Introduction to Design Patterns

“Design patterns are the best practices you didn’t know you needed until you hit the same design problem for the third time.”
Design patterns are **reusable solutions** to commonly occurring problems in software design. They provide a **shared vocabulary**, promote **best practices**, and help you build **flexible, extensible, and maintainable** systems.

Think of them as a **template** or **blueprint** that you can adapt to your specific use case.

Real-World Analogy: Furniture Assembly

Imagine assembling furniture from IKEA:

- You don't invent a new method to connect every screw or panel.
- You follow a **repeatable set of instructions** (the pattern).
- This ensures the outcome is **predictable, reliable, and efficient**.

That’s exactly what design patterns do in software. They reduce chaos by giving you a well-tested plan.

Why Should You Use Design Patterns?

🔁 Reusability

“Don’t just solve the problem. Solve it *well*, and solve it *once*.”
Design patterns are like reusable mental models. Instead of solving a problem from scratch every time, you apply a pattern that has already been validated by decades of software engineering practice.

For example:

- Instead of figuring out how to restrict a class to a single instance, just use the **Singleton Pattern**.
- Need a pluggable behavior at runtime? Reach for the **Strategy Pattern**.

This kind of reuse isn’t about copying code, it’s about **reusing ideas** and **structures** that work.

🔧 Maintainability

“Clean design today prevents chaos tomorrow.”

When your system is built using well-structured patterns:

- Each component has a clearly defined role.
- Logic is modular, isolated, and easier to understand.
- Making changes or fixing bugs becomes straightforward.

Patterns like **Factory Method**, **Decorator**, and **Observer** make your system easier to **modify without rewriting** everything.

If your business logic changes, you only need to tweak the specific implementation, not the whole design.

📖 Readability

“Good code is read more often than it is written.”

Design patterns give your code a **shared vocabulary**. When you name a class `AbstractFactory` or `Strategy`, experienced developers immediately understand the role that class plays.

The result? **Instant clarity**. New team members, code reviewers, or interviewers can understand your design faster and with fewer explanations.

⚙️ Flexibility

“Design for change. Because change *will* come.”

Patterns are often built around **abstraction**, **decoupling**, and **composition over inheritance**. This makes your system more adaptable to future changes.

- Want to add a new type of payment method? The **Strategy Pattern** lets you do it without touching existing payment logic.
- Need to introduce caching or logging to an existing service? The **Decorator Pattern** lets you do it without modifying the core class.
- Want to handle user actions as undoable commands? Use the **Command Pattern** for full control.

In other words, patterns empower your code to **respond to new requirements with minimal disruption**.

Categories of Design Patterns

The classic “Gang of Four” (GoF) book categorized design patterns into **three groups**:

1. Creational Patterns – How objects are created

They abstract the instantiation process and help make your system independent of how its objects are created.

- **Singleton** – Ensure a class has only one instance.
- **Factory Method** – Delegate object creation to subclasses.
- **Abstract Factory** – Create families of related objects.
- **Builder** – Construct complex objects step-by-step.
- **Prototype** – Clone existing objects instead of creating new ones.

2. Structural Patterns – How objects are composed

They help organize different classes and objects to form larger structures.

- **Adapter** – Makes one interface compatible with another.
- **Decorator** – Adds new responsibilities to an object at runtime.
- **Facade** – Provides a simplified interface to a complex system.
- **Composite** – Treats individual and grouped objects uniformly.
- **Proxy** – Acts as a placeholder or access control for another object.

3. Behavioral Patterns – How objects interact

They define how communication happens between objects.

- **Strategy** – Enables selecting an algorithm at runtime.
- **Observer** – Notifies dependent objects of state changes.
- **Command** – Encapsulates a request as an object.
- **State** – Allows an object to change behavior based on internal state.
- **Template Method** – Defines the skeleton of an algorithm in a base class.