

Alternative Component Architectures for Java

Author: Ted Faison, Faison Computing Inc.

Type: Technical Session

Track: JBuilder

Short Description

Confused with component models like ActiveX, JavaBeans and IFC? This presentation will give you an overview of each model, showing you their relative strengths and weaknesses. It will show you what types of programming scenarios each model is best suited for, and put you in a better position to decide which one is for you.

Abstract

Until recently, the choice for a component technology was straightforward. Each platform had only one major architecture: Windows with OLE/COM, the Macintosh with OpenDoc and OS/2 with SOM. Then there was CORBA, an industry attempt to bridge the platforms. With Java, the choices are no longer that simple: on top of the older models, you have JavaBeans and the Internet Foundation Classes. COM has been born again as ActiveX, and all of them are competing in the cross-platform market.

Introduction

Historians of computer software will probably characterize this decade as the golden age of component architectures. New architectures are popping up everywhere, and programmers are faced with more and more choices. Choices entail responsibility and knowledge, and it is getting increasingly difficult for programmers to master all the models that are out there now. In this presentation I'll be discussing three of the component architectures that are available to Web developers. To keep things focused, I'll be discussing those component models that are important to Windows programmers, since they make up the overwhelming majority of programmers out there. Those models are JavaBeans, the Netscape Internet Foundation Classes (IFC) and Microsoft ActiveX. I'll give highlights of the features, benefits and drawbacks of each, and compare them side-by-side so you can determine which model is appropriate for you.

As of the writing of this paper, keep in mind that the Java Foundation Classes (JFC) from Sun were not available yet. While IFC is a precursor to JFC, it is considerably different. IFC is a component model, JFC isn't. IFC has an architecture that supports persistence, data transfer, properties, event-handling, etc. The JavaBeans spec already supports those features and represents the framework JFC is layered on top of. JFC gives you advanced user interface components, and enables you to build world-class applications more easily than using the JDK Abstract Window Toolkit (AWT) alone.

Highlights of Beans, IFC and ActiveX.

The JavaBeans spec was written essentially to create a common framework for Application Builder tools to use, and to facilitate software development by supporting many of those features developers have been asking for, like serialization, versioning, properties, clipboard support, uniform data transfer and more. Before Beans, developers were forced to use a relatively low level approach to develop code, since tools were few and far between. The tools

that were available were primarily GUI builders, but didn't address the development of Java classes. JavaBeans defines an API that allows Application Builder tools to discover features of Beans at design-time, so items like Property and Event Inspectors become possible. The idea is that the better support JavaBeans gives builder tools, the easier it will be to develop code.

JavaBeans is a component architecture. It doesn't extend the AWT, in terms of the GUI components available. People developing real applications, like Netscape and Corel, have had to write a substantial amount of code to create full-featured user interfaces. Stuff users take for granted, like toolbars, tree lists, tree views and splitter windows are not available in the JDK 1.1 AWT. Each vendor came up with a proprietary solution. Netscape decided to make theirs public, and made IFC (including the full source code) available at their web site. Their pitch was for web site designers to use IFC to create really cool pages with the advanced controls. Pages containing IFC code contain a special button allowing users to download the IFC extensions into their Java environment. Using these extensions, browsers can view IFC-enabled pages just like any other pages. While IFC was designed to make advanced Web pages, it can be used to build regular applications that have nothing to do with the Web. JFC and IFC both support advanced GUI controls, and are both 100% pure Java, but they differ in the area of look-and-feel. IFC uses its own code to draw components, so IFC windows look the same on all platforms. The AWT and JFC take an alternative approach. They use special platform-dependent peer windows to display components in a platform-dependent way. Programs developed using AWT and JFC adapt to the look-and-feel conventions of the native machine. An AWT radio button on a Mac looks like all the other radio buttons on the Mac. With IFC programs, radio buttons look different. All the controls look different, which isn't to say they look bad, but users get used to seeing things in a given way. GUI operating systems like Windows and the Mac OS were created specifically to give all applications a common look and feel. I think Netscape's decision to change this fundamental idea was questionable.

ActiveX comes from an entirely different direction. COM was originally developed to help Microsoft create OLE. Back in those days, it was important to be able to drop an Excel spreadsheet into a Word application. The focus was on embedding or linking applications. Microsoft has been using COM to implement an increasing number of components used by both applications and the operating system itself. With the explosion of the internet, Microsoft has fine tuned OLE Custom Controls (OCX) for embedding in Web pages and distribution over the Web. ActiveX controls are generally smaller and easier to develop than full-fledged OCX controls. To build an ActiveX control, you don't need the full OLE 2.0 libraries, which are rather large. With Visual Basic 5.0 and Visual C++ 5.0 you develop Active controls using wizards, taking much of the pain out of the development process. You click a few buttons and you have an ActiveX component to customize with your own features. You can currently run certain ActiveX controls on the Macintosh and several versions of Unix, but only using special plug-in adaptors. The adaptors run inside your browser, so different adaptors must be developed for each browser. Obviously adaptors exist for Internet Explorer, but Microsoft also developed adaptors for Netscape Navigator and Mosaic.

