-

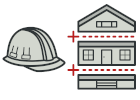
-

-

-

-

-



**Builder in C#**

**Builder** is a creational design pattern, which allows constructing complex objects step by step.

Unlike other creational patterns, Builder doesn’t require products to have a common interface. That makes it possible to produce different products using the same construction process.

[Learn more about Builder](https://refactoring.guru/design-patterns/builder)

**Complexity:** 2

**Popularity:**  3

**Usage examples:** The Builder pattern is a well-known pattern in C# world. It’s especially useful when you need to create an object **with lots of possible configuration options**.

**Identification:** The Builder pattern can be recognized in a class, which has a single creation method and several methods to configure the resulting object.

Builder methods often support chaining (for example, **someBuilder.setValueA(1).setValueB(2).create()).**

**Conceptual Example**

This example illustrates the structure of the **Builder** design pattern. It focuses on answering these questions:

* What classes does it consist of?
* What roles do these classes play?
* In what way the elements of the pattern are related?

**Program.cs: Conceptual example**

**using** System;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.Builder.Conceptual

{

// The Builder interface specifies methods for creating the different parts

// of the Product objects.

**public** **interface** **IBuilder**

{

**void** BuildPartA();

**void** BuildPartB();

**void** BuildPartC();

}

// The Concrete Builder classes follow the Builder interface and provide

// specific implementations of the building steps. Your program may have

// several variations of Builders, implemented differently.

**public** **class** **ConcreteBuilder** : IBuilder

{

**private** Product \_product = **new** Product();

// A fresh builder instance should contain a blank product object, which

// is used in further assembly.

**public** ConcreteBuilder()

{

**this**.Reset();

}

**public** **void** Reset()

{

**this**.\_product = **new** Product();

}

// All production steps work with the same product instance.

**public** **void** BuildPartA()

{

**this**.\_product.Add("PartA1");

}

**public** **void** BuildPartB()

{

**this**.\_product.Add("PartB1");

}

**public** **void** BuildPartC()

{

**this**.\_product.Add("PartC1");

}

// Concrete Builders are supposed to provide their own methods for

// retrieving results. That's because various types of builders may

// create entirely different products that don't follow the same

// interface. Therefore, such methods cannot be declared in the base

// Builder interface (at least in a statically typed programming

// language).

//

// Usually, after returning the end result to the client, a builder

// instance is expected to be ready to start producing another product.

// That's why it's a usual practice to call the reset method at the end

// of the `GetProduct` method body. However, this behavior is not

// mandatory, and you can make your builders wait for an explicit reset

// call from the client code before disposing of the previous result.

**public** Product GetProduct()

{

Product result = **this**.\_product;

**this**.Reset();

**return** result;

}

}

// It makes sense to use the Builder pattern only when your products are

// quite complex and require extensive configuration.

//

// Unlike in other creational patterns, different concrete builders can

// produce unrelated products. In other words, results of various builders

// may not always follow the same interface.

**public** **class** **Product**

{

**private** List<**object**> \_parts = **new** List<**object**>();

**public** **void** Add(**string** part)

{

**this**.\_parts.Add(part);

}

**public** **string** ListParts()

{

**string** str = **string**.Empty;

**for** (**int** i = 0; i < **this**.\_parts.Count; i++)

{

str += **this**.\_parts[i] + ", ";

}

str = str.Remove(str.Length - 2); // removing last ",c"

**return** "Product parts: " + str + "\n";

}

}

// The Director is only responsible for executing the building steps in a

// particular sequence. It is helpful when producing products according to a

// specific order or configuration. Strictly speaking, the Director class is

// optional, since the client can control builders directly.

**public** **class** **Director**

{

**private** IBuilder \_builder;

**public** IBuilder Builder

{

**set** { \_builder = **value**; }

}

// The Director can construct several product variations using the same

// building steps.

**public** **void** BuildMinimalViableProduct()

{

**this**.\_builder.BuildPartA();

}

**public** **void** BuildFullFeaturedProduct()

{

**this**.\_builder.BuildPartA();

**this**.\_builder.BuildPartB();

**this**.\_builder.BuildPartC();

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

// The client code creates a builder object, passes it to the

// director and then initiates the construction process. The end

// result is retrieved from the builder object.

