(GOF) Gang of Four Design Pattern Java

The book "**Design Patterns**: Elements of Reusable Object-Oriented Software" has 23 design patterns, which are grouped together as **Gangs of Four Design Patterns**.

Types of GoF Design Patterns

Three categories make up the GoF Design Patterns:

1. **Creational Patterns**: For creating objects in a flexible and reusable way.
2. **Structural Patterns**: For organizing classes and objects to create larger structures.
3. **Behavioral Patterns**: For defining how objects interact and share responsibility.

Designs used in the creation

The category of creational design patterns has 5 patterns.

Pattern Name	**Description**
Singleton	Ensures that only one instance of a class is created and provides a global point of access to that instance. **Example-** Configuration management, Database connection, Thread Pool Management, Logging and Caching, File System or Resource Manager, Service Locator
Factory	Provides a way to create objects without specifying their exact class type. **Example- Notification System, Shape Creation,** **1.Database Connection**

	Use Case: Creating connections to different databases (e.g., MySQL, PostgreSQL, Oracle) based on configuration. 2. **Credit Card Validator** **Use Case**: Creating validators for different credit card types (e.g., Visa, MasterCard). **3.File Reader** **Use Case**: Reading files in different formats (e.g., Text, CSV, XML).

Abstract Factory	The **Abstract Factory Design Pattern** is a **creational design pattern** used when you need to create families of related objects without specifying their concrete classes. **Example-** **1.Database Connection Factory** **Use Case**: Connect to different databases (e.g., MySQL, PostgreSQL, MongoDB) based on configuration. 2. **Payment Gateway Integration** **Use Case**: Integrate with different payment gateways (e.g., PayPal, Stripe) while maintaining a consistent interface. 3. **Cloud Service Providers** **Use Case**: Interact with different cloud providers (e.g., AWS, Azure, Google Cloud) for storage or compute services. 4. **GUI Framework (Cross-Platform UI)** **Use Case**: Create a family of UI components (e.g., Buttons, Checkboxes) for different platforms (e.g., Windows, Mac, Linux).

Builder	The **Builder Design Pattern** is a **creational design pattern** used to construct complex objects step by step. Unlike other creational patterns, it allows for the creation of objects with varying configurations without requiring a large number of constructors. step-by-step creation of an object and a way to obtain the object instance. Helps construct complex objects step-by-step. **Example-** **1.Building Immutable Objects (e.g., User Profile)** **Use Case**: Creating a user profile where some fields (e.g., name, age) are required, and others (e.g., address, email, phone) are optional. 2. **Building Complex Queries (e.g., SQL Queries)** **Use Case**: Constructing SQL queries dynamically with optional WHERE clauses and sorting. 3. **HTTP Requests (REST Clients)** **Use Case**: Constructing HTTP requests dynamically in a REST client.

Prototype	The **Prototype Design Pattern** is a **creational design pattern** that allows you to create new objects by copying (or cloning) an existing object instead of creating one from scratch. This is useful when object creation is expensive or when you want to create objects with the same configuration. The **Prototype Design Pattern** is a **creational design pattern** used to create duplicate objects or clones of an existing object while ensuring performance and efficiency. This pattern is particularly useful when object creation is costly (e.g., due to complex initialization) or when the exact type of the object isn't known until runtime. In the **Prototype Pattern**, objects are created by copying (cloning) an existing instance rather than instantiating a new object. It is implemented using a clone() method, and typically, the Cloneable interface in Java. **Example-** **1.Performance Optimization for Costly Object Creation** • Used when creating new objects from scratch is computationally expensive (e.g., when initialization involves database operations or complex calculations). **2. Dynamic Object Creation**

	• Helps in scenarios where the type of the object to be created is decided at runtime. **3. Avoiding Complex Constructors** • Simplifies object creation by copying an existing prototype instead of repeatedly using constructors with many parameters. **4. Object Caching** • Creates and caches frequently used objects to minimize re-creation. **5. Applications with Similar Objects** • Useful in GUI applications, game development, or document editing software where similar objects (e.g., shapes, enemies, components) need to be created repeatedly.

