Chapter 1

TAKING CONTROL OF AUTOCAD

All consistent axiomatic formulations of number theory include undecidable propositions.
— Kurt Gödel

It is fitting, I suppose, that having begun my professional life as an architect, I continue to look for the elegance of structural details. Later in this book you will find this predilection reflected in examples demonstrating the use of AutoCAD objects. Buckminster Fuller's *vector equilibrium*, a deceptively simple structure defined by the close packing of spheres, is almost infinitely scalable in the form of the geodesic dome. In Chapter 9 we use it to demonstrate AutoCAD's *PolyfaceMesh* entity.

In his book *Gödel, Escher, Bach: An Eternal Golden Braid*, Douglas Hofstadter discusses *metamathematics* in relation to music, visual art, and computer programming. The common thread in his dialogues is that understanding such systems generally requires jumping out of them in order to view them from a higher level. This is suggestive of an ancient model of creation, in which this world of action is contained within another, that of formation. AutoCAD's object model is built upon a similar and ingenious organizational foundation, the *Component Object Model* (COM). In the domain that is the subject of this book, the center of that next-higher shell has the appropriate name *IUnknown*.

The development of integrated solutions in Visual Basic and VBA depends on COM. The AutoCAD 2002 object model is constructed according to the rules of COM, which provides the shell in which it operates. A short course in COM may not be what you signed on for here, but understanding its basic concepts is important if you really want to take control of AutoCAD and make it work with other applications. Its fundamentals are simpler than you might imagine.

Components and Automation

COM establishes a standardized means by which one piece of software can call upon another for services. A server application shares its objects with other applications. Conversely, a program that uses other applications' objects is called a *client application*. This sharing of objects is accomplished through the COM technology known as *Automation*.

As Figure 1-1 illustrates, AutoCAD and Excel can fill the role of either client or server in VBA. The *client* application is the one that is launched by the user, which then calls upon the objects in the *server* application through COM interfaces. The components from both object models are then executed *in-process* with the client application. We will see examples of both configurations in later chapters.

By contrast, a Visual Basic application executes in its own memory space but calls upon the object models of other applications through the same COM interfaces. This is not to say that a VBA procedure cannot access more than one server application. It can. But a VB application runs independently, *out-of-process*, as an EXE program. In either case the actual components are located in dynamic link libraries (DLLs) or ActiveX controls.

Since the focus of this book is on AutoCAD VBA macros, most of the examples we present are written in that context. Later in this chapter, though, we look at a short VB program that passes information from Excel into AutoCAD without either application ever being visible. First, however, let's delve a little more deeply into COM.

The Foundation

To understand how Automation works, we need to look above the AutoCAD object model (which is the subject of Chapter 4) to see how COM-enabled applications communicate.

