Ronnie Souza • 2nd
.NET Developer| AI | AWS | C# | SQL | Entity Framework Core | xUnit | Docker | Dapper | Clean A...
1d •

Follow

Factory Pattern

In modern .NET and C# architecture, especially in scalable, service-oriented systems, the Factory Design Pattern is a true ally for those aiming for flexibility, low coupling, and clean code.

What is the Factory Pattern?

It's a creational design pattern that centralizes object creation logic, hiding the instantiation details. With it, you create objects in a controlled, safe, and extensible way, without scattering construction logic across your codebase.

Example – Multi-channel notification service (Image)

Imagine a system that sends alerts through different channels, depending on the user type or event. The Factory is used to abstract the correct channel instantiation:

Benefits:

Easy to add new channels (e.g., Push, WhatsApp)

Low coupling between client code and implementations

Centralized and reusable creation logic

Everyday scenarios where the Factory Pattern shines:

Multi-tenant systems with client-specific rules

 - A ConfigurationFactory generates tenant-specific configs while isolating logic.

Integrations with external APIs (e.g., AWS, Azure, Mercado Livre)

 - An HttpClientFactory provides pre-configured instances with headers, tokens, and retry policies.

Command creation in event-driven architectures

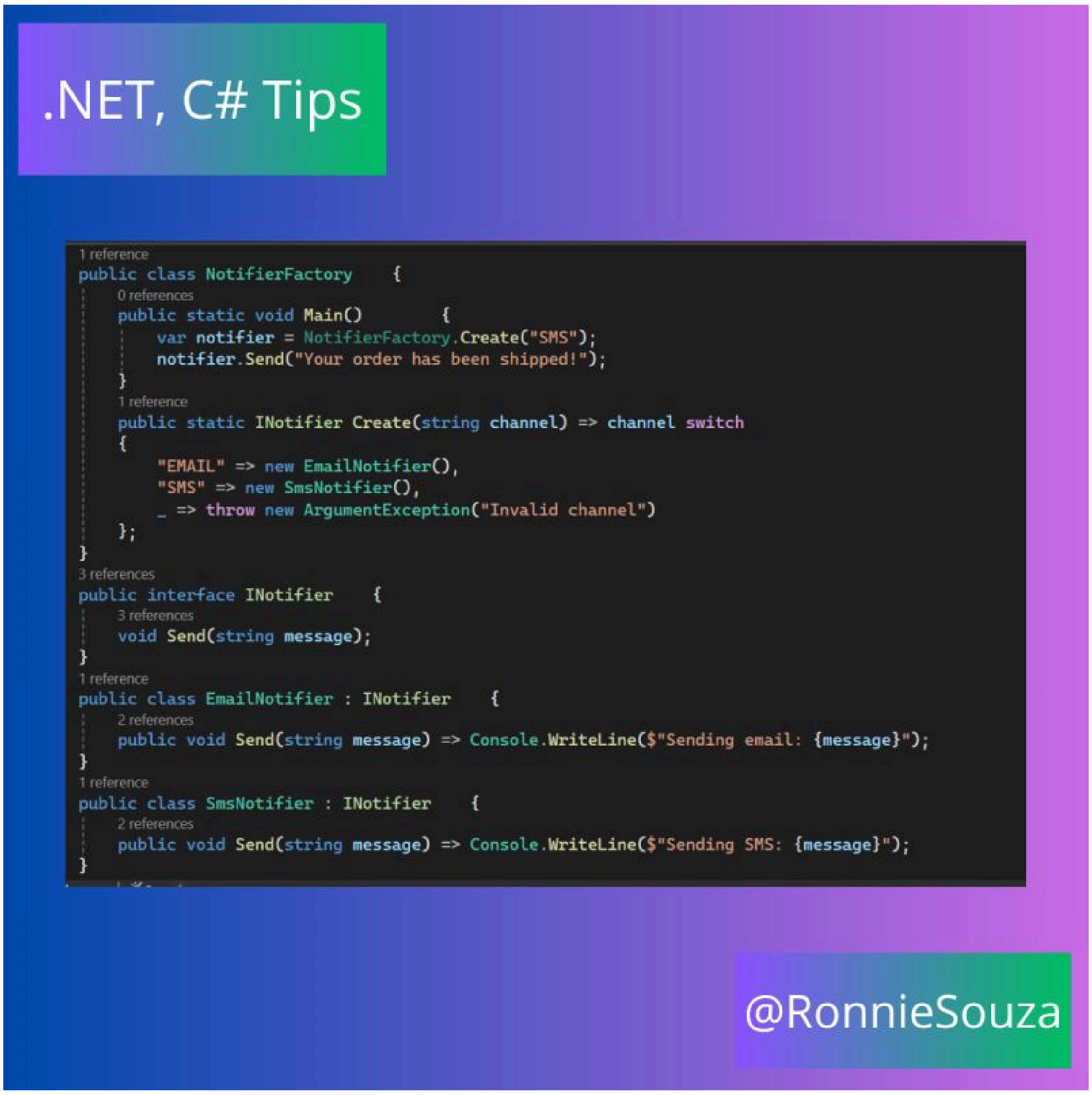
 - A CommandFactory dynamically builds commands from event bus payloads.

Factory isn't just about object creation. It's about building consistency, keeping the domain clean, and enabling future evolution.

#dotnet #csharp #backenddevelopment #cleancode #softwarearchitecture #microservices

#cloudcomputing #aws #designpatterns #scalability #softwareengineering #solid #factorypattern

#ddd



```
1 reference
public class NotifierFactory    {
    0 references
    public static void Main()        {
        var notifier = NotifierFactory.Create("SMS");
        notifier.Send("Your order has been shipped!");
    }
    1 reference
    public static INotifier Create(string channel) => channel switch
    {
        "EMAIL" => new EmailNotifier(),
        "SMS" => new SmsNotifier(),
        _ => throw new ArgumentException("Invalid channel")
    };
}
3 references
public interface INotifier    {
    3 references
    void Send(string message);
}
1 reference
public class EmailNotifier : INotifier    {
    2 references
    public void Send(string message) => Console.WriteLine($"Sending email: {message}");
}
1 reference
public class SmsNotifier : INotifier    {
    2 references
    public void Send(string message) => Console.WriteLine($"Sending SMS: {message}");
}
```

93 3 comments 2 reposts