Design Pattern in Software Development

Mostafa Ashrafı

What are Design Patterns?

- Reusable solutions for typical software design challenges are known as design patterns.
- Use to write more **structured**, **manageable**, and **scalable** code.
- Design patterns are not finished code but templates or blueprints only.

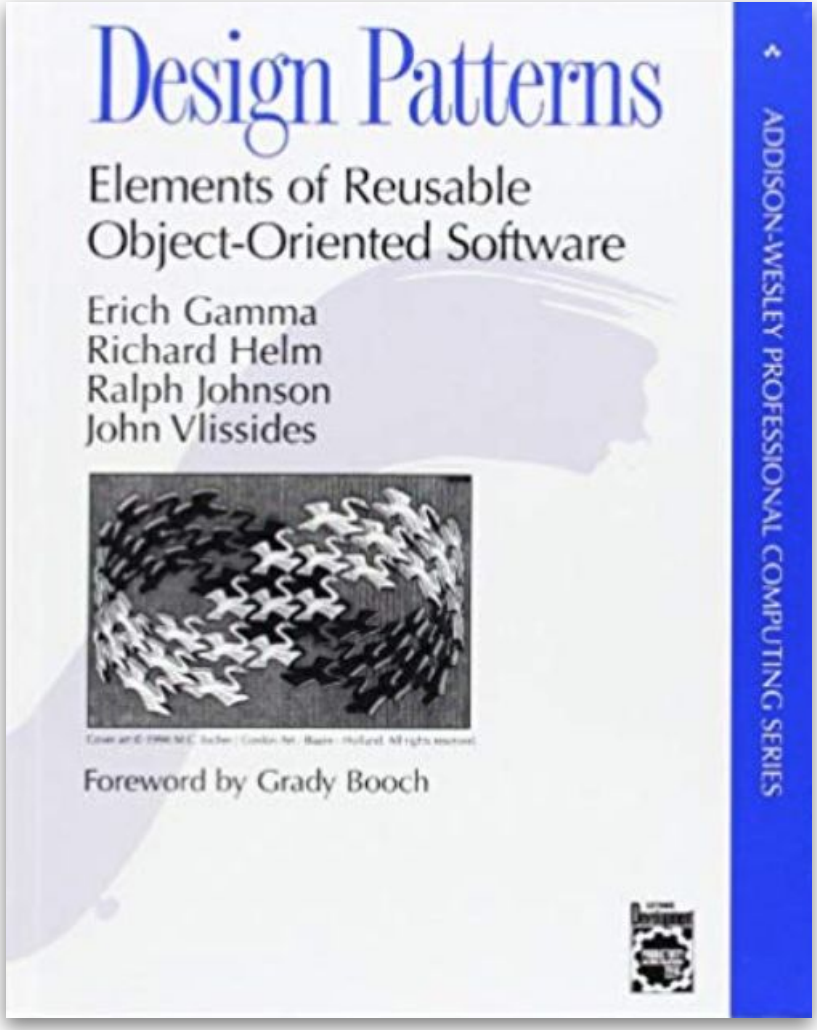
Key Characteristics of Design Patterns

- **Reusability**: Patterns can be applied to different projects and problems, saving time and effort in solving similar issues.
- **Standardization**: They provide a shared language and understanding among developers, helping in communication and collaboration.
- **Efficiency**: By using these popular patterns, developers can avoid finding the solution to same recurring problems, which leads to faster development.
- **Flexibility**: Patterns are abstract solutions/templates that can be adapted to fit various scenarios and requirements.

Gangs of Four

Gangs of Four Design Patterns is the collection of 23 design patterns from the book “Design Patterns: Elements of Reusable Object-Oriented Software”.

This book was first published in 1994.