Support for scripting environments

When you develop components, you use programming languages like C++, Object Pascal or Visual Basic. When you're done, you can glue components together in 2 basic ways: using more C++, Object Pascal or Visual Basic, or using a scripting language. Web pages commonly use scripting languages to orchestrate the Java Applets or ActiveX controls embedded on them. Most Web browsers support two main languages: JavaScript and VBScript. Both allow you to use simple syntax to access properties or invoke methods. Java components can be manipulated using JavaScript. Microsoft also has a custom version of JavaScript called JScript, which can also talk to ActiveX controls. Both JavaScript and VBScript support `object.property` access like this:

```
apple.color = Green
name = myObject.label
airplane.index = 2
```

VBScript is designed to support ActiveX Scripting components, meaning it works with ActiveX. You can't use JavaScript to talk to ActiveX components. Can you use VBScript to talk to Java applets? No. To summarize:

JavaScript works with Java components, VBScript works with ActiveX.

Can Beans, IFC and ActiveX be used together?

Because software reuse is so important, you want to use any existing components you can get, which creates the potential for mixing and matching components built in diverse architectures. Because JavaBeans and IFC are both pure Java implementations, it seems reasonable to believe that they can be freely mixed. Not so fast. The two models overlap in certain key areas, such as window management, persistence, event handling and data transfer. This translates into the need to handle components in each model differently. For example, JavaBeans uses an Event/Listener model, IFC doesn't. To connect two objects together, you must know whether they are JavaBeans or IFC components.

JavaSoft recently announced an ActiveX bridge, which lets you package a JavaBean as an ActiveX component. The bridge lets you embed your JavaBean as an ActiveX control in a Word or Excel document, or put your JavaBean anywhere an ActiveX control can be used.

You can't currently package an ActiveX control as a JavaBean, so you can't mix Java code with ActiveX components directly, unless your code is written specifically for the Windows platform using Visual J++. The Microsoft JVM has extensions allowing you to manipulate ActiveX controls using standard Java notation, allowing you to write code like:

```
import MyActiveXControl.*;

// MyActiveXControl is an ActiveX control !
MyActiveXControl control = new MyActiveXControl();
control.setX(50);
int x = control.getX();
```

Listing 1 - Calling an ActiveX control from Java.

Keep in mind that, to date, the only Java compiler that knows about the ActiveX Microsoft JVM extensions is Microsoft Visual J++, so the code in Listing 1 will only work with Visual J++.

The notation used in Listing 1 definitely looks like 100% pure Java code, and is simple to understand. But there's a whole lot of stuff going on beneath the surface. First there is that **import** statement, referencing not a Java package, but a COM Type Library File. The Microsoft JVM was in fact extended to look first for `.class` files, then for `.tlb`, `.idl` or `.odl` COM library files. If it finds a COM library, Visual J++ knows the classes are COM classes, and treats them differently from Java classes. For example, the **new** keyword normally creates objects from the Java heap. If the object being created is an ActiveX control, the object is created in the COM memory space, by calling `CoCreateInstance()`. The compiler also internally adds code to transparently support COM reference counting. Microsoft has gone to great lengths to make the integration of Java and ActiveX code seamless.

Visual J++ even lets you create ActiveX controls using Java as the development language. When you compile the Java code, Visual J++ doesn't create `.class` files. Instead it creates a DLL containing native binary code, with embedded calls to the COM libraries, for your component. Visual J++ makes it really easy to create ActiveX controls.

The ability to mix Java and ActiveX controls together sounds really cool. In reality, mixing component models is a prescription for headaches, if your aim is platform independence. If you have to mix ActiveX and Java components on a Web page, you can use Microsoft JScript to script them, but keep in mind that JScript is only supported currently on Windows platforms. If you need to mix IFC and JavaBeans together, things might get pretty hairy if you need to support some of the advanced mechanisms like uniform data transfer, clipboard operations, drag 'n drop and serialization, because the two models support those mechanisms differently.

How do these different architectures compare structurally?

Structurally, JavaBeans, IFC and ActiveX components have something in common. They are all containers, in a certain sense. They all allow you to put components inside components. They also support the notion of control through a scripting language, using properties and methods. That's about how far the similarities go, I'm afraid.

