# الگوی Builder در #C

## تعریف الگوی Builder

الگوی **Builder** یک الگوی طراحی **خلاقی** (Creational) است که ساخت یک شیء پیچیده را از نمایش آن جدا می‌کند، به طوری که با یک فرآیند ساخت یکسان می‌توان نمایش‌های متفاوتی ایجاد کرد. این الگو به ویژه برای اشیایی که نیاز به پارامترهای زیاد در سازنده دارند بسیار مفید است.

## مشکل اصلی که Builder حل می‌کند

1. **سازنده‌های پیچیده (Telescoping Constructor)**: وقتی یک کلاس نیاز به پارامترهای زیادی در سازنده دارد
2. **اشیا با ساختار پیچیده**: وقتی یک شیء نیاز به مراحل ساخت گام به گام دارد
3. **انعطاف‌پذیری**: وقتی نیاز به ایجاد انواع مختلفی از یک شیء با ترکیب‌های مختلفی از پارامترها داریم

## ساختار پایه Builder

// 1. محصول (Product)

public class Pizza {

public string Dough { get; set; }

public string Sauce { get; set; }

public string Topping { get; set; }

public void Display() {

Console.WriteLine($"Dough: {Dough}, Sauce: {Sauce}, Topping: {Topping}");

}

}

// 2. Builder انتزاعی

public interface IPizzaBuilder {

void BuildDough(); void BuildSauce(); void BuildTopping();

Pizza GetPizza();

}

// 3. Concrete Builder

public class HawaiianPizzaBuilder : IPizzaBuilder {

private Pizza \_pizza = new Pizza();

public void BuildDough() => \_pizza.Dough = "Cross";

public void BuildSauce() => \_pizza.Sauce = "Mild";

public void BuildTopping() => \_pizza.Topping = "Ham+Pinapple";

public Pizza GetPizza() => \_pizza;

}

public class SpicyPizzaBuilder : IPizzaBuilder {

private Pizza \_pizza = new Pizza();

public void BuildDough() => \_pizza.Dough = "Pan Baked";

public void BuildSauce() => \_pizza.Sauce = "Hot";

public void BuildTopping() => \_pizza.Topping = "Pepperoni+Salami";

public Pizza GetPizza() => \_pizza;

}

// 4. Director (سرآشپز)

public class Cook {

private IPizzaBuilder \_pizzaBuilder;

public void SetPizzaBuilder(IPizzaBuilder pb) => \_pizzaBuilder = pb;

public Pizza GetPizza() => \_pizzaBuilder.GetPizza();

public void ConstructPizza() {

\_pizzaBuilder.BuildDough();

\_pizzaBuilder.BuildSauce();

\_pizzaBuilder.BuildTopping();

}

}

## نحوه استفاده از الگو

class Program

{

static void Main(string[] args) {

// ایجاد Director (سرآشپز)

Cook cook = new Cook();

// ایجاد Builderهای مختلف

IPizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();

IPizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

// ساخت پیتزای هاوایی

cook.SetPizzaBuilder(hawaiianPizzaBuilder);

cook.ConstructPizza();

Pizza hawaiian = cook.GetPizza();

hawaiian.Display();

// ساخت پیتزای تند

cook.SetPizzaBuilder(spicyPizzaBuilder);

cook.ConstructPizza();

Pizza spicy = cook.GetPizza();

spicy.Display();

}

}

## مزایای Builder

1. **کنترل فرآیند ساخت**: امکان ساخت گام به گام شیء را فراهم می‌کند
2. **جداسازی ساخت از نمایش**: کد ساخت را از کد کسب و کار جدا می‌کند
3. **انعطاف‌پذیری**: امکان ایجاد انواع مختلفی از یک شیء با استفاده از یک فرآیند ساخت یکسان
4. **قابل درک‌تر**: نسبت به سازنده‌های پیچیده با پارامترهای زیاد، خوانایی بهتری دارد
5. **پیروی از اصل Single Responsibility**: هر Builder مسئول ایجاد یک نوع خاص است

