© 1996 VISUAL BASIC PROGRAMMER'S JOURNAL
FAWCETTE TECHNICAL PUBLICATIONS
ISSUE: AUGUST 1996

[Note that some of the material discussed in this article is replaced by Chapter 10 of Hardcore Visual Basic, second edition. The C++ and Basic samples discussed here are the same ones used in the book.]

The Future of ActiveX Development

Everything you know about control development is wrong. Maybe.

by Bruce McKinney

Six months ago there was only one way to develop controls for Visual Basic. These days most of us don't even know what to call them, much less how to create them. Microsoft alone offers three frameworks for control development with C++, and that's not to mention creating controls with Delphi, Java, or Visual Basic itself.

Internet opportunities have thrown Microsoft and indeed the whole software industry into confusion during the first half of 1996, but finally the smoke is beginning to clear. The ActiveX Template Library (ATL) now represents Microsoft's vision of how to develop the smallest, fastest ActiveX controls and other COM components.

Chances are you've never even heard of ATL, although preliminary versions of it have been available on the Internet for several months. You may be asking yourself why the company that brought you the Microsoft Foundation Classes is now proposing a new control development framework. Well, imagine MFC-based ActiveX controls on Internet pages. Visitors might have to download a megabyte or more of MFC support DLLs the first time they clicked on a page with an MFC-based control. Need I say more?

I won't even get into the MFC performance problems that inspired the Visual Basic team to develop a separate framework (which you can download from *http://www.microsoft.com/intdev/inttech/ctrlfmk.htm*). The next version of Visual Basic will use this framework in order to eliminate all MFC dependencies.

THE ACTIVEX TEMPLATE LIBRARY

The Visual C++ team saw the same problems, but decided to take a different approach. Most frameworks (including MFC) are based on libraries that are linked into client programs at either link time (static libraries) or run time (DLLs). ATL is based on templates that are expanded into code at compile time. You supply design-time type arguments that determine what code will be created. The advantage of a well-designed template library is that you get exactly the code you need and not much else. Table 1 shows how small an ATL component can be compared to other components.

Table 1

Component	Size
Basic EXE component	19K plus Basic support DLLs
Basic DLL component	10K plus Basic support DLLs
Statically-linked C++ MFC DLL component	160K
Dynamically-linked C++ MFC DLL component	16K plus MFC support DLLs
C++ ATL DLL component	16K

Table 1
Sieve performance. *The Basic EXE server SieveExe.Exe can also run stand-alone to get the timings for local classes. The client program SieveCli.Exe tests four COM components—SieveExe, SieveBas, SieveATL, and SieveMFC.*

ATL doesn't replace MFC. It just means that most people won't develop ActiveX components with MFC (although you still can if you want). MFC will focus on what it does best—making it easy to develop Win32-based applications. You can use MFC inside your ATL components, and you can use ATL components in your MFC applications. It's just that the two frameworks are separate. Borland takes the same approach with its separate OWL and BOCOLE libraries.

At the time I'm writing this article (June, 1996), version 1.0 of ATL is available for public inspection at *http://www.microsoft.com/intdev*. By the time you read it, version 1.1 should be available. Version 2.0, scheduled for release late this year, is the one that will change the world for control writers. [Note: Version 2.0 is the current release.]

In the meantime, you'll have to put up with a few limitations. The most important is that you can't easily use ATL to write what most people think of as controls. Technically, ActiveX has changed the definition of controls so that just about any ActiveX component (what we used to call an OLE server) qualifies as a control. A traditional control is simply a component with several specific interfaces—IDataObject, IOleControl, IConnectionPoint, IProvideClassInfo, and IOleInPlaceActiveObject, to name a few. In olden days, if a component didn't have all these interfaces, it wasn't a control. In the new ActiveX world, a control can have whatever interfaces it wants, but it should advertise what it has to potential containers through a COM Component Categories.

The MFC control wizards provide code for all the standard control interfaces. The ATL wizard provides code for only a few standard interfaces. You can add other interfaces if you want, so it's technically possible for COM experts to add all the traditional control interfaces. But with version 1 you you probably don't want to use it for anything except barebones in-process components. But what components! They're small and fast, and they don't require any support DLLs. Before we get into the code, let me make a few more points about ATL.

