Design Patterns

What is Design Pattern?

Design patterns are typical solutions to **commonly occurring problems** in software design. They are like pre-made blueprints that you can customize to solve a **recurring design problem** or **address a recurring scenario** in your code.

Why Design Pattern is Required?

- Solves common, recurring software problems.
- Improves code readability and maintainability.
- Promotes code reusability and flexibility.
- Reduces development time by providing proven solutions.

How Design Patterns Can Help

- Provides time-tested solutions to common issues.
- Promotes cleaner and more organized code.
- Improves collaboration with standardized approaches.
- Helps in managing complex software architecture.

History of Design Pattern

The concept of patterns was first described by Christopher Alexander in **A Pattern Language: Towns, Buildings, Construction**. The book describes a “language” for designing the urban environment. The units of this language are patterns. They may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

Software Adoption (1980s): In the 1980s, the software engineering community began recognizing the importance of reusable solutions to common problems in coding. However, the formalization of design patterns as a discipline within software engineering began to take shape in the late 1980s.

https://www.amazon.com/-/dp/0195019199

History of Design Pattern

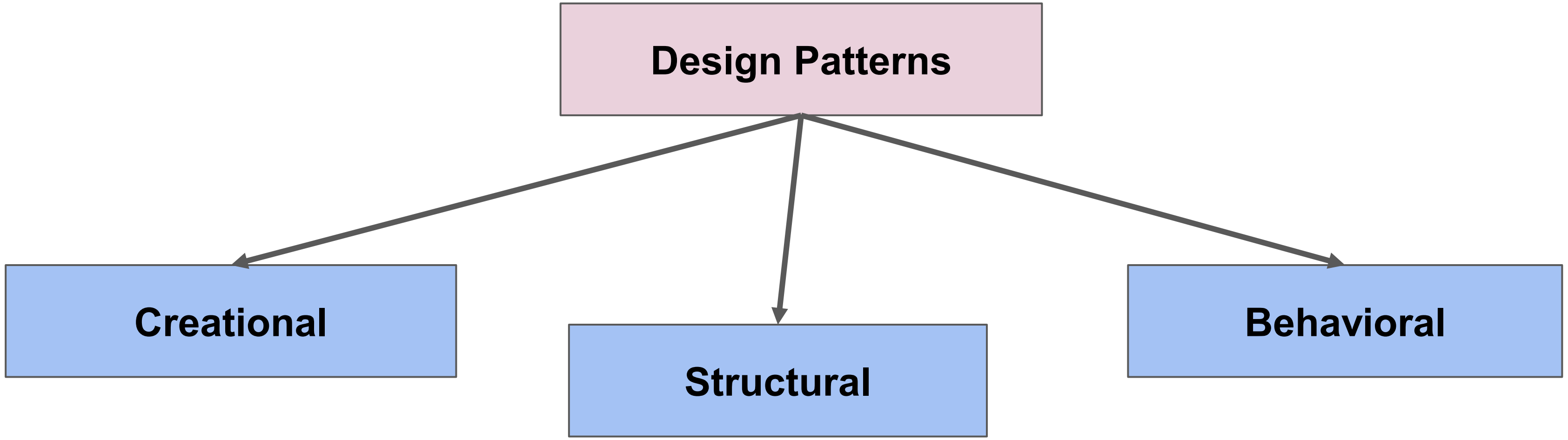
The "Gang of Four" (1994): The pivotal moment in the history of design patterns came with the publication of the influential book *Design Patterns: Elements of Reusable Object-Oriented Software* by **Erich Gamma**, **Richard Helm**, **Ralph Johnson**, and **John Vlissides**. This book, known as the "Gang of Four" (GoF) book, formalized 23 core design patterns, such as Singleton, Factory Method, and Observer. This book marked the beginning of widespread use of design patterns in object-oriented programming (OOP).

https://www.amazon.com/gp/product/0201633612/

Why should I use Design Patterns?

1. Design patterns are a toolkit of tried and tested solutions to common problems in software design.
2. Design patterns define a common language that you and your teammates can use to communicate more efficiently.

Classification of Design Patterns



Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented. Creational patterns give a lot of flexibility in what gets created, who creates it, and how it gets created. There are two main themes in these patterns:

- They keep information about the specific classes used in the system hidden.
- They hide the details of how instances of these classes are created and assembled.

Examples: Singleton, Factory Method, Abstract Factory, Builder patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships between objects, making code more efficient, flexible, and easy to maintain.

- This pattern is particularly useful for making independently developed class libraries work together.
- Structural Design Patterns describe ways to compose objects to realize new functionality.

Examples: Adapter, Composite, Proxy, Facade patterns

Behavioral Design Patterns

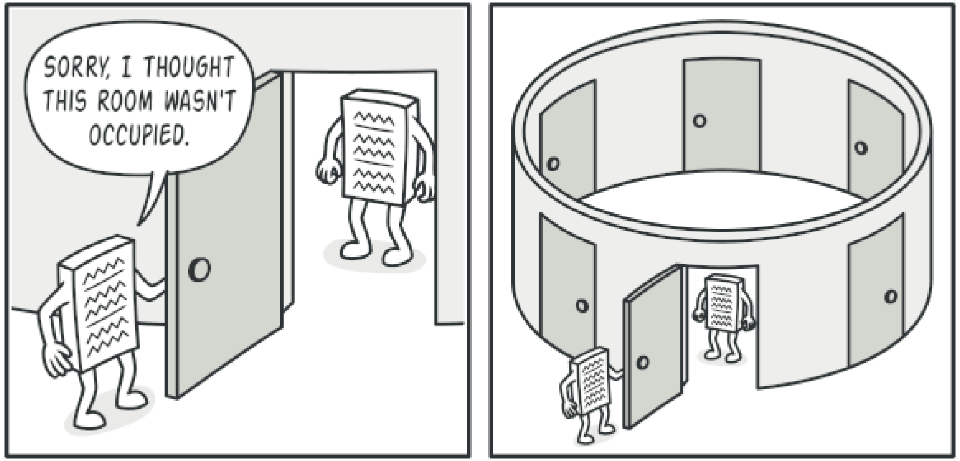
Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, making it easier to manage complex control flow and communication in a system.

Examples: Chain of responsibility, Observer, State, Strategy patterns

Singleton Pattern

What is Singleton Pattern?

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.



Properties of Singleton Pattern

1. Ensure that a class has just **a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource.

   Example: a database or a file.

1. Provide a global access point to that instance. Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

Real World Analogy

Imagine an office with 50 employees and one expensive printer. When anyone needs to print, whether it's Sarah with her report, John with invoices, or Maria with a presentation. They all send their documents to the same printer. Nobody creates a new printer each time they need to print; everyone shares the one that already exists, saving money and resources.

The Singleton pattern works exactly like the office printer: no matter how many times employees ask for "the printer," they always get the same single shared instance rather than creating wasteful duplicates.

Steps to Implement Singleton Pattern

Step 1: Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

Step 2: Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Implementing the Singleton Pattern

```java
public class Printer {
    private static Printer instance = null;

    private Printer() {
        System.out.println("Printer created!");
    }

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

    public void print(String document) {
        System.out.println("Printing: " + document);
    }
}

// Usage:
Printer p1 = Printer.getInstance();  // Creates printer
Printer p2 = Printer.getInstance();  // Returns same printer
System.out.println(p1 == p2);        // true - same instance!
```

When to Apply Singleton Pattern

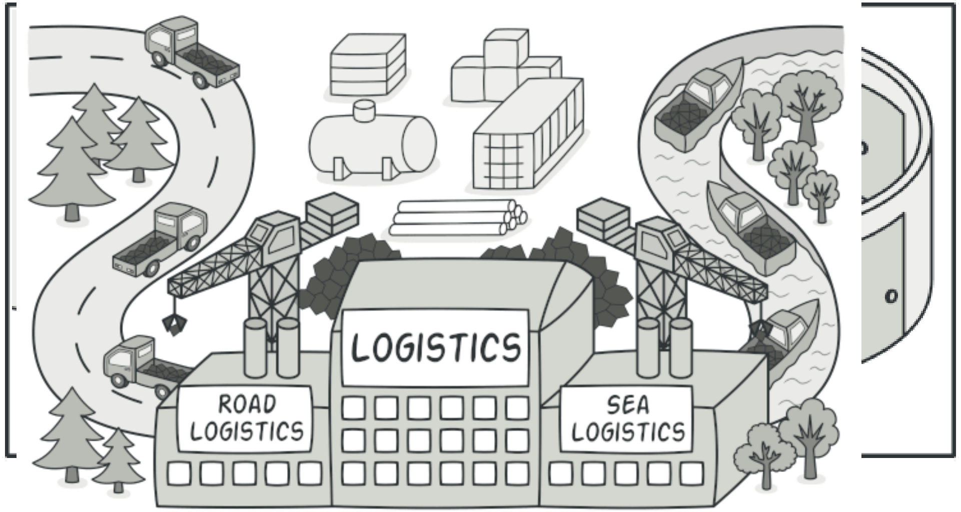
1. Use the Singleton pattern when a class in your program should have just **a single instance available to all clients**; for example, a single database object shared by different parts of the program.

1. Use the Singleton pattern when you need **stricter control over global variables.**

Factory Pattern

What is Factory Pattern?

Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Problem Domain

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by **trucks**, so the bulk of your code lives inside the **Truck class.**

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate **sea logistics** into the app.

LogisticsApp

Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Problem Domain

Great news, right? But how about the code? At present, most of your code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app’s behavior depending on the class of transportation objects.

Steps to Implement Factory Pattern

Step 1: Define an interface that all class categories/types will implement.