Coding of above Java API Design Pattern Below

1.Singleton Design Pattern

```
class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

        System.out.println(s1 == s2); // true, both refer to the same instance
    }
}
```

2.Factory Design Pattern

```
 // Currency Interface
public interface Currency {
    String getSymbol();
}

// USD Currency Implementation
public class USDCurrency implements Currency {
    @Override
    public String getSymbol() {
        return "$"; // Dollar symbol
    }
}

// INR Currency Implementation
public class INRCurrency implements Currency {
    @Override
    public String getSymbol() {
        return "₹"; // Rupee symbol
    }
}
```

```
// EUR Currency Implementation
public class EURCurrency implements Currency {
    @Override
    public String getSymbol() {
        return "€"; // Euro symbol
    }
}
// Currency Factory
public class CurrencyFactory {

    public static Currency getCurrency(String country) {
        if (country.equalsIgnoreCase("USA")) {
            return new USDCurrency();
        } else if (country.equalsIgnoreCase("India")) {
            return new INRCurrency();
        } else if (country.equalsIgnoreCase("Europe")) {
            return new EURCurrency();
        }
        throw new IllegalArgumentException("No such currency for country: " + country);
    }
}

public class FactoryPatternDemo {
    public static void main(String[] args) {
        // Get currency for USA
        Currency usd = CurrencyFactory.getCurrency("USA");
        System.out.println("USA Currency Symbol: " + usd.getSymbol());

        // Get currency for India
        Currency inr = CurrencyFactory.getCurrency("India");
        System.out.println("India Currency Symbol: " + inr.getSymbol());

        // Get currency for Europe
        Currency eur = CurrencyFactory.getCurrency("Europe");
        System.out.println("Europe Currency Symbol: " + eur.getSymbol());
    }
}
Output

USA Currency Symbol: $
India Currency Symbol: ₹
Europe Currency Symbol: €
```

2.Abstract Factory Design Pattern

```
// Abstract Product: Currency
public interface Currency {
    String getSymbol();
}
```

```
// Abstract Product: Country
public interface Country {
   String getName();
}
// Concrete Products for Asia
public class AsianCurrency implements Currency {
   @Override
   public String getSymbol() {
      return "₹"; // Indian Rupee symbol
   }
}

public class AsianCountry implements Country {
   @Override
   public String getName() {
      return "India";
   }
}
// Concrete Products for Europe
public class EuropeanCurrency implements Currency {
   @Override
   public String getSymbol() {
      return "€"; // Euro symbol
   }
}

public class EuropeanCountry implements Country {
   @Override
   public String getName() {
      return "Germany";
   }
}
// Abstract Factory
public interface RegionFactory {
   Currency createCurrency();
   Country createCountry();
}
// Factory for Asia
public class AsiaFactory implements RegionFactory {
   @Override
   public Currency createCurrency() {
      return new AsianCurrency();
   }

   @Override
   public Country createCountry() {
      return new AsianCountry();
   }
}
// Factory for Europe
public class EuropeFactory implements RegionFactory {
```

```
    @Override
    public Currency createCurrency() {
        return new EuropeanCurrency();
    }

    @Override
    public Country createCountry() {
        return new EuropeanCountry();
    }
}

// Factory Producer
public class FactoryProducer {
    public static RegionFactory getFactory(String region) {
        if (region.equalsIgnoreCase("Asia")) {
            return new AsiaFactory();
        } else if (region.equalsIgnoreCase("Europe")) {
            return new EuropeFactory();
                    }
                                throw new IllegalArgumentException("Invalid region: " + region);
    }
}
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        // Get factory for Asia
        RegionFactory asiaFactory = FactoryProducer.getFactory("Asia");
        Currency asianCurrency = asiaFactory.createCurrency();
        Country asianCountry = asiaFactory.createCountry();
        System.out.println("Asia Region - Country: " + asianCountry.getName() + ", Currency: " +
asianCurrency.getSymbol());

        // Get factory for Europe
        RegionFactory europeFactory = FactoryProducer.getFactory("Europe");
        Currency europeanCurrency = europeFactory.createCurrency();
        Country europeanCountry = europeFactory.createCountry();
        System.out.println("Europe Region - Country: " + europeanCountry.getName() + ", Currency: " +
europeanCurrency.getSymbol());
            }
}
```