- ATL is 32-bit. The library is based on templates, a feature not available in 16-bit versions of Microsoft Visual C++. Perhaps you could make the library work with a compiler from some other vendor that cares more about 16-bit development than Microsoft. You're on your own. But even if you get it working, don't count on another 16-bit version of Visual Basic in which to use your components.
- ATL can do dual interfaces. Defining and describing dual interfaces would take another article as long as this one. To make a long story short, there are two kinds of interfaces you can use in an ActiveX component: vtable-based COM interfaces that run very fast but work only for early-bound objects, and IDispatch interfaces that are much slower but can be late-bound. Dual interfaces combine both kinds into one. MFC always uses its own custom implementation of IDispatch, and so it does very fast late-bound servers. But as far as Visual Basic 4.0 is concerned, that's like saying MFC does very fast steam engines. You can see the numbers in Table 2. The advantages of dual interfaces don't apply to Visual Basic controls. Visual Basic 4.0 can't take advantage of dual interfaces in traditional controls, only in servers. Future versions of Visual Basic will take advantage of any dual interfaces you provide for them.
- Dual interfaces (and therefore ATL) provide very little advantage for out-of-process (EXE) servers. The speedup gained from using an early-bound vtable interface is usually washed out by the cost of cross-process data transfer. Most EXE servers are primarily applications and secondarily components. I recommend using MFC, or better yet, Visual Basic for this kind of component.

THE SIEVE OF ERATOSHENES

Enough background. Let's implement the Sieve of Eratosthenes DLL. If you read my book *Hardcore Visual Basic*, you'll remember this algorithm. I used it to illustrate the performance differences between the early binding and late, between EXE servers and DLL servers, and between Basic servers and C++ servers. In the book I showed the Basic versions of the Sieve in detail, but didn't describe the C++ version provided on the CD. This time I'm going to ignore the Basic version and concentrate on the C++ version. This is the fourth C++ framework I've used to code the Sieve, so I should have it down. The downloadable code includes Basic and MFC versions of the framework, but I won't be discussing them.

To start out, let's assume you've downloaded the ActiveX Template Library from one of the VBPJ download sites along with the samples from this article. If you like living on the edge, you can download from the Microsoft site mentioned earlier, but bear in mind that you'll probably get version 1.1, which is being coded even as I write. You'll want to use the latest version eventually, but I can't guarantee that it will match my description. You might be safer to start with the older version supplied with this article.

Install by by unzipping ATL.ZIP into your MSDEV directory:

```
C:\MSDEV> pkzip -d atl
```

This automatically puts the wizard, the templates, and everything else you need in the appropriate directories. You can skim through the two documentation files in \MSDEV\DOCS, but remember, this is alpha documentation. Don't take everything you read as gospel. Now load Visual C++ 4.x. If you have an earlier version or compiler from a different vendor, you might still get the library working, but you'd have to do without the wizard.

OFF TO SEE THE WIZARD

Now you're ready to generate your first component with the wizard. You know the drill from writing components with MFC. You select New from the File menu and choose Project Workspace from the resulting listbox. In the New Project Workspace dialog, select OLE COM as the project type and fill in the name and location of the project. You need to plan ahead when selecting the name. The tendency with a component that consists of a single object class is to give it the name of the class—Sieve for the Sieve class. This is usually a bad idea. You need a two separate C++ file names—one for the component and another for the class. Reserve the class name for the class.

I'm going to call my sample component SieveATL, to distingish it from versions I've written with other languages and other frameworks. After filling in the name, press Create and you'll come up to a wizard screen that looks a lot like other wizard screens you've seen in Visual C++. I'm not going to show you the screen because it's bound to change in future versions. Here are the choices and my recommended selections, which happen to match the defaults:

How many objects would you like in your project? **1**
How interfaces per object? **1**
Allow merging of proxy/stub code? **No**
Support MFC? **No**
Generate ODL and IDL or Generate IDL only? **ODL and IDL**
DLL or EXE? **DLL**
Dual Interface or Custom Interface? **Dual**

If some of the alternatives interest you, I'll let you explore them on your own. The only choice I'm going to talk about is the one between ODL and IDL. ODL is the Object Description Language that you can compile with MKTYPLIB.EXE. IDL is the Interface Description Language that you can compile with MIDL.EXE. Microsoft is in the process of merging ODL and IDL. With the next major release of Visual C++, the MIDL compiler will completely replace the wretched MKTYPLIB, and it won't be too soon for me. I'd like to use IDL and MIDL for all projects, and I do on my own projects, but the new version of MIDL won't be available to many readers when this article is published.

