# Builder Pattern

## **Builder**

**Definition**:

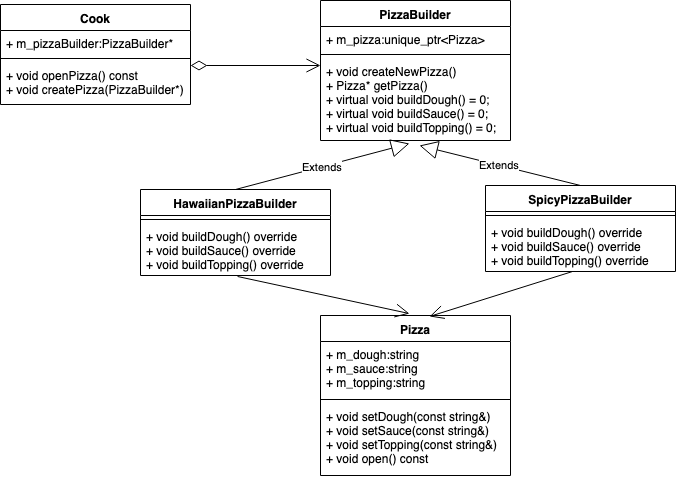
The Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.

This pattern allows a client object to construct a complex object by specifying **only its type and content**, being shielded from the details related to the object’s representation. This way the construction process can be used to create different representations.

**Problem:**We want to construct a complex object, however, we do not want to have a complex constructor member or one that would need many arguments. So we define an instance for creating an object but letting subclasses decide which class to instantiate and refer to the newly created object through a common interface.

**Solution:**Define an intermediate object whose member functions define the desired object part by part before the object is available to the client.

Builder Pattern lets us defer the construction of the object until all the options for creation have been specified.



* Pizza: Object we want to build
* PizzaBuilder: HawalianPizzaBuilder, SpicyPizzaBuilder, unique\_ptr<Pizza> m\_pizza;
* Cook (also called Director): PizzaBuilder\* m\_pizzaBuilder;
* Client: Use cook to create and open pizza, HawaiianPizzaBuilder hawaiianPizzaBuilder; cook.createPizza(&hawaiianPizzaBuilder);

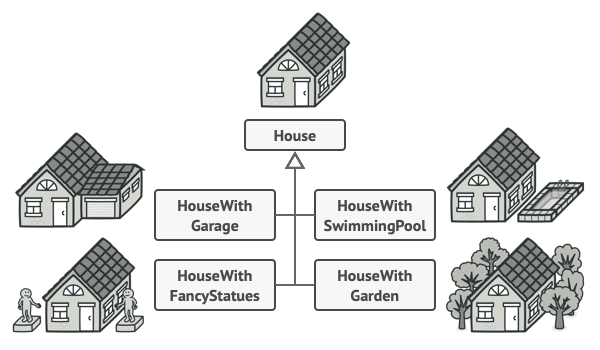
Details Explanation:

# Intent

**Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

# Problem

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.



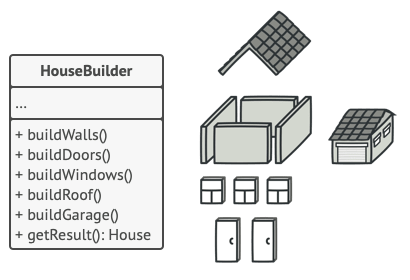
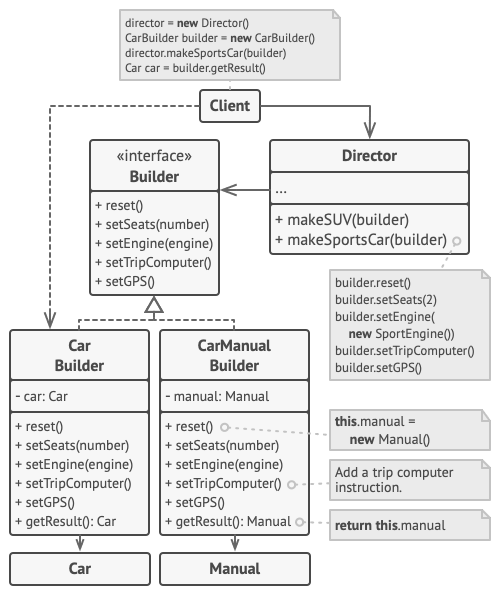
or example, let’s think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you’ll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

There’s another approach that doesn’t involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

# Solution

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.



# Applicability

**Use the Builder pattern to get rid of a “telescopic constructor”.**

 Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.

**class** **Pizza** {

Pizza(**int** size) { ... }

Pizza(**int** size, **boolean** cheese) { ... }

Pizza(**int** size, **boolean** cheese, **boolean** pepperoni) { ... }

// ...

Creating such a monster is only possible in languages that support method overloading, such as C# or Java.

The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don’t have to cram dozens of parameters into your constructors anymore.

**Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).**

 The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.

**Use the Builder to construct**[**Composite**](https://refactoring.guru/design-patterns/composite)**trees or other complex objects.**

 The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

A builder doesn’t expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

# Relations with Other Patterns

* Many designs start by using [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) (less complicated and more customizable via subclasses) and evolve toward [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory), [**Prototype**](https://refactoring.guru/design-patterns/prototype), or [**Builder**](https://refactoring.guru/design-patterns/builder) (more flexible, but more complicated).
* [**Builder**](https://refactoring.guru/design-patterns/builder) focuses on constructing complex objects step by step. [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.
* You can use [**Builder**](https://refactoring.guru/design-patterns/builder) when creating complex [**Composite**](https://refactoring.guru/design-patterns/composite) trees because you can program its construction steps to work recursively.
* You can combine [**Builder**](https://refactoring.guru/design-patterns/builder) with [**Bridge**](https://refactoring.guru/design-patterns/bridge): the director class plays the role of the abstraction, while different builders act as implementations.
* [**Abstract Factories**](https://refactoring.guru/design-patterns/abstract-factory), [**Builders**](https://refactoring.guru/design-patterns/builder) and [**Prototypes**](https://refactoring.guru/design-patterns/prototype) can all be implemented as [**Singletons**](https://refactoring.guru/design-patterns/singleton).