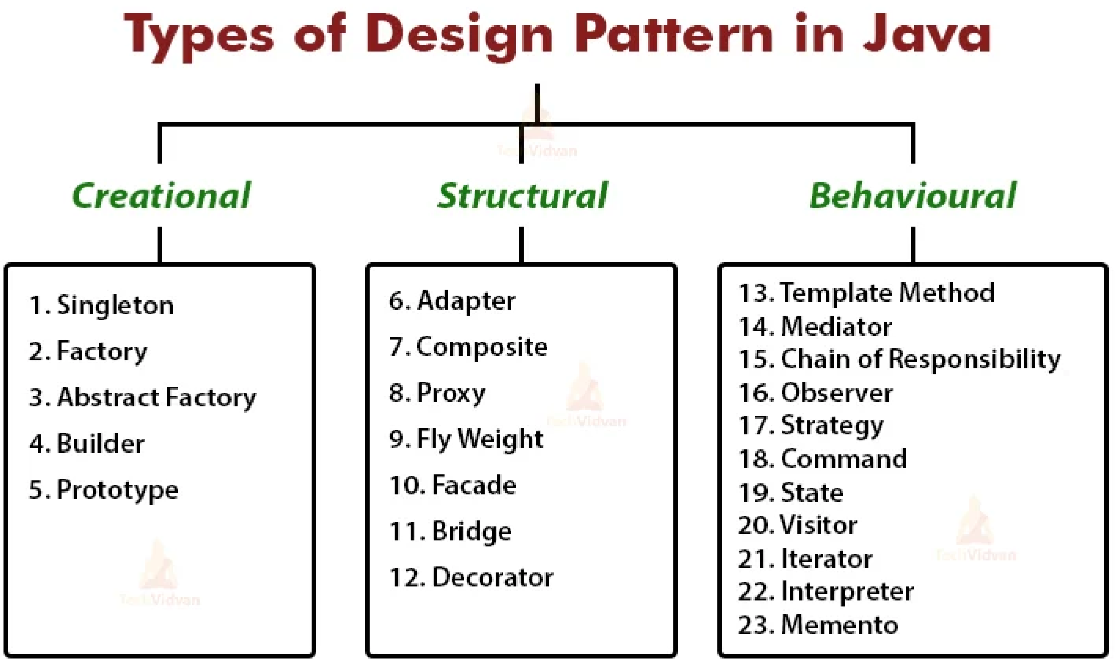
What are Apex Design Patterns?

Apex Design Patterns are best practices and reusable solutions to common problems encountered during Salesforce application development. These patterns help developers design robust, maintainable, and scalable code by following established design principles.

Why Do We Need Apex Design Patterns?

- **Maintainability**: Patterns provide a structured approach to coding, making the code easier to maintain and modify.
- **Reusability**: Solutions can be reused across different parts of the application or even across projects.
- **Scalability**: Well-designed code can handle growth and increased complexity more gracefully.
- **Efficiency**: Patterns optimize code performance and resource utilization.
- **Consistency**: Ensures a consistent approach to problem-solving, making the codebase easier for teams to understand and collaborate on.



Pattern	Description
Singleton	Ensure a class has only one instance and provide a global point of access to it.
Builder	Construct complex objects step by step.
Factory	Create objects without specifying the exact class of object to be created.

Pattern	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects.
Prototype	Create new objects by copying an existing object (prototype).
Adapter	Convert the interface of a class into another interface that clients expect.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies.
Decorator	Add behavior to objects dynamically.
Facade	Provide a simplified interface to a complex subsystem.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Chain of Responsibility	Pass requests along a chain of handlers.
Command	Encapsulate a request as an object for parameterization and queuing.
Interpreter	Define a representation for a language's grammar and provide an interpreter.
Iterator	Provide a way to access the elements of an aggregate object sequentially.
Mediator	Define an object that encapsulates how a set of objects interact.
Memento	Capture and externalize an object's internal state for later restoration.
Observer	Define a one-to-many dependency between objects for state change notifications.
State	Allow an object to alter its behavior when its internal state changes.
Strategy	Define a family of algorithms, making them interchangeable.
Template Method	Define the skeleton of an algorithm, deferring some steps to subclasses.
Visitor	Represent an operation to be performed on the elements of an object structure.

How Do We Use Apex Design Patterns?

1. Singleton Pattern

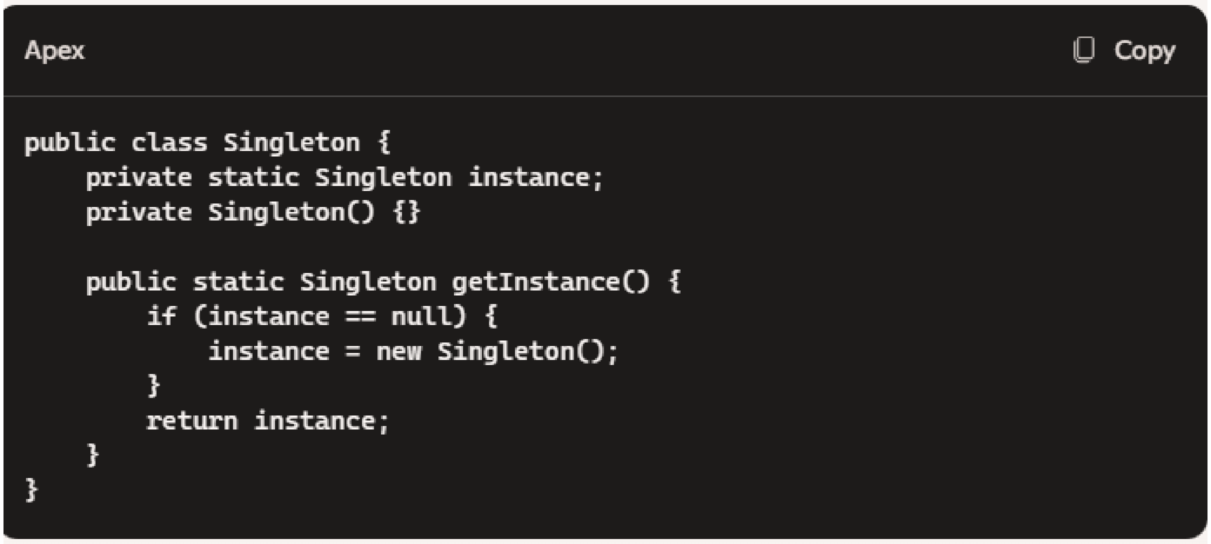
A creational design pattern that ensures a class has only one instance and provides a global point of access to it.

Purpose: Ensure a class has only one instance and provide a global point of access.

Usage: Used for classes that manage resources or configurations that should be shared across the application.

Problem Scenario: In a scenario where a developer needs to manage a shared resource, like a configuration setting, creating multiple instances of the class can lead to inconsistencies.

The Solution: Implementing Singleton Pattern:



```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}

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

Advantages of Singleton Pattern:

- Ensures a single, consistent instance of the class.
- Reduces memory usage by preventing multiple instances.
- Provides a global point of access to the resource.
- Simplifies resource management in multi-threaded applications.

2. Builder Pattern

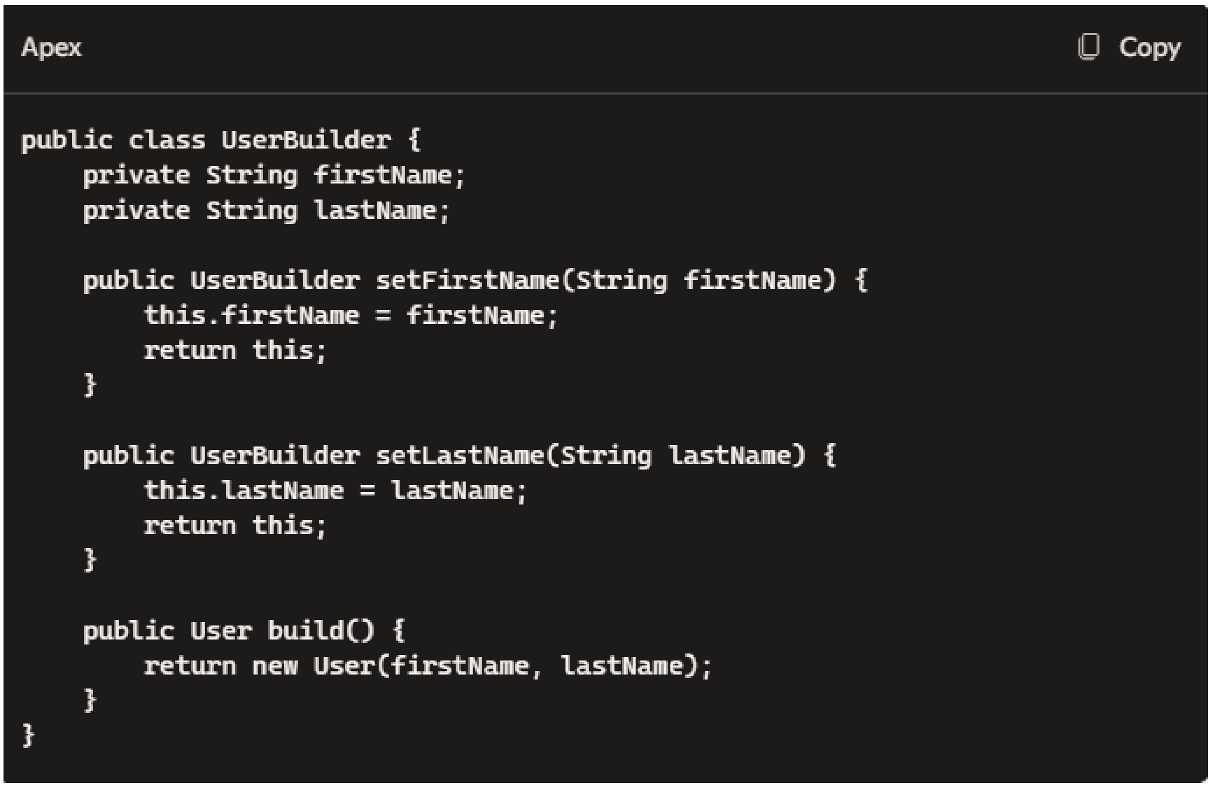
A creational design pattern that constructs complex objects step by step.

Purpose: Construct complex objects step by step.

Usage: Used when creating objects with multiple configurations or when an object requires many initialization parameters.

Problem Scenario: When creating an object with multiple optional configurations, constructors can become unwieldy and hard to manage.

The Solution: Implementing Builder Pattern:



```
public class UserBuilder {
    private String firstName;
    private String lastName;

    public UserBuilder setFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    public UserBuilder setLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }

    public User build() {
        return new User(firstName, lastName);
    }
}
```

Advantages of Builder Pattern:

- Simplifies object creation with multiple configurations.
- Provides a clear and fluent interface for object construction.
- Encapsulates the construction process.
- Enhances code readability and maintainability.

3. Factory Pattern

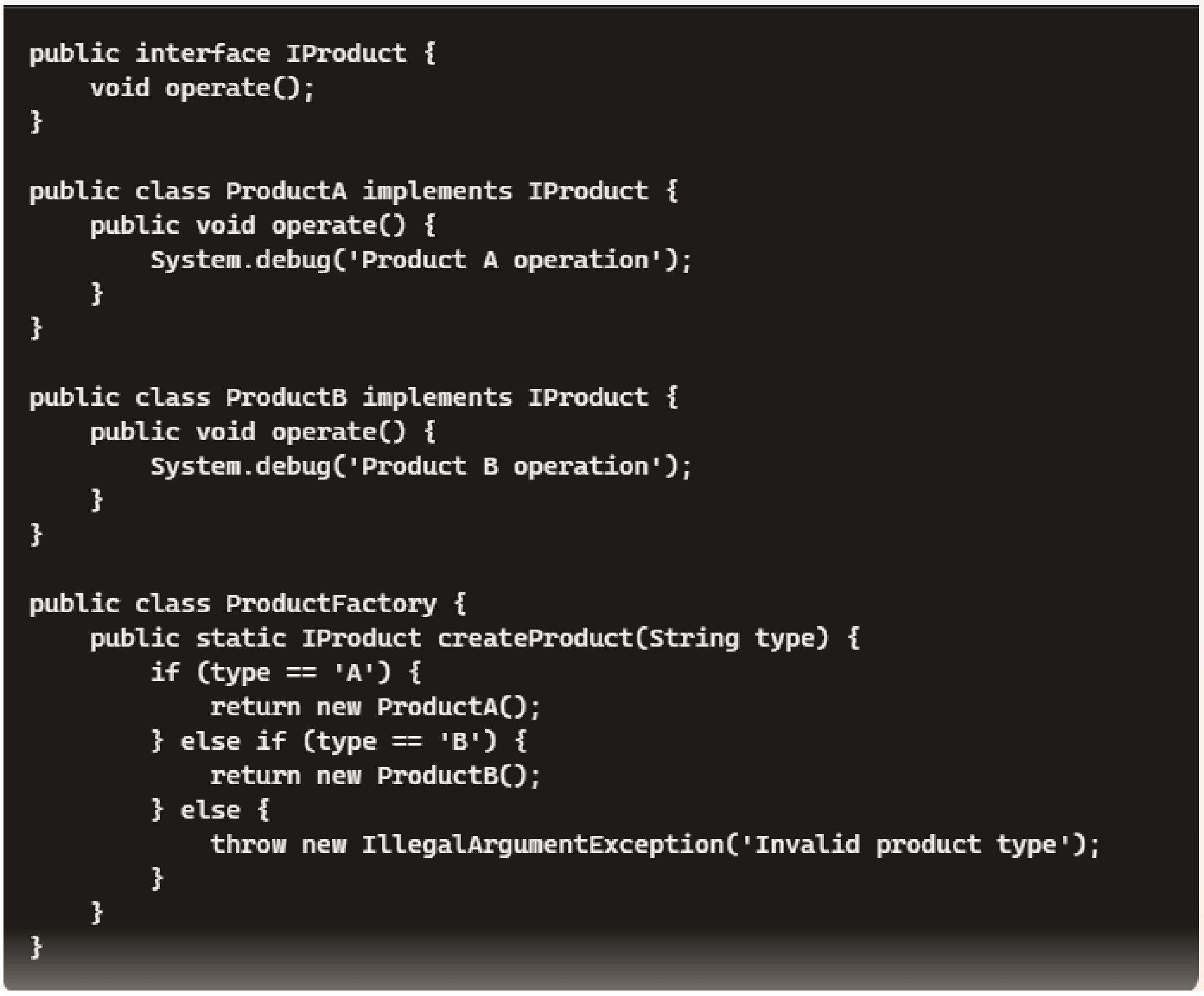
A creational design pattern that creates objects without specifying the exact class of object to be created.

Purpose: Creates objects without specifying the exact class of object that will be created.

Usage: Used when the creation process is complex, or the class to instantiate is selected at runtime.

Problem Scenario: When a system needs to instantiate objects of different classes, hardcoding the instantiation logic can lead to tight coupling and maintenance challenges.

The Solution: Implementing Factory Pattern:



```
public interface IProduct {
    void operate();
}

public class ProductA implements IProduct {
    public void operate() {
        System.debug('Product A operation');
    }
}

public class ProductB implements IProduct {
    public void operate() {
        System.debug('Product B operation');
    }
}

public class ProductFactory {
    public static IProduct createProduct(String type) {
        if (type == 'A') {
            return new ProductA();
        } else if (type == 'B') {
            return new ProductB();
        } else {
            throw new IllegalArgumentException('Invalid product type');
        }
    }
}
```

Advantages of Factory Pattern:

- Decouples object creation from the client code.
- Promotes code reusability and flexibility.
- Simplifies adding new types of objects without modifying existing code.

4. Abstract Factory Pattern

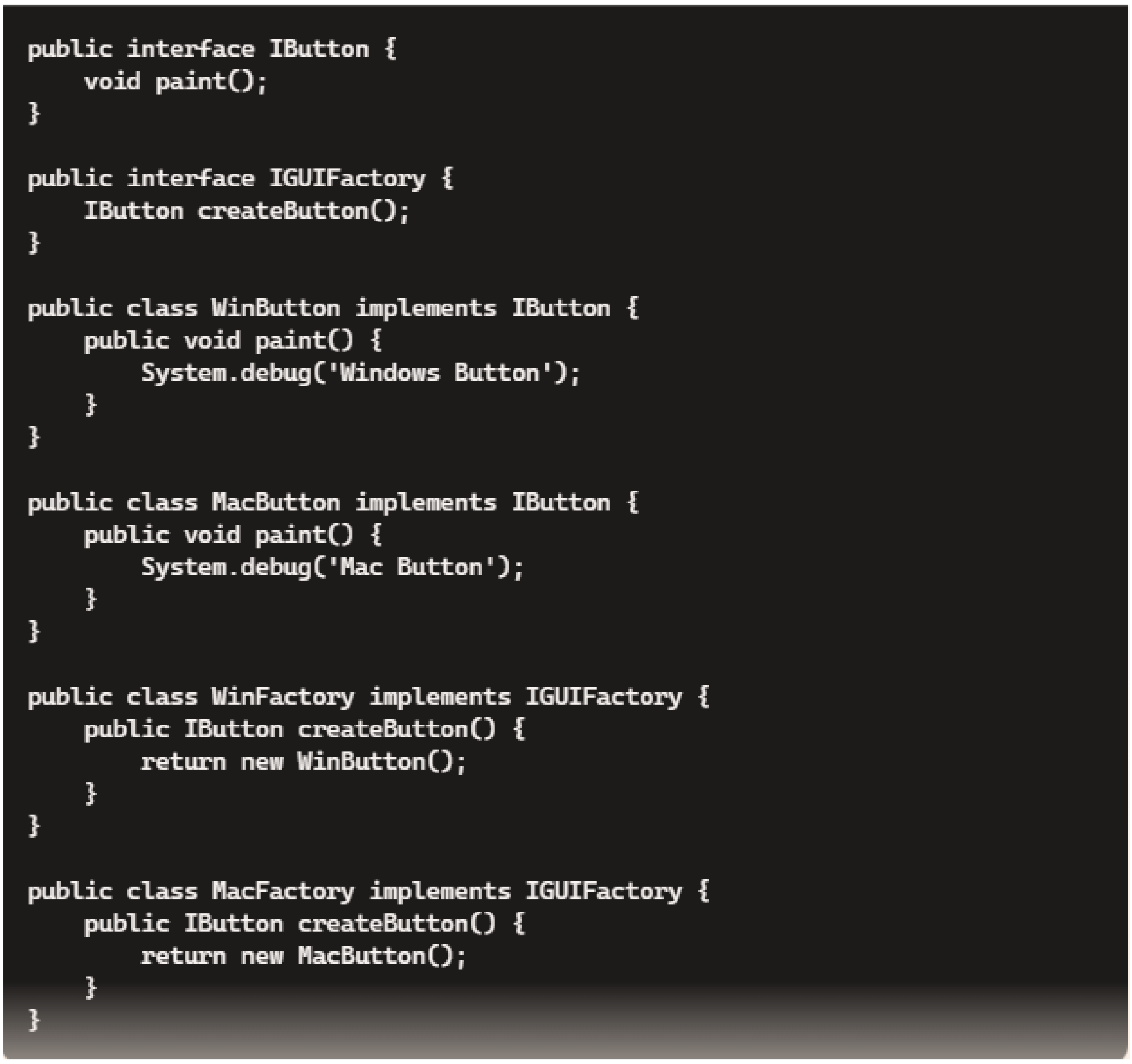
A creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Purpose: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Usage: Used when a system needs to be independent of how its objects are created.

Problem Scenario: When a system needs to be independent of how its objects are created and represented, hardcoding the instantiation logic can limit flexibility.

The Solution: Implementing Abstract Factory Pattern:



```
public interface IButton {
    void paint();
}

public interface IGUIFactory {
    IButton createButton();
}

public class WinButton implements IButton {
    public void paint() {
        System.debug('Windows Button');
    }
}

public class MacButton implements IButton {
    public void paint() {
        System.debug('Mac Button');
    }
}

public class WinFactory implements IGUIFactory {
    public IButton createButton() {
        return new WinButton();
    }
}

public class MacFactory implements IGUIFactory {
    public IButton createButton() {
        return new MacButton();
    }
}
```

Advantages of Abstract Factory Pattern:

- Promotes consistency among products.
- Encapsulates the creation of related objects.
- Enhances flexibility and scalability of the codebase.

5. Prototype Pattern

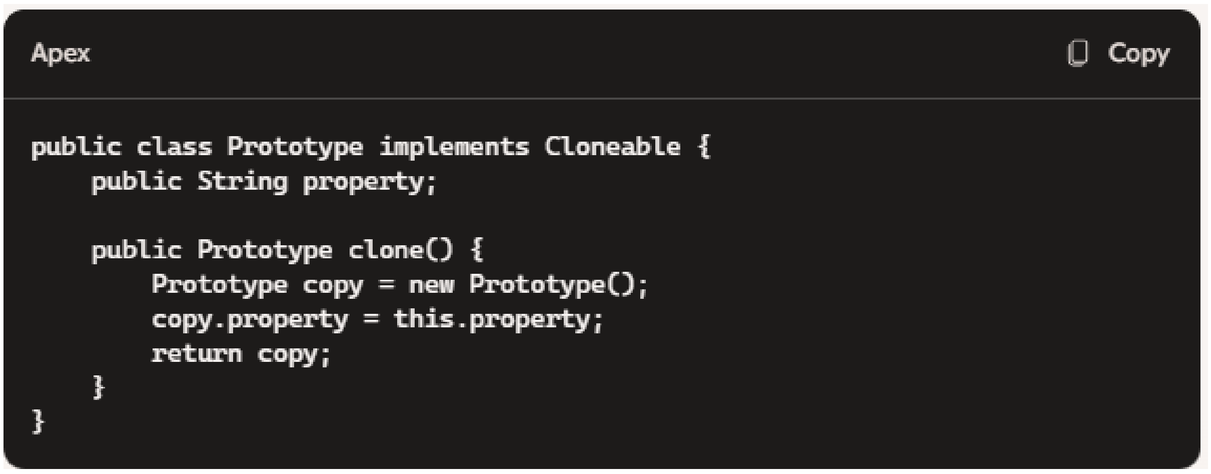
A creational design pattern that creates new objects by copying an existing object (prototype).

Purpose: Creates new objects by copying an existing object, known as the prototype.

Usage: Used when the cost of creating a new instance of a class is expensive or complex.

Problem Scenario: When object creation is expensive or complex, creating new instances from scratch can be inefficient.

The Solution: Implementing Prototype Pattern:



```
public class Prototype implements Cloneable {
    public String property;

    public Prototype clone() {
        Prototype copy = new Prototype();
        copy.property = this.property;
        return copy;
    }
}
```

Advantages of Prototype Pattern:

- Reduces the cost of creating new objects.
- Simplifies object creation when instances are similar.
- Enhances performance by reusing existing instances.

6. Adapter Pattern

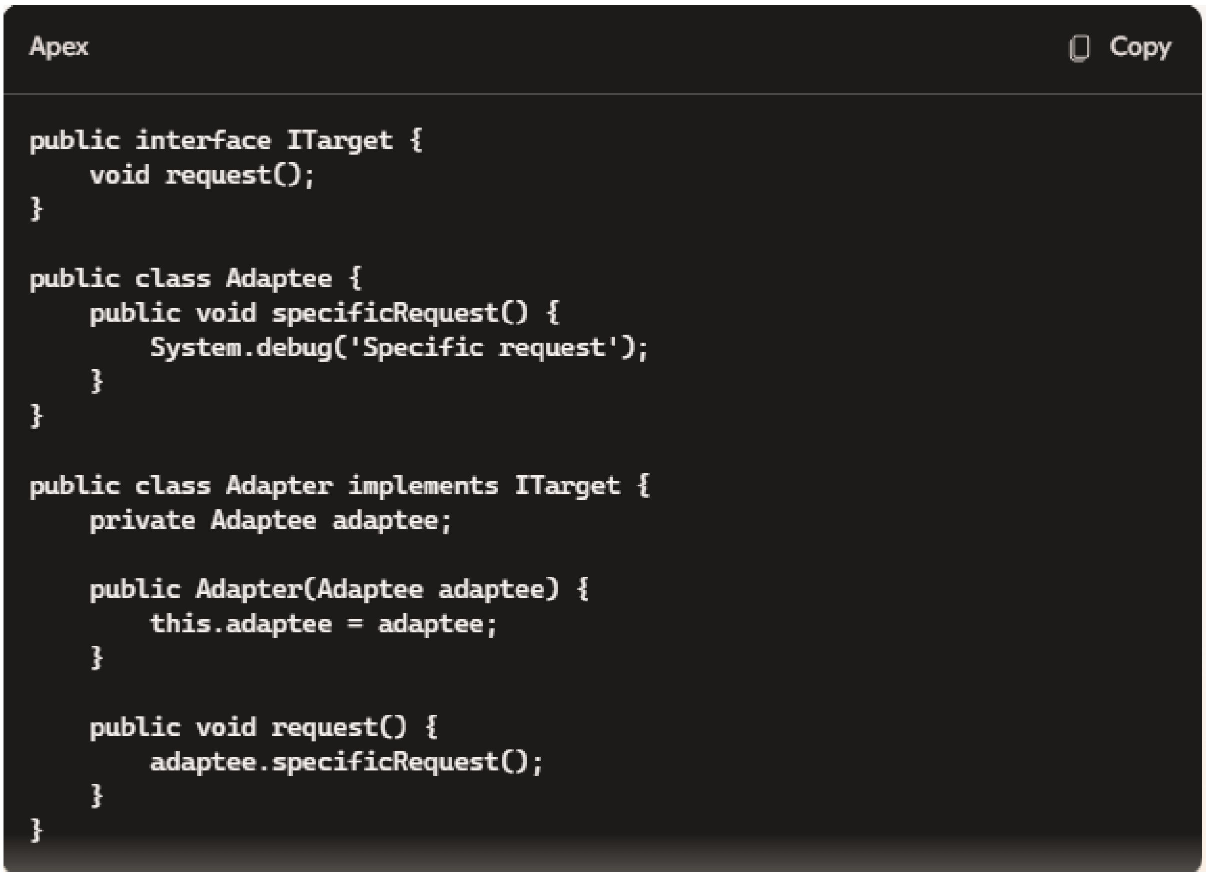
A structural design pattern that converts the interface of a class into another interface that clients expect.

Purpose: Convert the interface of a class into another interface that clients expect.

Usage: Used to make incompatible interfaces work together.

Problem Scenario: When integrating with an existing system, the interfaces might be incompatible, leading to integration challenges.

The Solution: Implementing Adapter Pattern:



```
public interface ITarget {
    void request();
}

public class Adaptee {
    public void specificRequest() {
        System.debug('Specific request');
    }
}

public class Adapter implements ITarget {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void request() {
        adaptee.specificRequest();
    }
}
```

Advantages of Adapter Pattern:

- Promotes code reuse by allowing integration with existing systems.
- Simplifies interactions between incompatible interfaces.
- Enhances flexibility and scalability of the system.

7. Bridge Pattern

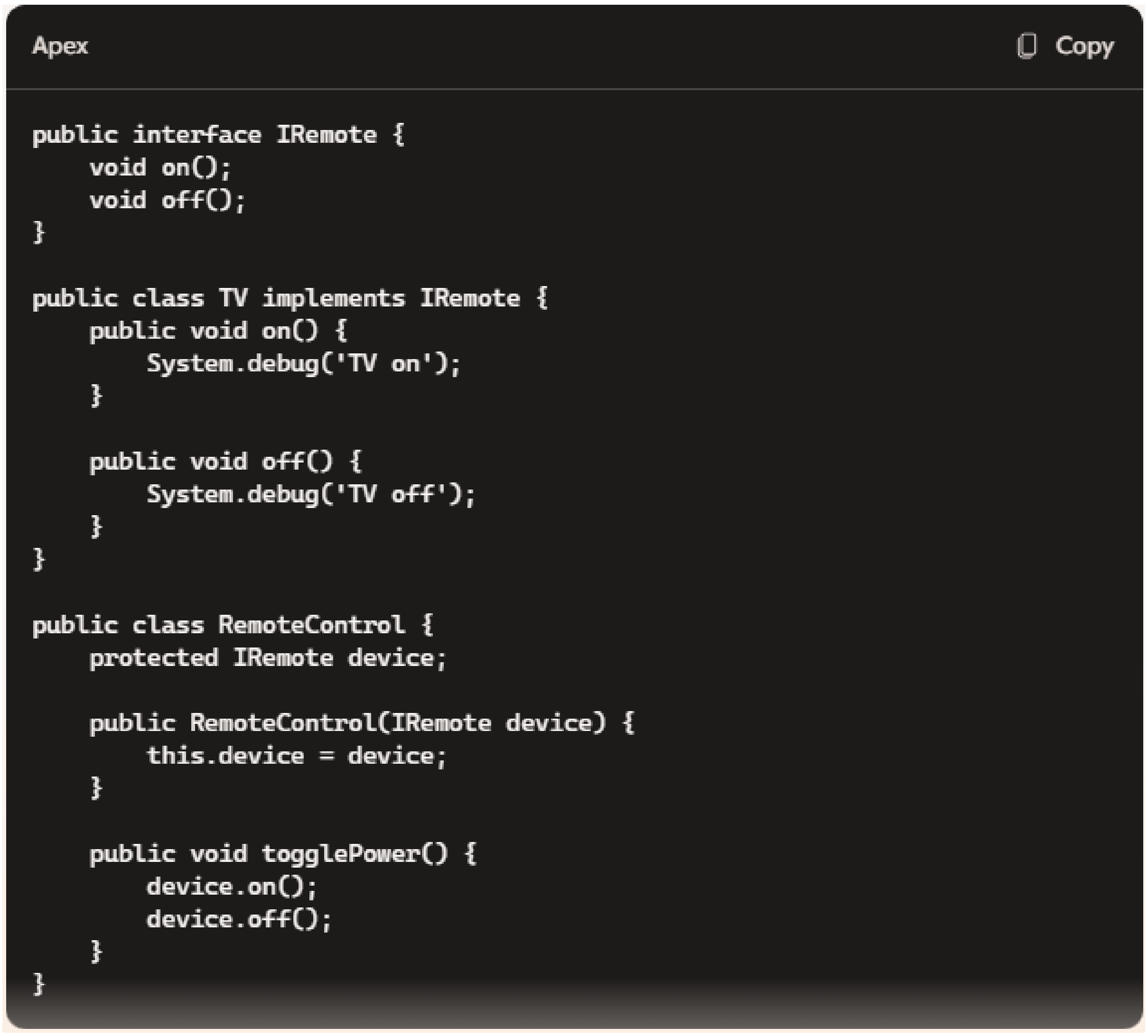
A structural design pattern that decouples an abstraction from its implementation so that the two can vary independently.

Purpose: Decouple an abstraction from its implementation so that the two can vary independently.

Usage: Used when you want to separate an object’s interface from its implementation.

Problem Scenario: When an object’s interface and its implementation need to vary independently, tight coupling can limit flexibility.

The Solution: Implementing Bridge Pattern:



```apex
public interface IRemote {
    void on();
    void off();
}

public class TV implements IRemote {
    public void on() {
        System.debug('TV on');
    }

    public void off() {
        System.debug('TV off');
    }
}

public class RemoteControl {
    protected IRemote device;

    public RemoteControl(IRemote device) {
        this.device = device;
    }

    public void togglePower() {
        device.on();
        device.off();
    }
}
```

Advantages of Bridge Pattern:

- Promotes flexibility by decoupling interface and implementation.
- Simplifies adding new implementations without modifying existing code.
- Enhances scalability and maintainability.

8. Composite Pattern

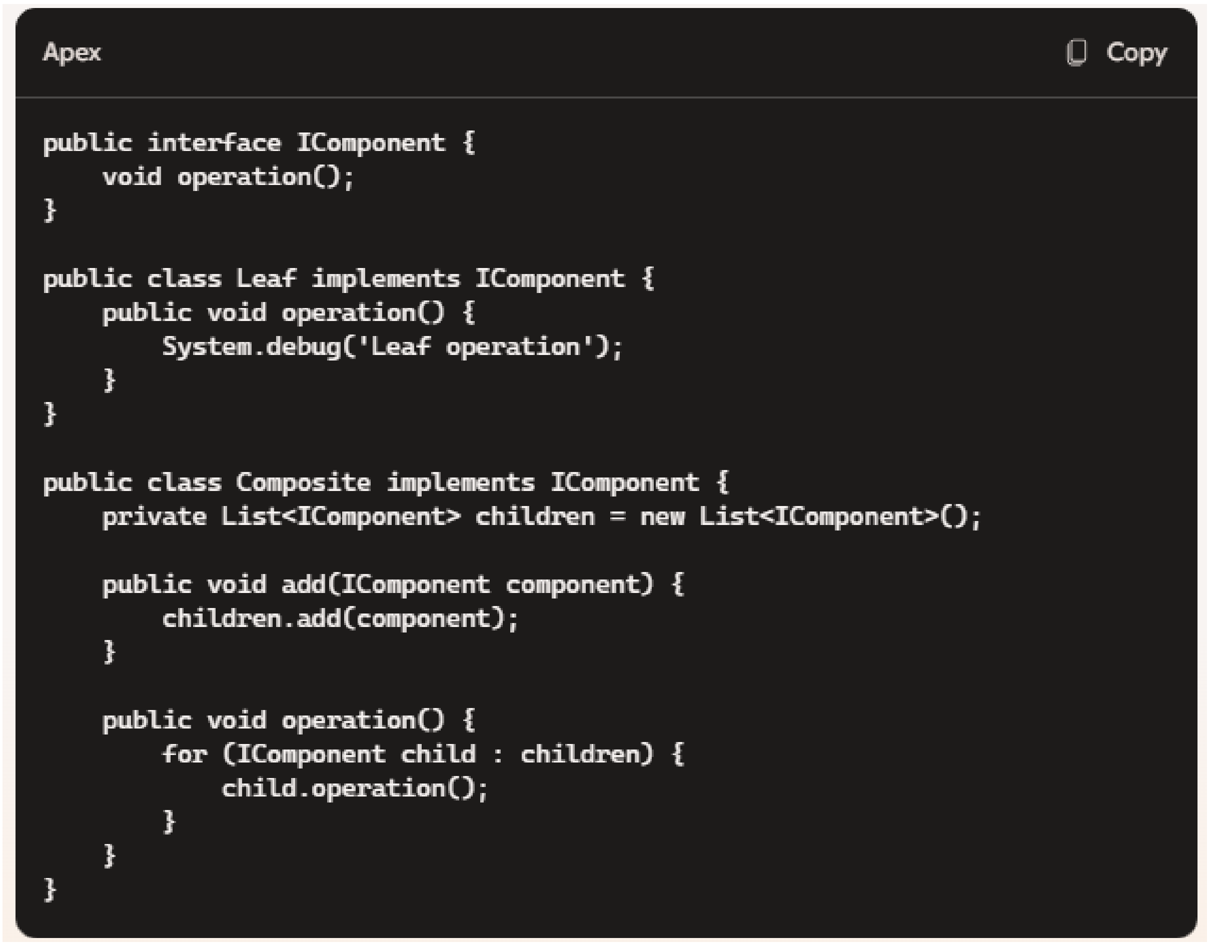
A structural design pattern that composes objects into tree structures to represent part-whole hierarchies.

Purpose: Compose objects into tree structures to represent part-whole hierarchies.

Usage: Used to treat individual objects and compositions of objects uniformly.

Problem Scenario: When dealing with a part-whole hierarchy, treating individual objects and compositions uniformly can be challenging.

The Solution: Implementing Composite Pattern:


Apex Copy

```
public interface IComponent {
    void operation();
}

public class Leaf implements IComponent {
    public void operation() {
        System.debug('Leaf operation');
    }
}

public class Composite implements IComponent {
    private List<IComponent> children = new List<IComponent>();

    public void add(IComponent component) {
        children.add(component);
    }

    public void operation() {
        for (IComponent child : children) {
            child.operation();
        }
    }
}
```

Advantages of Composite Pattern:

- Simplifies code by treating individual objects and compositions uniformly.
- Promotes code reuse through composition.
- Enhances flexibility and scalability.

9. Decorator Pattern

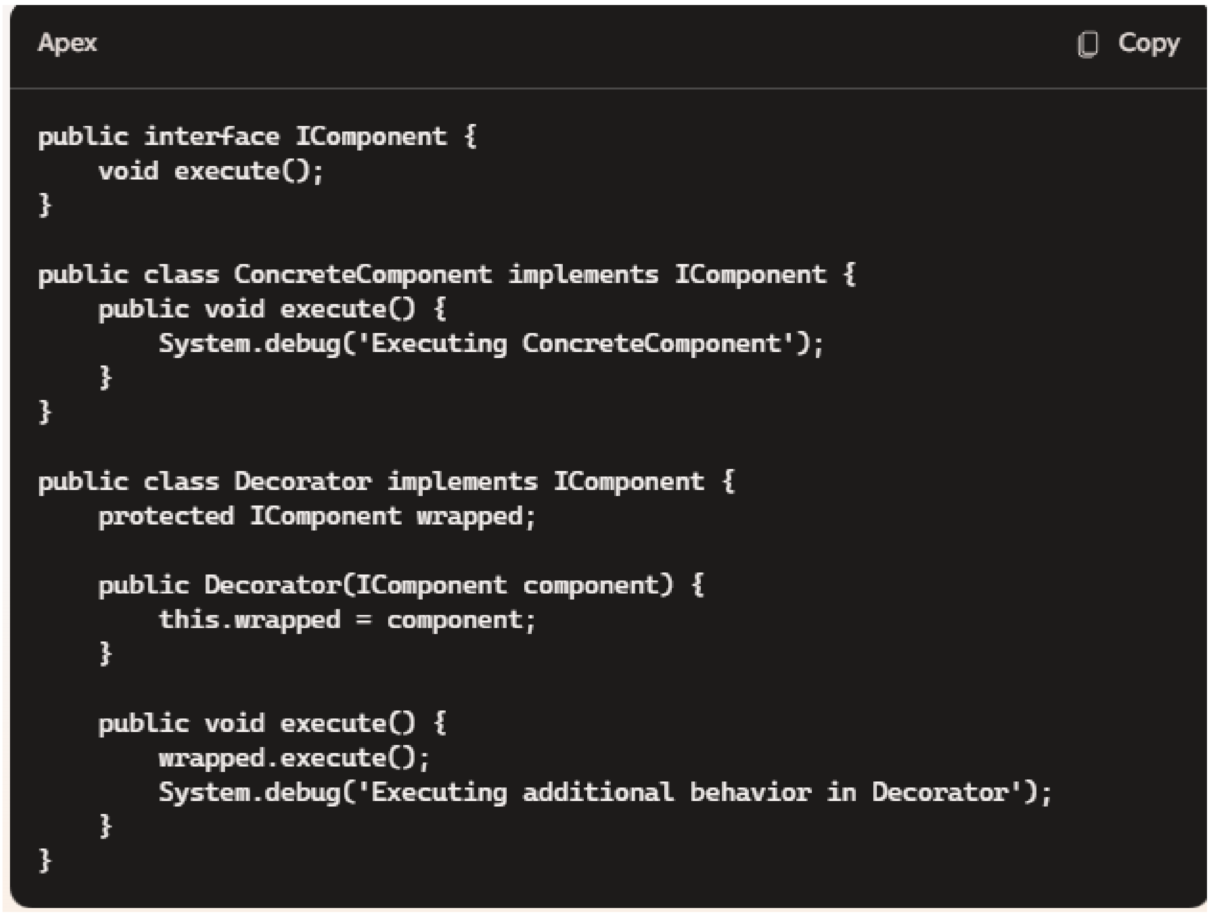
A structural design pattern that adds behavior to objects dynamically without affecting other objects from the same class.

Purpose: Add behavior to objects dynamically.

Usage: Used when you want to add responsibilities to objects without modifying their code.

Problem Scenario: When adding responsibilities to objects without modifying their code, the code can become cluttered and hard to manage.

The Solution: Implementing Decorator Pattern:


Apex

Copy