Output

```
Asia Region - Country: India, Currency: ₹
Europe Region - Country: Germany, Currency: €
USA Region - Country: United States, Currency: $
```

4.Builder Design Pattern

```
// The class to be built
public class User {
    // Required fields
    private final String name;
```

```
    private final int age;

    // Optional fields
    private final String address;
    private final String email;
    private final String phone;

    private User(UserBuilder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.address = builder.address;
        this.email = builder.email;
        this.phone = builder.phone;
    }

    // Static nested Builder class
    public static class UserBuilder {
        private final String name;
        private final int age;

        private String address;
        private String email;
        private String phone;

        // Constructor for required fields
        public UserBuilder(String name, int age) {
            this.name = name;
            this.age = age;
        }

        // Optional fields with setters returning the builder object
        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }

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

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

        // Build method to create User object
        public User build() {
            return new User(this);
        }
    }
```

```
    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", address='" + address + '\'' +
                ", email='" + email + '\'' +
                ", phone='" + phone + '\'' +
                '}';
    }
}

// Client Code
public class Main {
    public static void main(String[] args) {
        // Creating a User with required fields only
        User user1 = new User.UserBuilder("John Doe", 30).build();

        // Creating a User with additional optional fields
        User user2 = new User.UserBuilder("Jane Doe", 25)
                .address("123 Main Street")
                .email("jane.doe@example.com")
                .phone("555-1234")
                .build();

        System.out.println(user1);
        System.out.println(user2);
    }
}
Output:

User{name='John Doe', age=30, address='null', email='null', phone='null'}
User{name='Jane Doe', age=25, address='123 Main Street', email='jane.doe@example.com',
phone='555-1234'}
```

5.Prototype Design Pattern

```
//Prototype Interface

public interface Prototype {
    Prototype clone();
}

//Concrete Prototype

public class Shape implements Prototype {
    private String type;
    private String color;
```

```
    public Shape(String type, String color) {
        this.type = type;
        this.color = color;
    }

    @Override
    public Prototype clone() {
        return new Shape(this.type, this.color); // Cloning the object
    }

    @Override
    public String toString() {
        return "Shape [type=" + type + ", color=" + color + "]";
    }
}
public class Main {
    public static void main(String[] args) {
        // Original object (prototype)
        Shape originalShape = new Shape("Circle", "Red");

        // Cloning the original object
        Shape clonedShape = (Shape) originalShape.clone();

        // Verify that the cloned object is independent
        System.out.println("Original: " + originalShape);
        System.out.println("Cloned: " + clonedShape);
    }
}
```

Output

```
Original: Shape [type=Circle, color=Red]
Cloned: Shape [type=Circle, color=Red]
```

Designs used in the **Structural**

The category of creational design patterns has 5 patterns.

Patterns for Structural Design

The Gangs of Four design patterns book lists 7 structural design patterns.

Pattern Name	Description
Adapter	The **Adapter Design Pattern** is a structural design pattern that acts as a bridge between two incompatible interfaces. It allows objects with

	different interfaces to work together by converting the interface of one class into another that the client expects. Think of it like a **power adapter** for your laptop: you plug the adapter into an outlet with one type of socket and it converts the electricity so your laptop can charge. Similarly, in programming, an adapter allows incompatible classes to communicate. It allows two incompatible interfaces, classes, and services to communicate with each other. These incompatible components can talk to each other unless they change their code or the client's behavior. It takes input from one component (client), converts it to the expected format before giving it to the other component. We often get confused that the Adapter and the Bridge patterns are the same, however, they are not. The Adapter works on existing incompatible components, whereas the Bridge pattern is an up-front design. **Example-** **Uses Cases** 1. **Integration with Legacy Code**: a. When integrating a new system with an existing system that uses different interfaces. 2. **Third-Party Library Integration**: a. When you want to use a third-party library that has a different interface than what your application expects. 3. **Cross-Platform Applications**: a. Adapting functionality for different operating systems or devices. 4. **Data Transformation**: a. When you need to convert data from one format to another.
Bridge	The bridge design pattern is a type of structural design pattern which is used to split a large class into two separate inheritance hierarchies