Step 2: Create concrete classes from the **Interface.**

Step 3: Create the **Factory class.**

Step 4: Clients should request for anything using the Factory class.

Factory Pattern Implementation

```java
// Step 1: Define an interface that all notifications will implement
interface Notification {
    String send(String message);
}

// Step 2: Create concrete notification classes
class EmailNotification implements Notification {
    @Override
    public String send(String message) {
        return "📧 Sending EMAIL: " + message;
    }
}

class SMSNotification implements Notification {
    @Override
    public String send(String message) {
        return "📱 Sending SMS: " + message;
    }
}

class PushNotification implements Notification {
    @Override
    public String send(String message) {
        return "🔔 Sending PUSH notification: " + message;
    }
}
```

Factory Pattern Implementation

```java
// Step 3: Create the Factory
class NotificationFactory {
    /**
     * Creates and returns the appropriate notification object
     * based on the type requested
     */
    public static Notification createNotification(String notificationType) {
        switch (notificationType.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type: " + notificationType);
        }
    }
}
```

Factory Pattern Implementation

```java
// Step 4: Use the Factory
public class FactoryPatternDemo {
    public static void main(String[] args) {
        // Instead of creating objects directly, we use the factory

        // Get different notification types from the factory
        Notification email = NotificationFactory.createNotification("email");
        Notification sms = NotificationFactory.createNotification("sms");
        Notification push = NotificationFactory.createNotification("push");

        // Use them
        System.out.println(email.send("Welcome to our service!"));
        System.out.println(sms.send("Your verification code is 1234"));
        System.out.println(push.send("You have a new message"));

        System.out.println("\n--- Example with user input ---");
        // More realistic example: based on user preference
        String userPreference = "sms";  // This could come from user settings
        Notification notification = NotificationFactory.createNotification(userPreference);
        System.out.println(notification.send("Your order has been shipped!"));
    }
}
```

When to Apply Factory Pattern

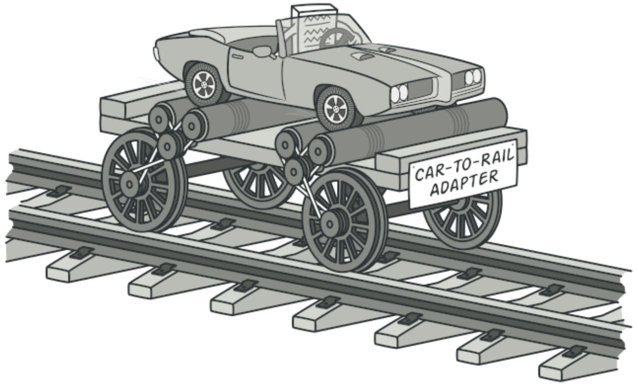
1. Use the Factory Method when you don’t know beforehand **the exact types and dependencies of the objects** your code should work with.
2. Use the Factory Method when you want to provide users of your library or framework with **a way to extend its internal components**.
3. Use the Factory Method when you want to **save system resources by reusing existing objects** instead of rebuilding them each time.

Structural Patterns

Adapter Pattern

What is Adapter Pattern?

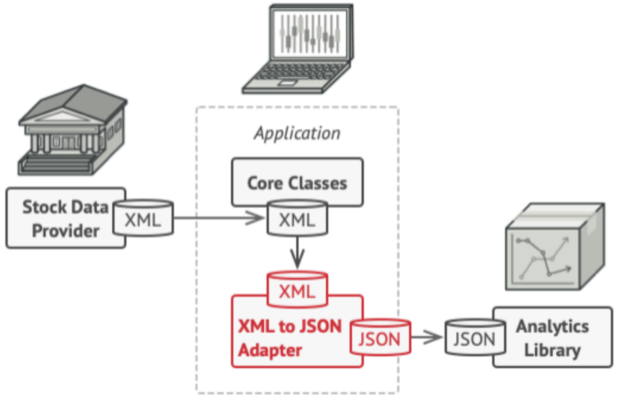
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Problem Domain

Imagine that you're creating a **stock market monitoring app**. The app downloads the stock data from multiple sources in **XML format** and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in **JSON format.**



Steps to Implement Adapter Pattern

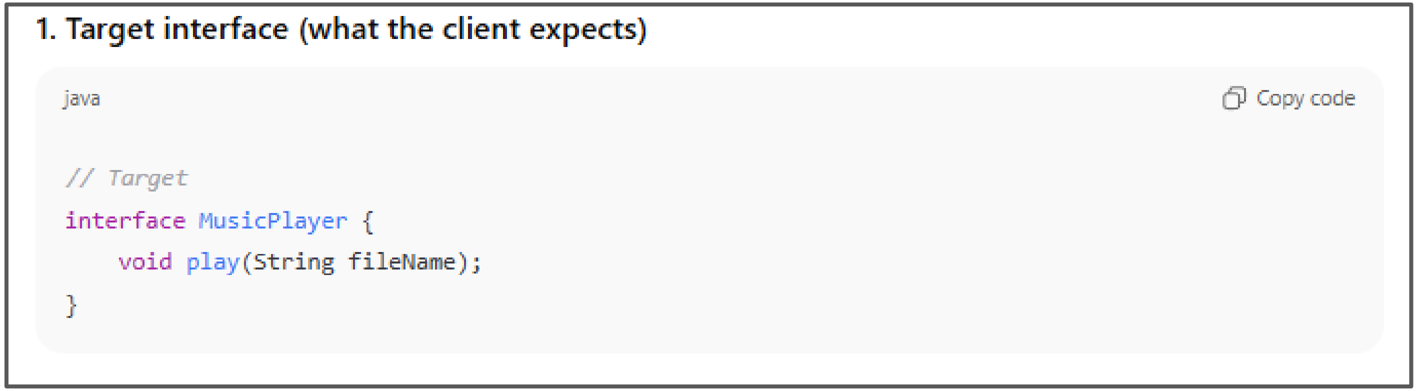
Step 1: Identify incompatible interfaces

Step 2: Define the Target interface that clients expect

Step 3: Create an Adapter that implements Target and wraps the Adaptee

Step 4: Inside Adapter methods, translate calls to the Adaptee’s methods

Pseudocode Implementation of Adapter Pattern

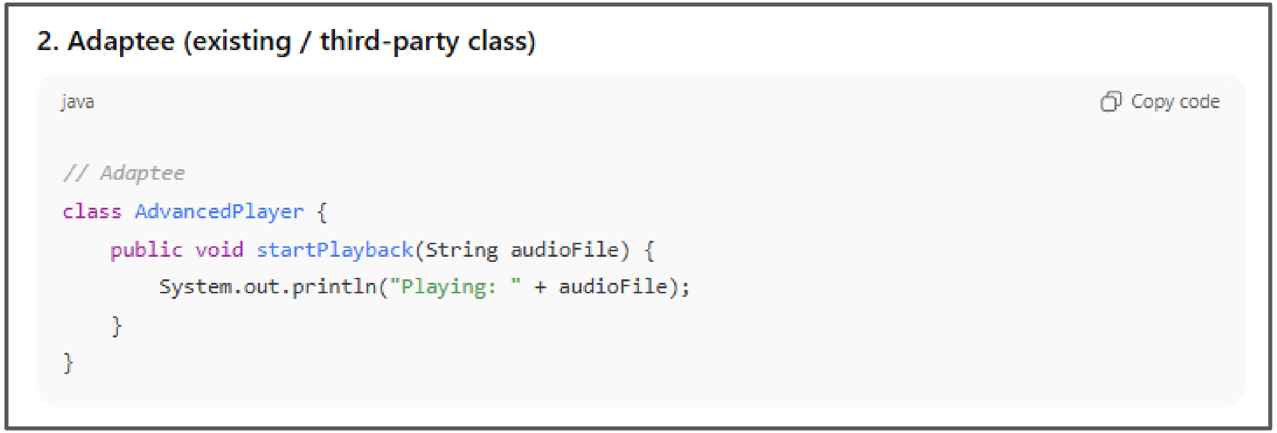


1. Target interface (what the client expects)

java

Copy code

```java
// Target
interface MusicPlayer {
    void play(String fileName);
}
```



