invoker) only needs to call the execute() method, without knowing the details of how the light is turned on.

Benefits of Encapsulation:
Decoupling: The client that initiates the request doesn't need to know about the receiver or how the request will be fulfilled, enabling greater flexibility and code reuse.

Flexibility: Commands can be queued, logged, or scheduled. Since commands are objects, they can be stored for later execution (e.g., logging for auditing or undo operations).

Undo/Redo functionality: By encapsulating operations as objects, the system can easily provide undo/redo functionality. A command can implement an undo() method that reverses its action.

Macro Commands: You can compose multiple commands into a single command to execute multiple actions together.

Use Cases:
GUI buttons: Each button click invokes a command that encapsulates the button's functionality (e.g., "Save", "Print").
Task scheduling: Commands can be queued or scheduled for execution at a later time.
Undo/Redo: Text editors or graphical applications often use the Command Pattern to implement undo/redo functionality.
In summary, the Command Pattern encapsulates requests as objects, allowing for flexibility, decoupling, and features like undo and task scheduling, making it ideal for scenarios that involve complex sequences of actions or undoable operations.

Question 7. Explain the Decorator Pattern. How can it be used to extend the functionalities of objects dynamically? Provide a code example.

Answer:

The Decorator Pattern is a structural design pattern used to dynamically extend the functionality of objects without modifying their original class. It wraps the original object in another object, which "decorates" it by adding additional behavior, either before or after delegating to the original object.

How It Works:
Component Interface: Defines the common interface for both the original object and its decorators.

Concrete Component: The object whose behavior will be extended.
Decorator Class: Implements the same interface as the component and holds a reference to a component object. It forwards requests to the component but may add extra behavior either before or after forwarding.
Concrete Decorators: Extend the functionality of the component dynamically by adding their own behavior around the wrapped component.
Benefits:
Flexible extension: Decorators allow adding functionality without modifying the original class, adhering to the Open/Closed Principle.
Composability: Multiple decorators can be applied to an object, allowing combinations of behaviors.

Runtime behavior extension: Since decorators can be applied dynamically, behaviors can be added or changed at runtime, unlike inheritance which is static.
Example Scenario:
Let’s say we have a simple coffee ordering system where we want to extend the base coffee with different add-ons like milk or sugar.

java
Copy code

```
// Component Interface
interface Coffee {
    String getDescription();
    double getCost();
}

// Concrete Component (Basic Coffee)
class SimpleCoffee implements Coffee {
```

```
    public String getDescription() {
        return "Simple Coffee";
    }

    public double getCost() {
        return 5.0;
    }
}

// Base Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public String getDescription() {
        return coffee.getDescription();
    }

    public double getCost() {
        return coffee.getCost();
    }
}

// Concrete Decorator 1: Adding Milk
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return super.getDescription() + ", Milk";
    }
```

```
    public double getCost() {
        return super.getCost() + 1.5; // Milk costs extra
    }
}

// Concrete Decorator 2: Adding Sugar
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return super.getDescription() + ", Sugar";
    }

    public double getCost() {
        return super.getCost() + 0.5; // Sugar costs extra
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        // Add milk
        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        // Add sugar
        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());
    }
```

}
Output:
bash
Copy code
Simple Coffee $5.0
Simple Coffee, Milk $6.5
Simple Coffee, Milk, Sugar $7.0
Explanation:
SimpleCoffee is the base component that provides the default coffee.
The MilkDecorator and SugarDecorator are decorators that extend the functionality by adding milk and sugar, respectively.
The CoffeeDecorator abstract class provides a base for concrete decorators by forwarding the basic behavior of the wrapped component while allowing additional behavior to be added.
When to Use:
When you want to add responsibilities to objects dynamically, without altering the structure of the existing class.
When multiple behaviors or enhancements need to be composed together, and inheritance would lead to a large and rigid hierarchy of subclasses.
The Decorator Pattern provides a clean, flexible way to extend the functionality of objects at runtime while keeping the original class intact.

