Developing Applications using the NGWS SDK: An Introduction

Tutorial

Abstract

This tutorial shows developers how to create NGWS applications and components using the PDC Tech Preview of the NGWS SDK and Visual Studio 7.0. The development tools in this next major release of Visual Studio will utilize the NGWS framework to allow developers to quickly build and deploy robust applications that take advantage of the new NGWS runtime environment. Using Visual Studio 7.0 tools with the NGWS SDK provides:

- A fully managed, protected, and feature rich application execution environment
- Application integration with Active Server Pages (ASP)
- Improved isolation of application components
- Simplified application deployment

This paper begins by demonstrating writing the classic “Hello World” program in two familiar languages –an update to C++ called “Managed Extensions for C++ (MC++)” and Visual Basic – as well as a new language named C# – designed specifically for the NGWS environment.

This simple program is then greatly expanded to show a small, componentized client/server application where both the client and server are written using each of the three languages. A fourth client program demonstrates calling these components from the new Windows-based WinForms forms library. The final client program calls these components from ASP+, the next generation of Active Server Pages for Web-based development. The steps necessary to construct, compile, and run each program are covered in detail. An appendix contains additional information on several useful developer utilities.

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

0200

CONTENTS

INTRODUCTION

This document was written to accompany the PDC Tech Preview of the NGWS SDK, and assumes you already have both the SDK and the associated PDC Tech Preview of Visual Studio 7.0. If you are working with later versions of either the SDK or Visual Studio, you will need to obtain an updated version of this document since many details of the underlying technology – particularly the names of the underlying objects and their members – are likely to have changed.

We will walk you through the process of developing several small programs that use the next generation of Visual Studio language tools to take advantage of the new NGWS framework and runtime. You will start with the simplest of programs – command-line versions of the traditional "Hello World" executable – in each of the three Visual Studio 7.0 languages:

- Managed Extensions for C++ (MC++)
- Visual Basic
- C#

These simple programs will introduce you to working with the new NGWS runtime as well as the process of developing for a managed environment where many common programming tasks – for instance, memory garbage collection (GC) and a rich class library – are already provided.

You will then be lead through the development of a small, componentized client/server application. This application shows a server DLL component – written in each of the three languages – being called from *five* different clients: Command-line applications for each of the same three languages, a Windows-enhanced client using the new WinForms library, and a Web server page utilizing a WebForm server control that demonstrates the new integration between Active Server Pages (ASP+), the NGWS framework and runtime, and the new language tools.

Prerequisites

To benefit most from this tutorial and accompanying samples, the reader should already be familiar with developing component-base applications using COM+ or Web-based applications using IIS and ASP or a similar framework. Throughout this document we will be introducing new terms related to the NGWS technology.

Finally, since the code samples are presented in C++ and Visual Basic (as well as the new language, C#), developers should already be familiar with at least one of these languages.

Tools required

In general, the NGWS SDK includes everything necessary to compile and test the samples that accompany this document. In particular, the SDK comes with command-line compilers for each of the languages, the runtime and associated files, and detailed documentation.

If you intend to run the ASP+ sample, you must install the NGWS SDK *after* installing IIS to recognize and properly handle .aspx files (this may require reinstalling the SDK).

COMMON CONCEPTS

All of the files needed to use the accompanying samples and start developing are installed with the SDK. The three language compilers are found in a directory under c:\Windows\ComPlus and the documentation (in the form of compiled HTML, or .chm files) is located in the Docs subdirectory of the SDK. The NGWS runtime files are located in the Windows System subdirectory. For the PDC Tech Preview of the NGWS SDK, the important files and corresponding versions are:

- **mscorlib.dll** – 2000.14.????
- **bc.exe** – 7.00.????
- **cl.exe** – 13.0.????
- **csc.exe** – 2000.14.????

Since source and project files for all three languages are plain text, any text editor – even Notepad – is adequate for examining and modifying the accompanying sample files. However, if you want to make substantial changes to the samples – for instance, creating a new application based on the sample – you will likely want to install the PDC Tech Preview of Visual Studio 7.0.

The NGWS SDK includes a variety of samples, several of which are useful to developers: These are described in Appendix A: “Exploring Namespaces”, in this document. To install and build these SDK samples, please refer to the instructions in StartSamples.htm in the \Samples subdirectory of the SDK.

The sample programs that accompany this tutorial - and other samples - are located in the self-extracting “samples.exe” file.

Finally, to compile and run the samples that accompany both the SDK and this tutorial, you must an environment variable correctly configured:

- **path** – Must include the subdirectories where the compilers are located.

Most aspects of programming with NGWS are the same for all compatible languages: each supported language compiler produces self-describing, *managed* intermediate language (IL) code. All managed IL code runs against the NGWS runtime, which provides cross-language integration, automatic memory management, cross-language exception handling, enhanced security, and a simplified model for component interaction.

NGWS also provides a common base class library organized into a single hierarchical tree of *namespaces*. At the root is the System namespace, which contains objects, including pre-defined types such as classes and interfaces, that can be used from any supported language. System objects – contained in mscorlib.dll– are used by all applications. The base class library also includes namespaces for both abstract base classes and derived class implementations for many other useful classes, including those for file IO, messaging, networking, and security. You can use these classes "as is" or derive your own classes from them.

Runtime-based class libraries are organized into hierarchical namespaces, and namespaces are stored in portable executable (PE) files – typically DLLs and EXEs. You can have several namespaces – including nested namespaces – in one PE file, and you can split a namespace across multiple PE files. One or more PE files are

combined together to create an *assembly*, which is a physical unit that can be deployed, versioned, and reused. The runtime uses assemblies to located and bind to the referenced types.

The most commonly used objects are relatively easy to locate: Objects in the System.* namespace are documented in the cpref.chm and the new window/form objects are documented in wfc.chm, both located (by default) in the \Docs subdirectory of the SDK. There are several other tools for working with the included namespaces, as well as any custom namespaces (see Appendix A: Exploring Namespaces)

Since all supported languages compile to the same IL and use the same runtime and associated base class library, programs in each of the supported languages appear similar. In fact, the runtime specifies a set of language features called the Common Language Specification (CLS), which includes the basic language features that languages must support for interoperability. For our “Hello World” sample programs, we only need to write to the Console to show that our program is executing properly. Therefore, we will be using WriteLine method of the Console class of the System namespace. When we start working with a componentized application in a later section we will see how to create a traditional Windows graphical application.

What Can Vary

The most significant difference between programming with the supported languages is, not surprisingly, their language syntax. Both C++ and Visual Basic have an established history and significant base of existing code, which were taken into account when they were updated for NGWS. C#, on the other hand, starts with a cleaner slate. Several advanced topics – such as the build process and defining namespaces – will be introduced with the corresponding sample code in the following sections, but here are some of the more obvious differences between the languages.

