METHODPOET

C# Builder Pattern: The Complete Guide to Mastering It

Let's face it: software design is hard.

In other words, you can't just write code without first thinking about the problems you're trying to solve. For example, one of the most common problems is correctly initializing a complex class with many properties.

Enter Builder Pattern.

The Builder design pattern in C# is a creational design pattern that can create a complex class in a step-by-step manner. It is useful when the construction of a complex object requires many different steps or when the order of those steps is important.

This comprehensive guide will discuss the C# Builder pattern, its advantages and disadvantages, how to use it, how to implement it in real-world projects, and much more.

Table of Contents

What is the Builder design pattern in C#?

the complex object rather than using the object's constructor.

The pattern was initially described by the Gang of Four in their book, Design Patterns: Elements of Reusable Object-Oriented Software.

The client creates a Builder object and delegates the task of creating the object to the Builder object.

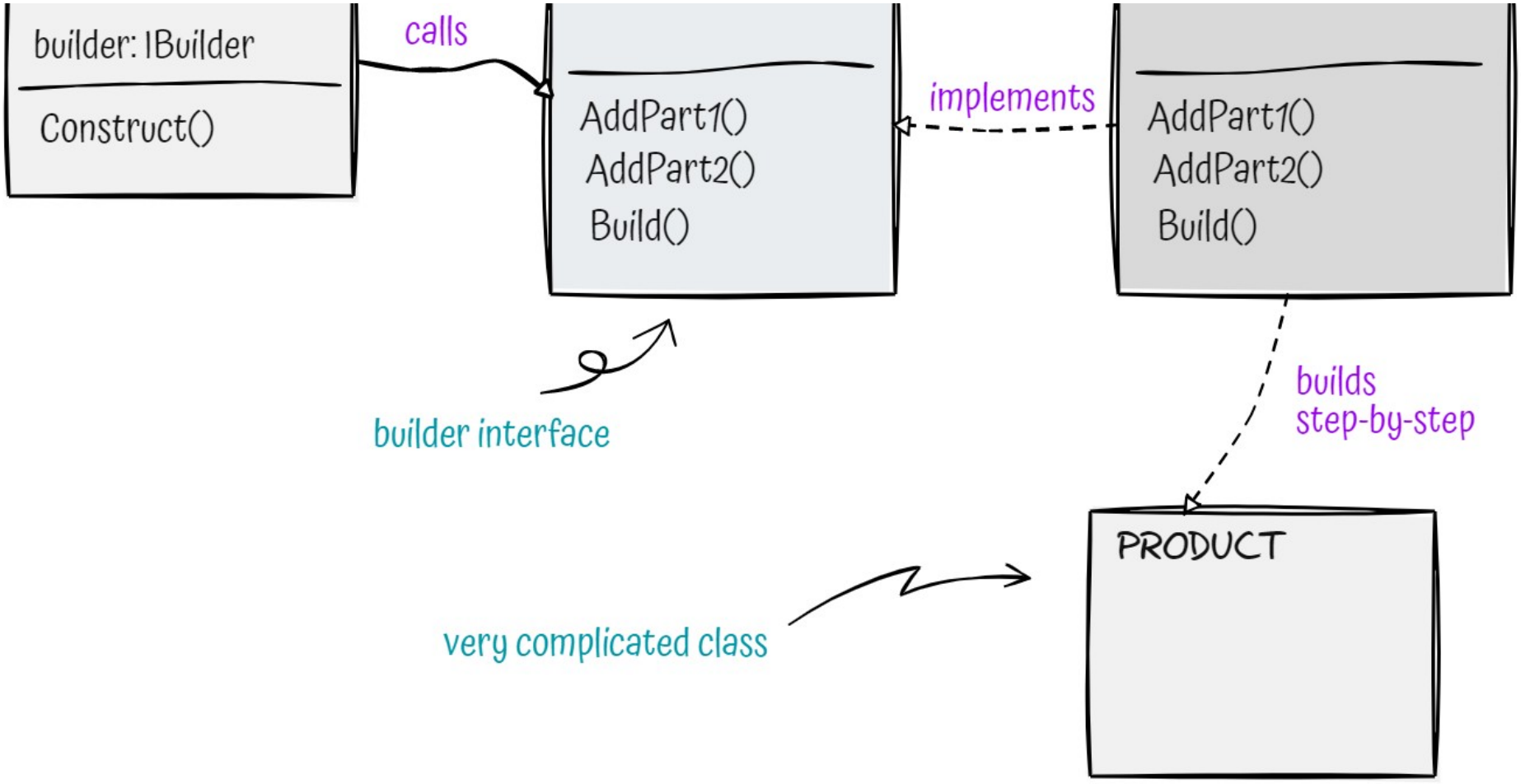
When the Builder finishes its work, it returns the completed complex object to the client.

