🎨 Design Patterns – Introduction

**📘 Origin:**

* Inspired by architect **Christopher Alexander**, who described **reusable architectural building patterns**.
* Brought into software by the **"Gang of Four" (GoF)**:
  + Erich Gamma
  + Richard Helm
  + Ralph Johnson
  + John Vlissides

**🧠 What is a Design Pattern?**

A **design pattern** is a general, reusable solution to a commonly occurring problem in software design.

**✅ Key Characteristics:**

* Not a complete design or implementation.
* Describes **problem**, **solution**, and **consequences**.
* Can be **reused** across projects and platforms.

**📘 Four Essential Elements of GoF Patterns**

1. **Name** – Short and descriptive (e.g., Singleton, Adapter)
2. **Problem** – When to apply it.
3. **Solution** – Structure of objects/classes involved.
4. **Consequences** – Trade-offs, benefits, and limitations.

**🎯 Purpose of Design Patterns**

* Capture **best practices**
* Solve recurring problems
* Promote **reusability** and **maintainability**
* Improve communication between developers using a **common language**

**🧩 Types of Design Patterns (GoF Classification)**

| **Type** | **Focus** |
| --- | --- |
| **Creational** | Object creation mechanisms |
| **Structural** | Class and object composition |
| **Behavioral** | Communication between objects |

**🏷️ Scope of Patterns**

| **Scope** | **Description** |
| --- | --- |
| **Class patterns** | Use inheritance (static relationships) |
| **Object patterns** | Use object composition (dynamic at runtime) |

**🏗️ Creational Patterns**

These patterns are about how **objects are created**.

**📌 Singleton Pattern**

**✅ Intent:**

Ensure a class has **only one instance** and provide a global point of access to it.

**💡 Motivation:**

* Printer spooler
* Database connection
* File system

**❌ Problem with Global Variables:**

* Cannot restrict creation of multiple instances.
* Clients must enforce the “one instance” rule — this violates encapsulation.

**✅ Better Solution:**

* Let the **class itself** control instance creation.

**✅ Implementation (Java):**

java

CopyEdit

public class Singleton {

private static Singleton uniqueInstance;

private Singleton() {} // Private Constructor

public static Singleton getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

return uniqueInstance;

}

}

**✅ Benefits:**

* Controlled access
* Lazy initialization
* Saves memory and resources

**🧱 Structural Patterns**

These patterns help compose **larger structures** from objects and classes.

**📌 Adapter Pattern**

**✅ Intent:**

Convert one interface into another expected by the client.

**💡 Motivation:**

Imagine a class that computes interest from a bank module — you want to use its features but don’t want to change your code or its interface.

**🧍 Real-World Example:**

* Power plug adapters: convert one type of plug to fit a different socket.

**📘 Structure:**

Adapter wraps an existing class and **translates** its interface to what the client expects.

**✅ Use Case:**

You want to **reuse an existing class**, but its interface doesn’t match what you need.

**🔄 Behavioral Patterns**

These patterns deal with **communication between objects** and how responsibilities are distributed.

**📌 Observer Pattern**

**✅ Intent:**

Define a one-to-many relationship so that when **one object changes**, all its **dependents are notified** automatically.

**💡 Motivation:**

In a weather station system:

* The **WeatherData** object holds temperature, humidity, pressure.
* Multiple **displays** (Current Conditions, Forecast, Statistics) must be updated when the data changes.

**🧍 Problem Without Observer:**

* WeatherData must know about all displays.
* Code becomes tightly coupled and hard to maintain.

**🧪 Observer Pattern Solution:**

|  |  |
| --- | --- |
| Role | Description |
| Subject | Holds core logic (WeatherData) |
| Observer | UI elements or output units that depend on subject's state |
| Subscription | Observers subscribe to updates from Subject |

**✅ Implementation Idea:**

* Observers **register** with Subject.
* Subject **notifies** all observers on state change.
* Observers can **unsubscribe** dynamically.

**📦 WeatherData Class – Before Applying Observer**

java

CopyEdit

void measurementsChanged() {

float temp = getTemperature();

float humidity = getHumidity();

float pressure = getPressure();

currentConditions.update(temp, humidity, pressure);

forecast.update(temp, humidity, pressure);

statistics.update(temp, humidity, pressure);

}

**❌ Problems:**

* Adding/removing displays requires changing code.
* Breaks Open-Closed Principle (OCP).
* No interface between displays and data provider.

**🧠 Observer Pattern – Applied**

**✅ Benefits:**

* Loose coupling
* Easily add/remove observers at runtime
* Reusable and scalable

**🧭 Summary Table**

| **Pattern Type** | **Example** | **Problem Solved** |
| --- | --- | --- |
| Creational | Singleton | Ensure one instance of a class |
| Structural | Adapter | Interface mismatch between two components |
| Behavioral | Observer | Notify multiple dependents of state change |

Would you like:

* UML diagrams of these patterns (Singleton, Adapter, Observer)?
* More GoF pattern explanations (e.g., Factory, Decorator, Iterator)?
* Practice scenarios for identifying which pattern to use?

Let me know how you’d like to continue!

🏗️ Creational Pattern – Singleton

|  |  |
| --- | --- |
| Aspect | Details |
| Intent | Ensure **only one instance** of a class exists globally |
| Motivation | Useful for systems like:• Printer Spooler• Database Connection• File System |
| Issue with Globals | Global variable approach doesn’t **enforce single instance** |
| Solution | Let class **control its own instantiation** |
| Java Code | java<br>private static Singleton uniqueInstance;<br>private Singleton() {}<br>public static Singleton getInstance() {<br>if (uniqueInstance == null)<br>uniqueInstance = new Singleton();<br>return uniqueInstance;<br>}<br> |
| Benefits | • Controlled access• Lazy initialization• Saves memory |

🧱 Structural Pattern – Adapter

|  |  |
| --- | --- |
| Aspect | Details |
| Intent | Convert **one interface to another** that clients expect |
| Motivation | Example: A bank’s interest calculator with a different API |
| Real-World Analogy | Power plug adapter |
| Structure | Adapter **wraps** incompatible class to adapt it |
| Use Case | Reuse existing class **without changing** its code |

🔄 Behavioral Pattern – Observer

|  |  |
| --- | --- |
| Aspect | Details |
| Intent | Establish **one-to-many** dependency. When one object changes, others are **notified automatically** |
| Example Use Case | Weather station with sensors and displays |
| Problem Without Observer | • Tightly coupled code• Difficult to add/remove observers• Violates Open-Closed Principle |
| Solution Roles | **Subject** – Holds core data (e.g., WeatherData)**Observer** – UI components (displays)**Subscription** – Observer registers with Subject |
| Benefits | • Loose coupling• Flexible runtime updates• Reusable, scalable design |