Cincom®

Cincom Smalltalk™

COM Connect User's Guide

P46-0123-05

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0123-05

Software Release 7.7

This document is subject to change without notice.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

Programs that use or publish objects adhering to Microsoft Component Object Model (COM) interface standards are able to interact with other COM-based applications independent of operating system or programming language restrictions.

This guide describes how to use VisualWorks COM Connect to access or publish COM objects from within VisualWorks applications. It also provides procedural and reference information for using COMAutomation (formerly called OLE Automation) with VisualWorks.

Audience

This guide is intended for experienced VisualWorks application developers who want to access or publish COM objects from within VisualWorks applications. To get the most out of this document, you should also be familiar with the Microsoft Windows 95/98 and NT programming environments, and with COM and Automation concepts.

Organization

This guide is organized as follows:

- This preface describes conventions used in this guide, related documents that might be helpful, and instructions for contacting Cincom if you need assistance.
- Chapter 1, COM Connect Basics introduces COM concepts and describes the basics of VisualWorks COM Connect facilities.
- Chapter 2, Using COM Objects describes how to write VisualWorks code capable of interacting with COM objects.
- Chapter 3, Implementing COM Objects describes how to provide access to your application through interfaces used by external COM clients.

- Chapter 4, COM Infrastructure Support describes COM infrastructure technologies supported in COM Connect.
- Chapter 5, COM Connect Development Tools describes tools for developing and debugging COM applications in Smalltalk, as well as tools for COM developers that are freely available from Microsoft.
- Chapter 6, Using Automation Objects describes how to create and access Automation objects, and how to access methods and properties of a particular Automation object through its dispatch interface (the *IDispatch* interface).
- Chapter 7, Using ActiveX Controls describes adding and configuring AciveX components for use in an application GUI.
- Chapter 8, Implementing Automation Objects describes the facilities and frameworks available for publishing externally accessible COM Automation objects.
- Chapter 9, Publishing Automation Objects discusses run-time image preparations for a published object, and object server application testing methods.
- Chapter 11, Exposing Classes Through Dual Interfaces describes how to implement dual interfaces for exposed ActiveX objects in your application.
- Chapter 12, Using Distributed COM describes how to support communication among objects on different computers—on a LAN, a WAN, or the Internet.
- Chapter 13, Automation Controller Framework describes the framework that provides Smalltalk wrappers for standard Automation objects, as well as abstract classes you can subclass to add support for other Automation objects.
- Chapter 14, Standard Automation Objects and Naming Guidelines describes the standard ActiveX objects, and discusses naming guidelines for creating and using objects that are unique to applications, especially user-interactive applications that support a multiple-document interface (MDI).
- Chapter 15, Under the Hood presents additional information that is not essential to learning how to use VisualWorks COM Connect Automation classes but helps you understand COM Connect technology.

- Chapter 16, COM Connect Server Examples provides examples of how to publish COM Automation objects.
- The Glossary defines many of the terms used in this guide.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.

Examples	Description
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Note: This is a different arrangement from how VisualWorks used the middle and right buttons prior to 5i.2.
If you want the old arrangement, toggle the **Swap Middle and Right Button** checkbox on the **UI Feel** page of the Settings Tool.

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help** > **About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help** > **About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe". You can then address emails to vwnc@cs.uiuc.edu.

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://www.cincomsmalltalk.com/CincomSmalltalkWiki

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Documentation

In addition to this guide, the following documents (available at bookstores) might be useful when developing interoperable VisualWorks applications:

Note: Web URLs referenced in this document were valid at the time of publication; however, web page references are volatile and therefore some of these links may become obsolete over time. If you find an obsolete link, please email **publications@cincom.com**.

Books

- *Inside COM*. Dale Rogerson. Microsoft Press.
  ISBN 1-57231-349-8.

  This book provides an excellent introduction for the programmer to the basic COM architectures and concepts.

- *Inside OLE*, Second Edition. Kraig Brockschmidt. Microsoft Press.
  ISBN 1-55615-843-2.

- *Understanding ActiveX and OLE*. David Chappell.
  ISBN 1-57231-216-5.

  This book provides a high-level overview of COM technology.

- *OLE2 Programmer's Reference*. Volume 1 (COM and OLE). Volume 2 (Automation). Microsoft Press.
  ISBN 1-57231-584-9.

- *Professional DCOM Programming*. Richard Grimes.
  ISBN: 1-86100-060-X.

- *Creating Components with DCOM and C++*. Don Box.
  ISBN 13: 9780614284423

- *The Underground Guide to Microsoft Office, OLE and VBA*. Lee Hudspeth, Timothy-James Lee. Addison Wesley Publishing Company. ISBN 0-201-41035-4.

COM Specification

http://msdn.microsoft.com/en-us/library/ms694363(VS.85).aspx

Resource Links

- The ActiveX Working Group.

  http://www.activex.org

- Cetus Links.

  http://www.objenv.com/cetus/oo_ole.html

DCOM and CORBA

- Comparing ActiveX and CORBA/IIOP.

  http://www.omg.org/library/activex.html

- Mappings and interoperability.(Chapter 13)

  ftp://ftp.omg.org/pub/docs/ptc/96-08-04.ps

1

COM Connect Basics

VisualWorks COM Connect provides support in Smalltalk for COM and related fundamental technologies based on COM. COM Connect supports:

- Basic COM functionality, including call-out and call-in of interface functions
- Distributed COM (DCOM)
- COM object server application delivery
- COM structured storage
- COM clipboard data transfer
- Automation
- COM events (connectable objects)
- Embedding Active-X Controls in a VisualWorks Applications

Examples are provided that demonstrate both using and implementing COM and Automation objects. A complete VisualWorks object server example is provided to demonstrate how to publish COM objects, and includes VisualBasic, Visual C++, and Java clients to demonstrate interoperable server development.

The Component Object Model

COM is a system object model that enables modular system construction and reliable application integration. COM is widely used as the basis of many features in the Windows family of operating systems and is the foundation of a number of technologies.

COM provides functions that enable you to build components that are distributed, and reusable.

Distributed COM (DCOM), discussed under Using Distributed COM, supports communication between clients and components located on different computers. This communication is identical to that between clients and components residing on the same computer.

Automation builds on top of COM to enable scripting tools and applications to manipulate objects that are exposed on Web pages or in other applications. Other technologies derived from COM include ActiveDirectory, OLE Messaging, Active Controls, Active Data Objects, ActiveX Scripting, Web Browsing.

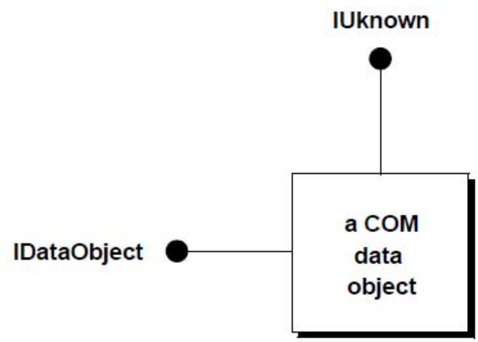
Objects and Interfaces

In COM, an object supports one or more interfaces. Each interface is a collection of functions that provide a related set of services to clients of the object. An interface is a collection of typed function signatures, representing a contract between a client and a server.

A number of standard interfaces are defined for common services and COM object implementors are encouraged to support existing interfaces where appropriate. COM object implementors can also define new interfaces as needed to publish the services of their server objects.

As shown in the following figure, an interface is uniquely identified by an interface id, or IID, which clients use to obtain an interface from a COM object. Interfaces are also referred to by a common name, which by convention is prefixed by the uppercase letter “I” to denote an interface. A function in a particular interface is discussed using the interface name and the function name together. For example, the `QueryInterface` function in the standard `IUnknown` interface is referred to by the `IUnknown::QueryInterface` notation.

Diagramming a COM Object



An important characteristic of the COM architecture is that a COM object can only be manipulated by clients by referencing its available interfaces. Clients only obtain interfaces, never direct references to an object, so a COM object is entirely encapsulated. If you consider the data object depicted in the above figure, a client can obtain references to the IUnknown or IDataObject interfaces supported by the object, but never has a reference directly to the object itself. In COM, only interfaces are real.

Publishing COM Objects

A COM object is published by registering information about its object class with COM. Published COM objects are identified by a class ID, commonly referred to as the CLSID. A COM object class can also be identified by its program ID, or PROGID, which is a short string name that identifies the application in the registry.

A published COM object class is supported by a class factory. A class factory is an object used by COM or the client to create new instances of the published object class. The IClassFactory interface contains a CreateInstance function, which allows clients to manufacture new objects.

Many COM identifiers, such as CLSID and IID, are GUID (Globally Unique IDentifier) values. A GUID is a 16-byte value that is guaranteed to be unique across any machine. Anyone can allocate a new GUID, which enables COM developers to independently publish new classes and interfaces with confidence that the identifiers they use are unique.

Clients use COM objects by obtaining interfaces and invoking functions, then releasing interfaces when they are done using their services. Clients can create new instances of published server objects using standard capabilities provided by COM. In some cases, COM objects can also be obtained from other COM objects already in use by a client. Typically, such dependent or related objects are obtained by invoking some COM interface function that is defined to return an interface from another COM object.

Distributed COM (DCOM) technology extends the existing COM architecture by providing network communication capabilities from the existing model to enabled distributed object applications. DCOM extends the basic concepts of objects and interfaces to the domain of distributed object applications. For information on using DCOM with an application, see Using Distributed COM.

COM Applications

COM applications are either clients or servers of COM objects, or both. COM server applications create and maintain objects. COM client applications are consumers of these objects. Many COM applications have both roles, in that they both use COM objects provided by other applications and implement COM objects themselves.

Each COM object is created and maintained by an object server application, which can implement one or more COM object classes. A class factory is supported for each COM object class that can be created independently by clients.

COM object server applications can be developed and written independently in any language. By using COM and COM-based technologies, you can integrate the services of different server applications with your application.

Learning More About COM

While this documentation is intended to provide the basic information needed to begin developing COM applications in VisualWorks Smalltalk, it does not provide detailed information about advanced topics. To supplement this material, refer to the widely available sources about COM.

A good introduction to COM and COM-based technologies is David Chappel's *Understanding ActiveX and OLE*. While much of his book is dedicated to discussing the OLE compound document architecture and the ActiveX controls technology, the book also provides a good introduction to the basic concepts of COM and the more fundamental COM-based technologies, such as Automation. It also discusses DCOM, which many view as a competitor to the Common Object Request Broker Architecture (CORBA) distributed object standard.

An excellent introductory text for programmers, specifically with regard to COM architecture and fundamental COM mechanisms, is provided by Dale Rogerson's Inside COM. This book is more technical and focused than Chappell's book.

The standard programmer's introductory text used by many is Kraig Brockschmidt's *Inside OLE*. This material is aimed at C and C++ programmers and provides a more detailed understanding of specific topics. However, much of this book is focused on the OLE container architecture rather than COM facilities.

More detailed programmer documentation is available in *Microsoft's OLE Programmer's Reference* manuals and in the programming tools provided by the Windows SDK. Other material is available from the MSDN Web site: http://msdn.microsoft.com.

COM Automation Basics

This section is an adaptation of an overview of Automation that can be browsed on Microsoft's *Developer Network Library Visual Studio 97* CD and on Microsoft's Web site.

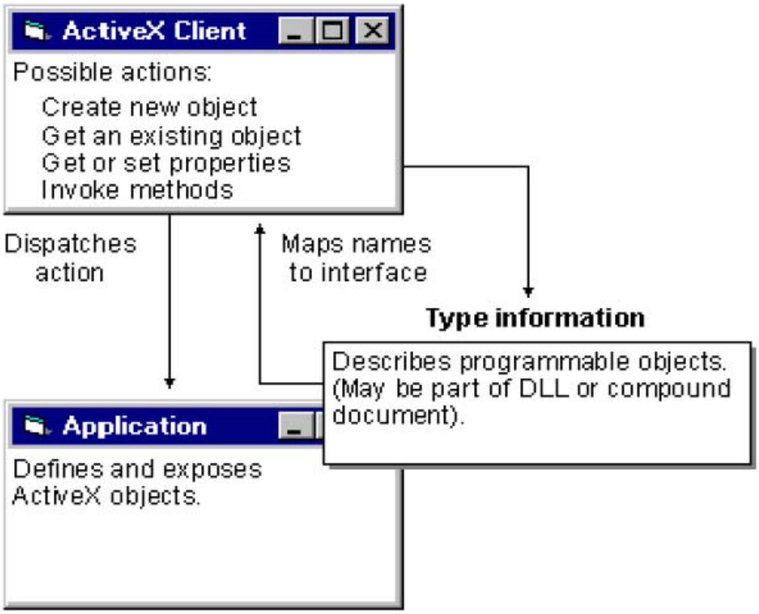
Overview of Automation

Automation (formerly called OLE Automation) is a technology that allows software packages to expose their unique features to scripting tools and other applications. Automation uses the Component Object Model (COM), but can be implemented independently of other COM-based technologies, such as the OLE container architecture or ActiveX controls. Using Automation, you can:

- Create applications and programming tools that expose objects.
- Create and manipulate objects exposed in one application from another application.
- Create tools that access and manipulate objects. These tools can include embedded macro languages, external programming tools, object browsers, and compilers.

The objects an application or programming tool exposes are called COM or ActiveX objects. Applications and programming tools that access those objects are called COM or ActiveX clients. ActiveX objects and clients interact as shown in the following figure:

ActiveX objects and clients



Applications and other software packages that support COM and ActiveX technology define and expose objects that can be acted on by COM and ActiveX components. COM and ActiveX components are physical files (for example .exe and .dll files) that contain classes, which are definitions of objects. Type information describes the exposed objects, and can be used by COM and ActiveX components at either compile time or runtime.

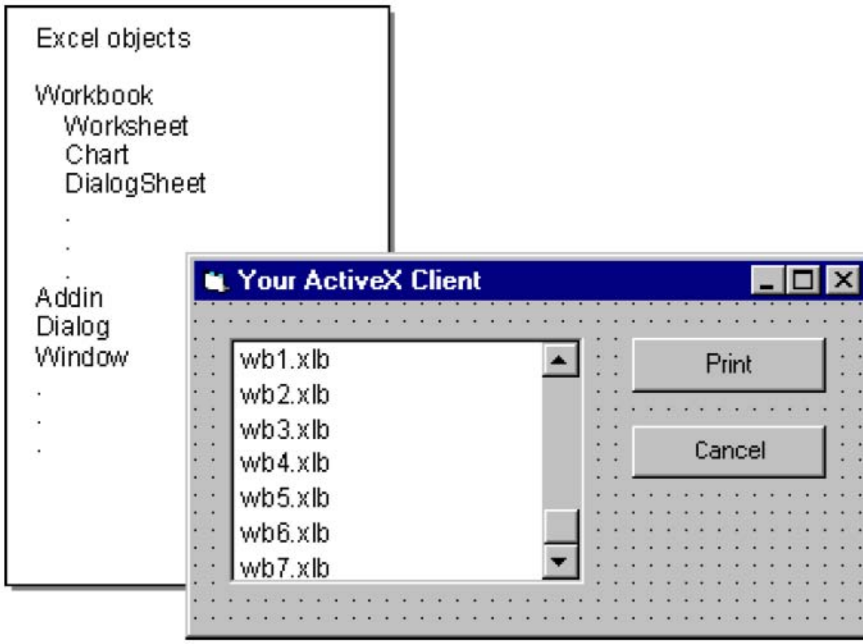
Why Expose Objects?

Exposing objects through Automation provides a way to manipulate an application's tools programmatically. This allows customers to use a programming tool that automates repetitive tasks that might not have been anticipated.

For example, Microsoft® Excel® exposes a variety of objects that can be used to build applications. One such object is the Workbook, which contains a group of related worksheets, charts, and macros;

the Microsoft Excel equivalent of a three-ring binder. Using Automation, you could write an application that accesses Microsoft Excel Workbook objects, possibly to print them, as in the figure below:

Accessing objects from an application



With Automation, solution providers can use your general-purpose objects to build applications that target a specific task. For example, you could create a general-purpose drawing tool to expose objects that draw boxes, lines, and arrows, insert text, and so forth. Another programmer could build a flowchart tool by accessing the exposed objects and then adding a user interface and other application-specific features.

Exposing objects to Automation or supporting Automation within a language offers several benefits:

- Exposed objects from many applications are available in a single programming environment. Software developers can choose from these objects to create solutions that span applications.
- Exposed objects are accessible from any macro language or programming tool that implements Automation. Systems

integrators are not limited to the programming language in which the objects were developed. Instead, they can choose the programming tool or macro language that best suits their own needs and capabilities.

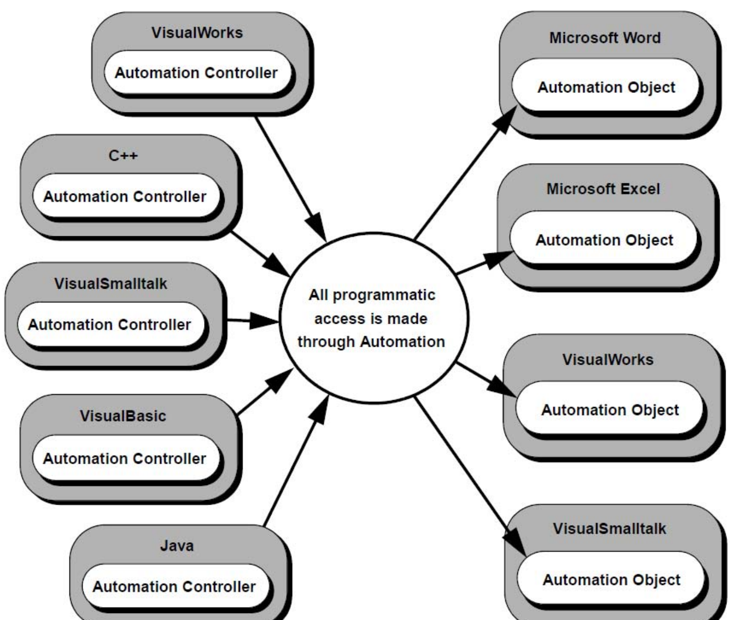
- Object names can remain consistent across versions of an application, and can conform automatically to the user's national language.

Automation = Cross-Application Macros

Automation allows programming languages and other tools running a script of some sort to access and manipulate objects without having compile-time knowledge of table layouts. Such tools, called automation controllers, can create objects from any components or applications as necessary, thus enabling end users or developers to write cross-application macros.

As shown in the following figure, a controller can gain access from VisualWorks, Visual, C++ or VisualBasic, through Automation to the exposed objects of OLE Server applications like Microsoft Word, Microsoft Excel, VisualWorks and VisualSmalltalk.

Cross-application access through automation

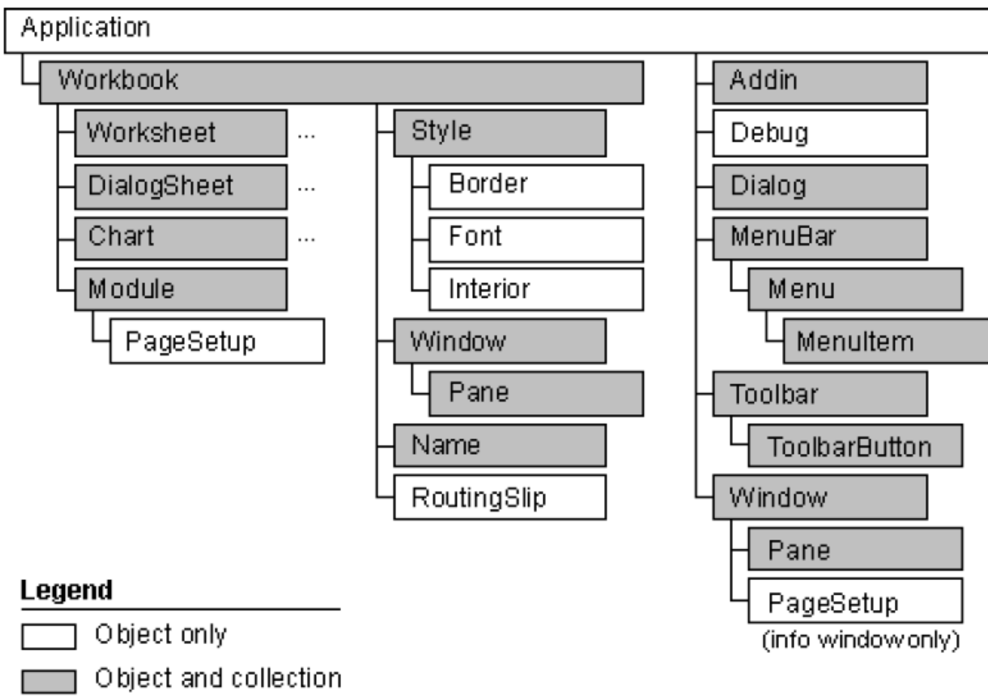


What Is An ActiveX Object?

An ActiveX object is an instance of a class that exposes properties, methods, and events to ActiveX clients. ActiveX objects support the COM standard. An ActiveX component is an application or library that is capable of creating one or more ActiveX objects. For example, Microsoft Excel exposes many objects that you can use to create new applications and programming tools. Within Microsoft Excel, objects are organized hierarchically, with an object named Application at the top of the hierarchy.

The figure below shows some of the objects in Microsoft Excel.

Microsoft Excel objects



Each ActiveX object has its own unique member functions. Exposing the member functions makes the object programmable by ActiveX clients. Three types of members for an object can be exposed:

- Methods are actions that an object can perform. For example, the Worksheet object in Microsoft Excel provides a Calculate method that recalculates values in the worksheet.