```
public interface IComponent {
    void execute();
}

public class ConcreteComponent implements IComponent {
    public void execute() {
        System.debug('Executing ConcreteComponent');
    }
}

public class Decorator implements IComponent {
    protected IComponent wrapped;

    public Decorator(IComponent component) {
        this.wrapped = component;
    }

    public void execute() {
        wrapped.execute();
        System.debug('Executing additional behavior in Decorator');
    }
}
```

Advantages of Decorator Pattern:

- Promotes code reuse by adding responsibilities dynamically.
- Simplifies adding new behavior without modifying existing code.
- Enhances flexibility and maintainability.

10. Flyweight Pattern

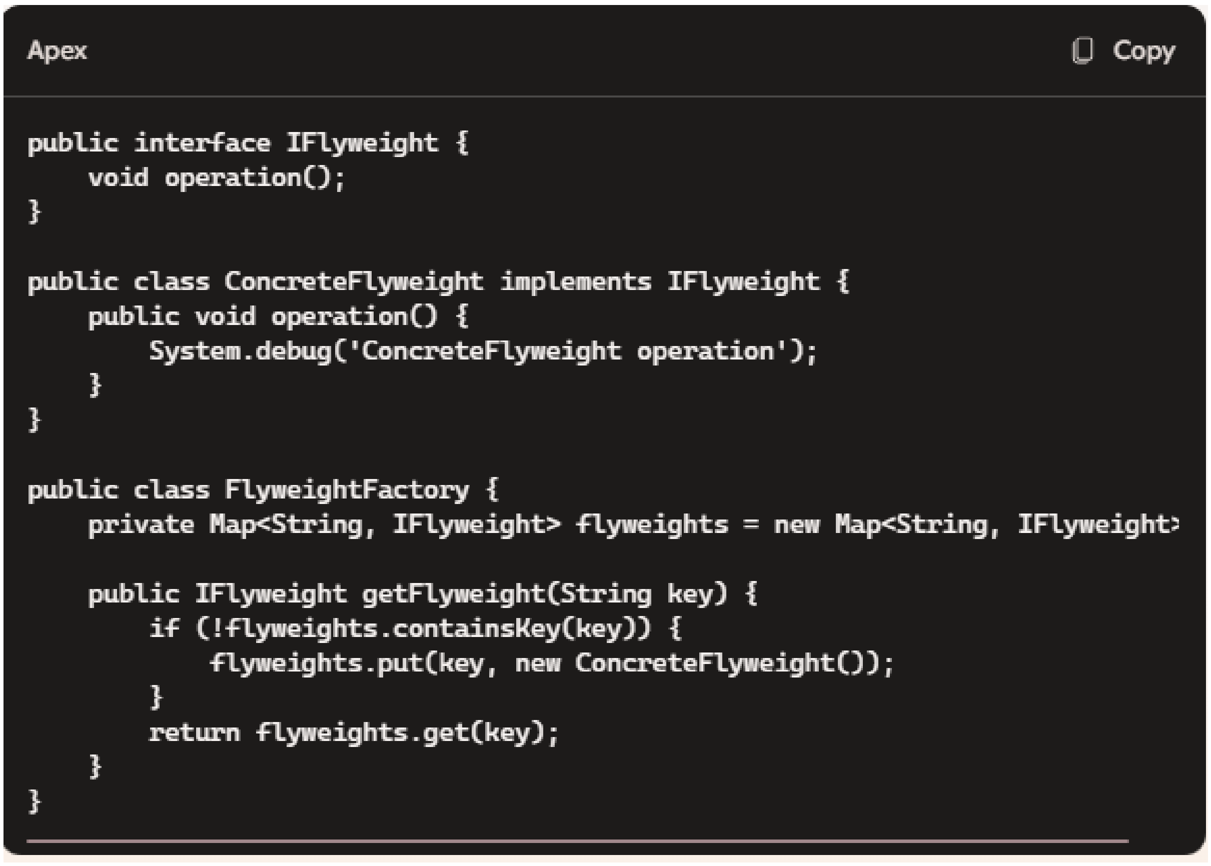
A structural design pattern that uses sharing to support large numbers of fine-grained objects efficiently.

Purpose: Use sharing to support large numbers of fine-grained objects efficiently.

Usage: Used when many objects of the same type are needed, saving memory by sharing common parts.

Problem Scenario: When creating many instances of the same type, memory consumption can become an issue.

The Solution: Implementing Flyweight Pattern:



```
public interface IFlyweight {
    void operation();
}

public class ConcreteFlyweight implements IFlyweight {
    public void operation() {
        System.debug('ConcreteFlyweight operation');
    }
}

public class FlyweightFactory {
    private Map<String, IFlyweight> flyweights = new Map<String, IFlyweight>

    public IFlyweight getFlyweight(String key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new ConcreteFlyweight());
        }
        return flyweights.get(key);
    }
}
```

Advantages of Flyweight Pattern:

- Reduces memory consumption by sharing common parts.
- Enhances performance and efficiency.
- Simplifies object management.

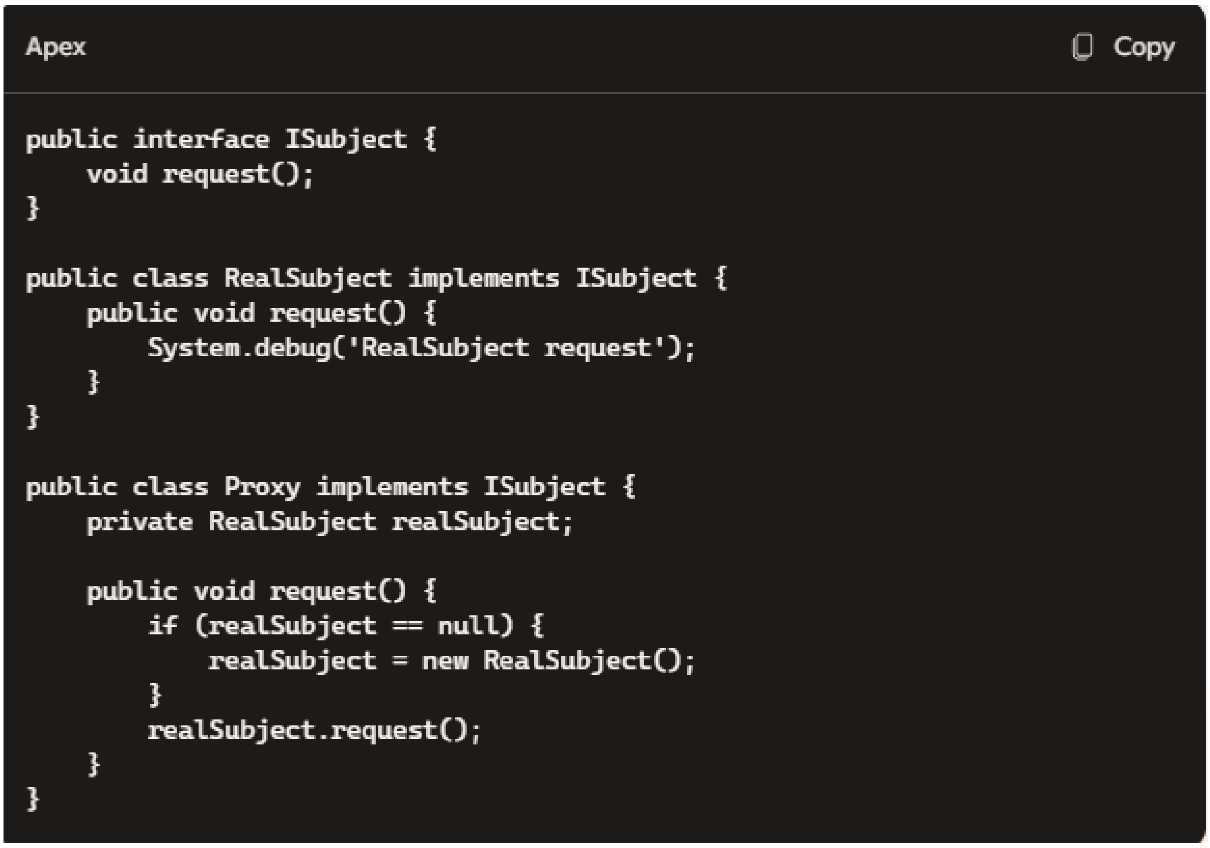
11. Proxy Pattern

A structural design pattern that provides a surrogate or placeholder for another object to control access to it.

Purpose: The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It is used to create a substitute object that controls access to a sensitive or resource-intensive object.

Problem Scenario: When you want to control access to an object that is resource-intensive to create, you might want to use a lightweight proxy that handles access to the real object.

The Solution: Implementing Proxy Pattern:


Apex

Copy

```
public interface ISubject {
    void request();
}

public class RealSubject implements ISubject {
    public void request() {
        System.debug('RealSubject request');
    }
}

public class Proxy implements ISubject {
    private RealSubject realSubject;

    public void request() {
        if (realSubject == null) {
            realSubject = new RealSubject();
        }
        realSubject.request();
    }
}
```

Advantages of Proxy Pattern:

- **Lazy Initialization**: Delays the creation and initialization of the real object until it is needed.
- **Access Control**: Provides a way to control access to the real object, enhancing security and encapsulation.
- **Cost Reduction**: Reduces the cost of accessing the real object by using a lightweight proxy.
- **Separation of Concerns**: Separates the client's interface from the real object, promoting cleaner code design.

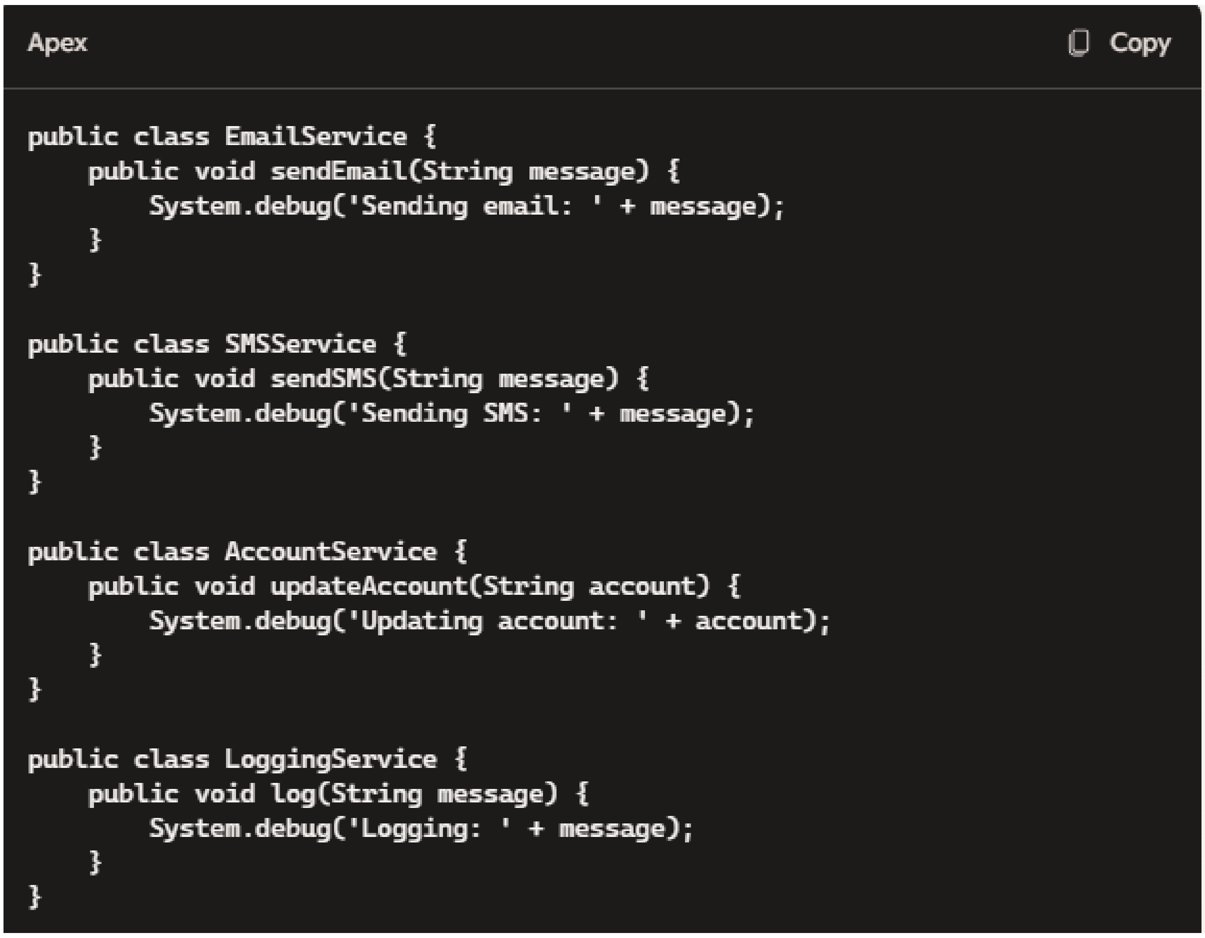
12. Facade Pattern

A structural design pattern that provides a simplified interface to a complex system.

Purpose: The Facade Pattern provides a simplified interface to a complex system. It offers a high-level view and hides the intricate workings of the subsystems.

Problem Scenario: When a developer needs to handle multiple subsystems, the code can become cluttered and hard to manage, making it prone to errors and difficult to extend or modify.

The Solution: Implementing Facade Pattern:

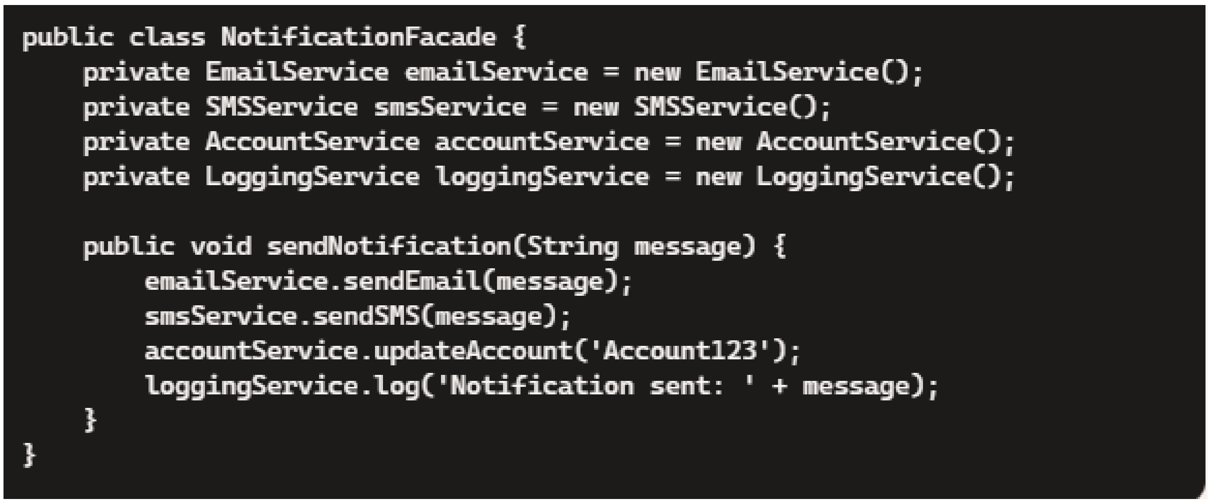