Question 8. What are the main differences between the Proxy Pattern and the Decorator Pattern? In which scenarios would you choose one over the other?

Answer:
The Proxy Pattern and the Decorator Pattern are both structural design patterns, but they serve different purposes and are used in different scenarios. Here's a breakdown of their main differences and typical use cases:

Proxy Pattern
Purpose:

The Proxy Pattern is used to provide a surrogate or placeholder for another object to control access to it. Proxies often manage access to the real object, which may be expensive to create or require protection.

Types of Proxies:

Virtual Proxy: Controls access to a resource that is expensive to create. It creates the real object only when it is needed.
Remote Proxy: Represents an object that is in a different address space, typically used in distributed systems to allow communication between different processes.
Protective Proxy: Controls access to an object by enforcing access rules or providing security.

Scenario Examples:

Virtual Proxy: If you have an image-loading application, a virtual proxy can be used to delay the loading of the actual image until it is displayed.
Remote Proxy: In a distributed application, a remote proxy could handle communication between a client and a server object.
Protective Proxy: In a financial application, a protective proxy could ensure that only authorized users can perform certain transactions.
Decorator Pattern
Purpose:

The Decorator Pattern is used to dynamically add or enhance functionalities of an object without altering its structure. It allows for extending the behavior of objects in a flexible and reusable manner.
How It Works:

Decorators are used to wrap a core object (which implements a certain interface) and add additional behavior. Each decorator implements the same interface as the core object and delegates to it while adding its own behavior.
Scenario Examples:

Enhancing Features: If you have a graphical user interface (GUI) application, you could use decorators to add features like borders, scrollbars, or additional styles to UI components like windows or buttons.
Adding Functionality: In a text processing application, decorators can be used to add features like spell checking, formatting, or encryption to the base text object.
Choosing Between Proxy and Decorator

Use Proxy When:

You need to control access to an object.
You need to manage the creation of an object due to performance considerations.
You want to provide access control or security features.
Use Decorator When:

You want to add or change functionality of an object at runtime.
You need a flexible way to combine different behaviors or responsibilities.
You want to avoid subclassing to extend functionality.
In summary, if you need to control or manage access to an object, go with the Proxy Pattern. If you want to add or modify the behavior of an object dynamically, the Decorator Pattern is more appropriate.

Question 9. Describe the Chain of Responsibility Pattern. How does it promote loose coupling, and what are some practical applications of this pattern?

Answer:
The Chain of Responsibility Pattern is a behavioral design pattern used to pass a request along a chain of handlers. Each handler in the chain can either process the request or pass it along to the next handler in the chain. This pattern helps to decouple the sender of a request from its receivers by allowing multiple objects to process the request without the sender needing to know which object will handle it.

How It Works
Handler Interface: Define a common interface for handling requests, often including a method for processing the request and a method for setting the next handler in the chain.

Concrete Handlers: Implement specific handlers that process requests or pass them along the chain. Each concrete handler can decide whether to handle the request or to pass it to the next handler in the chain.

Client Code: The client code initiates the request and starts the chain of responsibility by sending the request to the first handler in the chain.

Promoting Loose Coupling

The Chain of Responsibility Pattern promotes loose coupling in several ways:

Decoupling Request Senders and Handlers: The sender of a request does not need to know which handler will process the request. It simply passes the request to the first handler in the chain, which might process it or pass it along.

Flexible Handler Assignment: Handlers can be added or removed from the chain dynamically, allowing the chain to be modified at runtime based on specific needs or conditions.

Responsibility Separation: Each handler in the chain is only responsible for a specific part of the request handling, promoting a single responsibility principle and making the system easier to understand and maintain.

Practical Applications

Event Handling Systems: In GUI frameworks, events like clicks or key presses can be handled by a chain of event handlers. Each handler in the chain can decide whether to handle the event or pass it to the next handler.

Middleware in Web Frameworks: Web frameworks often use middleware chains to process HTTP requests. Each middleware component can handle aspects of the request (like authentication, logging, or modification) or pass the request to the next middleware.

