Perfect 👌 — let's do something **practical** and educational:
we'll combine both **Design Pattern (DP)** and **Code Smell** concepts into one example.

🎯 Goal:

You'll see:

1. ❌ A **bad implementation** (with code smells)
2. ✅ A **refactored version** using a **Design Pattern** — specifically the **Strategy Pattern**

⚠️ Example 1: Code Smell (Bad Design)

Here's a "smelly" version of a payment processor:

```
// ❌ CODE SMELL VERSION
public class PaymentProcessor {
    public void pay(String method, double amount) {
        if (method.equals("credit")) {
            System.out.println("Connecting to bank...");
            System.out.println("Verifying credit card...");
            System.out.println("Charging $" + amount + " to Credit Card.");
        }
        else if (method.equals("paypal")) {
            System.out.println("Connecting to PayPal...");
            System.out.println("Logging in...");
            System.out.println("Transferring $" + amount + " via PayPal.");
        }
        else if (method.equals("bkash")) {
            System.out.println("Connecting to bKash...");
            System.out.println("Verifying mobile number...");
            System.out.println("Sending $" + amount + " via bKash.");
        }
        else {
            System.out.println("Invalid payment method!");
        }
    }
}
```

💀 Code Smells:

1. **Long if–else chain** → violates **Open–Closed Principle** (OCP).
2. **Hardcoded logic** for each payment type.
3. Difficult to add a new payment method (must modify the same class).
4. No polymorphism — breaks **Single Responsibility Principle (SRP)**.

✅ Example 1: Refactored Using Strategy Design Pattern

We'll fix it using polymorphism and interface-based strategies.

```
// Step 1: Define the Strategy Interface
interface PaymentStrategy {
    void pay(double amount);
}

// Step 2: Concrete Strategy Implementations
class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Connecting to bank...");
        System.out.println("Verifying credit card...");
        System.out.println("Charging $" + amount + " to Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Connecting to PayPal...");
        System.out.println("Logging in...");
        System.out.println("Transferring $" + amount + " via PayPal.");
    }
}

class BkashPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Connecting to bKash...");
        System.out.println("Verifying mobile number...");
        System.out.println("Sending $" + amount + " via bKash.");
    }
```

```
}

// Step 3: Context Class
class PaymentContext {
    private PaymentStrategy strategy;

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

    public void executePayment(double amount) {
        strategy.pay(amount);
    }
}

// Step 4: Demo
public class Main {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        // Credit Card
        context.setPaymentStrategy(new CreditCardPayment());
        context.executePayment(250.0);

        // PayPal
        context.setPaymentStrategy(new PayPalPayment());
        context.executePayment(125.0);

        // bKash
        context.setPaymentStrategy(new BkashPayment());
        context.executePayment(99.0);
    }
}
```

🧠 Design Pattern Used

Strategy Pattern — allows selecting an algorithm (payment type) at runtime.

Benefits:

- ✅ No long if–else chain
- ✅ Easy to add new payment types (no modification to core logic)
- ✅ Follows OCP & SRP

- ✅ Cleaner, testable, reusable code

💡 How This Fixes Code Smells

Code Smell	Before	After
Long if–else	Used	Replaced with polymorphism
Violation of OCP	Must edit core class	Just add new class
Hard-coded logic	In one big method	Each behavior isolated
Tight coupling	`PaymentProcessor` knows all types	Context only knows interface

---

Would you like me to give another one using a **different pattern** (like Factory Method or Observer) with its "code smell" version and fixed version next?

Excellent 🔥 — let's go for another one that shows **a real-world code smell** fixed by a **Design Pattern**.

This time, we'll use the **Factory Method Pattern** to clean up a "smelly" example.

⚠️ Example 2: Code Smell (Bad Design)

Imagine you have an app that sends **notifications** to users.
You started small — just Email — but later added SMS and Push.

Here's the messy version 👇

```
// ❌ CODE SMELL VERSION
public class NotificationService {

    public void sendNotification(String type, String message) {
        if (type.equals("email")) {
            System.out.println("Connecting to mail server...");
            System.out.println("Sending EMAIL: " + message);
        }
        else if (type.equals("sms")) {
            System.out.println("Connecting to SMS gateway...");
            System.out.println("Sending SMS: " + message);
        }
```

```
        else if (type.equals("push")) {
            System.out.println("Connecting to Push service...");
            System.out.println("Sending PUSH notification: " + message);
        }
        else {
            System.out.println("Invalid notification type!");
        }
    }
}
```

💀 Code Smells:

1. **Long if–else chain** → every time you add a new notification, you must modify this class.
2. **Violates Open–Closed Principle (OCP)** → not open for extension.
3. **Tight coupling** → one class knows all notification details.
4. **Hard to maintain or test**.

✅ Example 2: Refactored Using Factory Method Pattern

We'll create a hierarchy of notification classes and a factory to handle creation.

```
// Step 1: Common Interface
interface Notification {
    void send(String message);
}

// Step 2: Concrete Implementations
class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Connecting to mail server...");
        System.out.println("Sending EMAIL: " + message);
    }
}

class SMSNotification implements Notification {
    @Override
```

```
    public void send(String message) {
        System.out.println("Connecting to SMS gateway...");
        System.out.println("Sending SMS: " + message);
    }
}

class PushNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Connecting to Push service...");
        System.out.println("Sending PUSH notification: " + message);
    }
}

// Step 3: Factory Class
class NotificationFactory {
    public Notification createNotification(String type) {
        if (type == null || type.isEmpty()) return null;

        switch (type.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type: " + type);
        }
    }
}

// Step 4: Client Code
public class Main {
    public static void main(String[] args) {
        NotificationFactory factory = new NotificationFactory();

        Notification email = factory.createNotification("email");
        email.send("Welcome to GreenTop!");

        Notification sms = factory.createNotification("sms");
        sms.send("Your order has been shipped.");

        Notification push = factory.createNotification("push");
        push.send("You've earned 10 reward points!");
    }
}
```

🧠 Design Pattern Used

Factory Method Pattern — centralizes object creation logic.

✅ Benefits Over the Smelly Code

Problem	Before	After
If–else chain	Huge block inside service	Centralized in one factory
Adding new type	Modify core logic	Add a new class only
Tight coupling	`NotificationService` knew all types	Each class independent
Maintainability	Poor	Excellent (open/closed principle)

💡 Extension Example:

To add a **SlackNotification**, you just create a new class:

```
class SlackNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Connecting to Slack...");
        System.out.println("Sending SLACK message: " + message);
    }
}
```

…and **no changes** needed in the client code. ✅

Would you like the next example to use a **Behavioral Pattern** (like **Observer** or **Command**) with a code smell before-and-after view?

Perfect 😎 — let's do another **Design Pattern + Code Smell** example — this time using a **Behavioral Pattern**:
➡️ the **Observer Pattern** (great for notification, event handling, or UI updates).

We'll start with a *bad, tightly coupled version* and then refactor it to a clean, flexible Observer pattern implementation.

⚠️ Example 3: Code Smell (Bad Design)

Imagine a **YouTube Channel** system that notifies subscribers when a new video is uploaded.

