Design Patterns

Creational Patterns – Builder Pattern

By: Alpesh Chaudhari

This document covers the Builder pattern, a key concept in Creational Design Patterns. This pattern which allows constructing complex objects step by step.

Builder Pattern

The **Builder Pattern** is particularly useful when dealing with the construction of complex objects that require multiple steps, especially when different combinations of fields or configurations are possible. It allows you to construct an object step-by-step, providing flexibility while avoiding the telescoping constructor anti-pattern (where multiple constructor overloads are used for different configurations).

When to Use the Builder Pattern

- **Complex Object Construction:** When an object involves multiple steps or stages to create, and the constructor would have too many parameters, leading to complexity or difficulty in understanding.
- **Immutability:** You want to create an immutable object, but need the flexibility to set various configurations during the build process.
- **Readable and Maintainable Code:** When you want to make the construction of objects more readable and maintainable. The builder pattern breaks the object creation into smaller, manageable parts.
- **Avoiding Constructor Overload Hell:** When a class has many optional parameters or variations of its construction. Instead of providing a long list of constructors, each with different parameters, the builder pattern offers a more structured and readable way to deal with these variations.
- **Step-by-Step Construction:** When you need to build an object in steps, and some of those steps might be optional or conditional based on specific configurations.

Key Benefits

- **Code clarity:** It separates object construction from its representation, improving code readability.
- **Flexibility:** Different configurations of an object can be built using the same construction process.
- **Immutability:** The builder allows the object to remain immutable while still supporting complex creation logic.

Problem

In a real-estate management application, you are tasked with building a system to handle **house construction**. Each house may vary in size, number of rooms, and additional features (like a garage, garden, swimming pool, and flooring material). Depending on customer preferences, a house can have different combinations of these features.

However, the system must enforce specific **business rules** to ensure valid configurations. For example:

- A house must have at least one bedroom and one bathroom.
- A swimming pool can only be added if the house has a garden.
- Each house must have a specified flooring material.

Moreover, the system must support multiple types of houses, such as:

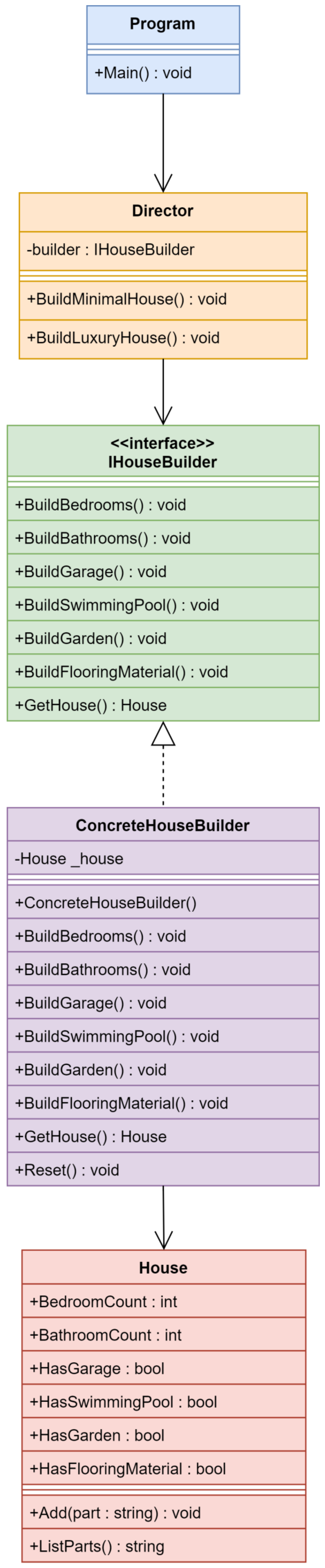
1. **Minimal Houses**: Houses with just basic features (e.g., bedrooms and bathrooms).

2. **Luxury Houses**: Houses with additional features (e.g., a garage, swimming pool, garden, and custom flooring).
3. **Custom Houses**: Houses that customers can configure on their own with selected features.