JavaBeans components come in two versions: heavyweight and lightweight. The former use objects rendered by the underlying platform, the latter don't. A JavaBeans Button is a heavyweight component, because it uses a native `BUTTON` control (manipulated through an object called a *peer*) created by the operating system. The native control requires a number of OS resources, and also is the source of events -- many of which you often have no interest for. Lightweight components paint themselves on the screen and are not tied into the platform's windowing system. They are fast to load, quick to execute and small in size. Lightweight components deliver a uniform look-and-feel across platforms, while heavyweight ones deliver a platform-specific appearance.

IFC also has support for heavy and lightweight components, but with a twist: the IFC has its own internal windowing subsystem. The GUI components you create are so-called *internal windows*, which correspond to JDK lightweight components. An IFC application typically has a native Window as the main frame, and internal windows for just about everything else. This arrangement makes IFC applications relatively fast, but also somewhat different from everything else on the user's desktop. IFC applications have their own look-and-feel, and certain native operations like drag and drop or clipboard commands don't work between IFC and non-IFC applications.

ActiveX components can be light or heavy. A light ActiveX component is one that paints itself and doesn't make use of GUI objects from the native Windowing system. ActiveX doesn't have the peer concept. When running on Windows platforms, heavyweight ActiveX controls call into the Windows API directly to create native Windows controls. On non-Windows platforms, the ActiveX adaptor translates these calls into platform-specific calls.

End-User Features Compared

JavaBeans, IFC and ActiveX all support an array of important features, but often in different ways. The differences are likely to be more important to end users than to developers. What really matters to users is how long they have to wait to download a component and how well components fit into their desktop environment. This translates into concerns for runtime requirements, component size, portability, look and feel, and security. I'll address each issue separately.

Runtime requirements

When users download an embedded component in a Web page, they want the code to be up and running in the shortest amount of time possible. Not only the size of the component is important, but also whether additional paraphernalia must be downloaded to support the component. JavaBeans components require nothing more than the Java bytecodes of the component itself. The Java Virtual Machine on the user's machine will supply everything else needed.

IFC components need a bit of extra help. They require the IFC support classes to be downloaded and installed in the user's Java environment before they will run correctly. In practice, Web pages that contain IFC components usually contain a button that looks like this:



Figure 1 - The Netscape IFC download button.

Clicking the button installs the IFC runtime environment into the user's browser automatically. On a Windows platform, what actually occurs is the user's browser downloads a 375 KB .CAB file named `nsifc10.cab`. With a 28.8 Kb/s modem the transfer takes about 2 minutes. The file is a Microsoft Cabinet file containing the .class files for the Netscape IFC environment. The files are extracted from the .CAB file and stored in the Java environment for the browser. For the Internet Explorer, the files are saved in the two directories

`C:\Windows\Java\Lib\Netscape\Application` and `C:\WindowsJava\Lib\Netscape\Util`. Once the IFC environment is setup, you can open all the IFC-enabled pages you want with no additional hassles, until you hit a page that uses a version of IFC that is later than the one on your machine. You don't get any error messages to tell you the versions are different. What usually happens is you find that certain applets don't seem to work correctly, or they don't appear at all. Techies like you and me might suspect an IFC version problem and click the IFC download button on the page, but ordinary users wouldn't have a clue what was going on.

ActiveX controls may or not require support code to function. If the controls were created with Delphi, they generally require no support code. If they were built with Visual C++, they may require an MFC runtime library file (MFC42.DLL or equivalent) that is over 900 KB in size. Visual Basic 5.0 lets you build really small ActiveX controls, but not for free: they need support from the Visual Basic Virtual Machine (VBVM), a 1.3MB behemoth, downloaded in compressed format inside a 730 KB .CAB file called `MSVBVMB5.CAB`. The VBVM gets cached on the user's machine, so it only needs to be downloaded once. Other than the 5-8 minute download time, installing the VBVM is entirely automatic. Still, users may not like the idea of downloading a 730KB library to run a 15 KB control.

Size

JavaBeans and IFC components are often quite small – in the 5-20 KB range. Java objects are not statically linked to some library, to support operations like I/O or string handling as in C++. The Java class loader is a runtime linker that links calls in one .class file to methods in another .class file, that often is part of the Java environment. There is a small runtime penalty to pay for this runtime linkage fix-up, but the payoff comes in the form of truly microscopic sizes. IFC components tend to be a little larger than JavaBeans, because they are part of a larger class library. IFC applications use custom IFC classes to achieve results supported by the JDK 1.2. Still, IFC classes are commonly under 30 KB. Size is definitely one of the biggest advantages of JavaBeans and IFC over ActiveX.