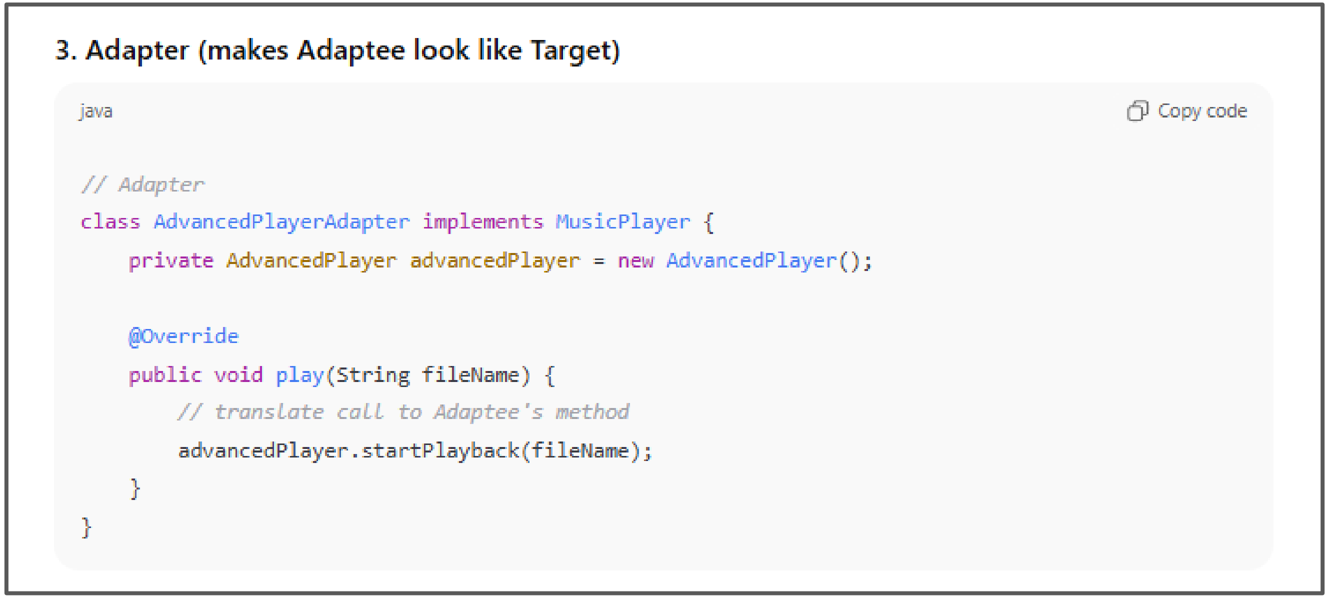
2. Adaptee (existing / third-party class)

java

Copy code

```java
// Adaptee
class AdvancedPlayer {
    public void startPlayback(String audioFile) {
        System.out.println("Playing: " + audioFile);
    }
}
```

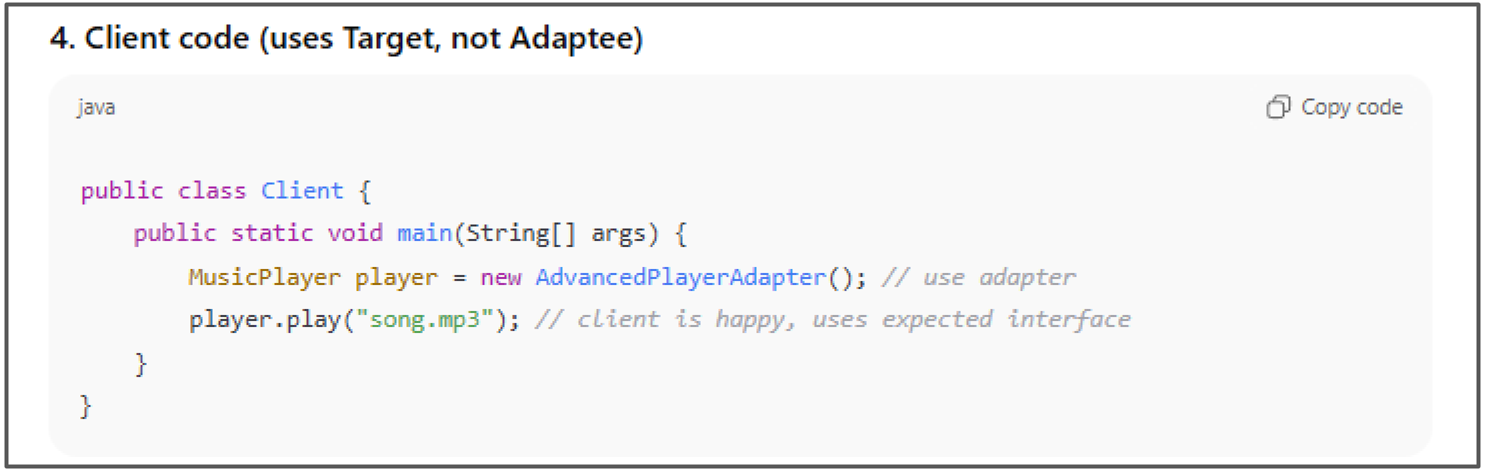
Pseudocode Implementation of Adapter Pattern


3. Adapter (makes Adaptee look like Target)

```java
// Adapter
class AdvancedPlayerAdapter implements MusicPlayer {
    private AdvancedPlayer advancedPlayer = new AdvancedPlayer();

    @Override
    public void play(String fileName) {
        // translate call to Adaptee's method
        advancedPlayer.startPlayback(fileName);
    }
}
```

Pseudocode Implementation of Adapter Pattern

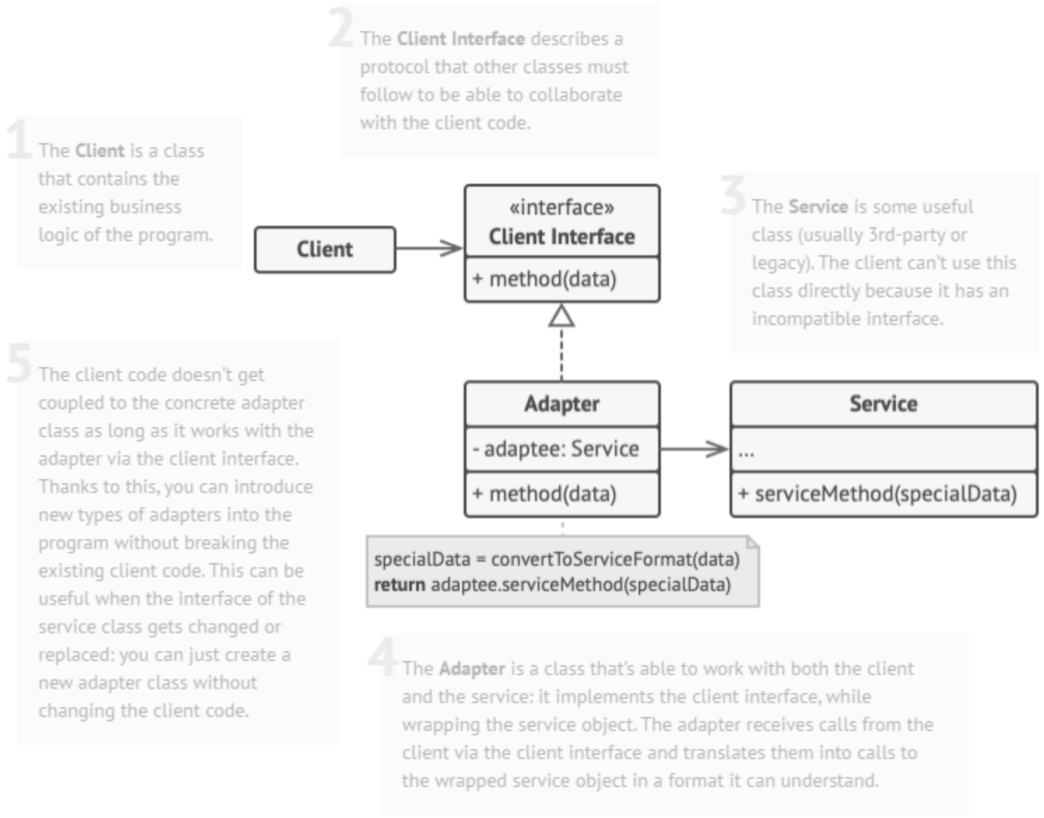

4. Client code (uses Target, not Adaptee)

java

Copy code

```java
public class Client {
    public static void main(String[] args) {
        MusicPlayer player = new AdvancedPlayerAdapter(); // use adapter
        player.play("song.mp3"); // client is happy, uses expected interface
    }
}
```

