Main Practices of Immutability in C# for Safer and More Efficient Code

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/fd75e6
out 21 2024

Abstract:
This article explores the concept of immutability in C#, highlighting its advantages, such as safety and efficiency in software development. It presents the implementation of immutable classes and structs, as well as tuples, which are immutable by default. The use of immutable collections is also discussed, emphasizing data integrity in complex applications.

Key words:
Immutability, C#, safe code, efficient code, programming practices, immutable types, immutable strings, functional programming, concurrency, thread-safe, software design, design patterns, encapsulation, immutable state, performance, code maintenance, best practices, immutable data structures, data security.

Understanding Immutability in C#

Immutability is a fundamental concept in programming, especially in languages like C#. When a variable is immutable, its state cannot be changed after its creation. This brings significant benefits in terms of safety and efficiency, as it reduces the risk of unwanted side effects and makes reasoning about the behavior of the code easier. With the increasing complexity of modern systems, where multiple parts of the code can interact in unexpected ways, immutability becomes a vital tool to ensure data integrity and application behavior predictability.

Advantages of Immutability

Immutability offers several advantages. First, it improves code safety. Since data cannot be changed, it is easier to ensure that functions will not unexpectedly modify the state of objects. Additionally, immutability can facilitate development in concurrent environments, where multiple threads may access the same data. This reduces the need for complex synchronization mechanisms, which can introduce bugs and additional complexity. Another important benefit is the greater ease of testing the code, as the predictability that immutability provides makes testing simpler and more effective.

Immutable Classes in C#

One of the most common ways to implement immutability in C# is through immutable classes. To create an immutable class, you must define its properties as read-only and initialize them through a constructor. Below is an example of an immutable class:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
```

```
        X = x;
        Y = y;
    }

    public Point Move(int deltaX, int deltaY)
    {
        return new Point(X + deltaX, Y + deltaY);
    }
}
```

In this example, the class Point is immutable. The method Move returns a new Point object with new coordinates, instead of modifying the existing object. This not only ensures that the original object remains unchanged, but also allows developers to easily track state changes over time, as each change results in a new object.

Immutable Structs

In C#, structs are value types that can also be used immutably. To create an immutable struct, you can follow a similar pattern to classes:

```
public struct Color
{
    public byte Red { get; }
    public byte Green { get; }
    public byte Blue { get; }

    public Color(byte red, byte green, byte blue)
    {
        Red = red;
        Green = green;
        Blue = blue;
    }

    public Color Mix(Color other)
    {
        return new Color(
            (byte)((Red + other.Red) / 2),
            (byte)((Green + other.Green) / 2),
            (byte)((Blue + other.Blue) / 2));
    }
}
```

The struct Color is an example of an immutable value type that can be used to represent RGB colors. Just like in classes, immutability in structs helps to avoid accidental modifications, as well as allowing each operation that results in a new color to not affect the original color.

Using Immutable Tuples

Tuples were introduced in C# 7 and are immutable by default. They provide a quick and easy way to group data without the need to create a class or struct. Here is an example:

```
var point = (X: 10, Y: 20);
var newPoint = (X: point.X + 5, Y: point.Y + 5);
```

Here, point is an immutable tuple. The new point is created without modifying the original. This makes tuples a popular choice for returning multiple values from methods without the complexity of defining a new class or struct.

Functional Patterns and Immutability

Immutability is a central concept in functional programming. In C#, we can adopt functional patterns using delegates and lambda expressions. Here is an example using LINQ:

```
var numbers = new List { 1, 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(n => n * n).ToList();
```

In this example, the operation of mapping the numbers to their squares is performed without modifying the original list, promoting immutability. This style of programming not only makes the code more readable but also facilitates maintenance and understanding of how data flows through the system.

Using Immutable Collections

Since C# 7.0, the System.Collections.Immutable library allows the use of immutable collections. These collections are useful when you want to ensure that a list or dictionary is not altered. An example of use is:

```
using System.Collections.Immutable;

var immutableList = ImmutableList.Create(1, 2, 3);
var newList = immutableList.Add(4);
```

The original list immutableList remains unchanged, while newList contains the new element. This is especially useful in applications where data integrity is critical, as it prevents unexpected modifications in collections that may be accessed from various parts of the code.

Best Practices for Implementing Immutability

Adopting immutability is not just about using immutable classes and structs. There are best practices that can help implement immutability effectively:

Use the readonly modifier for fields that should not be changed after initialization. This helps reinforce the intent that these fields are immutable.

Avoid methods that alter the state of the object; instead, create methods that return new objects. This approach allows you to maintain the integrity of existing objects.

Consider using immutable collections to avoid unwanted modifications. Immutable collections ensure that data is not changed inadvertently in parts of the code that should not have access to those changes.

Clearly document the intentions of immutability in your code. This helps other developers (or yourself in the future) quickly understand the reasons behind design decisions.

Use immutability in combination with design patterns like Builder or Prototype, which can help create instances of complex objects in a controlled and predictable manner.

Final Considerations on Immutability

Immutability is a powerful practice that can lead to safer and more efficient code. In a world where concurrency is increasingly common, implementing immutability can be a significant differentiator in software quality. C# offers various tools and techniques to facilitate the adoption of immutability, and it is a strategy that all developers should consider. By prioritizing immutability, you not only improve the safety and predictability of your code but also create a solid foundation for agile and sustainable development.

References

Microsoft. (2021). C# Programming Guide. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/
Microsoft. (2021). Immutable Collections. Available at: https://docs.microsoft.com/en-us/dotnet/standard/collections/immutable/

Nagib is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs at MIT and the University of Chicago. Other achievements include the authorship of a peer-reviewed article on chatbots presented at the University of Barcelona.