Create Mixins with Interfaces and Extension Methods

by Bill Wagner

Mixins are small utility classes that define some useful functionality that will usually become part of something larger. For example, a class that implements a serial number to uniquely identify an object, or a timestamp that reports when an object was created. In C++, these mixin classes would be created using regular class definitions, and larger classes that wanted to use the mixin behavior would simply inherit from the mixin, in addition to any base classes it needed. Because the .NET Framework does not support multiple inheritance, we've typically used interface to define mixins. That works, but it's not as powerful. One of the limitations of using interfaces to define mixin behavior is that every class that implements an interface must reimplement every method defined in that interface. That leads many developers to creating minimal interface definitions. (IEqualityComparer<T> has one method, IComparable<T> has only method). In those instances, the interface's full functionality is completely defined in those small methods. But, what about cases where your interface can, or should, contain many more methods in order to support the developers that want to use the interface? That's a perfect use case for extension methods. By creating extension methods using the interface, you can inject many more methods into the public definition of the interface, without forcing every developer to re-create those methods whenever he or she implements that interface.

I've created a simple example based on strings that represent file paths. I often write classes that contain files or directories, usually for offline caches, or similar needs. The System.IO.Path class contains many methods that work with strings that represent paths, but those static methods do not become part of my new type's public interface.

Extension methods can correct that. First, define a minimal interface that returns the string that represents the path to the target directory or file:

```
public interface IPath
{
string FilePath
{
get;
}
}
```

That's about as minimal as it gets. Any implementers only need to implement a single property: the string representing the path. But this simple interface is not that useful. Client code needs to add all the real functionality to the interface in order to get any work done. Instead of forcing all that work on our client developers, let’s create a set of extension methods that will be available as though they are implemented in every class that supports the IPath interface. For example, let’s write a pair of methods to see if the path is a directory or a file:

```
public static bool IsFile(this IPath srcPath)
{
return File.Exists(srcPath.FilePath);
}
public static bool IsDirectory(this IPath srcPath)
{
return Directory.Exists(srcPath.FilePath);
}
```

Those two methods can now be called on any type that implements the IPath interface, as though they were members of that type. Next, let’s add a method to see if the chosen path exists, as either a directory or a file:

```
public static bool Exists(this IPath srcPath)
{
bool fileExists = File.Exists(srcPath.FilePath);
bool dirExists = Directory.Exists(srcPath.FilePath);
return fileExists || dirExists;
}
```

After this much work, any type that supports the IPath interface must implement one property (the FilePath string property), and it acts as though it’s implemented 3 methods in addition to that one property. Great. Well, now suppose time passes. Suddenly, you get a new requirement (isn’t that the way it always works?) to produce an enumeration of all the string components that are part of a full path. It’s another extension method:

```
public static IEnumerable<string> FullPathComponents(this IPath srcPath)
{
string fullPath = Path.GetFullPath(srcPath.FilePath);
string[] components = fullPath.Split(
Path.VolumeSeparatorChar,
Path.PathSeparator,
Path.DirectorySeparatorChar);
foreach (string s in components)
{
if (String.IsNullOrEmpty(s) == false)
yield return s;
}
}
```

The beauty of this idiom is that every type that implemented the IPath interface now has access to the FullPathComponents() method. You did not have to update the interface, which would be a breaking change. You didn’t have to create some new IPath2 interface that supports the old methods in addition to the new one. Even better, all classes that implement the IPath interface are binary compatible with the new extension method. You don’t even need to recompile and re-release assemblies to get the new feature!

I’ll admit that none of these routines are rocket science, but the concept behind them is very significant. Anytime you create a new class that contains a path, you need to implement the IPath interface, which is a trivially simple property. The extension methods have added four new methods, and you can reuse the method declarations *and* the implementation. You can this extend with other methods yourself to get a better understanding of how you can define a minimal interface, and use extension methods to create as many convenience methods as you users might want.

The .NET Framework made extensive use of this in the upcoming Orcas release with the Enumerable class and the Queryable class. Enumerable contains more than 30 methods that are injected into the methods supported by IEnumerable<T>. Any class that implements IEnumerable<T>, which contains 1 method, GetEnumerator(), suddenly supports more than 30 operations.

It’s an important technique to remember: declare the minimum number of methods and properties in your interface, and provide the many convenience methods in a set of extension methods that are accessible to all the classes that implement that interface.