Here's the *smelly* version 👇

```
// ❌ CODE SMELL VERSION
public class YouTubeChannel {
    private String channelName;

    public YouTubeChannel(String name) {
        this.channelName = name;
    }

    public void uploadVideo(String videoTitle) {
        System.out.println(channelName + " uploaded a new video: " + videoTitle);
        // Notify subscribers directly (hardcoded)
        System.out.println("Notifying Alice via Email...");
        System.out.println("Notifying Bob via SMS...");
        System.out.println("Notifying Charlie via Push...");
    }
}
```

💀 Code Smells:

1. **Tight coupling** – the channel directly knows every subscriber.
2. **Hard to extend** – adding a new subscriber type requires editing the class.
3. **Violates Open–Closed Principle (OCP)**.
4. **Not reusable** – notification logic and business logic mixed together.

✅ Example 3: Refactored Using Observer Design Pattern

We'll decouple the notification logic completely.

```
// Step 1: Observer Interface
interface Subscriber {
    void update(String videoTitle);
}

// Step 2: Concrete Observers
class EmailSubscriber implements Subscriber {
    private String name;
    public EmailSubscriber(String name) { this.name = name; }

    @Override
    public void update(String videoTitle) {
        System.out.println(name + " received EMAIL: New video uploaded - " + videoTitle);
    }
}

class SmsSubscriber implements Subscriber {
    private String name;
    public SmsSubscriber(String name) { this.name = name; }

    @Override
    public void update(String videoTitle) {
        System.out.println(name + " received SMS: New video uploaded - " + videoTitle);
    }
}

class PushSubscriber implements Subscriber {
    private String name;
    public PushSubscriber(String name) { this.name = name; }

    @Override
    public void update(String videoTitle) {
        System.out.println(name + " received PUSH notification: " + videoTitle);
    }
}

// Step 3: Subject (Observable)
import java.util.ArrayList;
import java.util.List;

class YouTubeChannel {
    private String channelName;
    private List<Subscriber> subscribers = new ArrayList<>();

    public YouTubeChannel(String name) {
        this.channelName = name;
```

```
    }

    public void subscribe(Subscriber sub) {
        subscribers.add(sub);
    }

    public void unsubscribe(Subscriber sub) {
        subscribers.remove(sub);
    }

    public void uploadVideo(String videoTitle) {
        System.out.println("\n" + channelName + " uploaded: " + videoTitle);
        notifySubscribers(videoTitle);
    }

    private void notifySubscribers(String videoTitle) {
        for (Subscriber sub : subscribers) {
            sub.update(videoTitle);
        }
    }
}

// Step 4: Demo
public class Main {
    public static void main(String[] args) {
        YouTubeChannel techChannel = new YouTubeChannel("TechWithAI");

        Subscriber alice = new EmailSubscriber("Alice");
        Subscriber bob = new SmsSubscriber("Bob");
        Subscriber charlie = new PushSubscriber("Charlie");

        techChannel.subscribe(alice);
        techChannel.subscribe(bob);
        techChannel.subscribe(charlie);

        techChannel.uploadVideo("Observer Pattern in Java");

        // Bob unsubscribes
        techChannel.unsubscribe(bob);
        techChannel.uploadVideo("Factory Pattern Explained!");
    }
}
```

🧠 Design Pattern Used

Observer Pattern — defines a one-to-many relationship between a subject (YouTubeChannel) and observers (subscribers).

When the subject changes (uploads a new video), all observers are automatically notified.

✅ Benefits (Compared to Smelly Version)

Problem	Before	After
Tight coupling	Hardcoded subscribers	Dynamically added/removed
OCP violation	Modify class for new type	Just add new `Subscriber`
Poor maintainability	Logic mixed together	Clear separation of concerns
Reusability	Low	High — reusable event model

💡 Real-life analogy:

This is how **event systems** in apps work:

- **Subject** = Publisher / Event Source
- **Observers** = Listeners / Subscribers

Used in:

- UI frameworks
- Event-driven architectures
- Notification systems
- Realtime updates

Would you like the **next example** to show a **Structural Pattern** (like **Decorator** or **Adapter**) with code smell and refactoring too?

Excellent — let's go deeper 💪

Now we'll move to a **Structural Design Pattern** → the **Decorator Pattern** — one of the most powerful and misunderstood ones.

We'll start with a *smelly* (bad) version that violates good design principles, and then refactor it beautifully using the **Decorator Pattern**.

⚠️ Example 4: Code Smell (Bad Design)

Imagine you're building a **coffee shop system** where customers can order coffee with optional add-ons like **milk**, **sugar**, and **whipped cream**.

Here's the **smelly version** 👇

```
// ❌ CODE SMELL VERSION
public class Coffee {
    private boolean milk;
    private boolean sugar;
    private boolean whippedCream;

    public Coffee(boolean milk, boolean sugar, boolean whippedCream) {
        this.milk = milk;
        this.sugar = sugar;
        this.whippedCream = whippedCream;
    }

    public double getCost() {
        double cost = 5.0; // base coffee price
        if (milk) cost += 1.0;
        if (sugar) cost += 0.5;
        if (whippedCream) cost += 1.5;
        return cost;
    }

    public String getDescription() {
        String desc = "Plain Coffee";
        if (milk) desc += " + Milk";
        if (sugar) desc += " + Sugar";
        if (whippedCream) desc += " + Whipped Cream";
        return desc;
    }
}
```

💀 Code Smells:

1. **Too many flags (boolean parameters)** — hard to maintain.

2. **Violated Open–Closed Principle** — adding new toppings requires changing this class.

3. **Combinatorial explosion** — new add-ons multiply the number of possible combinations.

4. **Hard to extend dynamically** (e.g., can't add new toppings at runtime).

✅ Example 4: Refactored Using Decorator Design Pattern

Let's fix it elegantly.
We'll use **composition** instead of booleans.

```
// Step 1: Base Component
interface Coffee {
    double getCost();
    String getDescription();
}

// Step 2: Concrete Component
class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 5.0;
    }

    @Override
    public String getDescription() {
        return "Plain Coffee";
    }
}

// Step 3: Abstract Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

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

    public double getCost() {
        return decoratedCoffee.getCost();
```

```
    }

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

// Step 4: Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 1.0;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + " + Milk";
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.5;
    }

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

class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 1.5;
```

```
    }

    @Override
    public String getDescription() {
        return super.getDescription() + " + Whipped Cream";
    }
}

// Step 5: Demo
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());

        // Add whipped cream
        coffee = new WhippedCreamDecorator(coffee);
        System.out.println(coffee.getDescription() + " → $" + coffee.getCost());
    }
}
```

🧠 Design Pattern Used

Decorator Pattern — dynamically adds new behavior to objects without modifying their code.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Boolean flags	Many	None
Violates OCP	Yes	No — new add-ons can be new decorators
Code duplication	High	Low
Extensibility	Poor	Excellent

Runtime flexibility	Impossible	Fully supported

☕ Output Example

Plain Coffee → $5.0
Plain Coffee + Milk → $6.0
Plain Coffee + Milk + Sugar → $6.5
Plain Coffee + Milk + Sugar + Whipped Cream → $8.0

💡 Real-World Examples of Decorator Pattern

- Java I/O Streams (`BufferedReader`, `FileReader`, etc.)
- UI frameworks (adding visual effects dynamically)
- Middleware layers (logging, caching, compression decorators)

Would you like me to give you the **next example** using the **Adapter Pattern** (also a structural one) — with its "smelly" version and refactored version?