UML Structure of Adapter Pattern



```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

1 The **Client** is a class that contains the existing business logic of the program.

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

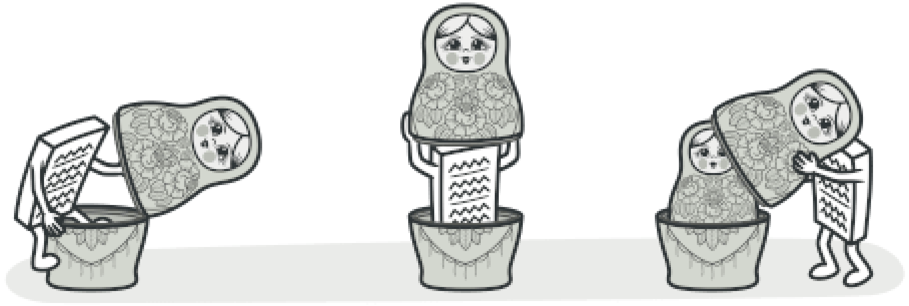
When to Apply Adapter Pattern

1. Use the Adapter class when you want to use some existing class, but its interface **isn’t compatible** with the rest of your code.
2. Use the pattern when you want to **reuse several existing subclasses** that lack some common functionality that can’t be added to the superclass.

Adapter Pattern

What is Adapter Pattern?

Decorator is a structural design pattern that lets you **attach new behaviors to objects** by placing these objects inside special wrapper objects that contain the behaviors.



Problem Domain

In a coffee shop app, customers start with a plain coffee but add extras like milk, sugar, whipped cream or caramel. Each extra changes the drink's price, and you want to combine toppings flexibly without creating many separate drink types.

1 way to address this:

- Make Coffee an Interface and implement it differently in different scenarios such as MilkCoffee, SugarCoffee and CaramelWhippedCreamCoffee.

Problem of this Approach??
This leads to a class explosion, where every new topping combination requires a new class (MilkCoffee, SugarCoffee, CaramelWhippedCreamCoffee, etc.), making the design hard to maintain and extend.

Steps to Implement Decorator Pattern

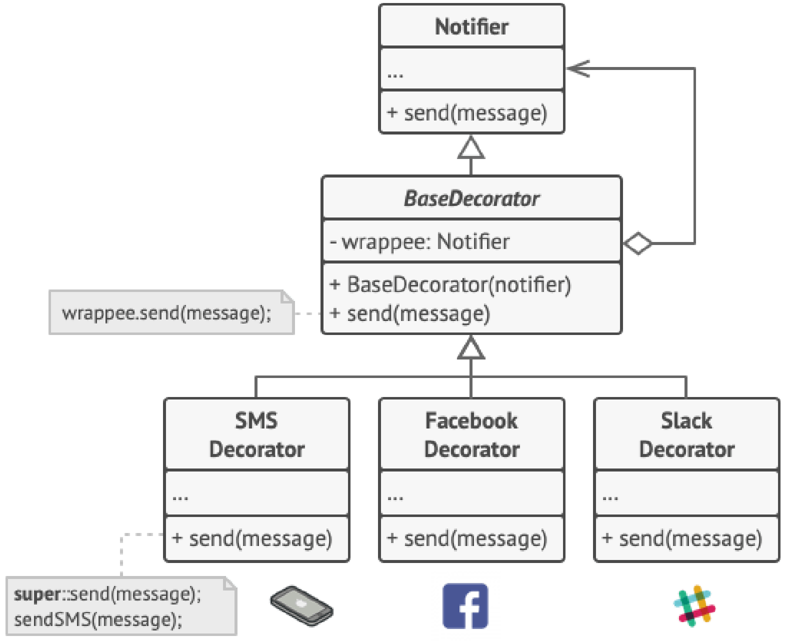
Step 1: Define a common interface for the base object and decorators (e.g., Coffee).

Step 2: Implement a concrete base class (e.g., PlainCoffee).

Step 3: Create decorator classes that implement the same interface and wrap another object.

Step 4: In each decorator, add extra behavior and then delegate to the wrapped object.

UML Structure of Decorator Pattern



Various notification methods become decorators.

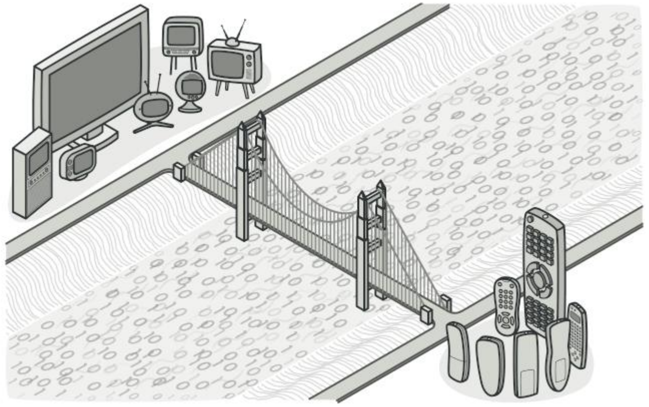
When to Apply Decorator Pattern

1. Use the Decorator pattern when you need to be able to **assign extra behaviors to objects at runtime** without breaking the code that uses these objects.
2. Use the pattern when it’s **awkward or not possible to extend an object’s behavior using inheritance.**

Bridge Pattern

What is Bridge Pattern?

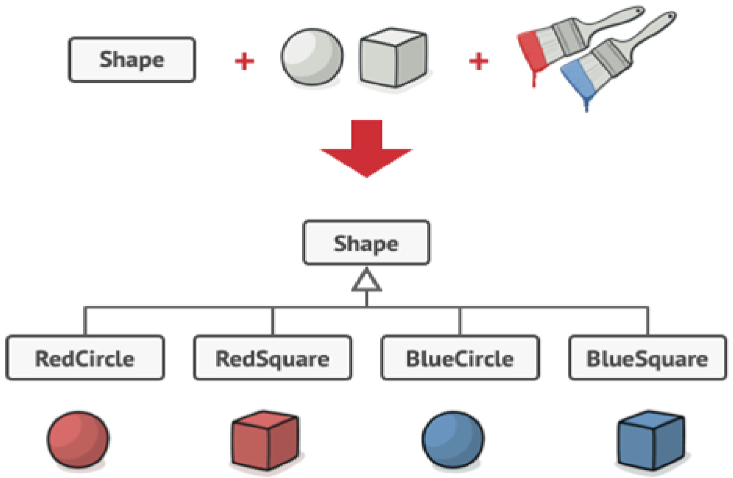
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—**abstraction and implementation**—which can be developed independently of each other.



Problem Domain

Abstraction? Implementation? Sound scary? Stay calm and let's consider a simple example.

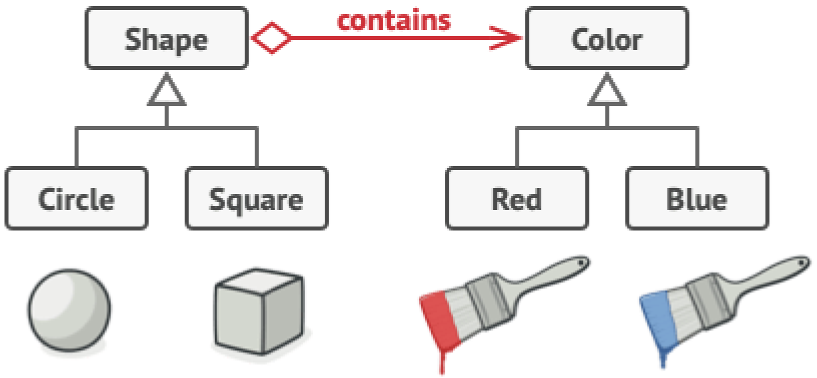
Say you have a geometric `Shape` class with a pair of subclasses: `Circle` and `Square`. You want to extend this class hierarchy to incorporate colors, so you plan to create `Red` and `Blue` shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as `BlueCircle` and `RedSquare`.