```
public class EmailService {
    public void sendEmail(String message) {
        System.debug('Sending email: ' + message);
    }
}

public class SMSService {
    public void sendSMS(String message) {
        System.debug('Sending SMS: ' + message);
    }
}

public class AccountService {
    public void updateAccount(String account) {
        System.debug('Updating account: ' + account);
    }
}

public class LoggingService {
    public void log(String message) {
        System.debug('Logging: ' + message);
    }
}
```



```
public class NotificationFacade {
    private EmailService emailService = new EmailService();
    private SMSService smsService = new SMSService();
    private AccountService accountService = new AccountService();
    private LoggingService loggingService = new LoggingService();

    public void sendNotification(String message) {
        emailService.sendEmail(message);
        smsService.sendSMS(message);
        accountService.updateAccount('Account123');
        loggingService.log('Notification sent: ' + message);
    }
}
```

Advantages of Facade Pattern:

- **Simplifies Complexity**: Provides an easy-to-use interface that hides the complex workings of subsystems.
- **Higher-Level Interface**: Allows users to interact at a higher level without dealing with intricate details.
- **Reduced Coupling**: Reduces the dependency between the client and the subsystems, promoting loose coupling.

- **Ease of Use**: Allows developers to focus on their tasks without being overwhelmed by system complexity.
- **Encapsulation**: Encapsulates changes within the facade, minimizing the impact on users when subsystems change.

13. Chain of Responsibility Pattern

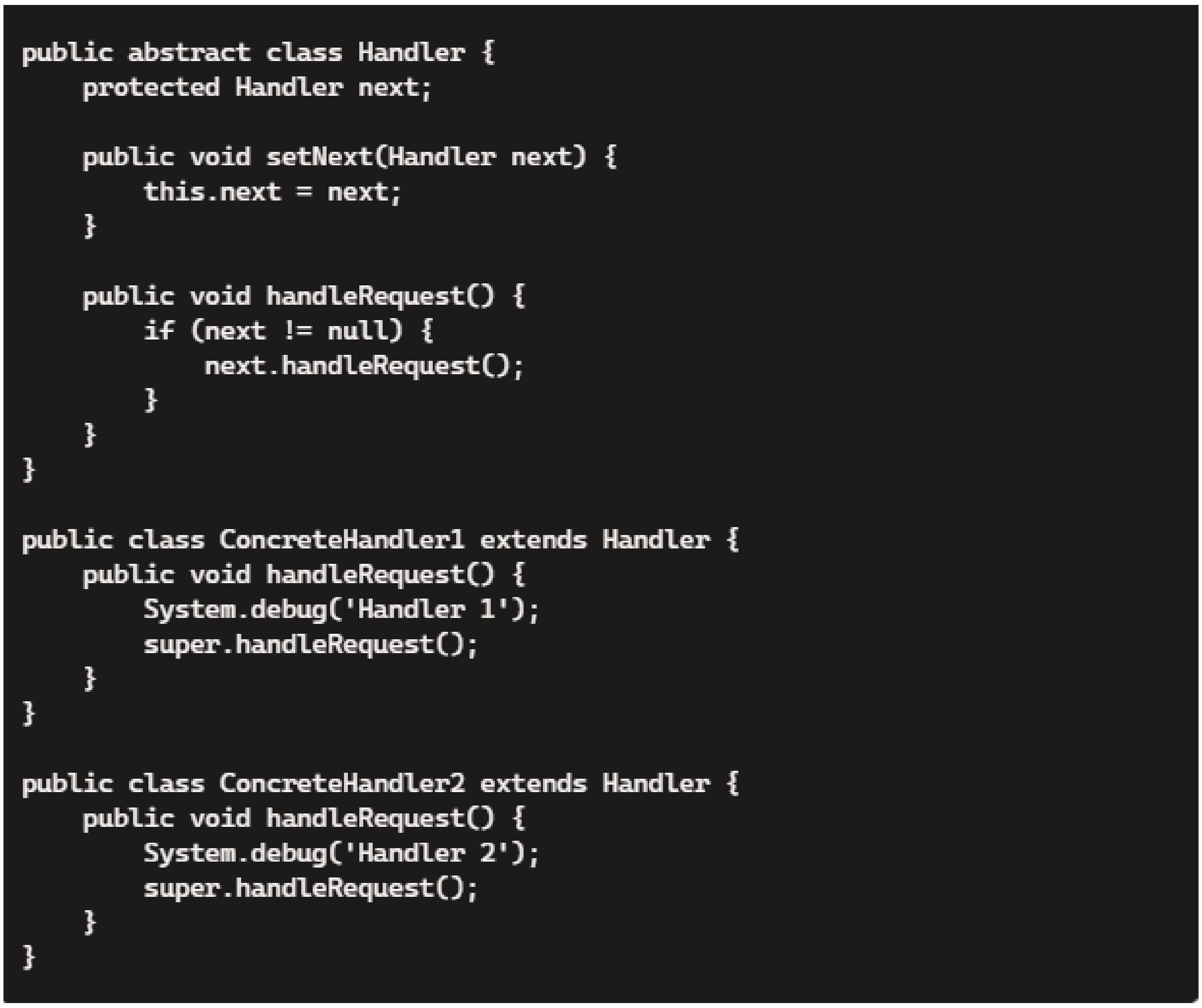
A behavioral design pattern that passes requests along a chain of handlers, where each handler decides either to handle the request or pass it along.

Purpose: Pass requests along a chain of handlers, where each handler decides either to handle the request or pass it along.

Usage: Used to decouple the sender of a request from its receiver, giving more than one object a chance to handle the request.

Problem Scenario: When handling a request, tight coupling between sender and receiver can limit flexibility and scalability.

The Solution: Implementing Chain of Responsibility Pattern:



```
public abstract class Handler {
    protected Handler next;

    public void setNext(Handler next) {
        this.next = next;
    }

    public void handleRequest() {
        if (next != null) {
            next.handleRequest();
        }
    }
}

public class ConcreteHandler1 extends Handler {
    public void handleRequest() {
        System.debug('Handler 1');
        super.handleRequest();
    }
}

public class ConcreteHandler2 extends Handler {
    public void handleRequest() {
        System.debug('Handler 2');
        super.handleRequest();
    }
}
```

Advantages of Chain of Responsibility Pattern:

- Decouples sender and receiver, enhancing flexibility.
- Simplifies adding new handlers without modifying existing code.
- Promotes a clean and maintainable request handling process.

14. Command Pattern

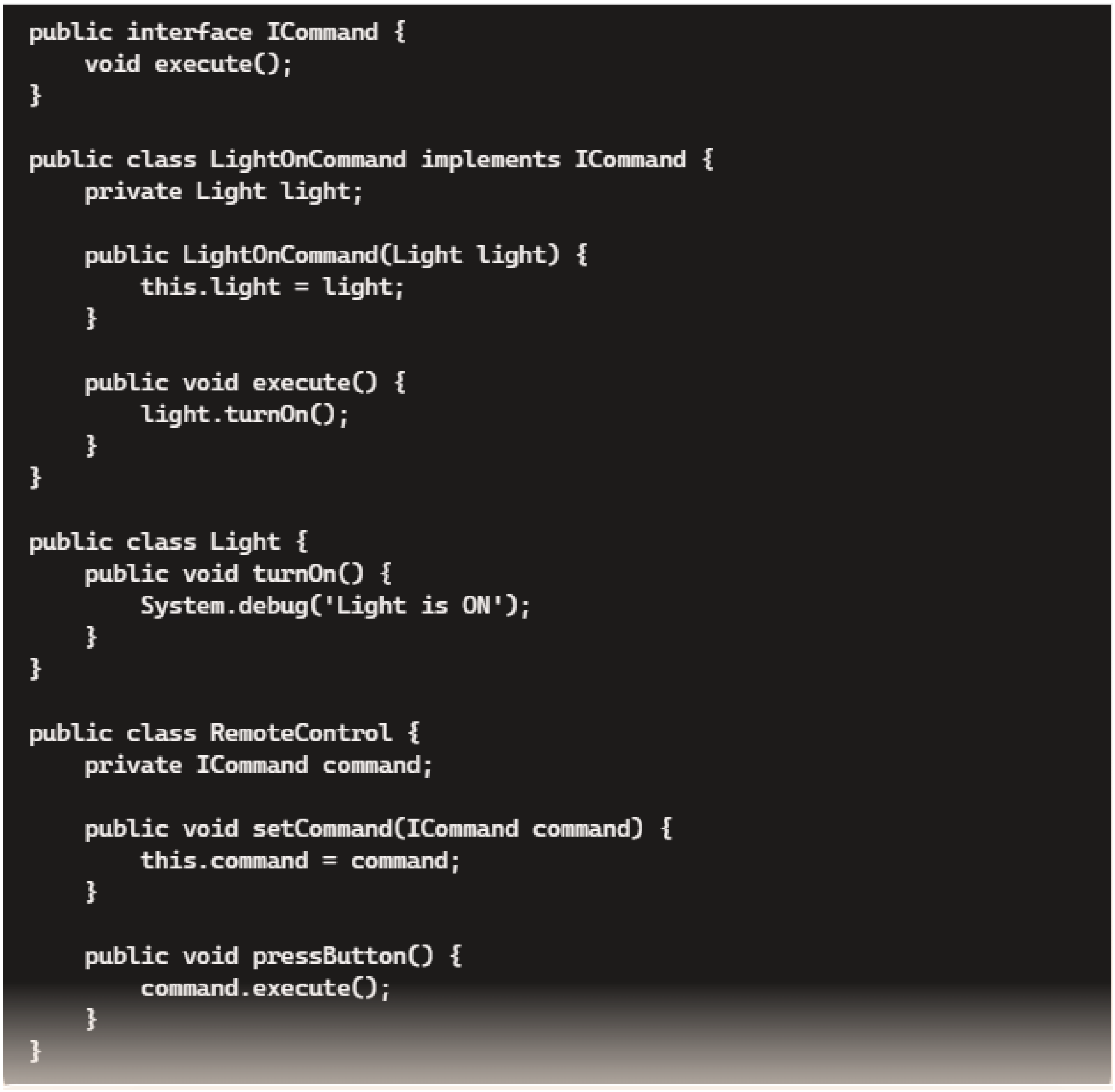
A behavioral design pattern that encapsulates a request as an object, allowing for parameterization and queuing of requests.

Purpose: Encapsulate a request as an object, allowing for parameterization and queuing of requests.

Usage: Used for implementing undoable operations.

Problem Scenario: When implementing undoable operations, tight coupling between request invokers and receivers can limit flexibility.

The Solution: Implementing Command Pattern:



```
public interface ICommand {
    void execute();
}

public class LightOnCommand implements ICommand {
    private Light light;

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

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

public class Light {
    public void turnOn() {
        System.debug('Light is ON');
    }
}

public class RemoteControl {
    private ICommand command;

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

    public void pressButton() {
        command.execute();
    }
}
```

Advantages of Command Pattern:

- Decouples request invokers and receivers.
- Simplifies implementing undoable operations.
- Enhances flexibility and scalability.

15. Interpreter Pattern

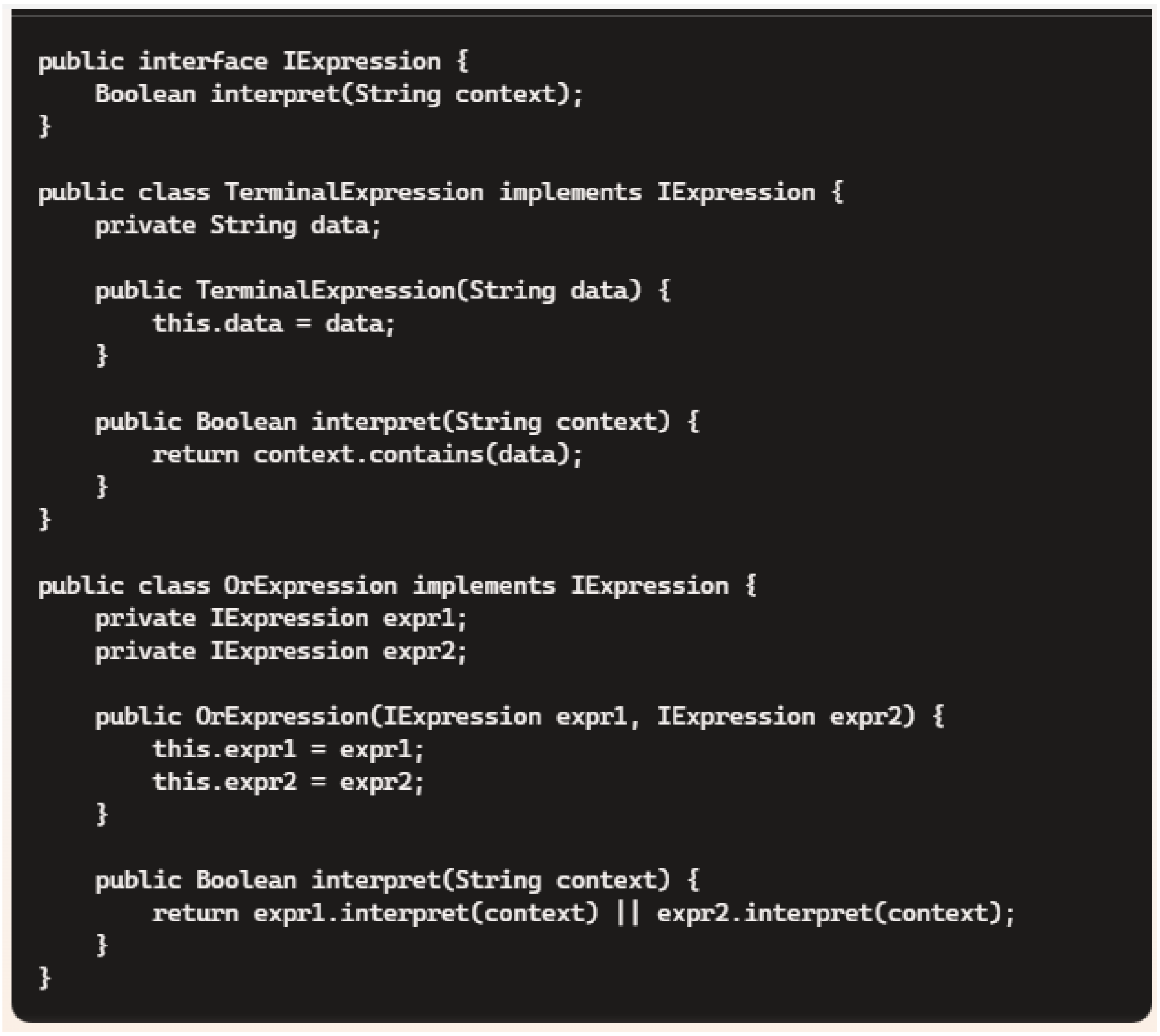
A behavioral design pattern that defines a representation for a language's grammar and provides an interpreter to interpret sentences in the language.

Purpose: Define a representation for a language's grammar and provide an interpreter to interpret sentences in the language.

Usage: Used to interpret expressions in a specific language or notation.

Problem Scenario: When interpreting expressions in a specific language or notation, hardcoding the interpretation logic can lead to tight coupling.

The Solution: Implementing Interpreter Pattern:



```java
public interface IExpression {
    Boolean interpret(String context);
}

public class TerminalExpression implements IExpression {
    private String data;

    public TerminalExpression(String data) {
        this.data = data;
    }

    public Boolean interpret(String context) {
        return context.contains(data);
    }
}

public class OrExpression implements IExpression {
    private IExpression expr1;
    private IExpression expr2;

    public OrExpression(IExpression expr1, IExpression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    public Boolean interpret(String context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}
```

Advantages of Interpreter Pattern:

- Simplifies interpreting expressions in a specific language or notation.
- Promotes code reuse and flexibility.
- Enhances maintainability and scalability.

16. Iterator Pattern

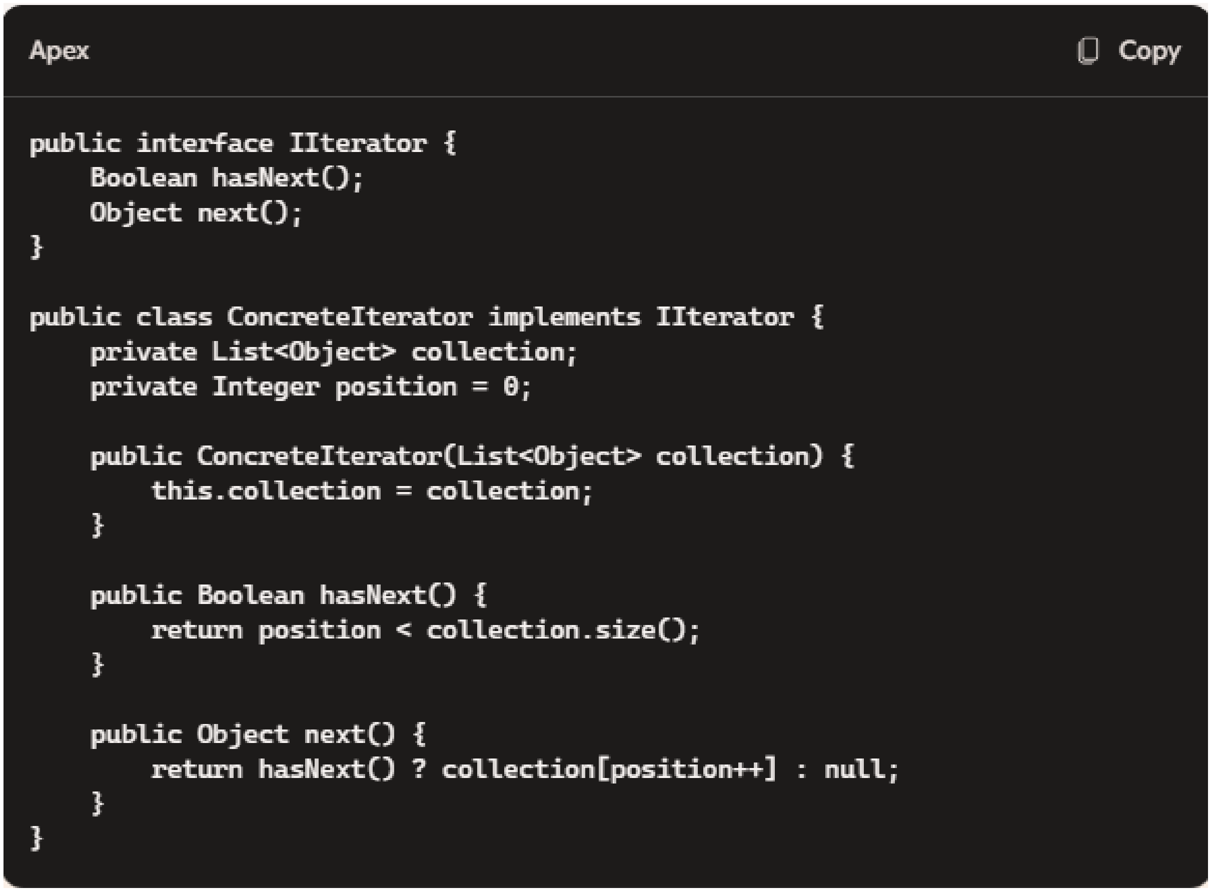
A behavioral design pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Purpose: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Usage: Used to iterate over a collection of objects.

Problem Scenario: When iterating over a collection of objects, tight coupling between the collection and its client can limit flexibility.

The Solution: Implementing Iterator Pattern:



```
public interface IIterator {
    Boolean hasNext();
    Object next();
}

public class ConcreteIterator implements IIterator {
    private List<Object> collection;
    private Integer position = 0;

    public ConcreteIterator(List<Object> collection) {
        this.collection = collection;
    }

    public Boolean hasNext() {
        return position < collection.size();
    }

    public Object next() {
        return hasNext() ? collection[position++] : null;
    }
}
```

Advantages of Iterator Pattern:

- Decouples the client from the collection implementation.
- Simplifies iterating over a collection of objects.
- Enhances flexibility and scalability.

17. Mediator Pattern

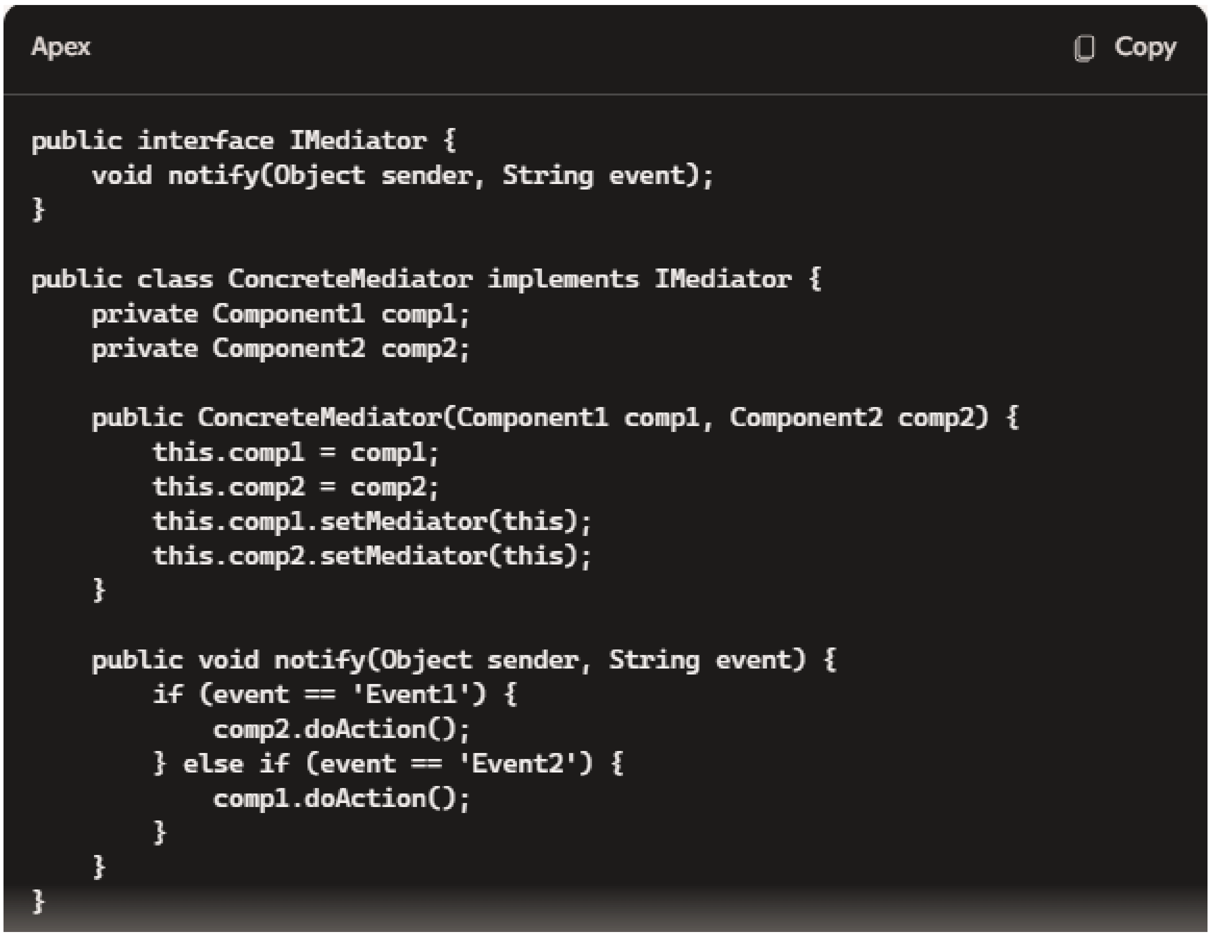
A behavioral design pattern that defines an object that encapsulates how a set of objects interact.

Purpose: Define an object that encapsulates how a set of objects interact.

Usage: Used to reduce the complexity of communication between multiple objects.

Problem Scenario: When multiple objects need to interact, tight coupling between them can lead to complex and hard-to-maintain code.

The Solution: Implementing Mediator Pattern:

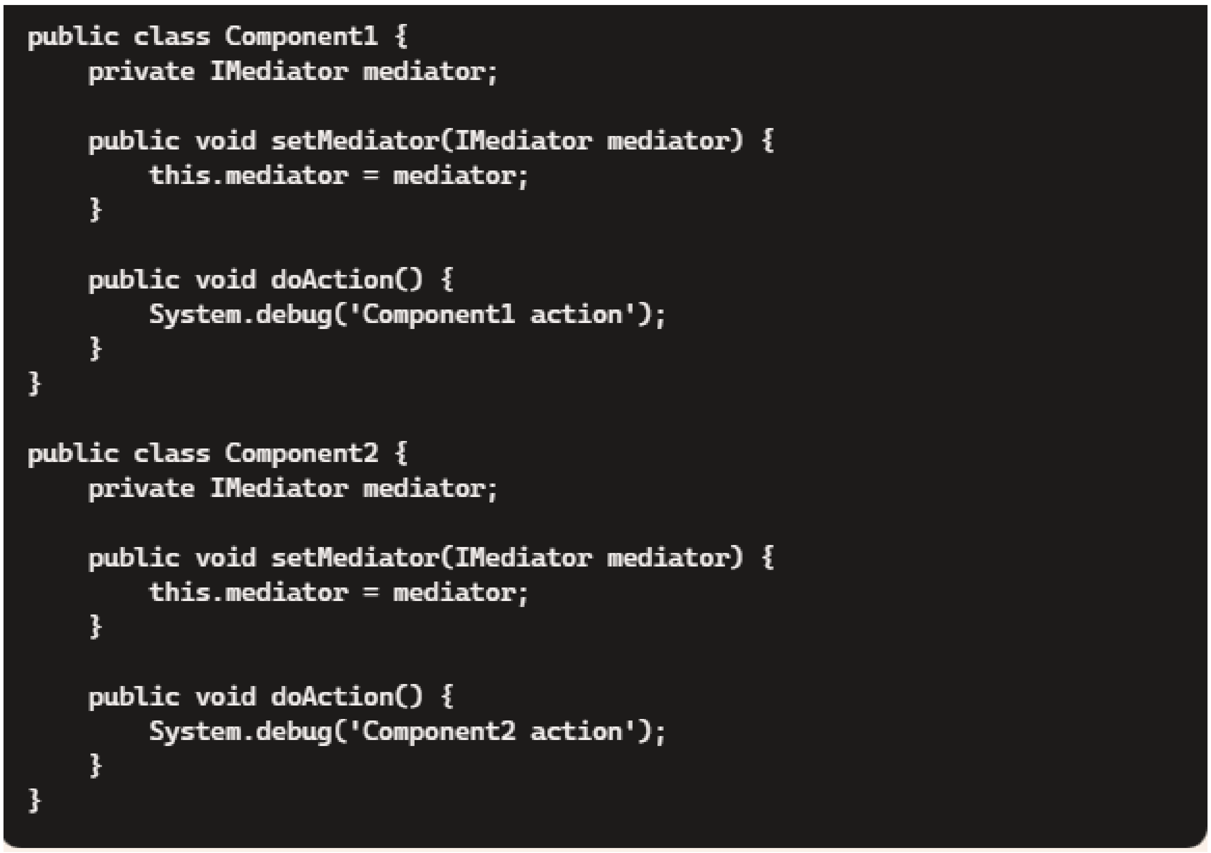