Awesome 👏 You're really getting the hang of this — let's now go for another **Structural Design Pattern** example:

➡️ The **Adapter Pattern** — a pattern used to connect **incompatible interfaces** (a common real-world issue).

We'll again start with a **code smell version**, then refactor it beautifully using the **Adapter Pattern**.

⚠️ Example 5: Code Smell (Bad Design)

You're building a system that needs to **charge devices**.
Initially, you only had an **Android charger**, but now you also need to charge **iPhones** — and the interfaces don't match.

Here's the messy, smelly version 👇

```
// ❌ CODE SMELL VERSION
class AndroidCharger {
    public void chargeAndroidPhone() {
```

```
        System.out.println("Charging Android phone with USB-C...");
    }
}

class IphoneCharger {
    public void chargeIphone() {
        System.out.println("Charging iPhone with Lightning connector...");
    }
}

// Bad design: client hardcodes compatibility checks
public class ChargerService {
    public void chargePhone(String type) {
        if (type.equalsIgnoreCase("android")) {
            AndroidCharger androidCharger = new AndroidCharger();
            androidCharger.chargeAndroidPhone();
        } else if (type.equalsIgnoreCase("iphone")) {
            IphoneCharger iphoneCharger = new IphoneCharger();
            iphoneCharger.chargeIphone();
        } else {
            System.out.println("Unknown device type!");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ChargerService service = new ChargerService();
        service.chargePhone("android");
        service.chargePhone("iphone");
    }
}
```

💀 Code Smells:

1. **Tight coupling** – `ChargerService` knows all charger types.
2. **If–else hell** – must edit every time a new phone type appears.
3. **Violation of Open–Closed Principle (OCP)** – not open for extension.
4. **Incompatible interfaces** handled manually — not scalable.

✅ Example 5: Refactored Using Adapter Pattern

Let's clean this up beautifully by introducing a **unified interface** and an **adapter** to bridge incompatibility.

```
// Step 1: Target Interface
interface PhoneCharger {
    void charge();
}

// Step 2: Adaptee Classes (existing incompatible classes)
class AndroidCharger {
    public void chargeAndroidPhone() {
        System.out.println("Charging Android phone with USB-C...");
    }
}

class IphoneCharger {
    public void chargeIphone() {
        System.out.println("Charging iPhone with Lightning connector...");
    }
}

// Step 3: Adapters (bridge between incompatible interfaces)
class AndroidChargerAdapter implements PhoneCharger {
    private AndroidCharger androidCharger;

    public AndroidChargerAdapter(AndroidCharger androidCharger) {
        this.androidCharger = androidCharger;
    }

    @Override
    public void charge() {
        androidCharger.chargeAndroidPhone();
    }
}

class IphoneChargerAdapter implements PhoneCharger {
    private IphoneCharger iphoneCharger;

    public IphoneChargerAdapter(IphoneCharger iphoneCharger) {
        this.iphoneCharger = iphoneCharger;
    }

    @Override
    public void charge() {
```

```
        iphoneCharger.chargeIphone();
    }
}

// Step 4: Client (no need to know internal details)
public class Main {
    public static void main(String[] args) {
        PhoneCharger androidAdapter = new AndroidChargerAdapter(new AndroidCharger());
        PhoneCharger iphoneAdapter = new IphoneChargerAdapter(new IphoneCharger());

        System.out.println("=== Charging Devices ===");
        androidAdapter.charge();
        iphoneAdapter.charge();
    }
}
```

🧠 Design Pattern Used

Adapter Pattern — converts one interface into another interface that the client expects.

It's like using a **plug adapter** when traveling:
⚡ You don't change your device, you just make it compatible.

✅ Benefits Compared to Smelly Version

Problem	Before	After
If–else chains	Everywhere	Removed
Tight coupling	High	Low — adapters handle differences
OCP violation	Must modify class	Just add a new adapter
Reusability	Poor	Excellent
Maintenance	Hard	Simple

⚙️ Output Example

```
=== Charging Devices ===
Charging Android phone with USB-C...
Charging iPhone with Lightning connector...
```

💡 Real-World Uses of Adapter Pattern

- Connecting APIs with different interfaces
- Making legacy code compatible with new systems
- Java I/O Streams (`InputStreamReader` adapts `InputStream` → `Reader`)
- Database driver adapters

Would you like the **next example** to demonstrate a **Behavioral Pattern** like **Command** or **State**, again with code smell and refactoring?

Perfect 🔥 You're mastering these —
let's now explore another **Behavioral Design Pattern** with a real **Code Smell → Refactored** example.

This time, we'll use the **Command Pattern**, which is great for **undo/redo**, **menu actions**, or **remote control systems**.

⚠️ Example 6: Code Smell (Bad Design)

Imagine you're writing code for a **Smart Home Remote** that can **turn lights on/off**, **turn TV on/off**, etc.
At first, you handle everything with `if-else` statements. 😬

```
// ❌ CODE SMELL VERSION
public class RemoteControl {
    public void pressButton(String action) {
        if (action.equalsIgnoreCase("light_on")) {
            System.out.println("Light is turned ON");
        } else if (action.equalsIgnoreCase("light_off")) {
            System.out.println("Light is turned OFF");
        } else if (action.equalsIgnoreCase("tv_on")) {
            System.out.println("TV is turned ON");
        } else if (action.equalsIgnoreCase("tv_off")) {
            System.out.println("TV is turned OFF");
        } else {
            System.out.println("Unknown action!");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        remote.pressButton("light_on");
        remote.pressButton("tv_off");
    }
}
```

💀 Code Smells:

1. **Massive if–else chain** — not scalable.
2. **Violation of Open–Closed Principle (OCP)** — adding a new device means editing this class.
3. **No abstraction** — RemoteControl directly knows device details.
4. **Hard to reuse commands or implement undo/redo**.

✅ Example 6: Refactored Using Command Design Pattern

We'll create **command objects** for each action and make the remote **agnostic** of the specific device.

```
// Step 1: Command Interface
interface Command {
    void execute();
}

// Step 2: Receiver Classes
class Light {
    public void turnOn() {
        System.out.println("Light is turned ON");
    }
    public void turnOff() {
        System.out.println("Light is turned OFF");
    }
}

class TV {
```

```
    public void turnOn() {
        System.out.println("TV is turned ON");
    }
    public void turnOff() {
        System.out.println("TV is turned OFF");
    }
}

// Step 3: Concrete Command Classes
class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) { this.light = light; }

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

class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) { this.light = light; }

    @Override
    public void execute() {
        light.turnOff();
    }
}

class TVOnCommand implements Command {
    private TV tv;
    public TVOnCommand(TV tv) { this.tv = tv; }

    @Override
    public void execute() {
        tv.turnOn();
    }
}

class TVOffCommand implements Command {
    private TV tv;
    public TVOffCommand(TV tv) { this.tv = tv; }

    @Override
    public void execute() {
        tv.turnOff();
    }
}
```

```
// Step 4: Invoker
class RemoteControl {
    private Command command;

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

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

// Step 5: Client Code
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        TV tv = new TV();

        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);
        Command tvOn = new TVOnCommand(tv);
        Command tvOff = new TVOffCommand(tv);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);
        remote.pressButton();

        remote.setCommand(tvOn);
        remote.pressButton();

        remote.setCommand(lightOff);
        remote.pressButton();

        remote.setCommand(tvOff);
        remote.pressButton();
    }
}
```

🧠 Design Pattern Used

Command Pattern — turns a request (like “turn light on”) into a standalone object.
This allows you to parameterize invokers, queue commands, or implement undo operations.