- Properties are functions that access information about the state of an object. The Worksheet object's Visible property determines whether the worksheet is visible.
- Events are actions recognized by an object, such as clicking the mouse or pressing a key. You can write code to respond to such actions. In Automation, an event is a method that is called, rather than implemented, by an object.

For example, you might expose objects like those listed in the following table in a document-based application by implementing these methods and properties:

ActiveX object	Methods	Properties
Application	Help	ActiveDocument
	Quit	Application
	Save	Caption
	Repeat	DefaultFilePath
	Undo	Documents
		Height
		Name
		Parent
		Path
		Printers
		StatusBar
		Top
		Value
		Visible
		Width
Document	Activate	Application
	Close	Author
	NewWindow	Comments
	Print	FullName
	PrintPreview	Keywords
	RevertToSaved	Name
	Save	Parent
	SaveAs	Path
		ReadOnly
		Saved
		Subject
		Title
		Value

Often, an application works with several object instances, which together make up a collection object. For example, an ActiveX application based on Microsoft Excel might have multiple workbooks. To provide an easy way to access and program the workbooks, Microsoft Excel exposes an object named Workbooks, which refers to all of the current Workbook objects. Workbooks is a collection object.

In the figure above, collection objects in Microsoft Excel are shaded. Collection objects let you work iteratively with the objects they manage. If an application is created with a multiple-document interface (MDI), it might expose a collection object named Documents with the methods and properties listed as follows:

Collection object	Methods	Properties
Documents	Add Close Item Open	Application Count Parent

What Is An ActiveX Client?

An ActiveX client is an application or programming tool that manipulates one or more ActiveX objects. The objects can exist in the same application or in another application. Clients can use existing objects, create new instances of objects, get and set properties, and invoke methods supported by the object.

VisualWorks can now be an ActiveX client, just like VisualBasic, Visual C++, or Java. You can use VisualWorks and similar programming tools to create applications that access Automation objects. You can also create clients in these ways:

- Write code within an application that accesses another application's exposed objects through Automation.
- Revise an existing programming tool, such as an embedded macro language, to add support for Automation.
- Develop a new application, such as a compiler or type information browser, that supports Automation.

How Do Clients and Objects Interact?

ActiveX clients can access objects in two different ways:

- By using the IDispatch interface
- By calling one of the member functions directly in the object's virtual function table (VTable)

An Automation interface is a group of related functions that provide a service. All ActiveX objects must implement the IUnknown interface because it manages all of the other interfaces that are supported by the object. The IDispatch interface, which derives from the IUnknown interface, consists of functions that allow access to the methods and properties of ActiveX objects.

A custom interface is a COM interface that is not defined as part of COM. Any user-defined interface is a custom interface.

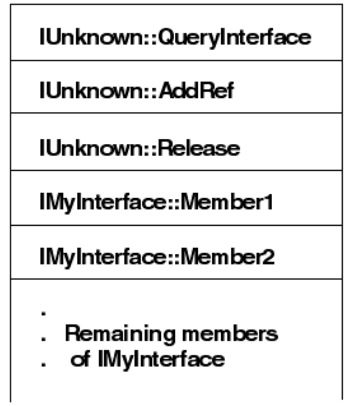
The VTable lists the addresses of all the properties and methods that are members of an object, including the member functions of the interfaces that it supports. The first three members of the VTable are the members of the IUnknown interface. Subsequent entries are members of the other supported interfaces. The following figure shows the VTable for an object that supports the IUnknown and IDispatch interfaces.

VTable for an object that supports the IUnknown and IDispatch interfaces

IUnknown::QueryInterface
IUnknown::AddRef
IUnknown::Release
IDispatch::GetIDsOfNames
IDispatch::GetTypeInfo
IDispatch::GetTypeInfoCount
IDispatch::Invoke

If an object does not support IDispatch, the member entries of the object's custom interfaces immediately follow the members of IUnknown. For example, the following figure shows the VTable for an object that supports a custom interface named IMyInterface.

VTable for an object that supports a custom interface

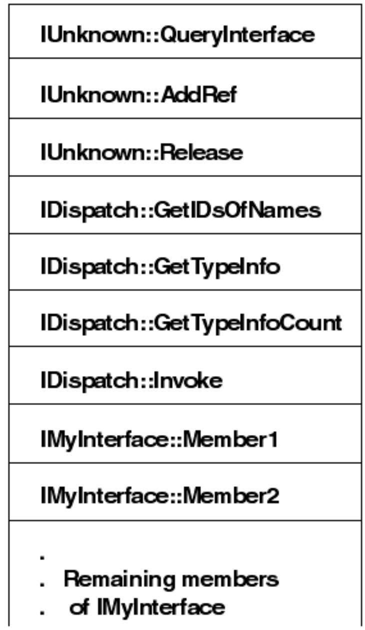


When an object for Automation is exposed, you must decide whether to implement an IDispatch interface, a VTable interface, or both. Microsoft strongly recommends that objects provide a dual interface, which supports both access methods.

In a dual interface, the first three entries in the VTable are the members of IUnknown, the next four entries are the members of IDispatch, and subsequent entries are the addresses of the dual interface members.

The following figure shows the VTable for an object that supports a dual interface named IMyInterface:

VTable for an object that supports a dual interface



In addition to providing access to objects, Automation also provides information about exposed objects. By using IDispatch or a type library, an ActiveX client or programming tool can determine which interfaces an object supports, as well as the names of its members. Type libraries, which are files or parts of files that describe the type of one or more ActiveX objects, are especially useful because they can be accessed at compile time. For information on type libraries, refer to What Is a Type Library? below, and Implementing Automation Objects.

Accessing an Object Through the IDispatch Interface

ActiveX clients can use the IDispatch interface to access objects that implement the interface. The client must first create the object, and then query the object's IUnknown interface for a pointer to its IDispatch interface.

Although programmers might know objects, methods, and properties by name, IDispatch keeps track of them internally with a number called the dispatch identifier (DISPID). Before an ActiveX client can access a property or method, it must have the DISPID that maps to the name of the member.

With the DISPID, a client can call the member IDispatch::Invoke to access the property or invoke the method, and then package the parameters for the property or method into one of the IDispatch::Invoke parameters.

The object's implementation of IDispatch::Invoke must then unpack the parameters, call the property or method, and handle any errors that occur. When the property or method returns, the object passes its return value back to the client through an IDispatch::Invoke parameter.

DISPIDs are available at runtime, and, in some circumstances, at compile time. At runtime, clients get DISPIDs by calling the IDispatch::GetIDsOfNames function. This is called late binding because the controller binds to the property or method at runtime instead of at compile time.

The DISPID of each property or method is fixed, and is part of the object's type description. If the object is described in a type library, an ActiveX client can read the DISPIDs from the type library at compile time, and avoid calling IDispatch::GetIDsOfNames. This is called ID binding. Because it requires only one call to IDispatch (the call to Invoke), rather than the two calls required by late binding, it is generally about twice as fast. Late-binding clients can improve performance by caching DISPIDs after retrieving them, so that IDispatch::GetIDsOfNames is called only once for each property or method.The IDispatch interface and class is wrapped in Smalltalk by the COMDispatchDriver class, which handles all the low-level mechanics of using the IDispatch API. The class COMDispatchDriver is described under Using Automation Objects.

Accessing an Object Through the VTable

Automation allows an ActiveX client to call a method or property accessor function directly, either within or across processes. This approach, called VTable binding, does not use the IDispatch interface. The client obtains type information from the type library or a header file at compile time, and then calls the methods and functions directly. VTable binding is faster than both ID binding and late binding because the access is direct, and no calls are made through IDispatch.

Using a dual interface form is described under Exposing Classes Through Dual Interfaces.

In-Process and Out-of-Process Server Objects

ActiveX objects can exist in the same process as their controller, or in a different process. In-process server objects are implemented in a dynamic-link library DLL) and are run in the process space of the controller. Because they are contained in a DLL, they cannot be run as stand-alone objects. Out-of-process server objects are implemented in an executable file and are run in a separate process space. Access to in-process objects is much faster than to out-of-process server objects because an interface function call does not need to make remote procedure calls across the process boundary.

The access mechanism (IDispatch or VTable) and the location of an object (in-process or out-of-process server) determine the fixed overhead required for access. The most important factor in performance, however, is the quantity and nature of the work performed by the methods and procedures that are invoked. If a method is time consuming or requires remote procedure calls, the overhead of the call to IDispatch can make a call to VTable more appropriate.

What Is a Type Library?

A type library is a file or part of a file that describes the type of one or more ActiveX objects. Type libraries do not store objects; they store type information. By accessing a type library, applications and browsers can determine the characteristics of an object, such as the interfaces supported by the object and the names and addresses of the members of each interface. A member can also be invoked through a type library. For details about the interfaces, refer to the OLE Programmer's Reference manuals or equivalent programmer documentation.

When ActiveX objects are exposed, creating a type library to make objects easily accessible to other developers is recommended. The simplest way to do this is to describe objects in an Object Description Language file (`.odl` or `.idl`), and then compile the file with the MIDL or MkTypLib utility, as described in "Type Libraries and the Object Description Language" on the MSDN web site.

For this release of Automation, the Microsoft Interface Definition Language (MIDL) compiler can also be used to generate a type library. For information about the MIDL compiler, refer to the *Microsoft Interface Definition Language Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK) section of the Microsoft Developer's Network (MSDN).

2

Using COM Objects

A COM application is an application that employs COM objects in defining its operation. COM objects provide services to clients by supporting interfaces, sets of functions that a client application can call to perform whatever processing is supported by the object.

VisualWorks COM Connect provides the mechanisms necessary to access external COM objects and invoke their processing by calling functions in COM interfaces. You can also, and often must, implement a COM object using COM Connect to make its interfaces available to external clients.

Acquiring COM Objects

In the simplest case, a Smalltalk COM application is a client of an external COM object. External COM objects occupy their own memory space, and you gain the use of an external COM object by establishing a connection with one or more of its interfaces.

The actual connecting work of creating a new COM server object is done by underlying COM mechanisms built into the COM library on the host platform. You need only be able to identify an object and request one of its interfaces.

To use a COM object in an application, you create a new instance of the object and send messages to it, as is usual for Smalltalk objects. The public Smalltalk interface implemented for the object manages the intricacies of invoking interface functions directly.

Basic COM Interface Support

The most important aspect of COM is the interface, which provides the connection between a COM object and its clients. A COM object typically supports several interfaces that together represent the services provided by the object.

An interface is simply a collection of related functions, representing a well-defined contract between an object and its clients. The interface definition specifies the syntax and semantics of each function in the interface.Remember that when you use a COM object, you always do so through an interface; you can never reference a COM object directly. To use the COM object, you obtain an interface reference through which you invoke the services supported by the object.

In COM Connect, interfaces are represented by subclasses of the COMInterface abstract superclass. Since all COM objects support the IUnknown interface, it is implemented by the IUnknown class under COMInterface. All other interfaces are implemented as subclasses of IUnknown.

Each interface reference that your application obtains on a COM object is represented by an instance of the concrete COMInterface class that supports that interface, as indicated by its IID. COM interface functions are invoked by sending messages to the COMInterface instance that represents the interface references.

Acquiring COM Interfaces and Creating COM Objects

COM objects implement specific behavior for the functions supported by their interfaces. To acquire a specific interface from a COM object, you must first have a reference to some interface supported by the object supporting that interface. From this interface you can then get references to additional interfaces supported by the object.

Typically, you first must create a new instance of a COM object that you know is installed on your system and listed in the system registration database. Once you have obtained the first interface on a COM object, which you get when creating a new instance of a published COM object class, you can obtain additional interfaces. Depending on the object and its capabilities, you might also be able to obtain references to other COM objects, through services published by the COM object you created.

You instantiate a COM object by calling either COM interface functions or API's. To create new COM object instances in Smalltalk, COM object creation API's are generally made available as class methods of suitable COMInterface subclasses. For example, the IMoniker interface class provides services for creating new instances of standard types of COM moniker objects and returning an IMoniker interface instance on the newly created moniker. Similarly, the COMCompoundFile class provides services for creating or opening COM structured storage files and obtaining an IStorage interface instance on the COM storage.

The general mechanism for instantiating a COM object is to use the object creation services provided by the IClassFactory interface class. The `createInstance:iid:` message instantiates a COM object of the specified CLSID and returns the specified interface from the newly created COM object. The COM object class can be specified either by the CLSID (a GUID) or by the PROGID string name that identifies the object server application to COM.

Since all COM objects implement IUnknown, you often ask for that interface first. The `createInstance:` message can be used in this case. For example, to create a new instance of some COM class and obtain its IUnknown interface, evaluate an expression of the form:

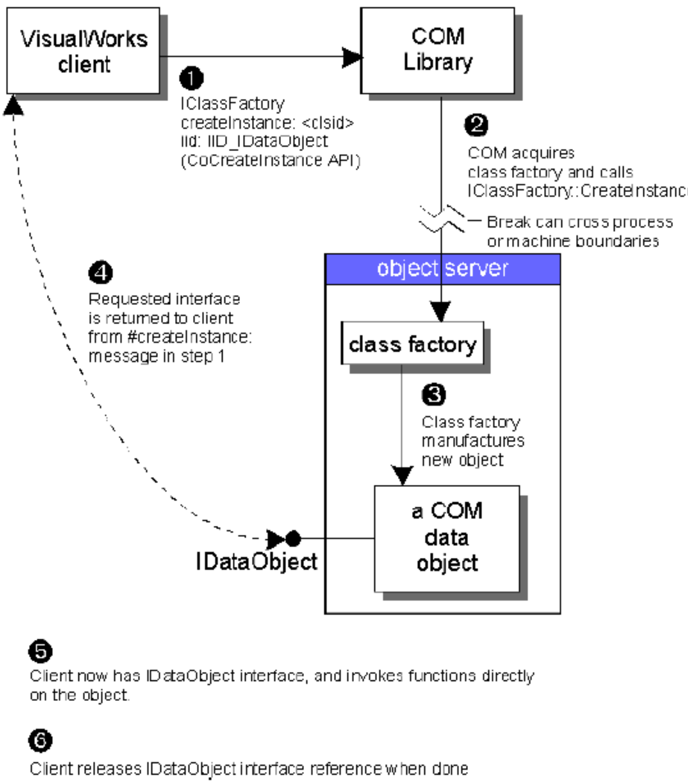
```
anIUnknown := IClassFactory createInstance: clsid.
```

where `clsid` is the class ID or ProgID name for the COM class. For example, the following code fragments demonstrate creating a standard Windows Paintbrush object by specifying either the CLSID or the ProgID of the COM object class:

```
" specify a COM class by a CLSID "
anIUnknown := IClassFactory createInstance: CLSID_Paintbrush_Picture.
" specify the COM class by a ProgID "
anIUnknown := IClassFactory createInstance: 'PBrush'.
```

The following figure depicts the processing that occurs when you create a new COM object using IClassFactory. When you create a new object, the COM library obtains the class factory for the object class you specify in your creation request, starting the object server application, if necessary, and creates a new COM object. The interface that you request is returned to you.

Creating a COM Object



The createInstance: family of messages in IClassFactory includes additional variations for advanced use, allowing you to specify additional attributes of the created object and its server application, including specifying a remote server to access an object through DCOM.

Once you have a reference to one of an object's interfaces, you can acquire additional interfaces by sending queryInterface: to a known interface. For example, having anIUnknown as above, you can acquire a reference to the object's IDataObject interface, if it supports this interface, by sending:

```
anIDataObject := anIUnknown queryInterface: IID_IDataObject
```

To create an inner object, for use within an aggregate object, you must specify the controlling unknown of the aggregate when instantiating the inner object. For example:

```
anInnerUnknown := IClassFactory createInstance: clsid
   iid: IID_IUnknown
   controllingUnknown: self controllingUnknown.
```

If you need to create several instances of the class, you can obtain an instance of the class factory for the COM object class by sending the message forCLSID: to IClassFactory. Once you have the class factory itself, you can create any number of objects. For example, to create two instances of some COM object, you could evaluate an expression such as the following:

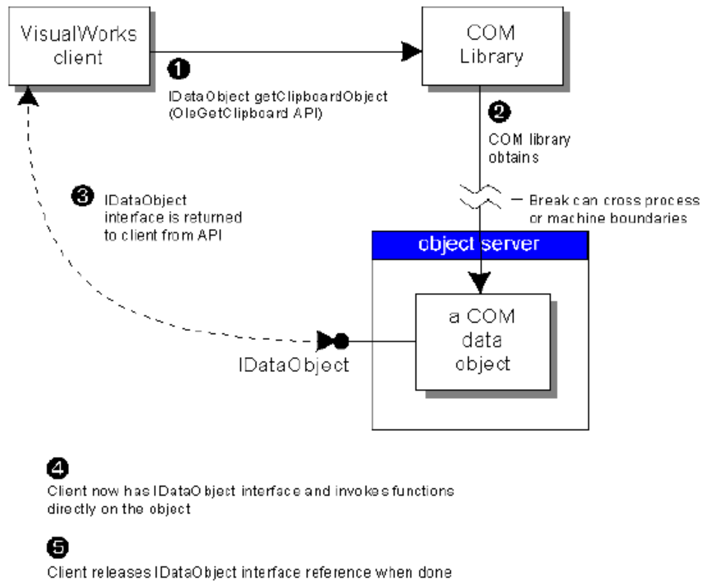
```
aClassFactory := IClassFactory forCLSID: clsid.

"Create and get IUnknown"
anUnknown := aClassFactory createInstance.

"Create and get IID_IDataObject"
anIDataObject := aClassFactory createInstance: IID_IDataObject.
aClassFactory release.
```

The following figure depicts one situation in which you acquire an interface to an existing COM object, rather than creating an entirely new object. The COM library provides a service that allows you to obtain an IDataObject interface on a COM data object representing the contents of the system clipboard.

Acquiring a COM Object



Managing Object References

COM interface reference counting is the mechanism in COM for managing object lifetimes in a system of cooperating software components. Briefly, the reference count mechanism is used to keep track of how many clients are using an interface and, ultimately, its underlying COM object. The object continues to provide services as long as any client is using at least one of its interfaces.

When a client acquires an interface, the reference count of that interface is incremented. When the client is done using the interface, it is responsible for releasing the interface by sending the release message, which invokes the IUnknown::Release function. Releasing the interface decrements its reference count and allows the object to determine whether its services are still required. When an object no longer has any clients, it can terminate itself, releasing any interfaces it has acquired.

The period between when a client obtains an interface and when it releases the interface constitutes the lifetime of the client's ownership of that interface. During this lifetime, the client, or any object with which the client shares the interface, can freely invoke any function in that interface.

The COM specification, with which you should be familiar, defines several rules for reference counting. The rules are fairly simple for client applications, and for a Smalltalk COM application these amount to requirements for releasing interfaces and for making separately countable references to interfaces.

In the simplest case, an application acquires a single reference to an interface, either by sending queryInterface: or through some class message that creates a new COM object. While in possession of the interface, the application can invoke any function supported by the interface. When finished using the interface, perhaps as part of the application shutdown processing, it releases the interface by sending release to the interface.

The simplest usage pattern is the case where you create a new COM object by requesting a specific interface that provides the services you want to use, invoke some services using that interface, and release the interface when you are done. The following illustrates this simple pattern:

```
| anIDataObject |
anIDataObject := IClassFactory createInstance: clsid
    iid: IDataObject iid.
" … invoke various functions in the IDataObject interface…"
anIDataObject release.
```

A more typical example occurs when you create a new COM object and use services from several interfaces supported by the object. This more typical pattern tends to look like the following:

```
| anIUnknown anIDataObject anIPersistFile|
"create a new COM object that contains some data "
anIUnknown:= IClassFactory createInstance: clsid.
" do something to its state via IDataObject "
anIDataObject := anIUnknown queryInterface: IID_IDataObject.
" … invoke, say, IDataObject ::SetData… "
anIDataObject release.
" now save this object to a file "
anIPersistFile := anIUnknown queryInterface: IID_IPersistFile.
anIPersistFile saveAs: 'tempobj.dat'.
anIPersistFile release.
" release the COM object when we are done with it "
anIUnknown release.
```

Note that the preceding two code fragments demonstrate two styles of specifying an IID argument. You can always send the iid message to an interface class to get the IID identifier of the interface. You can also use the constants defined in the COMConstants pool, using the standard C names of the interface IID constants, as long as COMConstants pool is included in the compilation name space context in which you are working (e.g., when writing a method in a class you add it to the class's list of pool dictionaries).

Usage is not always this simple, however, and can require several references to an interface. Smalltalk applications involve many objects with varying lifetimes. Any object can acquire an interface reference and, since the interface is represented by a Smalltalk object, can be shared with any other object. The application developer's responsibility is to make sure that objects have separate references to a shared interface when necessary.

Any time your application gets a new interface reference by making an API or a function call to a COM object, it receives a new, separately counted reference. Your application is responsible for ensuring that the interface is released when you are done using it. This applies to every interface reference that you obtain. For example, if you make more than one QueryInterface call to obtain the same interface from an object, you must invoke `release` for each separate reference you obtained.

Note that some care must be taken here, because the reference is represented as a Smalltalk object, and as an instance of a COMInterface subclass, it can be shared by any number of other Smalltalk objects.

You can always use an interface reference as an argument to any function call (message send), since the receiver only has the reference for the duration of the function call. Since the lifetime of the function call is completely contained within the lifetime of the sender, the sender and receiver objects can use the same copy of the interface, and the sender retains control of the lifetime of the reference.

Cooperating objects within your application can also share a reference to an interface that one of your application objects has acquired. An interface reference can be freely shared with other objects in your application, as long as the object that acquired the interface can ensure that no other object is still using the interface when it is ready to release it. In some cases, it is not possible for a cooperating object to assure the original owner that it will be finished using the interface when the owner is ready to release the interface. In this situation, a separate copy of the interface should be made by sending the separateReference message to the interface. This creates a separate reference to the interface that can be used independently by the cooperating object.