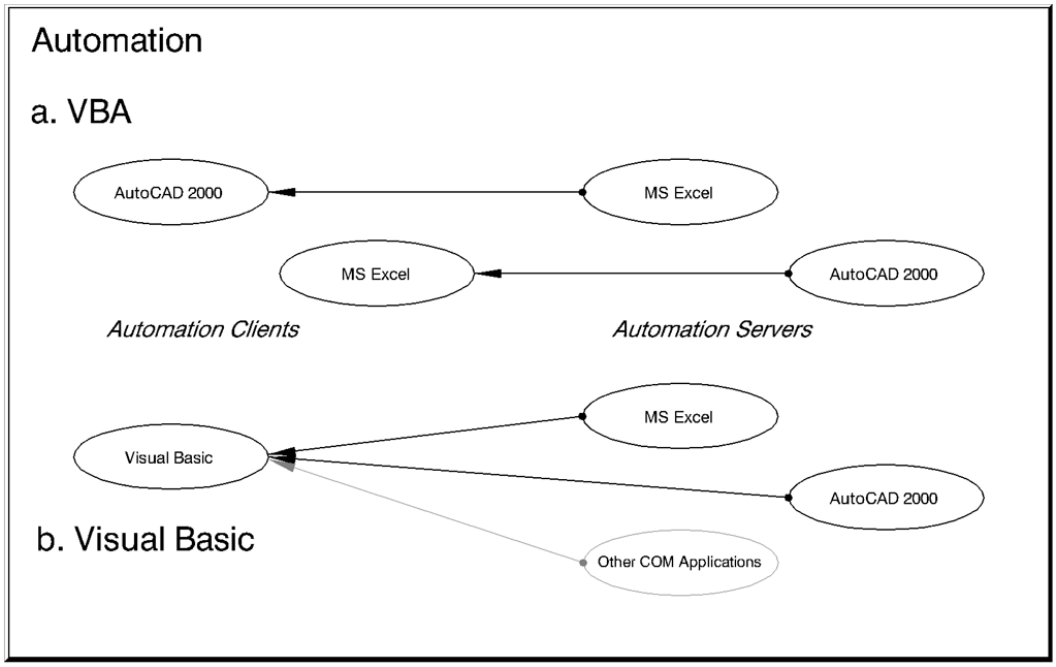

Figure 1-1
COM Automation

Characteristics of Objects

First, what is an *Object*? Objects are fundamentally regions of computer memory. A particular object is a specific region of memory with a name, a defined set of code and data (the object's attributes), and an interface. Each object is an *instance*, a specific occurrence of a (general) class. When an object is created, it is said to be *instantiated* from its class. Each object in C++, the language in which AutoCAD itself is now written, supports a single interface with a single set of methods. A COM object, on the other hand, has multiple interfaces, each set of which is identified by a different *Class*.

We speak of computer languages as being *object-oriented*. In addition to creating objects made up of methods and data, then organizing them according to classes, object orientation requires that three additional characteristics be present. *Inheritance* is one of them. COM objects support interface inheritance, which allows a child object to build on the features of the parent object, making them specific. In AutoCAD, for example, a *Line* is a special case of an *Entity*. But there is more to this hierarchy, as we shall see momentarily.

The second characteristic, *polymorphism*, allows a single object to appear in different guises at different times. COM allows Visual Basic objects to implement multiple interfaces, thus an *Entity* can be a *Line*, or it can be a *Circle*, or it can be a *PolyfaceMesh*! Moreover, COM provides for the evolution of software applications so that new functionality can be introduced without breaking old code.

The third defining characteristic of objects is *encapsulation*. The only way to access an object is through its methods, properties, or events. Methods are actions that you can tell the object to perform. Properties are characteristics that an object possesses, some of which you can set or modify. Events occur when an object changes its state, and you can create code that will execute when triggered by a specific event. The object's internal operation, however, is always concealed from the user in order to protect the object's data from being modified either accidentally or by design. (Visual Basic and VBA modules themselves implement another kind of encapsulation by defining procedures as being either public or private.)

Classes and Interfaces

But what, then, is a Class? A *Class* is a user-defined data type, an aggregation of standard data types (byte, double, string, etc.) used together for a specific purpose. COM classes are the means of defining *interfaces* with objects, complete with their own methods, properties, and events. An object's class defines whether the object is public and in what circumstances it can be created. *Type libraries*, the contents of which can be viewed using *object browsers*, are used to store descriptions of classes and their interfaces.

Automation Interfaces

An Automation interface, or simply *interface*, is a defined group of member functions through which clients communicate with component objects. It is important not to confuse the interface with the classes or objects themselves. An interface represents the functionality and expected behavior of a COM object in a definite (and permanent) manner. The uniqueness of each interface is guaranteed by its globally unique identifier (*GUID*), a 128-bit value assigned when the interface is initially defined. Once defined, interfaces are *never* changed. If a new version of an interface is required for whatever reason, a *new* interface is defined with its own *GUID*, and the old interface remains in place. Thus applications relying on the old interface can continue to function.

Binding

When you use an object in Visual Basic or VBA, you first declare it as an object and then create a reference to the object in an object variable. This process is known as *binding*. There are two types of binding, early and late, and as you might guess, late binding is the slower of the two. For example:

```
Dim xAP As Object
Set xAP = CreateObject("Excel.Application")
```

When a variable is declared simply *as object* or *as variant*, VB/VBA does not have enough information to determine at compile time what sort of object reference the variable will ultimately contain. This determination must be made at run time, hence the term *late* binding.

Early binding occurs when a specific type of object is specified in the declaration, as in the following code fragment:

```
Dim xAP as Excel.Application
Set xAP as Excel.Application
```

It follows, of course, that a variable declared as belonging to a specific class may only contain references to objects of that class. Whether object references are early or late bound is completely dependent on the way the variables are declared and has nothing to do with the manner of creating the objects. Use of early binding in creating the AutoCAD Application object is recommended, as the VB example later in this chapter shows.

Early binding is further subdivided into two types: *vtable* and *DispID*. Every property or method in a type library has a procedure identification number or *DispID* (dispatch identifier). *DispID* binding uses this number. If a component is represented in a type library but does not support *vtable* binding, VB uses the *DispID* during compilation to locate and bind the function.

With *vtable* binding, the fastest method, an offset address into a virtual function table provides direct access to the function. In general, if a client application declares object variables using explicit class names, *vtable* binding is assured. This is the method recommended in most circumstances and the one used by AutoCAD 2002. This is fortunate, because although you can control whether early or late binding is used by the way you declare object variables, the use of *vtable* versus *DispID* binding is controlled by the component object.