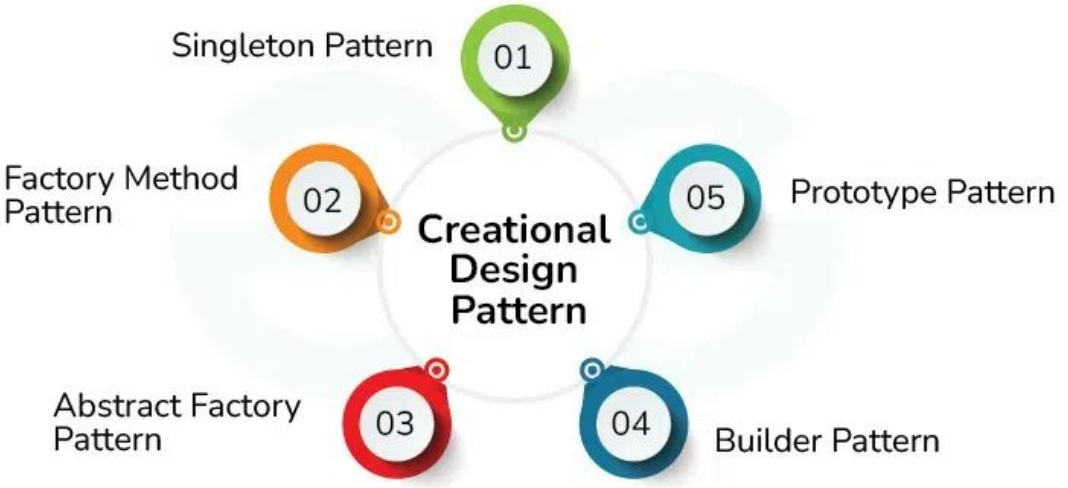
Types of Software Design Patterns

There are three types of Design Patterns:

- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

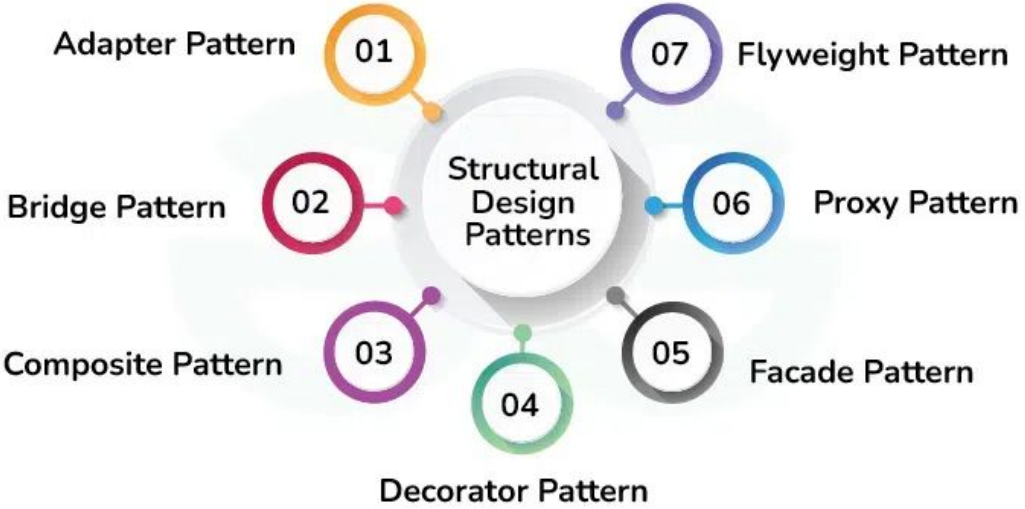
Creational Design Patterns

Focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed and represented.



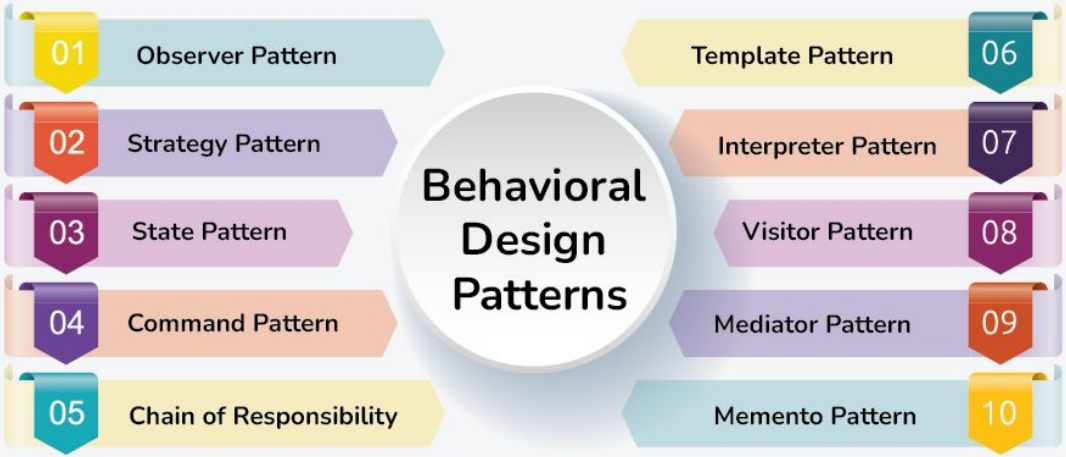
Structural Design Patterns

Solves problems related to how classes and objects are composed/assembled to form larger structures which are efficient and flexible in nature. Structural class patterns use inheritance to compose interfaces or implementations.