Each of your application objects can now use the interface independently and release it when they are done. The original owner releases the original interface reference when it is done using it, as usual, while the cooperating object is responsible for releasing its separate reference to the interface when it is done.

Using COM Interface Functions

A COM object client invokes the functions in the COM interfaces it has acquired by sending messages to an instance of the relevant COMInterface subclass. Subclasses of COMInterface define each COM interface that has been wrapped for use with COM Connect.

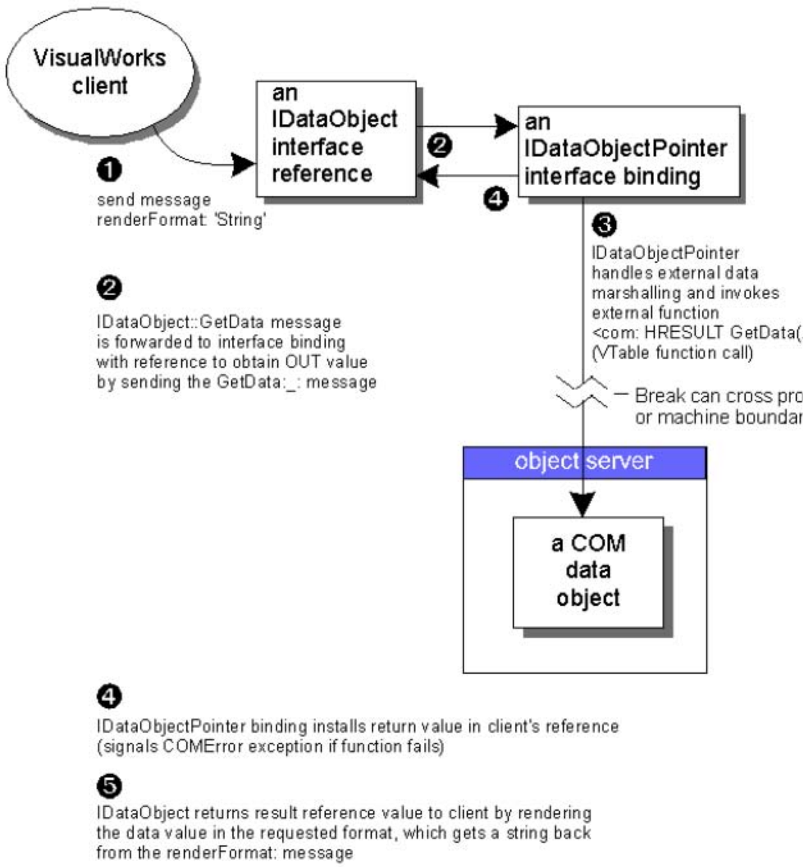
Subclasses of COMInterface are named with the common name of the interface, e.g., IUnknown, or IDataObject. The interface represented by a class is identified by its unique IID (interface identifier), which can be obtained by sending the iid message to the interface class. The IID of an interface instance can also be obtained by sending the iid message to the instance. Interface IID values are also defined in the COMConstants pool.

COMInterface subclasses provide the public interface to COM interfaces and their functions in Smalltalk. The public interface includes a message for every function defined in the COM interface. The class can also implement methods to simplify common usage patterns.

The interface function methods in a COMInterface class handle the details of invoking the native host-level interface and API functions. Where necessary, transformations are provided between Smalltalk values and the host data type values used to directly invoke the native host function. This encapsulation of the host data type management within the interface class allows you to invoke COM interface functions in a natural fashion in your Smalltalk application, with normal Smalltalk objects provided as arguments to the interface function message and returned as the message result value.

The following figure depicts the processing that occurs when you invoke an interface function by sending a message to a COMInterface instance.

Using a COM Interface



Error Handling

When you design a COM application, as with any application, it is your responsibility to provide suitable error handling. Almost all COM interface functions and API's can signal the `COMError` exception to indicate failure conditions. You must decide how to handle such error conditions in your application. Using the standard facilities of the Smalltalk exception system, you have great flexibility in determining how to implement error handling and where in your application to locate the logic to handle exception conditions resulting from function failures.

Almost all COM interface functions return a status code indicating success or failure as the function value. In Smalltalk, an error result from an interface function results in an instance of the error exception COMError being signalled. Processing of the interface function is terminated when an error is signalled.

A COMError includes the HRESULT return code, as well as a message describing the error. In your exception handler block, you can send the hresult message to the exception to obtain the HRESULT error code. You can send the description message to the exception to obtain a string describing the error. The COM error description is obtained from the host system.

In a small number of cases, a COM error code returned by an interface function is considered to be a "normal" return code and not signalled as an error. For example, the Smalltalk binding of the IUnknown::QueryInterface function treats the E_NOINTERFACE error code as a reasonable return condition, for which nil is returned, with no error signalled.

Some interface functions return a status code indicating that the function succeeded, but that provides additional information. For example, the IStorage::CreateStorage function returns STG_S_CONVERTED when the existing storage of the same name was replaced with a new storage containing the single stream CONTENTS.

In Smalltalk, a success result other than S_OK from an interface function is indicated by signalling COMResultNotification, which includes the HRESULT return code, as well as a message describing the result code. Clients can use the Smalltalk exception handling system to detect such success notifications if this is of interest. Processing proceeds normally after the notification is signalled, independent of whether a client exception handler is provided.

Managing Memory

A COM application must follow certain memory management rules whenever an allocated block of memory is passed as an argument to an interface function or a COM API. When you call a COM function that returns the address of a value, such as a structure or a string that the callee allocated to service your request, you must release the memory when you are done with the data by sending the release message.

COM interface functions and API calls classify parameters into one of three groups: IN, OUT, and IN/OUT. An IN parameter is allocated and freed by the caller. Since ownership of the memory is not transferred, an IN parameter can be allocated in whatever fashion is preferred by the caller. On the other hand, OUT and IN/OUT parameters that contain pointers to memory represent memory whose ownership is transferred, and so these must be allocated using the COM task memory allocator.

An OUT parameter is allocated by the callee then transferred to the caller; thus it must be released by the caller. An IN/OUT parameter is allocated and eventually freed by the caller, but the callee can free and reallocate the memory as necessary. Since ownership of the memory is transferred, IN/OUT and OUT parameters must be allocated and released using the COM task memory allocator.

A Smalltalk COM application is responsible for releasing any memory it obtains as a return value from an interface function, when it is done using that memory. This situation usually occurs when a data structure or string is allocated in external memory by an interface or API function that you called. You release the memory by sending it the `release` message. In most cases, the actual COM memory address is returned as a Smalltalk object, such as a String or COMStructure data structure class. Sending `release` ensures that any memory that was allocated is properly released when you are done with the data that was returned from the COM function.String return values are generally brought into Smalltalk memory immediately as a side effect of converting the Unicode string encoding returned by COM functions into a Smalltalk string in the host binding wrapper method of a COM interface or API function. Consequently, the memory containing the returned string is released before returning the string result value to the Smalltalk caller.

In Depth: Class Factories and Object Creation Contexts

This section takes an in depth look at the creation of objects. You can skip this part if you do not need to learn to control this process in detail.

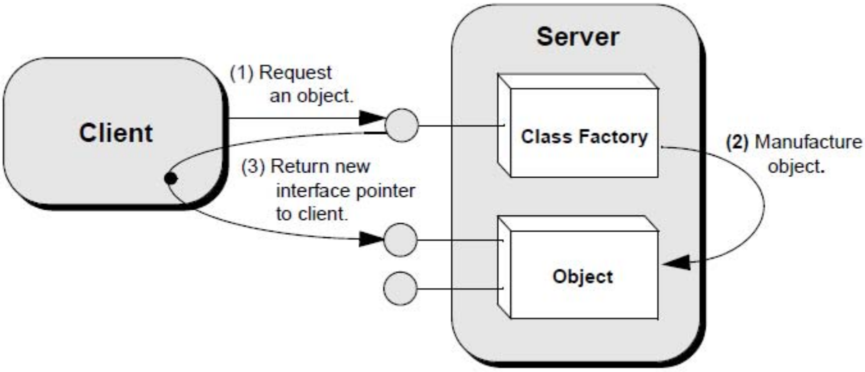
This section refers to the COSERVERINFO structure and the RemoteServerName key, both of which are described under Publishing COM Objects.

To create and use COM objects, you need a way to identify and create instances of an object class. Ideally, the class identification is tied as closely as possible to the CLSID. Storing the name in the registry as the value of the CLSID is a good technique for doing this. A class can also be identified (less precisely) using the ProgID or VersionIndependentProgID; these are keys in the registration database. Another way to find a class is by specifying a filename that is associated with the class.

In Smalltalk, the IClassFactory class is used to instantiate objects by wrapping the COM CoCreateInstance API.

The following figure illustrates what happens when a client requests a new instance of an object:

A COM Client creates objects through a Class Factory.



The steps involved are as follows:

1 The client asks COM to create a new object.

2 COM causes the class factory in the server to create a new object and answer an interface pointer.

3 COM returns the pointer to the client.

Note: The client and server can be in the same process, in separate processes, or in separate machines.

Accessing Objects With IClassFactory

The Smalltalk class IClassFactory provides several class messages, which are used to create a single uninitialized object of the class associated with a specified CLSID. Call a createInstance method when you want to create only one object on the local system.

To create a single object on a remote system, call a method with a serverName: argument. To create multiple objects based on a single CLSID, refer to the getClassObject method for the CLSID: method, which gets you an IClassFactory interface reference through which you can manufacture any number of objects of that COM class.

Class Context Definitions

Different pieces of code can be associated with one CLSID for use in different execution contexts, such as in-process, local, object handler, or remote. The context in which the caller is interested is indicated by the context: parameter of class methods in the COMDispatchDriver and IClassFactory classes. The context is a group of flags taken from the enumeration CLSCTX and defined in the COMConstants Pool Dictionary:

```
typedef enum tagCLSCTX {
   CLSCTX_INPROC_SERVER= 0x1,
   CLSCTX_INPROC_HANDLER= 0x2,
   CLSCTX_LOCAL_SERVER= 0x4,
   CLSCTX_REMOTE_SERVER= 0x10,
} CLSCTX;
```

The several contexts are tried in the sequence in which they are listed here. Multiple values can be combined (using bitwise OR) indicating that multiple contexts are acceptable to the caller:

```
#define CLSCTX_INPROC (CLSCTX_INPROC_SERVER |
   CLSCTX_INPROC_HANDLER)

#define CLSCTX_SERVER (CLSCTX_INPROC_SERVER |
   CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER)

#define CLSCTX_ALL(CLSCTX_INPROC_SERVER |
   CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER |
   CLSCTX_REMOTE_SERVER)
```

These context values have the following meanings, which apply to all remote servers, as well:

Value	Action Taken by the COM Library
CLSCTX_INPROC_SERVER	Load the in-process code (DLL) which creates and completely manages the objects of this class. If the DLL is on a remote machine, invoke a surrogate server as well to load the DLL.
CLSCTX_INPROC_HANDLER	Load the in-process code (DLL) which implements client-side structures of this class when instances of it are accessed remotely. An object handler generally implements object functionality that can be implemented only from an in-process module, relying on a local server for the remainder of the implementation. See Note 1 below.
CLSCTX_LOCAL_SERVER	Launch the separate-process code (EXE) which creates and manages the objects of this class. See Note 2 below.
CLSCTX_REMOTE_SERVER	Launch the separate-process code (EXE) on another machine that creates and manages objects of this class. The LocalServer32 or LocalService code that creates and manages objects of this class is run on a different machine. This flag requires Distributed COM to work.

The COM Library should attempt to load in-process servers first, then in-process handlers, then local servers, then remote servers. This order helps to minimize the frequency with which the library has to launch separate server applications, which is generally a much more time-consuming operation than loading a DLL, especially across the network.

Note 1:

For example, in OLE 2, built on top of COM, there is an interface called IViewObject through which a client can ask an object to draw its graphical presentation directly to a Windows device context (HDC) through `IViewObject::Draw`. However, an HDC cannot be shared between processes, so this interface can only be implemented inside as part of an in-process object. When an object server wants to provide optimized graphical output but does not want to completely implement the object in-process, it can use a lightweight object handler to implement just the drawing functionality where it must reside, relying on the local server for the full object implementation.

Note 2:

In some cases the object server might already be running and might allow its class factory to be used multiple times, in which case the COM Library simply establishes another connection to the existing class factory in that server, thus eliminating the need to launch another instance of the server applications entirely. While this can improve performance significantly, it is the option of the server to decide between a single-use or multiple-use class factory. See the `CoRegisterClassObject` function for more information.

Class Context Processing

When you are using IClassFactory object creation services on DCOM-enabled systems, before being passed to DCOM, the value CLSCTX_REMOTE_SERVER is added automatically to CLSCTX_SERVER and CLSCTX_ALL.

Given a set of CLSCTX flags, the execution context to be used depends on the availability of registered class codes and other parameters according to the following algorithm:

The first part of the processing determines whether CLSCTX_REMOTE_SERVER should be specified as follows:

1. If the call specifies either
   - an explicit server name (in the COSERVERINFO structure from a serverName: argument) indicating a machine different from the current machine, or
   - there is no explicit COSERVERINFO structure specified, but the specified class is registered with either the RemoteServerName or ActivateAtStorage named-value, then CLSCTX_REMOTE_SERVER is implied and is added to the context flags. The second case allows applications written prior to the release of DCOM to be the configuration of classes for remote activation to be used by client applications available prior to DCOM and the CLSCTX_REMOTE_SERVER flag.
2. If the explicit COSERVERINFO parameter indicates the current machine, CLSCTX_REMOTE_SERVER is removed (if present) from the context flags.

   The rest of the processing proceeds by looking at the value(s) of the context flags in the following sequence.

3 If the context flags includes CLSCTX_REMOTE_SERVER and no COSERVERINFO parameter is specified, if the activation request indicates a persistent state from which to initialize the object (with CoGetInstanceFromFile, CoGetInstanceFromIStorage, or, for a file moniker, in a call to IMoniker::BindToObject) and the class has an ActivateAtStorage sub-key or no class registry information whatsoever, the request to activate and initialize is forwarded to the machine where the persistent state resides.

4 If the context flags includes CLSCTX_INPROC_SERVER, the class code in the DLL found under the class's InprocServer32 key is used if this key exists. The class code runs within the same process as the caller.

5 If the context flags includes CLSCTX_INPROC_HANDLER, the class code in the DLL found under the class's InprocHandler32 key is used if this key exists. The class code runs within the same process as the caller.

6 If the context flags includes CLSCTX_LOCAL_SERVER, the class code in the Win32 service found under the class's LocalService key is used if this key exists. If no Win32 service is specified, but an EXE is specified under that same key, the class code associated with that EXE is used. The class code (in either case) is run in a separate service process on the same machine as the caller.

7 If the context flags is set to CLSCTX_REMOTE_SERVER and an additional COSERVERINFO parameter to the function specifies a particular remote machine, a request to activate is forwarded to this remote machine with the context flags modified to be CLSCTX_LOCAL_SERVER. The class code runs in its own process on this specific machine, which must be different from that of the caller.

8 Finally, if the context flags includes CLSCTX_REMOTE_SERVER and no COSERVERINFO parameter is specified, if a machine name is given under the class's RemoteServerName named-value, the request to activate is forwarded to this remote machine with the context flags modified to be CLSCTX_LOCAL_SERVER. The class code runs in its own process on this specific machine, which must be different from that of the caller.

3

Implementing COM Objects

COM applications are not usually pure client applications, but often must themselves present an interface to a serving object. To do this, an application must implement one or more COM objects that give access to interfaces used by the external client.

COM objects are implemented in Smalltalk as subclasses of the abstract superclass COMObject, which provides a framework for implementing the responsibilities of a COM object. A COM object is associated with one or more interfaces. The object implements methods that perform the functions in those interfaces and maintain its internal state.

COMObject Framework

The COMObject superclass provides a standard implementation for the IUnknown interface, which is required of all COM objects. The COMObject superclass also provides support for the COM reuse mechanism of aggregation as a standard capability. The IUnknown implementation handles object life cycle services, through the IUnknown AddRef and Release reference counting services, and it provides the framework for releasing COM object resources when the object is no longer in use. A standard framework is provided for supporting IUnknown::QueryInterface processing, so that subclasses have a simple way to hook interfaces they support into the COM IUnknown interface for negotiation and processing.

The COMObject framework allows you to focus on the specific capabilities your object class offers through its supported interfaces. Implementing a COM object in Smalltalk involves creating a concrete subclass of COMObject, specifying the set of interfaces supported by the object so that they can be exposed through IUnknown::QueryInterface, and creating methods to implement your object's supported interface functions.

Note: You can implement a COM object anywhere in the class hierarchy, but you must satisfy the public protocol requirements of the COMObject class and implement the IUnknown responsibilities of a COM object suitably. Subclassing the COMObject implementation framework makes the job of developing a new COM object much simpler, because the COMObject superclass handles all the standard responsibilities of supporting IUnknown and provides a basic framework for expressing the additional capabilities of a new object class.

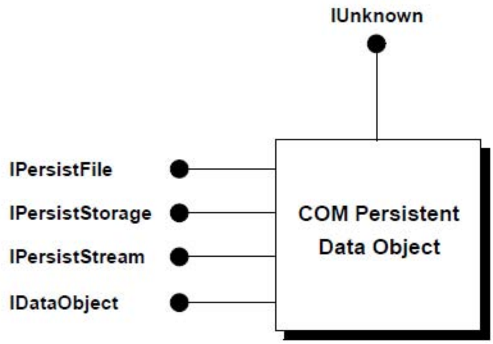
Implementation Examples

To demonstrate the basic techniques for implementing COM objects, two sample COM object implementations are provided with VisualWorks COM Connect. The COM object implementation samples are installed with the COM Connect software and can be found in the COM—COM Samples category in your VisualWorks system browser.

The COMRandomNumberGeneratorObject class publishes a simple random number generator (RNG) through a COM interface IRandomNumberGenerator. It demonstrates how to implement support for a simple COM interface and provide the standard services within the COMObject framework to support IUnknown operations. As discussed later in this document, the class also provides minimal functionality to support the IDataObject interface, in order to demonstrate some additional techniques for implementing interface function processing.

As shown in the following figure, the COMPersistent-DataObject class demonstrates a COM object with a simple internal state, which is exposed to clients through the standard IDataObject interface and allows clients to save and load the persistent data object (PDO) from various external backing stores, through the standard COM IPersist family of interfaces.

Supporting a COM interface



The COMPersistentDataObject sample is a slightly more complex COM object implementation than the RNG class. It supports several COM interfaces and uses the COM technique of aggregation in order to implement its support for the IDataObject interface. In addition, the PDO sample demonstrates how to add support for publishing a COM object as an object server application, which is achieved by adding logic during image startup for object server application initialization and providing an application termination policy to shut down the image when the server is no longer required.

Note: The COMRandomNumberGeneratorObject sample does not currently include a demonstration of publishing the RNG COM object as an object server application. This is not a fundamental deficiency of the COM RNG object implementation or of the VisualWorks COM support; doing so is simply a matter of providing some additional mechanisms for external publication, such as a type library, interface marshalling for the IRandomNumberGenerator custom interface, and a .REG file to register the object server application EXE in the system registry. The RNG sample is deliberately simple to focus on the internal Smalltalk programming tasks involved with implementing a COM object. Other examples that demonstrate the additional steps involved in creating and publishing a VisualWorks object server application are provided with COM Connect.

Both sample COM objects come with simple test drivers that demonstrate exercising the supported interface functions of the RNG and PDO objects. In addition, the COMPersistentDataObjectBrowser application is provided as a sample COM client of the persistent data object. Using the PDO browser tool, you can create new objects, save them to files in the various persistence formats, and reload saved objects.

Design Guidelines for Implementing COM Objects

When you implement a COM object by implementing a new subclass of COMObject, the functionality you publish through COM interfaces is typically provided in one of two ways. One technique is to implement the functions directly in your COMObject class. Alternatively, you can separate the core functionality of the object from the COM-specific responsibilities of the COMObject framework by delegating the function processing to another object.

Separating the object implementation from the COM object mechanisms is the same design principle that you apply when separating the domain objects from the view in an application. In both cases, the fundamental behavior of the domain object is separated from the mechanisms by which the object is exposed to users or clients, whether it be through an application view on a business object or a COM object exposing services through COM interfaces. Separation of the domain object from the COM object increases the reusability of your application objects, but requires some additional design work to structure your classes and their interactions appropriately.

Implementing the functionality of your COM object directly in your COMObject subclass is usually appropriate when the object's purpose is fundamentally COM-specific. For example, a class factory is only useful for COM purposes, so factoring the implementation to separate the domain processing from the `COMClassFactoryObject` itself is of no particular value. Similarly, the `COMPersistentDataObject` sample has no interesting functionality that might be useful to reuse or access other than through the COM interfaces that it supports, so the functionality of this COM object is implemented directly in the COMObject class.

It is useful to separate a domain object from the COMObject implementation that exposes its services through COM interfaces, when the application object can be used in ways other than simply by COM clients. For example, the COM random number generator sample separates the RNG implementation from the `COMRandomNumberGeneratorObject`, which exposes its services through a COM interface. This separation of the application object from the COM object mechanisms is a more flexible architecture, in that the RNG object can be used in other contexts and different implementations can easily be published through the COM interface simply by delegating to a different application object.