The Builder design pattern is a handy tool that can help you when you have a collection of pieces and need to build them all together. With the Builder, you create each piece of the complex object step-by-step. After that, you can connect them all together and return the completed object. This approach allows you to add new properties to a component only when needed.

The Builder pattern also allows you to create multiple variations of the same object easily. It is also helpful in creating objects with a lot of optional parameters.

How the Builder design pattern works

The following diagram represents how the builder design pattern works.



It has the following parts:

- **Director** – Director is a class that decides in what steps should the complex class be built. In this example, the name of the complex class is Product. It doesn't build the Product class directly. Instead, it calls the IBuilder interface to create parts of the Product class. In your code, the Director class is not necessary. You can call the IBuilder methods directly.
- **IBuilder** – IBuilder defines the interface with all the necessary methods to build a Product class. These methods are common to all implementations of the Builder. Usually, it has the Build method. Director calls that method in the end to get

it.

- **Product** – The Product object is a complicated class that you need to build.

The sample code:

```
class Director
{
    private IBuilder _builder;

    public Director(IBuilder builder)
    {
        _builder = builder;

        _builder.BuildPart1();
        _builder.BuildPart2();
        Product finishedProduct = _builder.Build();
    }
}

interface IBuilder
{
    void BuildPart1();
    void BuildPart2();
    Product Build();
}

class ConcreteBuilder : IBuilder
{
```

```
    public ConcreteBuilder()
    {
        _product = new Product();
    }

    public void BuildPart1()
    {
        _product.Part1 = "part1";
    }

    public void BuildPart2()
    {
        _product.Part2 = "part2";
    }

    public Product Build()
    {
        return _product;
    }
}

public class Product
{
    public string Part1 { get; set; }
    public string Part2 { get; set; }
}
```

You can also make the methods on the IBuilder interface a bit complicated by passing parameters to the methods:

```
{
    void BuildPart1(string part1);
    void BuildPart2(string part2);
    Product Build();
}
```

The methods can also have more than one parameter.

How do you implement a C# Builder pattern? Real-world code example in C#

For example, let's say you want to create a class for configuring the settings of a blog post. The settings could have many options:

- blog title
- author
- categories
- tags
- published date
- content
- SEO metadata description
- SEO metadata title

```
class BlogPostSettings
{
    public string Title { get; set; }
    public string Content { get; set; }
    public string Author { get; set; }
    public DateTime Date { get; set; }
    public IList<string> Categories { get; set; }
    public IList<string> Tags { get; set; }
    public string MetaDescription { get; set; }
    public string MetadataTitle { get; set; }
}
```

Start by creating an IBlogPostBuilder interface that will help you construct the blog post.

```
interface IBlogPostBuilder
{
    void AddTitle(string title);
    void AddContent(string body);
    void AddAuthor(string author);
    void AddDate(DateTime date);
    void AddCategory(string category);
    void AddTags(IEnumerable<string> tags);
    void AddMetadataTitle(string title);
    void AddMetadataDescription(string description);
```

