

## ***Building .NET Components***

**Component** : an object that is reusable and can interact with other objects.

### **.NET Framework component:**

“additionally provides features such as control over external resources and design-time support.” [Visual Studio Help].

As a good software engineer, you want to decompose your applications into small functional units with well-defined interfaces to each other. Hopefully these modules can be re-used in various applications and can be more easily maintained as components than as parts of a huge application.

In the Windows API and in MFC, components are implemented as DLLs (dynamic link libraries). In FCL, that is also true: when we choose as the project type *Class Library*, and build the project, a DLL file is produced. But the FCL library and Visual Studio provide some support to make the creation and use of class libraries easy. In MFC, there were quite a few technical difficulties if you wanted to include a class in a DLL and export its methods. (“Export” here means “make available for other components to use”.)

In the .NET Framework, a *component* is a class that implements the *System.ComponentModel.IComponent* interface or that derives directly or indirectly from a class that implements **IComponent**.

That is mysterious since we don’t know what the interface *IComponent* is. But we don’t need to know that. We only need to know that there are two important FCL classes that implement that interface: *Component* and *Control*. Corresponding to these choices, there are two “project types” available in Visual Studio:

- *Windows Control Library*: for a control that has a user interface; you will use the Form Editor to design this interface.
- *Class Library*: exposes functionality by offering classes and methods only, without needing a designable user interface.

## Hosting

Once you have created a component, you will want to use that component in some program. The program that uses a component is said to *host* the component. The component is said to *be sited in* the host program. The *IComponent* interface includes a property *Site*, which is a class through which the component interacts with its site, for example to query its container. There is also an event *Disposed* in the interface; when the host is destroyed its *Dispose* method is called, and that method should call the *Dispose* method of each of the contained components. (C# garbage collection has no way of knowing what the contained components are.)

A component may or may not have a visual representation. Of course, a control will have a visual representation, but a component might perform work invisibly, for example, accepting a query, contacting a database, receiving the answer from the data base and returning that answer as its return value. The host would be responsible for passing it the query and doing something with the answer, perhaps passing it to a control component for display.

## Design-Time Support

A component (not only a control) can be added to the toolbox of Visual Studio.NET, and then dragged and dropped onto a form. Support for this design-time feature is built into the *Component* and *Control* classes in FCL.

## Remotable Components

In the past, DLLs had to run on the same computer as the programs that called them. But .NET components can be *remotable*, which means they can send function parameters to a different machine over a network. There are two ways of accomplishing this:

- *Marshaling by reference.* A proxy (local function) is created that makes remote calls to the object. The parameters to those calls are serialized and sent through the network.
- *Marshaling by value.* A serialized copy of the object itself is sent. [“serialized” means, converted to a stream of bytes, as one has to do when saving a file.]

Here is what the documentation says about marshaling by reference and by value:

Remotable components that encapsulate system resources, that are large, or that exist as single instances should be marshaled by reference. The base class for components that are marshaled by reference is `System.ComponentModel.Component`. This base class implements **IComponent** and derives from **MarshalByRefObject**. Many components in the .NET Framework class library derive from **Component**, including **System.Windows.Forms.Control** (the base class for Windows Forms controls), **System.Web.Services.WebService** (the base class for XML Web services created using ASP.NET), and **System.Timers.Timer** (a class that generates recurring events).

Remotable components that simply hold state should be marshaled by value. The base class for components that are marshaled by value is `System.ComponentModel.MarshalByValueComponent`. This base class implements **IComponent** and derives from **Object**. Only a few components in the .NET Framework class library derive from **MarshalByValueComponent**. All such components are in the **System.Data** namespace (**DataColumn**, **DataSet**, **DataTable**, **DataView**, and **DataViewManager**).