OLE was developed in a time when size wasn't an issue. When the Redmontonians created OLE 2.0, stuff like drag 'n drop, in-place activation and menu merging were important. Whether an OLE component was 100 KB or 1000 KB didn't make much difference. That was then. Now, of course, size is everything. Microsoft took the OLE Custom Control architecture (OCX), simplified it a bit, and reintroduced it to us as ActiveX. If you look at the Web pages that contain ActiveX controls, such as those on the Microsoft Web site, you'll see that their average size is in the 25 to 250 KB range approximately. Not too bad, but still about ten times larger than typical Java-powered components. ActiveX controls developed using Visual Basic 5.0 are starting to show up in increasing numbers on the Web. These controls are often quite a bit smaller than older controls. It isn't uncommon to come across controls in the 10-25KB range. Keep in mind the Visual Basic Virtual Machine hit I mentioned in the previous section, which fortunately is only a one-time deal.

Portability

When a user is browsing the Web, s/he could be using any browser or platform. Loading a Web page should be a no-brainer, meaning everything on the page should work regardless of which platform the user's browser is running. JavaBeans run everywhere, or I should say everywhere there is a Java VM. IFC components require a Java VM and the IFC support classes mentioned earlier.

ActiveX components are somewhat portable. They run on Windows platforms, which constitute probably over 90% of all the machines out there. They also run on Macs, but require a special ActiveX adaptor. Adaptors for several flavors of Unix are also in the works. These adaptors are available from the Microsoft web site, and must be downloaded and installed in user machines before ActiveX will run. They provide a function similar to the Java VM, in that they insulate the ActiveX code from the details of the user's platform.

Look and Feel

Users can get religious over look-and-feel. All GUI environments provide essentially the same features, but users grow accustomed to seeing things in a certain way. A button should always look like a button, a check box like a check box. It is important for graphical elements of downloaded controls and applications to appear familiar to users. In this respect, JavaBeans and ActiveX have an advantage of IFC. The former two allow code to conform to the standards of the user's platform. On the other hand, IFC uses a single look-and-feel that was established by the Netscape developers. IFC controls draw themselves, without support from native code. While this certainly provides the benefit of a uniform and consistent appearance of IFC applications, it also makes IFC programs stick out like a sore thumb to the user. This is not to say the controls look bad. Just different. The following figure shows how an IFC window looks:



Figure 2 - A simple IFC view with a few controls.

Security

Most users are non-technical, which might lead developers to assume they don't care much about techie stuff like security. Wrong. They read the news and know all about computer viruses. They know viruses are some kind of disease computers can get from the internet that could wipe out their data. They also know that giving out sensitive information like credit card numbers over the internet can be dangerous. I won't address wire security here, because it really doesn't depend on which component model you are using, but on the low level protocols adopted.

Once code is downloaded into your system through a Web page, what can be done to prevent it from sabotaging your system or destroying your data? Java was built from the start to support a *sandbox* security model. When Java code is downloaded from a remote site, Java puts special restrictions on what that code can do in your system. There is a special `SecurityManager` object to oversee everything the code does. For example the code won't be allowed to access your file system – even to check for the existence of files or directories. The code won't be able to send data over the wire, except back to the host the code originated on.

ActiveX is another story. ActiveX is a native model, executing (at least on Windows platforms) at the same hardware privilege level as all other applications. This is good, because ActiveX controls can do anything they like. This is also bad, because ActiveX controls can do ANYTHING they like. The Microsoft security model is based on the notion of *trust* and *trustworthiness*. ActiveX code can be *code signed*, meaning the producer of the code tags it with some data attesting to its authenticity. Of course authentic code is by no means *guaranteed* to be harmless.

When your browser needs to download executable code from a web site, it has security options to prevent it from downloading code that wasn't signed. ActiveX controls from companies like Microsoft are digitally signed. Before downloading controls like these, your browser will display the Digital Certificate for the code, which looks something like this:



Figure 3 - A Digital Certificate for a signed ActiveX control.

The browser asks you, the end user, whether you trust the source of the code you are about to download. The certificate tells you the code hasn't been tampered with, but how can you be sure the code won't cause trouble on your system? Maybe it has a bug that on the 15th of each month reformats your hard disk. Because ActiveX controls are empowered to do anything they want, the end user is required to take responsibility and vouch for the safety of any controls downloaded. This may or may not be an acceptable solution to security.

Other Issues

As I stated earlier, the above issues are probably the most important to end users, but aren't the only ones. Drag-and-drop and clipboard operations come to mind. The two are related because they are based on the ability of a model to support a uniform method to transferring data. JavaBeans use a number of JDK 1.1 features built on the Uniform Data Transfer (UDT) architecture, which uses services from the native platform to access the clipboard. This means you can transfer data from Java to non-Java applications and vice-versa without any hocus-pocus. The IFC framework supports drag and drop using its own implementation, bypassing the native platform. The problem is that it doesn't allow you to transfer data between IFC and non-IFC programs. It doesn't even let you drag and drop between IFC and regular Java programs.