After that, you can implement the above interface:

```
class BlogPostBuilder: IBlogPostBuilder
{
    private BlogPostSettings _blogPostSettings = new BlogPostSettings();

    public BlogPostBuilder()
    {
        _blogPostSettings.Categories = new List<string>();
        _blogPostSettings.Tags = new List<string>();
    }

    public void AddTitle(string title)
    {
        _blogPostSettings.Title = title;
    }

    public void AddContent(string body)
    {
        _blogPostSettings.Content = body;
    }

    public void AddAuthor(string author)
    {
        _blogPostSettings.Author = author;
    }

    public void AddDate(DateTime date)
```

```
    }

    public void AddCategory(string category)
    {
        _blogPostSettings.Categories.Add(category);
    }

    public void AddTags(IEnumerable<string> tags)
    {
        _blogPostSettings.Tags = tags.ToList();
    }

    public void AddMetadataTitle(string title)
    {
        _blogPostSettings.MetadataTitle = title;
    }

    public void AddMetadataDescription(string description)
    {
        _blogPostSettings.MetaDescription = description;
    }

    public BlogPostSettings Build()
    {
        return _blogPostSettings;
    }
}
```

When you create an instance of this class, you can use it to configure the blog post settings.

```
{
    public BlogPostDirector()
    {
        var blogPostBuilder = new BlogPostBuilder();

        blogPostBuilder.AddTitle("My First Blog Post");
        blogPostBuilder.AddContent("This is my first blog post");
        blogPostBuilder.AddAuthor("John Doe");
        blogPostBuilder.AddDate(DateTime.Now);
        blogPostBuilder.AddCategory("C#");
        blogPostBuilder.AddCategory("Programming");

        var blogPostSettings = blogPostBuilder.Build();
    }
}
```

Finally, to get the completed instance of the BlogPostSettings, use the Build method.

When to use the Builder design pattern in C#?

You can use the C# Builder design pattern to create a class that has many options or properties to set up. In that case, the constructor is complicated or takes a lot of parameters.

The constructor for the BlogPostSettings would have the following signature:

```
    IList<string> tags, string metaDescription, string metadataTitle)
{
    Title = title;
    Content = content;
    Author = author;
    Date = date;
    Categories = categories;
    Tags = tags;
    MetaDescription = metaDescription;
    MetadataTitle = metadataTitle;
}
```

That looks very messy.

The Builder pattern separates the construction process into several steps, with each step corresponding to initializing a different property. Also, you don't have to follow a strict specific order when initializing an object with the Builder pattern. But in cases where you need to follow a specific order of creation, you can use progressive Builder. You will see later how to implement it.

For the above example, that means that you can first set the author, then the title, and vice versa. It also means you can call the AddCategory method several times. Each time you pass a new category, it gets appended to the overall list of blog post categories.

How to recognize the Builder Design Pattern

properties of the final object. However, you can recognize the Builder class in two ways:

- it usually has the Builder suffix in the name, or
- it has the Build (GetResult) method that returns the type it's creating.

What are the advantages of the builder design pattern?

One of the benefits of using the Builder design pattern is that it can help to improve the readability of your code.

Additionally, it can also help to make your code more maintainable. This is because the Builder design pattern can help to promote the separation of concerns.

Using the Builder design pattern, you can create a separate class for creating objects that you can reuse in many classes. It can keep your code organized and easy to follow. Additionally, using the Builder design pattern simplifies the complex object creation process.

What are the disadvantages of the builder design pattern?

Some disadvantages of the Builder design pattern are:

- **it increases the number of lines in the codebase**
- **it can be challenging to create a good builder implementation**
- **it can be challenging to change builders once they are created**

However, implementing the Builder design pattern can solve many problems, so the advantages outweigh the disadvantages.

You can use the Builder pattern in many different ways.

Simple Builder

The simple Builder represents the implementation you have seen so far. The Builder has the setter methods that construct the complex object. Typically, each method returns void. Only the Build method returns the type you are building.

Fluent Builder

The Fluent builder pattern is a creational pattern that makes object creation more straightforward and readable. The pattern is implemented by having a series of methods that return the builder object itself so that the method calls can be chained together. Except for the Build method, every method inside the Builder returns ***this***, that is, the return type is the builder type itself.