```apex
public interface IMediator {
    void notify(Object sender, String event);
}

public class ConcreteMediator implements IMediator {
    private Component1 comp1;
    private Component2 comp2;

    public ConcreteMediator(Component1 comp1, Component2 comp2) {
        this.comp1 = comp1;
        this.comp2 = comp2;
        this.comp1.setMediator(this);
        this.comp2.setMediator(this);
    }

    public void notify(Object sender, String event) {
        if (event == 'Event1') {
            comp2.doAction();
        } else if (event == 'Event2') {
            comp1.doAction();
        }
    }
}
```



```apex
public class Component1 {
    private IMediator mediator;

    public void setMediator(IMediator mediator) {
        this.mediator = mediator;
    }

    public void doAction() {
        System.debug('Component1 action');
    }
}

public class Component2 {
    private IMediator mediator;

    public void setMediator(IMediator mediator) {
        this.mediator = mediator;
    }

    public void doAction() {
        System.debug('Component2 action');
    }
}
```

Advantages of Mediator Pattern:

- Decouples objects, reducing complexity.
- Simplifies communication between objects.
- Enhances flexibility and maintainability.

18. Memento Pattern

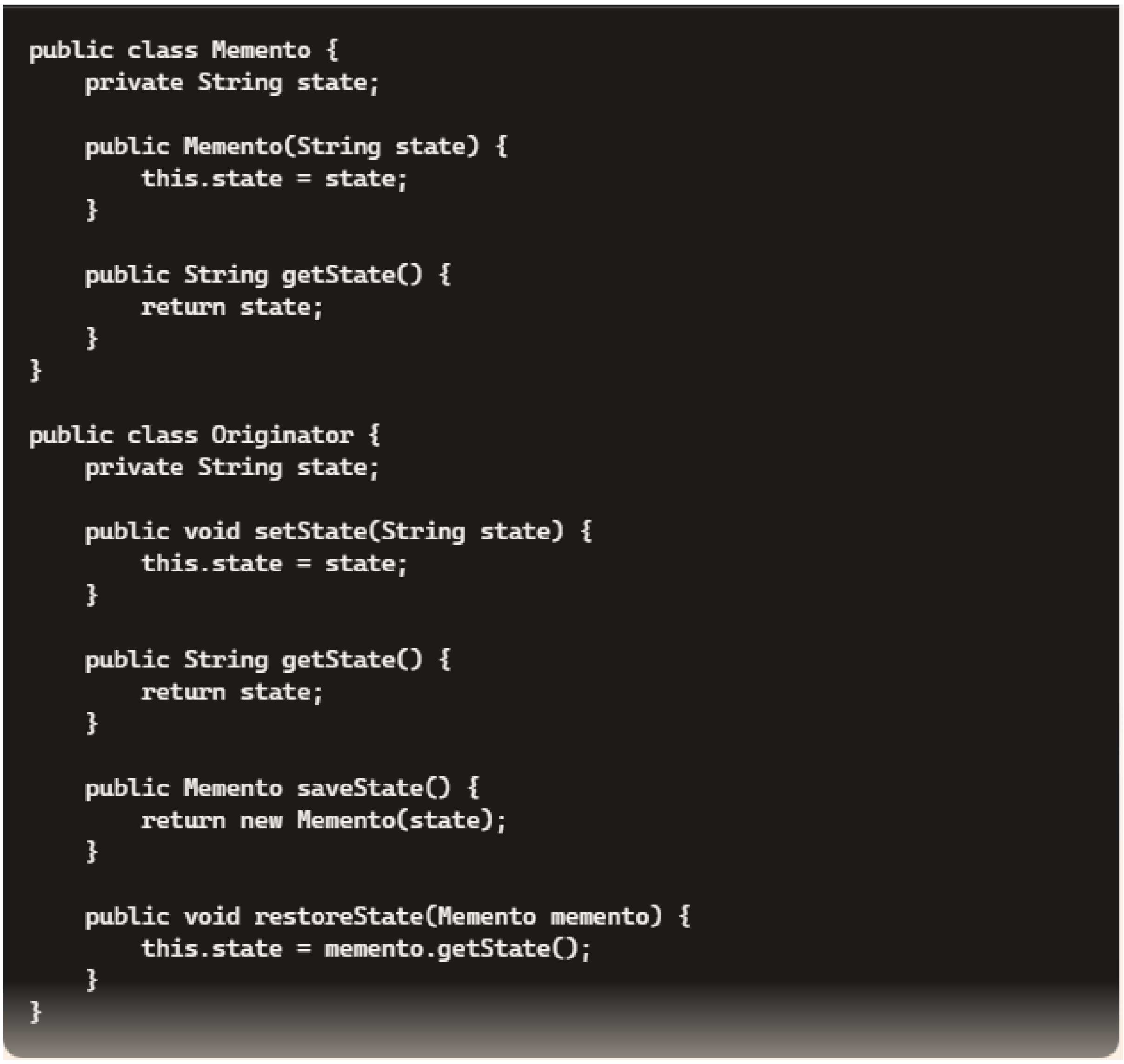
A behavioral design pattern that captures and externalizes an object's internal state so that the object can be restored to this state later.

Purpose: Capture and externalize an object's internal state so that the object can be restored to this state later.

Usage: Used to provide the ability to undo changes.

Problem Scenario: When implementing undo functionality, maintaining the object's previous states can be challenging.

The Solution: Implementing Memento Pattern:



```java
public class Memento {
    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}

public class Originator {
    private String state;

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

    public String getState() {
        return state;
    }

    public Memento saveState() {
        return new Memento(state);
    }

    public void restoreState(Memento memento) {
        this.state = memento.getState();
    }
}
```

Advantages of Memento Pattern:

- Simplifies implementing undo functionality.
- Promotes state encapsulation and isolation.
- Enhances flexibility and maintainability.

19. Observer Pattern

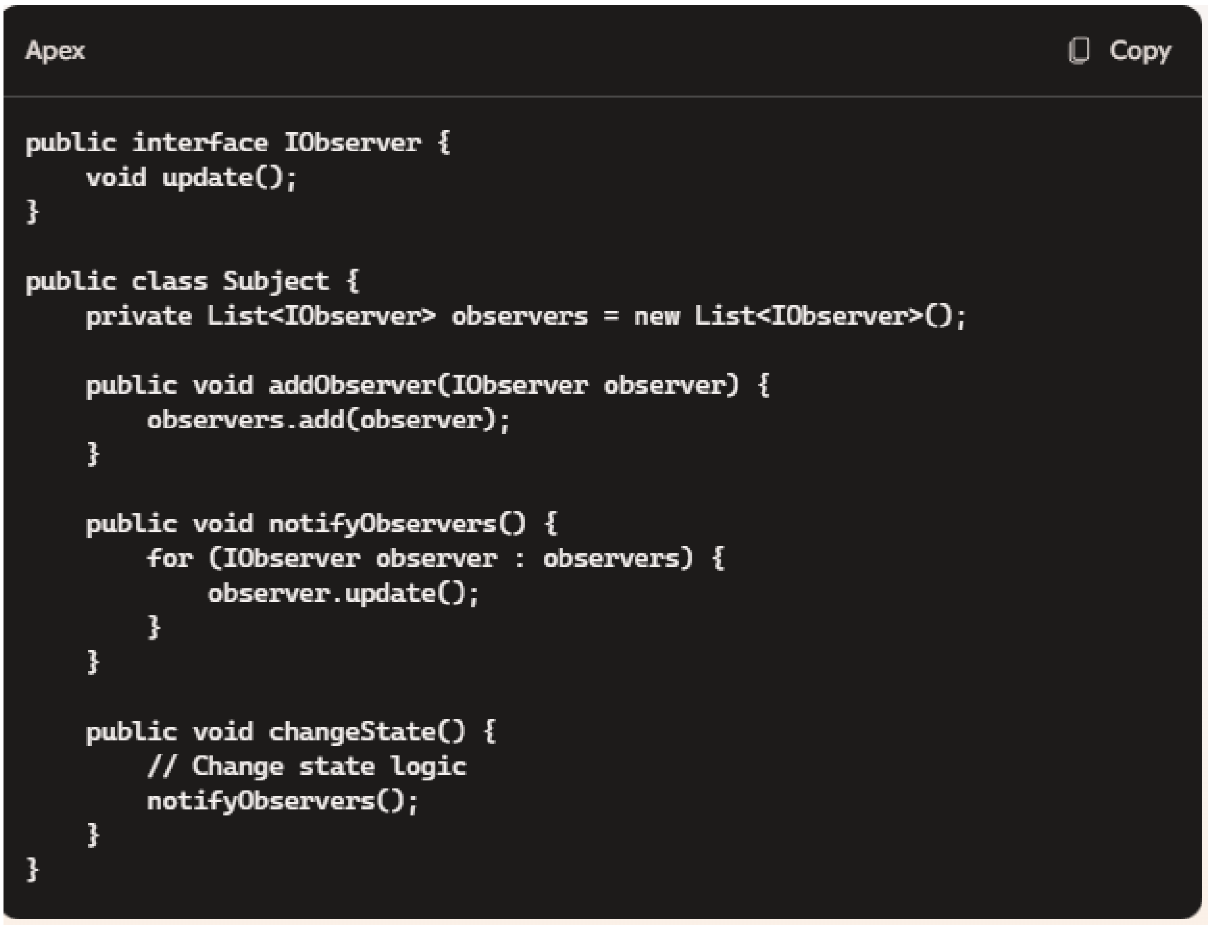
A behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Purpose: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Usage: Used to notify multiple objects of state changes.

Problem Scenario: When multiple objects need to be notified of changes in another object, tight coupling can limit flexibility.

The Solution: Implementing Observer Pattern:



```
public interface IObserver {
    void update();
}

public class Subject {
    private List<IObserver> observers = new List<IObserver>();

    public void addObserver(IObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        for (IObserver observer : observers) {
            observer.update();
        }
    }

    public void changeState() {
        // Change state logic
        notifyObservers();
    }
}
```

Advantages of Observer Pattern:

- Decouples the subject from its observers.
- Simplifies notifying multiple objects of state changes.
- Enhances flexibility and scalability.

20. State Pattern

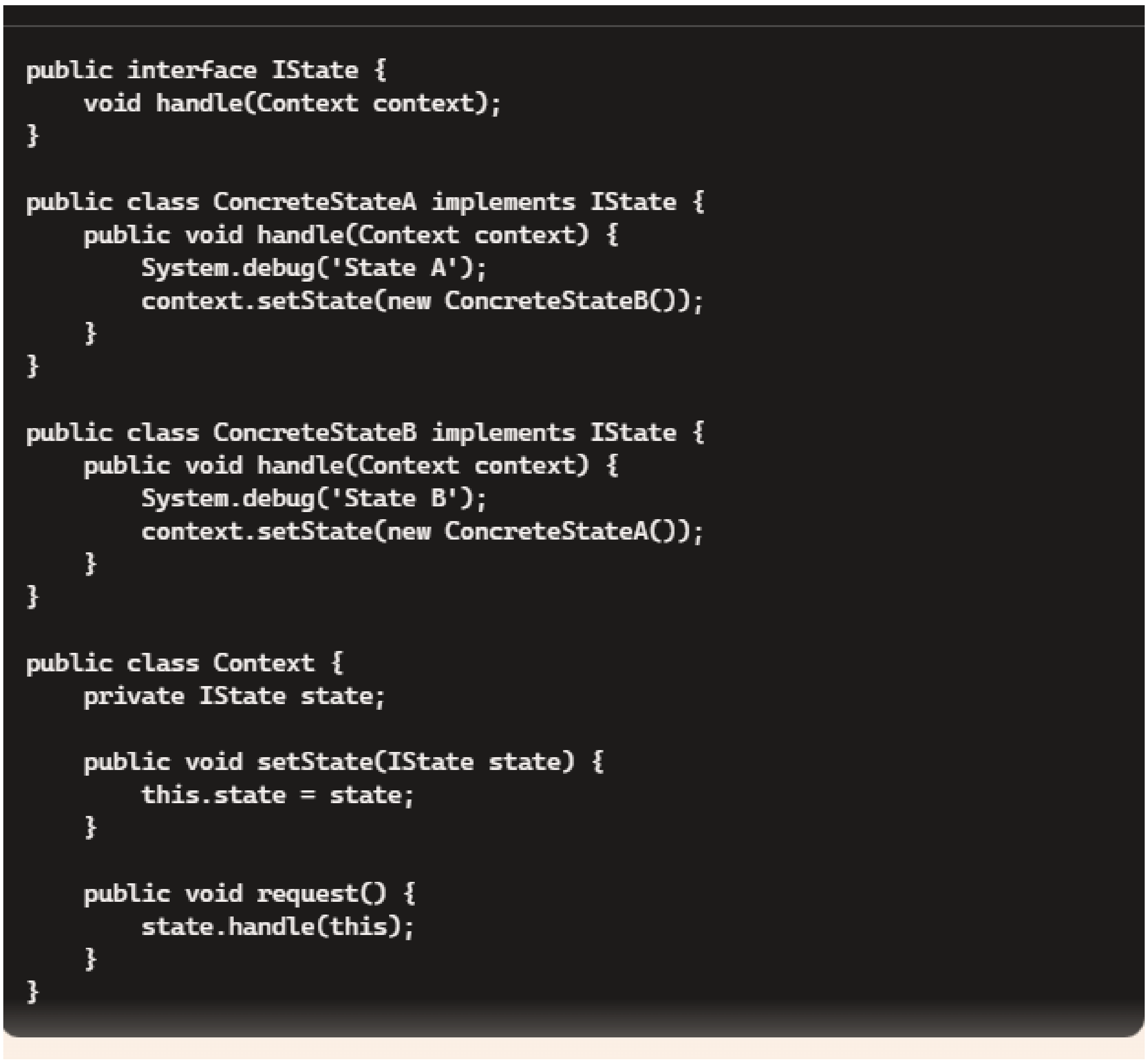
A behavioral design pattern that allows an object to alter its behavior when its internal state changes.

Purpose: Allow an object to alter its behavior when its internal state changes.

Usage: Used when an object's behavior depends on its state.

Problem Scenario: When an object's behavior depends on its state, implementing state-dependent behavior in a single class can lead to complex and hard-to-maintain code.

The Solution: Implementing State Pattern:



```
public interface IState {
    void handle(Context context);
}

public class ConcreteStateA implements IState {
    public void handle(Context context) {
        System.debug('State A');
        context.setState(new ConcreteStateB());
    }
}

public class ConcreteStateB implements IState {
    public void handle(Context context) {
        System.debug('State B');
        context.setState(new ConcreteStateA());
    }
}

public class Context {
    private IState state;

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

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

Advantages of State Pattern:

- Simplifies implementing state-dependent behavior.
- Decouples state-specific behavior from the context.
- Enhances flexibility and maintainability.