Case Sensitivity: C++ and C# are both case-sensitive, but VB is not. For a program to be CLS compliant, however, public members cannot differ only in their case. This restriction allows VB (and potentially other CLS-compliant languages) to both produce and use components created in other, case-sensitive, languages.

Referencing a Library: To use classes, which are located in namespaces, it is first necessary to obtain a reference to the assembly containing the desired namespace. All NGWS programs use – at a minimum – the System namespace, which is found in mscorlib.dll (typically located in the Windows System directory):

```
C++    #using <mscorlib.dll>
C#     (implicitly loaded)
VB     (implicitly loaded)
```

Note that MC++ uses a preprocessor directive (this may change before final release) and the file name of the DLL. C# and VB currently references the System assembly implicitly (this may also change), but for other assemblies it is necessary

to use the **/import** compile switch. Referenced libraries are generally located in the application directory or a subdirectory of the application. Libraries that are designed for use by many applications – for instance, tools provided by 3rd parties – are located in the *assembly cache* (currently %system%\assembly) and must follow specific guidelines. Application configuration files can provide additional options. For ASP+ applications, however, components should be located in "\Bin" subdirectory under the starting point for the application's virtual directory.

Importing a Namespace: Classes can be either referenced fully (e.g. System.IO.FileStream, similar to a fully qualified path name) or their namespace can be imported into the program, after which it is not necessary to fully qualify the contained class names. For convenient access to System objects, that namespace must be imported:

```
C++    using namespace System;
C#     using System;
VB     Imports System
```

Note that both MC++ and C# use a **using** statement, while VB uses **Imports**.

Referencing Object Members: Both VB and C# support the period as a scope resolution operator, which allows (for example) the syntax **Console.WriteLine** when referencing the **WriteLine** method of the **Console** object. C++ uses a double colon "::" as a resolution operator:

```
C++    Console::WriteLine("xxxxx");
C#     Console.WriteLine("xxxxx");
VB     Console.WriteLine "xxxxx"
```

Declaring Objects: In Managed C++ and C# (though not in VB), variables must be declared before they can be used. Objects are instantiated using the **new** keyword. The following are sample declaration / creation statements – declaring and creating an object of type **Comp**, in namespace **Lib**, with the name **myComp** – in each of the three languages:

```
C++    Lib::Comp* myComp = new Lib::Comp();
C#     Lib.Comp myComp = new Lib.Comp();
VB     Dim myComp As New Lib.Comp
```

Program Entry Point: Every executable program has to have an external entry point, where the application begins its execution. The syntax hasn't changed for Managed C++, but in C# and VB everything happens in a class:

```
C++    void main() {
       }
C#     class MainApp {
           public static void Main() {
           }
       }
VB     Public Module modmain
           Sub Main()
           End Sub
       End Module
```

Behind the scenes, however, the Managed C++ compiler encapsulate the entry

point in a class.

Defining a Namespace & Class: Each of the three languages supports the creation of custom namespaces as well as classes within those namespaces. All three languages handle this in code, though with slightly different syntax. For instance, note the “managed” class declaration for C++ and the fact that trailing semicolons are not needed for namespace and class declarations in C#:

```
C++    namespace CompVC {
          __gc class StringComponent {
          public:
             StringComponent() {
             }
          };
       };
C#     namespace CompCS {
          public class StringComponent {
             public StringComponent() {
             }
          }
       }
VB     Namespace CompVB
          Public Class StringComponent
             Public Sub New()
             End Sub
          End Class
       End Namespace
```

We will now, per tradition, build a very simple command line application: an executable that outputs the string “Hello World”. We will build three versions of this application, one in each of the Visual Studio 7.0 languages: Managed C++ (produced using Visual C++ 7.0), Visual Basic, and C#.

Installing the sample files that accompany this document places each of the three “Hello World” programs in a separate subdirectory - \vc, \cs, and \vb – below the \Samples\IntroDev\HelloWorld subdirectory. Each application uses a single source code file and a batch command-line build file. The Visual Basic version also uses a project file, though this requirement may change before final release of the product.

“Hello World” in Managed C++

Here is how “Hello World” looks in Managed C++:

Listing 1 “Hello World” in Managed C++(HelloVC.cpp)

```
#using <mscorlib.dll>

// Allow easy reference System namespace classes
using namespace System;

// Global function "main" is application's entry point
void main() {
    // Write text to the console
    Console::WriteLine(S"Hello World using Managed C++!");
}
```

Even though the entire program is only a few lines of code, there are several things worth noticing, beginning with:

`#using <mscorlib.dll>`

In Managed C++, #using is similar to the #import directive (which is used to incorporate information from a type library). Note that these are different than the #include directive, which is for incorporating source code rather than pre-built libraries. Also, to import the namespace into the program (to make it convenient to reference System objects without having to fully qualify their path), an additional statement is required:

`using System;`

Classes can be either referenced fully (e.g. System.IO.FileStream, similar to a fully qualified path name) or their namespace can be imported into the program, after which it is not necessary to fully qualify the contained class names. For convenient access to System objects, that namespace must be imported:

`void main() {…}`

Though, in our example, the entry point **`main`** takes no command line arguments, this can obviously be enhanced for non-trivial programs. Our entry point also doesn't return anything, though it's possible to modify the function to return a single 32-bit numeric value to be used as an exit code. The next line is:

`Console::WriteLine(S"Hello World using Managed C++!");`

The real meat of the program, this line outputs a string using the runtime Console type. The **`Console`** type can be used for both input and output of any string or numeric value using **`Read`**, **`ReadLine`**, **`Write`**, and **`WriteLine`** methods. As mentioned above in the section "What Can Vary", the double-colon in **`Console::WriteLine`** is required in C++ to indicate scope: It separates both a namespace from a class name as well as a class name from a static method. Finally, by specifying **`L`** in front of the string, we tell the compiler to make this a Unicode string. Note that we can omit the **`L`** and it will still compile, but this creates an ANSI string that will get converted at runtime into its Unicode equivalent (required by the runtime String class), which reduces performance. In general, the recommendation is to always use Unicode strings.