A High-Level View

Except when it comes to the question of bandwidth, it doesn't matter whether COM components are in the same place or on the other side of the planet. The terms *COM* and *DCOM* (*Distributed* COM) are often confused because the very concept of component implies distribution. Strictly speaking, COM becomes DCOM when network protocols replace local procedure calls. George Gilder, the pundit of the telecosm, tells us that soon we will enjoy infinite bandwidth at zero cost. Then it *really* won't matter!

Figure 1-2a illustrates an in-process client call with no intermediaries and therefore no overhead. Different processes that need to interact introduce some overhead because of the need to protect the processes from one another. This is the function of the operating system, which manages interprocess com-

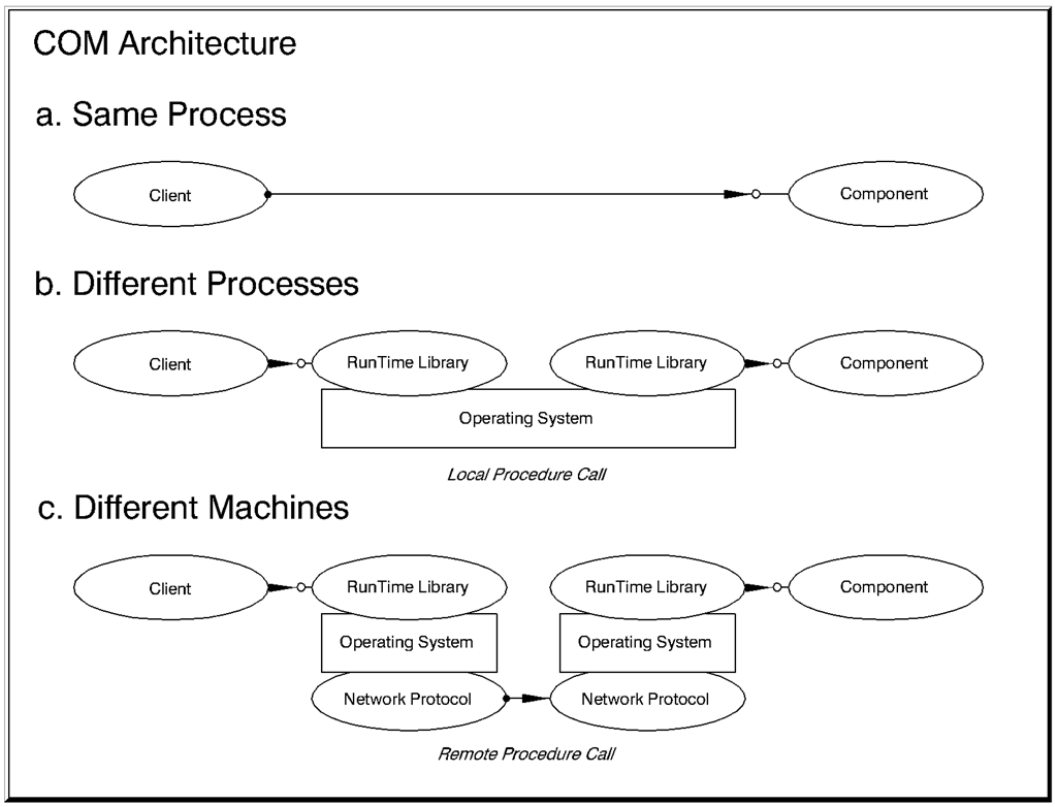

Figure 1-2
Component Object Model

munication through run-time libraries while providing the required shielding. Figure 1-2b shows this link as a local procedure call (LPC).

When the client and the components reside on different machines, the COM run-time uses the operating system's security provider, together with remote procedure calls (RPCs) to generate network packets in accordance with the DCOM wire-protocol standard. This arrangement is pictured in Figure 1-2c. The only essential difference between Figure 1-2b and c is the length of the connecting fiber.

Details

There are two COM interfaces above the AutoCAD object model that are essential to its operation: *IDispatch* and *IUnknown*. (Interface names begin with the letter *I* by convention.) These primary interfaces are located in your Windows\System subdirectory in a type library file called *StdOle2.tlb.*

Explicitly declared object variables provide access to an identification number called a procedure ID, or *DISPID*, for every property and method belonging to the object. AutoCAD's DISPIDs are found in its type library, *Acad.tlb*, and establish the necessary link to *IDispatch* through early binding. If an object variable is not explicitly declared, as an entity for example, without specifying what *kind* of entity, the method or property is accessed by name at run time, which is known as late binding.

IDispatch

All the interfaces in AutoCAD's object model except one, *IAcadObjectEvents*, inherit methods from the *IDispatch* interface that allow for late binding. If declared explicitly, they obtain type information from the *Acad.tlb* type library at compile time, supporting direct access through early *vtable* binding. For this reason they are said to *support dual interfaces.* As we have seen, the type of binding used is determined by the manner in which object variables are declared.

