A design pattern provides a general reusable solution for the common problems that occur in software design. The pattern typically shows relationships and interactions between classes or objects. The idea is to speed up the development process by providing well-tested, proven development/design paradigms. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using design patterns, you can make your code more flexible, reusable, and maintainable.

It’s not mandatory to always implement design patterns in your project. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use, you just have to try to understand the design patterns and their purposes. Only by doing that, you will be able to pick the right one.

**Goal:**  
Understand the purpose and usage of each design pattern in order to pick and implement the correct pattern as needed.

**Example:**   
In many real-world situations, we want to create only one instance of a class. For example, there can be only one active president of a country at any given time. This pattern is called a Singleton pattern. Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object is costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Types of Design Patterns**  
There are mainly three types of design patterns:

1. **Creational**   
   These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are the*Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype.*

Use case of creational design pattern-   
1) Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what the developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. This results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance, we can control load balance, unnecessary connections, etc.

2) Suppose you want to create multiple instances of a similar kind and want to achieve loose [coupling](https://www.geeksforgeeks.org/coupling-in-java/) then you can go for Factory pattern. A class implementing factory design pattern works as a bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as back end, but in the future need to change the database to the oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation, we should go for the factory design pattern in order to achieve loose coupling and the creation of a similar kind of object.

1. **Structural**   
   These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.*

Use Case Of Structural Design Pattern-

1) When 2 interfaces are not compatible with each other and want to establish a relationship between them through an adapter it’s called an adapter design pattern. The adapter pattern converts the interface of a class into another interface or class that the client expects, i.e adapter lets classes work together that could not otherwise because of incompatibility. so in these types of incompatible scenarios, we can go for the adapter pattern.

1. **Behavioral**   
   Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns.

Behavioral patterns are *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor*

Use Case of Behavioral Design Pattern-

1) The template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. For example, in your project, you want the behavior of the module to be able to extend, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, no one is allowed to make source code changes to it, i.e you can add but can’t modify the structure in those scenarios a developer can approach template design pattern.

**Creational Patterns**

[**#**](https://dofactory.com/net/design-patterns#creational)

|  |  |
| --- | --- |
| [Abstract Factory](https://dofactory.com/net/abstract-factory-design-pattern) | Creates an instance of several families of classes  Frequency of use:    high |
| [Builder](https://dofactory.com/net/builder-design-pattern) | Separates object construction from its representation |
| [Factory Method](https://dofactory.com/net/factory-method-design-pattern) | Creates an instance of several derived classes |
| [Prototype](https://dofactory.com/net/prototype-design-pattern) | A fully initialized instance to be copied or cloned |
| [Singleton](https://dofactory.com/net/singleton-design-pattern) | A class of which only a single instance can exist |

**Structural Patterns**

[**#**](https://dofactory.com/net/design-patterns#structural)

|  |  |
| --- | --- |
| [Adapter](https://dofactory.com/net/adapter-design-pattern) | Match interfaces of different classes |
| [Bridge](https://dofactory.com/net/bridge-design-pattern) | Separates an object’s interface from its implementation |
| [Composite](https://dofactory.com/net/composite-design-pattern) | A tree structure of simple and composite objects |
| [Decorator](https://dofactory.com/net/decorator-design-pattern) | Add responsibilities to objects dynamically |
| [Facade](https://dofactory.com/net/facade-design-pattern) | A single class that represents an entire subsystem |
| [Flyweight](https://dofactory.com/net/flyweight-design-pattern) | A fine-grained instance used for efficient sharing |
| [Proxy](https://dofactory.com/net/proxy-design-pattern) | An object representing another object |

**Behavioral Patterns**

[**#**](https://dofactory.com/net/design-patterns#behavioral)

|  |  |
| --- | --- |
| [Chain of Resp.](https://dofactory.com/net/chain-of-responsibility-design-pattern) | A way of passing a request between a chain of objects |
| [Command](https://dofactory.com/net/command-design-pattern) | Encapsulate a command request as an object |
| [Interpreter](https://dofactory.com/net/interpreter-design-pattern) | A way to include language elements in a program |
| [Iterator](https://dofactory.com/net/iterator-design-pattern) | Sequentially access the elements of a collection |
| [Mediator](https://dofactory.com/net/mediator-design-pattern) | Defines simplified communication between classes |
| [Memento](https://dofactory.com/net/memento-design-pattern) | Capture and restore an object's internal state |
| [Observer](https://dofactory.com/net/observer-design-pattern) | A way of notifying change to a number of classes |
| [State](https://dofactory.com/net/state-design-pattern) | Alter an object's behavior when its state changes |
| [Strategy](https://dofactory.com/net/strategy-design-pattern) | Encapsulates an algorithm inside a class |
| [Template Method](https://dofactory.com/net/template-method-design-pattern) | Defer the exact steps of an algorithm to a subclass |
| [Visitor](https://dofactory.com/net/visitor-design-pattern) | Defines a new operation to a class without change |