Technical Note

Multiple Class Inheritance

Author(s): R&D Department

Publication date: December 15, 2006

Multiple Class Inheritance

Overview

The .NET Framework does not support multiple class inheritance. Some people will say, "Who needs multiple inheritance?" My answer is very short… ME! (And if you are reading this article, you probably do too.) I don't need it often but when I need it, I need it badly. Because .NET does not support multiple inheritance, we have to simulate it through delegation. This can be long and tedious to implement. For this reason, we created a small utility called **MHGenerator** that automates the creation of code simulating multiple inheritance.

This article describes how to simulate multiple inheritance in C#. It also describes how to use the **MHGenerator** utility to automate this process.

Multiple inheritance

The .NET Framework supports single inheritance of classes, but allows multiple interface implementation. Single inheritance is simple to achieve: just define your class and add a **: BaseClass** in your declaration.

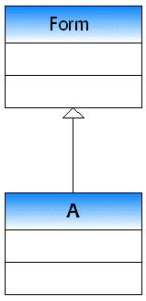

Figure 1. Single inheritance.

```
public class A : System.Windows.Forms.Form {
     …
}
```

To simulate multiple inheritance, you can use composition, redefine the base class and delegate the job to the embedded class.

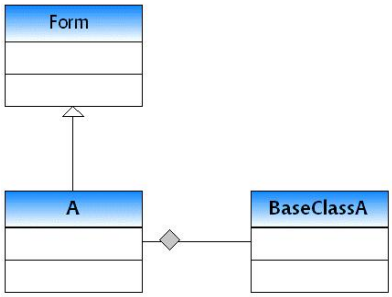

Figure 2. Simulating multiple inheritance using composition.

```
public class BaseClassA {
      private A   m_outer;

      public BaseClassA(A outer) {
            m_outer = a;
      }

      public void DoSomething() {}
}

public class A : System.Windows.Forms.Form {
      private BaseClassA      m_del;

      public A() {
            m_del = new BaseClassA(this);
      }

      public void DoSomething() {
            m_del.DoSomething();
      }
}
```

In this simple case, everything seems to work fine. However, you still cannot use class **A** when class **BaseClassA** is expected (in other words, class **A** is not a **BaseClassA**). You can implicitly cast your class, and thereby pass a reference to class **A** in a method expecting a class **BaseClassA**:

```
public class A : System.Windows.Forms {
      private BaseClassA      m_del;
      …

      public static implicit operator BaseClassA(A type) {
            return(type.m_del);
      }
}
```

You can also allow the explicit casting from **BaseClassA** to **A**:

```
public class BaseClassA {
      private A   m_outer;

…

public static explicit operator A(BaseClassA type) {
      return(m_outer);
}

…
}
```

Still, several problems remain:

- How to support the inheritance of methods and properties?
- How to handle protected members?
- How to handle events?

Inheritance of methods and properties

What happens if a method in **BaseClassA** is defined as virtual? It means that the method can be overridden. When overridden, the new method must be used each time a user of the class calls the method. The new method must also be called if referenced in the base class.

```
public class BaseClassA {
      public virtual void DoSomething() {
          MessageBox.Show("BaseClassA"}
      }

      public virtual void DoSomethingElse() {
          DoSomething();
      }
}

public class A : System.Windows.Forms {
      private BaseClassA      m_del;

      public A() {
            m_del = new BaseClassA();
      }

      public virtual void DoSomething() {
            m_del.DoSomething();
      }

      public virtual void DoSomethingElse() {
            m_del.DoSomethingElse();
      }
}
```

This code does not work as expected. If you create a new class **B** inheriting from class **A** and override the method **DoSomething**, problems occur.

```
public class B : A {
      public override void DoSomething() {
            MessageBox.Show("B");
      }
}
```

If you call method **B.DoSomething**, message *B* is displayed. Unfortunately, if you call method **B.DoSomethingElse**, message *BaseClassA* is displayed, which is not what you want.

