

C#... Virtual Delphi?

by Joanna Carter

C# is an excellent language as it stands. As with all languages, it has extra features that allow us to do things that we cannot do in other languages, but it is also lacking features that are available in other languages. C++, Java and Delphi programmers alike will soon find subtle differences between C# and the languages with which they are more familiar.

The feature that is missing from C# that I would like to focus on in this article is the concept of Virtual Constructors; a concept that gives a language like Delphi a great deal of power that C++ programmers can only dream of whilst busily writing Abstract Factories.

How Does Delphi Do It?

To understand what Virtual Constructors are, let us look at a small snippet of Delphi code:

```
type
  TShapeClass = class of TShape;
  TShape = class
    ...
    constructor Create(Pos: TPoint; Size:
      TPoint); virtual;
end;

TCircle = class(TShape)
  ...
  constructor Create(Pos: TPoint; Size:
    TPoint); override;
end;

TSquare = class(TShape)
  ...
  constructor Create(Pos: TPoint; Size:
    TPoint); override;
end;

var
  ShapeClass: TShapeClass;
  Shape: TShape;
begin
  ShapeClass := TCircle;
  Shape := ShapeClass.Create(Point(0, 0),
    Point(25, 25));
end;
```

The above code shows a simple hierarchy with a Class Reference type declaration of TShapeClass. Such a declaration allows us to hold a reference to a class type rather than an instance of a class. As you can see from the code snippet, the TShapeClass Class Reference variable can hold any class type that descends from TShape, allowing us to call any class method including the constructor of the base TShape class and have it execute the appropriate version of the constructor for the chosen subclass. The sample code will instantiate a TCircle because that is the type that was assigned to the Class Reference variable and because the constructor of TShape was declared as virtual and overridden in TCircle.

Unfortunately C# does not possess such Class Reference types and so such code would not be possible 'out of the box'.

Sharper C#

Although there is no Class Reference type already implemented in C#, there is nothing to stop us creating one of our own for use in a similar manner. One of the factors that makes a Class Reference work in Delphi and stops it doing so in C# is the syntax used for the declaration of the constructor.

In Delphi, constructors are usually named Create, no matter for which class they are the constructor. A constructor is also regarded as a special case of a class method.

In C#, constructors have to be named with the same name as the class for which they are the constructor. This prevents us from having the same named constructor in a class named Circle that we would have in a class named Shape.

What we need is a way of declaring a type that allows us to declare the type of a base class, have that type declare a constructor named Create and be able to assign any class that is a descendant of the base class to that class type. Then the same Create method that would return an instance of the base class would also return an instance of any of the sub classes.

Let us start by declaring a class that we can use as a fundamental Class Reference type:

```
class Class
{
  private Type classType;

  public Class(Type aType) {...}

  public Type ClassType
  {
    get {...}
    set {...}
  }

  public object Create(object[] args) {...}
}
```

Before we get into the details of the methods involved in the class, let's go through the interface to the class as declared above.

We are going to need a private field that can maintain a reference to the Type of the class that we will assign to this Class Reference; this is the purpose of classType.

Then we are going to need a constructor that allows us to create an instance of this Class Reference type for a given Type.

Next, we need a property that will allow us to refer to the Type that is current for the Class Reference and to set that Type to any of the types that belong to the hierarchy starting at the base class.

Finally, we need a method that we can use as a replacement for the Delphi concept of the single name constructor, so we will call that method Create. Notice that it is a static (non-virtual) method and that it takes an array of objects as its parameter.

Now let's look at each of those class members in more detail, ignoring the private field:

```
public Type ClassType
{
    get
    {
        return classType;
    }
    set
    {
        if (value.Equals(classType) |
            value.IsSubclassOf(classType))
            classType = value;
        else
            throw new TypeNotDerivedException
                (value, classType);
    }
}
```

The getter for the ClassType property simply returns whatever Type is held in the private classType field, but the setter has to check whether the type being passed in is either the same as the base class of our hierarchy or at least a type derived from our base class. Provided it meets those criteria, it is assigned to the private field. Passing of any other class will raise a custom exception.

Because this Class Reference class will be used for calling the constructor of any class that we choose, we have to be able to pass whatever parameters would normally belong in the constructor of that class. You will see later how we can create a specialised subclass of Class whose Create method takes a determinate parameter list, but for now we need to understand how we can call the constructor of an, as yet unknown, class.