Behavioral Design Patterns

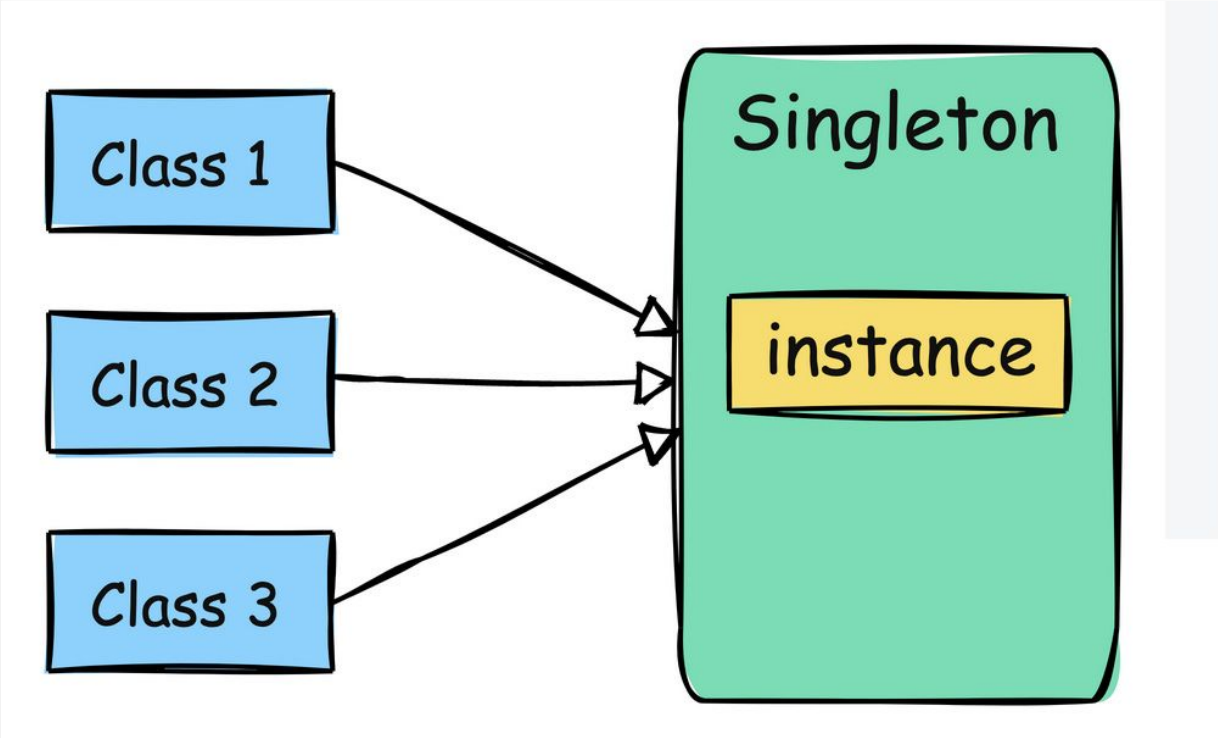
Concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.



Singleton Design Pattern

Singleton Design Pattern

Singleton design pattern ensures a class only has one instance, and provides a global point of access to it.



Singleton Design Pattern Principles

Single Instance: Singleton ensures that only one instance of the class exists throughout the application.

Global Access: Provide a global point of access to that instance.

Lazy or Eager Initialization: Support creating the instance either when needed (lazy) or when the class is loaded (eager).

Thread Safety: Implement mechanisms to prevent multiple threads from creating separate instances simultaneously.

Private Constructor: Restrict direct instantiation by making the constructor private, forcing the use of the access point

When to use Singleton Method Design Pattern?

- Consider using the Singleton pattern when you need to ensure that **only one instance** of a class exists in your application.

- If you think you might want to **extend** the class later, the Singleton pattern is a good choice. It allows for subclassing, so clients can work with the extended version without changing the original Singleton.

- This pattern is often used in situations like logging, managing connections to hardware or databases, caching data, or handling thread pools, where having just one instance makes sense

Advantages of the Singleton Design Pattern

- The Singleton pattern guarantees that there's only one instance with a unique identifier, which helps prevent naming issues.