The file build.bat contains the single line necessary to build this program:

`cl.exe /com+ HelloVC.cpp /link /entry:main`

The first item of note is the **`/com+`** switch, which tells the compiler to create

managed code, as required by the NGWS runtime. Also important is the **/entry:main** switch to indicate the entry point: This isn't required for traditional C++ programs which go through a more complicated initialization. Running this build file generates the following output:

```
C:\…\HelloWorld\vc>cl.exe /com+ HelloVC.cpp /link /entry:main
Microsoft (R) 32-bit C/C++ Optimizing Compiler…
Copyright (C) Microsoft Corp 1984-2000. All rights reserved.

HelloVC.cpp
Microsoft (R) Incremental Linker Version…
Copyright (C) Microsoft Corp 1992-2000. All rights reserved.

/out:HelloVC.exe
/entry:main
HelloVC.obj
```

Finally, running the resulting executable yields:

```
C:\…\HelloWorld\vc>hellovc
Hello World using Managed C++!
```

"Hello World" in C#

Here is how "Hello World" looks in C#:

Listing 2 "Hello World" in C# (HelloCS.cs)

```
// Allow easy reference System namespace classes
using System;

// This "class" exists only to house entry-point
class MainApp {
    // Static method "Main" is application's entry point
    public static void Main() {
        // Write text to the console
        Console.WriteLine("Hello World using C#!");
    }
}
```

This code is a little longer than the equivalent for managed C++. The syntax for accessing the core library is new, where we specify the namespace rather than the name of the file in which it is found:

```
using System;
```

The most striking difference is the class specification:

```
class MainApp {…}
```

In C#, all code must be contained in methods of a class. So, to house our entry point code, we must first create a class (the name doesn't matter here). Next, we specify the entry point itself:

```
void Main () {…}
```

The compiler requires this to be called **Main**. The entry point must also be marked with both **public** and **static**. Also, as with the managed C++ example, our entry point takes no arguments and doesn't return anything (though different signatures

for more sophisticated programs are certainly possible). The next line is:

```
Console.WriteLine("Hello World using C#!");
```

Again, this line outputs a string using the runtime Console type. In C# however, we're able to use period to indicate scope and we don't have to place an **L** before the string (in C#, all strings are Unicode).

The file build.bat contains the single line necessary to build this program:

```
csc helloCS.cs
```

In this admittedly simple case, we don't have to specify anything other than the file to compile. In particular, C# doesn't use the additional link step required by C++:

```
C:\…\HelloWorld\cs>build
C:\…\HelloWorld\cs>csc hellocs.cs
Microsoft (R) C# Compiler Version …[URT version…]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

The default output of the C# compiler is an executable of the same name, and running this program generates the following output:

```
C:\…\HelloWorld\cs>hellocs
Hello World using C#!
```

"Hello World" in Visual Basic

Finally, here is how "Hello World" looks in Visual Basic:

Listing 3 "Hello World" in VB (HelloVB.vb)

```
' Allow easy reference System namespace classes
Imports System

' Module houses the application's entry point
Public Module modmain
   ' "Main" is application's entry point
   Sub Main()
      ' Write text to the console
      Console.WriteLine ("Hello World using Visual Basic!")
   End Sub
End Module
```

This code is almost the same as for C#. The syntax for accessing the core library is new, where – like with C# – we specify the namespace rather than the filename:

```
Imports System
```

Other than that, there's not much else to say. The output line is almost the same as for the other languages, especially now that VB requires parentheses around the method parameter. Of course, VB does not require using semi-colons to terminate statements:

```
Console.WriteLine "Hello World using Visual Basic!"
```

The command line for compiling the sample Hello World VB program is:

```
vbc HelloVB.vb /out:HelloVB.exe /t:exe
```

where **/o** specifies the output file and **/t** indicates the target type. Executing the sample batch file containing this command-line yields:

```
C:\…\HelloWorld\vb>build

C:\…\HelloWorld\vb>vbc HelloVB.vb /out:HelloVB.exe /t:exe
Microsoft (R) Visual Basic Compiler Version …[URT version …]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

And executing the resulting executable produces:

```
C:\…\HelloWorld\vb>hellovb
Hello World using Visual Basic!
```

We will now create a component – in each of our three languages – that can, in turn, be used by each of those languages (shown in a later section of this document). This simple component essentially provides a wrapper for an array of strings and includes a **`GetString`** method (taking an integer and returning a string) as well as a read-only **`Count`** property containing the number of elements, which is used for iterating over all the members. The **`GetString`** method also illustrates the use of structured exception handling. Though limited, this string component illustrates the basics of creating reusable classes.

A Component in Managed C++

Here is how our simple string component looks in managed C++:

Listing 4 Component in Managed C++(CompVC.cpp)

```
#using <mscorlib.dll>
using namespace System;

namespace CompVC {
    __gc public class StringComponent {
    private:
       String* StringsSet[];

    public:
       StringComponent() {
          StringsSet = new String*[4];
          StringsSet[0] = new String(S"VC String 0");
          StringsSet[1] = new String(S"VC String 1");
          StringsSet[2] = new String(S"VC String 2");
          StringsSet[3] = new String(S"VC String 3");
       }

       String* GetString(int index) {
          if ((index < 0) || (index >= StringsSet->Length)) {
             throw new IndexOutOfRangeException();
          }
          return StringsSet[index];
       }

       __property int get_Count() { return StringsSet->Length; }
    };

};
```

As mentioned above, we use the **namespace** statement to create a new namespace to encapsulate the classes we will be creating:

```
namespace CompVC {…};
```

Note that this namespace may be nested and may be split between multiple files: A single source code file may also contain multiple non-nested namespaces. Since our namespace can contain managed and unmanaged classes (unlike VB and C#, which only have managed classes), we need to specify that our **StringComponent** class is managed:

```
__gc public class StringComponent {…};
```

This statement means that instances of **StringComponent** will now be created by the runtime and managed in the Garbage Collected heap. We could also have used the **/com+** compiler switch to make all of the classes in the program managed.

The class constructor – which executes each time a new instance of the class is created – has the same name as the class and doesn't have a return type.

```
public:
    StringComponent() {…}
```

Also, since this is now a managed class, we need to explicitly tell the compiler that the array of Strings is a managed object. Hence the **__gc** modifier when allocating the string:

```
StringsSet = new String*[4];
```

Here's the **GetString** method, which takes an integer and returns a string:

```
String* GetString(int index) {…
    return StringsSet[index];
}
```

Note the throw statement in the **GetString** method, which highlights the runtime-based exception handling:

```
throw new IndexOutOfRangeException();
```

This statement creates – and throws – a new object of type **IndexOutOfRangeException**, which is caught by the caller. This mechanism replaces the hResult-based error handling system used in previous versions of COM. Note that, using NGWS exception handling, all exceptions – including those we define for our own use – must be derived from **System::Exception**.

Finally, we create a read-only property **Count**:

```
__property int get_Count { return StringsSet->Count; }
```

Building our new C++ component is a little more complicated:

```
cl /com+ /c CompVC.cpp
link -noentry -dll /out:..\Bin\CompVC.dll CompVC
```

As with the simple "Hello World" C++ example, we need the **/com+** switch to tell the compiler to create NGWS managed code. We've also broken the link process out into a separate command: Both statements include the **/assembly** switch. Assemblies are the physical units that can be deployed, versioned, and reused. Each assembly establishes a set of types – runtime-based classes and other resources – that are meant to work together as well as an assembly manifest, which indicates what components are part of the assembly, what types are exported from the assembly, and any dependencies. The runtime uses assemblies to locate and bind to the types you reference.