The automation object implementation examples all follow this more generalized architectural pattern, separating the application object from the COM object class(es) that publish the services through COM interfaces. This separation makes it easier to evolve an implementation of an automation object from an initial simple publication, using the standard COMAutomationServer that supports IDispatch, into a dual interface object, which supports both IDispatch and VTable interface clients through a dual interface.

Supporting COM Interfaces

The capabilities of a COM object are determined by the interfaces that it exposes to clients and by the implementation of its interface functions. Specifying the interfaces supported by the object and implementing the processing of interface functions are the main activities of COMObject subclass implementation.

Every COM object must support the IUnknown interface, which is created when the object is instantiated. Additional interfaces supported by the object determine its personality and behavior. A COM object must be able to return each of its supported interfaces when requested by a client through an invocation of the IUnknown::QueryInterface function.

The COMObject framework class provides a standard implementation of the IUnknown::QueryInterface function, which handles requests for the IUnknown interface that are allocated and managed by the superclass, and delegates to its subclasses the responsibility for responding to other interface requests. Each subclass is responsible for providing storage for its other supported interfaces, typically by allocating an instance variable for each supported interface, and for reimplementing the getInterfaceForIID: method to handle requests for any interfaces other than IUnknown. The method should return the requested interface, if the interface is supported, or nil if it is not.

When you implement a COM object, you can choose either to create all of its supported interfaces at once, when the object is created, or to implement a "lazy" allocation scheme in which interfaces are constructed only when requested by a client. Pre-allocation is the usual technique for interfaces that are typically used during the lifetime of the object. Lazy allocation is appropriate when your object supports interfaces that are used only occasionally, perhaps by specific clients or under certain circumstances. Lazy allocation allows you to avoid the overhead of unnecessarily allocating external resources needed to support an interface. The object implementation framework allows either strategy to be implemented with equal ease.

To allocate a supported interface at object creation, you reimplement the initializeInterfaces method in your COMObject subclass. The COMRandomNumberGeneratorObject sample demonstrates the standard pattern for preallocating interfaces at object creation time; its initializeInterfaces method is as follows:

```
initializeInterfaces
   " Private - Allocate any interfaces that are expected
   to be required during the object's lifetime. Invoke the
   superclass method to ensure that the inner IUnknown
   is allocated. "

   super initializeInterfaces.
   iRandomNumberGenerator := IRandomNumberGenerator
      on: self.
```

The COM RNG object preallocates the IRandomNumberGenerator interface, which it stores in a iRandomNumberGenerator instance variable, because IRandomNumberGenerator is the primary service interface supported by the RNG object, and it is expected that all clients will want to use it.

Note: The on: message is sent to an interface class to construct an interface that is connected to your COM object. This is the standard interface construction technique. As discussed later in this section, the RNG object actually implements an optimized form of the interface binding using the directBindingOn: message. The discrepancy between the sample method shown here and the actual implementation in the RNG class is deliberate.

Each interface you construct in your COM object's initializeInterfaces method must also be listed in the getInterfaceForIID: method in your COMObject subclass. This is required to support the standard implementation of the IUnknown::QueryInterface function in COMObject. The getInterfaceForIID: method simply answers the value of the preallocated interface created in initializeInterfaces when the IID argument specifies that interface.

The COMRandomNumberGeneratorObject demonstrates the standard pattern for implementing the getInterfaceForIID: method in a COM object to support IUnknown::QueryInterface requests. This example demonstrates both preallocation when constructing an interface, and the lazy allocation technique for allocating interfaces on demand. In addition to its primary IRandomNumberGenerator interface, the COM RNG object also supports the standard IDataObject interface.

Since the RNG object's support for IDataObject services can be characterized as only minimal (in fact it does nothing useful!), it is expected to be used only rarely by clients. Consequently, the IDataObject interface is allocated on demand in the getInterfaceForIID: method when a client actually requests it through an IUnknown::QueryInterface call. The getInterfaceForIID: method in COMRandomNumberGeneratorObject is as follows:

```
getInterfaceForIID: iid
    " Private - answer the interface identified by the GUID
     <iid>. Answer nil if the requested interface is not
     supported by the receiver. "

    " preallocated interfaces that are always constructed
       are simply returned "
    iid = IRandomNumberGenerator iid
       ifTrue: [ ^iRandomNumberGenerator ].

    " lazy allocation constructs the supported interface
       on demand "
    iid = IID_IDataObject
       ifTrue: [
          iDataObject isNil
             ifTrue: [ iDataObject := IDataObject on: self ].
          ^iDataObject ].
    ^super getInterfaceForIID: iid
```

In these examples, a supported interface on a COM object is constructed by sending the message on: to the interface class. For example, an IDataObject interface for a COM object can be created by an expression of the form:

```
IDataObject on: self
```

The result is an IDataObject interface with a binding that dispatches invocations of the functions to the methods in your COM object class. The COM RNG actually constructs its IDataObject interface using the flexible configuration techniques discussed later in this section, rather than the standard on: interface construction message used in the preceding code example. The discrepancy between the sample method provided here and the actual implementation in the RNG class is deliberate.

The `on:` message creates a flexible binding between interface functions and your object; this binding can be configured in a number of different ways to allow you to easily specify the processing for each interface function supported by your object. For more information, see Configuring Interface Function Processing.

For the fairly common case where you implement all functions in an interface in your COM object, using the standard function selectors for your methods, you can construct the interface using a more efficient mechanism that binds the interface functions directly to your object. A direct binding of the interface, for the case where you have implemented the complete signature of the interface, is constructed by sending the `directBindingOn:` message to the interface class. For example, the COM RNG class implements methods for all the functions in the `IRandomNumberGenerator` interface. Consequently, it constructs the interface binding using an expression of the form:

```
IRandomNumberGenerator directBindingOn: self
```

The result is an `IRandomNumberGenerator` interface with a binding that dispatches invocations of the functions from the methods directly to the COM RNG object. You cannot configure the function processing through a function adaptor binding in this case, but if you implement the complete interface in your object anyway and do not need the additional flexibility of the adaptor, you get better performance using a direct binding.

In addition to supporting QueryInterface processing, the IUnknown support of a COMObject must provide for both shutting down the object when the last client interface reference on the COM object is released and releasing any external resources consumed by the object. You must implement two "housekeeping" methods in your COMObject subclass to support releasing the interfaces allocated by your object during its lifetime. These methods are discussed under Releasing a COM Object.

If an object you implement supports a large number of interfaces, all of which might not be used during its lifetime, or if ease of implementation is more important than optimizing the size of your objects, you can implement your COM object as a subclass of `COMObjectWithInterfaceStorage`, an abstract subclass of COMObject. This class allocates a dictionary to hold all allocated interfaces supported by the object, with the IID of the interface used as the key to perform lookups in the dictionary. Subclassing this branch of the COMObject hierarchy is slightly simpler, because you do not have to define instance variables for each supported interface and implement the housekeeping methods needed to release them when the object is shut down.

The COMObjectWithInterfaceStorage class has a standard implementation of the getInterfaceForIID: method, which obtains already-allocated interfaces directly from the interface storage dictionary. This method should not be reimplemented in its subclasses. Subclass participation to support QueryInterface processing is required only the first time a supported interface is requested by a client. When you subclass COMObjectWithInterfaceStorage, you need only reimplement the createInterfaceForIID: method, which simply returns a newly constructed instance of the requested interface if it is supported. This is slightly simpler to implement than getInterfaceForIID: and does not require an implementation of the two corresponding release methods, as must be done for a direct COMObject subclass.

The sample COM object implementation COMPersistentDataObject, provided with COM Connect, demonstrates the various techniques for supporting interfaces in a COMObjectWithInterfaceStorage subclass. This sample object contains a simple state that it exposes to clients through the IDataObject interface. The object can be made persistent on various backing storage mediums through the standard COM IPersist interfaces.

The createInterfaceForIID: method in COMPersistentDataObject is similar to the following:

```
createInterfaceForIID: iid
    " Private - answer a new instance of the interface
    identified by the GUID <iid> on the receiver.
    Answer nil if the interface is not supported. "

    iid = IID_IPersistFile
        ifTrue: [
            ^IPersistFile on:: self ].
    iid = IID_IPersistStorage
        ifTrue: [
            ^ IPersistStorage on: self ].
    iid = IID_ IPersistStream
        ifTrue: [
            ^ IPersistStream on: self ].
```

```
iid = IID_IDataObject
    ifTrue: [ "this is discussed later…" ].
^super createInterfaceForIID: iid
```

Note: The actual code in the createInterfaceForIID: method in the COMPersistentDataObject class is not exactly like the preceding. In particular, the PDO implementation uses a variety of techniques for configuring the interface processing of the IPersist interfaces. The persistence interface that it considers the most likely to be used by clients uses an optimized direct binding, while the other persistence interfaces are constructed using the flexible configuration capabilities of an adaptor interface binding to manage the name space collisions within this family of interfaces. In addition, the IDataObject support is provided using the COM aggregation technique to reuse the capabilities of another COM object. The discrepancy between the sample method provided here and the actual implementation in the class is deliberate.

A simple application that allows you to create, save, and load COMPersistentDataObject instances is also provided as part of the COM Connect samples. You can run this sample application by evaluating the following expression:

```
COMPersistentDataObjectBrowser open.
```

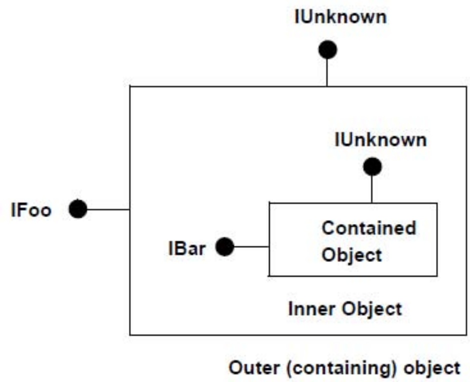
This application is a standard COM client that creates the COM PDO through a class factory, obtains interfaces using IUnknown::QueryInterface, and uses the object's services through these COM interfaces.

Reusing COM Objects

Reuse of existing components is accomplished in COM through the mechanisms of containment and aggregation. When you implement a COM object, you can create instances of other COM objects whose services you use to implement the functions of your own object. When you create a contained object, you obtain an interface and can keep it as long as you want to use the services of the inner object. Your object can use any services provided by an inner object through its supported interfaces, in the usual fashion.

The following figure depicts a COM object implemented using containment, which is a COM term that describes the function activity of using other objects in your implementation. Just as you might use a Dictionary when you implement an object that needs to maintain a mapping table as part of its state, you can use another COM object to use its services as part of your object implementation.

Reuse by Containment



Aggregation is a special case of containment that occurs when the controlling object wants to directly expose an interface of an inner object as its own. This is useful when you want to simply delegate the processing of all the functions of an interface directly to the inner object. Aggregation allows you to avoid the overhead of reimplementing all the functions of an interface when all your object needs to do is to relay the message to the inner object for processing.

When implementing a controlling object, you obtain the IUnknown interface of the inner object when you create it; this is referred to as the “inner IUnknown” of the contained object. Generally, you hold this inner unknown interface of the contained object for the lifetime of your object, and you release it when your object is itself released. As with interfaces that your object supports directly, you can choose to preallocate an inner object when your object is first created, typically by reimplementing the standard `initialize` method, or you can create the contained object on demand, to provide its services when the object is first needed.

When you use containment in your COM object implementation, reimplement the `releaseInnerObjects` method when the containing object is terminated to release the inner objects your containing object created during its lifetime.

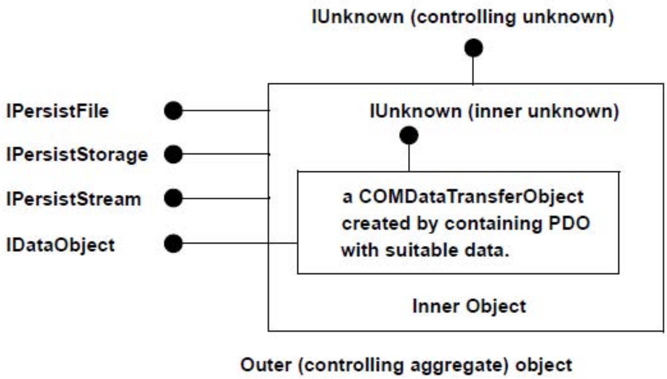
Using aggregation, you can expose any interfaces of an inner object as your own by implementing suitable logic to obtain the interface from the inner object when an `IUnknown::QueryInterface` request is made of your object. Because you are querying the inner object and thus obtaining a new separately reference counted copy of the interface, you must hook up QueryInterface support for inner objects of an aggregate by reimplementing the getInnerObjectInterfaceForIID: method.

Note: If you query an inner object within your aggregate object's getInterfaceForIID: method, as you ordinarily do to expose your object's supported interfaces to clients, double reference counting occurs. The standard COMObject implementation of IUnknown::QueryInterface assumes both that interfaces obtained in this fashion are obtained directly from the object's private state and that a separately reference counted copy must be returned to the caller, in conformance with the COM rules for interface reference counting. In the case of an interface obtained from an inner object, however, this generally results in a second undesired AddRef being done. This kind of reference counting bug in object implementations can be tricky to locate and debug - the symptom is that your object does not ever get released, since it still thinks there is an outstanding reference to it, because it has inadvertently bumped its own reference count an extra time.

The sample COM object COMPersistentDataObject demonstrates using aggregation to directly expose an interface of an inner object. The COMPersistentDataObject reuses the capabilities of an existing COM object to support the IDataObject interface. The inner object used by the aggregate is created on demand by the PDO implementation, which does a lazy allocation of the inner object only when a client actually requests the IDataObject interface. Once created, the inner IUnknown of the data object is saved in an instance variable of the controlling object, which now owns this inner object for the remainder of its lifetime.

The following figure illustrates reuse by aggregation.

Reuse by Aggregation



The getInnerObjectInterfaceForIID: method of COMPersistentDataObject looks like the following:

```
getInnerObjectInterfaceForIID: iid
    " Private - answer a separate reference to the interface
    identified by the GUID <iid> of an inner object, which
    is to be directly exposed to clients as an interface of the
    controlling object of an aggregate. Answer nil there is
    no such interface. "

    iid = IID_IDataObject
        ifTrue: [
            innerUnknownDO isNil
                ifTrue: [ self allocateDataObject ].
            ^innerUnknownDO queryInterface: IID_IDataObject ].
    ^super getInnerObjectInterfaceForIID: iid
```

Note: The actual code in the getInnerObjectInterfaceForIID: method in the COMPersistentDataObject class in your image is not exactly like the preceding. The discrepancy between the sample method provided here and the actual implementation in the COM object class is deliberate.

The inner object is released when the containing object is itself released by reimplementing the releaseInnerObjects method, as follows:

```
releaseInnerObjects
    " Private - release any inner objects owned by the receiver. "
    self releaseDataTransferObject.
    super releaseInnerObjects.
```

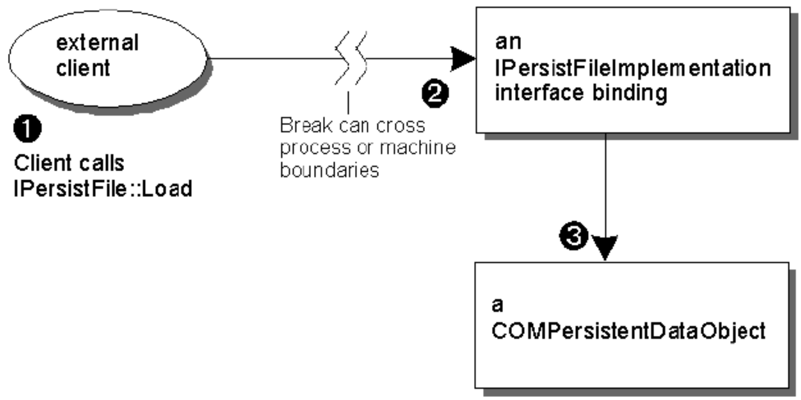
The COMObject framework provides complete support for aggregation, so that any COM object that you create can in turn be used by other objects through the mechanisms of containment and aggregation. If for some reason you create an object that you do not want reused within an aggregate, reimplement the supportsAggregation class method to return a false value.

Configuring Interface Function Processing

When you construct an interface that is supported by your object, the interface binding dispatches invocations of the functions in the interface to your object. For example, the binding for the IUnknown::AddRef function of the IUnknown interface causes the AddRef message to be sent to your COM object when the AddRef function is invoked. This interface is supported by every COM object and must be included among the interface functions of every interface your object supports. Similarly, interface functions are dispatched to your COM object for each function in a supported interface.

The following figure illustrates the processing involved when a client invokes an interface function implemented by a VisualWorks COM object.

Implementing an interface function



❷
External function call enters Smalltalk image at interface binding callback method.

❸
Interface binding unmarshalls arguments and sends Load:_: to the implementing object with String argument.

❹
The COM object performs the function (loads its data from the file) and returns S_OK to the caller (returns an error HRESULT if unsuccessful).

In many cases, the standard interface function dispatching is sufficient. When you decide to support an interface, you simply provide the necessary support for QueryInterface and for releasing, as described in the previous section, and implement methods in your class corresponding to the functions in the interface.

Note that you do not need to implement the IUnknown interface functions, since standard implementations of the interface negotiation and reference counting capabilities of a COM object are provided by the COMObject class. You should almost never reimplement the IUnknown operation methods.

When you support an interface, you must implement methods in your COM object class for the interfaces functions' selectors published by the interface. To get a list of function names in VTable order, use the functionNames method defined in COMInterface class. For example:

```
IDataObject functionNames inspect
```

You can also study the VTable function definitions from the C++ header files or an IDL specification of the interface. Another COMInterface message, `describeInterfaceFunctions`, can be used to obtain more detailed information about the interface functions. For example, to obtain a detailed description of the IDataObject interface functions, with complete type information about each function signature, evaluate the expression:

```
IDataObject describeInterfaceFunctions
```

Configuring a Direct Interface Binding

If your object implements the complete set of standard function selectors for an interface that you support, you can construct a more efficient binding of the interface onto your object by using the `directBindingOn:` message to create a direct binding of the interface. A direct binding requires that you support exactly the standard message signatures for every function in the supported interface. Unlike an adaptor binding constructed by sending the `on:` message to the interface class, a direct binding cannot be configured to map the function selectors onto alternate protocol supported by your object or to dispatch certain functions to other objects for processing. However, the flexibility of an adaptor binding is not always necessary, and in this case it is more efficient to use a direct binding.

Configuring an Adaptor Interface Binding

In some cases, you might need to configure the interface function processing for your object in a more flexible manner than is provided by the standard dispatching connections, which simply send the message corresponding to the interface function to your object. For example, if you support two interfaces that contain a function of the same name, to provide appropriate function processing for each distinct interface function, you need to configure the interface function processing to invoke two separate methods in your object class. Smalltalk interface support provides facilities that allow you to configure a customized interface binding when you need additional flexibility for your COM object implementation.

Configurable interface function processing for a COM object is enabled by constructing an interface binding on your object using the `adaptorBindingOn:` message. An adaptor binding uses a function adaptor to map the standard interface function message selectors onto specified function processing actions for which you registered function dispatch handlers. An adaptor binding makes it easy to map interface function selectors to alternate message names that are sent to your object. This is useful when two interfaces that your object supports have the same message selector for functions with different semantics. It can also be used to implement partial support for an interface by mapping some of the operations onto a standard “not implemented” handler.

An interface function handler is usually a method in the implementing object, since the implementation of the desired behavior often requires access to the object’s private state. Also, the behavior provided for functions in different interfaces supported by the object often must be coordinated, which requires that the state of the implementing object be known.

An interface function handler does not need to be implemented as a method in the COM object class, however. The function handler can be any evaluable action, including a block or a message sent to another object. For more information on interface function handlers, refer to Implementing Interface Functions.

An interface binding with customized function dispatching is typically constructed by a COMObject instance in one of two ways, depending on whether the object implements all of its interface functions itself, or implements only some of its interface functions, possibly delegating others to another object.

If the object itself implements all the interface functions, it can construct the supported interface using the on:selectors: class method, specifying itself as the implementing object and providing the list of message selectors to be sent to it for each function. The function processing selectors must be in VTable order when specified in this fashion. The standardIUnknownSelectors message can be sent to self to obtain the list of standard IUnknown function handler selectors. VTable selector lists should be constructed by using standardIUnknownSelectors and concatenating the remaining selectors of the interface to complete the VTable selector list.

Whenever possible, name the interface function methods in your COM object classes with the standard function name, using the native function name as the primary keyword of the method’s message selector (which typically has an uppercase first character, in contrast to the usual Smalltalk message naming convention) and using the standard anonymous keyword (the underscore character _:) for any additional argument keywords. For example, QueryInterface:_: is the standard message selector for the two-argument IUnknown::QueryInterface function. This is the convention assumed by the standard interface function dispatching. In addition, using the standard function name for your method provides a tangible reminder that you are in a COM interface function implementation, which has some special rules about how processing is implemented and results are returned to the caller.

If you find that you must implement two methods in your class to support functions with the same name, but from different interfaces, choose a simple convention to modify the basic message selector in a consistent way, so that your method selector reflects in an obvious way the interface function name and the containing interface.

If your object does not implement all functions in an interface, or if it delegates processing responsibility of any interface functions to another object, construct the interface for your object using the on: class method in the usual fashion, then reconfigure the function processing specifications as desired using the function adaptor of the interface. Send the functionAdaptor message to an adaptor interface to obtain the function adaptor used to configure interface function processing.

The standard mapping provided by an adaptor interface binding dispatches all interface functions to your object using the standard message selectors of the interface functions. You can reconfigure these default dispatching specifications in several ways, which allows great flexibility in specifying the function processing for an interface. A function dispatch handler for an interface binding is registered with the function adaptor using the standard registration messages of the Smalltalk application event system, such as when:send:to:, with the name of the interface function specified as the event name.