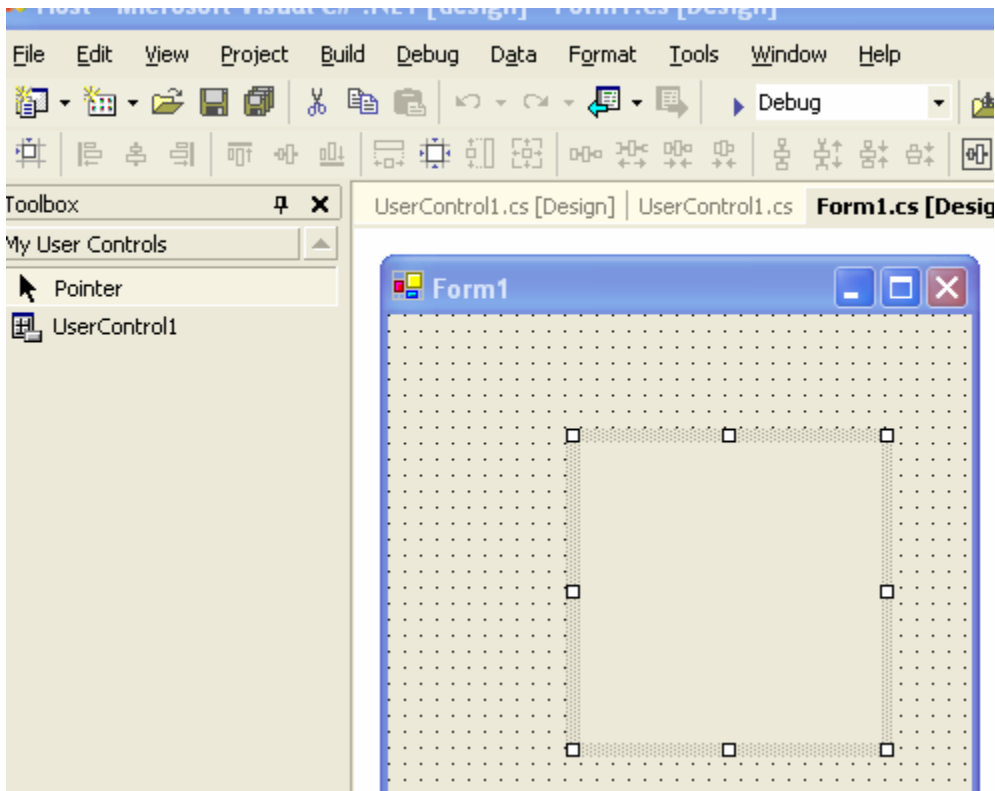
*Example.* Let's say that we possess code to display mathematical formulas presented as a String. We wish we had a *Formula* control that we could drag and drop onto a Windows form, with a property that would hold the string form of the formula, for example  $(x^3 + 1)/(x^2 - 1)$ , which would be displayed as

$$\frac{x^3 + 1}{x^2 - 1}$$

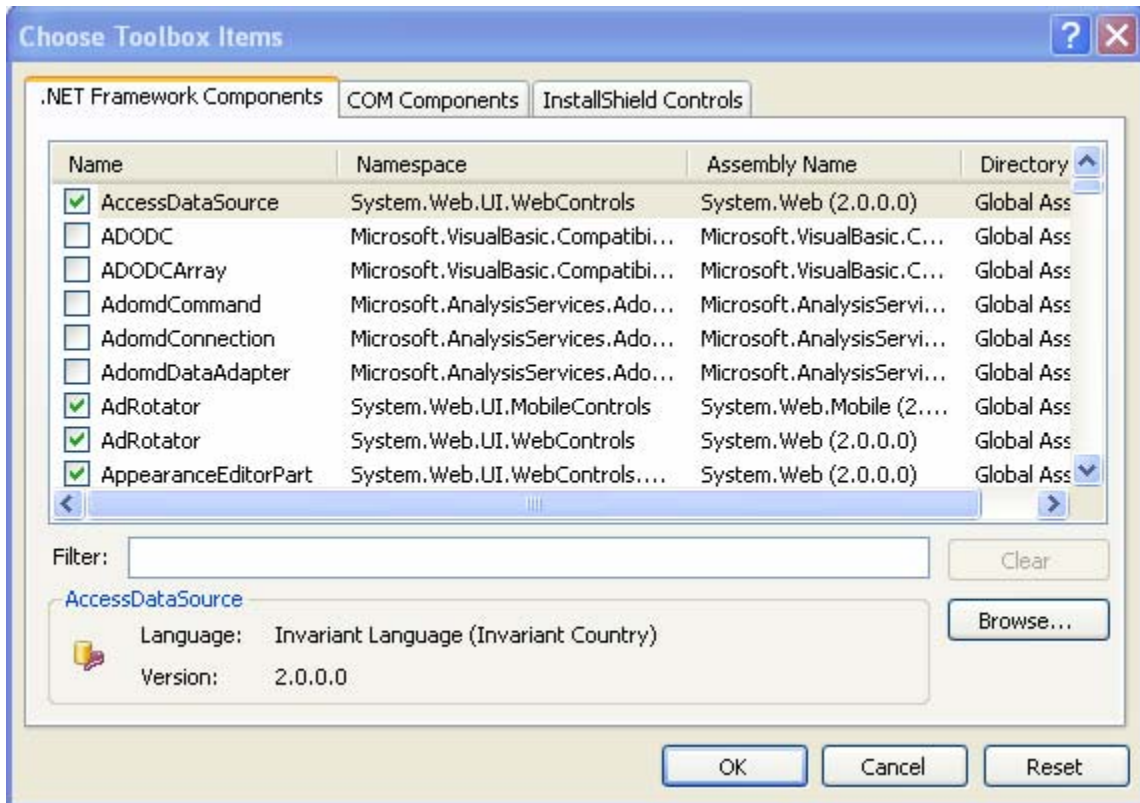
We will show how this is done, but skipping the complicated display code, we'll just have the control call *DrawString* to display the string form. This will still demonstrate how to create and use a custom control. Here are the steps to follow:

1. Create a new project, choosing *Windows Control Library*. Call your project, for example, *ControlDemo*. You'll see a form on the Form Designer, smaller than you see for an application.
2. Add a member variable `String formulaString;` to do that, just type in the declaration in the .cs file.

3. In the Form Designer, right-click your form and add a handler for the *Paint* event. In your *Paint* method, create a *Font* and *SolidBrush* and call *DrawString* to display *formulaString*.
4. Now create a second project, choosing *Windows Application*. Be sure to select the radio button to *add to solution*, rather than create a new solution. This will be the host application. Observe in Solution Explorer that you have a reference to *ControlDemo*.
5. The documentation alleges, and in Visual Studio 2003 it was true, that with the Form Designer open on your new project, you should see a tab at the left labeled *My User Controls*, and under that tab you should see your new control listed. Here is a screen shot from Visual Studio 2003, but *I couldn't get this to happen in Visual Studio 2005!*



6. Eventually, I figured out how to get my control to appear in the toolbox: Right-click the Toolbox in some blank place, and choose *Choose Items* from the context menu. That will bring up this dialog:



Click “Browse” and browse to the folder of your control project, then to its *Bin/Debug* subfolder, and select the .dll file containing the executable code. (Of course, if the control is already debugged and you’re just using it, then you would use the release version.) Now you should see *userControll1* (or whatever you named your control) in the Toolbox, and you can drag one onto your *Form1*.

6. In the constructor of your host project, add this line after the call to *InitializeComponents*:

```
userControll1.stringFormula = "(x^3+1)/(x^2-1)";
```

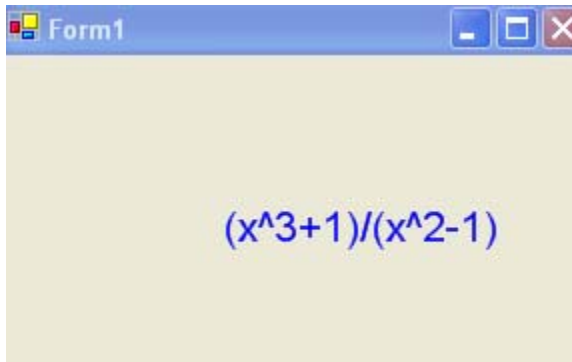
Incidentally, observe that Visual Studio wrote this code for you:

```
private void InitializeComponent()
{
    this.userControll1 = new ControlDemo.UserControll1();
    this.SuspendLayout();
}
```

It seems to work fine to add the initialization of *stringFormula* in *InitializeComponent*, but I’m wary of editing automatically generated code,

as sometimes what you write there disappears upon a later use of the design editor, so I think it is better to put your code in the constructor after the call to *InitializeComponent*.

7. Right-click your host project in Solution Explorer and choose *Set as Active Project*. Then run it. You should see the result



If you don't set the host project as active project first, you won't be able to run, because a component can't be the startup project.

### ***Final Remarks***

To make this a useful component, of course, we would have to write or import the code for the two-dimensional display of the formula. In practice that is a problem, because the code is in "unmanaged" C, not in "managed C++" or in C#. This problem is solvable, but not pretty.

Also, it would be nice if this component would work on a web page. I believe it would work on a web page if the web server is Microsoft's server (IIS), and if the user's browser is Internet Explorer, but it won't work otherwise. That makes it useless for web pages, in my opinion. But it still might be useful in building applications.

Although in theory you could "remote" this component, I wasn't able to accomplish it. *Add Web Reference* is only useful for using a web service.