- This pattern supports both eager initialization (creating the instance when the class is loaded) and lazy initialization (creating it when it's first needed), providing adaptability based on the use case.

- By keeping just one instance, the Singleton pattern can help **lower memory** usage in applications where resources are limited.

Disadvantages of the Singleton Design Pattern

- Singletons can make **unit testing difficult** since they introduce a global state. This can complicate testing components that depend on a Singleton, as its state can influence the test results.

- In **multi-threaded environments**, the process of creating and initializing a Singleton can lead to race conditions if multiple threads try to create it simultaneously.

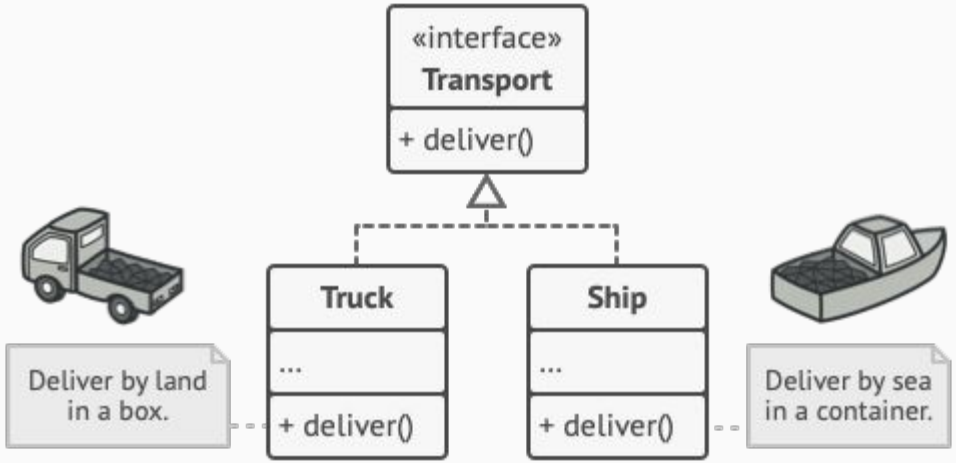
- The Singleton pattern creates a **global dependency**, which can complicate replacing the Singleton with a different implementation or using dependency injection.

