Builder Design Pattern

Last Updated : 03 Jan, 2025

The Builder Design Pattern is a creational 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. This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

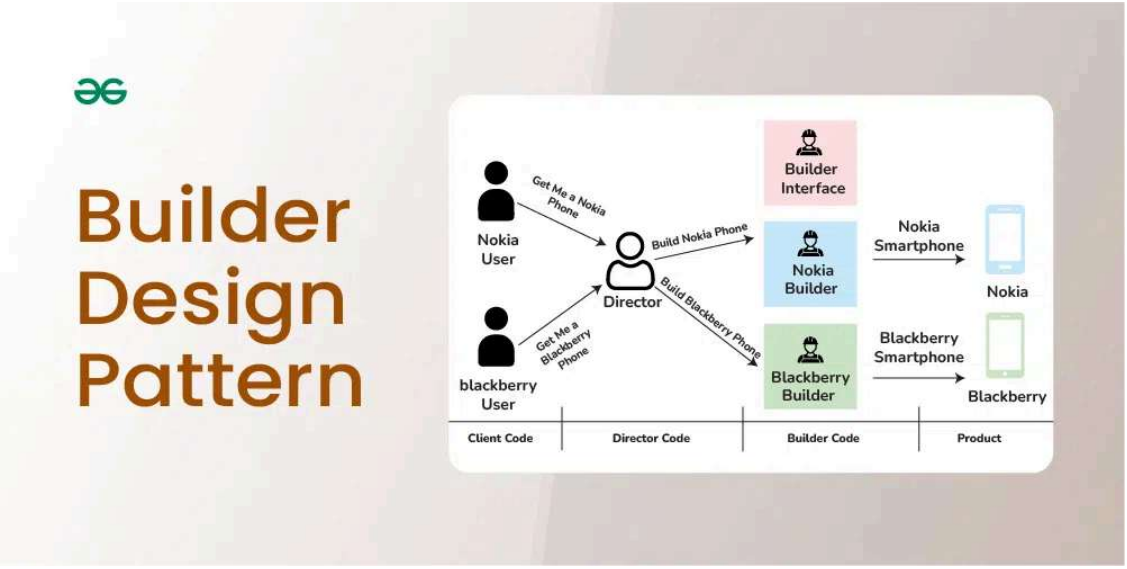


Table of Content

Components of the Builder Design Pattern

1. Product

The Product is the complex object that the Builder pattern is responsible for constructing.

- It may consist of multiple components or parts, and its structure can vary based on the implementation.
- The Product is typically a class with attributes representing the different parts that the Builder constructs.

2. Builder

The Builder is an interface or an abstract class that declares the construction steps for building a complex object.

- It typically includes methods for constructing individual parts of the product.
- By defining an interface, the Builder allows for the creation of different concrete builders that can produce variations of the product.

3. ConcreteBuilder

ConcreteBuilder classes implement the Builder interface, providing specific implementations for building each part of the product.

- Each ConcreteBuilder is custmized to create a specific variation of the product.
- It keeps track of the product being constructed and provides methods for setting or constructing each part.

4. Director

The Director is responsible for managing the construction process of the complex object.

- It collaborates with a Builder, but it doesn't know the specific details about how each part of the object is constructed.
- It provides a high-level interface for constructing the product and managing the steps needed to create the complex object.

5. Client

The Client is the code that initiates the construction of the complex object.

- It creates a Builder object and passes it to the Director to initiate the construction process.
- The Client may retrieve the final product from the Builder after construction is complete.

Steps to implement Builder Design Pattern

Below are the steps to implement Builder Design Pattern:

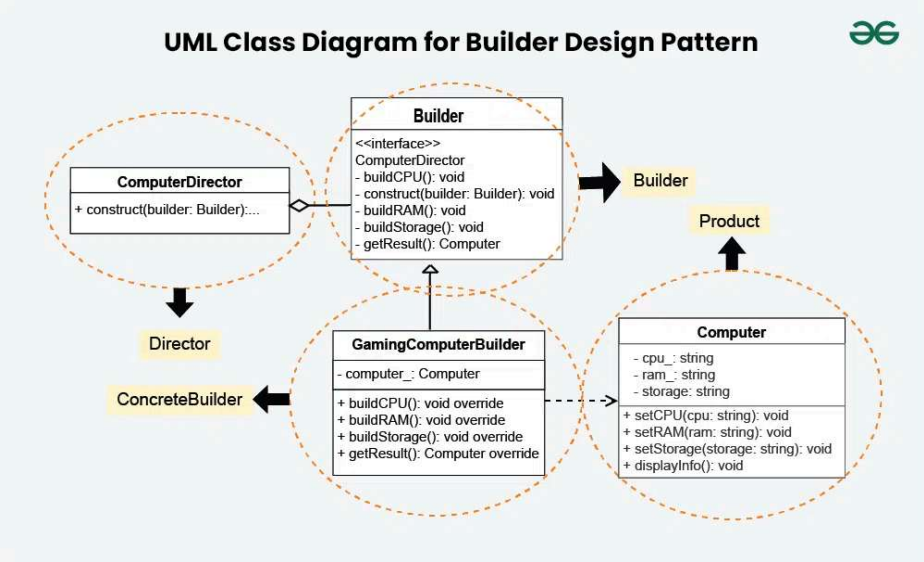
1. **Create the Product Class**: Define the object (product) that will be built. This class contains all the fields that make up the object.
2. **Create the Builder Class**: This class will have methods to set the different parts of the product. Each method returns the builder itself to allow method chaining.
3. **Add a Build Method**: In the builder class, add a method called `build()` (or similar) that assembles the product and returns the final object.
4. **Use the Director (Optional)**: If needed, you can create a director class to control the building process and decide the order in which parts are constructed.
5. **Client Uses the Builder**: The client will use the builder to set the desired parts step by step and call the `build()` method to get the final product.

Builder Design Pattern Example

Problem Statement:

> *You are tasked with 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.*

Implement the Builder design pattern to achieve this, allowing the construction of computers through a step-by-step process. Use the provided components – `Product` (Computer), `Builder` interface, `ConcreteBuilder` (GamingComputerBuilder), `Director`, and `Client`



1. `Product` (Computer)

```
// Product
class Computer {

private:
    string cpu_;
    string ram_;
    string storage_;


public:
    void setCPU(const std::string& cpu) {
        cpu_ = cpu;
    }

    void setRAM(const std::string& ram) {
        ram_ = ram;
    }

    void setStorage(const std::string& storage) {
        storage_ = storage;
    }

    void displayInfo() const {
        std::cout << "Computer Configuration:"
                  << "\nCPU: " << cpu_
                  << "\nRAM: " << ram_
```

```
                    << "\nStorage: " << storage_ << "\n\n";
    }
};
```

2. Builder

```
// Builder interface
class Builder {
public:
    virtual void buildCPU() = 0;
    virtual void buildRAM() = 0;
    virtual void buildStorage() = 0;
    virtual Computer getResult() = 0;
};
```

3. ConcreteBuilder

```
// ConcreteBuilder
class GamingComputerBuilder : public Builder {
private:
    Computer computer_;

public:
    void buildCPU() override {
        computer_.setCPU("Gaming CPU");
    }

    void buildRAM() override {
        computer_.setRAM("16GB DDR4");
    }

    void buildStorage() override {
        computer_.setStorage("1TB SSD");
    }

    Computer getResult() override {
```

```
        return computer_;
    }
};
```

4. Director

```
// Director
class ComputerDirector {
public:
    void construct(Builder& builder) {
        builder.buildCPU();
        builder.buildRAM();
        builder.buildStorage();
    }
};
```

5. Client

```
// Client
int main() {
    GamingComputerBuilder gamingBuilder;
    ComputerDirector director;

    director.construct(gamingBuilder);
    Computer gamingComputer = gamingBuilder.getResult();

    gamingComputer.displayInfo();

    return 0;
}
```