The *IDispatch* interface supports four methods:

1. GetTypeInfoCount
Retrieves the number of type information interfaces that the object provides (either 1 or 0); always 1 for AutoCAD objects.

2. GetTypeInfo
Retrieves the type information for an object, which can then be used to get the type information for an interface.

3. GetIDsOfNames
Maps a single member, along with an optional set of argument names, to a corresponding set of *DispIDs* (integers), which caches them for later use in subsequent calls to the *Invoke* method. *GetIDsOfNames* is used in late binding, when an *IDispatch* client binds to names at run time.

4. Invoke
Provides access to the methods and properties exposed by an object.

IUnknown

The *IUnknown* interface is quite literally the center of the COM universe. It allows clients to obtain pointers to other interfaces belonging to a given object and manages the existence of every object throughout its lifetime. All interfaces, including *IDispatch*, inherit from *IUnknown*, whose three methods constitute the uppermost entries in the *vtable* for all other interfaces. These three methods are as follows:

1. QueryInterface
Returns a pointer to the specific interface on an object to which a client currently holds an interface pointer. When a client accesses a component object to perform a function, all aspects of its internal behavior are hidden. Only through the interface pointer can the client access the functions exposed in the interface. It is this enforced encapsulation that enables COM to provide both local and remote transparency through an effective binary standard.

2. AddRef
Increments the reference count of calls to an object's interface.

3. Release
Decrements the reference count of calls to an interface on an object.

AddRef and *Release* together control the life spans of the objects in an executing program. This provides the mechanism by which, through inheritance, references to all components are dynamically resolved. These two methods simply maintain a count of the references to each component object while it is using the interface. As long as the reference count is greater than zero, the object must remain in memory. When the reference count decrements to zero, no other components reference the object, which can then unload safely.

```
 [
  odl,
  uuid(00000000-0000-0000-C000-000000000046),
  hidden
 ]
interface IUnknown
{[restricted] HRESULT _stdcall QueryInterface
                     ([in] GUID* riid,
                      [out] void** ppvObj);
 [restricted] unsigned long _stdcall AddRef();
 [restricted] unsigned long _stdcall Release();
};
```

Example 1-1. IUnknown

Example 1-1 illustrates *IUnknown*. It is written in *IDL* (Interface Development Language), which looks something like C/C++ and nothing like Visual Basic. A distinguishing feature is the use of *attributes*, which are the keywords in square brackets that specify the characteristics of the interface together with the data and methods within. The standard format in *IDL* begins with a header containing the interface attributes followed by the body of the interface enclosed in braces (curly brackets).

The most significant part of *IUnknown's* header is its *UUID*, the universally unique identifier (same as *GUID*). This is its 128-bit ID in the form of a five-node string comprised of 8 hexadecimal digits followed by three groups of 4 digits, and finally 12 digits. *Hidden* suppresses the display of the item in an object browser, as it cannot be accessed directly. The ODL attribute is no longer required but remains for backward compatibility.

The body of *IUnknown* contains the declarations of the three remote procedures in the interface, along with their data types. The *restricted* keyword specifies that a method cannot be called arbitrarily. In the *QueryInterface* method, *riid* is the requested interface identifier of the client program passing data into the remote procedure. *ppvObj* contains the address of the pointer variable requested in *riid*, which is passed out of the remote procedure.

So What Does this Mean to AutoCAD?

Earlier we pointed out that all except one of the interfaces in AutoCAD's object model inherit methods from the *IDispatch* interface, which in turn inherits from *IUnknown*. Figure 1-3 uses the *IAcadObject* interface to illus-

