🎯 Java Design Patterns (GoF) Collection

Welcome to the **Java Design Patterns (GoF)** repository! This project demonstrates the power of **Gang of Four (GoF)** design patterns through real-world **mini systems** implemented in Java. Each pattern is neatly explained and organized into categories with dedicated examples.

🧠 About the Project

This repository is built to help Java developers **learn, understand, and implement** core **Design Patterns** through practical, easy-to-understand **mini Java systems**. It covers the three main categories:

- 🛠️ **Creational Patterns**
- 🏗️ **Structural Patterns**
- 🔁 **Behavioral Patterns**

All examples follow **best coding practices** and include visual aids and commentary to enhance comprehension.

📁 Project Structure

- 📂 Example path: Design-Patterns/ Creational Patterns/ AbstractFactory Pattern/ src/ Food_Ordering_System/

Each pattern is organized in its own folder and includes:

- 📄 Java source code
- 🧪 A mini project/system implementing the pattern
- 📊 Inline comments, structure diagram, class diagram, participants, and consequences

---

🧰 Covered Design Patterns(TO-DO List)

🔨 **Creational Patterns**	🏗️ **Structural Patterns**	🔁 **Behavioral Patterns**
✅ Factory	❌ Adapter	✅ Chain of Responsibility(COR)
✅ Abstract Factory	❌ Bridge	❌ Command
❌ Builder	❌ Composite	❌ Iterator
❌ Prototype	✅ Decorator	❌ Mediator
✅ Singleton	✅ Facade	❌ Memento
	❌ Flyweight	✅ Observer
	✅ Proxy	✅ State

🔨 Creational Patterns	🏗 Structural Patterns	🔁 Behavioral Patterns
		❌ Strategy
		❌ Template Method
		❌ Visitor

✅ = Implemented | ❌ = Coming Soon

💻 Language & Tools

- 💡 **Language:** Java (JDK 8+)
- 🧱 **Architecture:** Object-Oriented Programming (OOP)
- ✅ **Status:** All implemented patterns are compiled and tested with sample use cases

---

📓 What is a Design Pattern?

✅ Definition (Based on GoF)

A **design pattern** is a **reusable solution** to a **common problem** that occurs in software design.

It is **not a complete code**, but a **template or guide** for how to solve a problem that keeps coming up in different projects.

> **GoF Definition:**
> "*A design pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.*"

🧠 Key Points

- Design patterns are **tried and tested solutions**.
- They are **language-independent** (but can be implemented in Java, C++, etc.).
- Improve **code reuse**, **flexibility**, and **maintainability**.
- GoF categorized **23 design patterns** into:
  - **Creational Patterns**
  - **Structural Patterns**
  - **Behavioral Patterns**

🧭 How to Select a Design Pattern & Solve Design Problems ? (GoF Way)

✅ Step-by-Step Approach from GoF

The **Gang of Four (GoF)** suggests using patterns by **understanding the problem first**, then finding a pattern that fits. Here's how:

🛠 1. Understand the Problem

- **What are you trying to solve?**
- Is it about **object creation**, **object structure**, or **object behavior**?
- Example questions:
  - Do you want to restrict object creation? (→ *Creational*)
  - Do you want to make objects easier to change or extend? (→ *Structural*)
  - Do you want objects to communicate flexibly? (→ *Behavioral*)

🔍 2. Classify the Problem

Check what kind of problem you're solving:

Type	Pattern Category
Object creation	Creational
Class structure	Structural
Object behavior	Behavioral

🧩 3. Look for Similar Use Cases

Read the **intent** and **applicability** section of each pattern in the GoF book.

Example:

> Want to ensure only one instance of a class?
> ✅ Use **Singleton Pattern**

💡 What Do You Mean by "Instance" in Java?

✅ Definition

An **instance** is a **concrete object** created from a **class**.
You can think of a class as a **blueprint**, and an instance as the **actual object** built using that blueprint.

🏠 Real-life Analogy

- **Class** = House Blueprint
- **Instance** = The actual House built from that blueprint

So when you create an object using `new`, you're **instantiating** a class — creating an instance!