Complete Combined code for the above example

Below is the full combined code for the above example:

```cpp
#include <iostream>
#include <string>

using namespace std;

// Product
class Computer {
public:
    void setCPU(const std::string& cpu) {
        cpu_ = cpu;
    }

    void setRAM(const std::string& ram) {
        ram_ = ram;
    }

    void setStorage(const std::string& storage) {
        storage_ = storage;
    }

    void displayInfo() const {
        std::cout << "Computer Configuration:"
                  << "\nCPU: " << cpu_
                  << "\nRAM: " << ram_
                  << "\nStorage: " << storage_ << "\n\n";
    }

private:
    string cpu_;
    string ram_;
    string storage_;
};

// Builder interface
class Builder {
public:
    virtual void buildCPU() = 0;
    virtual void buildRAM() = 0;
    virtual void buildStorage() = 0;
    virtual Computer getResult() = 0;
};

```

```cpp
// ConcreteBuilder
class GamingComputerBuilder : public Builder {
private:
    Computer computer_;

public:
    void buildCPU() override {
        computer_.setCPU("Gaming CPU");
    }

    void buildRAM() override {
        computer_.setRAM("16GB DDR4");
    }

    void buildStorage() override {
        computer_.setStorage("1TB SSD");
    }

    Computer getResult() override {
        return computer_;
    }
};

// Director
class ComputerDirector {
public:
    void construct(Builder& builder) {
        builder.buildCPU();
        builder.buildRAM();
        builder.buildStorage();
    }
};

// Client
int main() {
    GamingComputerBuilder gamingBuilder;
    ComputerDirector director;

    director.construct(gamingBuilder);
    Computer gamingComputer = gamingBuilder.getResult();

    gamingComputer.displayInfo();

```

```
    return 0;
}
```

Output

```
Computer Configuration:
CPU: Gaming CPU
RAM: 16GB DDR4
Storage: 1TB SSD
```

This code demonstrates the Builder design pattern where the `Computer` class is the product, `Builder` is the interface, `GamingComputerBuilder` is the concrete builder, `ComputerDirector` is the director, and the `Client` assembles the product using the builder and director.

When to use Builder Design Pattern?

The Builder design pattern is used when you need to create complex objects with a large number of optional components or configuration parameters. This pattern is particularly useful when an object needs to be constructed step by step, some of the scenarios where the Builder design pattern is beneficial are:

- **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.

- **Common Interface for Multiple Representations:** When you want to provide a common interface for constructing different representations of an object.

When not to use Builder Design Pattern?

While the Builder design pattern is beneficial in many scenarios, there are situations where it might be unnecessary. Here are some cases when you should avoid the Builder 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.
- **Immutable Objects with Final Fields:**
    - While working with a language that supports immutable objects with final fields (e.g., Java's `final` keyword), and the object's structure is relatively simple, you might prefer using constructors with parameters or static factory methods.
- **Increased Code Complexity:**
    - Introducing a builder class for every complex object can lead to an increase in code complexity.
    - If the object being constructed is simple and doesn't benefit significantly from a step-by-step construction process, using a builder might add unnecessary complexity to the codebase.
- **Tight Coupling with Product:**
    - If the builder is tightly coupled with the product it constructs, and changes to the product require corresponding modifications to the

builder, it might reduce the flexibility and maintainability of the code.

Next Article

Structural Design Patterns

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

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. Creational...

4 min read

Types of Creational Patterns

Factory method Design Pattern

The Factory Method Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. This pattern is...

8 min read

Abstract Factory Pattern

The Abstract Factory Pattern is one of the creational design patterns that provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation, in...

8 min read

Singleton Method Design Pattern in JavaScript

Singleton Method or Singleton Design Pattern is a part of the Gang of Four design pattern and it is categorized under creational design patterns. It is one of the most simple design patterns in terms of...

10 min read

Singleton Method Design Pattern

The Singleton Method Design Pattern ensures a class has only one instance and provides a global access point to it. It's ideal for scenarios requiring centralized control, like managing database connections or...

11 min read

Prototype Design Pattern

The Prototype Design Pattern is a creational pattern that enables the creation of new objects by copying an existing object. Prototype allows us to hide the complexity of making new instances from the client. The...

8 min read

Builder Design Pattern

The Builder Design Pattern is a creational 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 ca...

7 min read

---

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

3. Types of Behvioural Patterns

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages

World GK