To solve this problem, you will have to use a more complex scenario involving an interface and what we call a "delegater class". A "delegater class" is a class that dispatches calls to methods or properties to the base class (in our example, **BaseClassA**) or to the outer class (in our example, **A**). Therefore, you can use the following solution:

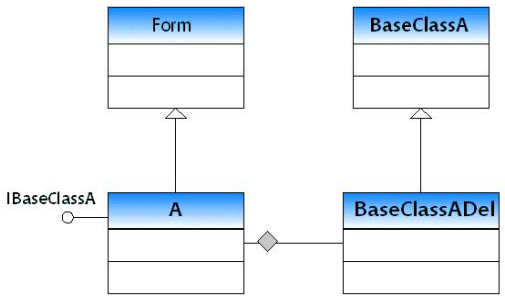

Figure 3. Multiple inheritance through an interface and a delegater class.

```
public interface IBaseClassA {
      void DoSomething();
      void DoSomethingElse();
}

public class BaseClassADel : BaseClassA {
      private IBaseClassA     m_outer;

      public BaseClassADel(IBaseClassA outer) {
            m_outer = outer;
      }

      public static explicit operator A(BaseClassADel type) {
            return(type.m_outer as A);
      }

      public override void DoSomething() {
            m_outer.DoSomething();
      }

      public void _baseDoSomething() {
            base.DoSomething();
      }

      public override void DoSomethingElse() {
            m_outer.DoSomethingElse();
      }

      public void _baseDoSomethingElse() {
            base.DoSomethingElse();
      }
}

public class A : System.Windows.Forms.Form, IBaseClassA {
      private BaseClassADel   m_del;

      public A() {
            m_del = new BaseClassADel(this);
      }

      public static implicit operator BaseClassA(A type) {
            return(type.m_del);
      }

      public virtual void DoSomething() {
            m_del._baseDoSomething();
      }

      public virtual void DoSomethingElse() {
            m_del._baseDoSomethingElse();
      }
}
```

Now, if you try again to inherit from class **A** and override method **DoSomething**, everything works well. Calling method **B.DoSomethingElse** correctly displays *B*.

The use of the **IBaseClassA** interface can seem like an overkill, but it will enable you to use explicit interface definition later to solve accessibility problems on virtual protected methods.

Virtual protected member

What happens if the **DoSomething** method is defined as *protected* instead of *public*? At the interface level, you cannot specify an accessibility modifier. The method is implicitly defined as *public*. In the delegater class, the method cannot be defined as *protected* either. Doing so will make the method inaccessible from class **A**. To implement the **IBaseClassA** interface, class **A** must define the **DoSomething** method and make it *public*. However, doing so makes *public* a method that the base class specifies as *protected*.

To solve this problem, you can simply choose to implement the **IBaseClassA.DoSomething** method explicitly:

```
public class A : System.Windows.Forms.Form, IBaseClassA {
      …
      protected virtual void DoSomething() {
            m_del._baseDoSomething();
      }

      IBaseClassA.DoSomething() {
            DoSomething();
      }
      …
}
```

The explicit definition of the **DoSomething** method lets you implement the **IBaseClassA** interface. Because this method is defined explicitly, you can access it directly only from a reference to the interface **IBaseClass** (a cast to the interface is needed), not from a reference to class **A**. The new definition of **DoSomething** being defined as *protected*, everyone is happy.

The implementation of properties follows the same approach as for the methods.

Events

Now add an event to class **BaseClassA** as follows:

```
public class BaseClassA {
      public event System.EventArgs MyEvent;
      public void DoSomething() {}
}
```

To implement inheritance from this class through delegation, you must be able to access the event through class **A**. You can simply add the event to this class.

```
public class A : System.Windows.Forms.Form, IBaseClassA {
      private BaseClassADel   m_del;

      public A() {
            m del = new BaseClassADel(this);
      }

      public event System.EventArgs MyEvent; // Bad! No! Don't! Grr…
       …
}
```

Doing so does not give you access to the event of class **BaseClassA**. It simply redefines a new event with the same name. Registering to this event will not register to the event of class **BaseClassA**. In fact, what you want is to allow registration to the event of class **BaseClassA** from class **A**. Fortunately, C# lets you do that using a not very well known syntax:

```
public class A : System.Windows.Forms.Form, IBaseClassA {
      private BaseClassADel   m_del;

      public A() {
            m_del = new BaseClassADel(this);
      }

      public event System.EventArgs MyEvent {
            add {
                  m_del.MyEvent += value;
            }
            remove {
                  m_del.MyEvent -= value;
            }
      }
       …
}
```

All this is very nice. If you want, you can simulate multiple inheritance in C#. To inherit from a class through delegation, you just have to type code, code and more code. Like most good programmers, you probably hate to type a massive amount of code that does nothing but delegate. Being involved in the development of REP++, which makes intensive use of reflection, I grabbed the idea of reflection to create a utility program that generates the code needed to inherit through delegation.