**var** **director** = **new** Director();

**var** **builder** = **new** ConcreteBuilder();

director.Builder = builder;

Console.WriteLine("Standard basic product:");

director.BuildMinimalViableProduct();

Console.WriteLine(builder.GetProduct().ListParts());

Console.WriteLine("Standard full featured product:");

director.BuildFullFeaturedProduct();

Console.WriteLine(builder.GetProduct().ListParts());

// Remember, the Builder pattern can be used without a Director

// class.

Console.WriteLine("Custom product:");

builder.BuildPartA();

builder.BuildPartC();

Console.Write(builder.GetProduct().ListParts());

}

}

}

**Output.txt: Execution result**

Standard basic product:

Product parts: PartA1

Standard full featured product:

Product parts: PartA1, PartB1, PartC1

Custom product:

Product parts: PartA1, PartC1

-

-

-

-

# Builder pattern

The **builder pattern** is a [design pattern](https://www.wikiwand.com/en/Software_design_pattern) designed to provide a flexible solution to **various object creation problems** in [object-oriented programming](https://www.wikiwand.com/en/Object-oriented_programming).

The intent of the Builder design pattern is to [**separate**](https://www.wikiwand.com/en/Separation_of_concerns)**the construction of a complex object from its representation**. It is one of the [Gang of Four design patterns](https://www.wikiwand.com/en/Design_Patterns).

## Overview

The Builder design pattern is one of the [*Design Patterns*](https://www.wikiwand.com/en/Design_Patterns)[[1]](https://www.wikiwand.com/en/Builder_pattern#citenoteGoF1) that describe how to solve recurring design problems in object-oriented software.

The Builder design pattern solves problems like:[[2]](https://www.wikiwand.com/en/Builder_pattern#citenote2)

* How can a class (the same construction process) **create different representations of a complex object**?
* How can a class that includes creating a complex object **be simplified**?

**Creating and assembling the parts of a complex object directly within a class is inflexible**. It commits the class to creating a particular representation of the complex object and makes it impossible to change the representation later independently from (without having to change) the class.

The Builder design pattern describes how to solve such problems:

* Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
* A class delegates object creation to a Builder object instead of creating the objects directly.

A class (the same construction process) can delegate to different Builder objects to create different representations of a complex object.

## Definition

The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so, the same construction process can create different representations.[[1]](https://www.wikiwand.com/en/Builder_pattern#citenoteGoF1)

## Advantages

Advantages of the Builder pattern include:[[3]](https://www.wikiwand.com/en/Builder_pattern#citenote03)

* Allows you to vary a product's internal representation.
* Encapsulates code for construction and representation.
* Provides control over steps of construction process.

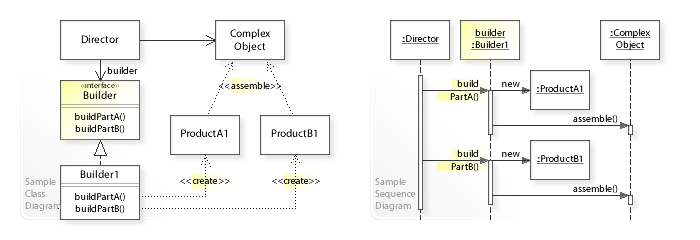
## Disadvantages

Disadvantages of the Builder pattern include:[[3]](https://www.wikiwand.com/en/Builder_pattern#citenote03)

* **A distinct ConcreteBuilder must be created for each type of product**.
* **Builder classes must be mutable.**
* **May hamper/complicate dependency injection.**