Factory Design Pattern

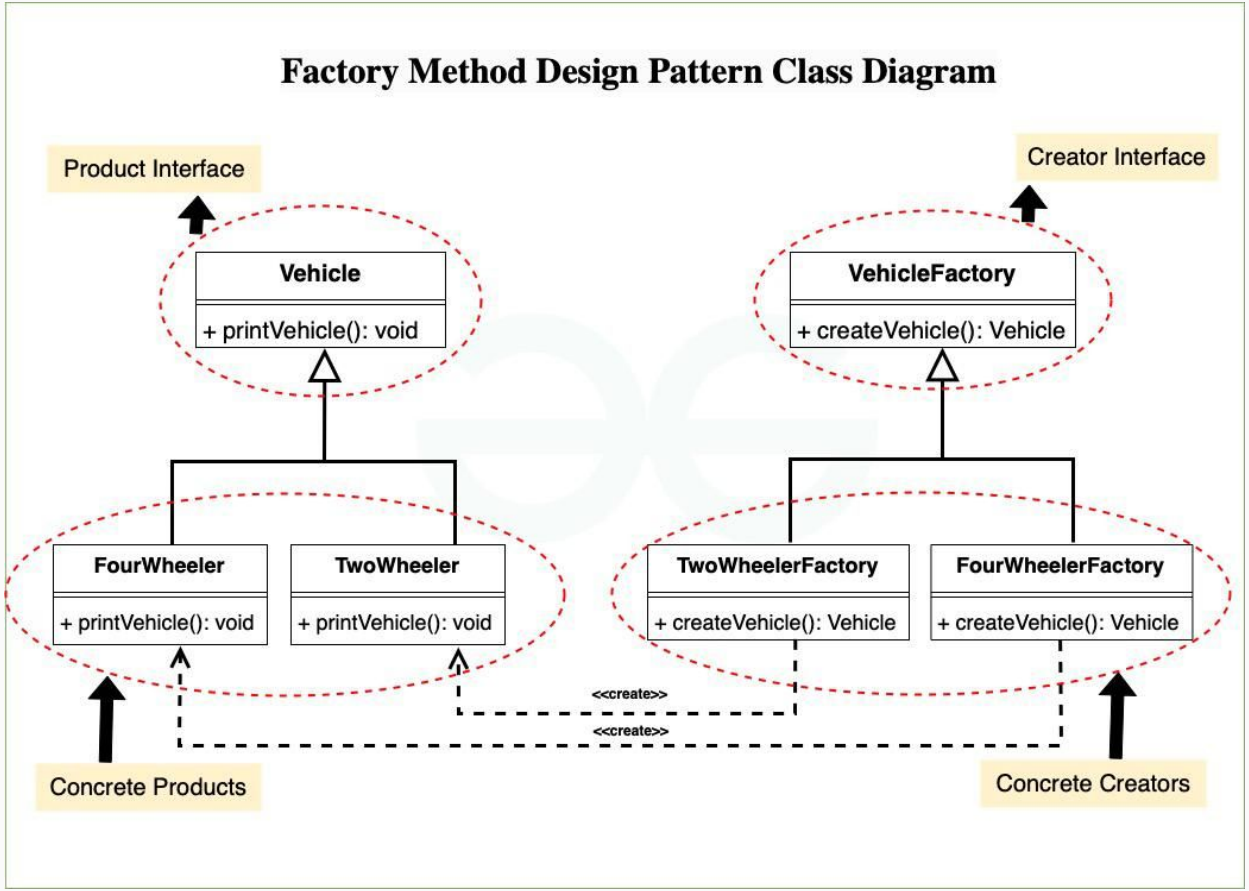
Factory Design Pattern

Provides an interface for creating objects in a superclass while allowing subclasses to specify the types of objects they create.

Also known as virtual constructor.



Components of Factory Method Design Pattern



Advantages of the Factory Method

- Separates creation logic from client code, improving flexibility.
- New product types can be added easily.
- Centralizes object creation logic across the application.
- Hides specific product classes from clients, reducing dependency.

Disadvantages of the Factory Method

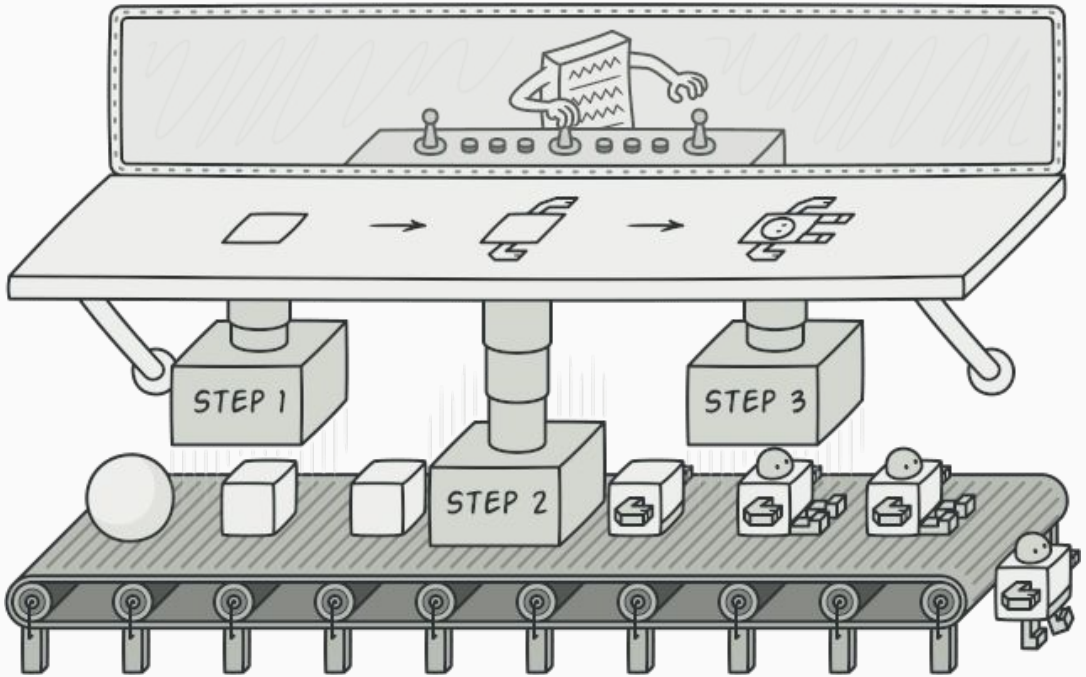
- Adds more classes and interfaces, which can complicate maintenance.
- Clients need knowledge of specific subclasses.
- May lead to unnecessary complexity if applied too broadly.
- Factory logic can be harder to test.