✅ Benefits Compared to Smelly Version

Problem	Before	After
If–else logic	Hardcoded	Removed
OCP violation	Modify class for new action	Add new Command class
Tight coupling	High	Low
Reusability	Poor	Excellent
Undo/Redo	Impossible	Easy to add later

⚙️ Output Example

Light is turned ON
TV is turned ON
Light is turned OFF
TV is turned OFF

💡 Real-World Examples of Command Pattern

- GUI buttons & menus (each menu item = command)
- Game engines (record/undo actions)
- Smart home systems
- Task queues (store & execute commands later)

Would you like the next one to show the **State Pattern** (another behavioral one, great for finite state machines, login systems, or media players)?

Perfect — you're doing great 👏
Now let's explore another **Behavioral Design Pattern** — the **State Pattern** — with a clear "code smell → refactor" journey.

The **State Pattern** is used when an object's behavior **changes based on its internal state** — without using messy `if-else` or `switch` statements.

⚠️ Example 7: Code Smell (Bad Design)

Imagine you're building a **music player** app.
It can be in one of three states:

- **Stopped**
- **Playing**
- **Paused**

Here's the *smelly* version with big `if-else` logic inside methods 👇

```
// ❌ CODE SMELL VERSION

public class MusicPlayer {

    private String state = "stopped";


    public void play() {

        if (state.equals("stopped")) {

            System.out.println("Starting music...");

            state = "playing";

        } else if (state.equals("playing")) {

            System.out.println("Music is already playing.");

        } else if (state.equals("paused")) {

            System.out.println("Resuming music...");

            state = "playing";

        }

    }


    public void pause() {
```

```
        if (state.equals("playing")) {
            System.out.println("Music paused.");
            state = "paused";
        } else if (state.equals("paused")) {
            System.out.println("Already paused.");
        } else {
            System.out.println("Can't pause. Music not playing.");
        }
    }

    public void stop() {
        if (!state.equals("stopped")) {
            System.out.println("Music stopped.");
            state = "stopped";
        } else {
            System.out.println("Music already stopped.");
        }
    }
}
```

💀 Code Smells:

1. **Too many if–else statements** → breaks OCP.
2. **State logic scattered everywhere.**
3. **Adding a new state (e.g., “fast forward”) means editing every method.**

4. **Violates Single Responsibility Principle.**

✅ Example 7: Refactored Using State Design Pattern

We'll encapsulate each state's behavior into its own class.
Now the `MusicPlayer` will **delegate** behavior to the current state object.

```
// Step 1: Define the State Interface

interface PlayerState {

    void play(MusicPlayer player);

    void pause(MusicPlayer player);

    void stop(MusicPlayer player);

}


// Step 2: Concrete States

class StoppedState implements PlayerState {

    @Override

    public void play(MusicPlayer player) {

        System.out.println("Starting music...");

        player.setState(new PlayingState());

    }


    @Override

    public void pause(MusicPlayer player) {

        System.out.println("Can't pause — music is stopped.");

    }
```

```
    @Override
    public void stop(MusicPlayer player) {
        System.out.println("Music already stopped.");
    }
}

class PlayingState implements PlayerState {
    @Override
    public void play(MusicPlayer player) {
        System.out.println("Music is already playing.");
    }

    @Override
    public void pause(MusicPlayer player) {
        System.out.println("Pausing music...");
        player.setState(new PausedState());
    }

    @Override
    public void stop(MusicPlayer player) {
        System.out.println("Stopping music...");
        player.setState(new StoppedState());
    }
}

class PausedState implements PlayerState {
```

```
    @Override
    public void play(MusicPlayer player) {
        System.out.println("Resuming music...");
        player.setState(new PlayingState());
    }

    @Override
    public void pause(MusicPlayer player) {
        System.out.println("Music already paused.");
    }

    @Override
    public void stop(MusicPlayer player) {
        System.out.println("Stopping music from pause...");
        player.setState(new StoppedState());
    }
}

// Step 3: Context Class
class MusicPlayer {
    private PlayerState state;

    public MusicPlayer() {
        state = new StoppedState(); // Default state
    }
```

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

    public void play() {
        state.play(this);
    }

    public void pause() {
        state.pause(this);
    }

    public void stop() {
        state.stop(this);
    }
}

// Step 4: Demo
public class Main {
    public static void main(String[] args) {
        MusicPlayer player = new MusicPlayer();

        player.play();   // Start music
        player.pause();  // Pause music
        player.play();   // Resume
        player.stop();   // Stop
```

```
        player.stop();   // Already stopped
    }
}
```

🧠 Design Pattern Used

State Pattern — allows an object to change its behavior dynamically when its internal state changes.

Instead of using if–else, each state class defines its own behavior.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Long if–else	Everywhere	Gone
Violation of OCP	Yes	No — add new states easily
Mixed responsibilities	State + Logic tangled	Separated
Code readability	Poor	Very clean
Maintenance	Hard	Simple and extensible

⚙️ Output Example

Starting music...

Pausing music...

Resuming music...

Stopping music...

Music already stopped.

💡 Real-World Uses of the State Pattern

- **Media players** (Play/Pause/Stop)
- **Vending machines** (Idle → CoinInserted → Dispense → Empty)
- **ATMs** (CardInserted → PinEntered → Transaction)
- **Game characters** (Idle → Running → Jumping)

Would you like me to give the **next example** with the **Template Method Pattern** (another behavioral one used in frameworks and abstract class designs)?

Awesome 😎 You're mastering design patterns step by step.
Now let's go for another — this time a **Creational Design Pattern** — the **Builder Pattern**, which is used to construct **complex objects step by step** and avoid **Telescoping Constructor** code smell.

⚠️ Example 8: Code Smell (Telescoping Constructor Problem)

Let's say you have a class `Computer` with many optional attributes.

Here's the **bad approach** 👇

```
// ❌ CODE SMELL VERSION
class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;
```

```
    private boolean hasWiFi;
    private boolean hasBluetooth;

    // Telescoping constructor — too many params!
    public Computer(String CPU, String GPU, int RAM, int storage, boolean hasWiFi, boolean
hasBluetooth) {
        this.CPU = CPU;
        this.GPU = GPU;
        this.RAM = RAM;
        this.storage = storage;
        this.hasWiFi = hasWiFi;
        this.hasBluetooth = hasBluetooth;
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", GPU=" + GPU + ", RAM=" + RAM +
                "GB, Storage=" + storage + "GB, WiFi=" + hasWiFi + ", Bluetooth=" + hasBluetooth
+ "]";
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        // Hard to read and maintain
        Computer pc = new Computer("Intel i7", "RTX 4060", 16, 512, true, false);
        System.out.println(pc);
    }
}
```

💀 Code Smells:

1. **Telescoping constructors** — too many parameters in a long list.
2. **Unreadable and error-prone** (easy to mix up arguments).
3. **Violates Single Responsibility** — constructor does both setup and validation.
4. **Hard to add new attributes** (must modify every constructor call).

✅ Example 8: Refactored Using Builder Pattern

We'll separate object creation logic into a **Builder class**, which sets each attribute fluently.