	(a collection of 'is-a' relationships); one for the implementations and one for the abstractions. These hierarchies are then connected to each other via object composition, forming a bridge-like structure.
Composite	The **Composite Design Pattern** is a **structural design pattern** that allows you to treat individual objects and groups of objects (composites) in the same way. It creates a tree-like structure where objects can represent both individual components (leaves) and collections of components (composites). In simple terms: • **Leaf**: An object that doesn't have any child objects (like a single file). • **Composite**: An object that contains other objects (like a folder that can contain files or other folders). **Use Cases of Composite Design Pattern** 1. **File System Structure**: o Files and directories, where directories can contain files or other directories. 2. **UI Component Hierarchy**: o A graphical user interface where components like buttons, text fields, and panels can be nested inside other components. 3. **Organization Chart**: o Employees in a company where managers can have subordinates, and subordinates can themselves be managers of other employees. 4. **Menu System**: o A menu structure where menus can have submenus, and submenus can have further options.

Decorator	Decorator design pattern is one of the structural design pattern. **Decorator design pattern** is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior.
Flyweight	A *Flyweight Pattern* says that just to reuse already existing similar kind of objects by storing them and create new object when no matching object is found.
Facade	*The **Facade Design Pattern** is a **structural design pattern** that provides a unified, simple interface to a set of interfaces in a subsystem. It acts as a facade or "front-facing interface" that hides the complexities of the system and makes it easier for the client to interact with it. Instead of exposing all the internal components, the Facade class simplifies the process by providing a single entry point.* *Facade pattern* hides the complexities of the system and provides an interface to the client using which the client can access the system. The **Facade Design Pattern** is a **structural design pattern** that provides a simple interface (facade) to a complex subsystem of classes. It hides the complexity of the system and makes it easier to use. Think of a **facade** as the front desk in a hotel: • Instead of interacting with each department (housekeeping, room service, maintenance), you interact with the front desk (facade), and they handle the rest for you. **Use Cases of Facade Design Pattern in Java Applications** 1. **Complex Subsystem Management**: ○ When you have a complex subsystem (e.g., multiple classes with complicated dependencies), you can simplify its usage for the client by providing a facade. 2. **Third-Party Integration**: ○ Wrapping third-party libraries with a facade to provide an easy-to-use API and hide external complexities.

	3. **E-commerce Platforms**:
	   - A facade can unify subsystems like inventory management, payment processing, and order management into a single interface for users.
	4. **Payment Gateway**:
	   - A facade simplifies integration with multiple payment processors by exposing a single interface for handling payments.
	5. **Report Generation**:
	   - If a system involves multiple services (e.g., data fetching, filtering, and formatting), a facade can combine these steps into a single, simplified process.
	6. **Microservices Communication**:
	   - Facade can act as an aggregator for multiple microservices, exposing a unified interface for the client to interact with.

Proxy	It is used to restrict access to another object, create a substitute or stand-in for it.
	The **Proxy Design Pattern** is a **structural design pattern** that provides a surrogate or placeholder for another object to control access to it. It is commonly used to add an extra layer of functionality, such as security, lazy initialization, or performance optimization, without modifying the actual object.
	Use Cases of Proxy Design Pattern
	1. **Virtual Proxy**:
	   - Used for lazy initialization or on-demand loading of a resource-heavy object.
	   - Example: Loading a large image or video file only when it is required to be displayed.
	2. **Remote Proxy**:

	◦ Represents an object that exists in a remote location (e.g., another server). ◦ Example: Communicating with a remote service in distributed systems. 3. **Protection Proxy**: ◦ Controls access to an object by adding security checks. ◦ Example: User authentication and authorization for accessing specific services. 4. **Logging or Auditing Proxy**: ◦ Adds logging functionality for monitoring method calls and usage. ◦ Example: Logging database queries or API calls. 5. **Cache Proxy**: ◦ Caches the result of expensive operations for subsequent calls. ◦ Example: Caching data fetched from a database to reduce load.

Patterns for Behavioral Design

The GoF design patterns list eleven behavioral design patterns.

