# Unit 4: Introduction to Patterns

| | |
|---|---|
| Unit/Module | Unit 4 |
| Person | Austin Makasare |
| Status | complete |

## Meaning of Pattern, Definition of Design Pattern

### Meaning of Pattern

A **pattern** refers to a *recurring solution to a common problem*.

If a problem occurs repeatedly, and the same solution has proven to work effectively each time, that solution is known as a **pattern**.

In software design:

- A **pattern** is a **template or model** that helps perform specific tasks efficiently.
- It captures **design knowledge** that can be reused.
- It describes **relationships and interactions** between classes or objects in object-oriented programming (OOP).
- Patterns are **not algorithms** or universal solutions — rather, they provide *guidance* on how to structure your solution.
- The main goal is to **speed up software development** by using well-tested, proven methods.

### Design Pattern

A **Design Pattern** is a *reusable and general solution* for common problems in **software design**, especially in **object-oriented programming**.

Key points from the chapter:

- Design Patterns became popular in the early 1990s.
- They define **objects, classes, interfaces, and their relationships** to solve design problems.
- A design pattern is **not a complete design**; it's a **template or guide** for solving problems that can occur in different situations.
- It provides **tested and proven development paradigms**, which help make the software:
  - **More flexible**
  - **Reusable**
  - **Maintainable**
- Patterns are **not mandatory** in every project; they are mainly for **solving recurring design problems**.

### Need of Design Pattern

- Design patterns provide efficient communication between designers,
- It solves an issue just by referring to a pattern name.
- It reduces the time to find the solution by reusing tested and proven development paradigms.
- Design patterns provide flexibility as it reduces the time for understanding the design.
- It helps to understand the basics of object-oriented design more easily and quickly.
- Design patterns provide a set of best practices and solutions that help developers build software that is robust, scalable, and maintainable.

- By using design patterns, developers can focus on delivering business value and solving real-world problems, rather than on the low-level details of software design.

---

## What makes a Pattern (GOF)

1. **Pattern Name**: It is a meaningful and descriptive name that summarises the pattern's problem, solution, and consequences..
2. **Problem**: It is a statement that describes the problem the pattern addresses, including the context and the forces that drive the solution.
3. **Solution**: It is a description of the elements that make up the pattern and the relationships between them, including the structure and behaviour of the pattern.
4. **Consequences**: It is a description of the trade-offs and results of using the pattern, including any drawbacks or limitations and the impact on the system as a whole.

By understanding the Problem, Solution, and Consequences, developers can evaluate whether a pattern is appropriate for a particular problem and make informed decisions about its implementation.

## GOF Design Pattern

**GOF** stands for **Gang of Four** — a group of four computer scientists who first formalized and published a set of foundational design patterns in their 1994 book:

> "Design Patterns: Elements of Reusable Object-Oriented Software"
>
> by **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**

These patterns became the **standard reference** for object-oriented software design.

GOF Design Patterns are **23 standard patterns** that offer **proven solutions** to common problems faced in software design.

They help developers:

- Write **scalable**, **maintainable**, and **reusable** code.
- Follow **best practices** for software design.
- Communicate design ideas effectively using a **shared vocabulary**.
- Provide **reusable and language-independent** strategies for solving common software design issues.
- Help developers build **flexible, reusable, and maintainable** systems.
- Reduce development time and effort by applying **tested and proven** design paradigms.

The 23 patterns are grouped into **three main categories**:

1. **Creational Patterns:** The design patterns that deal with the creation of an object. It describes how to create or instantiate objects.

   Examples:

   - **Abstract Factory** – Creates instances of several related classes.
   - **Builder** – Separates object construction from its representation.
   - **Factory Method** – Creates an instance of a subclass based on a condition.
   - **Prototype** – Creates objects by cloning an existing instance.
   - **Singleton** – Ensures only one instance of a class exists.

2. **Structural Patterns:** The design patterns in this category deal with the class structure such as Inheritance and Composition. It describes how/ objects are composed and combined to form larger structures.

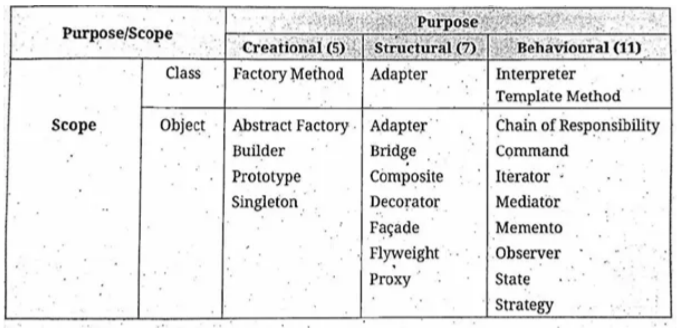
Examples:

- **Adapter** – Converts one interface into another expected by clients.
- **Bridge** – Separates abstraction from implementation.
- **Composite** – Represents part-whole hierarchies as trees.
- **Decorator** – Dynamically adds responsibilities to objects.
- **Facade** – Provides a unified interface to a complex subsystem.
- **Flyweight** – Shares objects efficiently to minimize memory use.
- **Proxy** – Acts as a placeholder or representative for another object.

3. **Behavioral Patterns:** This type of design patterns provide solutions for better interaction between objects, how to provide loose coupling, and flexibility to extend •easily in future. It describes how objects communicate with each other.