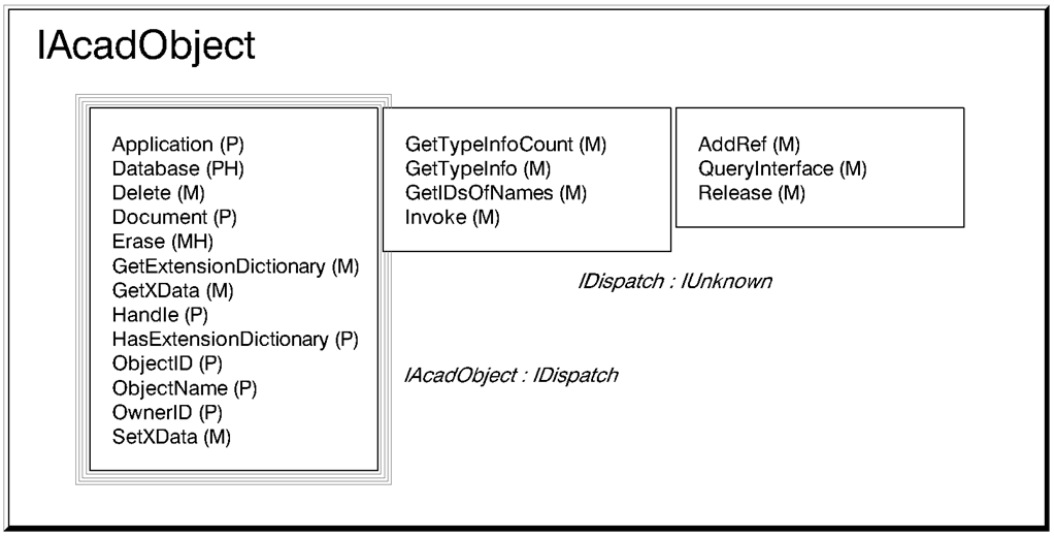

Figure 1-3
Object Inheritance

trate how AutoCAD objects inherit the necessary methods that allow them to interoperate through COM.

IAcadObject inherits *IDispatch*'s four methods along with the three belonging to *IUnknown*. These methods are thus passed down to all the objects that inherit from *IAcadObject*. There are 13 methods and properties to which *IAcadObject* provides direct access, of which you will see only 11 in the VBA object browser. The method and property designated with an *H* are hidden, meaning that they serve internal functions in AutoCAD and are not directly accessible. The *Database* property returns the database object, and the *Erase* property erases the entity object. These member functions, which are essential to maintaining the AutoCAD database, are inherited by virtually all the objects in AutoCAD along with the visible ones.

Creating a Drawing with Visual Basic

Before we complete our discussion of COM, let us see how VB and COM can be used to pass data from one application to another. Example 1-2 is a short but interesting VB procedure that reads some text data from an Excel

```
Sub Main()
    Dim xAP As Excel.Application
    Dim xWB As Excel.Workbook
    Dim xWS As Excel.Worksheet
    Set xAP = Excel.Application
    Set xWB = xAP.Workbooks.Open("C:\A2K2_VBA\IUnknown.xls")
    Set xWS = xWB.Worksheets("Sheet1")
    MsgBox "Excel says: ""  & Cells(1, 1) & """

    Dim A2K As AcadApplication
    Dim A2Kdwg As AcadDocument
    Set A2K = CreateObject("AutoCAD.Application")
    Set A2Kdwg = A2K.Application.Documents.Add
    MsgBox A2K.Name & " version " & A2K.Version & _
         " is running."

    Dim Height As Double
    Dim P(0 To 2) As Double
    Dim TxtObj As AcadText
    Dim TxtStr As String
    Height = 1
    P(0) = 1: P(1) = 1: P(2) = 0
    TxtStr = Cells(1, 1)
    Set TxtObj = A2Kdwg.ModelSpace.AddText(TxtStr, _
         P, Height)

    A2Kdwg.SaveAs "C:\A2K2_VBA\IUnknown.dwg"
    A2K.Documents.Close
    A2K.Quit
    Set A2K = Nothing

    xAP.Workbooks.Close
    xAP.Quit
    Set xAP = Nothing
End Sub
```

Example 1-2. Visual Basic Automation

spreadsheet, creates an AutoCAD drawing, places the text into the drawing, and then saves the drawing. What is interesting is that except for the two message boxes that are included to let you know when Excel and AutoCAD are running, neither application is ever visible. Before you can run this program, you will need to create an Excel worksheet with some text in the first cell (A1), then save the workbook to a desired location and exit from Excel. (Any text is OK, but "Hello, World!" is the traditional choice.)

Setting Available References

In Visual Basic 6.0, which we are using for Example 1-2, you set references to the available type libraries using a dialog box accessed through the Project→References menu. The VB *References* dialog is illustrated in Figure 1-4. [In AutoCAD VBA you use an identical dialog box, but it is accessed using the Tools→References menu in the Visual Basic Editor. AutoCAD's Interactive Development Environment (IDE) is discussed in Chapter 2.] For our example procedure, we need references to the AutoCAD 2000 and Microsoft Excel type libraries, which have been checked as shown in Figure 1-4.

Writing the Procedure

The procedure, which is called *Main*, is in five paragraphs. (Normally, a VB project has a startup form. We do not use any forms in this example, so VB requires that the *Sub* procedure be called *Main*.) The first paragraph sets up Excel. You need to declare (`Dim` statements) three object variables for the Excel components you are going to call: *Application*, *Workbook*, and *Worksheet*. Then the `Set` statement is used to assign an object reference to each variable. Setting *xAP* invokes Excel. Setting *xWB* calls the Excel *Workbooks* property with the *Open* method to open a file containing the desired spreadsheet. Here we've used the name *C:\IUnknown.xls*. You should change this to the path and file name of the Excel Workbook you created earlier. Setting *xWS* to the name of the specific worksheet (using the default name *Sheet1*) completes this sequence. The standard VB/VBA message box (`MsgBox` will become very familiar throughout the course of this book) presents a dialog box that displays the text being processed.