For convenience, the sample components for this document are maintained in a "..\Bin" subdirectory relative to the source code: To compile the component to that location, we simply specify the qualified filename using the **/out** parameter. We could also place the compiled components in the assembly cache if they were going to be used with other programs. And even though we specified an output file with a .dll file extension, we need the additional **-dll** switch to create a DLL rather than an executable.

A Component in C#

Here is how our simple string component looks in C#:

Listing 5 Component in C# (CompCS.cs)

```
using System;

namespace CompCS {
    public class StringComponent {
        private string[] StringsSet;

        public StringComponent() {
            StringsSet = new string[] {
                "C# String 0",
                "C# String 1",
                "C# String 2",
                "C# String 3"
            };
        }

        public string GetString(int index) {
            if ((index < 0) || (index >= Count)) {
                throw new IndexOutOfRangeException();
            }
            return StringsSet[index];
        }

        public int Count {
            get { return StringsSet.Count; }
        }
    }
}
```

As mentioned above, we use the **namespace** statement to create a new namespace to encapsulate the classes we will be creating:

namespace CompCS {…}

This namespace may be nested and may be split between multiple files: A single source code file may also contain multiple non-nested namespaces. A containing namespace is required, since all C# code must be contained in a class.

public class StringComponent {…}

This statement means that instances of **StringComponent** will now be created by the runtime and managed in the Garbage Collected heap. The class constructor – which executes each time a new instance of the class is created – has the same name as the class and doesn't have a return type.

public StringComponent() {…}

Since C# uses only managed types, we don't have to do anything special – as we had to do with managed C++ - when declaring variables.

Here's the **GetString** method, which takes an integer and returns a string:

```
public string GetString(int index) {
    …
    return StringsSet[index];
}
```

Note the **throw** statement in the **GetString** method:

```
throw new IndexOutOfRangeException();
```

This statement creates – and throws – a runtime-based exception handling object of type **IndexOutOfRangeException**.

Another way of returning strings would be to use an indexer instead of the **GetString** method:

```
public string this[int index] {
    if ((index < 0) || (index >= Count)) {
       throw new IndexOutOfRangeException();
    }
            return StringsSet[index];
}
```

An indexer would allow the client to use a much more natural syntax of:

```
StringSetVar[index]
```

VB, however, doesn't support indexers so, in the interest of keeping the component code the space for each of the three languages, we have chosen to use a distinct **GetString** method.

Finally, we create the read-only property **Count**:

```
public int Count {
    get { return StringsSet.Count; }
}
```

The compile process for a C# component is only a little more complicated than for a stand-alone program:

```
csc /out:..\Bin\CompCS.dll /target:library CompCS.cs
```

As with the C++ component, we use the **/out** switch to put the compiled component in the "..\Bin" relative subdirectory for convenience. Likewise, we need the **/target:library** switch to actually create a DLL rather than an executable with a .dll file extension.

A Component in VB

Here is the full source code listing for our sample string component in VB:

Listing 6 Component in VB (CompVB.vb)

```
Imports System

Option Explicit

Namespace CompVB

    Public Class StringComponent

        Private StringSet(4) As String

        Public Sub New()
            MyBase.New
            StringSet(0) = "VB String 0"
            StringSet(1) = "VB String 1"
            StringSet(2) = "VB String 2"
            StringSet(3) = "VB String 3"
        End Sub

        Public Function GetString(ByVal index as Integer) ↵
               As String
            If ((index < 0) or (index >= Count)) then
               throw new IndexOutOfRangeException
            End If
            GetString = StringSet(index)
        End Function

        ReadOnly Property Count() As Long
            Get
               Count = StringSet.Length
            End Get
        End Property

    End Class

End Namespace
```

Like with C++ and C#, both the namespace and the class name are specified in code (previous versions of VB used filenames to indicate class names).

In VB, class constructors are given the name **`New`** rather than the name of the class, as is done for the other languages. Since constructors don't return a value, using VB it is implemented as a **`Sub`** rather than a **`Function`**:

```
Public Sub New()
    …
End Sub
```

Also note the statement:

```
MyBase.New
```

This statement, which is required, calls the constructor on the base class. In C++ and C#, the call to the base class constructor is generated automatically by the compiler.

Here's the **`GetString`** method (in VB, subroutines which return values are called

CLIENTS FOR THE SIMPLE COMPONENTS

functions), which takes an integer and returns a string:

```
Public Function GetString(ByVal index as Integer) As String
    If ((index < 0) or (index >= Count)) then
        throw new IndexOutOfRangeException
    End If
    GetString = StringSet(index)
End Function
```

Note the **throw** statement in the **GetString** method, which highlights the new runtime-based exception handling:

```
throw new IndexOutOfRangeException
```

This statement creates – and throws – a new object of type **IndexOutOfRangeException**. Previous versions of the VB runtime implemented an **Err** object.

Finally, we create the read-only property **Count**:

```
ReadOnly Property Count() As Long
    Get
        Count = StringSet.Length
    End Get
End Property
```

The command-line build is quite simple, the only change being to output the component to the relative "..\Bin" subdirectory for convenience:

```
vbc CompVB.vb /out:..\Bin\CompVB.dll /t:library
```

We will now use the components – written in each of our three NGWS runtime compatible languages – that we created in the previous section. We need not define additional namespaces within the clients: instead, we will be importing and using classes defined by the component namespaces. We will also use the new WinForms library to create a true Windows application, rather than the console applications shown so far. Finally, we will show client code for an ASP+ page that places the output in an HTML HTTP response.

A Client in Managed C++

Here is how our client looks in Managed C++:

Listing 7 Client in Managed C++(ClientVC.cpp)

```
#using <mscorlib.dll>
using namespace System;

#using "..\Bin\CompCS.dll"
#using "..\Bin\CompVC.dll"
#using "..\Bin\CompVB.dll"

// method "Main" is application's entry point
void main() {

    // Iterate over component's strings and dump to console
    CompCS::StringComponent* myCSStringComp = ↵
     new CompCS::StringComponent();
    Console::WriteLine ↵
     (S"Strings from C# StringComponent");
    for (int index = 0; index < myCSStringComp->Count; ↵
     index++) {
       Console::WriteLine(myCSStringComp-> ↵
        GetString(index));
    }

    // Iterate over component's strings and dump to console
    CompVC::StringComponent* myVCStringComp = ↵
      new CompVC::StringComponent();
    Console::WriteLine ↵
     (S"\nStrings from VC StringComponent");
    for (int index = 0; index < myVCStringComp->Count; ↵
     index++) {
       Console::WriteLine(myVCStringComp-> ↵
        GetString(index));
    }

    // Iterate over component's strings and dump to console
    CompVB::StringComponent* myVBStringComp = ↵
     new CompVB::StringComponent();
    Console::WriteLine(S"\nStrings from VB StringComponent");
    for (int index = 0; index < myVBStringComp->Count; ↵
     index++) {
       Console::WriteLine(myVBStringComp-> ↵
        GetString(index));
    }
}
```