Combining ODL and IDL is the only choice available to most readers. There's a third choice that isn't mentioned—use ODL without IDL the way MFC projects do. But for historical reasons that you can't change and so needn't question, the wizard won't help you implement the more convenient ODL-only route. Nothing to do but hit the Next button.

WHAT'S IN A NAME?

At this point you'll come to a screen with a listbox showing the component name you selected earlier—SieveATL in the example—and an Edit Names button. Ask yourself this question: "Do I trust a computer to choose my variable and file names?" Then press the Edit Names button.

This is where the wizard and I part company. In my opinion the wizard's suggestions are even worse than Visual Basic's idea that you might want to give the name Command1 to both the caption and the variable name of a button. Here's a table showing what the wizard proposes and what I recommend instead:

Element	**Wizard Suggestion**	**My Suggestion**
Short Name	SieveATL	CSieveATL
Class Name	CSieveATLObject	CSieveATL
Header File	SieveATLObj.h	Sieve.h
Implementation File	SieveATLObj.Cpp	Sieve.Cpp
Type Name	SieveATL Object	Eratosthenes
Type ID	SIEVEATL.SieveATLObject.1	SIEVEATL.CSieveATL

The "short name" is most important because it will be the class name in your Visual Basic client programs. The class name is the C++ implementation class name, and it makes sense that the internal class name should be the same as the external class name. They are the same class, after all. For the header and implementation files, I just want the shortest name possible that doesn't conflict with the component file names. I give a semi-random Type Name to illustrate the fact that the Wizard throws this name into the bit bucket (in my sample, at least). You'll search for it in vain in the generated files.

Finally, the Type ID is the name that you would use with CreateObject if you were ever foolish enough to instantiate an object from a dual interface component at run time. This name can be anything but is traditionally a combination of the component and class names. Some people (and some wizards) append a number at the end to indicate the version, but Visual Basic components rarely use this convention and neither do I.

The wizard's name preferences highlight a practice that has become common with COM components for reasons I don't understand. The word "object" is often used to describe a COM class. In normal object-oriented terminology, an object is a specific instance of a class. It is created by the client. And yet for some reason the term "COM object" is often used to describe classes that haven't yet been instantiated. Just say no to this terminology.

When you press the Finish button after changing the names, the wizard generates all the files for you, but you're not finished. Visual C++ doesn't know the build steps to compile an IDL file with MIDL or to register a server with REGSVR32.EXE. You have to add custom build rules. Don't waste your time looking in the documentation files for instructions on how to do it. The steps are actually explained in comments at the top of the component implementation file—SieveATL.Cpp. Once you find the instructions, follow them to set the build steps in the Custom Build tab of the Settings dialog. The wizard will add these steps automatically in a future version.

Now you are ready to build a working ActiveX component. It won't do anything, but you might still be interested in building a Release version to see how small an ATL component can be. The ATL documentation file tells how to reduce the size even further if your component (like SieveATL) doesn't need the C run-time library.

SIEVE METHODS AND PROPERTIES

You know the next step: load Class Wizard and define the properties and methods. Alas, Class Wizard doesn't yet know anything about ATL. You must act as wizard, coordinating related declarations and definitions in four different files—Sieve.Cpp, Sieve.h, SieveATL.Idl, and SieveATL.Odl. In addition, the MIDL compiler will act as a wizardlet, generating different versions of the declarations in two additional files—SieveATL.h and SieveATL_i.c.

One of the document files that comes with ATL contains a cursory and not entirely accurate description of how to write methods, but we're going to ignore it and do methods and properties my way. But first let's look at what a Sieve requires.