Using the **Builder Pattern** allows:

- **Step-by-step construction** of complex objects (houses) with optional parts.
- **Validation** of configurations to ensure valid houses are built.
- **Flexibility** to create predefined (e.g., minimal, luxury) or custom house configurations.
- **Separation of concerns**, making the construction process modular, reusable, and easier to maintain.

UML Class Diagram :



Here's how you can structure your code:

1. IHouseBuilder Interface

Purpose:

The IHouseBuilder interface defines the **blueprint** for building different parts of a house. Each method corresponds to a specific part (like bedrooms, bathrooms, a garage, etc.) that can be added to a house.

```
// The Builder interface specifies methods for creating the different parts of the House
objects.
public interface IHouseBuilder
{
    void BuildBedrooms();
    void BuildBathrooms();
    void BuildGarage();
    void BuildSwimmingPool();
    void BuildGarden();
    void BuildFlooringMaterial();
}
```

2. ConcreteHouseBuilder Class

Purpose:

- The ConcreteHouseBuilder implements the IHouseBuilder interface and provides **specific steps** to build a house.
- Each method in the builder creates or adds a part of the house (e.g., bedrooms, bathrooms, garage, etc.).
- The Reset method clears the current house and prepares the builder for a new house creation.

```
public class ConcreteHouseBuilder : IHouseBuilder
{
    private House _house = new House();

    public ConcreteHouseBuilder()
    {
        this.Reset();
    }
    public void Reset()
    {
        this._house = new House();
    }
    // Implementations of the building steps
    public void BuildBedrooms()
    {
        this._house.Add("4 Bedrooms");
        _house.BedroomCount = 4;  // Add specific tracking of the count of bedrooms
    }
    public void BuildBathrooms()
    {
        this._house.Add("3 Bathrooms");
        _house.BathroomCount = 3; // Add specific tracking of the count of bathrooms
    }
    public void BuildGarage()
    {
        this._house.Add("Garage");
        _house.HasGarage = true;
    }

    public void BuildSwimmingPool()
    {
        this._house.Add("Swimming Pool");
        _house.HasSwimmingPool = true;
    }

    public void BuildGarden()
    {
        this._house.Add("Garden");
        _house.HasGarden = true;
    }

    public void BuildFlooringMaterial()
    {
        this._house.Add("Wooden Flooring");
        _house.HasFlooringMaterial = true;
    }

    // Retrieve the constructed House object.
    public House GetHouse()
    {
        // Validation logic before returning the final product.
        if (_house.BedroomCount < 1)
        {
            throw new InvalidOperationException("A house must have at least one 
bedroom.");
        }

        if (_house.BathroomCount < 1)
        {
            throw new InvalidOperationException("A house must have at least one 
bathroom.");
        }

        if (_house.HasSwimmingPool && !_house.HasGarden)
        {
            throw new InvalidOperationException("A swimming pool can only be added if the 
house has a garden.");
        }

        if (!_house.HasFlooringMaterial)
        {
            throw new InvalidOperationException("A house must have flooring material 
set.");
        }

        // Return the valid house
        House result = this._house;
        this.Reset(); // Reset the builder to be reused for another house
        return result;
    }
}
```

4.House Class

Purpose: The House class is the product being constructed by the builder. It holds a list of all parts (e.g., bedrooms, bathrooms, etc.) added during the building process.

```
// The Product class, which in this case is a House, that can consist of multiple parts.
public class House
{
    private List<object> _parts = new List<object>();
    // Tracking state for validation
    public int BedroomCount { get; set; }
    public int BathroomCount { get; set; }
    public bool HasGarage { get; set; }
    public bool HasGarden { get; set; }
    public bool HasSwimmingPool { get; set; }
    public bool HasFlooringMaterial { get; set; }
    // Add parts to the house
    public void Add(string part)
    {
        this._parts.Add(part);
    }
    // List the house components
    public string ListParts()
    {
        string str = string.Empty;
        for (int i = 0; i < this._parts.Count; i++)
        {
            str += this._parts[i] + ", ";
        }
        str = str.TrimEnd(',', ' ');
        return "House parts: " + str + "\n";
    }
}
```

