Dealing with Conflicting Interfaces: Part II - .NET

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In Part I we discussed the issue of conflicting or colliding interfaces and saw how we can address that in Java. While the Java solution is workable, it is not elegant. We can't directly substitute an object of the "implementing" class for the desired interface. .NET offers an interesting facility called "Explicit interface." This allows for a class to override multiple methods with the same signature in a class. Explicit interfaces provide an easy mechanism to implement conflicting interfaces.

Conflicting Interfaces

I assume you have read the article "Dealing with Conflicting Interfaces: Part I – Java." We have two interfaces `Pianist` and `Athlete`. Both the interfaces have a play method. We are interested in implementing these two interfaces in the class `Person`.

The problem in .NET as in Java is that you can't have more than one implementation of the same method with the same signature in a class. However, .NET provides a simple workaround to address this problem.

Explicit Interface

A class may implement some or all of the methods of an interface as an explicit implementation. If a method is implemented *explicitly*, then the method is not accessible directly using a reference of the type. Instead you will have to use the reference of the interface type to access it. The explicit method implementation is not declared public and it is implemented using the fully qualified name of the method, that is, interface name followed by a dot and then the method name.

Pleasure and perils of Explicit Interface

A class is allowed to override more than one method with the same name and signature, as long as no more than one of these methods is implemented implicitly. This alleviates the problem of conflicting or colliding interfaces as we will see later.

You don't have to make all the methods of an interface explicit in your class. This allows for you to implement and expose some methods of the interface as implicit methods while eliminating some methods from the class's public interface. The explicit method then can only be accessed using the interface reference as shown below. Say we have an interface `I1` and a class `SomeClass` that implements it.

```
public interface I1
{
      void Foo1();
      void Foo2();
}

public class SomeClass : I1
```

```
{
      public void Foo1() {…}
      void I1.Foo2() {…}

}
```

Now, we can access `Foo1()` using `SomeClass` reference as in

```
SomeClass ref1 = new SomeClass();
ref1.Foo1();
```

We may also access `Foo1()`using the interface `I1` reference as in:

```
I1 refI1 = ref1;
refI1.Foo1();
```

However, the method `Foo2()` is not accessible using the reference of type `SomeClass`. It is accessible only through reference of type `I1`.

```
ref1.Foo2(); // ERROR
refI1.Foo2(); // OK
```

You may be tempted to use explicit interface to partially implement an interface. In the explicit method you may throw an exception that the method is not implemented. You may argue that a user of the class will not be able to directly call the methods you have *withdrawn*. If they cast the object as the interface and then call the method, then they will get a runtime exception. For instance, you can see an example of this behavior in the `System.Array` class. This class implements the `IList` interface, but makes most of the methods explicit. If you try to treat an array as `IList`, and call the `Add()` method, you will get a runtime exception. Use caution in using explicit interfaces for implementing an interface partially. You may end up violating the *Liskov's Substitution Principle* which states that "an instance of derived must be substitutable wherever an instance of base is used, without the need for the user to know the difference." Your code may not be extensible.

Using Explicit Interface to address colliding interfaces

Let's first see the two interfaces in C#.

```
public interface Pianist
{
      string getName();
      void play();
}

public interface Athlete
{
      string getName();
      void play();
}
```

Now, let's implement the `Pianist` interface in the class `Person`:

```
public class Person : Pianist
{
	public String getName()
	{
		return "Joe";
	}

	public void play()
	{
		Console.WriteLine("Joe playing Piano");
	}
}
```

And, finally, here is our implementation of the `Athlete` interface:

```
public class Person : Pianist, Athlete
{
	public String getName()
	{
		return "Joe";
	}

	public void play()
	{
		Console.WriteLine("Joe playing Piano");
	}

	public void playAthlete()
	{
		Console.WriteLine("Joe sprints");
	}

	void Athlete.play()
	{
		playAthlete();
	}
}
```

The test case that exercises this code is shown below:

```
class MyTestCase
{
	public static void usePianist(Pianist p)
	{
		Console.WriteLine("Using Pianist");
		Console.WriteLine("Name: " + p.getName());
		p.play();
	}

	public static void useAthlete(Athlete a)
	{
		Console.WriteLine("Using Athlete");
		Console.WriteLine("Name: " + a.getName());
```

```
            a.play();
        }

        [STAThread]
        static void Main(string[] args)
        {
            Person p = new Person();

            usePianist(p);
            useAthlete(p);

            Console.WriteLine("In Main");
            Console.WriteLine(p.getName());
            p.play();
        }
    }
```

And the output from the above code is:

```
Using Pianist
Name: Joe
Joe playing Piano
Using Athlete
Name: Joe
Joe sprints
In Main
Joe
Joe playing Piano
```

In `Person` class, I have chosen to implement the `play()` method as implicit interface for the `Pianist play()` method. You may implement that as an explicit method, if you desire. The user of `Person` class may use the `playAthelete()` method directly on a reference of type `Person`, but can call the `play()` method of `Athlete` directly only using a reference of type `Athlete`.

Conclusion

In Part I we say how to deal with conflicting interfaces in Java. In this part II we have seen how we can handle that in .NET. Explicit interfaces in .NET allow us to effectively implement methods when their names and signature collide. Use caution in using explicit interfaces, however, when it comes to partially implementing interface methods. This may lead to extensibility problems.

References

1. http://msdn.microsoft.com