The Sieve of Eratosthenes algorithm was published in *Byte* magazine back in 1983 and the function has been used as for benchmarking ever since. You pass the function the maximum number you want to evaluate, and it calculates all the prime numbers up to the maximum. When benchmarking, you don't care what the prime numbers are. You only care how long it takes to calculate them. But this is an unrealistic test of a server. You have to at least pretend that a server has a client to consume your output. And a normal client will consume prime numbers one at a time in order to keep an even flow rather than wait for all to be calculated before using the first.

Here are the methods and properties I implement in all my Sieve classes:

Member	**Purpose**
NextPrime property (readonly)	Get the next prime number
MaxPrime property	Get or set the maximum number before starting the generator
Primes property (readonly)	Get the current count of prime numbers
ReInitialize method	Reinitialize the generator
GetAllPrimes method	Get all the prime numbers at once and return them in an array

If we had a server like with those members, we could use it from a Basic client with code like this:

```
Dim sieveATL As New CSieveATL
sieveATL.MaxPrime = txtMaxPrime.Text
' Get one at a time
ms = timeGetTime()
For i = 1 To cIter
    sieveATL.ReInitialize
    Do
        iPrime = sieveATL.NextPrime
        If fDisplay And iPrime Then
            With lstOutput
                .AddItem iPrime
                .TopIndex =.ListCount - 1
                .Refresh
            End With
        End If
    Loop Until iPrime = 0
Next
txtTime.Text = timeGetTime() - ms
txtPrimes.Text = sieveATL.Primes
```

That code comes from SieveCli.Vbp. The Sieve Client application tests four variations of the sieve class—a Basic Exe component, a Basic DLL component, a C++ MFC component, and the one we're really interested in, the ATL version. Figure 1 shows the the Sieve Client program in action.

THE CSIEVEATL CLASS

Now to the C++ implementation. First step is to finish what the wizard started in Sieve.h. You must enter the methods, properties, and private data members for your class. Listing 1 shows the completed class. I'll go over the parts I added.

The property and method declarations go at the bottom just after the public statement. If you like to follow directions, insert them like this:

```
STDMETHOD(get_NextPrime)(short * pi);
```

The STDMETHOD macro is carefully designed to allow you to use C or C++ from the same property or method declaration, but only a masochist would write a COM object in C. If you examine the STDMETHOD macro, you can see what it's hiding:

```
virtual HRESULT STDMETHODCALLTYPE
        get_NextPrime(short * piRet);
```

I prefer to declare my methods and properties out in the open without the macro. Notice that the functions are virtual (they go through a v-table) and they have STDMETHODCALLTYPE, which is actually __stdcall on Win32 platforms. Like all COM methods, the get and put functions return an HRESULT—a code through which an error can be specified. The return value that will be seen by the client must be returned through a pointer. The last parameter of any method with a return (including property get functions) must be a pointer to the return value. Don't worry. Visual Basic will take care of this so that your methods and properties will have the syntax you expect on the client side.

Next you should add any private member variables your class needs. You can see the ones requires by CSieveATL at the bottom of Listing 1. One other detail: By default the wizard assumes that your class won't need a constructor, so it gives you this empty one:

```
CSieveATL() {}
```

It also assumes you won't want a destructor. You need to correct these assumptions if your class must be constructed and destructed.

```
CSieveATL();
~CSieveATL();
```

I also insert the following constant at the top of the file:

```
const short PRIME_MAX = 32767;
```

I don't even remember why I made the original Basic CSieve class use Integer (short in C++) rather than Long, but I've kept it the same in C++ for compatibility.

While you're looking at Sieve.h in Listing 1, notice how the class is derived from CComDualImpl, ISupportErrorInfo, and CComObjectBase. It also has a COM_MAP block defining several interfaces. This is the code handles reference counting, interface querying, object creation with class factories, type library connections, error handling, registration, and all the other messy COM details that most of us prefer to ignore. Of course, as your components get more complicated, you won't be able to ignore details forever. When you need to understand what ATL is doing, look in the implementation files. ATLBase.h and ATLCOM.h are included from StdAfx.h, and ATLImpl.Cpp is included from StdAfx.Cpp.