Number of class combinations grows in geometric progression.

Solution

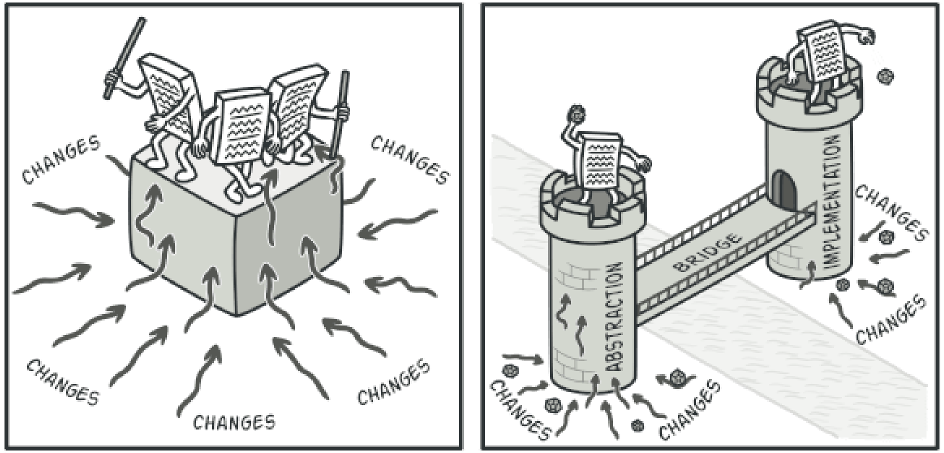
Switching from **inheritance** to **object composition**



You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

Solution

Switching from **inheritance** to **object composition**



Yo

Making even a simple change to a monolithic codebase is pretty hard because you must understand the entire thing very well. Making changes to smaller, well-defined modules is much easier.

hies.

Steps to Implement Bridge Pattern

Step 1: Define the **Abstraction** & **Implementor** interfaces

- Abstraction interface declaring high-level operations.
- Implementor interface with low-level operations that Abstraction will delegate to.

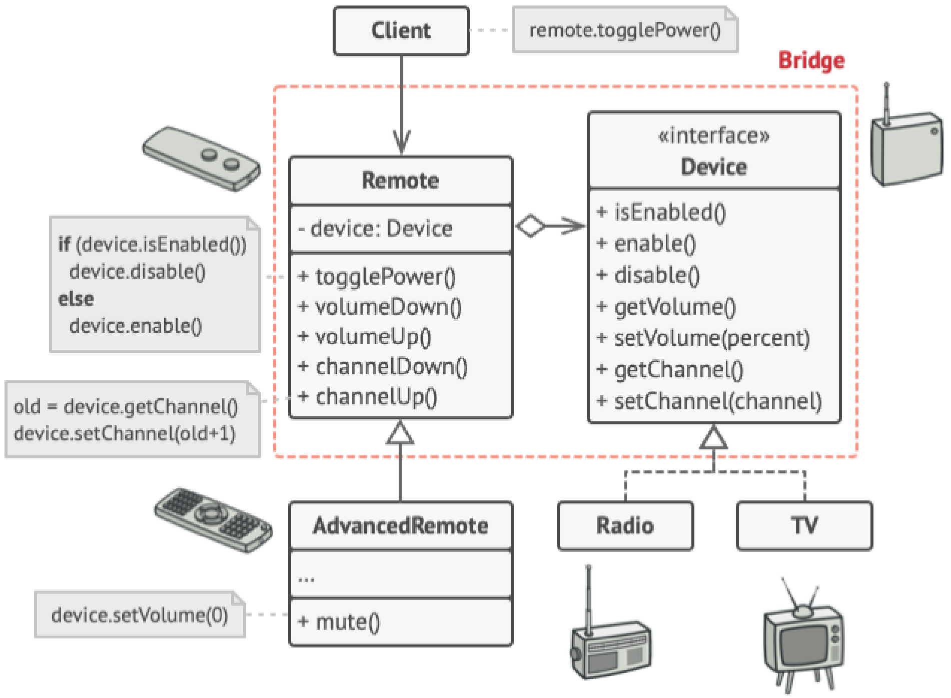
Step 2: Implement the **Implementor** interface in multiple **ConcreteImplementor** classes, each providing a different variant of the low-level behavior.

Steps to Implement Bridge Pattern

Step 3: Link Abstraction to Implementor (the “bridge”)
[delegate work to the Implementor object instead of doing it directly.]

Step 4: At runtime, instantiate a ConcreteImplementor. Then, combine any Abstraction with any ConcreteImplementor and call methods.

UML Structure of Bridge Pattern



The original class hierarchy is divided into two parts: devices and remote controls.

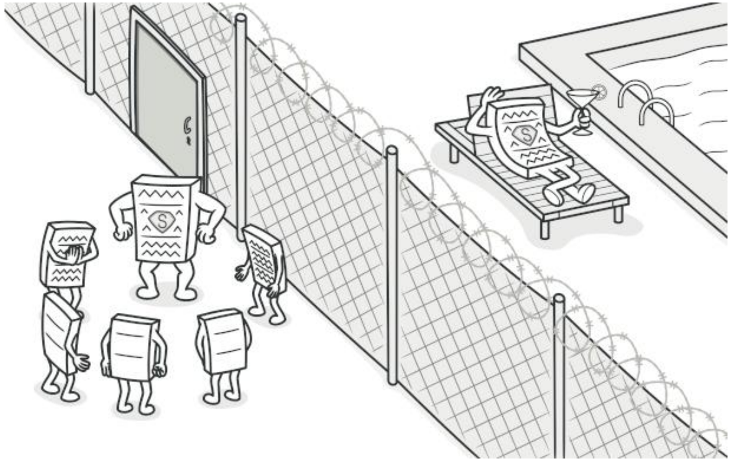
When to Apply Bridge Pattern

1. Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality
2. Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.
3. Use the Bridge if you need to be able to switch implementations at runtime.

Proxy Pattern

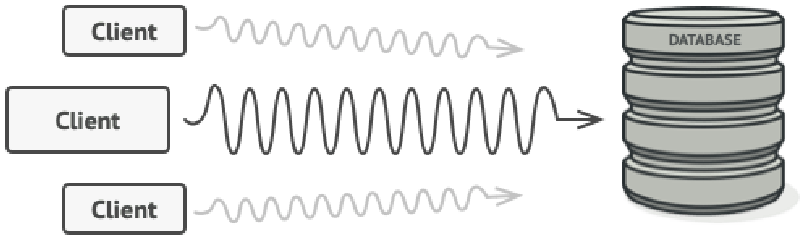
What is Proxy Pattern?

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



Problem Domain

Why would you want to control access to an object? Here is an example: you have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.

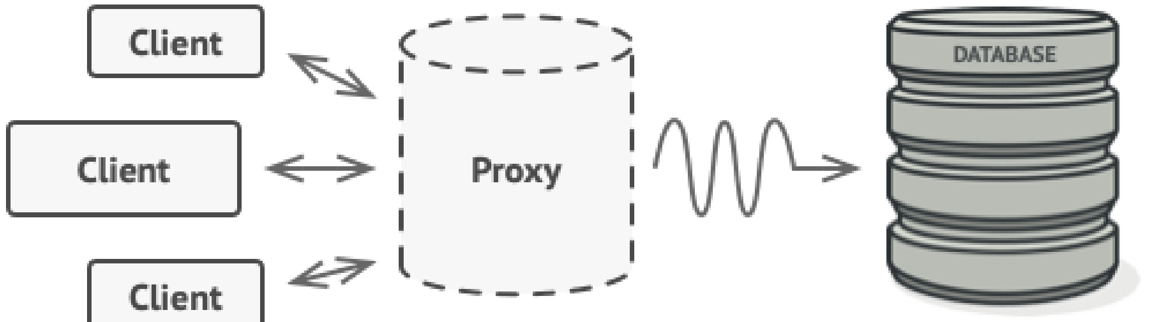


Database queries can be really slow.

You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

In an ideal world, we'd want to put this code directly into our object's class, but that isn't always possible. For instance, the class may be part of a closed 3rd-party library.

Solution



The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

Steps to Implement Proxy Pattern

Step 1: Create an interface/abstract class with the methods clients will use.

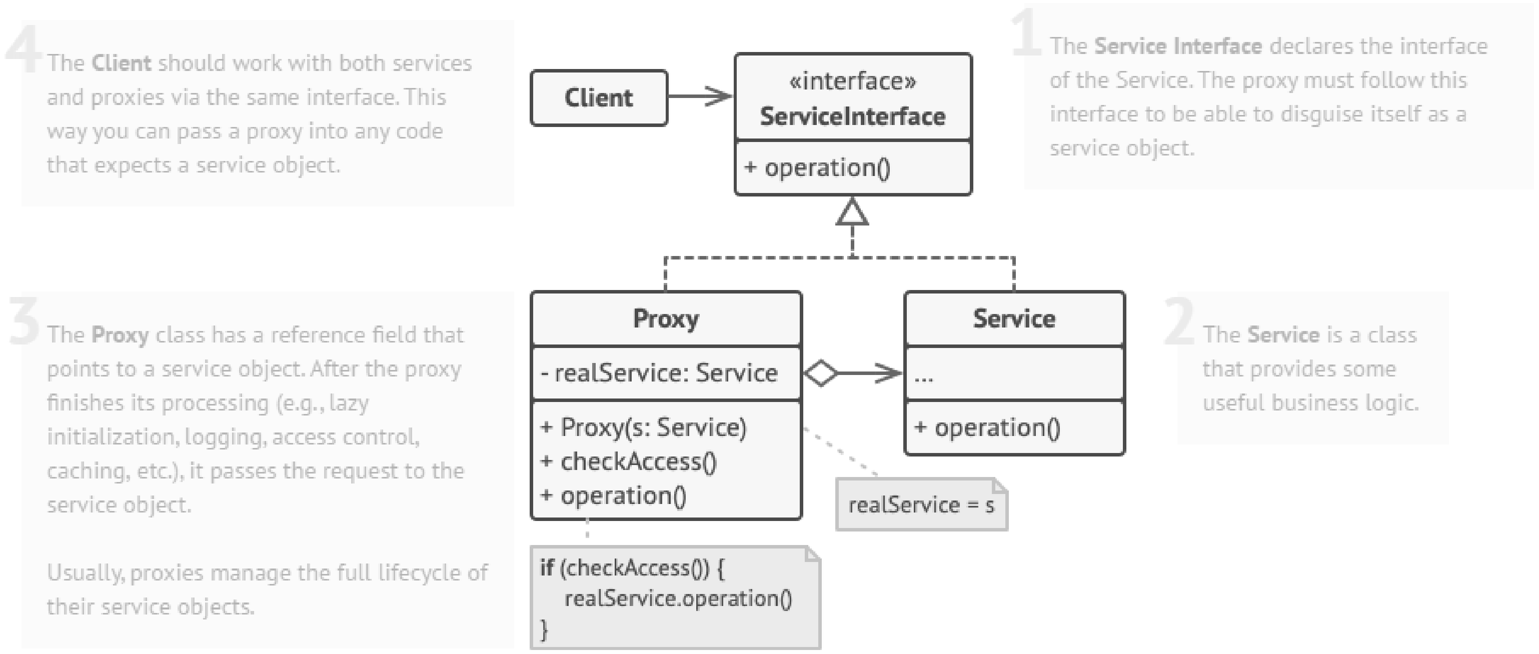
Step 2: Create the real **Service** class that does the actual work and implements that interface.

Step 3: Create the proxy class. Inside each method, add extra logic (e.g., logging, caching, security checks), then delegate to actual **Service**.

Step 4: In client code, work with the interface type **(Service)** but instantiate the proxy **(new ProxyService())**, so all calls go through the proxy.

ServiceInterface obj = new ProxyService();

How to Implement Proxy Pattern