ActiveX supports drag and drop using native platform services. I'm told the Macintosh ActiveX adaptor supports drag and drop, but Microsoft hasn't released information on whether it intends to support clipboard operations or drag and drop on Unix platforms.

Developer Features Compared

From the developer's perspective, there are significant differences between JavaBeans, IFC and Active components. The biggest is probably the level of complexity. To create JavaBeans and IFC components, you obviously need to know Java. I'll ignore the language learning curve, which is relatively small for C++ and ObjectPascal programmers. There is no class hierarchy or framework to learn to develop JavaBeans. There is no JavaBean class from which you derive your Beans. JavaBeans is a specification, and a rather simple one at that. It is mostly based on features provided by the Java language itself, like serialization. JavaBeans event handling is a matter of understanding the simple event-listener model. Properties are based on a simple naming convention for methods. In short, JavaBeans is a fairly easy specification to master.

IFC is a bit more complicated. Because it is built with a class hierarchy, the first thing to do is study and understand this hierarchy. To create a serializable object, you derive it from the IFC class `CoDable`. Then you need to understand the IFC persistence model, which gets no help from the Java language. You, the developer, must add methods to each serializable class to stream in or out required class data. IFC comes with its own view hierarchy, based on a root window and a series of nested internal windows. More stuff to learn. The overall learning curve is not too bad, but then consider the overlap that exists between IFC and JFC, and you might decide that learning IFC just isn't worth it.

ActiveX is the true beast of the three. It doesn't matter what Microsoft says about ActiveX, or how many wizards you use. ActiveX is complicated, especially if you plan on using any OLE 2.0 features. Then the learning curve becomes essentially vertical. There are dozens and dozens of interfaces to deal with and, hundreds, if not thousands, of functions. The complexity of OLE alone reaches or surpasses that of the Windows API. That's the bad news. The good news is that ActiveX is a dressed down version of OLE, made lighter for faster download times. There is also the new Active Template Library (ATL) that allows C++ programmers to develop simple, lightweight components with sizes rivaling JavaBeans components or Visual Basic ActiveX controls. ATL achieves small footprints by statically linking to your ActiveX component only those library functions that are absolutely indispensable. The problem is that you lose certain features, like dynamic memory management, support for GUI components, and others.

There are several other develop issues, like support for non-GUI components, layout policies, event handling, properties, persistence and packaging. Let's look at these one at a time.

Layout Policies

To make programs have a consistent look across platforms with different video resolutions, you really need to have some kind of code that makes decisions on window layout at runtime. JavaBeans use the JDK 1.1 layout manager classes. There are several of them, for the most common layouts. For complicated ones, you can usually get by with a series of nested panels -- each with its own layout policy. The IFC inherits layout managers from JDK 1.0.2, and adds a few extra capabilities.

ActiveX comes from the Windows world, and therefore knows nothing about runtime layout. ActiveX controls, like practically all controls in Windows, use screen locations that are established at design time. There are no layout managers available to generic windows. About the closest things I've seen to a layout manager in Windows are in dockable toolbars and the new Internet Explorer 4.0 `CoOlBar`. These windows can be reshaped at runtime using the mouse, and layout their child controls in a manner reminiscent of the Java `FlowLayout` and `GridLayout` policies. For ordinary windows, like dialog boxes, no layout policies in sight. Unless you create your own, you have to test your controls at different video resolutions by hand, making sure text isn't clipped and controls don't overlap. You also don't want to let users resize dialog boxes, because ActiveX controls don't automatically know how to adapt to varying window sizes.

Event Handling

JavaBeans supports event handling with a special *Event/Listener* model, which is very simple, efficient and adaptable. The idea is this: assume you have two objects. One is capable of generating events, the other of handling events. Say you have a button as the event source. You have a beeper object that you want to activate when the button is clicked. You setup the Beeper as an event listener, then add it to the button's list of listeners. Any time the button is clicked, it sends an event to each of its listeners. The technique is very flexible, because there are many types of events a object can be a listener for. The events are also grouped into categories, so you can create listeners for mouse events, keyboard events, and so on. Rather than receiving all events and handling only a small subset, a listener receives only those it has indicated interest in.

IFC does not use the JavaBeans event/listener model. It builds on the JDK 1.0.2 model, which requires object to override the `handleEvent()` method and process only those events it is interested in. This arrangement requires the Java environment to call your `handleEvent()` method for every event, even though your handler is only interested in a small subset of all events. As a result, `handleEvent()` is called many times unnecessarily, which reduces the event-handling bandwidth of the entire system.