The second paragraph does the same thing for AutoCAD. *A2K* is the object variable for the AutoCAD application object (*AcadApplication*) and *A2Kdwg* names the instance of *AcadDocument*. (We discuss *Application*, which is the

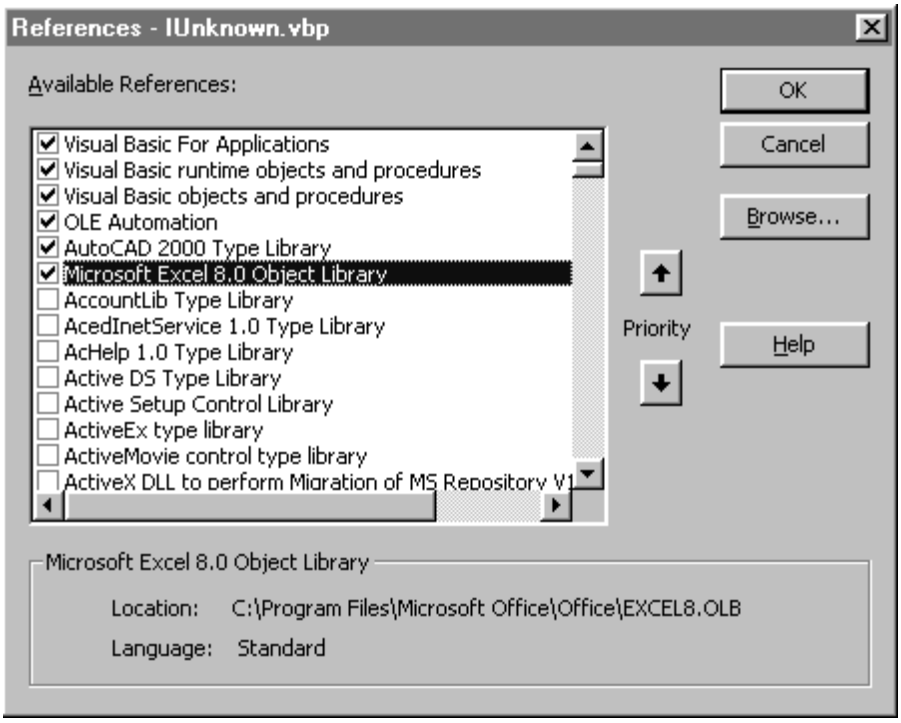

Figure 1-4
References Dialog

root object of AutoCAD's object model, together with the *Documents* collection and *Document* object in Chapter 4.) Using the `Set` statement once again, we instantiate the *AutoCAD.Application* and create a new drawing using the *Add* method on the *Documents* collection. A second *MsgBox* tells us that AutoCAD is now running.

The actual work is done in the third and central paragraph of our VB program, where we declare the variables required for creating a *Text* object in AutoCAD. (The *Text* object, along with AutoCAD's other two dozen graphic objects, or entities, is covered in Chapter 9.) Briefly, we need numeric variables to specify the height of the text and the point at which it is to be inserted (an array for its *X*, *Y*, and *Z* coordinates). Also, we need an object variable for the *Text* object and a string to contain the text itself. We specify the height and the insertion point by assigning values to *Height* and the *P( )* array. Then we assign the string value of *Cells(1, 1)* to *TextStr* using Excel's *Cells* method, and finally, we set the *Text* object (*TxtObj*) using AutoCad's *AddText* method. We leave *ModelSpace* as a little mystery for the time being.

One major difference between VB and VBA that we might mention at this point is use of the *A2Kdwg* object variable to represent the active drawing. In VBA the active drawing can always be referred to as *ThisDrawing*. This is not the case in VB, where you must define explicit variables for the AutoCAD application and the current document.

In the fourth paragraph we name and save our new drawing in the desired location using AutoCAD's *SaveAs* method. Here we've used the name *C:\IUnknown.dwg*. We then *Close* this single new member of the *Documents* collection and *Quit* the application. In the final paragraph we use the corresponding Excel methods to do the same thing to our spreadsheet. In both cases, setting the application object variables to *Nothing* releases the memory allocated to those processes.

More About Components

In Chapter 4 we begin to examine AutoCAD's object model in detail, working our way from the *Application* object on down. First, however, we need to complete the picture of how AutoCAD fits into the overall COM structure. We conclude this chapter by introducing the tables of object methods and properties (and events) that conclude half the chapters in this book.