Validation Frameworks: In a validation system, different validators can be arranged in a chain to check various conditions or rules. Each validator handles a specific validation task and passes the request along if it doesn't handle it.

Logging Systems: A chain of responsibility can be used for logging, where each handler in the chain processes the log message (e.g., console, file, network) based on its level or type.

Data Processing Pipelines: In data processing applications, a chain of responsibility can be used to process data through various stages or transformations, where each stage can either process the data or pass it to the next stage.

Overall, the Chain of Responsibility Pattern is useful in scenarios where you need to pass a request through a series of potential handlers and want to achieve a flexible, loosely coupled design.

Question 10. Explain the concept of the State Pattern. How does it allow an object to alter its behavior when its internal state changes? Provide an example of its implementation.

Answer:

The State Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. Essentially, the State Pattern helps an object to appear to change its class when its state changes, enabling it to respond differently to the same inputs depending on its current state.

Concept of the State Pattern

1. State Interface:

Defines an interface for encapsulating the behavior associated with a particular state of the context.

2. Concrete States:

Implement the State interface to define specific behaviors for different states.

3. Context:

Maintains an instance of a Concrete State subclass that defines the current state. It delegates state-specific behavior to the current state object.

4. State Transitions:

The Context can change its state based on certain conditions or events. The state transition is typically managed by the state objects themselves or by the context.

How It Works

State Interface:

Defines the operations that can be performed and the state-specific behavior.

Concrete State Classes:

Each concrete state implements the State interface and provides the specific behavior associated with that state.

Context:

Maintains a reference to a State object and delegates state-specific behavior to the current State object.

State Transition:

The Context changes its state by switching its reference to different State objects.

Example Implementation in Java

Let’s consider a simple example: a TrafficLight system where the light changes its behavior depending on its current state (Red, Yellow, Green).

State Interface

java

Copy code

```
public interface TrafficLightState {
    void handleRequest(TrafficLightContext context);
}
```

Concrete States

java

Copy code

```
public class RedState implements TrafficLightState {
    @Override
    public void handleRequest(TrafficLightContext context) {
        System.out.println("Red light. Cars must stop.");
        context.setState(new GreenState()); // Transition to Green
    }
}

public class GreenState implements TrafficLightState {
    @Override
    public void handleRequest(TrafficLightContext context) {
```

```
        System.out.println("Green light. Cars can go.");
        context.setState(new YellowState()); // Transition to Yellow
    }
}

public class YellowState implements TrafficLightState {
    @Override
    public void handleRequest(TrafficLightContext context) {
        System.out.println("Yellow light. Cars should prepare to stop.");
        context.setState(new RedState()); // Transition to Red
    }
}
```

Context

java

Copy code

```
public class TrafficLightContext {
    private TrafficLightState state;

    public TrafficLightContext() {
        // Initial state
        state = new RedState();
    }

    public void setState(TrafficLightState state) {
        this.state = state;
    }

    public void request() {
        state.handleRequest(this);
    }
}
```

Client Code

java

Copy code

```
public class Main {
```

```
    public static void main(String[] args) {
        TrafficLightContext context = new TrafficLightContext();

        // Cycle through traffic light states
        context.request(); // Red light
        context.request(); // Green light
        context.request(); // Yellow light
        context.request(); // Red light again
    }
}
```

Explanation

State Interface (TrafficLightState):

Defines the handleRequest method that each state will implement.

Concrete States (RedState, GreenState, YellowState):

Implement the behavior for each traffic light state and define state transitions by setting the next state in the TrafficLightContext.

Context (TrafficLightContext):

Maintains the current state and delegates state-specific behavior to the current state object.
It also facilitates state transitions by setting new states.

Client Code:

The TrafficLightContext is used to request state-specific behavior, which automatically transitions between states.

The State Pattern allows an object to change its behavior when its internal state changes without requiring complex conditionals or state management logic in the object itself. Instead, state-specific behavior is encapsulated in state objects, making the code more maintainable and easier to understand.