```
if (checkAccess()) {
   realService.operation()
}
```

1 The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

2 The **Service** is a class that provides some useful business logic.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

Usually, proxies manage the full lifecycle of their service objects.

4 The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

When to Apply Proxy Pattern

1. Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
2. Access control (protection proxy). This is when you want only specific clients to be able to use the service object.
3. Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.
4. Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.

Behavioral Patterns

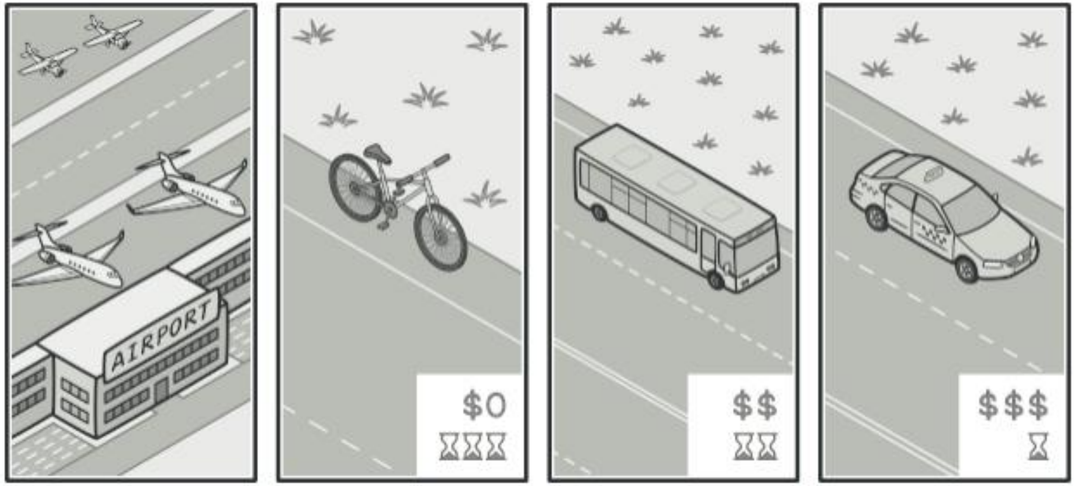
Strategy Pattern

What is Strategy Pattern?

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem Domain

Consider a Navigator app where different types of routing is possible.



Various strategies for getting to the airport.

Steps to Implement Strategy Pattern

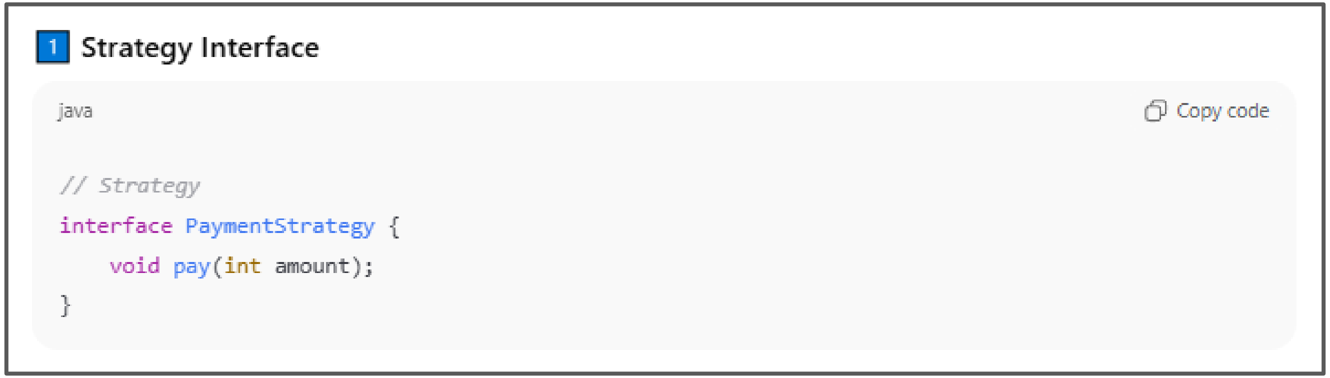
Step 1: Define the **Strategy interface**

Step 2: Create the **Strategy classes**

Step 3: Create the **Context class**

Step 4: Use it from the Client code

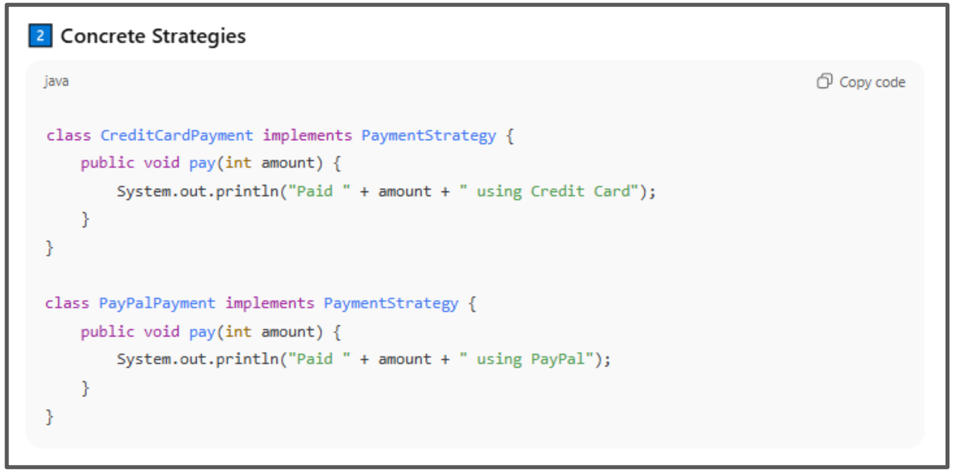
Steps to Implement Strategy Pattern



1 Strategy Interface

java Copy code

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



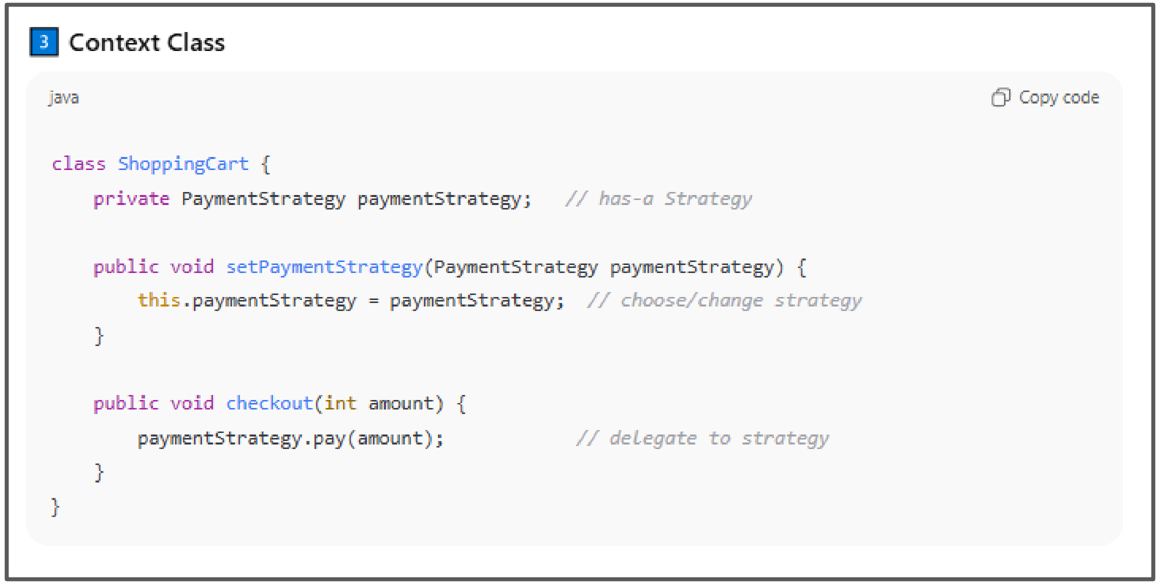
2 Concrete Strategies

java Copy code

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

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

Steps to Implement Strategy Pattern



3 Context Class

java

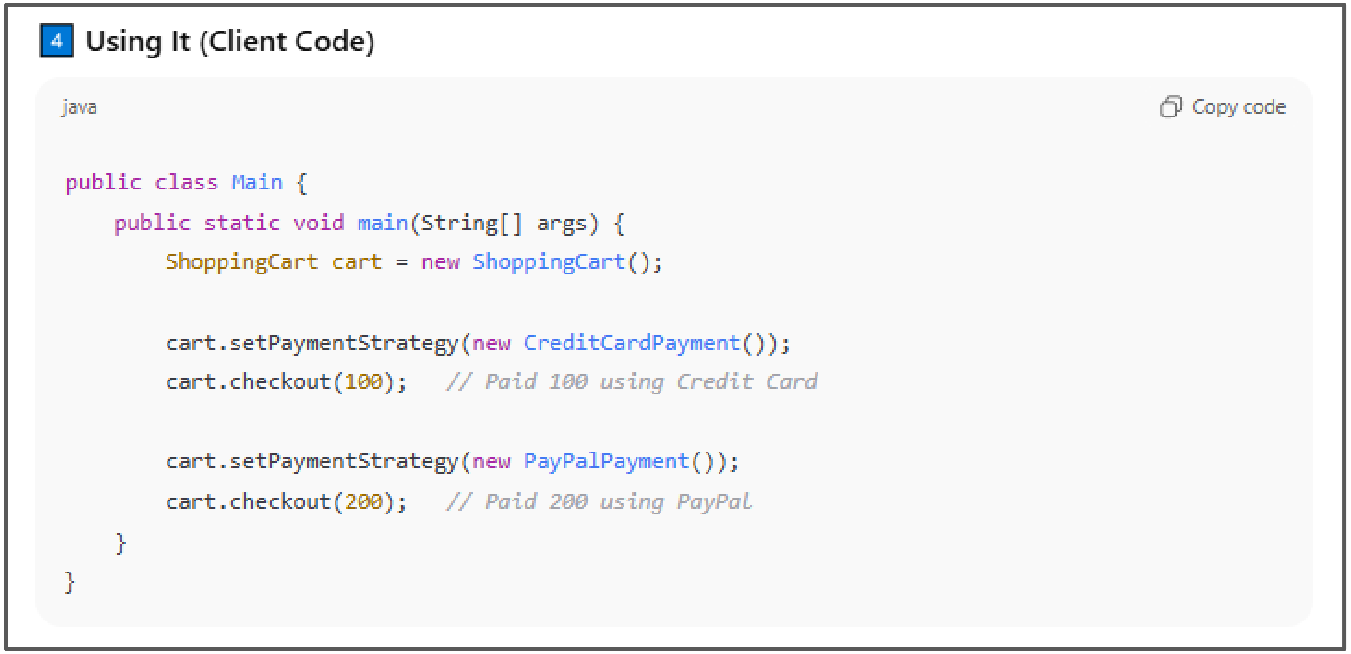
Copy code