ODL AND IDL FILES

Before we get to the implementation of SieveATL, let's put the corresponding declarations in the ODL file. This file will be compiled by MKTYPLIB to build a type library that will automatically be embedded into the DLL as a resource (as specified in SieveATL.Rc). You load the DLL in the Visual Basic References dialog. Visual Basic will read all the class information it needs out of the type library. Listing 2 shows the SieveATL.Odl.

It declares the same methods and properties declared in the class, but in a slightly different format that includes attributes. Your Visual Basic customers will justifiably regard you with contempt if you fail to provide at least a helpstring for each method and property. You should probably also create a help file and use the helpfile and helpcontext attributes so that users of your component can get complete information in the Object Browser.

Next step is to fill the IDL file. This file will be compiled by MIDL to create the marshalling code for transferring data between the server and client. COM marshalling is far beyond the scope of this column. You'll have to take it on faith. If you check the C++ code generated by MIDL in the files SieveATL.h and SieveATL_i.c, you'll see what I mean. I'm not going to show a complete listing because SieveATL.Idl looks like a stripped down version of SieveATL.Odl. The interface section looks like this:

```
// Properties
[propget]
HRESULT NextPrime([out, retval] short * pi);
[propget]
HRESULT MaxPrime([out, retval] short * pi);
[propput]
HRESULT MaxPrime([in] short i);
[propget]
HRESULT Primes([out, retval] short * pi);
// Methods
HRESULT ReInitialize(void);
HRESULT AllPrimes([in, out] SAFEARRAY ** ai);
```

You don't need to give a helpstring because the old version of MIDL we're using here wouldn't know what to do with it. The new version combines the functionality of MIDL and MKTYPLIB, so if you can get your hands on it, you'll need only one file.

IMPLEMENTING CSIEVEATL

We're finally ready to implement the methods and properties in Sieve.Cpp. Listing 3 shows the entire implementation, but we'll look closely at the MaxPrime property here.

```
// MaxPrime property
HRESULT CSieveATL::get_MaxPrime(short * piRet)
```

```
{
    *piRet = m_iMaxPrime;
    return NOERROR;
}

HRESULT CSieveATL::put_MaxPrime(short iMaxPrime)
{
    if (iMaxPrime <= 0) {
        return Error(OLESTR("Invalid maximum"));
    }
    m_iMaxPrime = iMaxPrime;
    return NOERROR;
}
```

Notice how get_MaxPrime returns its value through the piRet pointer variable. You can see how the HRESULT is used. Methods and property functions should either return NOERROR or call the Error method of the CComObjectBase class. Notice that strings passed to the Error method should use the OLESTR macro so that they come right on platforms (such as Win32) where COM requires Unicode strings.

I'm not going to say anything more about the implementation except to call your attention to the difference between the way the sieve algorithm is implemented in the AllPrimes method and in the NextPrime property. AllPrimes looks and works much like the original Sieve function published in *Byte*, but you'll have to study the NextPrime property for a while to recognize it as the same algorithm.

THE RESULTS

ATL is a little rough around the edges, but even with the limited wizard support in the pre-release version you can see that it won't be all that hard to turn out small components. And the library has many other features that I wasn't able to show in this article. One of the more exciting is its support for creating enumeration classes. If you don't like Visual Basic collections, throw them out and write your own version.

The ultimate test of a component is performance. That's where ATL really comes through in both size and speed. Of course, it remains to be seen whether the template approach will work as well for more traditional controls with windows, events, persistence, property pages, and all the rest. But the ActiveX Template Library is off to an encouraging start.

LISTINGS:

Listing 1

```
// Sieve.h : Declaration of the CSieveATL
#include "resource.h"          // main symbols

const short PRIME_MAX = 32767;

// CSieveATL

class CSieveATL :
    public CComDualImpl<ICSieveATL, &IID_ICSieveATL,
                        &LIBID_SIEVEATLLib>,
    public ISupportErrorInfo,
    public CComObjectBase<&CLSID_CSieveATL>
{
public:
    CSieveATL();
    ~CSieveATL();
BEGIN_COM_MAP(CSieveATL)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(ICSieveATL)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
// Use DECLARE_NOT_AGGREGATABLE(CSieveATL) if you
// don't want your object to support aggregation
DECLARE_AGGREGATABLE(CSieveATL)
// ISupportsErrorInfo
    virtual HRESULT STDMETHODCALLTYPE
        InterfaceSupportsErrorInfo(REFIID riid);

// ICSieveATL
public:
      virtual HRESULT STDMETHODCALLTYPE
            get_NextPrime(short * piRet);
      virtual HRESULT STDMETHODCALLTYPE
            get_MaxPrime(short * piRet);
      virtual HRESULT STDMETHODCALLTYPE
            put_MaxPrime(short i);
      virtual HRESULT STDMETHODCALLTYPE
            get_Primes(short * piRet);
      virtual HRESULT STDMETHODCALLTYPE
            ReInitialize();
      virtual HRESULT STDMETHODCALLTYPE
            AllPrimes(SAFEARRAY ** ai);
private:
      short * m_af;                // Array of flags
      int       m_iCur;            // Current index
      short     m_iMaxPrime;       // Maximum index
      short     m_cPrime;          // Count of primes
};
```

Listing 2

```
// SieveATL.Odl : Type library source

// This file will be processed by the Make Type Library
// (MKTYPLIB) tool to produce the SieveATL.Tlb

[     uuid(18315EE2-B8D4-11CF-AEC1-444553540000),
      version(1.0),
      helpstring("Sieve of Eratosthenes As C++ ATL DLL")
]
library SIEVEATLLib {
      importlib("stdole32.tlb");

      [     uuid(18315EE3-B8D4-11CF-AEC1-444553540000),
            dual
      ]
      interface ICSieveATL : IDispatch {
            // Properties
            [propget, helpstring("The next prime number")]
            HRESULT NextPrime([out, retval] short * pi);

            [propget,
            helpstring("The maximum prime number to evaluate")]
            HRESULT MaxPrime([out, retval] short * pi);
            [propput]
            HRESULT MaxPrime([in] short i);

            [propget,
            helpstring("Count of calculated prime numbers")]
            HRESULT Primes([out, retval] short * pi);

            // Methods
            [helpstring("Reinitialize the prime number counter")]
            HRESULT ReInitialize(void);

            [helpstring("Get all primes in a given array")]
            HRESULT AllPrimes([in, out] SAFEARRAY(short) * ai);
      };

      [     uuid(18315EE7-B8D4-11CF-AEC1-444553540000),
            helpstring("CSieveATL Class")
      ]
      coclass CSieveATL {
            [default] interface ICSieveATL;
```

```
    };
};
```