☕ Java Example

```java
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    void sayHello() {
        System.out.println("Hello, my name is " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Aman"); // s1 is an instance of Student
        Student s2 = new Student("Riya"); // s2 is another instance

        s1.sayHello();  // Output: Hello, my name is Aman
        s2.sayHello();  // Output: Hello, my name is Riya
    }
}
```

🧠 4. Use Pattern Selection Hints (GoF Tips)

GoF provides clues based on problem types:

Situation	Pattern Suggestion
You want to vary object creation	Abstract Factory, Builder, Prototype
You want one object only	Singleton
You want to compose objects into trees	Composite
You want to change behavior at runtime	Strategy, State
You want to avoid tight coupling	Observer, Mediator

☕ Example: Solving with Design Pattern in Java

Problem: You need to create different types of documents (PDF, Word) but don't want to hardcode creation.

Solution: Use **Factory Method** pattern.

```java
abstract class Document {
    public abstract void open();
}

class PDFDocument extends Document {
    public void open() {
        System.out.println("Opening PDF Document");
    }
}

class WordDocument extends Document {
    public void open() {
        System.out.println("Opening Word Document");
    }
}

abstract class Application {
    public abstract Document createDocument();
}

class PDFApp extends Application {
    public Document createDocument() {
        return new PDFDocument();
    }
}

class WordApp extends Application {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

🛠️ Creational Pattern

🛠️Abstract Factory Pattern

📌 Intent

> **"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."**
> (GoF, Design Patterns: Elements of Reusable Object-Oriented Software, Page 87)

📃 Also Known As

- **Kit** (mentioned informally in the GoF book)

📈 Applicability (When to Use)

Use the Abstract Factory pattern when:

- You want to create families of related objects that must be used together.
- You want to enforce consistency among products.
- You want to hide the concrete classes from the client code.
- The system should be independent of how its products are created, composed, and represented.

(Reference: GoF, Page 88)

👥 Participants (GoF Book, Page 90)

1. **AbstractFactory**

- Declares an interface for operations that create abstract product objects.

2. **ConcreteFactory**

- Implements the operations to create concrete product objects.

3. **AbstractProduct**

- Declares an interface for a type of product object.

4. **ConcreteProduct**

- Defines a product object to be created by the corresponding concrete factory.

5. **Client**

- Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

⚠ Consequences (GoF Book, Page 91)

- ☑ It **isolates concrete classes**.
- ☑ It makes it easy to **exchange product families**.
- ☑ Promotes **consistency among products**.
- ✖ It can be **difficult to support new kinds of products** (requires changing the factory interface).

🔗 Related Patterns (GoF Book, Page 95)

Related Pattern	Why Related?
Factory Method	Abstract Factory is often implemented using Factory Methods.
Prototype	Sometimes used instead of factories if many product configurations are possible.
Builder	Used for assembling complex products step by step, but Abstract Factory focuses on families of products.

🧠 Principles Used in **Abstract Factory** Pattern

Open-Closed Principle (OCP)

- **Why it's used:** The Abstract Factory adheres to the **Open-Closed Principle**. You can introduce new families of products by adding new concrete factories, without altering existing client code.
- **Benefit:** It makes it easy to extend the system with new product families without breaking the existing system or requiring modifications to the client code.

❓ Why Use the Abstract Factory Pattern?

The **Abstract Factory** pattern is used for several important reasons. It provides a way to create families of related objects without specifying their concrete classes. Below are some key reasons to use the Abstract Factory pattern:

1. **Ensures Consistency Among Products**

- The Abstract Factory ensures that the client can only create objects that belong to the same family, thereby ensuring consistency between related objects.
- It helps in enforcing a consistent theme or style, like creating buttons, checkboxes, and other UI components with the same visual appearance across different platforms.

2. **Promotes Loose Coupling**

- Just like the Factory Method, the Abstract Factory promotes loose coupling between the client code and concrete classes.
- The client interacts only with abstract products and factories, never knowing or depending on the specific implementation of the products.
- This allows you to change the family of products without affecting the client code.

3. **Eases Product Family Variations**

- The Abstract Factory is ideal when the system should be independent of how its products are created, composed, and represented.
- It allows you to create different sets of products (like different themes or platform-specific components) without changing the rest of the system.
- You can easily add new families of related products as needed.

4. **Encourages Code Reusability**

- It fosters reusability by centralizing object creation in factory classes, ensuring that product families are created consistently throughout the application.
- The pattern allows easy extension to accommodate new product families by adding new concrete factories without modifying existing code.

5. **Facilitates Future Extension**

- Adding new types of products to a system often requires changes in the factory. With the Abstract Factory pattern, new product families can be added without altering existing client code.