```
// Step 1: Product class
class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;
    private boolean hasWiFi;
    private boolean hasBluetooth;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.GPU = builder.GPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.hasWiFi = builder.hasWiFi;
        this.hasBluetooth = builder.hasBluetooth;
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", GPU=" + GPU + ", RAM=" + RAM +
                "GB, Storage=" + storage + "GB, WiFi=" + hasWiFi + ", Bluetooth=" + hasBluetooth
+ "]";
    }

    // Step 2: Builder Inner Class
    public static class Builder {
        private String CPU;
        private String GPU;
        private int RAM;
        private int storage;
        private boolean hasWiFi;
        private boolean hasBluetooth;

        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public Builder setGPU(String GPU) {
            this.GPU = GPU;
            return this;
        }
```

```
        public Builder setRAM(int RAM) {
            this.RAM = RAM;
            return this;
        }

        public Builder setStorage(int storage) {
            this.storage = storage;
            return this;
        }

        public Builder enableWiFi(boolean hasWiFi) {
            this.hasWiFi = hasWiFi;
            return this;
        }

        public Builder enableBluetooth(boolean hasBluetooth) {
            this.hasBluetooth = hasBluetooth;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}

// Step 3: Client code
public class Main {
    public static void main(String[] args) {
        Computer gamingPC = new Computer.Builder()
                .setCPU("Intel i9")
                .setGPU("RTX 4090")
                .setRAM(32)
                .setStorage(1024)
                .enableWiFi(true)
                .enableBluetooth(true)
                .build();

        System.out.println(gamingPC);
    }
}
```

🧠 Design Pattern Used

Builder Pattern — separates the construction of a complex object from its representation, allowing step-by-step creation with readable syntax.

✅ Benefits

Problem	Before	After
Too many constructor params	Yes	No
Hard to read	Yes	Fluent and readable
Inflexible	Yes	Easy to extend
Mixed logic	Yes	Clean separation
OCP violation	Yes	No

⚙️ Output Example

Computer [CPU=Intel i9, GPU=RTX 4090, RAM=32GB, Storage=1024GB, WiFi=true, Bluetooth=true]

💡 Real-World Uses

- `StringBuilder` in Java
- `AlertDialog.Builder` in Android
- `HttpClient.Builder` in networking
- `Lombok @Builder` annotation
- Any system with **many optional parameters**

Would you like the **next one** to be:

- 🔹 a **Behavioral Pattern** (like *Observer* or *Strategy*), or
- 🔹 a **Structural Pattern** (like *Adapter* or *Composite*)?

  Perfect! Let's do **another example**, this time a **Creational Pattern** — the **Singleton Pattern**.

The **Singleton Pattern** is used when you need **only one instance of a class** throughout your application. A common code smell it fixes is **“global/static objects misused”**.

⚠️ Example 9: Code Smell (Bad Design)

Suppose you are creating a **Logger** class.
The naive approach allows multiple instances:

```
// ❌ CODE SMELL VERSION
class Logger {
    public Logger() {
        // Constructor
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}

public class Main {
    public static void main(String[] args) {
        Logger logger1 = new Logger();
        Logger logger2 = new Logger();

        logger1.log("First message");
        logger2.log("Second message");

        System.out.println(logger1 == logger2); // false — multiple instances exist
    }
}
```

💀 Code Smells:

1. **Multiple instances allowed** — can lead to inconsistent logging.
2. **Global state mismanagement** — two objects can log independently.
3. **Violates Single Responsibility Principle** if other code tries to manage the instance.
4. **Hard to control resource usage** — e.g., file handles in real loggers.

✅ Example 9: Refactored Using Singleton Pattern

We'll ensure **only one instance exists**:

```
// Step 1: Singleton Logger
class Logger {
    // Private static instance
    private static Logger instance;

    // Private constructor
    private Logger() { }

    // Public method to get instance
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}

// Step 2: Client
public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("First message");
        logger2.log("Second message");

        System.out.println(logger1 == logger2); // true — only one instance exists
    }
}
```

🧠 Design Pattern Used

Singleton Pattern — ensures only one instance of a class exists and provides a global point of access.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Multiple instances	Yes	No — only one instance
Global state mismanagement	Possible	Controlled
Resource handling	Risky	Safe (single instance)
Consistency	Low	High

⚙️ Output Example

- Log: First message
- Log: Second message
- true

💡 Real-World Uses

- Logger classes
- Database connection pools
- Configuration managers
- Thread pools

If you want, I can next give a **Prototype Pattern** example, which is another **creational pattern** solving the “expensive object creation” code smell.

Do you want me to do that next?

Perfect! Let’s go for **another Creational Pattern** — this time the **Prototype Pattern**.

The **Prototype Pattern** is useful when **creating a new object is expensive** (time or resources). It fixes the **code smell** of repeatedly creating objects with the same configuration.

⚠️ Example 10: Code Smell (Bad Design)

Suppose you are creating **multiple user profiles** with default settings.
Each time, you manually set all the fields:

```
// ❌ CODE SMELL VERSION
class UserProfile {
    String name;
    String role;
    boolean notificationsEnabled;
    int theme; // 0 = light, 1 = dark

    public UserProfile(String name, String role, boolean notificationsEnabled, int theme) {
        this.name = name;
        this.role = role;
        this.notificationsEnabled = notificationsEnabled;
        this.theme = theme;
    }

    public void display() {
        System.out.println(name + " | " + role + " | Notifications: " + notificationsEnabled + " |
Theme: " + theme);
    }
}

public class Main {
    public static void main(String[] args) {
        UserProfile user1 = new UserProfile("Alice", "Admin", true, 1);
        UserProfile user2 = new UserProfile("Bob", "Admin", true, 1);
        UserProfile user3 = new UserProfile("Charlie", "Admin", true, 1);

        user1.display();
        user2.display();
```

```
        user3.display();
    }
}
```

💀 Code Smells:

1. **Duplicate object creation logic** — all Admin users have the same settings.
2. **Hard to maintain** — changes in defaults require editing multiple places.
3. **Performance hit** if object creation is expensive.

✅ Example 10: Refactored Using Prototype Pattern

We'll make the `UserProfile` class **cloneable** so we can copy an existing object:

```
// Step 1: Prototype Class
class UserProfile implements Cloneable {
    String name;
    String role;
    boolean notificationsEnabled;
    int theme;

    public UserProfile(String name, String role, boolean notificationsEnabled, int theme) {
        this.name = name;
        this.role = role;
        this.notificationsEnabled = notificationsEnabled;
        this.theme = theme;
    }

