Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

MORE *Effective* C#

Second Edition

COVERS C# 7.0

50 Specific Ways to Improve Your C#

Content Update Program

FREE...See Details Inside

Bill Wagner

More Effective C#

Second Edition

implicitly enhanced. In that sense, base classes provide a way to extend the behavior of several types efficiently over time. When you add and implement functionality in the base class, all derived classes immediately incorporate that behavior. Adding a member to an interface, however, breaks all the classes that implement that interface. They will not contain the new method and will no longer compile. Each implementer must update that type to include the new member. Alternatively, if you need to add functionality to an interface without breaking the existing code, you can create a new interface and have it inherit from the existing interface.

Choosing between an abstract base class and an interface is a question of how best to support your abstractions over time. Interfaces are fixed: You release an interface as a contract for a set of functionality that any type can implement. In contrast, base classes can be extended over time, and those extensions then become part of every derived class.

The two models can be mixed to reuse implementation code while supporting multiple interfaces. One obvious example in the .NET Framework is the `IEnumerable<T>` interface and the `System.Linq.Enumerable` class. The `System.Linq.Enumerable` class contains a large number of extension methods defined on the `System.Collections.Generic.IEnumerable<T>` interface. That separation enables very important benefits. Any class that implements `IEnumerable<T>` appears to include all those extension methods, but those additional methods are not formally defined in the `IEnumerable<T>` interface. As a consequence, class developers do not need to create their own implementation of all those methods.

As an example, consider the following class, which implements `IEnumerable<T>` for weather observations:

```
public enum Direction
{
    North,
    NorthEast,
    East,
    SouthEast,
    South,
    SouthWest,
    West,
    NorthWest
}
```

```
public class WeatherData
{
    public WeatherData(double temp, int speed,
        Direction direction)
    {
        Temperature = temp;
        WindSpeed = speed;
        WindDirection = direction;
    }
    public double Temperature { get;  }
    public int WindSpeed { get;  }
    public Direction WindDirection { get;  }
    public override string ToString() =>
        @$"Temperature = {Temperature}, Wind is {WindSpeed}
mph from the {WindDirection}";
}

public class WeatherDataStream : IEnumerable<WeatherData>
{
    private Random generator = new Random();

    public WeatherDataStream(string location)
    {
        // Elided
    }

    private IEnumerator<WeatherData> getElements()
    {
        // Real implementation would read from
        // a weather station.
        for (int i = 0; i < 100; i++)
        yield return new WeatherData(
            temp: generator.NextDouble() * 90,
            speed: generator.Next(70),
            direction: (Direction)generator.Next(7)
        );
    }

    public IEnumerator<WeatherData> GetEnumerator() =>
        getElements();
```

```
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() =>
        getElements();
}
```

To model a sequence of weather observations, the WeatherStream class implements `IEnumerable<WeatherData>`. That means creating two methods: the `GetEnumerator<T>` method and the classic `GetEnumerator` method. The latter interface is explicitly implemented so that client code will naturally be drawn to the generic interface rather than the version typed as `System.Object`.

The implementation of those two methods means that the WeatherStream class supports all the extension methods defined in `System.Linq.Enumerable`. That means WeatherStream can be a source for LINQ queries:

```
var warmDays = from item in
                    new WeatherDataStream("Ann Arbor")
               where item.Temperature > 80
               select item;
```

LINQ query syntax compiles to method calls. For example, the preceding query translates to the following calls:

```
var warmDays2 = new WeatherDataStream("Ann Arbor").
    Where(item => item.Temperature > 80);
```

In this code, the Where and Select calls might seem to belong to IEnumerable<WeatherData>, but they do not. That is, those methods appear to belong to IEnumerable<WeatherData> because they are extension methods, but they are actually static methods in `System.Linq.Enumerable`. The compiler translates those calls into the following static calls:

```
// Don't write this; presented for explanatory purposes only
var warmDays3 = Enumerable.Select(
                    Enumerable.Where(
                    new WeatherDataStream("Ann Arbor"),
                    item => item.Temperature > 80),
                    item => item);
```

The preceding code illustrates that interfaces really can't contain implementation. You can emulate that state by using extension methods. LINQ does so by creating several extension methods on `IEnumerable<T>` in the class.