- This supports the **Open-Closed Principle**, making it easy to extend the system to support new product families.

6. **Encourages Platform Independence**

- In cases where a system needs to support multiple platforms (e.g., different UI toolkits for different operating systems), the Abstract Factory pattern can help abstract platform-specific details and provide platform-independent code for the client.

7. **Simplifies Object Creation for Complex Families**

- When products within a family have complex relationships or need to be created together, the Abstract Factory provides a structured way to create the entire family without exposing the details to the client.

Example Scenario:

Consider a system that needs to create UI components such as buttons and checkboxes in two different styles: **Light Theme** and **Dark Theme**. The Abstract Factory pattern allows you to create families of related objects (buttons and checkboxes) for both themes, and the client code can interact with abstract product interfaces, keeping it decoupled from the concrete product classes.

By using the Abstract Factory pattern, you ensure that the client always creates products that belong to the same theme (Light or Dark) without having to know the details of how each product is implemented.

Thus, the **Abstract Factory** pattern helps in managing complex product families, ensuring consistency, and promoting flexibility and maintainability in your code.

🛠 Factory Method Pattern

📌 Intent

> **"Define an interface for creating an object, but let subclasses alter the type of objects that will be created."**
> This pattern allows a class to delegate the responsibility of creating an object to its subclasses, without specifying the exact class of object that will be created.

📃 Also Known As

- **Virtual Constructor**

📈 Applicability (When to Use)

Use the **Factory Method** pattern when:

- You need to create objects of a class, but you don’t want to specify the exact class of the object.
- You want to let subclasses decide which class to instantiate.
- You want to control the instantiation process of a class.
- You need to delegate the responsibility of object creation to child classes.

Example use cases:

- When the class cannot anticipate the type of objects it needs to create.
- When the client code shouldn't be aware of the specific class types and their instantiation details.

👥 Participants

1. **Creator (Abstract Class)**

- Declares the factory method, which returns an object of type Product.
- May also define a default implementation of the factory method.

2. **ConcreteCreator (Concrete Class)**

- Implements the factory method to create a specific product object.

3. **Product (Abstract Class)**

- Declares an interface for objects created by the factory method.

4. **ConcreteProduct (Concrete Class)**

- Implements the `Product` interface and defines the objects that are created by the factory method.

5. **Client**

- Uses the `Creator` class and its factory method to create a `Product`.

⚠️ Consequences

The **Factory Method** pattern has several advantages and disadvantages:

✅ Benefits:

- **Loose coupling**: The client code is decoupled from the concrete class of the object being created.
- **Subclasses can decide the product type**: The decision of which object to instantiate is delegated to subclasses.
- **More flexible and extensible**: You can introduce new products without modifying the existing code.

❌ Drawbacks:

- **Increase in number of classes**: You may end up with more classes, especially when there are several concrete product variants.
- **Requires a hierarchy of creators**: If there are multiple creators, it can lead to more complex code and hierarchy.

🔗 Related Patterns

Related Pattern	Why Related?
Abstract Factory	The Abstract Factory pattern uses Factory Methods to create a family of related objects.
Prototype	If you want to clone an object instead of creating a new one, the Prototype pattern is an alternative.
Builder	The Builder pattern creates complex objects step-by-step. It's useful when you need to control the creation process in greater detail.

🧠 Principles Used in **Factory Method** Pattern

Open-Closed Principle (OCP)

- **Why it's used:** The Factory Method follows the **Open-Closed Principle** because the client code can easily add new types of products without modifying existing code. To add a new product, you only need to introduce a new concrete class and extend the factory.
- **Benefit:** The system can evolve without affecting existing functionality, making it flexible and adaptable to future requirements.

❓ Why Use the Factory Method Pattern?

The **Factory Method** pattern is widely used for several reasons:

1. **Promotes Loose Coupling**

- The Factory Method separates the creation logic of objects from the client code. This reduces the dependency between client code and concrete product classes.
- By using the Factory Method, client code interacts with a product interface, not knowing or caring about the specific class that implements the product.

2. **Enhances Flexibility**

- The pattern provides flexibility in terms of the types of objects that can be created. New types of products can be added without modifying the client code or other parts of the application.
- You can change the product that’s created by modifying the concrete creator class, without changing the code that uses the product.

3. **Subclasses Decide the Object Creation**

- The responsibility of object creation is delegated to subclasses, allowing them to decide which specific class to instantiate. This can be especially useful when the exact type of object to be created is not known until runtime.

4. **Fosters Code Reusability**

