Tutorial: Update interfaces with default interface methods

Article • 11/03/2022 • 6 minutes to read

You can define an implementation when you declare a member of an interface. The most common scenario is to safely add members to an interface already released and used by innumerable clients.

In this tutorial, you'll learn how to:

- ✓ Extend interfaces safely by adding methods with implementations.
- ✓ Create parameterized implementations to provide greater flexibility.
- ✓ Enable implementers to provide a more specific implementation in the form of an override.

Prerequisites

You'll need to set up your machine to run .NET, including the C# compiler. The C# compiler is available with Visual Studio 2022 or the .NET SDK .

Scenario overview

This tutorial starts with version 1 of a customer relationship library. You can get the starter application on our samples repo on GitHub . The company that built this library intended customers with existing applications to adopt their library. They provided minimal interface definitions for users of their library to implement. Here's the interface definition for a customer:

C#

```csharp
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

They defined a second interface that represents an order:

```C#
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

From those interfaces, the team could build a library for their users to create a better experience for their customers. Their goal was to create a deeper relationship with existing customers and improve their relationships with new customers.

Now, it's time to upgrade the library for the next release. One of the requested features enables a loyalty discount for customers that have lots of orders. This new loyalty discount gets applied whenever a customer makes an order. The specific discount is a property of each individual customer. Each implementation of `ICustomer` can set different rules for the loyalty discount.

The most natural way to add this functionality is to enhance the `ICustomer` interface with a method to apply any loyalty discount. This design suggestion caused concern among experienced developers: "Interfaces are immutable once they've been released! This is a breaking change!" *Default interface implementations* for upgrading interfaces. The library authors can add new members to the interface and provide a default implementation for those members.

Default interface implementations enable developers to upgrade an interface while still enabling any implementors to override that implementation. Users of the library can accept the default implementation as a non-breaking change. If their business rules are different, they can override.

Upgrade with default interface methods

The team agreed on the most likely default implementation: a loyalty discount for customers.

The upgrade should provide the functionality to set two properties: the number of orders needed to be eligible for the discount, and the percentage of the discount. This makes it a perfect scenario for default interface methods. You can add a method to the `ICustomer` interface, and provide the most likely implementation. All existing, and any new implementations can use the default implementation, or provide their own.

First, add the new method to the interface, including the body of the method:

C#

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

The library author wrote a first test to check the implementation:

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,
31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

Notice the following portion of the test:

C#

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

That cast from `SampleCustomer` to `ICustomer` is necessary. The `SampleCustomer` class doesn't need to provide an implementation for `ComputeLoyaltyDiscount`; that's provided by the `ICustomer` interface. However, the `SampleCustomer` class doesn't inherit members from its interfaces. That rule hasn't changed. In order to call any method declared and implemented in the interface, the variable must be the type of the interface, `ICustomer` in this example.

Provide parameterization

That's a good start. But, the default implementation is too restrictive. Many consumers of this system may choose different thresholds for number of purchases, a different length of membership, or a different percentage discount. You can provide a better upgrade experience for more customers by providing a way to set those parameters. Let's add a static method that sets those three parameters controlling the default implementation:

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

There are many new language capabilities shown in that small code fragment. Interfaces can now include static members, including fields and methods. Different access modifiers are also enabled. The additional fields are private, the new method is public. Any of the modifiers are allowed on interface members.

Applications that use the general formula for computing the loyalty discount, but different parameters, don't need to provide a custom implementation; they can set the arguments through a static method. For example, the following code sets a "customer appreciation" that rewards any customer with more than one month's membership:

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

Extend the default implementation

The code you've added so far has provided a convenient implementation for those scenarios where users want something like the default implementation, or to provide an unrelated set of rules. For a final feature, let's refactor the code a bit to enable scenarios where users may want to build on the default implementation.

Consider a startup that wants to attract new customers. They offer a 50% discount off a new customer's first order. Otherwise, existing customers get the standard discount. The library author needs to move the default implementation into a `protected static` method so that any class implementing this interface can reuse the code in their implementation. The default implementation of the interface member calls this shared method as well:

C#

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

In an implementation of a class that implements this interface, the override can call the static helper method, and extend that logic to provide the "new customer" discount:

```C#
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

You can see the entire finished code in our samples repo on GitHub . You can get the starter application on our samples repo on GitHub .

These new features mean that interfaces can be updated safely when there's a reasonable default implementation for those new members. Carefully design interfaces to express single functional ideas that can be implemented by multiple classes. That makes it easier to upgrade those interface definitions when new requirements are discovered for that same functional idea.