This makes the code more readable and less cluttered.

```
class FluentBlogPostBuilder
{
    private readonly BlogPostSettings _blogPostSettings = new BlogPostSettings();

    public FluentBlogPostBuilder()
    {
        _blogPostSettings.Categories = new List<string>();
        _blogPostSettings.Tags = new List<string>();
    }

    public FluentBlogPostBuilder WithTitle(string title)
```

```
        return this;
    }

    public FluentBlogPostBuilder WithContent(string body)
    {
        _blogPostSettings.Content = body;
        return this;
    }

    public FluentBlogPostBuilder WithAuthor(string author)
    {
        _blogPostSettings.Author = author;
        return this;
    }

    public FluentBlogPostBuilder WithDate(DateTime date)
    {
        _blogPostSettings.Date = date;
        return this;
    }

    public FluentBlogPostBuilder WithCategory(string category)
    {
        _blogPostSettings.Categories.Add(category);
        return this;
    }

    public FluentBlogPostBuilder WithTags(IEnumerable<string> tags)
    {
        _blogPostSettings.Tags = tags.ToList();
```

```
    public FluentBlogPostBuilder WithMetadataTitle(string title)
    {
        _blogPostSettings.MetadataTitle = title;
        return this;
    }

    public FluentBlogPostBuilder WithMetadataDescription(string description)
    {
        _blogPostSettings.MetaDescription = description;
        return this;
    }

    public BlogPostSettings Build()
    {
        return _blogPostSettings;
    }
}
```

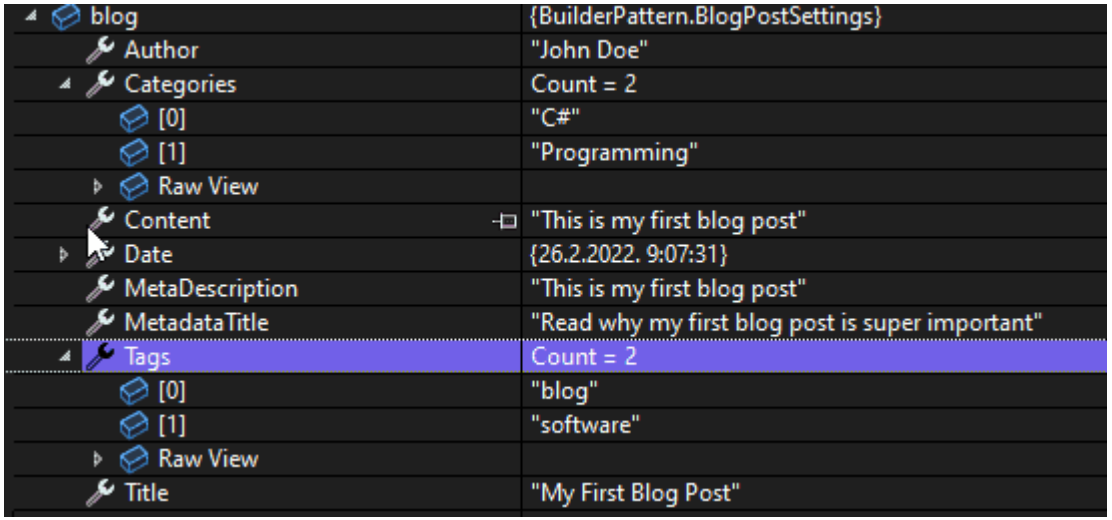
You can call the above builder to produce the blog post.

```
var blogPostBuilder = new FluentBlogPostBuilder();

var blogPost = blogPostBuilder
    .WithTitle("My First Blog Post")
    .WithContent("This is my first blog post")
    .WithAuthor("John Doe")
    .WithDate(DateTime.Now)
    .WithCategory("C#")
```

```
    .WithMetadataDescription("This is my first blog post")
    .WithMetadataTitle("Read why my first blog post is super important")
    .Build();
```

The above example produces the following object.



