

# software design pattern & software architecture

software design pattern is a general, reusable solution of how to solve a common problem when designing an application or system. Unlike a library or framework, which can be inserted and used right away, a design pattern is more of a template to approach the problem at hand.



# 1. Creational Design Patterns

A creational design pattern deals with object creation and initialization, providing guidance about which objects are created for a given situation. These design patterns are used to increase flexibility and to reuse existing code.

- **Factory Method:** Creates objects with a common interface and lets a class defer instantiation to subclasses.
- **Abstract Factory:** Creates a family of related objects.
- **Builder:** A step-by-step pattern for creating complex objects, separating construction and representation.
- **Prototype:** Supports the copying of existing objects without code becoming dependent on classes.
- **Singleton:** Restricts object creation for a class to only one instance.

# 2. Structural Design Patterns

A structural design pattern deals with class and object composition, or how to assemble objects and classes into larger structures.

- **Adapter:** How to change or adapt an interface to that of another existing class to allow incompatible interfaces to work together.
- **Bridge:** A method to decouple an interface from its implementation.
- **Composite:** Leverages a tree structure to support manipulation as one object.
- **Decorator:** Dynamically extends (adds or overrides) functionality.
- **Façade:** Defines a high-level interface to simplify the use of a large body of code.
- **Flyweight:** Minimize memory use by sharing data with similar objects.
- **Proxy:** How to represent an object with another object to enable access control, reduce cost and reduce complexity.

# 3. Behavioral Design Patterns

A behavioral design pattern is concerned with communication between objects and how responsibilities are assigned between objects.

- **Chain of Responsibility:** A method for commands to be delegated to a chain of processing objects.
- **Command:** Encapsulates a command request in an object.
- **Interpreter:** Supports the use of language elements within an application.
- **Iterator:** Supports iterative (sequential) access to collection elements.
- **Mediator:** Articulates simple communication between classes.
- **Memento:** A process to save and restore the internal/original state of an object.
- **Observer:** Defines how to notify objects of changes to other object(s).
- **State:** How to alter the behavior of an object when its stage changes.
- **Strategy:** Encapsulates an algorithm inside a class.
- **Visitor:** Defines a new operation on a class without making changes to the class.
- **Template Method:** Defines the skeleton of an operation while allowing subclasses to refine certain steps

## Why Do We Need Design Patterns?

1. Proven Solution
2. Reusable
3. Expressive
4. Prevent the Need for Refactoring Code
5. Lower the Size of the Codebase

# architectural pattern

architectural patterns that can be applied to the design of the software as a whole. What is an architectural pattern? A general, reusable solution to common problems in architecture

## 1. MVC Design Pattern

- **Model** – the backend business logic and data
- **View** – the interface components to display the data. Leverages the Observer pattern to update with Model and display the updated model when necessary.
- **Controller** – Input is directed here first, processing the request through the model and passing it back to view

## 2. MVP Design Pattern

- **Model** – the backend business logic and data
- **View** – input begins here and the requested action is presented here
- **Presenter** – One-to-one listening to the views and models, processing the request through the model and passing it back to view

## 3. MVVM Design Pattern

- **Model** – the backend business logic and data
- **View** – input begins here and the requested action is presented here
- **View-Model** – has no reference to view, its only purpose is to maintain the state of view and manipulate the model as the actions of view change