If you implement only a few functions in the interface, you can easily reconfigure the interface binding to support only the standard IUnknown functions and the specific functions for which you have implemented methods in your object. To install a minimal function binding specification, which dispatches the standard IUnknown operations to your object and answers the E_NOTIMPL result code for all other functions in the interface, send installMinimumDispatchHandlers to the interface binding that you are constructing.

Using the standard registration facilities of the Smalltalk application event system, you can then configure function processing for the specific functions that you support. This makes it easy to support an interface in which you need to implement a few functions, without having to implement a large number of methods that do nothing besides returning E_NOTIMPL to the caller.

The COMRandomNumberGeneratorObject example demonstrates using an adaptor binding to easily implement partial support for an interface. The COM RNG object provides a degenerate implementation of IDataObject that does nothing useful, simply to show how easy it is when you want to do very little. The IDataObject interface has eight functions and the RNG object is interested in supporting only one of them, the EnumFormatEtc function (which does not actually do much either, but that is what the implementor intends in this case).

Rather than implementing methods for each of the seven unsupported functions, which would all simply answer the E_NOTIMPL result code, the COM RNG constructs an adaptor binding for the IDataObject interface that maps the unsupported functions onto a standard not-implemented handler provided by the function adaptor. A function dispatch handler is then configured for the supported EnumFormatEtc function.

The adaptor binding construction of the IDataObject interface in the COMRandomNumberGeneratorObject class looks like the following:

```
createIDataObject
    " Private - answer a new IDataObject interface
    on the receiver. "

    | anIDataObject |
    anIDataObject := IDataObject adaptorBindingOn: self.
    anIDataObject functionAdaptor
        installMinimumDispatchHandlers;
        when: #EnumFormatEtc
            send: #EnumFormatEtc:_:
            to: self;
        yourself.
    ^anIDataObject
```

After constructing the adaptor binding, the installMinimumDispatchHandlers message is sent to the function adaptor to install the default not-implemented handler for all functions other than the minimum set required to support IUnknown. (Generally, the IUnknown operations should never be remapped. They are configured to dispatch the IUnknown functions to the standard COMObject methods.) A function dispatch handler is then configured for the single function supported in the RNG object.

Note: The actual implementation of RNG does a little more than is shown in the preceding code sample. For example, it also demonstrates configuring a block as a function handler. The discrepancy between the sample method provided here and the actual implementation in the COM object class is deliberate.

To delegate processing of interface functions to an object other than the COM object that supports the interface, simply configure a function handler that sends the appropriate message to the object that you want to have perform the actual function processing. For example, the installMinimumDispatchHandlers message causes function handlers to be installed that send a message to the interface binding rather than to your implementing object. Any interface function that you do not subsequently configure with a specific handler that dispatches the function call to your implementing object is then handled by the adaptor binding without notifying you, since presumably you do not have an implementation of that function.

Implementing Interface Functions

The majority of the logic in a subclass of COMObject is the implementation of the function processing for the interfaces that are supported by the object. The methods that you implement and the private state that you maintain in your COM object class are determined entirely by the purpose of your object and the behavior that you choose to give the object. As with any Smalltalk class, you can use existing classes or create new classes as appropriate to produce the desired behavior in your COM object.

A function processing method should indicate an error condition by returning an appropriate HRESULT value. It should not signal the COMError exception. By design, signalling COMError is the responsibility of COM interface pointer and interface implementation classes. Signalling COMError in a function handler is at best redundant and at worst returns a nonspecific result to an external caller in an expensive way. All the standard HRESULT values are defined in the COMStatusCodeConstants pool.

Output arguments are provided by the caller as value references. Your function handler method sets the value of an output argument by sending the value: message to the caller's reference.

A function processing method must follow the COM rules for interface reference counting and memory allocation when an interface or a block of memory allocated by the callee is returned to the caller. An interface returned to the caller must be reference counted by the callee. This is usually done either by obtaining a separate reference to an interface using the standard IUnknown::QueryInterface service or by sending separateReference to a known interface instance. Memory that is allocated by the callee and returned to the caller must be allocated using the COM task allocator. Reference counting is discussed in more detail under Using COM Objects. Memory management and using the COM task memory allocator is also discussed this chapter.

Releasing a COM Object

A Smalltalk COM object maintains an overall reference count on the object, which is incremented and decremented as its interfaces are reference counted. The reference count of a COMObject instance is set to zero when the object is created. The reference count is incremented and decremented each time the IUnknown::AddRef and IUnknown::Release functions are invoked on any interface supported by the object. When the reference count is decremented to zero, you can assume there are no longer any clients of any of the interfaces supported by your COM object. Under the COM reference counting rules, a COM object is allowed (and indeed expected) to release its resources and destroy itself when its reference count reaches zero.

Termination processing of a COM object is implemented in the releaseResources method. The standard cleanup performed by the implementation in COMObject includes releasing the external resources of all supported interfaces. This is an important service, because each interface you supply to an external client causes external memory to be allocated. This memory is not managed by the Smalltalk object memory manager, and must be released explicitly.

To support releasing the resources of interfaces supported by your object, you must reimplement the following methods in your COMObject subclass:

- allocatedInterfacesDo:, which must evaluate a one-argument block on each interface that has been allocated during the object's lifetime. The primary use of this message is by the object release logic in COMObject to release the resources used by a COM object when the last client interface reference in released.

- resetAllocatedInterfaces, which clears any references in the object's instance variables to the deallocated interfaces.

> **Note:** These two methods do not need to be implemented if you have subclassed COMObjectWithInterfaceStorage.

For example, the cleanup methods in the COM random number generator object sample are as follows:

```
allocatedInterfacesDo: aOneArgBlock
    " Private - enumerate the interfaces supported by the
    receiver which have been allocated during its lifetime
    and evaluate <aOneArgBlock> with each. "

    super allocatedInterfacesDo: aOneArgBlock.
    iRandomNumberGenerator notNil
        ifTrue: [
            aOneArgBlock value: iRandomNumberGenerator ].
    iDataObject notNil
        ifTrue: [ aOneArgBlock value: iDataObject ].

resetAllocatedInterfaces
    " Private - reset the references to the interfaces
    supported by the receiver. "

    super resetAllocatedInterfaces.
    iRandomNumberGenerator := iDataObject := nil.
```

The releaseInnerObjects method should be reimplemented in any subclass of COMObject that creates inner objects for reuse through containment or aggregration. For example, the sample COM class COMPersistentDataObject implements this method to release the data object it uses to support the IDataObject interface through aggregation.

The releaseResources method should be reimplemented in any subclass of COMObject whose instance state contains resources other than supported interfaces and inner objects that should be released when the object's lifetime is terminated.

Returning Values From an Interface Function

COM interface functions are almost always designed to conform to the convention that the return value of the function is an HRESULT value, which provides the caller with a status code describing the outcome of the function. Output values are returned to the caller through OUT arguments, which the callee sets if the function succeeds.

The standard HRESULT status codes are defined as pool variables in the COMStatusCodeConstants pool. When you implement a function in your COM object, you can answer an HRESULT as the return value of the function using these constants.

Arguments to your interface function that have OUT semantics are provided by the interface binding mechanisms as reference values. In your interface function method, you return a value for an OUT parameter by sending the value: message to the provided reference argument. Note that you should never set the value of an output argument unless you are also returning a success result code from your function.

For example, the sample COMRandomNumberGenerator object implements the IRandomNumberGenerator::Next function, as follows:

```
Next: resultReference
    " Private - implement the
    RandomNumberGenerator::Next operation. "

    resultReference value: rng next.
    ^S_OK
```

As with most COM interface functions, this function returns a value to the caller by sending value: to the reference argument to set the return value, and it answers an HRESULT status code that indicates whether the function succeeded.

Special considerations exist when the value you return is an interface. Specifically, any interface value returned by a COM function must be reference counted by the callee (your COM object, in this case). The interface reference that your function returns becomes the responsibility of the caller, which is required to release its reference when done using the interface returned to the caller. The next section discusses in more detail how the COM reference counting rules apply when returning an interface value from a function that you implement, and it discusses function implementation techniques you can use.

Implementing Reference Counting

COM objects implemented in Smalltalk must implement the COM reference counting rules for the interfaces they support, whenever interfaces are returned from a function implementation to the caller.

Every time a request is made for a COM interface, the supporting object must return a new reference to that interface and increment the count for that interface by one. To do this in Smalltalk, send the `separateReference` message to the interface instance. For example, the standard implementation in COMObject of `IUnknown::QueryInterface` returns a separately reference-counted copy of the interface to its caller as the return value of the function.

There are two standard patterns in Smalltalk COM object interface function implementations for returning an interface to a caller as the value of an output argument. To return an interface to which your object already has a reference, use the `separateReference` message when setting the value of the output argument. For example, if your COM object has acquired some interface that it uses in its processing and has the interface saved in an instance variable, you would return this interface to a caller using an expression such as the following:

```
resultReference value: someInterface separateReference.
```

The second pattern occurs when you return an interface that you obtain by asking some other COM object for a reference to a particular interface. This is typically done using an expression of the form:

```
resultReference value: ( anInterfaceInstVar queryInterface: iid ).
```

Since the `queryInterface:` message always returns a separately reference-counted interface to the caller, you return this directly to the caller.

Some reference counting rules also apply to interfaces passed as input arguments to a function. An interface that is passed as a function argument does not need to be reference counted by the caller and can be used freely by the callee for the duration of the function invocation. Within the implementation of an interface function in a COM object you implement, you do not have to do anything if you use only an interface argument during the actual function call. Do not increment the reference count of the interface, and do not release it. In most cases, this means that you can use the interface arguments in your function implementation without worrying about reference counting.

If your implementation of an interface function needs to keep an interface argument for use after the function is completed, however, you must keep a separate reference to the interface argument. For example, to save an interface argument in an instance variable of your COM object for later use, evaluate an expression of the form:

```
someInterface := anInterfaceArgument separateReference.
```

Since your object now owns a new, separately reference counted copy of the interface, you are responsible for controlling the lifetime of the reference and must release it when finished using it.

These two patterns cover most of the situations that you encounter when implementing a COM object in Smalltalk. Special situations might arise as a result of shared use of an interface or in cases where reference counting cycles can occur. Generally these can be resolved by remembering that an interface is an object in Smalltalk and analyzing the lifetime characteristics of how the interface is used by the various clients. If the interface needs to be used by another object and the lifetime of that usage is not known to be contained within the lifetime of the owner's usage, create a separately reference-counted copy of the interface.

Reference counting cycles can occur in situations where two objects are mutually dependent, with each holding interface references to the other. This situation can be resolved by making suitable exceptions to the normal reference counting rules, after careful analysis of object dependencies and lifetimes.

Memory Management

In an earlier section, it was explained that an application is responsible for releasing any memory block received from an interface or API function call. This applies also to COM objects implemented in Smalltalk.

You must follow the COM memory allocation rules when implementing an interface function in a COM object that is responsible for allocating and returning memory to the caller. In this case, you must use the COM task allocator to allocate the memory.

The COMMemoryAddress class is used to allocate memory blocks using the COM task allocator. To allocate a block of memory, send the `allocateMemory:` class message to COMMemoryAddress, specifying in the message argument the size in bytes of the memory to be allocated. All IN/OUT parameters passed to a COM function, or returned from a COM interface implementation as the value of an OUT parameter that contains a pointer to a memory block, must be allocated in this way.

The most common situation in which you need to concern yourself with memory allocation occurs when you implement a function that allocates a structure and returns a pointer to the structure as the value of an OUT argument. The COMStructure class provides standard services for allocating structures in external COM task memory to facilitate properly allocated structures to return as OUT argument values. The `createExternalStructure:` message is sent to the COMStructure class to allocate the COM structure whose name is specified as the message argument using the COM task memory allocator.

For example, suppose you implement a function that returns a newly allocated POINT to the caller as the value of an OUT argument of the function. The C declaration of such a function would look like the following:

HRESULT GetExtent (/* [out] */ POINT * lpExtent)

Your method that implements this function would look like the following:

```
GetExtent: resultReference
    "Implement the IFoo::GetExtent function."
    aPointStruct :=
        COMStructure createExternalStructureNamed: #POINT.
    aPointStruct
        memberAt: #x put: 2;
        memberAt: #y put: 4;
    resultReference value: aPointStruct.
    ^S_OK.
```

The COM memory that you allocated for the return value is released by the caller.

4

COM Infrastructure Support

This chapter describes various COM infrastructure technologies that are supported in VisualWorks COM Connect.

COM Pools

The following pools define COM pool variables:

COMAutomationConstants	Constants used in COM Automation applications
COMConstants	COM constants and enumerated type values
COMStatusCodeConstants	COM function status codes and HRESULT values

The pool variables contained in these pools are exactly those defined as the constant name in the C header files.

Basic COM Data Types

COM Connect provides support for the basic COM data types as described in these sections.

Globally Unique Identifiers

A globally unique identifier (GUID) is a 16-byte (128-bit) value that is guaranteed to be unique. GUID values are used extensively in COM; for example, they are used as COM object class identifiers (CLSID) and interface identifiers (IID). In Smalltalk, the GUID class contains GUID values.

A new GUID is allocated by evaluating the expression:

```
GUID new
```

This is typically done when allocating a GUID to identify a new interface or COM object class that you want to publish.

A GUID can be created by sending the asGUID message to a string containing the display name of a GUID. For example, the IID of the IUnknown interface can be obtained by evaluating the expression:

```
'{00000000-0000-0000-C000-000000000046}' asGUID
```

Note: You usually obtain this particular value simply by using the IID_IUnknown constant in the COMConstants pool or by sending the iid message to the IUnknown class.

There are also instance creation messages in the GUID class that invoke COM APIs to obtain GUID values.

Conversions between CLSID and ProgID values are supported. To convert a CLSID to the corresponding ProgID name of its application, you use the asProgID message. For example, to obtain the ProgID of the Windows Paintbrush application, evaluate the following expression:

```
CLSID_Paintbrush_Picture asProgID
```

To obtain the CLSID of a ProgID, you send the asGUID message to the string name of the application. For example, you can obtain the CLSID of the Excel application object by evaluating the following expression:

```
'Excel.Application' asGUID
```

HRESULT Values

The standard convention, used by almost all COM interface and API functions, is to return an HRESULT status code as the function result value. The abstract class HRESULT provides the standard COM services for testing HRESULT values, as well as for accessing fields and constructing HRESULT values.

For example, the standard test for a successful HRESULT that is commonly used in COM interface binding classes is expressed as follows:

```
| hresult |
hresult := anInterface someFunction: ...
      " invoke some interface function "
( HRESULT succeeded: hresult )
      ifTrue: [ ... success logic ... ]
      ifFalse: [ ... error handling ... ].
```

The HRESULT class also provides utility services for obtaining information about an HRESULT code, such as the error message defined by the operating system or a string describing the condition represented by an HRESULT.

COM Enumerators

Enumeration is provided in COM by a family of interfaces that enumerate collections containing a specific element type. Collections are homogeneous and each element type has a unique interface defined for enumerating collections of that type.

Using COM Enumerators

In Smalltalk, enumerators of all element types are supported by the interface class IEnum. Supported enumeration element types are interfaces, structures, and strings. An IEnum interface reference that is obtained from an API or interface function has an instance-specific IID associated with it, which identifies the type COM enumeration interface for the element type of the collection being enumerated. For example, an enumerator on a collection of IUnknown interface values is an IEnum interface instance with an IID of IID_IEnumUnknown, while an enumerator on a collection of FORMATETC data format description structures is an IEnum instance with an IID of IID_IEnumFORMATETC.

The standard COM enumerator interfaces and their element types are predefined in the IEnum class. Additional enumerator interfaces can be supported by registering the IID and the element type in IEnum enumerator registry. Register an interface and type by sending the class message:

```
IEnum registerEnumeratorIID: aGUID for: aClass
```

Implementing COM Enumerators

Enumerators are implemented in Smalltalk by the COMEnumeratorObject family of COM object implementations. To support an IEnum interface, you create an instance of the appropriate enumerator object class, providing a homogeneous collection of interface or structure instances. Clients can then enumerate the elements of the collection that you provided using the appropriately typed IEnum interface, which is supported by the Smalltalk COM enumerator.

To provide an enumerator on a collection of interfaces, you create a COMInterfaceEnumerator on a collection of interfaces. For example, if you are implementing a COM object that supports the IEnumMoniker interface to allow clients to enumerate a collection of monikers that your object owns, you would construct an enumerator by an expression such as:

```
COMInterfaceEnumerator forIID: IID_IEnumMoniker
    elements: ( "… a collection of monikers … " )
```

To provide an enumerator on a collection of structures, you create a COMStructureEnumerator on a collection of structures. For example, if you are implementing a COM data object that supports the IEnumFORMATETC interface, to allow clients to enumerate a collection FORMATETC structures describing the formats in which your object allows its data to be obtained or set, you construct an enumerator by an expression such as:

```
COMStructureEnumerator forIID: IID_IEnumFORMATETC
    elements: ( "… a collection of FORMATETC structures… " )
```

To provide an enumerator on an arbitrary collection of values that are rendered using the self-describing VARIANT structure used in Automation, you create a COMVariantEnumerator on a collection of values. In contrast to other collections, the values exposed through a VARIANT collection need not be homogeneous. Any value that can be represented as a VARIANT can be contained in the collection. For example, if you are implementing a COM data object that supports the IEnumVARIANT interface, to allow clients to enumerate a collection of attribute values, construct an enumerator with an expression such as:

```
COMVariantEnumerator on: #( 1 3.14 -99 'string' true false )
```

COM Monikers

A COM moniker is a reference to a specific COM object that can be stored in persistent external storage and reloaded in the future. The primary operation of a moniker is to bind to the referenced object. The IMoniker class is the interface class for the primary interface to COM monikers.

COM provides a number of standard system moniker types, such as file, item, and composite monikers. To create new instances of the system-supported monikers, the IMoniker class provides a number of class messages, such as createFileMoniker: to create a file moniker, and createItemMoniker: to create an item moniker.

COM Structured Storage Support

One of the basic technologies of COM is structured storage, which is a hierarchical model of persistent storage similar to the directory and file model of hierarchical file systems. COM structured storage is used by container applications to manage compound document files.

COM storages and streams provide support for various storage and access modes. A COM storage is a directory-like object that can contain streams and other storages. A COM stream contains data bytes. The COM structured storage facilities include support for managing shared access to storage elements and a transaction model for controlling how changes in working state are committed to persistent storage.

The COM persistent storage facility is also referred to as compound document storage. A structured storage file containing a compound document is also referred to as a compound file or a compound document file.

The IStorage interface provides operations on storages. An IStorage provides services to create or open the streams and substorages that it contains, enumerate its contents, move and copy elements between storages, rename elements, delete elements, and commit changes to the contents or revert to the original state of the contents.

A COM storage that exists as a file in the file system is accessed using the COMCompoundFile class, which is an IStorage with additional class services to create and open a root storage document file.

The IStream interface is used to read and write the underlying bytes in a stream. An IStream provides services to read and write data bytes and to commit changes, or to revert to the original state of the stream.

A temporary storage or stream that is backed by Win32 global memory, rather than a permanent or temporary file in the file system, can be created using the COMGlobalMemoryStorage and COMGlobalMemoryStream classes, respectively.

A COMReadWriteStream is a Smalltalk stream on the data bytes of an IStream. As with the FileConnection and file system external stream classes, to which IStream and COMReadWriteStream correspond, most operations on a data stream by a Smalltalk client are done through the COMReadWriteStream instance, with the underlying IStream instance rarely manipulated directly. A COM stream can support either byte or character semantics for interpreting the underlying data bytes.

You can construct a COM stream by sending the message on: to COMReadWriteStream, providing an IStream instance. The default interpretation is to support character semantics, for consistency with the string-oriented behavior of the existing Smalltalk Stream classes. To specifically create a byte or character COM stream, you can send the asByteStream or asCharacterStream message directly to the IStream.

COM Uniform Data Transfer Support

Uniform Data Transfer is a set of interfaces that allows COM applications to exchange data in a standard way. Central to uniform data transfer is the IDataObject interface, which allows a data transfer object to communicate to the outside world.

An application that implements a data transfer object supporting the IDataObject interface can use the object in various data transfer mechanisms, such as the system clipboard or drag-drop transfer. The IDataObject interface contains methods for retrieving, setting, querying, and enumerating data, and handle data exchange notifications.

The COMDataTransferObject class is a COM object provided by COM Connect that supports the IDataObject interface and can be used to provide the source data for a data transfer operation. A data transfer object is configured with a set of one or more values, one for each format in which the data can be rendered.

Clipboard Data Transfer

COM clipboard support is provided using the IDataObject interface. A server application places an IDataObject on the clipboard, while a data consumer obtains an IDataObect from the clipboard.

COM provides two APIs for setting and retrieving data from the clipboard: OleGetClipboardData and OleSetClipboardData. These APIs are wrapped in the getClipboardObject method in IDataObject, and in the copyToClipboard method in IDataObject and in COMDataTransferObject.

The IDataObject clearClipboard message clears the clipboard contents. The OleFlushClipboard API empties the clipboard and removes the IDataObject instance. This API is wrapped by the flushClipboard class method in IDataObject.

A COMDataTransferObject is used as a data source object to copy data onto the system clipboard. One or more formats can be specified, according to the form of the data being provided and the rendering formats supported by the data source. The following code fragment copies data in the form of a string to the clipboard:

```
| aDataTransferObject |
aDataTransferObject := COMDataTransferObject new.
aDataTransferObject addRendering: ' Hello world '
    format: 'String'.
aDataTransferObject copyToClipboard.
```

The IDataObject interface provides services for obtaining data from the clipboard. The following code fragment demonstrates obtaining data in the form of a string from the clipboard

```
| anIDataObject aString |
anIDataObject := IDataObject getClipboardObject.
( aString := anIDataObject renderFormat: 'String' ) isNil
    ifTrue: [ MessageBox warning: 'no string available' ].
Transcript show: 'Got a string via COM data transfer:'; cr;
    tab; show: aString; cr.
anIDataObject release.
```

For additional information about other supported data transfer formats, consult Microsoft’s programmer’s reference manuals.