AutoCAD Inheritance

As we mentioned earlier, all AutoCAD objects inherit from *Idispatch*, with the exception of *IAcadObjectEvents*, which inherits directly from *IUnknown*. *IAcadObjectEvents* is the event interface for AutoCAD entities, which occurs whenever an object is modified (the *Modified* event). There are two other interfaces in the AutoCAD type library that handle events: *_DAcadApplicationEvents* and *_DAcadDocumentEvents*. These are the event interfaces for *AcadApplication* and *ThisDrawing*. They are called *dispinterfaces*, which means that they define a set of methods and properties (not in the *vtable* for the object) that invoke a response to designated events. (In Chapter 13 we explore AutoCAD events in detail.)

IAcadIdPair and IAcadState

The *IAcadState* interface supports a special object (*AcadState*) that can be used to monitor the state of AutoCAD from out-of-process applications (i.e.,

VB). The *IsQuiescent* property or the *GetAcadState* method of the *Application* object can be used to check to see whether AutoCAD is idle and ready to accept automation calls. The *IAcadIdPair* interface, which returns information pertaining to the process of copying objects, is discussed in Chapter 12.

IAcadEntity and IAcadObject

Of particular interest to us in traversing AutoCAD's object model is the *IAcadObject* interface, which is the parent of just about everything else. All nongraphic objects are direct descendants of *IAcadObject*, as is *IAcadEntity*. All graphic objects descend from *IAcadEntity*, some, such as dimensions, through other intermediate classes of objects. We have included *IAcadDimension* in Figure 1-5 because it contains a unique set of dimensional entities. *Blocks* and *BlockReferences* are also major subclasses that can contain *all* of AutoCAD's entities. *Blocks* are the subject of Chapter 8, and *Dimensions* are covered in Chapter 11.

Method, Property, and Event Matrices

The matrices shown in Figures 1-6 and 1-7 are the first of 10 sets that designate *all* the methods, properties, and events belonging to the objects treated in their respective chapters. The matrices for this chapter include the *General* methods and properties belonging to the objects discussed above.

As a graphic aid to finding detailed information pertaining to each method and property, the left-hand column of each matrix contains a bullet if the item is treated in that chapter. From there it should be relatively easy to locate the item based on the object or objects to which it belongs. (All events are discussed in Chapter 13.)

To make it easy to follow the inheritance of objects in the object model, different symbols are used to designate methods and properties belonging to major subclasses of *IAcadObject*. A solid bullet (●) indicates that a method or property is provided by the interface supporting that specific object. Table 1-1 provides descriptions of the symbols used.

Hidden methods or properties are those that are marked as such in the AutoCAD type library and therefore do not normally appear in the VBA object