MHGenerator

The **MHGenerator** program is a command line utility generating a class in C# that inherits directly from a base class and indirectly from a second base class through delegation. Figure 4 illustrates what the utility does.

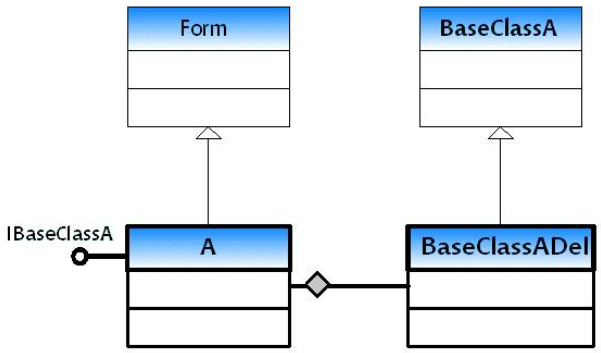

*Figure 4. **MHGenerator** generates class **A** and its associated delegater class **BaseClassADel**.*

In this Figure, **A** is the new generated class. **A** directly inherits from the **Form** class. **A** delegates to class **BaseClassA** through the **BaseClassADel** delegater class. In this scheme, **A** seems to inherit from both **Form** and **BaseClassA** classes.

To use this utility, you must pass the arguments described in Table 1. You can also call the utility using a command file by using the @ prefix in front of the file.

*Table 1. Arguments to the **MHGenerator** command line utility.*

Argument	Description
/as:AssemblyName	You must specify the full path of the assemblies containing the source classes and their references. The following assemblies are already loaded by the utility and must not be specified: **mscorlib**, **System.Windows.Forms** and **System.Web**. Each assembly must have its own */as* clause.
/if:InheritFromClassName	Name of the class from which the new class directly inherits. This class must have a public, parameterless constructor.
/dt:DelegateToClassName	Name of the class to which the new class delegates.
/nt:NewTypeName	Name of the new class.
/sc:Scope	Scope of the new class. The scope can be *public*, *protected*, *private* or *internal*.
/out:OutFileName	Name of the resulting file. If not specified, the name of the file is **NewTypeName.cs** where **NewTypeName** is the name of the new class.
/hf:HelpFile	Name of a generated XML help file. If this switch is used, the help file of the generated member will use this help file as a template.
/hl	Hide the generated code to the debugger using the #line hidden attribute. Using this switch eases the debugging of the application.
/v	Verbose. Provides more information while generating the resulting class.
/w	Display warnings, if any.

Limitations

The generator requires that the base class from which the new class inherits directly (in our example, **Form**) defines a parameterless, nonprivate constructor. The class that is delegated to (**BaseClassA**), however, does not have this prerequisite. It only needs one or more accessible constructors. Effectively, the constructor of the new class needs to call the base classes' constructors. If both base classes (**Form** and **BaseClassA**) have many constructors, things get more complex. It can also cause constructor overloading clashes.

Name clash

What happens when the two base classes contain methods with the same signature? Must you take the methods from the first or the second class? Or delegate to both? There is no way to tell. For this reason, **MHGenerator** will generate the conflicting methods in the delegater class (**BaseClassADel**) but will not add them in the final generated class. It will also display a warning (if the */w* option was specified). If necessary, the problem can be fixed later by subclassing the generated class (**A**) and calling the delegater class method.

In .NET, all objects inherit from **System.Object**, continually causing name clashes. Because of that, all methods in **System.Object** are ignored by the generator in order to remove the related warning.

What about inheriting from three base classes?

To inherit from three base classes, use the generator twice. To inherit from four base classes, well, I'm sure you get the picture...