# Design Patterns

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

## What does the pattern consist of ?

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

## Classification of patterns

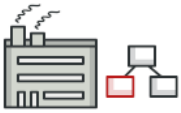
Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed. The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

# There are three main groups :

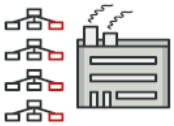
## (1) Creational Design Patterns :

Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



**Factory Method**

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

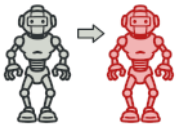


**Abstract Factory**

Lets you produce families of related objects without specifying their concrete classes.

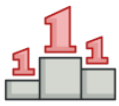
**Builder**

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



**Prototype**

Lets you copy existing objects without making your code dependent on their classes.

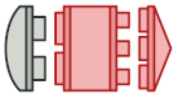


**Singleton**

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

# (2) Structural Design Patterns :

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.



### Adapter

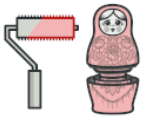
Allows objects with incompatible interfaces to collaborate.

### Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

### Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

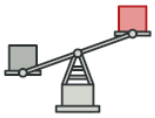


### Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

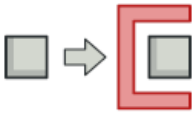
### Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



### Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

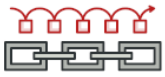


### Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

# (3) Behavioral Design Patterns :

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.



### Chain of Responsibility

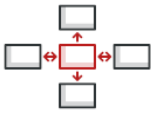
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

### Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

### Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



### Mediator

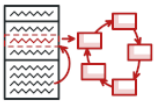
Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

### Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

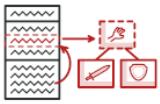
### Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



### State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



### Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

### Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

### Visitor

Lets you separate algorithms from the objects on which they operate.

# DESIGN PATTERNS

## in different programming languages