- By defining the object creation in a common method, the code can be reused in different parts of the program or across different projects.
- If multiple creators share the same method of creating products, they can all use the same base class or interface, promoting code reuse.

5. **Helps in Code Maintenance**

- The pattern simplifies code maintenance by isolating the object creation logic from the rest of the code. Changes to how an object is created (e.g., switching from one implementation to another) can be confined to the factory method, avoiding widespread changes across the system.

6. **Encourages Open-Closed Principle**

- The Factory Method pattern encourages the open-closed principle, where you can add new product types without changing the existing code. You just need to create a new subclass of the creator and implement the factory method.

7. **Improves Testability**

- Since object creation is abstracted, it's easier to mock or substitute real product instances with test doubles or stubs during unit testing. This improves the testability of the application.

Example Scenario:

You are designing a system that needs to work with multiple types of documents (e.g., PDFs, Word files). By using the **Factory Method**, you can define an interface for document creation, allowing subclasses to decide which document type to create without modifying the existing code where documents are processed.

Thus, the **Factory Method** pattern helps in managing complex object creation while keeping your code maintainable, flexible, and scalable.

🛠️ **Singleton Pattern** (GoF)

📌 Intent

> **"Ensure a class has only one instance, and provide a global point of access to it."**
> (GoF, Design Patterns: Elements of Reusable Object-Oriented Software, Page 127)

The **Singleton pattern** ensures that a class has only one instance and provides a global access point to that instance. This pattern is often used when a single shared resource (like a configuration manager, logging service, or database connection) needs to be accessed across different parts of an application.

📃 Also Known As

- **Single Instance Pattern**
- **One Object Pattern**

📈 Applicability (When to Use)

Use the Singleton pattern when:

- You need to control access to a shared resource.
- You want to restrict the instantiation of a class to **one instance only**.
- You need a global access point to that single instance.
- You want to ensure that there is **only one object** managing a resource throughout the application's lifecycle, such as logging or configuration settings.

(Reference: GoF, Page 128)

👥 Participants (GoF Book, Page 130)

1. **Singleton**

- Ensures that only one instance of the class is created and provides a global point of access to that instance.

2. **Client**

- Accesses the singleton instance to interact with the resource managed by the Singleton class.

⚠️ Consequences (GoF Book, Page 131)

- ✅ **Controlled access to the sole instance**: Ensures that the Singleton class is the only one providing access to the resource.
- ✅ **Lazy instantiation**: The Singleton object can be created when it's first needed, not necessarily when the program starts.
- ✅ **Global point of access**: The singleton instance can be accessed from any part of the program.
- ❌ **Global state**: Having a global point of access to a single instance can lead to tight coupling and difficulty in testing because the Singleton class essentially becomes a global object.
- ❌ **Single responsibility**: The Singleton pattern can violate the **Single Responsibility Principle (SRP)** if the singleton class handles too much functionality. The global nature of the Singleton may also make the system harder to maintain.

🔗 Related Patterns (GoF Book, Page 134)

Related Pattern	Why Related?
Factory Method	Singleton can sometimes be combined with Factory Methods to manage the instantiation process of a single instance in a controlled way.
Abstract Factory	If you need to ensure a single instance of a factory, you can use the Singleton pattern to enforce this behavior.

Related Pattern	Why Related?
Proxy	A Proxy pattern can be used to manage access to the Singleton, especially if access control is needed to ensure only one client at a time interacts with it.

❓ Why Use This Pattern?

The **Singleton pattern** is used when there is a need to manage shared resources, such as database connections, configuration settings, logging, etc., throughout the application. This ensures that these resources are only initialized once and can be accessed globally.

Reasons for Using Singleton Pattern:

1. **Global Access**: It provides a global point of access to the unique instance.
2. **Control over Resource Management**: It ensures that only one instance of the resource is ever created, saving system resources and simplifying the code.
3. **Lazy Initialization**: The Singleton instance is created only when it is needed, which can optimize resource usage and application startup time.

🧠 Principles Used in Singleton Pattern

1. **Single Responsibility Principle (SRP)**

- The Singleton ensures that the responsibility of managing the resource is given to a single class, making it easier to maintain. However, if the Singleton class takes on too many responsibilities, it can violate SRP.

2. **Lazy Initialization**

- Singleton can be initialized only when it's needed. This principle helps in saving memory and improving the startup performance of the application.

3. **Encapsulation**

- The Singleton pattern encapsulates the logic of object creation and access. Clients don't need to know how the instance is created, only that they can access it globally.

🏗 Structural Patterns

🏗 Decorator Pattern

📌 **Intent**

The **Decorator Pattern** allows you to dynamically add behavior to an object at runtime, without altering its structure. It provides a flexible alternative to subclassing for extending functionality.