4.Director Class

Purpose: The Director class is responsible for defining the order in which the construction steps should occur. It dictates predefined configurations for the house, such as building a minimal house or a full-featured luxury house.

```
public class Director
{
    private IHouseBuilder _builder;

    public IHouseBuilder Builder
    {
        set { _builder = value; }
    }
    public void BuildMinimalHouse()
    {
        _builder.BuildBedrooms();
        _builder.BuildBathrooms();
    }
    public void BuildLuxuryHouse()
    {
        _builder.BuildBedrooms();
        _builder.BuildBathrooms();
        _builder.BuildGarage();
        _builder.BuildSwimmingPool();
        _builder.BuildGarden();
        _builder.BuildFlooringMaterial();
    }
}
```

5. Client Class

Purpose: The **Client** class acts as the client of the builder pattern. It demonstrates how to use the builder and director classes to create different types of houses.

```
public class Client
{
    public void Main()
    {
        var director = new Director();
        var houseBuilder = new ConcreteHouseBuilder();
        director.Builder = houseBuilder;

        // Build a minimal house (Valid Configuration)
        try
        {
            Console.WriteLine("Building a minimal house:");
            director.BuildMinimalHouse();  // Only builds bedrooms and bathrooms
            Console.WriteLine(houseBuilder.GetHouse().ListParts());
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        // Build a full-featured luxury house (Valid Configuration)
        try
        {
            Console.WriteLine("Building a luxury house:");
            director.BuildLuxuryHouse();
            Console.WriteLine(houseBuilder.GetHouse().ListParts());
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        // Try to build an invalid house (Swimming pool without garden)
        try
        {
            Console.WriteLine("Building an invalid house (Pool without garden):");
            houseBuilder.BuildBedrooms();
            houseBuilder.BuildBathrooms();
            houseBuilder.BuildSwimmingPool(); // Invalid because there's no garden
            Console.WriteLine(houseBuilder.GetHouse().ListParts());
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        // Try to build a house without setting flooring material (Invalid)
        try
        {
            Console.WriteLine("Building an invalid house (No flooring):");
            houseBuilder.BuildBedrooms();
            houseBuilder.BuildBathrooms();
            houseBuilder.BuildGarage();
            Console.WriteLine(houseBuilder.GetHouse().ListParts());
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

Output

```
Building a minimal house:
Error: A house must have flooring material set.
Building a luxury house:
House parts: 4 Bedrooms, 3 Bathrooms, 4 Bedrooms, 3 Bathrooms, Garage, Swimming Pool,
Garden, Wooden Flooring

Building an invalid house (Pool without garden):
Error: A swimming pool can only be added if the house has a garden.
Building an invalid house (No flooring):
Error: A swimming pool can only be added if the house has a garden.
```

When Not to Use the Builder Pattern

There are cases where the Builder Pattern might be overkill.

For example:

1. **Simple Objects:** If you have an object with just a few fields and none of the fields are optional, or there's no complex construction logic, then direct field assignment or a constructor may be sufficient.
    - Example: A Point object with X and Y coordinates.
2. **No Validation or Constraints:** If there's no need for validation or conditional logic during object creation, then the builder might not be necessary.

Note: For further examples of Builder Pattern use cases in real-life development, please refer to my GitHub repository titled "DesignPattern." In this repository, I have included an additional example of the Builder pattern.

GitHub: https://github.com/iam-alpesh/DesignPatterns.git

Thank You

I would like to express my gratitude to the following individuals and resources that helped me in understanding and implementing the Builder design pattern:

References

1. **Dive into Design patterns** by Alexander Shvets. Refactoring.Guru, 2021.
2. **Desing Patterns**. Refactoring Guru, https://refactoring.guru/