Builder Design Pattern

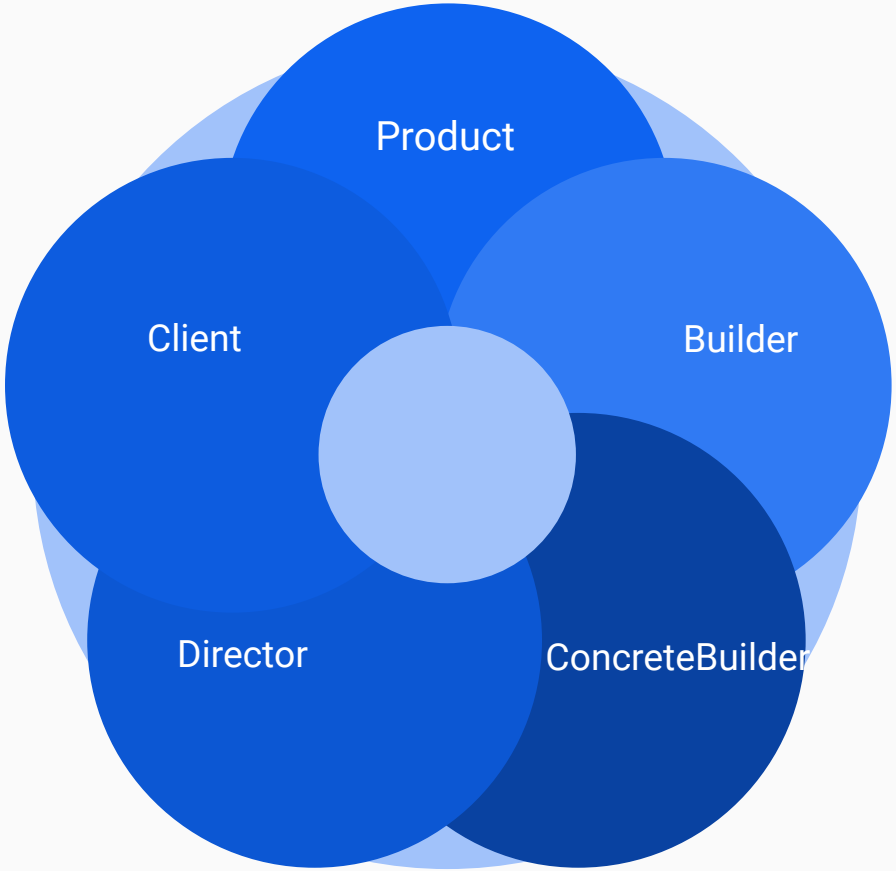
Builder Design Pattern

Used in software design to construct a complex object step by step.

It allows the construction of a product in a step-by-step manner, where the construction process can change based on the type of product being built.



Components of the Builder Design Pattern



Builder Design Pattern Example

Problem Statement:

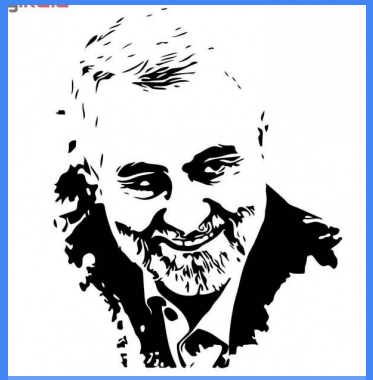
Implementing a system for building custom computers. Each computer can have different configurations based on user preferences. The goal is to provide flexibility in creating computers with varying **CPUs**, **RAM**, and **storage** options.

When to use Builder Design Pattern?

- **Complex Object Construction**: When you have an object with many optional components or configurations and you want to provide a clear separation between the construction process and the actual representation of the object.
- **Step-by-Step Construction**: When the construction of an object involves a step-by-step process where different configurations or options need to be set at different stages.
- **Avoiding constructors with multiple parameters**: When the number of parameters in a constructor becomes too large, and using telescoping constructors (constructors with multiple parameters) becomes unwieldy and error-prone.
- **Configurable Object Creation**: When you need to create objects with different configurations or variations, and you want a more flexible and readable way to specify these configurations.

When NOT to use Builder Design Pattern?

- **Simple Object Construction**: If the object you are constructing has only a few simple parameters or configurations, and the construction process is straightforward, using a builder might be unnecessary.
- **Performance Concerns**: In performance-critical applications, the additional overhead introduced by the Builder pattern might be a concern. The extra method calls and object creations involved in the builder process could impact performance, especially if the object construction is frequent.
- **Increased Code Complexity**: Introducing a builder class for every complex object can lead to an increase in code complexity.



Thank You