That brings us to the topic of using interfaces as parameters and return values. An interface can be implemented by any number of unrelated types. Coding to interfaces provides greater flexibility for other developers than coding to base class types. That's important because of the single inheritance hierarchy enforced by the .NET type system.

The following three methods perform the same task:

```
public static void PrintCollection<T>(
    IEnumerable<T> collection)
{
    foreach (T o in collection)
        Console.WriteLine($"Collection contains {o}");
}

public static void PrintCollection(
    System.Collections.IEnumerable collection)
{
    foreach (object o in collection)
        Console.WriteLine($"Collection contains {o}");
}

public static void PrintCollection(
    WeatherDataStream collection)
{
    foreach (object o in collection)
        Console.WriteLine($"Collection contains {o}");
}
```

The first method is most reusable. Any type that supports `IEnumerable<T>` can use that method. In addition to WeatherDataStream, you can use `List<T>`, `SortedList<T>`, any array, and the results of any LINQ query. The second method will also work with many types, but uses the less preferable nongeneric `IEnumerable`. The third method is far less reusable; it cannot be used with `Arrays`, `ArrayLists`, `DataTables`, `Hashtables`, `ImageLists`, or many other collection classes. Coding the method using interfaces as its parameter types is far more generic and far easier to reuse.

Using interfaces to define the APIs for a class also provides greater flexibility. The WeatherDataStream class could implement a method that returned a collection of WeatherData objects. That would look something like this:

```
public List<WeatherData> DataSequence => sequence;
private List<WeatherData> sequence = new List<WeatherData>();
```

Unfortunately, this code leaves you vulnerable to future problems. At some point, you might change from using a List<WeatherData> to exposing an array, a `SortedList<T>`. Any of those changes will break the code. Sure, you can change the parameter type, but that's changing the public interface to your class. Changing the public interface to a class causes you to make many more changes to a large system; you would need to change all the locations where the public property was accessed.

Another problem with this code is more immediate and more troubling: The `List<T>` class provides numerous methods to change the data it contains. Users of your class could delete, modify, or even replace every object in the sequence—which is almost certainly not your intent. Luckily, you can limit the capabilities of the users of your class. Instead of returning a reference to some internal object, you should return the interface that you intend clients to use—in this case, IEnumerable<WeatherData>.

When your type exposes properties as class types, it exposes the entire interface to that class. Using interfaces, you can choose to expose only those methods and properties you want clients to use. The class used to implement the interface is an implementation detail that can change over time (see Item 17).

Furthermore, unrelated types can implement the same interface. Suppose you're building an application that manages employees, customers, and vendors. Those entities are unrelated, at least in terms of the class hierarchy. Nevertheless, they share some common functionality. They all have names, and you will likely display those names in controls in your applications.

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string Name => $"{LastName}, {FirstName}";
    // Other details elided
}

public class Customer
{
    public string Name => customerName;
```

```
    // Other details elided
    private string customerName;
}

public class Vendor
{
    public string Name => vendorName;

    // Other details elided
    private string vendorName;
}
```

The Employee, Customer, and Vendor classes should not share a common base class, but they do share some properties: names (as shown earlier), addresses, and contact phone numbers. You could factor out those properties into an interface:

```
public interface IContactInfo
{
    string Name { get; }
    PhoneNumber PrimaryContact { get; }
    PhoneNumber Fax { get; }
    Address PrimaryAddress { get; }
}

public class Employee : IContactInfo
{
    // Implementation elided
}
```

This new interface can simplify your programming tasks by letting you build common routines for unrelated types:

```
public void PrintMailingLabel(IContactInfo ic)
{
    // Implementation deleted
}
```

This single routine works for all entities that implement the IContactInfo interface. Now Customer, Employee, and Vendor all use the same routine—but only because you factored them into interfaces.