Listing 3

```
// Sieve.cpp : Implementation of CSieveATL class
#include "stdafx.h"
#include "SieveATL.h"
#include "Sieve.h"

HRESULT CSieveATL::InterfaceSupportsErrorInfo(REFIID riid)
{
   if (riid == IID_ICSieveATL)
       return NOERROR;
   return S_FALSE;
}

// CSieve Constructor and destructor
CSieveATL::CSieveATL()
{
   m_af = 0;
   m_iMaxPrime = PRIME_MAX;
}

CSieveATL::~CSieveATL()
{
   m_iMaxPrime = 0;      // Just a place for breakpoints
}

// NextPrime property
HRESULT CSieveATL::get_NextPrime(short * piRet)
{
   // Loop until we find a prime or exceed maximum
   m_iCur++;
   while (m_af[m_iCur]) {
       m_iCur++;
       // When maximum exceeded, return 0
       if (m_iCur > m_iMaxPrime) {
           *piRet = 0;
           return NOERROR;
       }
   }
   // Cancel multiples of this prime
   int i;
   for (i = m_iCur + m_iCur; i < m_iMaxPrime; i += m_iCur) {
       m_af[i] = 1;
   }
   // Count this prime and return it
   m_cPrime++;
   *piRet = m_iCur;
   return NOERROR;
}

// MaxPrime property
HRESULT CSieveATL::get_MaxPrime(short * piRet)
{
   *piRet = m_iMaxPrime;
   return NOERROR;
}

HRESULT CSieveATL::put_MaxPrime(short iMaxPrime)
{
   if (iMaxPrime <= 0) {
       return Error(OLESTR("Invalid maximum"));
   }
```

```
    m_iMaxPrime = iMaxPrime;
    return NOERROR;
}

// Primes property
HRESULT CSieveATL::get_Primes(short * piRet)
{
    *piRet = m_cPrime;
    return NOERROR;
}

// Reinitialize method
HRESULT CSieveATL::ReInitialize()
{
    if (m_af) {
        delete m_af;
    }
    m_af = new short[m_iMaxPrime];
    memset(m_af, 0, sizeof(short) * m_iMaxPrime);
    m_iCur = 1;
    m_cPrime = 0;
    return NOERROR;
}

// AllPrimes method
HRESULT CSieveATL::AllPrimes(LPSAFEARRAY * ppsa)
{
    int i;
    long c;
    short * af;
    SAFEARRAYBOUND asabound[1];
    // Validate array
    if (SafeArrayGetDim(*ppsa) != 1)
         goto AllPrimesError;
    if (SafeArrayGetLBound(*ppsa, 1, &c) != NOERROR)
        goto AllPrimesError;
    if (c != 0)
        goto AllPrimesError;
    if (SafeArrayGetUBound(*ppsa, 1, &c) != NOERROR)
        goto AllPrimesError;
    if (c > PRIME_MAX)
        goto AllPrimesError;
    if (SafeArrayGetElemsize(*ppsa) != sizeof(short))
        goto AllPrimesError;
    // Unlock direct access to data
    if (SafeArrayAccessData(*ppsa, (void **)&af) != NOERROR)
        goto AllPrimesError;

    // Maximum prime is the size of the array
    m_iMaxPrime = (short)c;
    m_cPrime = 0;
    // Sieve of Eratosthenes algorithm
    for (m_iCur = 2; m_iCur < m_iMaxPrime; m_iCur++) {
        if (!af[m_iCur]) {
            // Found a prime
            for (i = m_iCur + m_iCur; i < m_iMaxPrime;
                                                    i += m_iCur) {
                af[i] = 1;   // Cancel its multiples
            }
            // Write prime into position in array, count it
            af[m_cPrime] = m_iCur;
            m_cPrime++;
        }
    }
    // Relock direct access to data
    if (SafeArrayUnaccessData(*ppsa) != NOERROR)
        goto AllPrimesError;

    // Resize to include only the data
    asabound[0].lLbound = 0;
    asabound[0].cElements = m_cPrime;
    if (SafeArrayRedim(*ppsa, asabound) != NOERROR)
        goto AllPrimesError;
    m_iCur = 1;
    return NOERROR;

AllPrimesError:
    return Error(OLESTR("Invalid array"));
}
```