Child Builders

You can also use parent-children relation with the Builder design pattern. You define a parent class with shared behavior and then define a child class that builds a part of the object.

```
class Chapter
{
    public string Title { get; set; }
    public string Content { get; set; }
    public string OpeningQuote { get; set; }
```

```
class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public string Publisher { get; set; }
    public IList<Chapter> Chapters { get; set; }
}

interface IBookBuilder
{
    BookCoverBuilder AddBookCover();
    BookChapterBuilder AddChapter();
    Book Build();
}

class BookBuilder: IBookBuilder
{
    private readonly Book _book = new Book();

    public BookBuilder()
    {
        _book.Chapters = new List<Chapter>();
    }

    public BookCoverBuilder AddBookCover()
    {
        return new BookCoverBuilder(this, _book);
    }

    public BookChapterBuilder AddChapter()
```

```
    }

    public Book Build()
    {
        return _book;
    }
}

class BookCoverBuilder
{
    private readonly BookBuilder _parentBookBuilder;
    private readonly Book _book;

    public BookCoverBuilder(BookBuilder parentBookBuilder, Book book)
    {
        _parentBookBuilder = parentBookBuilder;
        _book = book;
    }

    public BookCoverBuilder WithTitle(string title)
    {
        _book.Title = title;
        return this;
    }

    public BookCoverBuilder WithAuthor(string author)
    {
        _book.Author = author;
        return this;
    }
```

```
    {
        _book.Publisher = publisher;
        return this;
    }

    public BookChapterBuilder AddChapter()
    {
        return _parentBookBuilder.AddChapter();
    }
}

class BookChapterBuilder
{
    private readonly BookBuilder _parentBookBuilder;
    private readonly Book _book;

    private readonly Chapter _chapter = new Chapter();

    public BookChapterBuilder(BookBuilder parentBookBuilder, Book book)
    {
        _parentBookBuilder = parentBookBuilder;
        _book = book;
    }

    public BookChapterBuilder WithTitle(string title)
    {
        _chapter.Title = title;
        return this;
    }
```

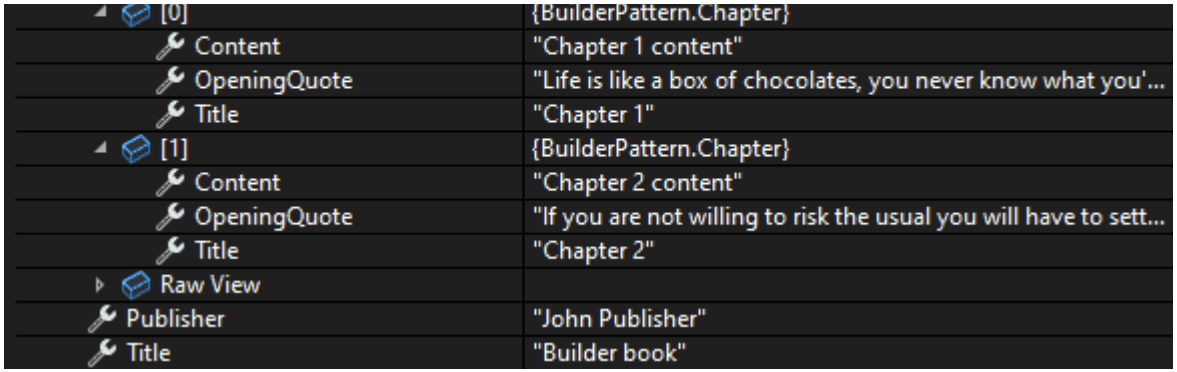
```
            _chapter.Content = content;
            return this;
        }

        public BookChapterBuilder WithOpeningQuote(string openingQuote)
        {
            _chapter.OpeningQuote = openingQuote;
            return this;
        }

        public BookChapterBuilder AddChapter()
        {
            _book.Chapters.Add(_chapter);
            return _parentBookBuilder.AddChapter();
        }

        public Book Build()
        {
            _book.Chapters.Add(_chapter);
            return _parentBookBuilder.Build();
        }
    }
```

The example from above produces the following complex object.



Progressive Builder (progressive interface)

A progressive interface is a variation of the Builder pattern that uses multiple builder patterns in a sequential way. What that means is that:

1. You call the first Builder and the methods on it.
2. The first Builder returns the second Builder.
3. The second Builder only offers you some methods you can use and returns the next Builder and so on.

This allows you to define a fixed sequence of method-chaining calls. The advantage of this implementation is that you essentially guide the developer on using the code. It also ensures that the object is built in the correct order.

