Software design patterns

Youssef Yasser Mohammed Hussein

What's a Design Pattern in Software engineering?

In software engineering, a 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.

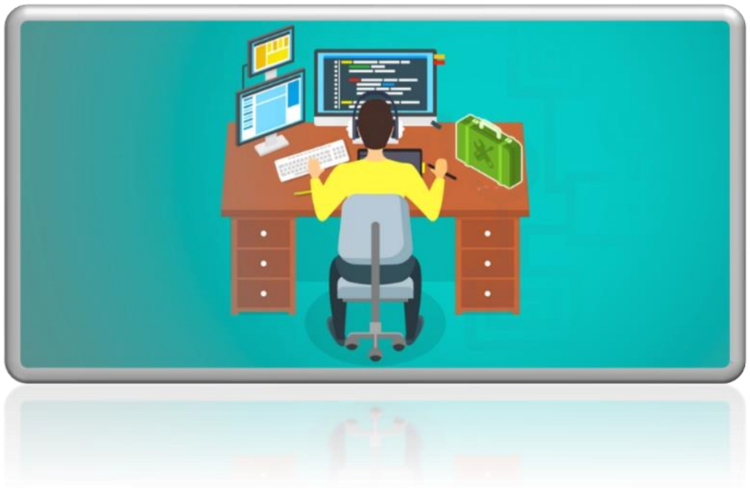
Design patterns are used to support object-oriented programming (OOP), a paradigm that is based on the concepts of both objects (instances of a class; data with unique attributes) and classes (user-defined types of data).

Why Do We Need Design Patterns?

Design patterns offer a best practice approach to support object-oriented software design, which is easier to design, implement, change, test and reuse. These design patterns provide best practices and structures.

In brief : design patterns are solution for some problems that we face in software design process.

Design patterns can be used with any programming language it is not related to any language and make your software process easier and save your time by avoiding the problems that faced the developers before.

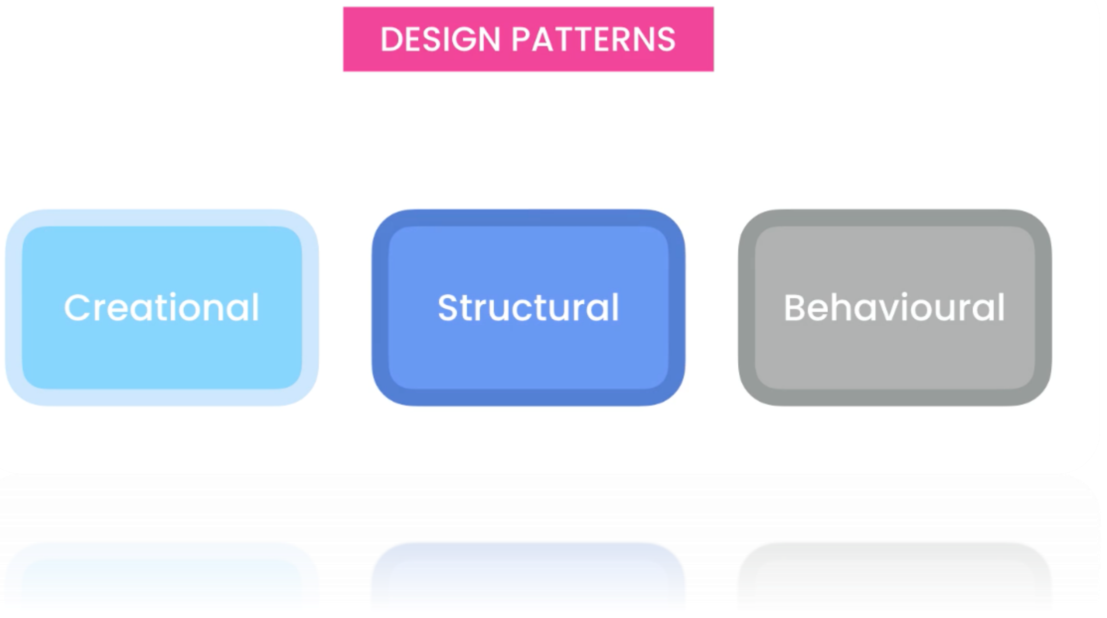


Types of design patterns....

Creational Design Patterns

They focus on how to create objects and the relation between each other.

- **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.



Structural Design Patterns

They focus on how to organize the classes and create a huge structure from the other classes.

- **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.

Behavioral design pattern

They focus on the interactions between objects and responsibility of each one.

- **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.