```java
class ShoppingCart {
    private PaymentStrategy paymentStrategy;   // has-a Strategy

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

    public void checkout(int amount) {
        paymentStrategy.pay(amount);            // delegate to strategy
    }
}
```

Steps to Implement Strategy Pattern



4 Using It (Client Code)

java

Copy code

```java
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(100);   // Paid 100 using Credit Card

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(200);   // Paid 200 using PayPal
    }
}
```

How to Implement Strategy Pattern

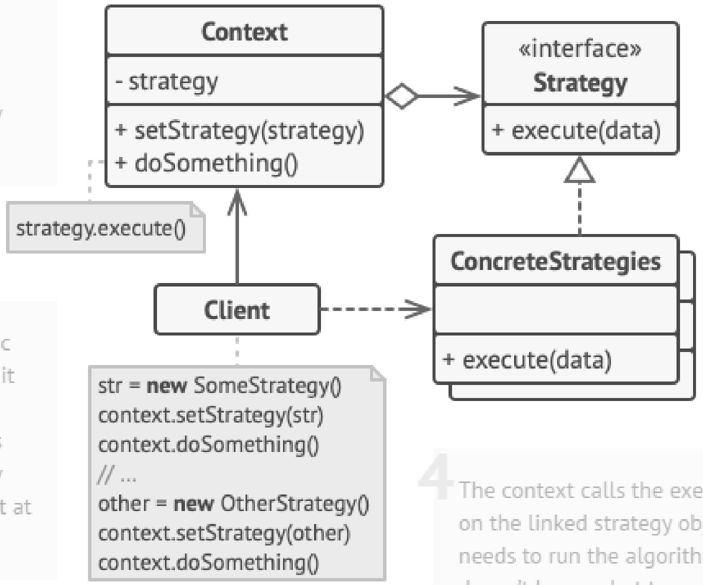
1 The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.

2 The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

3 **Concrete Strategies** implement different variations of an algorithm the context uses.

4 The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

5 The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

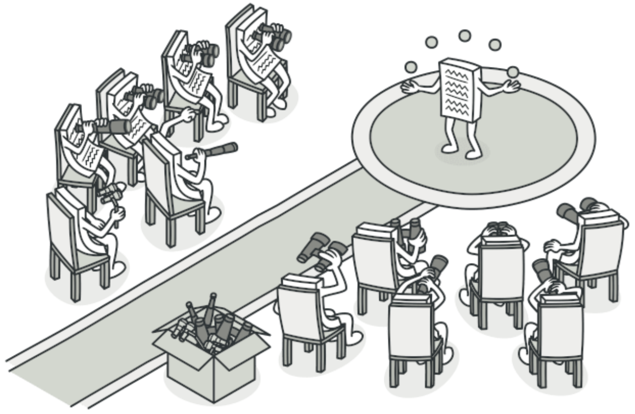
When to Apply Strategy Pattern

1. Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
2. Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.
3. Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.
4. Use the pattern when your class has a massive conditional statement that switches between different variants of the same algorithm.

Observer Pattern

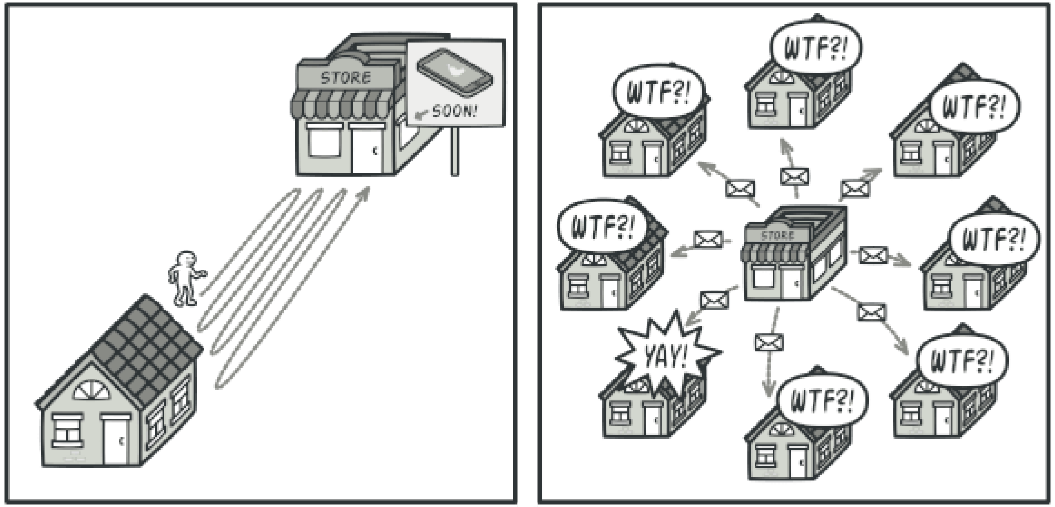
What is Observer Pattern?

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



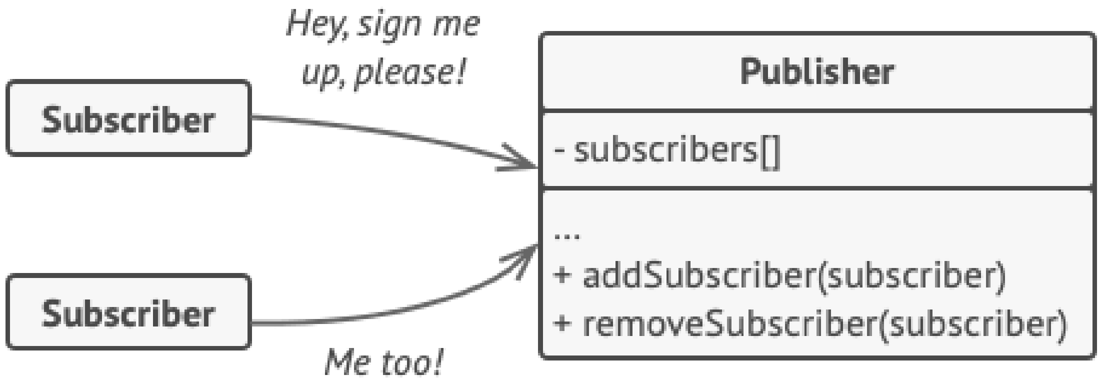
Problem Domain

Suppose, a customer wants to get notified from a store when the new version of the IPhone is launched.



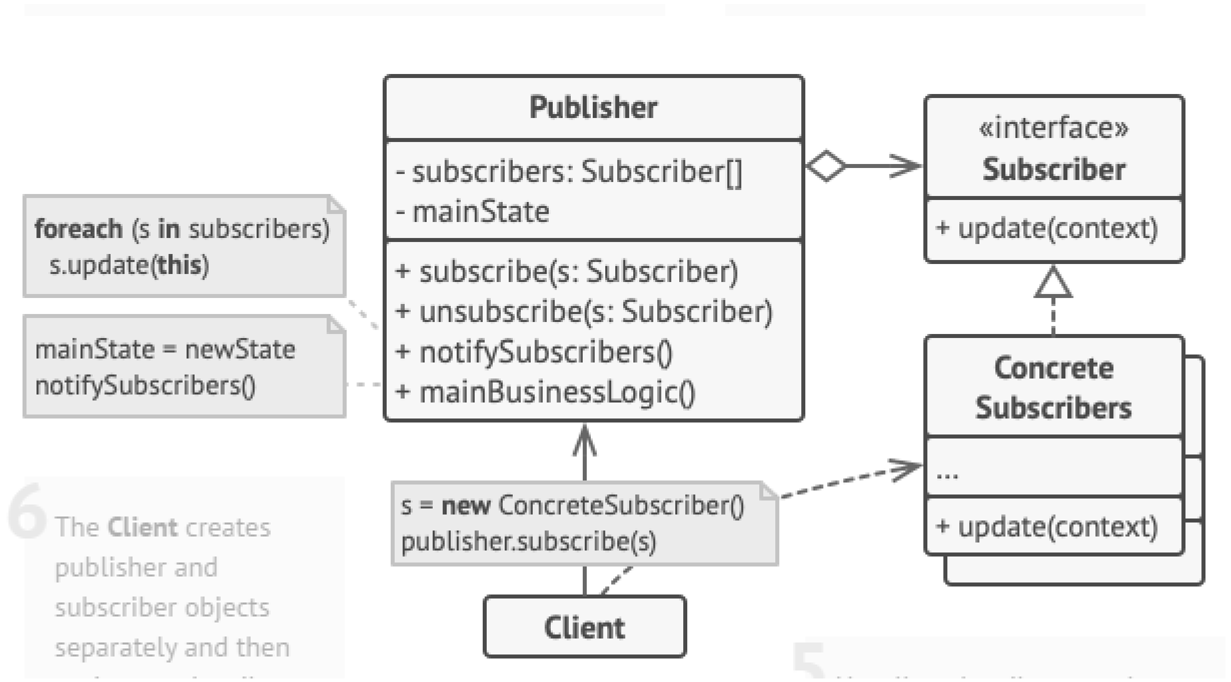
Visiting the store vs. sending spam

Solution



A subscription mechanism lets individual objects subscribe to event notifications.

UML Diagram of Observer Pattern


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
Concrete Subscribers
...
+ update(context)
Client
6
The Client creates publisher and subscriber objects separately and then
5

```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

Steps to Implement Observer Pattern

Step 1: (Define contracts) Create Publisher and Subscriber interfaces, Subscriber exposes a method like **update(event)** or **handle(message)**.

Step 2: (Manage subscriptions) Publisher keeps a list of subscribers and provides **subscribe() / unsubscribe()** methods.

Step 3: (Publish events) When something relevant happens, the publisher calls each subscriber’s update/handle with the event data.