For example, we want to build a Resume (CV) object. It needs to follow the correct order:

1. Add school education
2. Add college education
3. Add work experience

```
public class ResumeBuilder
{
    private readonly Resume _resume = new Resume();

    public ResumeBuilder()
    {
        _resume.Education = new List<Education>();
        _resume.WorkExperience = new List<Work>();
    }

    public ElementarySchoolBuilder AddSchoolEducation()
    {
        return new ElementarySchoolBuilder(this, _resume);
    }
}

public class ElementarySchoolBuilder
{
    private readonly Education _education = new Education();
    private readonly ResumeBuilder _resumeBuilder;
    private readonly Resume _resume;

    public ElementarySchoolBuilder(ResumeBuilder resumeBuilder, Resume resume)
    {
        _resumeBuilder = resumeBuilder;
        _resume = resume;
        _education.IsElementary = true;
    }
```

```
        _education.Name = name;
        return this;
    }

    public ElementarySchoolBuilder WithDescription(string description)
    {
        _education.Description = description;
        return this;
    }

    public CollegeEducationBuilder AddCollegeEducation()
    {
        _resume.Education.Add(_education);
        return new CollegeEducationBuilder(_resumeBuilder, _resume);
    }
}

public class CollegeEducationBuilder
{
    private readonly Education _education = new Education();
    private readonly ResumeBuilder _resumeBuilder;
    private readonly Resume _resume;

    public CollegeEducationBuilder(ResumeBuilder resumeBuilder, Resume resume)
    {
        _resumeBuilder = resumeBuilder;
        _resume = resume;
        _education.IsElementary = false;
    }
```

```
        _education.Name = name;
        return this;
    }

    public CollegeEducationBuilder WithDescription(string description)
    {
        _education.Description = description;
        return this;
    }

    public WorkExperienceBuilder AddWorkExperience()
    {
        _resume.Education.Add(_education);
        return new WorkExperienceBuilder(_resumeBuilder, _resume);
    }
}

public class WorkExperienceBuilder
{
    private readonly Work _work = new Work();
    private readonly ResumeBuilder _resumeBuilder;
    private readonly Resume _resume;

    public WorkExperienceBuilder(ResumeBuilder resumeBuilder, Resume resume)
    {
        _resumeBuilder = resumeBuilder;
        _resume = resume;
    }

    public WorkExperienceBuilder WithTitle(string title)
```

```
        return this;
    }

    public WorkExperienceBuilder WithDescription(string description)
    {
        _work.Description = description;
        return this;
    }

    public WorkExperienceBuilder WithCompany(string company)
    {
        _work.Company = company;
        return this;
    }

    public WorkExperienceBuilder WithStartDate(DateTime startDate)
    {
        _work.StartDate = startDate;
        return this;
    }

    public WorkExperienceBuilder WithEndDate(DateTime endDate)
    {
        _work.EndDate = endDate;
        return this;
    }

    public WorkExperienceBuilder IsCurrent(bool isCurrent)
    {
        _work.IsCurrent = isCurrent;
```