COM Event Support

The COM architecture defines a generalized event model based on the dispatch technology, which is the foundation of the automation technology. The COM event system provides for connecting an object that generates events with clients interested in receiving notifications of those events through a dispatch interface connection.

Overview of Connectable Object Technology

A COM object that generates events supports an outgoing dispatch interface for each event set that it supports. An interested client registers to receive event notifications by constructing a matching dispatch interface that it connects to the event source. This allows the object generating the events to notify clients when the event occurs by invoking the appropriate member function in the client's incoming dispatch interface.

When a client is no longer interested in receiving event notifications, it disconnects the notification channel that it has provided the event source object.

Overview of Receiving COM Events in VisualWorks

You can create an event sink through which event notifications are received by creating an instance of COMEventSink. The event sink is configured by providing it with the IID of an event set interface and specifications describing the supported events. As when configuring an dispatch driver to use an automation object from another application, there are services provided in the COMDispatchSpecificationTable class, which construct the dispatch member specifications for an event set interface from its type library information. There are also development utilities provided in the COMAutomationTypeAnalyzer class, which can be used at development time to explore dispatch interface definitions in type libraries and cache event table specifications used to configure an event sink.

After you have configured an event sink and are ready to receive event notifications, you establish a notification channel to connect the event sink to an event source object. The COMEvent-Sink object you have created handles the mechanics of establishing a connection with the event source object.

Once the event notification channel between the source and sink objects is connected, the event sink receives COM event notifications from the event source object through its incoming dispatch interface. When an event sink receives an event notification from the event source object to which it is connected, it provides suitable transformations between the dispatch value encodings and appropriate Smalltalk values for all event argument values and triggers a Smalltalk application event.

The names of the application events that are triggered by the event sink are determined by the selectors that are defined in the dispatch member specifications that are provided for each event when you configure the event sink. In your application, you can configure a COMEventSink and then register event handlers for COM events using the standard facilities of the Smalltalk application event system.

When you no longer want to receive event notifications, you must disconnect the event sink from the event source object. The COMEventSink object handles the mechanics of destroying the connection to the event source object.

Using a COM Event Sink

Using an event sink to receive COM event notifications in VisualWorks involves the following steps:

1. Configure the event sink.
2. Connect the event sink to the event source object.
3. Register Smalltalk application event handlers on the event sink to receive notifications.
4. Disconnect the event sink from its source object when done.

Configuring an Event Sink

To configure an event sink, you need the IID of the event interface and a COMDispatchSpecificationTable containing the specifications of the supported events. The information needed to configure an event sink can be constructed dynamically when the event sink is being constructed, if suitable information is available at execution time from the event source object itself or a type library. Alternatively, the specifications can be determined during development and defined by a specification literal, which is used to efficiently configure the event sink at execution time.

Event specifications can be constructed dynamically directly from the event source object if it supports the interface IProvideClassInfo, which allows a client to access the type information about the component object class. Obtaining event type information directly from an event source object is done using the getEventTypeInfoOf: service of COMDispatchSpecificationTable. The argument is any interface that you already have on the event source object. If the object supports an event interface and can directly provide type information about its events, the getEventTypeInfoOf: message returns an ITypeInfo interface from which the event set specifications can be derived. If not, this message returns nil, in which case you must either attempt to obtain the event set ITypeInfo interface from some other source such as a type library or use specifications that you have constructed at development time. Once you have obtained the event set ITypeInfo interface, you can pass it to the constructEventSinkSpecificationTable: service in COMDispatchSpecificationTable to construct a COMDispatchSpecificationTable containing the event specifications.

For example, suppose you have created an object and have obtained some interface anInterface on the object (typically but not necessarily IUnknown or Dispatch). The following code fragment demonstrates constructing an event sink on this object by configuring the sink with dynamically constructed type information obtained directly from the event source object.

```
| anITypeInfo eventSpecifications |

 " construct the event specifications from the object's type info "
anITypeInfo := COMDispatchSpecificationTable getEventTypeInfoOf:
    anInterface.
anITypeInfo isNil
    ifTrue: [ ^nil ].  " no type info or not an event source object "
[
eventSpecifications := COMDispatchSpecificationTable
    constructEventSinkSpecificationTable: anITypeInfo.
] ensure: [
    anITypeInfo release ].

^COMEventSink iid: eventSpecifications iid
    specificationTable: eventSpecifications.
```

An alternative to dynamically constructing the event specifications at runtime from an ITypeInfo interface of the event source object is to construct the specifications during development and cache them in a dispatch table specification literal, from which the COMDispatchSpecificationTable needed to configure the event sink can be efficiently instantiated at runtime. Instantiating the event specifications from a literal specification is faster than constructing them from an ITypeInfo each time you connect to an event source object.

To construct an event specification literal at development time, you need the ITypeInfo interface that describes the event set. As when constructing the event specifications at the time you connect an event sink to an event source object, you can obtain the type information about the event set either by creating an instance of the event source object and using the getEventType-InfoOf: service of

COMDispatchSpecificationTable as described above or by obtaining the appropriate ITypeInfo from a type library containing the specifications of the event source object. Event set interfaces are marked as outgoing dispatch interfaces (the IMPLTYPEFLAG_FSOURCE flag). If the object supports multiple event sets, the primary event set is distinguished by being marked with the IMPLTYPEFLAG_FDEFAULT flag. The generateEventTypeInfoSpecification: service of COMAutomationTypeAnalyzer is used to generate a dispatch table specification literal for the event set.

Note that the event specification table is indexed by DISPID of the event members, because the COM event mechanism is based on the source object triggering events in the dispatch interface that is supported by the event sink. This is denoted by a dispatch specification table whose key is the symbol #memberID.

After you have used COMAutomationTypeAnalyzer to generate the event set specification literal, you need to save it so that you can obtain it at runtime. Typically, you save event specifications by creating a method containing the dispatch specification table literal in your application class that establishes the event sink. Event specifications are instantiated by sending the message decodeAsLiteralArray to the literal array.

For example, suppose that in your class that is going to construct an event sink you have saved the event specifications literal in a method named eventSpecificationLiteral. The following code fragment demonstrates constructing an event sink configured for the event source object from the cached event set specifications that you obtained at development time.

```
| eventSpecifications |
eventSpecifications := self eventSpecificationLiteral
   decodeAsLiteralArray.
^COMEventSink iid: eventSpecifications iid
   specificationTable: eventSpecifications.
```

Connecting an Event Sink

After an event sink has been created and configured with the event specifications, you must connect the sink to the event source object in order to enable notifications to be received. The event sink is connected by sending it the establishConnectionTo: message, providing any interface on the event source object as an argument. The event sink handles the mechanics of establishing a connection to the event source object and maintaining the information needed to terminate the connection when you are done with the event sink.

Continuing the example from the previous section, where anEventSink was created and configured so that it can be connected to an object that has anInterface, the connection is established by simply evaluating the following:

```
anEventSink establishConnectionTo: anInterface.
```

Registering Handlers on an Event Sink

To receive notifications of events from your event sink after it is connected to the event source object, you configure the event sink with the processing actions for the events of interest to you by registering event handlers using the standard when:send:to: family of messages of the Smalltalk application event system. The event names that you use for registering event handlers are determined by the selectors defined in the event specification table, which by default are formed from the textual event name with anonymous keywords appended as needed to denote argument values.

Following the convention of the application event system, an event name is a symbol formed by one or more keywords denoting event argument values. This is of course the same convention you are familiar with for keyword message selectors in Smalltalk. For example, typical event names might be:

LButtonUp:with:	LButtonUp - integer x, y arguments
Loaded	Loaded - no arguments
Saved:	Saved - string argument with file name

Generally, event names are spoken according to the basic textual keyword from which the event name is formed, e.g., the Loaded event and the Saved event, but the event handler that you register to specify the action to evaluate when an event of interest is triggered uses the keyword event name symbol, e.g., #Loaded and #Saved:.

For example, suppose you have configured an event sink and connected it to an event source that triggers notification events when the object is loaded from or saved to a backing file. These hypothetical events might be called Loaded, with no arguments, and Saved, with the file name provided as argument. To register handlers for these events, you would write something like the following:

```
anEventSink
    when: #Loaded
        send: #ringBell to: Screen default;
    when: #Saved:
        evaluate: [ :name |
            Transcript show:'Saved to: ', name; cr ].
```

Of course, in a real application you would probably register handlers that send messages to your application object and cause something useful to happen.

If you want to have all event notifications from the event source routed to a single destination, you can register a handler for the event sink relay event eventNotification:arguments:. The first argument of this event is the name of the event, while the second argument is an array containing the argument values provided by the event (if any). However, carefully consider the performance implications before using this generic event notification relay mechanism, since the dispatch parameters from the event source must be realized as a Smalltalk arguments array for each notification, even if the relayed notification is not used for any useful purpose.

The generic relay event can be useful for development tools, such as the COMEventTraceViewer, which is provided with COM Connect to enable you to hook up a trace window to an event sink in which you can view trace information about each event and its arguments as COM event notifications are received. However, be extremely cautious about using the facility in a real application, as it might cause noticeable performance degradation of your application.

Disconnecting an Event Sink

When you are done using an event sink and are no longer interested in receiving COM event notifications from the event source object, you must disconnect the event sink from the event source object:

```
anEventSink releaseConnection.
```

Note: In general, it is probably advisable to disconnect the event sink before releasing the last interface that you hold on the object. It is probably unwise to assume that holding an event sink keeps the object alive. It is a good idea to manage disconnecting the event sink with your application shutdown logic or other application logic that releases the event source object.

Note: While in most cases you send the release message to a COM resource when you are done using it, you do not need to do so when you are done using an event sink. An event sink is simply a Smalltalk object you are using that happens to use COM services in order to implement its functions. You use a sink by connecting and disconnecting it from an event source object that provides notifications; it is not an interface that you acquired and thus you do not need to release it when you are done.

VisualWorks Extensions

The COM Connect software includes a small number of extensions to existing VisualWorks classes and some new general-purpose facilities that are used by COM Connect. The support components that are brought in with COM Connect and used by the COM Connect software, but which are not inherently COM-specific facilities, are discussed in this section.

Image Management Services

The ImageConfiguration class represents the configuration of the image, notably whether the image is a development configuration or a deployment image. The command line arguments for this invocation of the image are also part of the public protocol.

By default, an image is considered to be a development image. When you prepare a deployment image for delivery, you can install the runtime image configuration setting by evaluating the expression:

```
ImageConfiguration isDevelopment: false.
```

To test whether the image is configured as a development image or a deployment image, evaluate the expression:

```
ImageConfiguration isDevelopment
```

The ImageManager class coordinates the startup, save, and shutdown of the image. Other subsystems can arrange to be notified of these operations by configuring application event handlers.

Image startup notifications are provided by the ImageManager coreStartupCompleted and startupCompleted events, where the former is triggered early in image startup (prior to the installation of the window system) and the latter is triggered after the base system startup processing is completed. To configure additional image startup processing that should be performed only in a development image, register a handler for the ImageManager developmentStartup event. To configure additional image startup processing that should be performed only in a deployment image, register a handler for the ImageManager deploymentStartup event.

Image shutdown notifications are provided by the ImageManager shutdown event. Application-specific processing, such as cleanup or release of external resources, can be configured by registering handlers for this event.

Image save notifications in a development image are provided to enable you to configure processing that needs to be involved at image save time. When an image save is about to be initiated, ImageManager triggers the veto-able confirmSave event to announce the impending save operation. If you need to be involved in a decision about whether an image save can be allowed, you register a handle for this event. If for some reason you need to veto the proposed image save operation, you can do so by evaluating the following expression:

```
ImageManager abortSaveImage.
```

If the proposed image save operation has not been vetoed, the image save operation is initiated by triggering the aboutToSave event. Any processing that your application needs to do when an image save operation is going to proceed, but has not yet started, should be installed by configuring a handler for this event. Finally, ImageManager triggers the saveCompleted event when the image save operation has been completed.

Note: The ImageManager events that provide image lifecycle operation notifications are derived from existing dependent notification capabilities provided by the VisualWorks ObjectMemory class. As noted in the previous section that reviewed the application event system, however, configuring your processing using application event notifications is typically more straightforward than expressing the equivalent interaction through a dependent notification, which requires your application to handle the routing of the notification according to the source object and aspect information provided with the notification. It is generally much easier to configure your image processing using the ImageManager events, letting the ImageManager handle the routing and dispatching, so that you need only implement the actual notification handler logic of interest to your application.

User Interface Extensions

A few useful user interface extensions are provided with COM Connect.

The FileDialog class provides a small set of standard services for opening a dialog to obtain a file path name from the user. FileDialog can be used in your application when an interaction with the user is desired, to obtain the name of a file to be opened or the name of the file to which data is to be saved.

Services for obtaining the name of a file to be opened are:

```
FileDialog openFile.
FileDialog openFile: '*.txt'.
FileDialog openFileTitle: 'Open File' pattern: '*.txt'.
```

Services for obtaining the name of a file in which data is to be saved are:

```
FileDialog saveFile: 'Untitled.txt'.
FileDialog saveTitle: 'Save As' fileName: 'Untitled.txt'.
```

Platform-specific bindings of the FileDialog protocol to host UI dialogs can be provided, as desired. If no platform-specific binding is defined, a standard VisualWorks emulated dialog is used to support the operations.

The MessageBox class provides a small set of standard services for opening a dialog to display a message to the user or to obtain confirmation for some request.

To display a notification to the user:

```
MessageBox message: 'This is a message for you.'.
MessageBox warning: 'Consider yourself warned!'.
MessageBox notify: 'Title String' withText: 'Message text...'.
```

To obtain a confirmation from the user:

```
MessageBox confirm: 'Is this what you want?'.
MessageBox threeStateNotify: 'Title String' withText: 'Message
   text...'.
```

Platform-specific bindings of the MessageBox protocol to host UI dialogs can be provided, as desired. If no platform-specific binding is defined, a standard VisualWorks emulated dialog is used to support the operations.

The TextWindow class combines the Smalltalk expression evaluation support of a workspace with the file-backing services of a file editor view, without any limitation on the size of the text that can be contained in the view other than what the creator of the text window view might want to specify.

To open a text window:

```
TextWindow open.
TextWindow label: 'TextWindow Example '.
```

To open a text window with some initial text contents displayed in the view:

```
TextWindow openOn: 'This is some initial text'.
TextWindow openOn: 'This is some initial text'
   label: 'TextWindow Example'
```

To open a text window on the contents of a file:

```
TextWindow openOnPathName: 'readme.txt'.
```

The ListDialog and MultiSelectListDialog classes provide services for allowing a user to make a selection from a list of choices presented in a dialog. A list dialog can be configured with a title and one or more lines of explanatory text, in addition to the list of choices. The MultiSelectListDialog provides additional operation buttons to facilitate user actions like selecting all or none of the list items in a multiple-choice list dialog.

Working With External Structures

External data structures are modeled in Smalltalk using the DLL & C Connect definition facilities and external data support facilities. To make it easier to work with host data structures in a VisualWorks development image, inspectors are provided for C structure type definitions (instances of CCompositeType) and structure instances that are allocated in either Smalltalk memory or in external heap storage memory (CComposite and CCompositePointer instances).

In some cases, it is useful to create a structure wrapper class in Smalltalk to encapsulate operations on the contents of external structures or to provide associated services to create and manipulate such structures. The ExternalStructure class is used to wrap instances of external structures, which might be allocated in either Smalltalk object memory or in external memory such as the heap storage accessed with the malloc family of external data services.

ExternalStructure can either be instantiated directly, to wrap a specific structure instance, or used as a framework for implementing subclasses, which provide customized structure wrappers with an extended protocol appropriate for a particular external structure. ExternalStructure provides a few convenience operations that make it easier to work with external structures in a Smalltalk application. Facilities are also provided for configuring an ExternalStructure instance with protocol adaptor specifications, which allow you to automatically wrap member access operations with additional processing, to provide additional semantics for manipulating structure members.

DLL & C Connect Extensions

The DLL & C Connect extension included with COM Connect provides improved support for using Boolean values when working with external structures and functions. Improved support for certain standard host platform procedure call conventions is also provided. These DLL & C Connect improvements are supported by syntax extensions that you use in external data and function declarations, as well as by enhanced support mechanisms in the VisualWorks object engine and run-time support services.

The syntax for defining external data types is extended to support a new __bool modifier for integer data types. A structure member or external function argument that is defined as a __bool integer value accepts Smalltalk Boolean values directly. Without this feature, when you work with a value that is semantically a boolean but declared in the C fashion as a standard integer value, you are responsible for providing integer values corresponding to the standard C TRUE and FALSE values whenever setting a structure member or providing a function argument.

Similarly, you must work with the integer value that you obtain back from a structure member or a function call. Often, this results in providing explicit conversion between a Smalltalk Boolean and the corresponding C integer constant at the point where the external function or structure is used, so that the rest of your application need not be concerned with mixing representations.

The new `__bool` modifier enables you to work directly with Boolean values in external structures and functions, and let the VisualWorks Object Engine external data support handle any necessary mapping to the C encoding, automatically. Any integer data type can be marked as a boolean value, using the following syntax as a DLL & C Connect declaration:

```
[const] __bool char
[const] __bool short
[const] __bool int
[const] __bool long
[const] __bool long long
```

The structure member accessing operations `memberAt:` and `memberAt:put:` accept and return a Smalltalk Boolean for a `__bool` member. When you invoke external function with an argument declared as a `__bool` value, you simply provide the argument as a normal Smalltalk Boolean, and no conversion is needed to ensure that the argument value is mapped to the corresponding C integer encoding.

In Win32 generally and almost uniformly in COM, a standard data type and function definition convention is followed for defining how a function should return a status code and result values to its caller. The data type HRESULT is a standard data type with a well-defined specification of how status values are encoded to indicate success or failure of the operation, the facility that reported the status, and a specific status code value. By convention, a function is defined to return HRESULT as the function value, with any output values returned by the function to its caller declared as OUT arguments to the function.

The HRESULT return type is now supported by a new integer data type modifier `__hresult`, which provides improved support for detecting and reporting external function call failures. A function that is declared with an `__hresult` return value returns from an external function call with a SystemError exception, generated automatically by the VisualWorks Object Engine, if the HRESULT return value indicates an error code.

The SystemError for an HRESULT error code is named #'hresult error' and the parameter value is the HRESULT code returned by the external function. The HRESULT error code from the external access failure exception can be handled immediately in failure code that you write for a specific function declaration. More generally, the HRESULT failure can be mapped to a Smalltalk exception in the standard externalAccessFailedWith: processing that is used when an external access failure is reported by the object engine for an external function call. This automatic mapping into a Smalltalk exception eliminates the need to write status code checking logic explicitly everywhere you call an HRESULT function, and it is a far more efficient mechanism for detecting and reporting error conditions.

Win32 Support Facilities

To enable COM Connect support on Windows platforms, some Win32 support facilities are provided to enable access to basic Win32 system services.

The Win32RegistrationDatabase class provides services for accessing and manipulating the contents of the Win32 system registration database.

The Win32ClipboardInterface class provides services for accessing the system clipboard and clipboard format constants on a Win32 system. Standard Win32 CF_ clipboard format constants are defined in the Win32Constants pool.

The Win32FileDialog class provides a Smalltalk binding to the standard Win32 file dialog. The Windows file dialog is generally used indirectly, through the platform-independent FileDialog class. However, clients that are willing to accept an explicit dependency on the Win32 platform in their applications can choose to use the Win32FileDialog class directly, in order to exploit enhanced capabilities that are specific to the Windows dialog.

The Win32Message class provides a Smalltalk binding to the standard Win32 message box dialog. The Windows message dialog is generally used indirectly, through the platform-independent MessageBox class.

The Win32GlobalMemoryAddress is used to allocate memory using the Win32 global memory allocator. Instances of Win32GlobalMemoryAddress represent an external memory address, which references memory allocated outside Smalltalk object space, using the Win32 global memory allocator. A Win32 global memory address can be used interchangeably with a CPointer that references external memory allocated from the Smalltalk external memory heap. The standard Win32 GMEM_ constants used to specify the option flags that control a global memory allocation operation are defined in the Win32Constants pool.

COM Host Binding Framework

COM interfaces are typically provided by COM servers external to your VisualWorks COM application. To use such an interface in a Smalltalk COM application, it must be wrapped by Smalltalk classes and methods.

This section describes the host binding implementation framework provided by COM Connect for representing interfaces for use in an application. It is assumed in this section that you already know how to access COM interfaces from another language, such as C, and so you only need specific pointers to implementing the same functionality in Smalltalk.

Implementing a host binding wrapper for a COM API or interface requires that you first use the capabilities of DLL and C Connect to create the necessary data type definitions in an ExternalInterface class. Specifically, all COM data type definitions are created in the COMExternalInterface class. This pool of C type definitions provides the context for all API functions, which are wrapped by function declarations implemented in subclasses of COMDynamicLinkLibrary, and interface definitions, which are wrapped by function declarations implemented in subclasses of COMInterfacePointer. Refer to the DLL and C Connect Guide for additional information about the external interface capabilities that VisualWorks provides and available developer support tools.

Note that the material in this section is of interest only if you need to provide support for an API or COM interface that is not already provided by COM Connect. COM applications are insulated from this lowest layer of the COM Connect architecture by COMInterface interface wrapper classes and other Smalltalk service classes that expose various COM facilities.