    // Step 2: Implement clone method
    public UserProfile cloneProfile(String newName) {
        try {
            UserProfile clone = (UserProfile) this.clone();
            clone.name = newName; // change only the name
            return clone;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

    public void display() {
```

```
        System.out.println(name + " | " + role + " | Notifications: " + notificationsEnabled + " |
Theme: " + theme);
    }
}

// Step 3: Client
public class Main {
    public static void main(String[] args) {
        // Original prototype
        UserProfile adminPrototype = new UserProfile("Prototype", "Admin", true, 1);

        // Create clones
        UserProfile user1 = adminPrototype.cloneProfile("Alice");
        UserProfile user2 = adminPrototype.cloneProfile("Bob");
        UserProfile user3 = adminPrototype.cloneProfile("Charlie");

        user1.display();
        user2.display();
        user3.display();
    }
}
```

🧠 Design Pattern Used

Prototype Pattern — creates new objects by cloning an existing object instead of constructing from scratch.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Duplicate object creation	Yes	No — clone prototype
Maintainability	Hard	Easy — change default in one place
Performance	Slower if construction is expensive	Faster — clone existing object
Flexibility	Low	High — modify only the fields needed

⚙️ Output Example

Alice | Admin | Notifications: true | Theme: 1
Bob | Admin | Notifications: true | Theme: 1
Charlie | Admin | Notifications: true | Theme: 1

💡 Real-World Uses

- Game development (clone enemies or characters)
- Document templates
- GUI components with default settings
- Expensive object initialization (e.g., database connections, large datasets)

If you want, I can next give **another Behavioral Pattern** like **Observer** or **Mediator**, which is very common in event-driven programming.

Do you want me to do that next?

Perfect! Let’s go for another **Structural Design Pattern** — this time the **Facade Pattern**.

The **Facade Pattern** is used to **simplify complex subsystems** by providing a unified interface. It fixes the **code smell of “too many dependencies / tightly coupled code”**, where the client has to interact with many classes directly.

⚠️ Example 11: Code Smell (Bad Design)

Suppose you’re building a **home theater system**.

Without a facade, the client has to interact with each component separately:

```java
// ❌ CODE SMELL VERSION
class Amplifier {
    public void on() { System.out.println("Amplifier on"); }
    public void off() { System.out.println("Amplifier off"); }
}

class DVDPlayer {
    public void on() { System.out.println("DVD Player on"); }
    public void play(String movie) { System.out.println("Playing " + movie); }
    public void off() { System.out.println("DVD Player off"); }
}

class Projector {
    public void on() { System.out.println("Projector on"); }
    public void off() { System.out.println("Projector off"); }
}

public class Main {
    public static void main(String[] args) {
        Amplifier amp = new Amplifier();
        DVDPlayer dvd = new DVDPlayer();
        Projector projector = new Projector();

        amp.on();
        dvd.on();
```

```
        dvd.play("Inception");
        projector.on();

        // later turn off
        dvd.off();
        projector.off();
        amp.off();
    }
}
```

💀 Code Smells:

1. **Client knows too much about subsystem** — tight coupling.
2. **Repeated complex setup logic** in multiple places.
3. **Hard to maintain** — adding/removing devices breaks client code.
4. **Violates Law of Demeter** — client should not interact with so many classes directly.

✅ Example 11: Refactored Using **Facade Pattern**

We create a **single Facade class** that simplifies the interface:

```java
// Step 1: Subsystem Classes
class Amplifier {
    public void on() { System.out.println("Amplifier on"); }
    public void off() { System.out.println("Amplifier off"); }
}

class DVDPlayer {
    public void on() { System.out.println("DVD Player on"); }
    public void play(String movie) { System.out.println("Playing " + movie); }
    public void off() { System.out.println("DVD Player off"); }
}

class Projector {
    public void on() { System.out.println("Projector on"); }
    public void off() { System.out.println("Projector off"); }
}

// Step 2: Facade
class HomeTheaterFacade {
    private Amplifier amp;
    private DVDPlayer dvd;
    private Projector projector;

    public HomeTheaterFacade(Amplifier amp, DVDPlayer dvd, Projector projector) {
        this.amp = amp;
```

```
        this.dvd = dvd;
        this.projector = projector;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        amp.on();
        dvd.on();
        projector.on();
        dvd.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting movie theater down...");
        dvd.off();
        projector.off();
        amp.off();
    }
}

// Step 3: Client
public class Main {
    public static void main(String[] args) {
        Amplifier amp = new Amplifier();
        DVDPlayer dvd = new DVDPlayer();
        Projector projector = new Projector();
```

```
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, dvd, projector);
        homeTheater.watchMovie("Inception");
        System.out.println("--- Movie finished ---");
        homeTheater.endMovie();
    }
}
```

🧠 Design Pattern Used

Facade Pattern — provides a simplified interface to a complex subsystem, reducing coupling and improving maintainability.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Tight coupling	High — client interacts with all subsystems	Low — client interacts only with Facade
Repeated setup logic	Yes	Encapsulated in Facade
Maintainability	Hard	Easy — change subsystem internally without affecting client

| Readability | Low | High — simple, clear methods |

⚙️ Output Example

```
Get ready to watch a movie...
Amplifier on
DVD Player on
Projector on
Playing Inception
--- Movie finished ---
Shutting movie theater down...
DVD Player off
Projector off
Amplifier off
```

💡 Real-World Uses

* **JDBC** — `DataSource` simplifies DB connection setup
* **Spring's RestTemplate / WebClient** — simplifies HTTP calls
* **UI frameworks** — one API to control many components

* **Home automation / smart devices**

If you want, I can next give another **Structural Pattern** like **Composite**, which is very common for **tree-like structures** (e.g., file systems, UI components).

Do you want me to do that next?

Perfect! Let's go for another **Structural Design Pattern** — this time the **Composite Pattern**.

The **Composite Pattern** is used to treat **individual objects and compositions of objects uniformly**, which fixes the **code smell of repeated tree traversal logic** or **nested `if-else` for hierarchy**.

⚠️ Example 12: Code Smell (Bad Design)

Suppose you are building a **file system** with **files** and **folders**.

A naive approach might use different methods for folders and files:

```
// ❌ CODE SMELL VERSION
class File {
    private String name;
    public File(String name) { this.name = name; }
    public void showDetails() { System.out.println("File: " + name); }
}

class Folder {
    private String name;
    private List<File> files = new ArrayList<>();
```

```
    public Folder(String name) { this.name = name; }
    public void addFile(File f) { files.add(f); }
    public void showDetails() {
        System.out.println("Folder: " + name);
        for(File f : files) f.showDetails();
    }
}

public class Main {
    public static void main(String[] args) {
        File file1 = new File("file1.txt");
        File file2 = new File("file2.txt");

        Folder folder1 = new Folder("Documents");
        folder1.addFile(file1);
        folder1.addFile(file2);

        folder1.showDetails();
    }
}
```

💀 Code Smells:

1. **Duplicate traversal logic** — each folder needs its own loop.
2. **Hard to add nested folders** — need special handling.

3. **Tightly coupled client** — must know whether it's a file or folder.
4. **Violates Open/Closed Principle** — adding new component types requires modifying client.

✅ Example 12: Refactored Using Composite Pattern

We'll create a **common interface** for files and folders so they can be treated uniformly:

```
import java.util.*;


// Step 1: Component
interface FileSystemComponent {
    void showDetails();
}


// Step 2: Leaf
class File implements FileSystemComponent {
    private String name;
    public File(String name) { this.name = name; }
    @Override
    public void showDetails() {
        System.out.println("File: " + name);
    }
}


// Step 3: Composite
class Folder implements FileSystemComponent {
    private String name;
```

```
    private List<FileSystemComponent> children = new ArrayList<>();
    public Folder(String name) { this.name = name; }

    public void add(FileSystemComponent component) { children.add(component); }
    public void remove(FileSystemComponent component) { children.remove(component); }

    @Override
    public void showDetails() {
        System.out.println("Folder: " + name);
        for(FileSystemComponent c : children) c.showDetails();
    }
}

// Step 4: Client
public class Main {
    public static void main(String[] args) {
        File file1 = new File("file1.txt");
        File file2 = new File("file2.txt");

        Folder folder1 = new Folder("Documents");
        folder1.add(file1);
        folder1.add(file2);

        Folder root = new Folder("Root");
        root.add(folder1);
        root.add(new File("readme.md"));
```

```
        root.showDetails();
    }
}
```

🧠 Design Pattern Used

Composite Pattern — lets clients treat individual objects (File) and compositions (Folder) uniformly.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Duplicate traversal logic	Yes	No — handled in `Folder` composite
Hard to nest components	Yes	Easy — folder can contain folder or file
Client complexity	High	Low — client treats all as `FileSystemComponent`
Maintainability	Hard	Easy — add new types without changing client

---

⚙️ Output Example

Folder: Root

Folder: Documents

File: file1.txt

File: file2.txt

File: readme.md

💡 Real-World Uses

- File system directories and files
- UI components (windows, panels, buttons)
- Organization hierarchy (employees, managers)
- Graphics/scene trees in game engines

If you want, I can next give another **Behavioral Pattern** like **Mediator** or **Chain of Responsibility**, which are very useful in event-driven or request-handling systems.

Do you want me to do that next?

Perfect! Let's do another **Behavioral Pattern** — the **Chain of Responsibility Pattern**.

The **Chain of Responsibility Pattern** is used to **avoid coupling the sender of a request to its receiver**. It fixes the **code smell of long `if-else` or `switch` statements** for handling multiple conditions.

⚠️ Example 13: Code Smell (Bad Design)

Suppose you are implementing a **support ticket system** with different levels of support: **Level1**, **Level2**, **Level3**.

A naive approach uses `if-else`:

```
// ❌ CODE SMELL VERSION

class SupportTicket {

    String issue;

    int level; // 1, 2, 3
```

```
    public SupportTicket(String issue, int level) { this.issue = issue; this.level = level; }
}

class SupportHandler {
    public void handleTicket(SupportTicket ticket) {
        if(ticket.level == 1) {
            System.out.println("Level 1 support resolved: " + ticket.issue);
        } else if(ticket.level == 2) {
            System.out.println("Level 2 support resolved: " + ticket.issue);
        } else if(ticket.level == 3) {
            System.out.println("Level 3 support resolved: " + ticket.issue);
        } else {
            System.out.println("Ticket cannot be resolved: " + ticket.issue);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SupportHandler handler = new SupportHandler();
        handler.handleTicket(new SupportTicket("Password reset", 1));
        handler.handleTicket(new SupportTicket("Server down", 3));
    }
}
```

💀 Code Smells:

1. **Long if–else chains** — grows with more levels.
2. **Tightly coupled logic** — handler must know all levels.
3. **Hard to extend** — adding new support levels requires modifying handler.
4. **Violates Open/Closed Principle** — not easily extensible.

✅ Example 13: Refactored Using Chain of Responsibility Pattern

We create a **handler hierarchy** and let requests **pass along the chain** until handled.

```
// Step 1: Abstract Handler
abstract class SupportHandler {
    protected SupportHandler next;

    public void setNextHandler(SupportHandler next) { this.next = next; }

    public abstract void handle(SupportTicket ticket);
}

// Step 2: Concrete Handlers
class Level1Support extends SupportHandler {
    @Override
    public void handle(SupportTicket ticket) {
        if(ticket.level == 1) {
            System.out.println("Level 1 support resolved: " + ticket.issue);
        } else if(next != null) {
```

```
            next.handle(ticket);
        }
    }
}

class Level2Support extends SupportHandler {
    @Override
    public void handle(SupportTicket ticket) {
        if(ticket.level == 2) {
            System.out.println("Level 2 support resolved: " + ticket.issue);
        } else if(next != null) {
            next.handle(ticket);
        }
    }
}

class Level3Support extends SupportHandler {
    @Override
    public void handle(SupportTicket ticket) {
        if(ticket.level == 3) {
            System.out.println("Level 3 support resolved: " + ticket.issue);
        } else if(next != null) {
            next.handle(ticket);
        } else {
            System.out.println("Ticket cannot be resolved: " + ticket.issue);
        }
```

```
    }
}

// Step 3: Request class
class SupportTicket {
    String issue;
    int level;
    public SupportTicket(String issue, int level) { this.issue = issue; this.level = level; }
}

// Step 4: Client
public class Main {
    public static void main(String[] args) {
        SupportHandler level1 = new Level1Support();
        SupportHandler level2 = new Level2Support();
        SupportHandler level3 = new Level3Support();

        // Build chain
        level1.setNextHandler(level2);
        level2.setNextHandler(level3);

        // Create tickets
        SupportTicket ticket1 = new SupportTicket("Password reset", 1);
        SupportTicket ticket2 = new SupportTicket("Software crash", 2);
        SupportTicket ticket3 = new SupportTicket("Server down", 3);
```

```
        // Process tickets
        level1.handle(ticket1);
        level1.handle(ticket2);
        level1.handle(ticket3);
    }
}
```

🧠 Design Pattern Used

Chain of Responsibility Pattern — decouples sender and receiver, allowing multiple handlers to process requests dynamically.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Long if–else	Yes	No — chain handles requests
Tightly coupled	Yes	No — sender doesn't know receivers
Extensibility	Hard	Easy — add new handler classes without modifying existing ones
Maintainability	Low	High — clear, modular handlers

⚙️ Output Example

Level 1 support resolved: Password reset

Level 2 support resolved: Software crash

Level 3 support resolved: Server down

💡 Real-World Uses

- GUI event handling (Swing, JavaFX)
- Logging frameworks (loggers in a chain)
- Workflow or approval systems
- Exception handling in some frameworks

If you want, I can next give another **Behavioral Pattern** like **Mediator Pattern**, which is useful for **reducing complex communication between multiple objects**.

Do you want me to do that next?

Perfect! Let's make a **brand new lab-standard question** — this time we'll use the **Decorator Pattern** with **Data Class / Lazy Class code smell**.

🧪 Lab Question 15 — Decorator Pattern + Code Smell Refactoring

📘 Scenario:

You are developing a **coffee shop application**.
Currently, there is a `Coffee` class with a `getCost()` method. To add extra toppings (like milk, sugar, chocolate), the programmer created multiple subclasses (`CoffeeWithMilk`, `CoffeeWithSugar`, `CoffeeWithChocolate`) and combined them manually.

The current design is **rigid** and **not scalable**: every new topping requires a new subclass.

Your task is to:

1. Identify the **code smells** in the current design.
2. Choose the **appropriate design pattern**.
3. Refactor the code to remove the smells and make the system **extensible** for any new toppings.

💀 Existing Code (Before Refactoring)