The first thing to note is the importing of the three components, all of which are now located in the "..\Bin" relative subdirectory:

```
#using "..\Bin\CompCS.dll"
#using "..\Bin\CompVC.dll"
#using "..\Bin\CompVB.dll"
```

The client code that calls the three string components is identical except for specifying which library to use. The first statement in each of the three sections declares a new local variable of a type **`StringComponent`** (defined in the component), initializes it, and calls its constructor:

```
CompCS::StringComponent* myCSStringComp = ↵
    new CompCS::StringComponent();
```

After writing out a string to the **Console** to say we're entering this part of the program, the client then iterates over the members – up to the value of the **Count** property – of appropriate string component:

```
for (int index = 0; index < myCSStringComp->Count; ↵
    index++) {
    Console::WriteLine(myCSStringComp-> ↵
    GetString(index));
}
```

That's all that's required, and everything's repeated for the other two language components. If we had used the indexer approach rather than the separate **GetString** method, the calling code would have been a more natural:

```
myCSStringComp[index]
```

Building our new C++ client is straightforward:

```
cl.exe /com+ ClientVC.cpp /link /entry:main ↵
    /out:..\bin\ClientVC.exe
```

As with the previous C++ examples, we need the **/com+** switch to tell the compiler to create NGWS runtime managed code. Running the resulting program yields:

```
C:\…\CompTest\Bin>clientvc
Strings from C# StringComponent
C# String 0
C# String 1
C# String 2
C# String 3

Strings from VC StringComponent
VC String 0
VC String 1
VC String 2
VC String 3

Strings from VB StringComponent
VB String 0
VB String 1
VB String 2
VB String 3
```

A Client in C#

Here is how our client looks in C#:

Listing 8 Client in C# (ClientCS.cs)

```
using System;

using CompVC;
using CompCS;
using CompVB;

// This "class" exists to house the application's entry-point
class MainApp {
  // Static method "Main" is application's entry point
  public static void Main() {

    // Iterate over component's strings dump to console
    CompCS.StringComponent myCSStringComp = new ↵
        CompCS.StringComponent();
    Console.WriteLine("Strings from C# StringComponent");
    for (int index = 0; index < myCSStringComp.Count; ↵
        index++) {
      Console.WriteLine(myCSStringComp.GetString(index));
    }

    // Iterate over component's strings dump to console
    CompVC.StringComponent myVCStringComp = new ↵
        CompVC.StringComponent();
    Console.WriteLine("\nStrings from VC StringComponent");
      for (int index = 0; index < myVCStringComp.Count; ↵
          index++) {
        Console.WriteLine(myVCStringComp.GetString(index));
      }

    // Iterate over component's strings dump to console
    CompVB.StringComponent myVBStringComp = new ↵
        CompVB.StringComponent();
    Console.WriteLine("\nStrings from VB StringComponent");
    for (int index = 0; index < myVBStringComp.Count; ↵
        index++) {
      Console.WriteLine(myVBStringComp.GetString(index));
    }
  }
}
```

Unlike the Managed C++ example, we don't have to import the libraries at this point: Instead, we can specify them in the compile process. The advantage of specifying a library with the **`using`** statement is that, by incorporating the namespace into the program, we can then reference types in the library without fully qualifying the type name. Since our particular example has the same type name (**`StringComponent`**) in each of the components, we still have use the fully qualified name to remove any ambiguity when referring to the method (**`GetString`**) and property (**`Count`**), which are common to each component. C# also provides a mechanism called aliasing to address this problem: If you changed the **`using`** statements to:

```
using VCStringComp = CompVC.StringComponent;
using CSStringComp = CompCS.StringComponent;
using VBStringComp = CompVB.StringComponent;
```

then you would not have to fully qualify the names.

The client code is virtually identical to the C++ example except for the scope resolution operators. Again, the code that calls the three string components is also the same except for specifying which library to use. As with the managed C++ example, the first statement in each of the three sections declares a new local variable of a type **StringComponent**, initializes it, and calls its constructor::

```
CompCS.StringComponent myCSStringComp = new ↵
    CompCS.StringComponent();
```

After writing out a string to the console to say we're entering this part of the program, the client then iterates over the members – up to the value of the **Count** property – of appropriate string component:

```
for (int index = 0; index < myVCStringComp.Count; ↵
     index++) {
   Console.WriteLine(myVCStringComp.GetString(index));
}
```

That's all that's required, and everything's repeated for the other two language components.

Building our new C# client is straightforward. Now that we're using components in our own relative "..\Bin" subdirectory, we need to explicitly include them using the **/i** compilation switch:

```
csc /i:..\Bin\CompCS.dll;..\Bin\CompVB.dll; ↵
    ..\Bin\CompVC.dll /out:..\Bin\ClientCS.exe ↵
    ClientCS.cs
```

Also, since we're building a client program rather than a component that might be called from other programs, it is not necessary to use the **/cls+** switch.

Other than that, the process is the same as with the previous C# examples. Running the resulting program yields:

```
C:\Com20Dev\CompTest\Bin>clientcs
Strings from C# StringComponent
C# String 0
C# String 1
C# String 2
C# String 3

Strings from VC StringComponent
VC String 0
VC String 1
VC String 2
VC String 3

Strings from VB StringComponent
VB String 0
VB String 1
VB String 2
VB String 3
```

A Client in VB

here's the full source code listing for our client in VB:

Listing 9 Client in Managed VB (Component1.cls)

```
Imports System
Imports System.Collections

Imports CompCS
Imports CompVB
Imports CompVC

Option Explicit

Public Module modmain

    'The main entry point for the application
    Sub Main()

        Dim Count As Integer

        ' Display result strings from C# Component
        Dim MyCompCS As New CompCS.StringComponent
        Console.WriteLine("Strings from C# StringComponent")
        For Count = 0 To MyCompCS.Count - 1
            Console.WriteLine(MyCompCS.GetString(Count))
        Next
        Console.WriteLine

        ' Display result strings from Visual C++ Component
        Dim MyCompVC As New CompVC.StringComponent
        Console.WriteLine("Strings from VC StringComponent")
        For Count = 0 To MyCompVC.Count - 1
            Console.WriteLine(MyCompVC.GetString(Count))
        Next
        Console.WriteLine

        ' Display result strings from Visual Basic Component
        Dim MyCompVB As New CompVB.StringComponent
        Console.WriteLine("Strings from VB StringComponent")
        For Count = 0 To CInt(MyCompVB.Count) - 1
            Console.WriteLine(MyCompVB.GetString(Count))
        Next

    End Sub

End Module
```

Like the C# example's **`using`** statement, we specify the libraries with the **`Imports`** statement that then incorporate the namespace into the program so we can reference types in the library without fully qualifying their type names. Since our particular example has the same type name (**`StringComponent`**) in each of the components, we still have to use the fully qualified name to remove any ambiguity.