> **Intent (GoF):**
> *"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."*

📝 Also Known As

- **Wrapper** (Sometimes used as a synonym in certain contexts)

📈 Applicability (When to Use)

Use the **Decorator Pattern** when:

1. You want to add responsibilities to objects dynamically and selectively.
2. You want to extend the behavior of classes in a flexible and reusable way.
3. Subclassing is impractical or undesirable due to an explosion of subclasses for every combination of behaviors.

🏗 Structure Diagram

(Refer to GoF's book for detailed structural diagrams of the Decorator Pattern. The structure typically involves the **Component**, **ConcreteComponent**, **Decorator**, and **ConcreteDecorator** classes.)

👥 Participants

1. **Component**:

- Defines the interface that is common to all objects that can have responsibilities added to them dynamically.

2. **ConcreteComponent**:

- Implements the **Component** interface and defines the core functionality.

3. **Decorator**:

- Implements the **Component** interface and contains a reference to a **Component** object. It delegates requests to the **Component** and may add additional behavior.

4. **ConcreteDecorator**:

- A subclass of **Decorator** that adds responsibilities to the **ConcreteComponent**.

⚠ Consequences (GoF)

- ☑ **Favorable Consequences**:

  - **Extensibility**: It allows functionality to be added to individual objects at runtime.
  - **Flexibility**: You can combine different decorators to add varied functionality without modifying the object itself.
  - **Code Reusability**: It promotes code reuse by allowing new functionality to be added without modifying existing code.

- ❌ **Unfavorable Consequences**:

  - **Complexity**: It can introduce many small classes, leading to increased complexity in the system.
  - **Overhead**: The dynamic nature of decorators may lead to slight performance overhead.

🔗 Related Patterns (GoF)

Related Pattern	Why Related?
Adapter	Both **Adapter** and **Decorator** provide ways to add functionality, but the main difference is that **Adapter** changes the interface to make it compatible, while **Decorator** adds responsibilities to the object dynamically.
Composite	**Composite** and **Decorator** both involve treating individual objects and compositions of objects uniformly. However, **Composite** deals with structuring objects in a tree, while **Decorator** adds behavior to objects.

🎯 Why Use This Pattern?

The **Decorator Pattern** is useful when you want to add additional behavior to an object, but subclassing is not a feasible option due to the combinatorial explosion of subclasses or because you need the flexibility to add and remove behaviors at runtime. By using decorators, you can compose new behaviors without altering existing code.

💡 Principle Applied

Favor Composition Over Inheritance:
The **Decorator Pattern** applies the principle of **composition over inheritance**. Instead of using inheritance to add new behavior, it uses composition to add functionality to objects dynamically, making it a more flexible and reusable design.

🏗 Facade Pattern

📌 Intent

The **Facade Pattern** provides a simplified interface to a complex subsystem, hiding the complexities and making the subsystem easier to use for the client. It provides a higher-level interface that makes the subsystem easier to access and interact with.

> **Intent (GoF):**
> *"Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use."*

📝 Also Known As

- **Facade**

📈 Applicability (When to Use)

Use the **Facade Pattern** when:

1. You need to provide a simple interface to a complex subsystem of classes.
2. You want to reduce dependencies and simplify interaction with the system for clients.
3. You need to make a system easier to use by decoupling the client code from complex subsystem components.

🏗 Structure Diagram

(Refer to GoF's book for detailed structural diagrams of the Façade Pattern. The structure typically involves **Facade** and **Subsystem** classes.)

👥 Participants

1. **Facade**:

- Defines a simplified interface to the subsystem. It delegates client requests to the appropriate subsystem components.

2. **Subsystem classes**:

- These classes implement the subsystem functionality. The **Facade** delegates client requests to them, simplifying the interaction with these components.

⚠ Consequences (GoF)

- ☑ **Favorable Consequences**:
  - **Simplified interface**: Reduces the complexity for the client by providing a simplified interface to the subsystem.
  - **Loose coupling**: Clients are decoupled from the subsystem, which reduces dependencies and makes the system easier to maintain.
  - **Easy integration**: A Façade can be used to integrate multiple subsystems into a single interface.
- ❌ **Unfavorable Consequences**:

  - **Potential for oversimplification**: If the Façade becomes too abstract, it may hide important functionality that the client may need access to.
  - **Tight coupling to Façade**: Clients become dependent on the Façade, which may limit flexibility in some cases.

🔗 Related Patterns (GoF)

Related Pattern	Why Related?
Adapter	Both **Facade** and **Adapter** simplify interfaces. However, **Adapter** works by changing the interface of an existing class, while **Facade** simplifies the interface to a group of classes.
Proxy	The **Proxy** pattern also provides a surrogate interface, but while the Façade simplifies access to a subsystem, the Proxy may be used for controlling access, lazy initialization, or other purposes.

🎯 Why Use This Pattern?

The **Façade Pattern** is useful when you want to reduce the complexity of interacting with a complex system. By providing a single, unified interface to the subsystem, it simplifies the process for the client and makes the system easier to maintain and integrate.