COM Data Structures

COM data structures are usually accessed through structure wrapper classes, which are implemented as subclasses of the COMStructure abstract class. These classes provide methods for manipulating COM-specific data types in structure fields. Structures are allocated using the class messages defined in COMStructure.

COMStructure can be used as a wrapper for any host structure. A customized structure wrapper class can be created as a subclass of COMStructure if you want to support convenience protocol methods, add services that encapsulate complex operations on the contents or the structure, or provide extended semantics for releasing the structure.

COM Function Binding Classes

The COM Connect framework allows accessing COM objects defined outside Smalltalk, using callout facilities, as well as implementing COM objects in Smalltalk, allowing call-in access by client objects. A typical COM application involves an exchange between externally defined and internally defined COM objects.

The COM Connect support framework uses classes described in the following sections, which discuss implementing wrapper classes that provide bindings for COM interfaces and API's.

COMDynamicLinkLibrary

The COM API functions are supported by API primitive methods in subclasses of COMDynamicLinkLibrary. The abstract superclass provides utility services to support native API function invocation. A COM DLL defines the API function primitives that invoke a native host API function directly and manage the low-level host data type transformations involved in an external function call.

Subclasses of COMDynamicLinkLibrary have the following responsibilities:

- Implement public protocol for the API functions.
- Perform argument transformations between Smalltalk objects and host data types.
- Implement methods to call API function primitives.
- Signal COM exception conditions to a Smalltalk caller.

COM API functions should usually be invoked through messages supported by the appropriate COMInterface class.

COMInterfacePointer

An interface has a binary representation in memory consisting of a pointer to a pointer to a list of functions (the interface VTable). Instances of COMInterfacePointer contain the first of these pointers, which indirectly points to entries in the VTable. An interface pointer class defines the COM function primitives that directly invoke a native host function in the interface and manage the low-level host data type transformations involved in an external function call.

Subclasses of COMInterfacePointer have the following responsibilities:

- Implement public protocol for the interface functions.
- Perform argument transformations between Smalltalk objects and host data types.
- Implement methods to call COM function primitives.
- Signal COM exception conditions to a Smalltalk caller.

The COMInterfacePointer class hierarchy should parallel the COMInterface hierarchy, providing the same inheritance structure. Each interface pointer class is uniquely identified by its IID, which can be obtained by sending the iid message to the interface.

The public protocol of an interface pointer class should consist of exactly the interface of the interface functions. “Civilizing” functions should not be defined in these classes.

Calling a COM function is similar to making an API call, but employs the special COM calling convention for invoking a function entry point in an interface VTable.

COMInterfaceImplementation

Interfaces are implemented in Smalltalk as subclasses of COMInterfaceImplementation. These primarily provide call-in services allowing external clients to invoke a COM object implemented in Smalltalk.

The COMInterfaceImplementation classes provide the call-in binding to allow external clients to invoke interfaces supported by a COM object implemented in Smalltalk. Since all interfaces support IUnknown, all implementations are defined under IUnknownImplementation.

The interface implementation callback mechanism relies on the C type definition of the VTable structure containing the function pointers of the interface. VTable structure definitions are created in the COMInterfaceVTableSignatures class, using the standard facilities of DLL and C Connect to define the C structure. The VTable structure must reflect the underlying C calling convention, with an explicit first argument (typically named This) representing the interface implementation.

An interface implementation manages the interface data structure in external memory, and dispatches function processing to support invocation of its interface functions.

The COMInterfaceImplementation class hierarchy should parallel the COMInterface hierarchy, providing the same inheritance structure. Each interface implementation class is uniquely identified by its IID, which can be obtained by sending the iid message to the interface.

5

COM Connect Development Tools

COM Connect provides several tools to assist you in developing and debugging COM applications in VisualWorks Smalltalk. The tools can be launched in the VisualLaunch from the **Tools** > **COM** menu.

A number of useful tools for COM developers are also freely available from Microsoft. Refer to the MSDN website for information about and access to these tools.

COM Resource Browser

A COM application that you develop typically uses a variety of services provided by other COM objects or the COM platform support. At any point during the activity of your COM client processing, you own a number of interface pointers to objects you have created or acquired. You might also own COM memory that you have allocated or acquired. If your application publishes COM objects, you might also be supporting a number of interfaces that are exported to clients outside of your Smalltalk image.

The COM Resource Browser allows you to browse the COM resources that are currently in use by or exported from your Smalltalk COM application. It can be very helpful to analyze the COM resource usage of your application at various points in time, which can help you understand the interactions between your application and other COM applications. It is also useful for tracking down resource leaks, in interfaces or memory, of which you have acquired ownership but have not released.

To open the COM Resource Browser, select **Tools > COM > Browse Resources** in the Launcher.

Another useful expression opens the resource browser only when resources are currently in use by your COM application session:

```
COMSessionManager sessionHasResources
   ifTrue: [ COMResourceBrowser open ].
```

The browser displays five lists.

Special resources:

Special COM resources that are managed by the basic COM Connect infrastructure on behalf of all clients in the session. The COM task memory allocator is most commonly listed.

Owned interfaces:

The interfaces used by your application.

Owned memory:

The addresses of any memory allocated by the COM task allocator that you own.

Exported interfaces:

Interfaces defined by object implemented in Smalltalk and exported to external clients over their lifetime.

Exported objects:

COM objects implemented by this Smalltalk application that are currently in use.

The browser also has two buttons:

Update lists

Updates the resource browser view when you have performed activity elsewhere in your application that affects the COM resource usage state of the session. Typically you must refresh the view either when you run something that acquires COM resources or after you run some portion of your application that releases resources it was using.

Clean up lists

Updates the resource browser view after performing a garbage collect to ensure that spurious references are eliminated.

Inspecting Resources

You can inspect any owned or exported resource by selecting it and invoking the **Inspect** command from the context menu, or by doubleclicking on it.

The **Registries** menu opens inspectos on the underlying resource tracking registries used to support COM resource management.

Releasing Resources

To free a resource, select the **release** command on the context menu. This is useful when cleaning up resource leaks or recovering from the effects of a damaged object you are developing. However, you must use the release capability with extreme caution, and with a good understanding of what you are doing.

Releasing interfaces from your application can cause unexpected failures. Even more dangerous, forcibly releasing an exported object that is still in use by another application can have serious consequences, including crashing the client of your object or hanging your system.

The **Cleanup** menu provides operations that release mass quantities of owned or exported resources. These brute-force cleanup operations are even more dangerous than the individual resource release operations, and they are prone to failure due to subtle order dependencies that can occur as a result of interactions between owned and exported resources. Use these global cleanup operations with extreme caution.

Common Resources

The IMalloc class obtains and manages a single reference to the COM task memory allocator, which is shared by all clients in the process that need to allocate COM memory for any reason. If you allocate COM memory, either directly through the IMalloc functions available though the shared interface reference:

```
IMalloc taskMemoryAllocator
```

or as a consequence of allocating external COM memory through services provided by the COMMemoryAddress and COMStructure classes, you see an entry in the special resources list of the COMResourceBrowser.

The IRunningObject table also manages a special shared interface reference to the system running object table, which is used for certain operations involving monikers.

COM Trace Manager and COM Trace Viewer

The COM Trace Manager and COM Trace Viewer tools provide dynamic insight into the behavior of your application and its interactions with other COM objects.

COM tracing capabilty can be installed/uninstalled and enabled/disabled during application development, so that the overhead of tracing is incurred only when you need it. When tracing is installed, you can dynamically enable and disable tracing on specific types of function calls and individual interfaces to observe the desired behavior.

COM Trace Manager is a prerequisite for the viewer. To install it, select **Tools > COM > Install Trace Manager**. You can uninstall it later with **Tools > COM > Uninstall Trace Manager**.

The COM Trace Viewer tool provides the primary user interface to the COM Connect tracing capability. To open the COM Trace Viewer, select **Tools > COM > Trace Viewer**.

The viewer allows you to set several global flags with a few checkboxes:

- **enable tracing** - controls whether tracing is currently enabled
- **trace callout** - Trace outgoing interface function calls
- **trace callin** - Trace incoming interface function calls from external clients
- **trace internal calls** - Trace internal interface calls between Smalltalk clients and objects in the same image.

In most cases, you will be interested in the outgoing and incoming interface function calls, since these two categories of interface function calls allow you to observe the interface between your application and external COM clients or server objects.

The **Callout options** and **Callin options** buttons allow you to select which function (classes) specifically to trace, providing better focus to the trace. This allows you to configure the trace settings of your system so that you can observe specific interactions, while ignoring interfaces that represent activity in which you are not interested (at least at the moment).

Because interface tracing can produce voluminous output, you typically use the COMTraceViewer selectively. Generally, you want to turn the global **enable tracing** switch off while configuring the desired trace option settings and getting your application to the state at which you want to observe its activity. When you are ready to collect trace feedback, turn on the global tracing switch and perform the operations of interest in your application. When you are done collecting information, simply turn off the global tracing switch again.

COMInterfaceTraceAdaptor

An interface trace adaptor is interposed between the caller and callee when interface tracing is enabled. A trace adaptor supports the message protocol of the interface that it is tracing and simply forwards the function call to the real implementor of the interface after recording suitable information in the trace log, according to the current trace option settings you have configured.

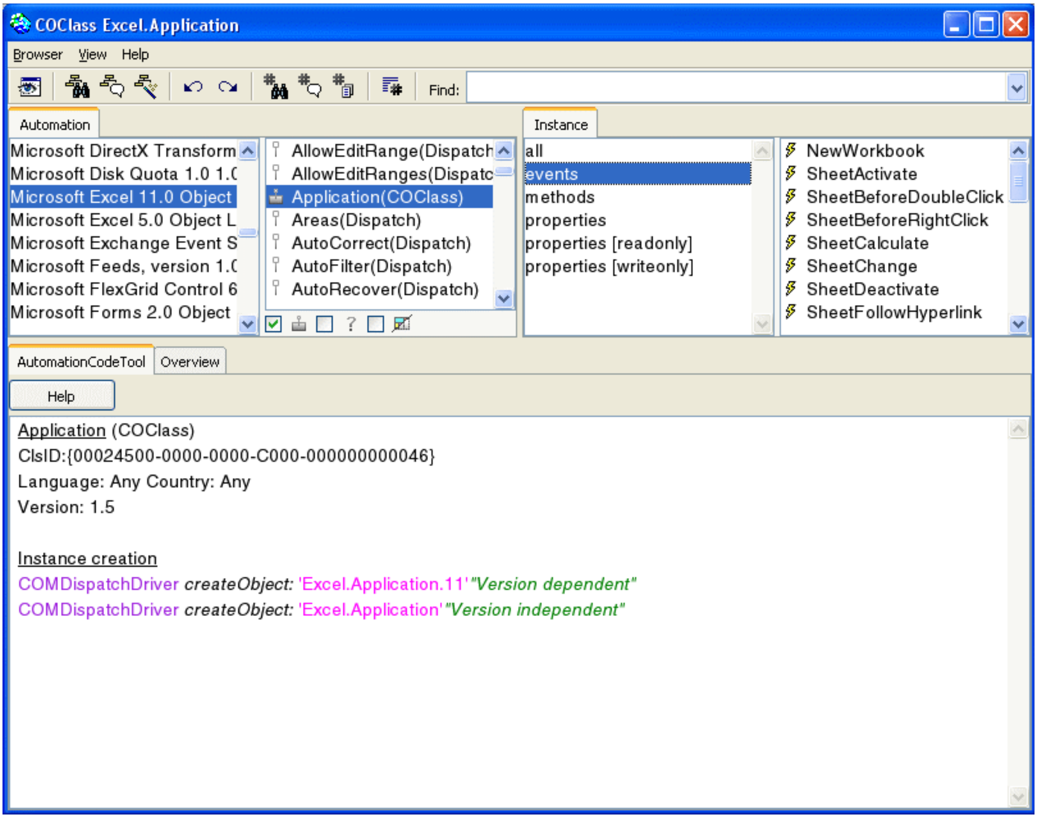
The interface function call trace output is provided by a set of special development support classes that provide both generalized and customized interface tracing capabilities. Interface tracing is provided by subclasses of COMInterfaceTraceAdaptor.

The default trace of argument and result values that is provided is generally adequate for many interfaces. If you want to customize the tracing output for a particular interface, you can implement a trace adaptor class and simply add implementations for any functions whose default tracing you want to override.

Automation Browser

The Automation Browser is an extension to the system browser that supports viewing type libraries and their contained type descriptions.

To open the Automation Class Browser open select **Tools > COM > Browse Automation Classes…** from the Launcher menu.



The browser consists of four upper list panes and a special source description pane in the lower half of the window:

Type Library pane

The first, leftmost list pane contains all type libraries registered in the system. Type libraries contain detailed information about objects and types provided by a specific COM server.

Types pane

The second list pane contains the types described for the type library selected in the type library pane. Types may be

- CoClasses
- Interfaces

- Enumerations
- Aliases
- Structures
- Unions

Each type is identified visually by an icon. Below this pane three check boxes select which types are listed: Coclasses and Interfaces (including Dispatched); all other types; all types that are system types or are marked as hidden.

Protocols pane

The third pane contains the protocols of the selection in the Types pane. These are not protocols in the Smalltalk sense but static categories for each kind of type. While, for example, Interfaces and CoClasses have the protocols “methods” and “properties,” Enumerations will have the protocol “constants.”

Members pane

The rightmost list pane displays the members of the selected type depending on the selection in the Protocol pane.

Source Description pane

Selecting a member in the Members pane displays detailed informationin the lower half of the window. Unlike the System Browser, the pane does not allow editing of its contents. The Source Description pane has an option to explain selected text, such as parameter types of methods.

Usage Features

The browser supports these features.

Explain

Explain on the context menu attempts to provide a short explanation for whatever is selected in the Source Description pane and will allow you to open a new Browser on the selected element, if the element is a custom type provided by an Automation Server. This feature operates across type libraries and for hidden types.

Search

You may search for a type by using the search input field in the top-right of the browser window. As in the System Browser, if you enter the name of a type to search for and presses Enter the Browser will list Types it finds in a dialog. Select the Type to navigate to. The search string may contain the name of a type library, (e.g., a possible query may be "Word.Application"). If nothing is found, the user will be asked whether to search for methods. Method search can be enforced by adding an hash sign (#) in front of the search phrase.

Search References

Two context menu items allow searching references to a specific type, which are occurrences of a type inside another type or as parameter/return value type. Type aliases will be resolved and references to the alias will also be listed. **Local References** lists all references inside the type library in which the type is defined. **Global References** searches all type libraries for references that are registered on the computer. Searching all global references may take some time.

Instance creation

The browser allows creating instances of a coclass and directly interacting with it in an inspector. To do this, select a coclass type in the Type list pane and select "Create Instance and inspect" from its context menu. An inspector will open on the created coclass.

Implementors

It is now possible to browse implementors of a method or property. There are two versions, local implementors and global implementors. A search for local implementors conducts a search within a single type library whereas a global implementors search searches all type libraries registered on the computer.

Inspector Extensions

The Inspector has been enhanced to show additional information about instances of `COMDispatchDriver`.

The **Basic** tab has been extended to support the display and update of Automation properties of the inspected object. You may set properties of the remote server object in an inspector just by typing a new value in the inspector and accepting it.

Most application classes have a Visible property. When such an object is created, this property usually is set to `false`, which means that the application window is invisible. When the value is changed to `true` and accepted, the application window should become visible.

Automation Member Description tab

In addition to Automation property display support, the inspector shows additional tabs, depending on the kind of Automation object being inspected.

The **Automation Member Description** tab is displayed for all kinds of classes represented by an `AdvancedDispatchDriver`.

On the left it contains a list of all members of the Automation object. That means methods, properties and events.

On the right, VisualBasic and Smalltalk representations of the selected member will be displayed. The actual kind of information displayed will vary depending on the type of the selected member. While for methods the call code will be displayed, for events the inspector will display the code for being informed about the occurrence.

Both VB and Smalltalk representations are displayed because specific types of information can be more easily acquired from either of them. For example, the Smalltalk syntax does not say anything about the result type but provides detailed information about classes which may be used to pass arguments.

The tab also contains a help button which opens a help window on the selected member if there is a help file available.

An additional feature of the tab is the Send/Send-And-Dive Functionality. This Automation version of the send functionality allows calling parameterized methods and properties by providing a dialog which allows entering parameter values. An Automation send can be performed by selecting **Send/Send and Dive** from a member's context menu.

Automation items tab

For collection coclasses, an additional tab will be displayed which provides access to all items of the collection.

Editing items is supported in the inspector (if the coclass supports it) but make sure the correct type of object is used.

COM Automation Editor

The COMAutomationEditor is a simple workspace window that can be connected to the dispatch interface of a COM automation object and allows you to interactively evaluate Smalltalk expressions that access properties and invoke functions of an automation object to which it is connected.

The sample COM random number generator discussed elsewhere in this documentation to demonstrate the basic techniques of implementing a COM object and supporting an interface can also be used through COM automation.

The COMAutomationRandomNumberGenerator is a subclass of the COMRandomNumberGeneratorObject object that allows the functions of the IRandomNumberGenerator interface to be accessed by COM automation by adding support for the IDispatch interface. A simple test driver class that demonstrates writing interface test cases for the RNG object also provides a utility operation to open a COMAutomationEditor on an automation-enabled RNG object. You can open the automation editor on an automation-enabled RNG test object by evaluating the following expression:

```
COMAutomationRNGTestDriver testInAutomationEditor.
```

Within the text pane of the automation editor view, you can type and evaluate expressions that send messages to the automation object to which the view is connected. By default, the automation object is referenced by the dispatcher variable name. You can evaluate the following sample expressions to exercise the random number generator COM object's functions through the automation dispatch interface:

```
" get the next RNG value "
dispatcher invokeMethod: 'Next'.

" get the current property values "
dispatcher getProperty: 'LowerBound'.
dispatcher getProperty: 'UpperBound'.
```

```
" change the bounds of the RNG sequence "
dispatcher setProperty: 'LowerBound' value: 10.
dispatcher setProperty: 'UpperBound' value: 20.
```

The automation editor can be connected to any automation object that supports the IDispatch interface. It provides the interactive flexibility, standard within the Smalltalk development environment, so that you can use COM automation objects to dynamically explore the results of sending messages to an object in order to observe an object's behavior and results.

COM Event Trace Viewer

The COM Event Trace Viewer allows you to obtain debug tracing information from COM event notifications. The COM Event Trace Viewer can be hooked up as an event sink on any event-generating COM object to dynamically observe COM event notifications. A trace control toggle switch allows you to dynamically enable and disable event notification tracing.

To open the COM Event Trace Viewer on a particular interface, send:

```
COMEventTraceViewer openOn: anInterface
```

COM Automation Type Analyzer

The COM Automation Type Analyzer provides a number of services for analyzing COM type libraries in order to generate descriptive reports and Smalltalk bindings for use in automation applications, such as specification tables used for configuration. This is strictly a programmatic tool, invoked by evaluating various expressions; this tool has no user interface.

The services provided by the COM Automation Type Analyzer are discussed in the section on implementation of COM automation objects that support the IDispatch interface under Implementing Automation Objects.

Interface Class Generation Tools

To create or use a COM interface, you need to define interface wrapper classes that provide your client or COM object with a Smalltalk binding for the functions defined in the interface. While

COM Connect provides interface wrapper classes for many of the standard COM interfaces, you might need to create new interface classes if you want to use a standard COM interface for which a Smalltalk binding is not already provided.

You must also create interface classes when you define a new interface for a COM object that you publish. For example, if you published an object through Automation using the COMAutomationServer provided with COM Connect to provide IDispatch support, and you want to augment your object with dual interface support, you must create interface wrapper classes for the dual interface that you define, both to expose your object's automation properties and methods through Vtable entry points, and for the benefit of clients that want to exploit the improved efficiency of a static interface binding.

Smalltalk COM Interface Binding Architecture

The Smalltalk binding of a COM interface has a two-level architecture. The lowest level of the interface binding is implemented by a class that handles the direct interaction between the Smalltalk process and external objects. A COMInterfacePointer binding class handles external function callout from the Smalltalk process to invoke an interface function supported by an external COM object. A COMInterfaceImplemention binding class handles external function callbacks to the Smalltalk process from external clients that are invoking a function in an interface supported by a Smalltalk COM object. The low-level interface bindings are encapsulated by the COMInterface wrapper classes, which provide Smalltalk clients with the protocol for invoking COM interface functions, which is natural to a Smalltalk programmer.

When you write a COM application, you work with COM interfaces using COMInterface instances, which allows you to write code that is natural for a Smalltalk programmer and uses the standard Smalltalk objects that you normally work with as arguments and message return values. The two-level interface binding architecture insulates you from the low-level mechanics of external function calls, and leaves you free to focus on your application logic.

Interface Class Responsibilities

COMInterface Framework

A COMInterface subclass is created to provide Smalltalk protocol corresponding to the functions in the interface. The COMInterface interface reference class provides a protocol that makes it easy for a Smalltalk client to invoke the interface functions, such as by encapsulating the details of obtaining OUT parameter values, so that the value can simply be returned to the caller. A COMInterface subclass typically also provides convenience operations that simplify function invocation for common cases that you expect to encounter. For example, the IClassFactory interface class createInstance:controllingUnknown: message invokes the IClassFactory::CreateInstance function and returns the interface that is obtained from the COM function through an OUT argument result value as the return value of the message. This service makes it easy for a Smalltalk client to create a new COM object using a protocol that is normal in the Smalltalk environment. The createInstance message is provided by the IClassFactory interface wrapper class as convenience protocol for the common case where you want to create a non-aggregated object and obtain its IUnknown interface.