The client code is virtually identical to the C++ and C# examples except for the minor things such as the scope resolution operators and absence of a line termination character. Again, the code that calls the three string components is also

the same except for specifying which library to use. As with the MC++ and C# examples, the first statement in each of the three sections declares a new local variable of a type **`StringComponent`**, initializes it, and calls its constructor:

```
Dim MyCompCS As New CompCS.StringComponent
```

After writing out a string to the console to say we're entering this part of the program, the client then iterates over the members – up to the value of the **`Count`** property – of appropriate string component:

```
For Count = 0 To MyCompVC.Count - 1
  Console.WriteLine(MyCompVC.GetString(Count))
Next
```

That's all that's required, and everything's repeated for the other two language components.

The command-line build is quite simple, the only change being to output the component to the relative "..\Bin" subdirectory:

```
vbc ClientVB.vb /reference:..\Bin\CompCS.dll /reference:..
   \Bin\CompVB.dll /reference:..\Bin\CompVC.dll /out:..
   \bin\ClientVB.exe /t:exe
```

A Windows Client using WinForms

All of the examples so far have been command-line programs that wrote to the system console. Now that you have seen the entire development process, let's rewrite our client application to use the new Windows-based WinForms library, which is also available to all NGWS runtime compatible languages. In our example, we'll use Visual Basic and here's the full source code listing:

Listing 10 WinForms Client in VB (Client.cls)

```
Imports System
Imports System.Collections
Imports System.WinForms

Imports CompCS
Imports CompVB
Imports CompVC

Option Explicit

Public Module modmain
    Public Const vbCrLf = CChar(13) & CChar(10)
    Public Class Client

        Inherits Form

        'Required by the WinForms Designer
        Private components As System.ComponentModel.Container
        Private Button2 As System.WinForms.Button
        Private Button1 As System.WinForms.Button
        Private Label1 As System.WinForms.Label

        Sub New()
            MyBase.New
            InitForm ' Required by WinForms Designer.
        End Sub

        'Client form overrides dispose to clean up component list.
        Overrides Public Sub Dispose()
            MyBase.Dispose
            components.Dispose
        End Sub

        'The main entry point for the application
        Shared Sub Main()
             Application.Run(New Client)
        End Sub

        'NOTE: This procedure required by the WinForms Designer
        'It can be modified using the WinForms Designer.
        'Do not modify it using the code editor.
        Private Sub InitForm()
             Me.components = New System.ComponentModel.Container
             Me.Button1 = New Button
             Me.Button2 = New Button
             Me.Label1 = New Label

             Button1.SetLocation(200, 248)
             Button1.TabIndex = 1
             Button1.Text = "&Close"
             Button1.SetSize(75, 23)
             Button1.AddOnClick(New EventHandler ↵
                (AddressOf Me.Button1_Click))
```

```
    Button2.SetLocation(120, 248)
    Button2.TabIndex = 2
    Button2.Text = "&Execute"
    Button2.SetSize(75, 23)
    Button2.AddOnClick(New EventHandler ↵
       (AddressOf Me.Button2_Click))

    Label1.SetLocation(8, 8)
    Label1.TabIndex = 0
    Label1.TabStop = False
    Label1.Text = ""
    Label1.SetSize(272, 232)

    Me.Text = "Client"

    Me.Controls.Add(Button2)
    Me.Controls.Add(Button1)
    Me.Controls.Add(Label1)

End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ↵
      ByVal e As System.EventArgs)
    Me.Close  ' End Application
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, ↵
      ByVal e As System.EventArgs)

    ' Local Variables
    Dim myCompCS As New CompCS.StringComponent
    Dim myCompVB As New CompVB.StringComponent
    Dim myCompVC As New CompVC.StringComponent

    Dim StringCount As Integer

    ' Clear Label
    Label1.Text = ""

    ' Display results from C# Component
    For StringCount = 0 To CInt(myCompCS.Count) - 1
      Label1.Text = Label1.Text & ↵
            MyCompCS.GetString(StringCount) & vbCrLf
    Next
    Label1.Text = Label1.Text '& vbCrLf

    ' Display results from Visual Basic Component
    For StringCount = 0 to CInt(MyCompVB.Count) - 1
        Label1.Text = Label1.Text & ↵
            myCompVB.GetString(StringCount) & vbCrLf
    Next
    Label1.Text = Label1.Text '& vbCrLf
```

```
            ' Display results from Visual C++ Component
            For StringCount = 0 To CInt(myCompVC.Count) - 1
                Label1.Text = Label1.Text & ↵
                    myCompVC.GetString(StringCount) & vbCrLf
            Next

        End Sub

    End Class

End Module
```

In the PDC Tech Preview of the NGWS SDK, the WinForms library is located in the System.WinForms namespace, in particular:

```
Imports System.WinForms
```

By importing the namespaces, we can then refer to an included type – like **`Button`** – without having to fully qualify the type name – like **`Microsoft.WFC.UI.Button`**.

This next interesting line of code illustrates inheritance, one of the most powerful features of the NGWS runtime:

```
Inherits Form
```

With this one statement, we specify that our Client class inherits all of the functionality in the **`Form`** class in the WinForms library. Language independence is an important aspect of the NGWS runtime's inheritance model: not only can we inherit from the runtime, we can inherit from classes written in any NGWS runtime compatible language.

Next we declare the object types that we'll be using on our form, such as:

```
Private Button1 As System.WinForms.Button
```

Now we're finally read to execute some code. Here's the constructor for the **`Client`** form, which creates an instance of the base class and then calls the **`InitForm`** method:

```
Sub New()
    MyBase.New
    InitForm ' Required by WinForms Designer.
End Sub
```

And here's the entry point for the program itself, which starts everything off by creating a new instance of the **`Client`** form:

```
Shared Sub Main()
     Application.Run(New Client)
End Sub
```

The **`InitForm`** method sets up the form and all of its controls. For **`Button1`**, for example, we create a new button from the Button type:

```
Me.Button1 = New Button
```

We then move it, set it's caption (or **`Text`** property), and then resize it:

```
Button1.SetLocation(200, 248)
Button1.TabIndex = 1
Button1.Text = "&Close"
Button1.SetSize(75, 23)
```

Then comes the tricky part: hooking up **Click**, just one of the **Button** type's many events, to our own subroutine:

```
Button1.AddOnClick(New EventHandler ↵
    (AddressOf Me.Button1_Click))
```

Finally, we add the button to the form's Controls collection:

```
    Me.Controls.Add Button1
```

The following code highlights the event subroutine that executes when the user clicks on Button1:

```
Private Sub Button1_Click(ByVal sender As System.Object, ↵
          ByVal e As System.EventArgs)
     Me.Close   ' End Application
End Sub
```

Actually, the only thing that happens here is that the form's **Close** method is called, thus ending the application. In this particular subroutine we are ignoring the arguments.