## معایب Builder

1. **پیچیدگی**: نیاز به ایجاد کلاس‌های متعدد دارد
2. **حجم کد**: ممکن است منجر به کدنویسی بیشتر نسبت به روش‌های ساده‌تر شود
3. **یادگیری**: درک الگو برای توسعه‌دهندگان تازه‌کار ممکن است دشوار باشد

## مثال کاربردی: ساخت ایمیل

public class Email

{

public string From { get; set; }

public string To { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

public List<string> Attachments { get; set; } = new List<string>();

public void Send()

{

Console.WriteLine($"Sending email from {From} to {To}");

Console.WriteLine($"Subject: {Subject}");

Console.WriteLine($"Body: {Body}");

if (Attachments.Any())

Console.WriteLine($"Attachments: {string.Join(", ", Attachments)}");

}

}

public interface IEmailBuilder

{

IEmailBuilder SetFrom(string from);

IEmailBuilder SetTo(string to);

IEmailBuilder SetSubject(string subject);

IEmailBuilder SetBody(string body);

IEmailBuilder AddAttachment(string attachment);

Email Build();

}

public class EmailBuilder : IEmailBuilder

{

private Email \_email = new Email();

public IEmailBuilder SetFrom(string from) { \_email.From = from; return this; }

public IEmailBuilder SetTo(string to) { \_email.To = to; return this; }

public IEmailBuilder SetSubject(string subject) { \_email.Subject = subject; return this; }

public IEmailBuilder SetBody(string body) { \_email.Body = body; return this; }

public IEmailBuilder AddAttachment(string attachment) {

\_email.Attachments.Add(attachment);

return this;

}

public Email Build() => \_email;

}

// استفاده با روش Fluent Interface

var email = new EmailBuilder()

.SetFrom("sender@example.com")

.SetTo("receiver@example.com")

.SetSubject("Meeting")

.SetBody("About our project")

.AddAttachment("project-plan.pdf")

.Build();

email.Send();

## تفاوت با الگوهای مشابه

* **Factory Method**: برای ایجاد یک محصول منفرد استفاده می‌شود
* **Abstract Factory**: برای ایجاد خانواده‌ای از محصولات مرتبط استفاده می‌شود
* **Builder**: برای ساخت اشیای پیچیده با فرآیند ساخت گام به گام استفاده می‌شود

## استفاده در #C و .NET

1. **StringBuilder**: یک نمونه کلاسیک از الگوی Builder
2. **UriBuilder**: برای ساخت آدرس‌های URL
3. **ConfigurationBuilder**: در ASP.NET Core برای ساخت پیکربندی

## بهترین روش‌های پیاده‌سازی

1. **روش Fluent Interface**: با برگرداندن this از هر متد، امکان زنجیره‌ای کردن فراخوانی‌ها را فراهم کنید
2. **Director اختیاری**: می‌توانید Director را حذف کنید و ساخت را مستقیماً توسط کلاینت کنترل کنید
3. **اعتبارسنجی**: در متد Build اعتبارسنجی پارامترهای ضروری را انجام دهید
4. **Immutable Builder**: برای ساخت اشیای immutable می‌توانید از این الگو استفاده کنید
5. **Generic Builder**: برای مواردی که چندین نوع محصول مشابه دارید:

public abstract class Builder<T> where T : new()

{

protected T \_product = new T();

public T Build()

{

var result = \_product;

\_product = new T(); // آماده برای ساخت شیء بعدی

return result;

}

}

این الگو زمانی مناسب است که:

* الگوریتم ساخت یک شیء پیچیده باید مستقل از اجزای آن باشد
* فرآیند ساخت باید امکان ایجاد نمایش‌های مختلف از شیء را فراهم کند
* شیء نیاز به پارامترهای زیادی در سازنده دارد (بیش از 4-5 پارامتر)

-------------------------------------------------------------------------------------------------------