   Examples:

   - **Chain of Responsibility** – Passes requests along a chain of handlers.
   - **Command** – Encapsulates a request as an object.
   - **Interpreter** – Implements a simple language grammar.
   - **Iterator** – Accesses elements sequentially without exposing structure.
   - **Mediator** – Centralizes communication between objects.
   - **Memento** – Captures and restores an object's state.
   - **Observer** – Notifies dependent objects automatically of changes.
   - **State** – Changes object behavior when its internal state changes.
   - **Strategy** – Encapsulates algorithms and makes them interchangeable.
   - **Template Method** – Defines the structure of an algorithm, letting subclasses modify steps.
   - **Visitor** – Adds new operations without changing class definitions.



| Purpose/Scope | | Purpose | | |
|---|---|---|---|---|
| | | Creational (5) | Structural (7) | Behavioural (11) |
| Scope | Class | Factory Method | Adapter | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Façade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy |

## Describing Design Patterns.

Describing design patterns in a clear and concise manner helps developers understand the essence of the pattern and its impact on software design. It also makes it easier for developers to evaluate whether a pattern is appropriate for a particular problem and to make informed decisions about its implementation.

The template includes the following 12 components:

1. **Pattern Name and Classification:** The pattern's name and its category (Creational, Structural, or Behavioral).
2. **Intent:** A brief summary of the pattern's purpose and what it aims to achieve.

3. **Also Known As:** Any other common names for the pattern.
4. **Motivation (Problem, Context):** A description of the problem the pattern solves and the context in which that problem typically occurs.
5. **Applicability (Solution):** A description of the situations where the pattern can be applied as a solution.
6. **Structure:** A visual representation of the pattern, showing its elements and their relationships.
7. **Participants, Collaborations (Dynamics):** A description of the components (classes or objects) involved in the pattern and how they interact with each other.
8. **Implementation:** A description of how the pattern can be implemented in a specific programming language or platform.
9. **Sample Code:** A code example that shows how the pattern is implemented.
10. **Known Uses:** A list of real-world examples or scenarios where the pattern has been used successfully.
11. **Consequences:** A description of the trade-offs and results of using the pattern, including any limitations or drawbacks.
12. **Related Patterns:** A list of other patterns that are similar or related, offering additional context or alternative solutions.

---

## Pattern Categories & Relationships between Patterns

- A pattern language consists of a collection of related design patterns. Some patterns are useful for structuring the system, some are used for modifications of component of the system and some are helpful for implementing design aspects in specifics programming languages.

1. **Architectural Patterns**:
    - Just like the architecture of a building, software architecture describes the design and collection of components into systems that make up the building blocks of software.
    - It converts software characteristics to high-level structure.
    - Examples: Microservice, Event-driven architectural style.
2. **Design Patterns:**
    - A design pattern can be considered as a reusable solution for the commonly occurring problems in software design.
    - A Design pattern is not a finished design; rather, it is a template to solve the problem in many different solutions.
    - It is the specification that could help in the implementation of software.
    - Examples: Creational, Structural, Behavioral Pattern.
3. **Idioms:**
    - Idioms represent low-level patterns.
    - Idioms are used to describe how to solve implementation-specific problems in a programming language, such as Memory management in C++".
    - Idioms can directly address the implementation of a specific design pattern.

---

## Organizing the Catalogue

| Category | Design Patterns | Scope |
|---|---|---|
| **Creational Patterns (5)** | - Abstract Factory<br>- Builder<br>- Factory Method | Provides object creation mechanisms that increase flexibility and reuse of existing code. |

| | | |
|---|---|---|
| | - Prototype<br>- Singleton | |
| **Structural Patterns (7)** | - Adapter<br>- Bridge<br>- Composite<br>- Decorator<br>- Facade<br>- Flyweight<br>- Proxy | Describes ways to assemble objects and classes into larger structures while maintaining loose coupling. |
| **Behavioral Patterns (11)** | - Chain of Responsibility<br>- Command<br>- Interpreter<br>- Iterator<br>- Mediator<br>- Memento<br>- Observer<br>- State<br>- Strategy<br>- Template Method<br>-Visitor | Focuses on communication between objects and provides ways of assigning responsibilities between objects. |

## Patterns and Software Architecture

**Design Patterns** and **Software Architecture** are *closely related concepts* in software engineering.

While they serve different purposes, both aim to make software **well-structured, maintainable, scalable, and reusable**.

In simple terms:

- **Patterns** solve *specific recurring problems* in design.
- **Software architecture** defines the *overall structure and organization* of a system.

Together, they provide a **blueprint** for building high-quality software.

**What are Patterns in this Context?**

Patterns are:

- **Proven and reusable solutions** to common software design problems.
- Not tied to a specific programming language.
- Used to improve **maintainability**, **scalability**, and **code quality**.
- Represent **best practices** developed and refined over time.

Using patterns:

- Helps developers follow *tested design principles*.
- Encourages **consistent and efficient** software design across projects.
- Simplifies complex design problems through **reusable templates**.

---

**What is Software Architecture?**

**Software Architecture** is the **high-level design or blueprint** of a software system.

It defines:

- The **overall structure** (modules, components, layers).
- The **relationships** among components.
- The **interaction and data flow** between parts of the system.

In other words, it’s about **how the system is organized and how it behaves** as a whole.

Good architecture ensures that a system is:

- **Scalable** – can handle growth in users or features.
- **Maintainable** – easy to update and extend.
- **Efficient** – performs well under various conditions.
- **Reliable** – handles errors and faults gracefully.

---