The real meat of the program, which uses the same code we saw in the VB client example, is located in the **Button2_Click** event. Instead of writing to the Console, however, the WinForms sample adds to the **Text** property of the label on the form:

```
Label1.Text = Label1.Text & ↵
   myCompVC.GetString(StringCount) & vbCrLf
```

Running the application creates the following dialog. When the "Execute" button is clicked, the strings are written to the label on the surface of the form:

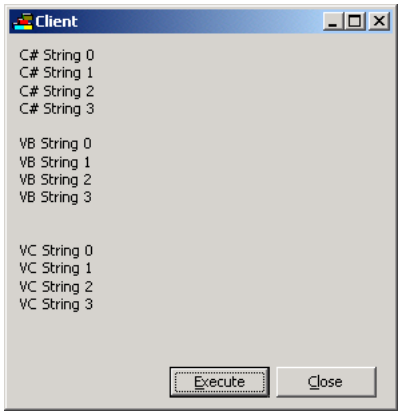

Figure WinForms Client in VB

A Client using ASP+

One of the most important features of NGWS is the ability to execute the same code in IIS's Active Server Pages (ASP+) as was used in our stand-alone client applications. Of course, there are a few differences: The ASP+ page generally produces HTML in response to an HTTP request, and the page itself is compiled

dynamically, unlike what was done for the previous examples. Each ASP+ page is individually parsed and the syntax is checked. Finally, a NGWS runtime class is produced, which is compiled and then invoked. ASP+ caches the compiled object, so subsequent requests don't go through the parse/compile step and thus execute much faster.

Most existing ASP pages use JScript or VBScript: We have chosen to show the page using C#, which would be a natural evolution for JScript code (VBScript code would upgrade quite naturally to VB):

Listing 11 Client in ASP+ (ClientASP+.aspx)

```
<%@ Page Language="C#" Description="ASP+ Component Test" %>
<%@ Import Namespace="CompCS"%>
<%@ Import Namespace="CompVC"%>
<%@ Import Namespace="CompVB"%>

<html>
<script language="C#" runat=server>
void Page_Load(Object sender, EventArgs EvArgs) {
    String Out = "";
    Int32 Count = 0;

    // Iterate over component's strings and concatenate
    Out = Out + "Strings from C# StringComponent<br>";
    CompCS.StringComponent myCSStringComp = new ↵
       CompCS.StringComponent();
    for (int index = 0; index < myCSStringComp.Count; index++) {
       Out = Out + myCSStringComp.GetString(index) + "<br>";
    }
    Out = Out + "<br>";

    // Iterate over component's strings and concatenate
    Out = Out + "Strings from VC StringComponent<br>";
    CompVC.StringComponent myVCStringComp = new ↵
       CompVC.StringComponent();
    for (int index = 0; index < myVCStringComp.Count; index++) {
       Out = Out + myVCStringComp.GetString(index) + "<br>";
    }
    Out = Out + "<br>";

    // Iterate over component's strings and concatenate
    Out = Out + "Strings from VB StringComponent<br>";
    CompVB.StringComponent myVBStringComp = new ↵
       CompVB.StringComponent();
    for (int index = 0; index < myVBStringComp.Count; index++) {
       Out = Out + myVBStringComp.GetString(index) + "<br>";
    }

    Message.InnerHtml = Out;
}
</script>
<body>
   <span id="Message" runat=server/>
</body>
</html>
```

This is essentially the same code as for the stand-alone client examples, except that we are building a string (named **out**) that we are then assigning to a property of an HTML server control. We could also have used the familiar **Response.Write** to write the string directly into the HTML output stream.

The page specifies C# as a language:

```
<%@ Page Language="C#" Description="ASP+ Component Test"
```

but we could just as easily have used VB or even JScript.

Importing libraries in ASP+ is also a little different:

```
<%@ Import Namespace="CompVB"%>
<%@ Import Namespace="CompCS"%>
<%@ Import Namespace="CompVC"%>
```

In these examples, delimited by "<%…%>" to indicate script code, we are specifying both the namespace *and* the physical assembly. As mentioned above, the assemblies must be located "\Bin" subdirectory of the application's starting point.

Another subtlety of this page is this line:

```
<script language="C#" runat="server">
```

This tells the server to execute the code on the server rather than sending the code text back to the client as part of the HTML stream.

ASP+ WebForms provides special recognition of six methods named:

- Page_Init
- Page_Load
- Page_DataBind
- Page_PreRender
- Page_Dispose
- Page_Error

These methods are automatically connected to event handlers for the standard page events. The most commonly used event is Load, which contains most of the code for our sample program:

```
void Page_Load(Object sender, EventArgs EvArgs) {…}
```

The rest of the code is pretty straightforward: We're just concatenating together a longer string `out` that we then add to the HTML with this statement:

```
StringOut.innerHTML = Out
```

Testing this page requires a few steps. First, it's necessary for the test machine to have the following installed:

- Internet Information Services (IIS)
- NGWS runtime
- ClientASP+.aspx
- CompVC.dll, CompCS.dll, and CompVB.dll compiled components

Installing the PDC Tech Preview of the NGWS SDK on a machine that already has IIS installed will enable that machine to run ASP+ (if you install IIS after installing the SDK, you should reinstall the SDK). Next, you must configure a virtual directory (using Internet Services Manager) that points to the directory where ClientASP.aspx is located. To create a virtual directory using the Internet Information Services snap-in:

- Select the Web site or FTP site to which you want to add a directory.
- Click the Action button, and then point to New, and select Virtual Directory.
- Use the New Virtual Directory Wizard to complete this task.

SUMMARY

If you are using NTFS, you can also create a virtual directory by right-clicking a directory in Windows Explorer, clicking Sharing, and then selecting the Web Sharing property sheet. For more information, see the topic “Creating Virtual Directories” in the IIS documentation, which is located at http://localhost/iisHelp/ on the machine on which IIS is installed.

Third, the compiled component dlls should be located in “\Bin” subdirectory under the starting point for the application virtual directory.