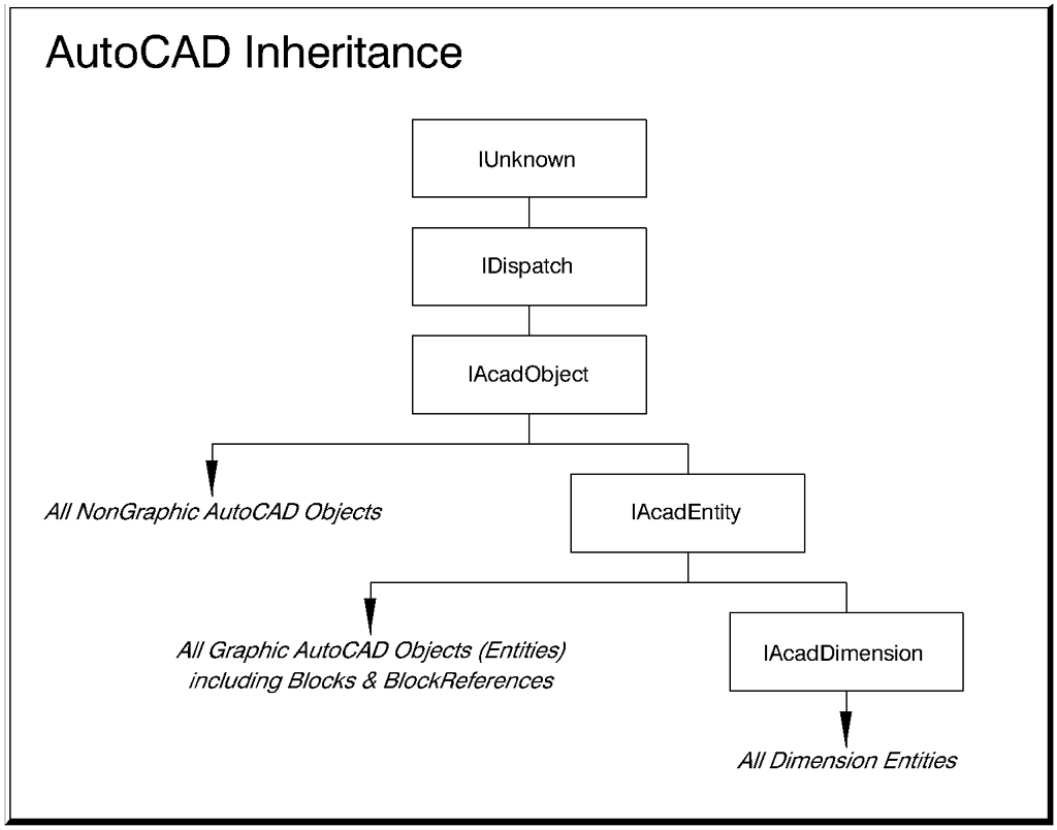

Figure 1-5
AutoCAD Inheritance

browser. Methods and properties suppressed by AutoCAD appear in the object browser but cannot actually be invoked upon those objects. The best example of this is the *Delete* method, which is inherited from the *IAcadObject* interface but cannot be used with any of the intrinsic AutoCAD collections because they cannot be deleted.

In Figure 1-6 you can clearly see the beginning of the inheritance from *IAcadObject* to *IAcadEntity* and *IAcadDimension*.

METHODS	IAcadDimension	IAcadEntity	IAcadIdPair	IAcadObject	IAcadObjectEvents	IAcadState
ArrayPolar	▲	●				
ArrayRectangular	▲	●				
Copy	▲	●				
Delete	○	○		●		
Erase	×	×		×		
GetBoundingBox	▲	●				
GetExtensionDictionary	○	○		●		
GetXData	○	○		●		
Highlight	▲	●				
IntersectWith	▲	●				
Mirror	▲	●				
Mirror3D	▲	●				
Move	▲	●				
Rotate	▲	●				
Rotate3D	▲	●				
ScaleEntity	▲	●				
SetXData	○	○		●		
TransformBy	▲	●				
Update	▲	●				

Figure 1-6
General Methods

PROPERTIES	IAcadDimension	IAcadEntity	IAcadIdPair	IAcadObject	IAcadObjectEvents	IAcadState
Application	○	○	●	●		●
Color	▲	●				
Database	×	×		×		
DecimalSeparator	●					
Document	○	○		●		
EntityName	×	×				
EntityType	×	×				
Handle	○	○		●		
HasExtensionDictionary	○	○		●		
Hyperlinks	▲	●				
IsCloned			●			
IsOwnerXlated			●			
IsPrimary			●			
IsQuiescent						●
Key			●			
Layer	▲	●				
Linetype	▲	●				
LinetypeScale	▲	●				
Lineweight	▲	●				
Normal	●					
ObjectID	○	○		●		
ObjectName	○	○		●		
OwnerID	○	○		●		
PlotStyleName	▲	●				
Rotation	●					
ScaleFactor	●					
StyleName	●					
SuppressLeadingZeros	●					
SuppressTrailingZeros	●					
TextColor	●					
TextGap	●					
TextHeight	●					
TextMovement	●					
TextOverride	●					
TextPosition	●					
TextPrefix	●					
TextRotation	●					
TextStyle	●					
TextSuffix	●					
ToleranceDisplay	●					
ToleranceHeightScale	●					
ToleranceJustification	●					
ToleranceLowerLimit	●					
TolerancePrecision	●					
ToleranceSuppressLeadingZeros	●					
ToleranceSuppressTrailingZeros	●					
ToleranceUpperLimit	●					
Value			●			
VerticalTextPosition	●					
Visible	▲	●				

Figure 1-7
General Properties

Table 1-1 Matrix Symbols

Symbol	Interface Name or Description
●	Locally defined method or property
■	IacadBlock
▼	IAcadDimension
▲	IAcadEntity
❍	IAcadObject
◆	IAcadPlotConfiguration
❑	IAcadBlockReference
◗	IAcadDatabase
❖	IAcadIdPair
✻	IAcadState
×	Hidden method or property
⊗	Method or property suppressed by AutoCAD

Summing Up...

In this chapter we have gained an understanding of Microsoft's Component Object Model (COM), which is the mechanism that allows AutoCAD to communicate with other applications using ActiveX Automation interfaces. COM is the foundation upon which Visual Basic and VBA are built. Following the rules of ActiveX, you can design macros that operate entirely within AutoCAD or share information and processes with other programs as either client or server applications.

In Chapter 2 we look at the two types of VBA projects in AutoCAD and its development environment. Although one of the most attractive features of VBA is its standardization among different applications, there are some features that are unique to AutoCAD. These features relate principally to the ways in which you can manage projects and invoke macros from the CAD interface.