SIDEBAR

COM, OLE, and ActiveX

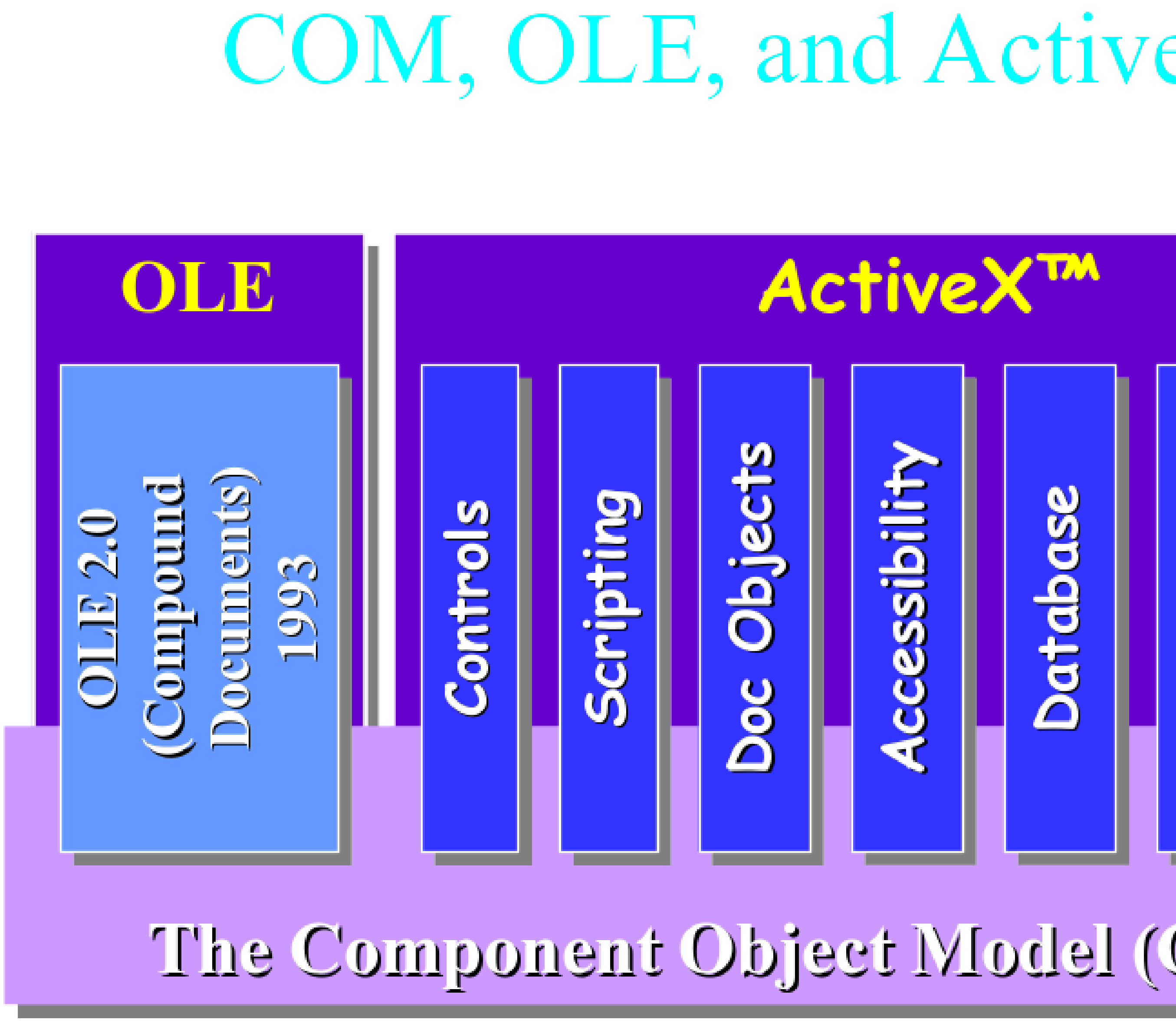
Unless you've been on another planet for the last few years, you've probably heard of OLE and have some idea of what it is. But perhaps you're still confused about how it relates to COM and ActiveX. Welcome to the club.

Originally, OLE meant Object Linking and Embedding, and it described the compound document technology. But OLE quickly grew far beyond that description. Automation, for example, had no connection to linking and embedding except that both are based on the same technology—the Component Object Model (COM). As OLE grew, Microsoft changed the acronym into a word. OLE meant all the technologies based on COM, and the letters weren't supposed to stand for anything. But language is a living thing that doesn't always take kindly to manipulation. Many publications continued to use nonsensical terminology such as Object Linking and Embedding Automation.

Microsoft has finally gone back to using OLE as an acronym meaning Object Linking and Embedding. The family of technologies that used to be called OLE is now called COM. OLE is just one branch of the COM tree. The other main branch is called ActiveX—and it includesActiveX controls, ActiveX Doc Objects, and ActiveX Scripting, as shown in Figure 2.

In the Visual Basic department, the term server is out and component is in. What we used to call an OLE server is now a COM component. In contexts where it matters whether the component is contained within a DLL or an EXE file, you can use the term ActiveX DLL or ActiveX Application.

The new terminology probably makes more sense, but a lot of trees will die and a lot of editors will tear their hair out before we all get used to it.



[This should be a screen dump of the Sieve Client application in action. You can see it on page 544 of Hardcore Visual Basic, Second Edition.]

Figure 1
The Sieve Client application. This *sample application tests four different OLE components that use the Sieve of Eratosthenes altgorithm to calculate prime numbers. The figure shows the Display checkbox on so that you can see that the Sieve does indeed calculate prime numbers. You should normally uncheck this box when benchmarking so that you'll be timing calculations rather than screen display.*