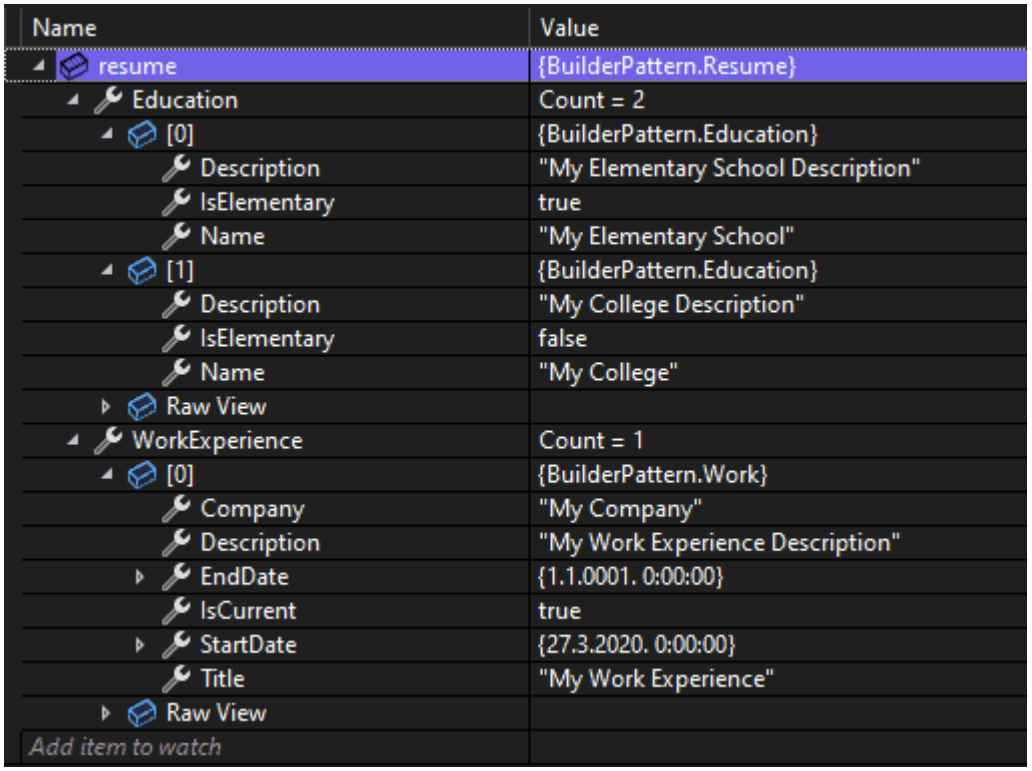
```
    public WorkExperienceBuilder AddWorkExperience()
    {
        _resume.WorkExperience.Add(_work);
        return new WorkExperienceBuilder(_resumeBuilder, _resume);
    }

    public Resume Build()
    {
        _resume.WorkExperience.Add(_work);
        return _resume;
    }
}
```

And the usage:

```
var initialBuilder = new ResumeBuilder();
Resume resume = initialBuilder
    .AddSchoolEducation()
        .WithName("My Elementary School")
        .WithDescription("My Elementary School Description")
    .AddCollegeEducation()
        .WithName("My College")
        .WithDescription("My College Description")
    .AddWorkExperience()
        .WithTitle("My Work Experience")
        .WithDescription("My Work Experience Description")
        .WithCompany("My Company")
        .WithStartDate(new DateTime(2020, 3, 27))
```

With the above code, you achieve the following result:



The advantage of this approach is that you can't build an illegal resume. For example, you can't add college education without elementary education. Nor can you add work experience without an elementary and college education.

You might argue that you can have work experience without an education. That is true, but in this case, the business rule is that you need to have all elements to build the correct resume object.

Which projects use the Builder pattern in .NET?

The C# projects that use the Builder pattern are:

- **Fluent Assertions**
- **Fluent Validation**
- **Fluent Migrator**
- **Bogus**
- **DbUp**

Let's see what they are.

Fluent Assertions

The Fluent Assertions library is a library for assertions in the C# unit testing. It makes it easier to write assertions and helps to avoid common mistakes. It also provides many features to make assertions more useful, including supporting asserting that collections are empty, that null values are not present, and more.

An example of the fluent builder syntax:

```
book.ISBN.Should().Be("9780321146533");
book.ISBNCheckDigit.Should().Be('3');
book.Author.Should().Be("Kent Beck");
book.Name.Should().Be("Test Driven Development: By Example");
```

validation rules in a DSL (Domain-Specific Language), making it easy to read and understand. The library is also extensible, so you can easily add custom validation rules.

It is designed to help you create self-describing validations for your models.

An example of the builder syntax:

```
public class CustomerValidator : AbstractValidator<Customer> {
  public CustomerValidator() {
    RuleFor(x => x.Surname).NotEmpty();
    RuleFor(x => x.Forename).NotEmpty().WithMessage("Please specify a first name");
    RuleFor(x => x.Address).Length(10, 200);
  }
}
```

Fluent Migrator

The Fluent Migrator library provides a framework for performing automated database migrations. It allows you to create a migration script that automatically updates your database schema when run.

This can be a huge time-saver, especially if you have a large database. The library is easy to use and well-documented, so you can quickly get up and running with it.