## Structure

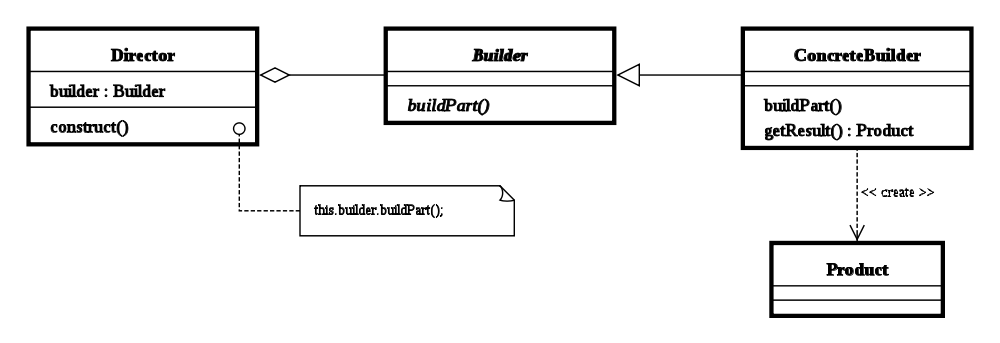
### **UML class and sequence diagram**



A sample UML class and sequence diagram for the Builder design pattern.[[4]](https://www.wikiwand.com/en/Builder_pattern#citenote4)

In the above [UML](https://www.wikiwand.com/en/Unified_Modeling_Language) [class diagram](https://www.wikiwand.com/en/Class_diagram), the Director class doesn't create and assemble the ProductA1 and ProductB1 objects directly. Instead, the Director refers to the Builder interface for building (creating and assembling) the parts of a complex object, which makes the Director independent of which concrete classes are instantiated (which representation is created). The Builder1 class implements the Builder interface by creating and assembling the ProductA1 and ProductB1 objects.  
The [UML](https://www.wikiwand.com/en/Unified_Modeling_Language) [sequence diagram](https://www.wikiwand.com/en/Sequence_diagram) shows the run-time interactions: The Director object calls buildPartA() on the Builder1 object, which creates and assembles the ProductA1 object. Thereafter, the Director calls buildPartB() on Builder1, which creates and assembles the ProductB1 object.

### **Class diagram**



**Builder**

**Abstract interface for creating objects (product).**

**ConcreteBuilder**

**Provides implementation for Builder**. It is **an**[**object able to construct other objects**](https://www.wikiwand.com/en/Factory_(software_concept)). **Constructs and assembles parts to build the objects.**

## Examples

A [C#](https://www.wikiwand.com/en/C%EF%BC%83_(programming_language)) example:

*/// <summary>*

*/// Represents a product created by the builder*

*/// </summary>*

**public** **class** Bicycle

{

**public** **string** Make { **get**; **set**; }

**public** **string** Model { **get**; **set**; }

**public** **int** Height { **get**; **set**; }

**public** **string** Colour { **get**; **set**; }

**public** **Bicycle**(**string** make, **string** model, **string** colour, **int** height)

{

Make = make;

Model = model;

Colour = colour;

Height = height;

}

}

*/// <summary>*

*/// The builder abstraction*

*/// </summary>*

**public** **interface** IBicycleBuilder

{

**string** Colour { **get**; **set**; }

**int** Height { **get**; **set**; }

Bicycle GetResult();

}

*/// <summary>*

*/// Concrete builder implementation*

*/// </summary>*

**public** **class** GTBuilder : IBicycleBuilder

{

**public** **string** Colour { **get**; **set**; }

**public** **int** Height { **get**; **set**; }

**public** Bicycle **GetResult**()

{

**return** Height == 29 ? **new** Bicycle("GT", "Avalanche", Colour, Height) : **null**;

}

}

*/// <summary>*

*/// The director*

*/// </summary>*

**public** **class** MountainBikeBuildDirector

{

**private** IBicycleBuilder \_builder;

**public** **MountainBikeBuildDirector**(IBicycleBuilder builder)

{

\_builder = builder;

}

**public** **void** **Construct**()

{

\_builder.Colour = "Red";

\_builder.Height = 29;

}

**public** Bicycle **GetResult**()

{

**return** **this**.\_builder.GetResult();

}

}

**public** **class** Client

{

**public** **void** **DoSomethingWithBicycles**()

{

**var** builder = **new** GTBuilder();

**var** director = **new** MountainBikeBuildDirector(builder);

*// Director controls the stepwise creation of product and returns the result.*

director.Construct();

Bicycle myMountainBike = director.GetResult();

}

}

The Director assembles a bicycle instance in the example above, delegating the construction to a separate builder object that has been given to the Director by the Client.

-

-

-

-