Assuming everything is configured correctly, executing the file with, for instance, the following URL:

`http://localhost/Com20Dev/ClientASP+.aspx`

should result in an instance of Internet Explorer with a display similar to this:

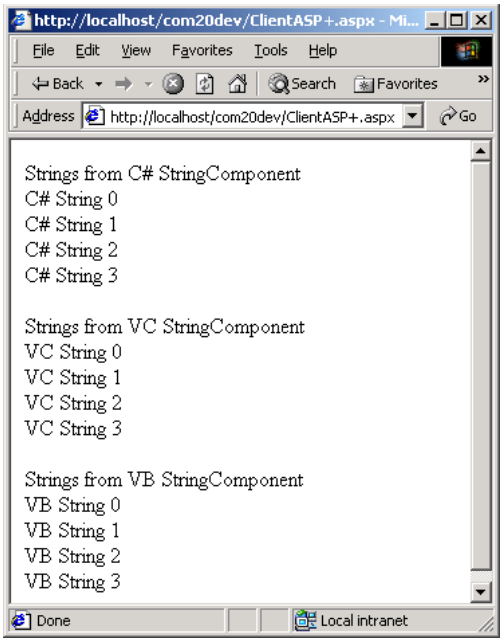

Figure ASP+ Client in C#

This tutorial introduced several sample programs that described the process of developing NGWS applications. You learned how several different languages – in our examples managed C++, C#, and Visual Basic – could be used to both create and call components written in any NGWS runtime compatible language. You also learned how to use the new WinForms library to create true Windows applications. Finally, you saw how the same languages can be used inside ASP+ to provide substantially more power and flexibility than was previously available to Web developers using embedded scripting languages.

Developers creating server applications should also review the “ASP+ and Web Server Overview” topic area (under “Preliminary Developer Documentation” and then “Technologies” in that same help file). Advanced developers will also benefit from reading the overview material under the “NGWS runtime Overview” topic, which includes information on the Execution Engine (EE) and Virtual Object System

APPENDIX A: EXPLORING NAMESPACES

(VOS).

For the latest information on NGWS, visit the Technology Preview Web site at http://dapdweb.microsoft.com/ngws/sdk, which includes a pointer to all the NGWS newsgroups on Microsoft news server. Feedback – including reporting bugs – on the SDK can be submitted through http://dapdweb.microsoft.com/ngws/sdk/feedback.

ClassView

One of the samples that accompany the SDK itself (located in a self-extracting executable in, by default, the c:\Program Files\Com20SDK\Samples subdirectory) is a full-featured class viewer based on XML. To use this tool, you need to first extract the samples, and then run Default.aspx in the \ClsView subdirectory:

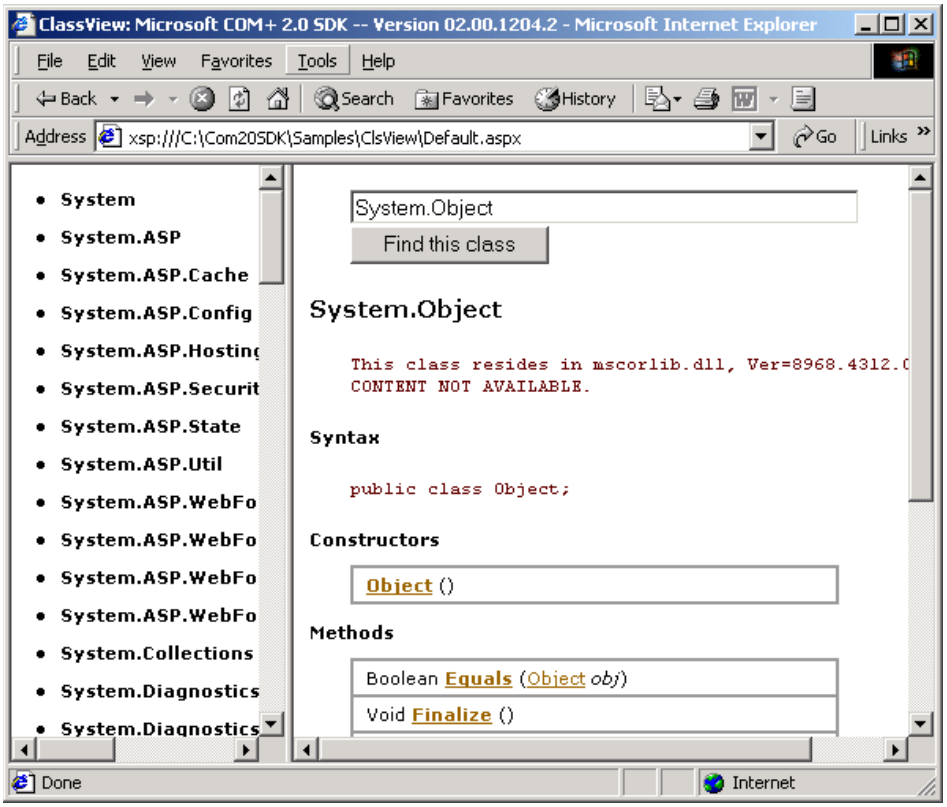

Figure ClassView Sample in NGWS SDK

FindType

Another sample that accompanies the SDK itself (again, located in a self-extracting executable in, also by default, the c:\Program Files\Com20SDK\Samples subdirectory) is command-line type locater that supports substring searches. To use this tool, you need to first extract the samples, build this sample using this command-line in the \FindType subdirectory:

```
NMAKE -F MAKEFILE
```

and then run FindType.exe. For example:

```
C:\Program Files\Com20SDK\Samples\FindType>findtype String
class   System.String
class   System.StringBuilder
…
class   System.IO.StringReader
class   System.IO.StringWriter
```

Intermediate Language Disassembler (ILDASM)

You can also explore the namespaces in files– either those that come with the runtime or those created by yourself or others - using the command-line IL disassembler tool to create a Windows output, for example:

```
C:\WINNT\ComPlus\v2000.14.1807>ildasm System.Net.dll
```

produces the following display:

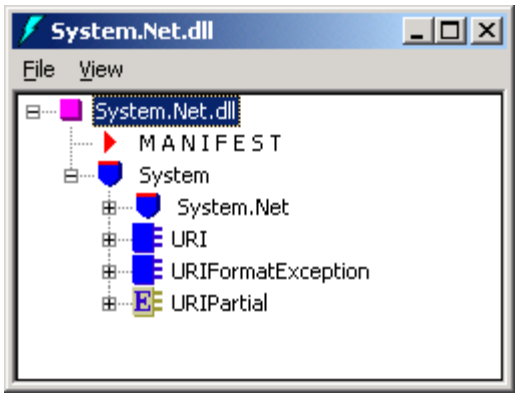

Figure IL Disassembler in NGWS SDK

Each of the namespace nodes represents a separate namespace, which can be expanded to explore the class objects and their methods and properties.