ActiveX uses an event model based on the concept of event *sources* and *sinks*, similar conceptually to the JavaBeans model. When an event source fires an event, you can have another object process it by installing an *event sink*. ActiveX controls don't usually send events directly to other ActiveX controls. Normally an ActiveX control is placed in a window, which acts as an ActiveX control container. Events fired from controls are sent to the container window, which then takes appropriate actions or dispatches the events to other ActiveX controls. The coding aspect is rather complex, and you normally use a Wizard to create the low-level code. The whole process requires the writing of event dispatch maps, event sink maps, dispatch functions, and other code-intensive stuff. Wizards simplify the process considerably, but you still have to understand the overall picture.

Properties

Properties are attributes of an object. Properties are generally accessed at runtime, either by an external caller object, or by a script fragment, perhaps running in a browser. The beauty of properties is that they make it simple to change attributes, without needing to know anything about function calls, return types, or parameter ordering. JavaBeans support properties, not by an extension of the Java language, but through a simple method naming convention. Given an attribute that you wish to expose as a JavaBeans property, you create a *getter* and *setter* method to access it. For example:

```
Class MyClass {
    Color color;
    void setColor(Color theColor);
    Color getColor();
    // ...
}
```

The setter method has the prefix `set`, and takes a `Color` parameter. The getter has the `get` prefix and returns an object of type `Color`. Scripting languages that are JDK 1.1 compliant recognize method signatures for getters and setters and allow you to access the encapsulated properties using code like:

```
apple.Weight = 1.0
```

In the ActiveX world, there are two basic kinds of properties: stock and custom. Stock properties are standard Windows-types attributes for things like a control's caption, border style, background color and window handle. Custom properties are everything else.

Adding a property is not a completely trivial process, and is normally done using a wizard. The idea is to create an *access strategy* for the property. The simplest strategy grants direct access. Say you want to add a `Caption` stock property to your ActiveX control. Using the Visual C++ ClassWizard, you select the `Caption` property. The wizard adds a macro in your control's `Dispatch` map, like this:

```

BEGIN_DISPATCH_MAP(CMyActiveXControlCtrl, COleControl)
   //{{AFX_DISPATCH_MAP(CMyActiveXControlCtrl)
    DISP_STOCKPROP_CAPTION()
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

```

The wizard also adds the line

```
[id(DISPID_CAPTION), bindable, requestedit] BSTR Caption;
```

to the .odl file of your control. The line declares a property with the external name `Caption`, of type `BSTR` (an OLE string) and assigns a dispatch ID for users to access the property through.

Custom properties are harder to create. You first decide upon an access strategy. The simplest provides direct access to a property, which is equivalent to giving a data member public access in the OOP world. Other access strategies allow you to setup a notification function that is called when a property is changed. The most flexible strategy involves the use of `Get` and `Set` methods to access a property, much as in JavaBeans. A direct-access property can be created using a wizard. If you create a property called `Color` of type `long`, the entry

```
DISP_PROPERTY_NOTIFY(CMyActiveXControlCtrl, "Color", m_color, OnColorChanged, VT_I4)
```

must be added to your control's Dispatch map. Among other things, the macro defines a notification function that is called when the property is modified.

Creating a custom property called `Weight`, with full `set` and `get` functions requires the addition of the line:

```
DISP_PROPERTY_EX(CMyActiveXControlCtrl, "Weight", GetWeight, SetWeight, VT_R4)
```

To your control's Dispatch map. Using a wizard to create the entry, you'll also get skeleton definitions for the `get` and `set` functions, which will look something like this:

```

float CMyActiveXControlCtrl::GetWeight()
{
    // TODO: Add your property handler here
    return 0.0f;
}

void CMyActiveXControlCtrl::SetWeight(float newValue)
{
    // TODO: Add your property handler here
    SetModifiedFlag();
}

```

You have to provide a place to store the value of your property, since the wizard doesn't know what you want. Commonly, you simply add a data member, such as `fWeight`, to your ActiveX control class.

IFC posts a disappointing last place in the Properties category. Being built on top of JDK 1.0.2, the IFC has no notion of properties.