21. Strategy Pattern

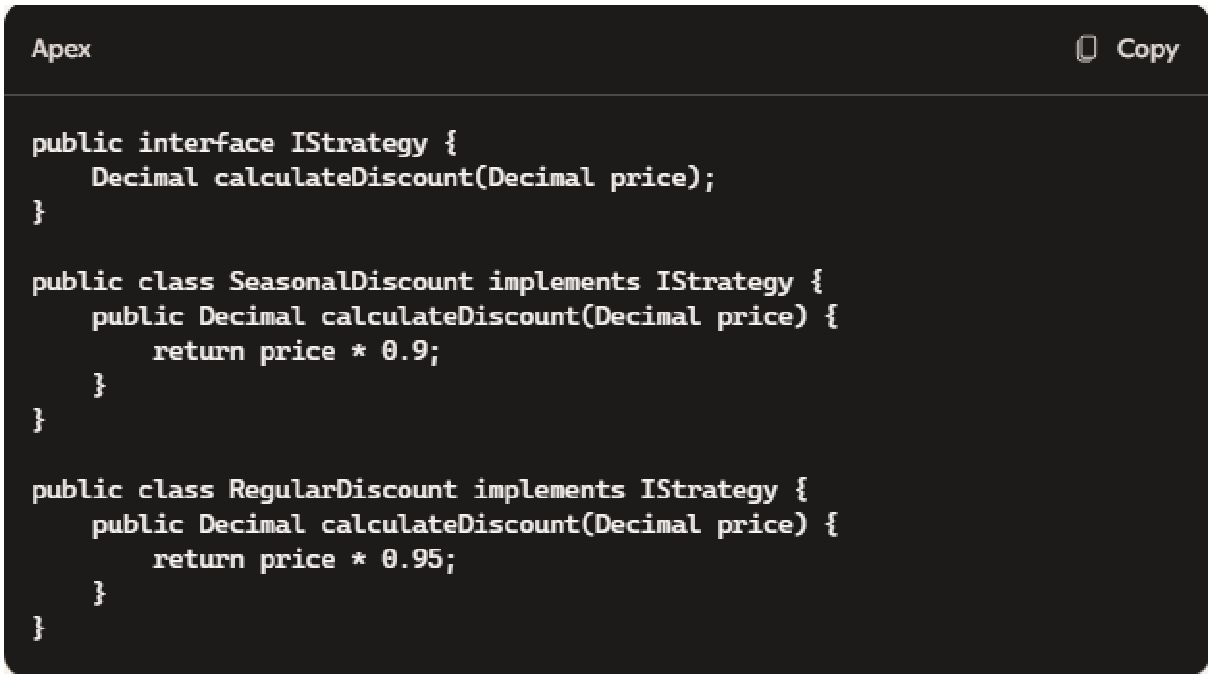
A behavioral design pattern that defines a family of algorithms, making them interchangeable.

Purpose: Define a family of algorithms, making them interchangeable.

Usage: Used to implement different algorithms based on the situation.

Problem Scenario: When implementing multiple algorithms, hardcoding the algorithm selection logic can lead to tight coupling and limited flexibility.

The Solution: Implementing Strategy Pattern:



```apex
public interface IStrategy {
    Decimal calculateDiscount(Decimal price);
}

public class SeasonalDiscount implements IStrategy {
    public Decimal calculateDiscount(Decimal price) {
        return price * 0.9;
    }
}

public class RegularDiscount implements IStrategy {
    public Decimal calculateDiscount(Decimal price) {
        return price * 0.95;
    }
}
```

Advantages of Strategy Pattern:

- **Flexibility**: Easily switch between different algorithms at runtime without modifying the client code.
- **Maintainability**: Encapsulate each algorithm in its own class, making the code easier to maintain and understand.
- **Reusability**: Reuse existing algorithms across different contexts or applications.
- **Extensibility**: Add new algorithms without changing the existing client code, adhering to the Open/Closed Principle.
- **Separation of Concerns**: Decouple the algorithm implementation from the context in which it's used, promoting cleaner code design.

22. Template Method Pattern

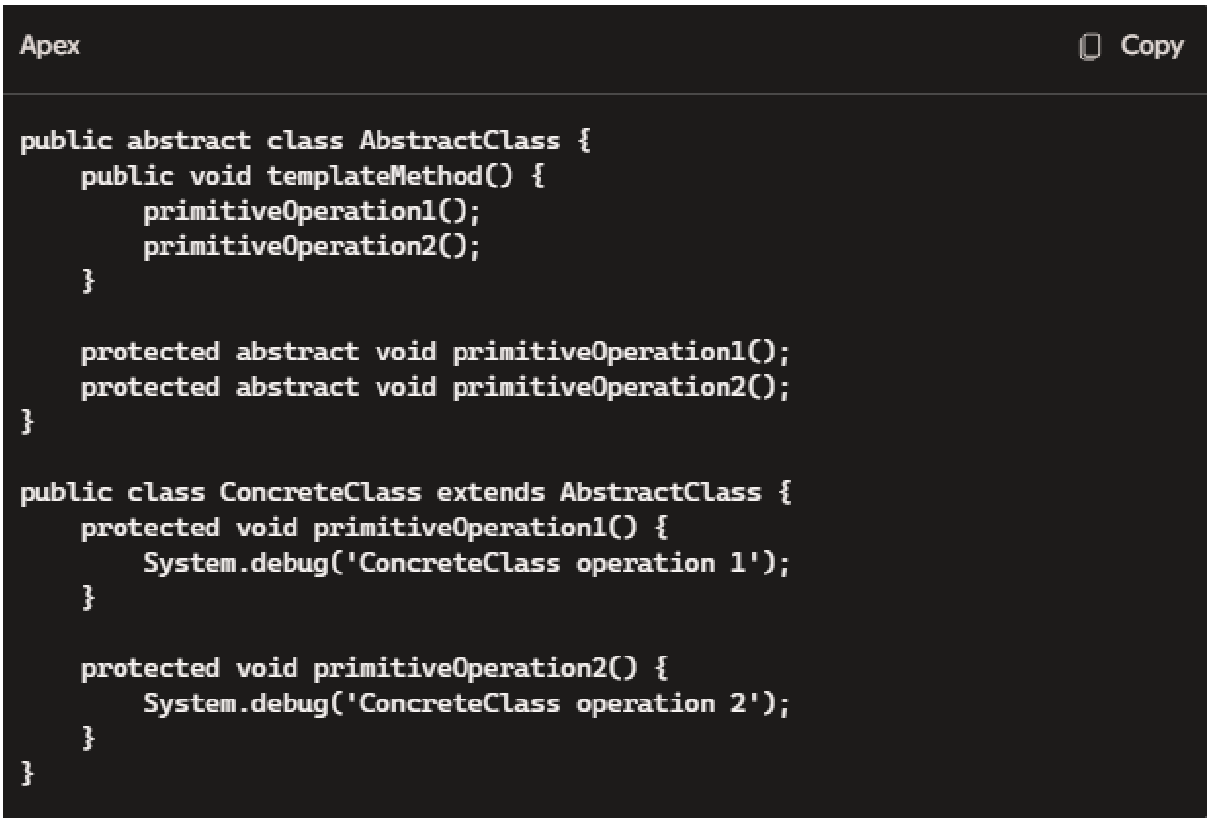
A behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

Purpose: Define the skeleton of an algorithm in a method, deferring some steps to subclasses.

Usage: Used to outline the steps of an algorithm while allowing subclasses to provide specific implementations.

Problem Scenario: When an algorithm consists of multiple steps, and the steps can vary among subclasses, hardcoding the entire algorithm can lead to duplicated code.

The Solution: Implementing Template Method Pattern:



```apex
public abstract class AbstractClass {
    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }

    protected abstract void primitiveOperation1();
    protected abstract void primitiveOperation2();
}

public class ConcreteClass extends AbstractClass {
    protected void primitiveOperation1() {
        System.debug('ConcreteClass operation 1');
    }

    protected void primitiveOperation2() {
        System.debug('ConcreteClass operation 2');
    }
}
```

Advantages of Template Method Pattern:

- Promotes code reuse by defining the common steps of an algorithm.
- Simplifies creating variations of an algorithm by overriding specific steps.
- Enhances flexibility and maintainability.

23. Visitor Pattern

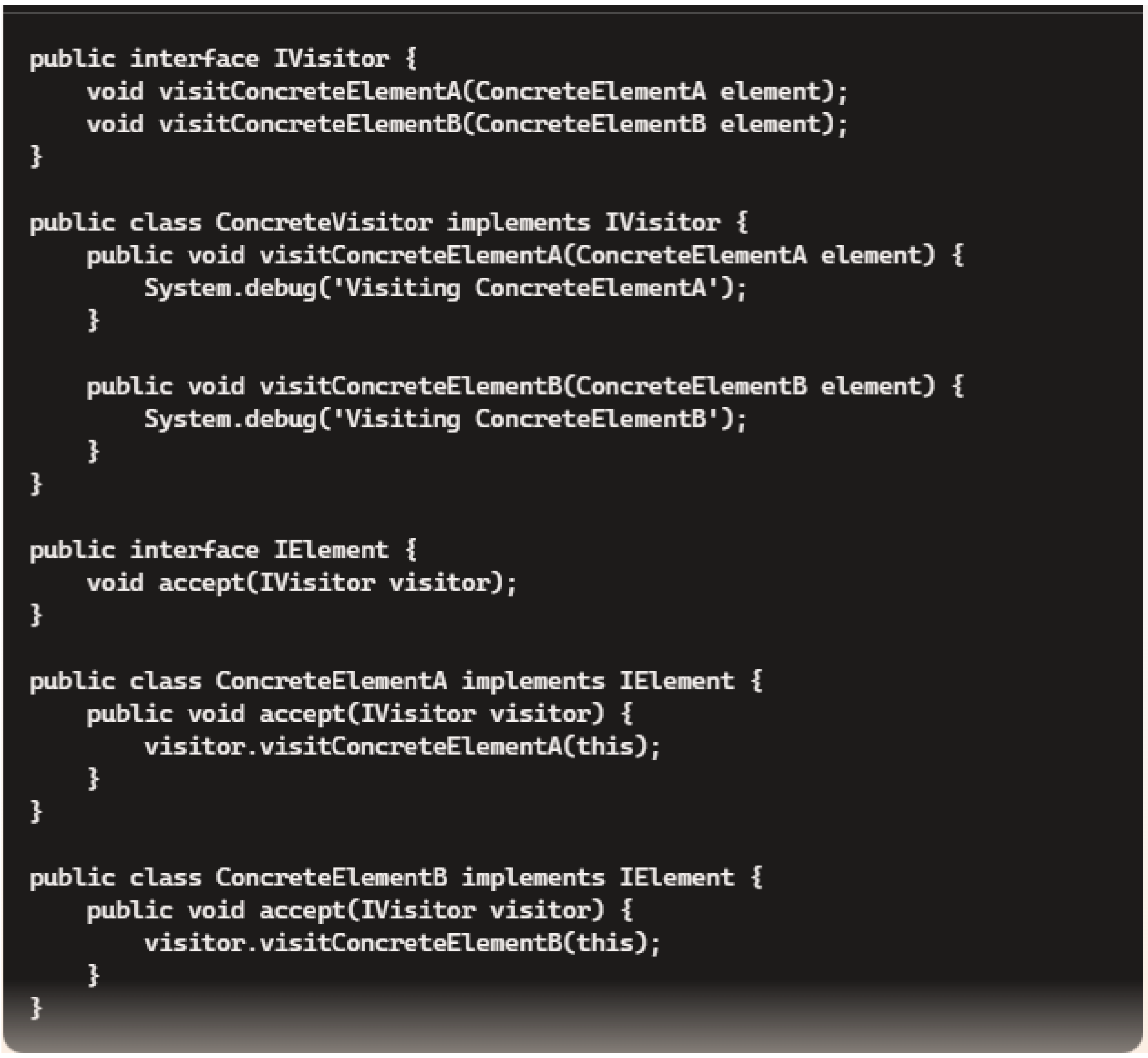
A behavioral design pattern that represents an operation to be performed on the elements of an object structure.

Purpose: Represent an operation to be performed on the elements of an object structure.

Usage: Used to add new operations to an existing object structure without modifying it.

Problem Scenario: When adding new operations to an existing object structure, modifying the structure classes can lead to tight coupling and maintenance challenges.

The Solution: Implementing Visitor Pattern:



```
public interface IVisitor {
    void visitConcreteElementA(ConcreteElementA element);
    void visitConcreteElementB(ConcreteElementB element);
}

public class ConcreteVisitor implements IVisitor {
    public void visitConcreteElementA(ConcreteElementA element) {
        System.debug('Visiting ConcreteElementA');
    }

    public void visitConcreteElementB(ConcreteElementB element) {
        System.debug('Visiting ConcreteElementB');
    }
}

public interface IElement {
    void accept(IVisitor visitor);
}

public class ConcreteElementA implements IElement {
    public void accept(IVisitor visitor) {
        visitor.visitConcreteElementA(this);
    }
}

public class ConcreteElementB implements IElement {
    public void accept(IVisitor visitor) {
        visitor.visitConcreteElementB(this);
    }
}
```

Advantages of Visitor Pattern:

- Simplifies adding new operations to an object structure.
- Promotes separation of concerns by isolating operations from the object structure.
- Enhances flexibility and maintainability.