The .NET framework contains a class called System.Reflection.ConstructorInfo that allows us to get at the metadata of a constructor and call it in runtime code.

```
public object Create(object[] args) {...}
{
    Type[] lArgTypes = new Type[args.Length];
```

The first thing that our Create method has to do is to find out what types are being passed in as parameters to our method. To do this we have to declare a local variable of array of Type that is large enough to take one Type for each of the parameters.

```
for (int i = 0; i < args.Length; i++)
    lArgTypes[i] = args[i].GetType();
```

Then we need to examine each of the parameters and assign their type to one of the local array's elements.

```
ConstructorInfo lConstructor =
    classType.GetConstructor(
BindingFlags.Instance | BindingFlags.NonPublic
| BindingFlags.Public, null, lArgTypes, null);
```

Having filled the lArgTypes array, we can then pass that array to the GetConstructor method of System.Type in order to obtain a ConstructorInfo object that describes the constructor for the class that we wish to instantiate. We use the BindingFlags enumeration to specify that we want any constructor for our type, if it is public or not, that takes a parameter list that matches those passed into this Create method.

```
if (lConstructor != null)
    return lConstructor.Invoke(args);
else
    throw new
        ConstructorNotFoundException();
}
```

If the call to GetConstructor is successful, the IConstructor variable will no longer be empty and we can then call the Invoke method of the ConstructorInfo object to obtain an instance of our required class.

Creating a Class Of Class

Trying to use the generic Class Reference class that we have just described in application code would be very clumsy because we would have to continually pass in arrays of objects instead of known parameter lists that are strictly typed.

What we need to do is to derive a class from Class that is specialised to the hierarchy whose virtual constructor we would like to call. But first, we will declare the class that is the base class of our example TShape hierarchy.

```
class Shape
{
    private Point origin;
    private Size dimensions;

    ...

    public Shape(Point origin, Size dimensions)
    {
        this.origin = origin;
        this.dimensions = dimensions;
    }
}
```

The important part of this class for the purposes of this article is the constructor. Shape has a constructor that takes a Point and a Size parameter and this is intended to be the signature that we want to use for our virtual constructor. So now, we need to derive from our generic Class Reference class in order to give ourselves a specialised version of the virtual Create method that returns an instance of Shape or any of its derivatives.

```
class ShapeClass : Class
{
    public ShapeClass() : base(typeof(Shape)) {}

    public Shape Create(Point origin, Size
        dimensions)
    {
        return (Shape) base.Create(new object[2]
            {origin, dimensions});
    }
}
```

The constructor of ShapeClass calls the constructor of Class passing the Type of the Shape class as a parameter. This will be used to obtain the correct constructor in the ConstructorInfo used in Class.Create.

The signature of the Create method should match that of the constructor of the base class in the hierarchy, in this case Shape. All this function does is to pass the parameters expected by the constructor signature to the array of object parameter of Class.Create so that that method can use them to find the correct constructor.

Having created a Shape Class Reference class, we can now go on to derive from Shape to give us two more classes for use in this example.

```
class Circle : Shape
{
    public Circle(Point origin, Size dimensions)
        : base(origin, dimensions) {}
}

class Square : Shape
{
    public Square(Point origin, Size dimensions)
        : base(origin, dimensions) {}
}
```

Using the Class Of Class

Using ShapeClass in C# is remarkably similar to using a Class Reference in Delphi. Here is a simple example that creates a 'class of Shape' and assigns both Circle and Square to that Class Reference in order to create instances of those classes by use of the Create 'virtual constructor'.

```
{
    ShapeClass sc = new ShapeClass();

    sc.ClassType = typeof(Circle);
    Shape s = sc.Create(new Point(0, 0), new
        Size(10, 10));

    sc.ClassType = typeof(Square);
    s = sc.Create(new Point(10, 10), new
        Size(25, 25));
}
```

If we just repeat the Delphi code here, you will see just how similar the two languages can become.

```
var
    ShapeClass: TShapeClass;
    Shape: TShape;
begin
    ShapeClass := TCircle;
    Shape := ShapeClass.Create(Point(0, 0),
        Point(25, 25));
end;
```

In my next article I will look at Virtual Class Methods, using a similar technique to circumvent their lack in C#.

Conclusion

The aim of this article has been to allow programmers who are now using C# to have a mechanism similar to the Class Reference types and Virtual Constructors that are a part of the Delphi language.

Use of a Class Reference type greatly simplifies code in that it avoids the need for the Abstract Factories that would otherwise have been needed.

C# constructors have different names because they need to match the class name; this prevents virtual constructors from working.

Delphi constructors use a 'single name' constructor named Create; this allows that name to be overridden in subclasses, thus allowing virtual constructors.

We created a Class Reference class that contained a Create method that took an array of objects and called that from a derived class (ShapeClass) that took a determinate parameter list that reflected the constructor signature we wanted to use as a virtual constructor (for Shape).

We discovered that the code for dealing with Class References in C# is remarkably similar to that used in Delphi.

Joanna Carter is a writer, developer and trainer specialising in requirements-driven training and consultancy.



Apart from writing articles, Joanna is engaged in the writing of a book on Design Patterns for Delphi, but she is still available if you need someone to help you with your projects; whether that be analysis, design or even coding. You can e-mail her at [joannac@ carterconsulting.org.uk](mailto:joannac@carterconsulting.org.uk) and her web site can be found at carterconsulting.org.uk