Persistence

JavaBeans uses JDK 1.1 features to support persistence. There are two ways to make an object persistent: the easy way, which operates at the language level, and the hard way, which requires your own code. The first requires you to simply make your component implement the `Serializable` interface. Built-in features in the Java environment

take care of reading and writing the individual fields of your component to/from a stream. The hard way is to take explicit control over serialization by making your component implement the `Externalizable` interface. You might want to do this if you want to serialize a component in an unusual way, perhaps compressing it or encrypting it. Using the default serialization technique, you get a tremendous amount of power with little effort. All it takes is the code:

```
FileOutputStream file = new FileOutputStream("MYFILE.SER");
ObjectOutput output = new ObjectOutput(file);
output.writeObject(new MyObject() );
```

IFC uses a different approach. Being built on top of JDK 1.0.2, it has no language support for serialization. To make an object persistent, you derive it from the class `Codable`. Writing an object to persistent storage is accomplished through a process called *encoding*. An `Archiver` class, which governs the operation, calls your class' `encode` method, passing it an `Encoder` object. Your class will use the encoder to write its fields as pairs of key/values. If your class had a field derived from `Codable` whose name was `title`, you would serialize it like this:

```
public class MyView extends View {
    Codable title;
    public void encode(Encoder encoder) throws CodingException {
        super.encode(encoder);
        encoder.encodeObject("title", title);
    }
}
```

A similar process is used to read objects from a stream.

ActiveX controls support persistence by implementing the OLE interfaces `IPersistStorage`, `IPersistStream`, `IPersistFile` and a few others. ActiveX container windows control the serialization of the ActiveX controls they contain by calling methods of the `IPersist` interfaces. It is the code you write that determines whether and how an ActiveX control saves its state. By default ActiveX controls are not persistent. There are no magical classes that provide automatic persistence. If you create ActiveX controls using MFC, then it's relatively easy to support persistence through the MFC-supplied classes. MFC uses a few classes, like `CArchive` and `COleStreamFile`, and a sprinkling of macros to make serialization easy. The MFC classes hide all the gruesome `IPersist` interfaces, allowing you to save your active control with code along these lines:

```
void CMyActiveXControl::Serialize(CArchive& theArchive)
{
    if (theArchive.IsStoring()) {
        theArchive << color;
        theArchive << weight;
        theArchive << caption;
    } else {
        theArchive >> color;
        theArchive >> weight;
        theArchive >> caption;
    }
}
```

The `CArchive` object is created by the MFC class `ColeControl`, like this:

```
HRESULT ColeControl::SaveState(IStream* theStream)
{
```

```

HRESULT hr = S_OK;
TRY {
    COleStreamFile file(theStream);
    CArchive archive(&file, CArchive::store);
    Serialize(archive);
}
CATCH_ALL(e) {
    hr = E_FAIL;
    DELETE_EXCEPTION(e);
}
END_CATCH_ALL

return hr;
}

```

Web Packaging

Components are used extensively in Web applications. Often Web pages contain features that require pre-installed components on the user machine. JavaBeans and ActiveX components have special packaging formats that make them easy to distribute over the internet, and easy to install on user machines.

The JDK 1.1 defined a new archiving format for Java code called *Java Archive*, or simply JAR. A JAR file uses compression technology, based on the popular ZIP format. JAR files allow you to combine one or more files into a single file that can be downloaded in a single http transaction. The file can be compressed or not. You can install JAR files into the Java CLASSPATH, and the Java system will treat the file like a directory containing its own files. JAR files contain a special file, called a *manifest* file, which describes the contents of the JAR. To create a JAR file, you use the JDK tool called `jar`, like this:

```
jar -cvf myjar.jar *.java
```

This command creates a JAR file called `myjar.jar` and adds to it all the `.java` files in the current directory. The `jar` command has all sorts of options, allowing you to disable compression, include a manifest file, get a listing of the contents of a JAR, etc. To extract files from a JAR, you use the `jar` command like this:

```
jar -xvf myjar.jar
```

Manifest files contain a list of the signed files in a JAR. The manifest may also contain a list of all the files in the JAR, but not necessarily. For each signed file, the manifest file lists the message digest algorithms used, and the digest itself. Assume the file `myjar.jar` contained in the signed file `debug.h`. The manifest file would have an entry looking like this:

```

Name: debug.h
Digest-Algorithms: SHA MD5
SHA-Digest: e9xcpF3C9iw9xHS03My+Dkc1Ao
MD5-Digest: tXv0W4Sk/011G31ZtDOBw==

```

The digest value is shown in base64 format. To sign a JAR file, you use the `javakey` utility, using a command line like this:

```
javakey -gs mySignature demo.jar
```

where `mySignature` is a file known as a *Certificate Directive* file, created separately using the `javakey` tool.