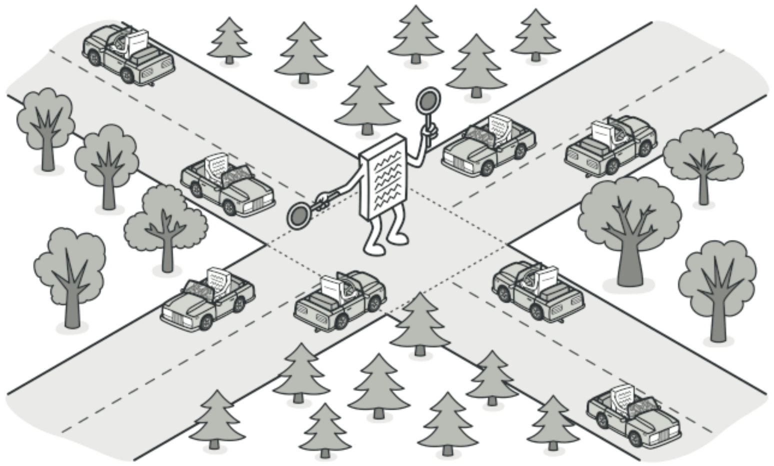
When to Apply Observer Pattern

1. Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
2. Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

Mediator Pattern

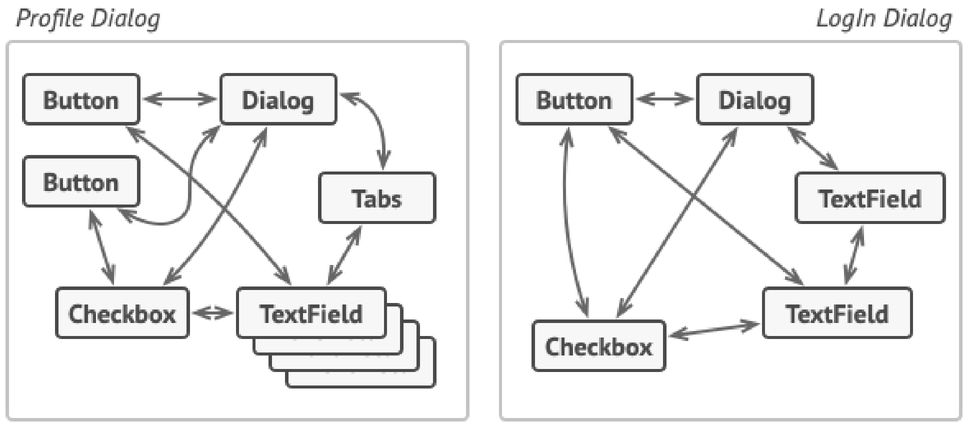
What is Mediator Pattern?

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



Problem Domain

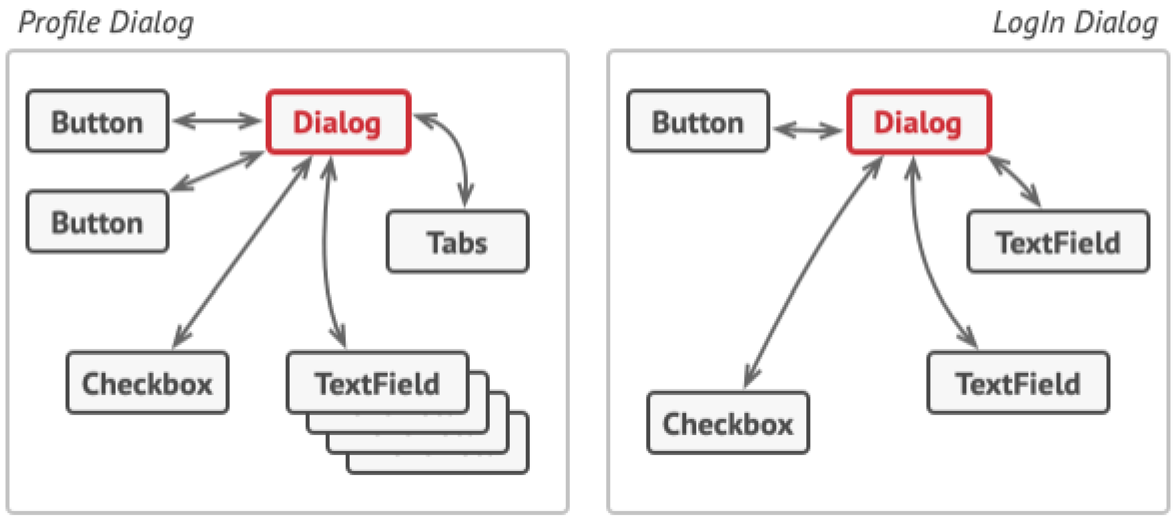
Say you have a dialog for creating and editing customer profiles. It consists of various form controls such as text fields, checkboxes, buttons, etc.



Relations between elements of the user interface can become chaotic as the application evolves.

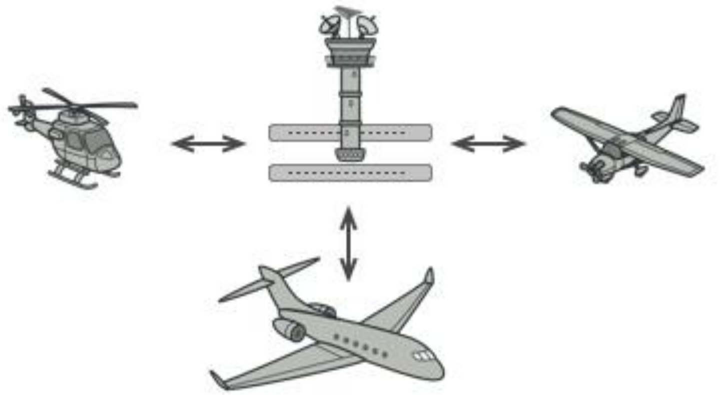
Some of the form elements may interact with others. For instance, selecting the "I have a dog" checkbox may reveal a hidden text field for entering the dog's name. Another example is the submit button that has to validate values of all fields before saving the data.

Solution



UI elements should communicate indirectly, via the mediator object.

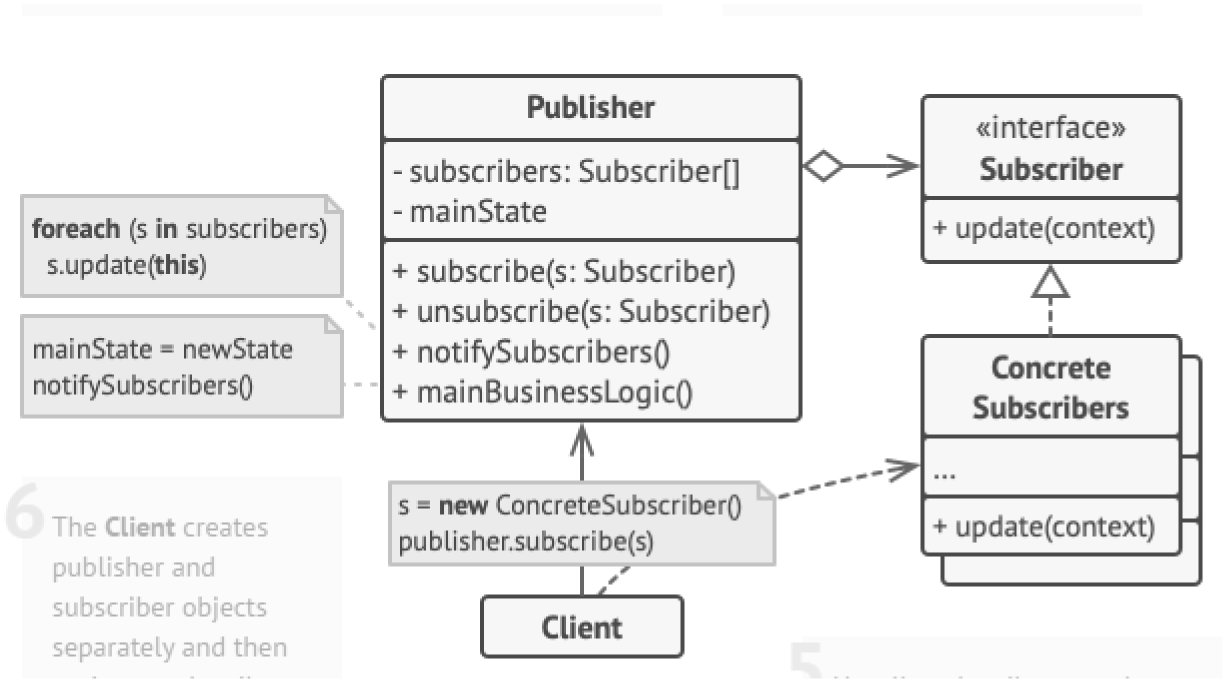
Another Problem Scenario



Aircraft pilots don't talk to each other directly when deciding who gets to land their plane next. All communication goes through the control tower.

Visiting the store vs. sending spam

UML Diagram of Mediator Pattern


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
Concrete Subscribers
...
+ update(context)
Client
6
The Client creates publisher and subscriber objects separately and then
5

```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

Steps to Implement Mediator Pattern

Step 1: Identify components that interact heavily and create a Mediator interface/class that will coordinate them.

Step 2: Each participant holds a reference to the mediator instead of referencing each other directly.

Step 3: Implement concrete mediator logic (e.g., notify(sender, event) or specific methods) that routes requests/notifications between participants.

Step 4: Make concrete colleague classes call the mediator whenever they need to communicate, and react to callbacks from the mediator instead of directly calling other colleagues.

When to Apply Mediator Pattern

1. Use the Mediator pattern when it’s hard to change some of the classes because they are tightly coupled to a bunch of other classes.
2. Use the pattern when you can’t reuse a component in a different program because it’s too dependent on other components.
3. Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.