💡 Principle Applied

Encapsulation:
The **Façade Pattern** adheres to the principle of **Encapsulation** by hiding the complexities of the subsystem from the client. This allows the subsystem's internal structure to change without affecting the client code, which only interacts with the simplified interface provided by the Façade.

🏗️ Proxy Pattern

📌 Intent

The **Proxy Pattern** provides a surrogate or placeholder for another object to control access to it. This control can include access control, lazy initialization, logging, remote access, and more.

> **Intent (GoF):**
> *"Provide a surrogate or placeholder for another object to control access to it."*

📝 Also Known As

- Surrogate

📈 Applicability (When to Use)

Use the **Proxy Pattern** when:

1. You need to **control access** to an object for reasons like security, logging, or lazy loading.
2. The object is **expensive to create**, and you want to instantiate it only when necessary (Virtual Proxy).
3. The object is located **remotely**, and a local object represents it (Remote Proxy).
4. You want to add functionality like **caching or counting references** (Smart Reference Proxy).

👥 Participants

1. **Subject (Interface)**:

- Declares the common interface for RealSubject and Proxy so they can be used interchangeably.

2. **RealSubject**:

- The real object that the proxy represents and to which it delegates the requests.

3. **Proxy**:

- Maintains a reference to the RealSubject and controls access to it. It may also add additional behavior.

⚠ Consequences (GoF)

- ✅ **Favorable Consequences**:
  - **Access control**: You can control access to the object (e.g., authentication).
  - **Performance optimization**: Delay the creation of costly objects (lazy loading).
  - **Additional behavior**: Add functionality without modifying the real subject.
- ❌ **Unfavorable Consequences**:
  - **Increased complexity**: Adds extra classes and layers of indirection.
  - **Overhead**: May introduce performance overhead, especially if the proxy is not lightweight.

🧾 Types of Proxy

1. **Virtual Proxy**:

- Delays creation and initialization of expensive objects.

2. **Remote Proxy**:

- Represents an object in a different address space (e.g., server).

3. **Protection Proxy**:

- Controls access based on access rights.

4. **Smart Reference Proxy**:

- Adds extra behavior when the object is accessed (e.g., logging, counting).

🔗 Related Patterns (GoF)

Related Pattern	Why Related?
Decorator	Both add behavior, but Decorator enhances behavior of the object, while Proxy controls access.
Adapter	Both involve wrapping another class, but Adapter changes the interface while Proxy controls access.
Façade	Façade simplifies access to subsystems, Proxy controls access to specific objects.

🎯 Why Use This Pattern?

The **Proxy Pattern** is useful when you want to **add a layer of control** over how and when clients interact with a particular object. This is especially important when dealing with remote objects, expensive object creation, or when enforcing access permissions.

💡 Principle Applied

Single Responsibility Principle (SRP):
The Proxy Pattern separates the responsibility of access control or additional behavior from the actual business logic of the object (RealSubject). Each class has a clear responsibility: the proxy manages access, and the real subject performs the core functionality.

🔁 Behavioral Patterns

🔁 Chain of Responsibility Pattern

📌 Intent

> **“Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.**
> **Chain the receiving objects and pass the request along the chain until an object handles it.”**
> — *Design Patterns: Elements of Reusable Object-Oriented Software (GoF)*

📃 Also Known As

- CoR

📈 Applicability (When to Use)

Use the Chain of Responsibility pattern when:

- More than one object may handle a request, and the handler isn't known in advance.
- You want to decouple the sender and receiver of a request.
- You want to issue a request to multiple objects without specifying the receiver explicitly.
- The set of handlers should be specified dynamically.

👥 Participants (GoF Book)

1. **Handler (Abstract Class or Interface)**

- Defines an interface for handling requests.
- Optionally implements the successor link (next handler).

2. **ConcreteHandler**

- Handles the request or forwards it to the next handler.

3. **Client**

- Initiates and configures the chain of handlers.

⚖️ Consequences (GoF Book)

- ✅ Loose coupling between sender and receiver.
- ✅ New handlers can be added easily without modifying existing code.
- ✅ Simplifies the object that sends the request.
- ❌ No guarantee that every request will be handled unless a default handler is provided.
- ❌ Can be hard to follow the request path when debugging.

🔗 Related Patterns

Related Pattern	Why Related?
Composite	CoR can use a tree structure for handlers.
Command	Command objects can be processed by a chain of handlers.
Decorator	Both can modify behavior dynamically and rely on recursive structures.
Mediator	Mediator centralizes control; CoR distributes it across the chain.

❓ Why Use This Pattern?

- To **avoid tight coupling** between request senders and receivers.
- To **delegate responsibility** across a chain of objects dynamically.

- To promote **Open/Closed Principle** by allowing new handlers to be added without changing existing code.
- To make the **request processing flexible** and configurable at runtime.

Principle Used

Single Responsibility Principle (SRP)

> Each handler class has **one reason to change** — how it handles the request.

Open/Closed Principle (OCP)

> You can **add new handlers** to the chain without changing existing ones.

Getting Started

1. **Clone the Repository**:

```
git clone https://github.com/SonaniAkshit/Design-Patterns-Java.git
cd Design-Patterns-Java
```

State Pattern

Intent

> **"Allow an object to alter its behavior when its internal state changes.**
> **The object will appear to change its class."**
> — *Design Patterns: Elements of Reusable Object-Oriented Software (GoF)*

Also Known As

- Objects for States

Applicability (When to Use)

Use the **State pattern** when:

- An object's behavior depends on its state, and it must change its behavior at runtime depending on that state.
- Code has numerous `if` or `switch` statements based on object state.
- You want to avoid cluttering a single class with behavior for many states.

👥 Participants (GoF Book)

1. **Context**

- Maintains an instance of a ConcreteState subclass that defines the current state.
- Allows clients to change the state of the object.

2. **State (Interface or Abstract Class)**

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

3. **ConcreteState**

- Implements behavior associated with a state of the Context.

⚖️ Consequences (GoF Book)

- ☑ **Organizes state-specific behavior** into separate classes.
- ☑ Makes adding new states easy by creating new subclasses.
- ☑ Eliminates conditional statements in the context class.
- ☑ Improves code clarity by separating state-specific logic.
- ❌ Can result in many small classes.
- ❌ Context can become dependent on state subclasses.

🔗 Related Patterns

Related Pattern	Why Related?
Strategy	Similar structure, but State transitions are automatic; Strategy is chosen explicitly.
Flyweight	Flyweight can be used to share state objects.
Observer	State objects can notify observers when the state changes.

❓ Why Use This Pattern?

- To **cleanly separate behaviors** based on object state.
- To **eliminate long chains of conditionals** in code (e.g., `if-else`, `switch`).
- To make code **easier to maintain and extend** by adding new states as new classes.
- To let the object **change behavior dynamically** without changing the class.

🧭 Principle Used

▧ Open/Closed Principle (OCP)

> New states can be added without modifying the context class.

▧ Single Responsibility Principle (SRP)

> Each state class encapsulates behavior specific to a single state.

🔁 Strategy Pattern

📌 Intent

> **"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."**
> — *Design Patterns: Elements of Reusable Object-Oriented Software (GoF)*

🧾 Also Known As

- Policy

📈 Applicability (When to Use)

Use the **Strategy pattern** when:

- You need to define multiple variants of an algorithm or behavior.
- You want to avoid using many conditional statements (like `if`, `else`, or `switch`).
- Different behaviors are required based on context, but you want to **encapsulate** them separately.
- You want to enable **selecting an algorithm at runtime**.

👥 Participants (GoF Book)

1. **Strategy (Interface or Abstract Class)**

- Declares an interface common to all supported algorithms.

2. **ConcreteStrategy**

- Implements the algorithm using the Strategy interface.

3. **Context**

- Maintains a reference to a Strategy object.
- May allow clients to set or change the Strategy object at runtime.
- Delegates behavior to the Strategy object.

⚖️ Consequences (GoF Book)

- ✅ Promotes **flexibility and reusability** of algorithms.
- ✅ Eliminates conditional logic for selecting behavior.
- ✅ Makes unit testing easier by isolating each strategy.
- ❌ Increases the number of classes.
- ❌ Clients must understand differences between strategies to select appropriately.

🔗 Related Patterns

Related Pattern	Why Related?
State	Similar structure, but Strategy is chosen by the client, while State changes automatically.
Decorator	Strategy can be combined with Decorator for dynamic behavior selection and layering.