You can use JAR files with Java applets on web pages, to reduce the download time, using the HTML code:

```

<applet code=MyApplet.class
        archive="jars/myapplet.jar"

```

```

width=460 height=160>
  <param name=foo value="bar">
</applet>

```

ActiveX controls can also be packaged in a way similar to JavaBeans. ActiveX controls are normally distributed in special compressed files called *cabinets*, having the suffix .CAB. Cabinet files are created using a Microsoft tool called DIAMOND. When you create a .CAB file, you can create a .DDF (Diamond Description File) that describes the layout and contents of the cabinet to DIAMOND. A simple .DDF file looks something like this:

```

.OPTION EXPLICIT
.Set CabinetNameTemplate=MYCAB.cab
.Set DiskDirectoryTemplate=
.Set Cabinet=on
.Set Compress=on
.Set ReservePerCabinetSize=6144
MyFirstActiveXControl.ocx
AjavaBean.class
SomeOtherFile.txt

```

This .DDF file tells DIAMOND to create a .CAB file with the name MYCAB.CAB, to use compression and to add the files MyFirstActiveXControl.ocx, AjavaBean.class and SomeOtherFile.txt to it. You can add any number and type of files to a cabinet. To actually create the .CAB file, you need to install the Microsoft Cabinet Development Kit, and run DIAMOND like this:

```
DIAMOND /F MYCAB.DDF
```

Microsoft now provides an alternative way to create cabinets, bypassing the .DDF file. The command CABARC takes parameters from the command line and directly creates a cabinet. Using CABARC, you could pack all the files in a directory with the command:

```
cabarc n mycab.cab *.*
```

You can include a .INF file in the cabinet file to indicate how the contained files need to be extracted on the user's machine. The .INF file contains a whole bunch of fields, most of which are cut and pasted from a standard template file. A simple .INF file might look something like this:

```

[version]
signature="$CHICAGO$"
AdvancedINF=2.0
[Add.Code]
myactivexControl.ocx = MyActivexControlSection
ajavabean.class = AJavaBeanSection
someotherfile.txt = SomeOtherFileSection
[MyActivexControlSection]
file-win32-x86=thiscab
clsid={D030768D-BA7A-101A-B57A-0000C0C3ED5F}
FileVersion=1,0,0,0
RegisterServer=yes
[AJavaBeanSection]
FileVersion=4,20,0,6164
[SomeOtherFileSection]
FileVersion=4,2,0,6256
...
...

```

You can use a .CAB file to distribute a JavaBean class on a web page, using HTML code like this:

```
<APPLET CODE="MyJavaBean.class" WIDTH=100 HEIGHT=100>
<PARAM NAME="cabbase" VALUE="mycab.cab">
</APPLET>
```

or you can package an ActiveX control on an HTML document like this:

```
<OBJECT
  CLASSID="clsid:12345678-9abc-def1-1234567890ab"
  CODEBASE="cabs/mycab.cab#Version=1,0,0,12">
</OBJECT>
```

Browsers on Windows platforms, such as Internet Explorer, know how to take a .CAB file, examine its .INF file and extract all the files to their destination directories, updating the system registry where necessary. Cabinet files can also be signed, using the wizard provided in the Microsoft utility SIGNCODE, allowing a good degree of security for downloaded code.

So JavaBeans use JARs, ActiveX controls use cabinets. What does IFC use? Nothing in particular. The IFC Framework was not designed with a specific Web packaging technology. IFC controls can be distributed in a standard compressed file formats, such as .ZIP or tar.z, or using .CAB or .JAR files.

Conclusion

Okay, we have these three component architectures. The question in your mind now is, "*So which architecture is best?*". Unfortunately the answer is not simple. Few statements can be made categorically in life. For every pro there seems to be a con. Take ActiveX as an example. ActiveX was designed first and foremost for Microsoft Windows. ActiveX components contain native code compiled for the x86 platform. But ActiveX controls can also run on the Mac and soon on Unix. On the other hand, ActiveX is very fast, but running on non-Intel machines requires an interpreter, which kills the speed advantage of ActiveX. If you want true platform independence, don't use ActiveX. If your components only need to run on Windows, ActiveX is a reasonable (and possibly a great) choice. You can even take advantage of the Microsoft Java Virtual Machine and write code using a mixture of Java and ActiveX together.

In today's world, it is becoming increasingly important for code to run on several platforms. Programs are being distributed over the internet and the easiest way to steer clear of platform nightmares is to use 100 % pure Java solutions. Both IFC and JavaBeans are pure Java, and will run anywhere a Java VM is present -- which essentially means anywhere. The IFC is the precursor to the Java Foundation Classes, and can be considered a proprietary class library competing against JavaBeans. Because it was developed at the same time as JavaBeans, it overlaps Beans in several areas, making it difficult to use Beans and IFC at the same time. Look at how data transfer and persistence are implemented in the two models. Same results, but incompatible implementations.

My recommendation for programmers who want true platform independence is to use JavaBeans exclusively, and to take advantage of the extended features of the new Java Foundation Classes to give a world-class appearance to their user interfaces.