The code example of the builder syntax:

```
public class AddLogTable : Migration
{
    public override void Up()
    {
        Create.Table("Names")
            .WithColumn("Id").AsInt64().PrimaryKey().Identity()
            .WithColumn("Text").AsString();
    }

    public override void Down()
    {
        Delete.Table("Names");
    }
}
```

Bogus

The Bogus is a library that is used for fake data generation for unit tests. The library is used to generate fake data that you can use to test the functionality of applications and other systems. The library is open source and is available on GitHub.

Code example:

```
var testUsers = new Faker<User>()
    .RuleFor(u => u.Gender, f => f.PickRandom<Gender>())
    .RuleFor(u => u.FirstName, (f, u) => f.Name.FirstName(u.Gender))
    .RuleFor(u => u.LastName, (f, u) => f.Name.LastName(u.Gender));
```

up-to-date at all times.

Code example:

```
var upgrader =
    DeployChanges.To
        .SqlDatabase(connectionString)
        .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly())
        .LogToConsole()
        .Build();
```

What are related patterns to the builder design pattern?

There are a few related patterns to the builder design pattern.

One is the Composite design pattern, which allows you to treat a group of objects as a single object. It can be useful when you want to create a complex object by combining many simpler objects. You can use the Builder design pattern to build the Composite object.

Another related pattern is the Abstract Factory design pattern. It is similar to the Builder since you can create complex objects. However, the Builder and Abstract Factory patterns have different purposes. You can use the Builder pattern to create complex objects by separating the object's construction from its representation.

Conclusion

When you're looking to build a complex application and want to avoid the complexity of building everything from scratch, then the Builder design pattern can be an excellent option.

To summarize, the builder design pattern provides developers with an interface for creating complex objects that consist of pieces. By following best practices and keeping the API as simple as possible, developers can use the Builder design pattern to create easy-to-use and flexible APIs.

This article provided you with some concrete examples of when and how to use it, and you saw that it is used in various software development contexts.

If you'd know someone who would benefit from this article, please share it.

If you want to learn more about design patterns, check the separate in-depth article about design patterns in C#.

FAQ

What is fluent API in C#?

A fluent API is a C# programming technique used to create a more readable and easily understood codebase. A fluent API allows chaining of method calls, which can improve the readability of the code. It is designed to be intuitive and straightforward, allowing you to code more expressively.

chaining technique that allows the individual steps of object construction to be chained together. It is a concise and readable method of object construction.

What is the difference between Factory Method and Builder patterns?

The Factory Method pattern is a creational design pattern that uses factories to create objects without specifying the exact type of object to be created. The Builder Pattern is another creational design pattern that uses a Builder to create objects by assembling their parts.

Builder pattern allows for the construction of complex objects by using a step-by-step process. On the other hand, The Factory Method creates objects without using a step-by-step process.

For more comprehensive comparison between these two patterns, check out this article.

Is StringBuilder a Builder pattern?

Yes, StringBuilder is an implementation of the Builder pattern. It allows you to build up a string by adding pieces to it.

Why do we need a Builder design pattern?

The builder design pattern is used to create complex objects in a step-by-step fashion. This allows the objects to be created without having to know the exact steps needed to create them. It also allows the objects to be created in a way specific to the application's needs.

Kristijan Kralj

Recent Posts

Building Solid Foundations: Exploring SOLID Principles in C#

To build a clean architecture in C#, laying a solid foundation for your codebase is important. One of the fundamentals of clean architecture in C# is adherence to the SOLID principles, which serve as...

CONTINUE READING

4 Amazing Benefits of Integration Testing (+ 4 Drawbacks, Sadly)

So you're working on a new software project and about to reach the testing phase. That's great news! But have you considered the different testing types you need to perform? One of them...

CONTINUE READING

ABOUT ME

Hello! My name is Kristijan Kralj, and I am a C# software developer with 10 years of experience.

I have worked on various software projects ranging from simple programs to large enterprise systems. As a developer, I have acquired a wealth of experience and knowledge in C#, software architecture, unit testing, DevOps, and Azure. I enjoy working on complex systems that require creative solutions.

When I'm not glued to my computer screen, I like to spend time with my wife and two kids.