Pattern Name	Description
Chain of responsibility	The **Chain of Responsibility Design Pattern** is a behavioral design pattern that allows a request to pass through a chain of handlers. Each handler in the chain decides whether to process the request or pass it to the next handler in the chain. **Simple Explanation** Think of a customer service system where a support request can be handled by different departments, such as Level 1 Support, Level 2 Support, or a Manager. If one department

	can't handle the request, it is passed to the next one in the chain. This pattern is useful when multiple handlers can process a request, but the handler is determined dynamically at runtime. A request from either the client is delivered to a chain of objects to be processed as part of the Chain of Responsibility technique used in software design to promote loose coupling.
Command	The **Command Design Pattern** is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request. This allows the request to be parameterized, queued, logged, or undone/redone. It is used to implement loose coupling in a request-response model, utilize the Command Command Pattern.
Interpreter	Explains how to express a language's grammar and offers an interpreter to deal with it. The Interpreter design pattern is a behavioral design pattern that defines a way to interpret and evaluate language grammar or expressions. It provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes. Each class represents a rule or expression in the grammar, and the pattern allows these classes to be composed hierarchically to interpret complex expressions.
Iterator	An iterator was once used to offer a common method of browsing through a collection of objects.
Mediator	A mediator is a device that offers a centralized communication channel between various system elements.
memento	When we want to store an object's state for subsequent restoration, we utilize the memento design pattern.
Observer	The **Observer Design Pattern** is a **behavioral design pattern** where **one object (Subject)** maintains a list of **other objects (Observers)** and notifies them whenever there is a change in

	its state. It is like a **publish-subscribe mechanism**, where multiple objects are "subscribed" to listen for changes in one object.
	
	The **Observer Design Pattern** is a behavioral design pattern that defines a one-to-many dependency between objects. When the **subject** (the object being observed) changes its state, all its **observers** are notified and updated automatically.
	
	Real-Life Example
	
	Imagine you're subscribed to a YouTube channel. When the channel uploads a new video, you (the observer) are notified. The channel acts as the **subject** and you (the subscriber) are the **observer**.
	
	Use Cases of Observer Pattern
	
	1. **Event-driven systems**:
	  - Notifications in user interfaces.
	  - Stock price updates in trading apps.
	2. **Messaging systems**:
	  - Real-time notifications in chat or social media apps.
	3. **Publish-subscribe systems**:
	  - News feed updates in apps like Facebook or Twitter.

Strategy	The **Strategy Design Pattern** is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. Instead of implementing multiple algorithms directly in the class, the Strategy pattern encapsulates each algorithm in its class and makes them interchangeable. When there are several algorithms available for a given task and the client chooses the actual implementation that will be used at runtime, the strategy pattern is employed.

	Simple Explanation Imagine you're building a payment system that can handle multiple payment methods (e.g., Credit Card, PayPal, Google Pay). Each payment method has a different algorithm for processing payments. Using the Strategy pattern, you can encapsulate each payment method in a separate class and select the desired one at runtime.
State	When an object changes its behavior based on its internal state, the state design pattern is applied.
Template Method	The **Template Method Design Pattern** is a behavioral design pattern that defines the skeleton of an algorithm in a base class and lets subclasses override specific steps without changing the overall structure of the algorithm. This pattern is useful when you have an algorithm that has fixed steps, but some steps may vary depending on the implementation in subclasses. Deferring some of the implementation tasks to the subclasses through the usage of a template method. **Use Cases** 1. **Report Generation**: Generating reports where the structure is the same, but the data source may vary (e.g., PDF, Excel, HTML reports). 2. **Game Development**: Defining the flow of a game (e.g., initialize game, start game, end game), while allowing different game types to implement specific details. 3. **Order Processing Systems**: As demonstrated in the example, where the process is consistent but payment and delivery methods vary. 4. **UI Frameworks**: Defining a template for rendering a UI component (e.g., setup, draw, teardown), with customization allowed in certain steps.

Visitor	When we need to conduct an operation on a collection of objects that are all of a similar type, we utilize the visitor pattern. The **Visitor Design Pattern** is a **behavioral design pattern** that allows you to **add new operations** to a group of related classes without modifying their structure. Instead of putting the behavior in the class itself, the operation is defined in a separate object (the **Visitor**), which you "visit" each class with. It’s like having a visitor in your house: • The **Visitor** (a guest) performs actions (operations) on various **Elements** (rooms in your house) without changing the house itself.