```
class Coffee {
    public double getCost() {
        return 50.0;
    }
}

class CoffeeWithMilk extends Coffee {
    @Override
    public double getCost() {
        return super.getCost() + 10;
    }
}

class CoffeeWithSugar extends Coffee {
    @Override
    public double getCost() {
        return super.getCost() + 5;
    }
}
```

```
class CoffeeWithChocolate extends Coffee {
    @Override
    public double getCost() {
        return super.getCost() + 15;
    }
}

public class CoffeeShop {
    public static void main(String[] args) {
        Coffee c1 = new CoffeeWithMilk();
        System.out.println("Cost: " + c1.getCost());

        Coffee c2 = new CoffeeWithChocolate();
        System.out.println("Cost: " + c2.getCost());
    }
}
```

🔍 Part A — Identify the Problems

1. What **code smells** do you notice?
2. Why is this design **hard to extend** when adding a new topping (like caramel)?
3. Which **design pattern** can you use to solve the problem?

💡 Hints

- Look for **Data Class / Lazy Class** smells: each subclass is tiny and just adds cost.
- Adding new toppings requires creating **many subclasses**, violating **Open/Closed Principle**.

🧠 Expected Concepts

Category	Code Smell	Description
Dispensable	Data Class / Lazy Class	Subclasses only store extra cost, no real logic
Change Preventers	Shotgun Surgery	Adding a new topping requires editing multiple places
Design Principle Violation	Open/Closed Principle	Hard to extend without modifying code

Design Pattern to Apply: ✅ Decorator Pattern

- Decorator allows **dynamic addition of responsibilities** to objects without creating new subclasses.

🔧 Refactored Code (After Applying Decorator Pattern)

```
// Step 1: Component interface

interface Coffee {

    double getCost();

    String getDescription();

}
```

```
// Step 2: Concrete component
class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 50.0;
    }

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

// Step 3: Decorator abstract class
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

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

    public abstract double getCost();
    public abstract String getDescription();
}

// Step 4: Concrete decorators
```

```
class Milk extends CoffeeDecorator {
    public Milk(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 10;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
}

class Sugar extends CoffeeDecorator {
    public Sugar(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 5;
    }
```

```
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }
}

class Chocolate extends CoffeeDecorator {
    public Chocolate(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 15;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Chocolate";
    }
}

// Step 5: Client code
public class CoffeeShop {
    public static void main(String[] args) {
        Coffee c1 = new Milk(new SimpleCoffee());
```

```
        System.out.println(c1.getDescription() + " => Cost: " + c1.getCost());


        Coffee c2 = new Chocolate(new Sugar(new SimpleCoffee()));

        System.out.println(c2.getDescription() + " => Cost: " + c2.getCost());

    }

}
```

🖼️ Expected Output

Simple Coffee, Milk => Cost: 60.0

Simple Coffee, Sugar, Chocolate => Cost: 70.0

🗒️ Lab Submission Requirements

1. **Code Smell Analysis (2–3 sentences):**
   The original code had **Data Class / Lazy Class** smells, as each subclass only added a small cost.
   It also violated **Open/Closed Principle** because adding new toppings required creating new subclasses.

2. **Design Pattern Used:**
   ✅ **Decorator Pattern** — allows dynamic addition of responsibilities (toppings) to objects without subclass explosion.

3. Submit **before and after refactoring** code files with screenshots.

If you want, I can **generate a full set of 5–6 lab questions**, covering:

- Factory Pattern
- Strategy Pattern

- Observer Pattern
- Decorator Pattern
- Builder Pattern

…so you have a **ready-to-practice lab sheet** for all common design patterns + code smells.

Do you want me to do that?

Perfect! Let's go for a **Structural Pattern** — the **Proxy Pattern**.

The **Proxy Pattern** is used to **control access** to an object, often to add **lazy initialization, access control, logging, or caching**. It fixes the **code smell of repeated checks or expensive object creation** scattered throughout the code.

⚠️ Example 16: Code Smell (Bad Design)

Suppose you have a **heavy image** class that loads an image from disk.
A naive approach loads it every time, even if not needed:

```
// ❌ CODE SMELL VERSION

class RealImage {

    private String filename;


    public RealImage(String filename) {

        this.filename = filename;

        loadFromDisk();

    }


    private void loadFromDisk() {

        System.out.println("Loading " + filename);

    }
```

```
    public void display() {
        System.out.println("Displaying " + filename);
    }
}

public class Main {
    public static void main(String[] args) {
        RealImage img1 = new RealImage("photo1.jpg"); // Loads immediately
        img1.display();

        RealImage img2 = new RealImage("photo2.jpg"); // Loads immediately
        img2.display();
    }
}
```

💀 Code Smells:

1. **Expensive object creation** — loads every image even if it won't be displayed immediately.
2. **Repeated checks or initialization** — client has to manage it manually.
3. **Tightly coupled client** — client must handle loading logic.

✅ Example 16: Refactored Using Proxy Pattern

We create a **proxy** that controls access and loads the image **only when needed**:

```
// Step 1: Subject interface
```

```
interface Image {
    void display();
}

// Step 2: Real Subject
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + filename);
    }
}

// Step 3: Proxy
class ProxyImage implements Image {
    private RealImage realImage;
```

```
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename); // Lazy loading
        }
        realImage.display();
    }
}

// Step 4: Client
public class Main {
    public static void main(String[] args) {
        Image img1 = new ProxyImage("photo1.jpg"); // Not loaded yet
        Image img2 = new ProxyImage("photo2.jpg"); // Not loaded yet

        // Load only when display is called
        img1.display();
        img1.display(); // Already loaded, no extra loading
        img2.display();
    }
```

}

🧠 Design Pattern Used

Proxy Pattern — provides a placeholder for another object to control access, lazy initialization, caching, or access control.

✅ Benefits Compared to Smelly Version

Problem	Before	After
Expensive object creation	Always loaded	Lazy loading on demand
Repeated initialization logic	Scattered	Encapsulated in proxy
Tightly coupled client	Yes	Client uses proxy, no need to handle loading
Maintainability	Low	High — proxy manages object access

⚙️ Output Example

Loading photo1.jpg

Displaying photo1.jpg

Displaying photo1.jpg

Loading photo2.jpg

Displaying photo2.jpg

Notice **photo1.jpg** only loads **once**, thanks to the proxy.

💡 Real-World Uses

- Virtual Proxy: lazy loading (e.g., images in UI)
- Remote Proxy: access to remote services (e.g., RMI)
- Protection Proxy: access control (e.g., file system permissions)
- Smart Reference: logging, caching, or counting references

If you want, I can next give another **Structural Pattern** like **Decorator Pattern**, which is very useful for **adding behavior dynamically** without modifying the original class.

Do you want me to do that next?