Builder Pattern

Simplified

Builder Pattern

Real-World Example

Imagine you're a **custom home builder** who creates **personalized houses**. Each client has specific desires for their ideal home, from the number of bedrooms to the presence of a garage, or the inclusion of luxurious amenities. Traditional construction processes often struggle to efficiently accommodate these varied demands. This is where the Builder design pattern steps in as a solution.

Definition

The Builder design pattern is a creational design pattern that is used to construct a complex object step by step. It separates the construction of an object from its representation, allowing you to create different variations of an object by using the same construction process.

To address this, you begin by creating a **House class that serves as the blueprint** for the final product. To streamline the construction process, you introduce the **HouseBuilder interface**, which defines methods for configuring different aspects of the house.
With the HouseBuilder interface in place, you proceed to implement concrete builder classes like the **Luxury House Builder** and the **Eco-Friendly House Builder**. Now, clients who aspire to build their custom dream homes can interact with the builder system to specify their unique needs and preferences. They have the option to choose the builder that aligns with their vision, set the desired features, and then request the final house product.

Fun fact

it's employed in the construction of complex objects like StringBuilder in Java and StringBuilder in C#, allowing for efficient string manipulation.

Representing the blueprint for the final product

```
public class House
{
    public int Bedrooms { get; set; }
    public bool HasGarage { get; set; }
    public List<string> Amenities { get; set; }

    public void Display()
    {
        //Print house details
    }
}
```

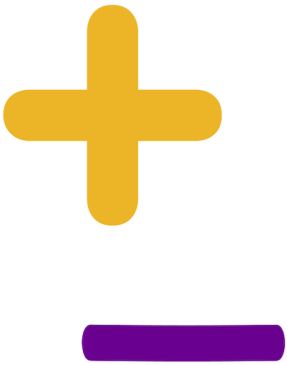
```
public class HouseDirector
{
    public void Construct(IHouseBuilder builder,
                          int bedrooms,
                          bool hasGarage,
                          List<string> amenities)
    {
        builder.BuildBedrooms(bedrooms);
        builder.BuildGarage(hasGarage);
        builder.BuildAmenities(amenities);
    }
}
```

```
public class LuxuryHouseBuilder : IHouseBuilder
{
    private House house = new House();

    public void BuildBedrooms(int bedrooms)
    {
        house.Bedrooms = bedrooms;
    }

    public void BuildGarage(bool hasGarage)
    {
        house.HasGarage = hasGarage;
    }

    public void BuildAmenities(List<string> amenities)
    {
        house.Amenities = amenities;
    }

    public House GetHouse()
    {
        return house;
    }
}
```

interface defining methods for configuring the house

```
HouseDirector director = new();
IHouseBuilder luxuryBuilder = new();
IHouseBuilder ecoFriendlyBuilder = new();

List<string> luxAmenities = new { "Swimming pool", "Home theater" };
List<string> ecoAmenities = new { "Energy-efficient appliances" };

director.Construct(luxuryBuilder, 5, true, luxuryAmenities);
House luxuryHouse = luxuryBuilder.GetHouse();

director.Construct(ecoFriendlyBuilder, 3, false, ecoFriendlyAmenities);
House ecoFriendlyHouse = ecoFriendlyBuilder.GetHouse();

luxuryHouse.Display();
ecoFriendlyHouse.Display();
```

Construct a luxury house



PROS

Simplifies Complex Objects:
Breaks down complex object creation into manageable steps.

Parameterized Construction:
Supports flexible object creation with optional parameters.

Fluent Interface:
Allows for a clear and expressive way to build objects.

Reusability:
Same builder can create different object configurations.

CONS

Not Always Needed:
Unnecessary for simpler object creation.

Increased Complexity:
Adds classes and complexity, especially with complex objects.

Risk of Incomplete Objects:
If used incorrectly, may create incomplete objects.

Use the Builder...

when you need your code to **construct various versions of a product**, such as stone and wooden houses, with different attributes.

when you want to **assemble intricate objects** like composite trees or other complex structures.

DO YOU LIKE THE POST?

REPOST IT!