Factory Method	Can be used to instantiate specific strategies.

❓ Why Use This Pattern?

- To **decouple algorithms** from the classes that use them.
- To **change behavior dynamically** at runtime without modifying existing code.
- To **encapsulate behaviors** for better organization, testing, and maintenance.
- To allow **multiple interchangeable behaviors**, such as different sorting, payment, or formatting strategies.

🎯 Principle Used

▧ Open/Closed Principle (OCP)

> You can introduce new strategies (algorithms) without modifying the existing code.

▧ Single Responsibility Principle (SRP)

> Each strategy has one responsibility: implementing a single algorithm.

🔁 Observer Pattern

📌 Intent

> **"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."**
> — *Design Patterns: Elements of Reusable Object-Oriented Software (GoF)*

📃 Also Known As

- Publish-Subscribe
- Dependents
- Listener

📈 Applicability (When to Use)

Use the **Observer pattern** when:

- One object (subject) needs to notify other objects (observers) about changes without being tightly coupled.
- You want to implement **event-driven systems** (e.g., GUI frameworks, messaging, real-time updates).
- The number and types of dependents (observers) are unknown at compile time and may change at runtime.
- You need a clean way to **decouple** publishers from subscribers.

👥 Participants (GoF Book)

1. **Subject (Observable)**

- Maintains a list of observers.
- Provides methods to attach, detach, and notify observers.

2. **Observer**

- Defines an interface for objects that should be notified of subject changes.

3. **ConcreteSubject**

- Stores state of interest to ConcreteObservers.
- Notifies observers when its state changes.

4. **ConcreteObserver**

- Implements the Observer interface.
- Updates its state to match the subject's state when notified.

⚖️ Consequences (GoF Book)

- ✅ Promotes **loose coupling** between subject and observers.
- ✅ Supports **broadcast communication** to multiple observers.
- ✅ Easy to add or remove observers at runtime.
- ❌ Can lead to unexpected updates if not carefully managed.
- ❌ May cause performance issues if many observers are registered or frequent updates occur.

🔗 Related Patterns

Related Pattern	Why Related?
Mediator	Centralizes communication between objects instead of allowing direct notifications.
Singleton	The subject might be a Singleton so all observers reference the same instance.
Event Queue	Often used alongside observer to queue and handle notifications asynchronously.

❓ Why Use This Pattern?

- To implement **event systems** where many objects must respond to changes in another.
- To keep **components independent**, yet allow them to **communicate indirectly**.
- To improve **maintainability and scalability** in applications that involve state changes and notifications.
- To implement **reactive programming** styles in UI frameworks, monitoring tools, messaging systems, etc.

🎯 Principle Used

▧ Principle: Loosely Coupled Design

> Subjects and observers are loosely coupled — the subject only knows that the observer implements a specific interface.

▧ Open/Closed Principle (OCP)

> New observers can be added without changing the subject.

▧ Dependency Inversion Principle (DIP)

> The subject and observers depend on abstractions (interfaces), not concrete implementations.

🙏 Thank You!

Thank you for exploring the **Gang of Four (GoF) Java Design Patterns** with this project.
Whether you're a student, developer, or enthusiast — I hope this repository helped you **learn, revise, or implement** patterns with clarity and confidence.

If you found this helpful, feel free to ⭐ star the repo and share it with your fellow Java learners!

Happy Coding! 💻 ☕
— *Sonani Akshit (MCA Student)*

---

language Java

Stars 2