COMInterfacePointer Framework

To invoke functions in an interface that is supported by a COM object implemented in another application, from either a local process or a process running on a remote system through DCOM, or from an object whose server is packaged as a DLL to run in the client's address space, a COMInterfacePointer subclass is needed to provide the interface binding for making external interface function calls. A interface pointer binding class defines the primitive COM external function declarations for each function in the interface and provides any argument marshalling support needed to convert between Smalltalk values and the host data type representations needed by the external function. For example, an interface pointer binding for a function that takes a string argument and obtains an interface back from the called function through an OUT argument maps the Smalltalk string argument into a string pointer in external memory and maps a “raw” interface pointer address result value from the OUT argument into a suitable Smalltalk interface object.

COMInterfaceImplementation Framework

To support an interface on a COM object implemented in Smalltalk, a COMInterfaceImplementation subclass is needed to provide the interface binding for allowing external clients to invoke the interface functions. An interface implementation binding class provides external function callback methods that are used by the VisualWorks COM binding mechanisms provided by the Object Engine to map external function invocations to the implementing interface binding in the image. The interface implementation binding's external function callback methods provide any argument marshalling support needed to convert between the host data type representations used by the external function and corresponding Smalltalk values.

In addition to the primary responsibility of supporting function invocation from external clients, an interface implementation binding usually provides a full set of function methods to allow internal Smalltalk clients to invoke the interface methods directly, without going through an external callout and callback into the same image, with the associated overhead of marshalling and unmarshalling Smalltalk values through the external data representations.

Directly connecting a Smalltalk client of the interface to the interface implementation binding installed in the interface reference is a performance optimization that can be exploited at the discretion of the application developer. It is also useful for testing object implementations, enabling test drivers to be written to exercise the functions supported by a COM object without the additional factor of external argument marshalling interposed in the test case.

Creating the Interface Type Definitions

To create the interface wrapper classes for a COM interface, you must first ensure that any data types referenced in the interface functions are defined. The external data types are defined using the usual facilities of VisualWorks DLL & C Connect. Most of the standard COM data types you are likely to use are already declared in the `Win32ExternalInterface` and `COMExternalInterface` classes, which define the external type definition name space for API and COM interface function declarations. If your interface requires any data types that are not already defined, you must import the declarations by creating DLLCC type definitions in COMExternalInterface. You can do this manually, by creating the <C:…> declaration in a method in a browser, or using the tools provided with DLLCC.

To define a Smalltalk binding for an interface, you must first create a type definition for the interface type in `COMExternalInterface`. By convention, the Smalltalk interface type definitions always map to a generic declaration of an interface pointer named `__IAnonymous`. While this is not strictly correct from the point of the C type declarations, where the actual interface type declaration is a structure containing a pointer to a structure defining the VTable function layout of the interface, it is sufficient for the purposes of the Smalltalk binding.

For example, the standard COM IClassFactory interface is declared in COMExternalInterface, as follows:

```
IClassFactory
    "Define the interface data type. Using __IAnonymous instead of
    __IClassFactory is a space optimization that avoids defining
    extraneous data types that are not needed by the COM Connect
        runtime."
    "<C: typedef struct __IClassFactory IClassFactory>"
    <C: typedef struct __IAnonymous IClassFactory>
```

You must also provide a declaration of the interface VTable, a structure containing the function pointers of the interface, by creating a VTable type declaration in the COMInterfaceVTableSignatures class. The name of the VTable definition type is by convention the name of the interface, prefixed by a double-underscore and with 'Vtbl' as the suffix. For example, the definition of the IClassFactory interface VTable is declared in COMInterfaceVTableSignatures, as follows:

```
__IClassFactoryVtbl
    "
    <C: struct __IClassFactory {
            struct __IClassFactoryVtbl * lpVtbl;
        }>
    <C: typedef struct __IClassFactory IClassFactory>
    "
    <C: struct __IClassFactoryVtbl {
            HRESULT ( __stdcall * QueryInterface)(IClassFactory * This,
                const IID * const riid, void * * ppvObject);
            ULONG ( __stdcall * AddRef)(IClassFactory * This);
            ULONG ( __stdcall * Release)(IClassFactory * This);
            HRESULT ( __stdcall * CreateInstance)(IClassFactory * This,
                IUnknown * pUnkOuter, const IID * const riid, void * *
                ppvObject);
            HRESULT ( __stdcall * LockServer)(IClassFactory * This,
                BOOL fLock);
        }>
```

Note that the interface function declarations in the VTable structure must conform to the C calling convention, in which the object that supports the interface is explicitly declared as the first parameter of the function and by convention is named 'This'. The VTable function declaration is the only place where this explicit recognition of the receiver is exposed. The <COM:…> function declaration in a COMInterfacePointer class follows the C++ notation, in which the receiver is an implicit argument of the function.The VTable type declarations in the COMInterfaceVTableSignatures class are part of the development support environment of COM Connect. This class and the associated VTable type declarations are not required by the COM runtime binding mechanisms and do not need to be included in a deployed image.

Creating COM Interface Wrapper Classes

After you have defined the interface type and the VTable layout, you can use the interface class generation tools provided with COM Connect to create a rough draft of the interface wrapper classes needed to provide the Smalltalk binding of a COM interface. It is important to understand that the interface class generation tools are creating a prototype of the interface wrapper class for you. Because the tools currently work only from the interface data type declarations, they do not have sufficient semantic information available to guarantee that a correct interface class can be generated automatically. Consequently, you must recognize that the tool is generating only a rough draft, which you need to review and complete. It is usually a pretty good cut and solves ninety percent of the problem, but you do need to do the final polishing by hand.

In most cases, the interface generation tool indeed generates the correct code for the interface wrapper class, or provides hints about how the needed code is likely to look. The tools handle the majority of the straightforward cases and correctly generate most of the standard “boilerplate” program text for you; so primarily, your concern is reviewing the result for semantic correctness and providing the contextual and semantic input to resolve difficult cases.

Watch for certain cases where the tool has difficulty. In general, be sure to check OUT parameters that are GUID values or structures. Because a GUID argument is passed as a pointer, the tool cannot always distinguish between the usual case of an IN parameter and the relatively infrequent cases where the GUID value is actually an OUT value that is set by the callee. The default assumption is to treat GUID values as IN arguments, so you must correct the generated code if the value is semantically an OUT parameter. The generated code for structure arguments has similar difficulties. Because structures are usually passed as pointer values, the interface generation tools cannot distinguish correctly between IN values (the usual case) and OUT values. Some cases also exist where string arguments are not handled correctly, so watch out for these.

Another typical case that the tool cannot handle in isolation is the case where the function takes an IID argument that identifies an interface and returns an interface pointer of the specified interface as an OUT argument value. The generated code notes this and provide hints about what to do, but you must provide the correct expression for obtaining the OUT value in the wrapper method. Usually, this entails filling in the suggested expression with the name of the input argument that specifies the IID. To generate the prototype COMInterfacePointer binding for an interface to support external callout function invocation of its functions, evaluate an expression of the form:

```
COMInterfacePointerClassGenerator
   generateInterfacePrototypeFor: #IFoo
```

where #IFoo is the name of the interface whose VTable structure is defined by the declaration of #__IFooVtbl.

To generate the prototype COMInterfaceImplementation binding for an interface that is supported by a Smalltalk COM object, to support external function callback invocation of its functions, evaluate an expression of the form:

```
COMInterfaceImplementationClassGenerator
   generateInterfacePrototypeFor: #IFoo
```

where #IFoo is the name of the interface whose VTable structure is defined by the declaration of #__IFooVtbl.

To generate the prototype COMInterface wrapper class for an interface to provide a usable interface for clients in your Smalltalk application, evaluate an expression of the form:

```
COMInterfaceClassGenerator generateInterfacePrototypeFor: #IFoo
```

where #IFoo is the name of the interface whose VTable structure is defined by the declaration of #__IFooVtbl.

While there is generally less review and correction of argument handling required for a COMInterface class than for the low-level binding classes, you should spend time customizing the interface protocol supported by the interface class to conform to the usual conventions of Smalltalk message naming. Usually, you should provide suitable keywords for the argument values in the message selector, following the usual Smalltalk programming style. Also, to determine whether to support any additional protocol that provides convenience services to Smalltalk clients for common patterns of function invocation, review the semantic specifications of the interface and the intended use of the services that it provides.

6

Using Automation Objects

The COMDispatchDriver class is used to create Automation objects, access existing Automation objects, and access methods and properties of a particular Automation object through it's dispatch interface (the IDispatch interface).

Creating an Automation Object

Automation servers always provide at least one type of object. Complex applications might support a number of objects. For example, a word processing application might provide an application object, a document object, and a toolbar object.

To create an Automation object, assign the object returned by the COMDispatchDriver createObject: class method to a variable, as follows:

```
aDispatchDriver := COMDispatchDriver createObject: 'ProgID'.
```

The createObject: method creates an Automation object, based on the specified ProgID. The ProgID has the form:

```
AppName.ObjectName[.VersionNumber]
```

The AppName is the name of the application, and the ObjectName identifies the type of object to create. Here is an example for creating an Excel spreadsheet using a version-independent ProgID:

```
aDispatchDriver := COMDispatchDriver createObject:
    'Excel.Application'.
```

The COMDispatchDriver createObject: class message can also create a new dispatch driver from a ProgID or a CLSID. For example:

From a ProgID:

```
aDispatchDriver := COMDispatchDriver createObject:
'Excel.Application.5'.
```

From a CLSID:

```
aCLSID := '{00020841-0000-0000-C000-000000000046}' asGUID.
aDispatchDriver := COMDispatchDriver createObject: aCLSID.
```

The COMDispatchDriver message release should always be called when you are done with an object:

```
aDispatchDriver release.
```

Some applications require that you call a Quit or Close method in order release all resources on the server and have the server itself quit. The release message just releases the interface, as far as the client is concerned. The server object might not be coded to quit and release itself from memory, when no clients are referencing it.

Working with version-independent ProgIDs and CLSIDs is described in more detail in Implementing Automation Objects.

The following table shows some of the Microsoft Office application object types and class names:

Application	Object Type	Class Name
Excel, version 5.0	Application Worksheet Chart	Excel.Application Excel.Worksheet Excel.Chart
Project, version 4.0	Application Project	MSProject.Application MSProject.Project
Word, version 4.0	WordBasic	Word.Basic
Word 97	Application	Word.Application

Unlike Microsoft Excel 7, Microsoft Word 7 is an application with a monolithic or unitary object model. Word does not have an object hierarchy. All methods are accessed through the top-level object called Word.Basic. Starting with Word97, Word’s object model will be broken up into a hierarchy.

Creating Visible and Invisible Objects

Generally, automation application objects can start themselves as visible or invisible. For example when you start a Microsoft Word 7 Word.Basic object, the application does not display itself. Most application objects have a Visible property that you can set to true or false to make the application show or hide itself on the display screen.

Obtaining an Active Application Object

The onActiveObject: method returns the currently active object of the specified ProgID. For example:

```
aDispatchDriver := COMDispatchDriver
    onActiveObject: 'Spreadsheet.Application'.
```

If there is no active object of the class Spreadsheet.Application, an error occurs.

Activating an Automation Object From a File

Many Automation applications let the user save objects in files. For example, a spreadsheet application that supports Worksheet objects lets the user save the worksheet in a file. The same application might also support a Chart object that the user can save in a file.

To activate an object from a file, use one of the following COMDispatchDriver methods:

- pathName: aFileName
- onActiveObject: aProgID
- pathName: aFileName progID: aProgID

The aFileName argument is a String containing the full pathname of the file to be activated. For example, an application named **Spreadsheet.exe** creates an object that was saved in a file named **Revenue.spd**. The following invokes **Spreadsheet.exe**, loads the **Revenue.spd** file, and assigns **Revenue.spd** to a variable:

```
aDispatchDriver := COMDispatchDriver pathName:
    'C:\My Documents\Revenue.spd'.
```

In addition to activating an entire file, some applications let you activate a specific item within a file. To activate part of a file, add an exclamation point (!) or a backslash (\) to the end of the file name, followed by a string that identifies the part of the file you want to activate. For information on how to create this string, refer to the object's documentation.

For example, if Spreadsheet.exe is a spreadsheet application that uses R1C1 syntax, the following code could be used to activate a range of cells within Revenue.spd:

```
aDispatchDriver := COMDispatchDriver
   pathName: 'C:\My Documents\Revenue.spd!R1C1:R10C20'.
```

These examples invoke an application and activate an object. In these examples, the application name (Spreadsheet.exe) is never specified. When one of these method is used to activate an object, the registry determines the application to invoke and the object to activate based on the file name or ProgID that is provided. If a ProgID is not provided, Automation activates the default object of the specified file.

Some ActiveX components, however, support more than one class of object. Suppose the spreadsheet file, Revenue.spd, supports three different classes of objects: an Application object, a Worksheet object, and a Toolbar object, all of which are part of the same file. To specify which object to activate, an argument must be supplied for the optional ProgID parameter. For example:

```
aDispatchDriver := COMDispatchDriver
   pathName: 'C:\My Documents\Revenue.spd'
   progId: 'Spreadsheet.Toolbar'.
```

This statement activates the Spreadsheet.Toolbar object in the file Revenue.spd.

Setting a Property

The COMDispatchDriver setProperty:value: message is used to set property values. For example:

```
aDispatchDriver setProperty: 'Name' value: 'Gary'.
```

The example above sets a property called 'Name' to a String 'Gary'.

```
aDispatchDriver setProperty: 'Regions' value: #( 'North' 'South'
 'East' 'West' ).
```

The example above sets a property named 'Regions' to the values 'North', 'South', 'East' and 'West' defined in the array.

The property name is a String and the argument is any Smalltalk object that can be mapped to an Automation data type.

Getting a Property

The COMDispatchDriver getProperty: message is used to get property values. For example:

```
aName := aDispatchDriver getProperty: 'Name'
```

This example gets a property named 'Name'.

The property name is a String and the answer is a Smalltalk object whose class is mapped from an Automation data type.

Calling a Method

The COMDispatchDriver message invokeMethod: is used to invoke an object's methods. For example:

```
aDispatchDriver invokeMethod: 'Calculate'.
```

The method name is a String. The method can answer a Smalltalk object whose class is mapped from an Automation data type.

Calling a Method With Arguments

The COMDispatchDriver invokeMethod:with: or invokeMethod:withArguments: message is used to invoke an object's methods with argument values. All positional arguments must be specified and can be any Smalltalk object that can be mapped to an Automation data type. For example:

```
aDispatchDriver
   invokeMethod: 'Insert' with: ' some text'.
```

The example above invokes a fictitious method 'Insert' with one String argument.

```
arguments := Array with: 'Gary' with: 'Los Angeles' with: 31.
aDispatchDriver
   invokeMethod: 'SubmitData' withArguments: arguments.
```

The example above invokes a fictitious method, SubmitData, with two String arguments and one Integer argument.

The method name is a String and the argument can be any Smalltalk object that can be mapped to an Automation data type. If only one argument is passed, use the invokeMethod:with: method. Use invokeMethod:withArguments: for any number of arguments.

Calling a Method With Named Arguments

The COMDispatchDriver invokeMethod:withNamedArguments: message is used to invoke an object's methods with named arguments. Using named arguments lets you submit a subset of all possible parameters. Named arguments are passed to a method using a standard Smalltalk Dictionary where the keys are the parameter names are the values the parameter values. For example:

```
"Build the Dictionary."
namedArgs := Dictionary new
    at: 'Name' put: 'Gary';
    at: 'OrderDate' put: Date today;
    at: 'WidgetId' put: 'W1234';
    at: 'Quantity' put: 3;
    yourself.

aDispatchDriver
    invokeMethod: 'SubmitOrder'
    withNamedArguments: namedArgs.
```

The method invoked might have many more arguments and can supply default values for the missing arguments.

In Microsoft Excel 7, the OpenText method normally takes the following arguments: filename, origin, startRow, dataType, textQualifier, consecutiveDelimiter, tab, semicolon, comma, space, other, otherChar, fieldInfo. Using named arguments you can just provide a filename and nothing else. Using named arguments saves from having to provide default values for all parameters.

Calling a Method With Arguments by Reference

The COMDispatchDriver messages invokeMethod:with:, invokeMethod:withArguments: or invokeMethod:withNamedArguments: can be used to invoke an object's method and pass one or more arguments by reference. When you want to pass a parameter by reference, an intermediary object must be created with the message asValueReference. The new value can be read after the method call by the message value to the reference. For example:

```
"Create the reference"
resultReference := COMVariantValueReference new.

"Build the argument array"
args := Array with: 'This argument is By Value'
    with: resultReference.

"Invoke the method"
aDispatchDriver invokeMethod: 'MyMethod'
    withArguments: args.

"Retrieve the reference argument"
myNumber := resultReference value.
```

The method name is a String and the argument can be any Smalltalk object that can be mapped to an Automation data type.

Subscribing for events

COMDispatchDriver instances can inform interested parties about occurring Automation events using the common TriggerEvent mechanism. This means it is possible to evaluate a block or send a message to a previously specified object on occurrence of such an event.

Registering for an event is done using one of the following methods:

- when:send:to:
- when:do:
- when:do:for:

Unsubscription is achieved using on of the following methods:

- unsubscribe:
- unsubscribe:from:

Simple Calling Syntax

Although it is possible to access all kind of functionality of an Automation object in the previously described way there is also a more simple way of calling a method or setting a property.

If you would be calling a Smalltalk method you would not want to do that by sending an invokeMethod: message passing the actual method name as a symbol or the arguments in an array you have to create before.

Therefore COMDispatchDriver provides the functionality which allows calling Automation methods as if they were Smalltalk methods.

Calling Automation Methods

Calling a parameterless method is accomplished by simply sending the name of the method to the COMDispatchDriver instance:

```
aCOMDispatchDriver MyAutomationMethod
```

Unlike Smalltalk methods, Automation methods may start with an uppercase letter.

Sending a one-argument message is achieved by adding a colon to the message name.

```
aCOMDispatchDriver MyAutomationMethod: anArgument
```

For any subsequent parameters additional keywords can to be added:

```
aComDispatchDriver MyAutomationMethod: firstArgument
   withName: aString
```

The keyword itself does not matter and is only used to pass an additional parameter. So you may chose a keyword describing the parameter or use a generic keyword like with:.

Accessing properties

The same scheme can also be applied when accessing properties. The difference is that, depending on whether the property value should be retrieved or set, the prefix get or set has to be added to the selector.

Retrieving the value of some Name property can be done in the following way:

```
aCOMDispatchDriver getName
```

Setting the value of the Name property to a new value can be done like this:

```
aCOMDispatchDriver setName: aString
```

Any further rules already mentioned for methods can also be applied here. For example, accessing a collection item might be accomplished by:

```
aCollectionDispatchDriver getItem: anIndex
```

Accessing an element of a two-dimensional array could be achieved in the following way:

```
aCollectionDispatchDriver getItem: x with: y
```

Considerations

When accepting a Smalltalk method with such sends, the compiler will very probably complain about sending non-existent method. This can be ignored. As long as the object exists in the Automation object it will work.

When sending messages with multiple parameters, all keywords after the first do not really matter–any valid Smalltalk keyword may be used. For the sake of being able to find message senders again, using a consistent naming scheme.

Data Types

Automation data types and Smalltalk classes are mapped as follows:

Map of Automation data types and Smalltalk classes

Automation Data Type	Smalltalk Class
VT_I4	Integer
VT_UI1	Integer
VT_I2	Integer
VT_R4	Float
VT_R8	Double
VT_BOOL	Boolean
VT_ERROR	Integer
VT_CY	FixedPoint with a scale of 4
VT_DATE	Timestamp
VT_BSTR	String
VT_UNKNOWN	IUnknown

Automation Data Type	Smalltalk Class
VT_DISPATCH	COMDispatchDriver or IDispatch
VT_ARRAY (Combined with another type)	Array (of Smalltalk objects)

When you get a property or a return value from a method invocation, the Smalltalk object you get back has been translated from an Automation data type. When you pass a Smalltalk object to an Automation object, use an object whose class matches its Automation counterpart. An Automation server can coerce objects (from the type you supply to the type the server needs) for you, but it is not obligated to do so.

Functions vs. Procedures

Some Automation methods are defined not to return any data at all by using the VT_VOID return type. From a programming languages point of view, this is the distinction between a function (which always has a return value) and a procedure (which never has a return value). Unless you are using the Variant specification policy, these procedures (the VT_VOID methods) must be invoked with the invokeProcedure: methods. Invoking an Automation procedure with an invokeMethod: call raises an error.

For example, most Word 7 methods are defined as procedures:

```
aDispatchDriver := COMDispatchDriver createObject: 'Word.Basic'.
[
aDispatchDriver
   invokeProcedure: 'FileNewDefault';
   invokeProcedure: 'InsertDateField';
   invokeProcedure: 'InsertTimeField';
   invokeProcedure: 'AppClose'.
] ensure: [ aDispatchDriver release ].
```

All comments that apply to the invokeMethod: method also apply to the invokeProcedure: methods. Note that Word 7 is the only application observed that uses procedures.

Object Destruction

The controller must provide a way for the user to say. “This object is no longer needed,” which internally calls the object’s release function followed by COMSessionManager freeUnusedLibraries, if wanted.

COMDispatchDriver are automatically released by finalization when no longer in use. While there is not general requirement as to when the release method is called explicitely, some applications require that you call a Quit or Close method in order to release all resources on the server and have the server itself quit. The release message only releases the interface as far as the client is concerned. The server object might not be coded to quit and release itself from memory when there are no clients referencing it.

The freeUnusedLibraries message unloads any DLLs that were loaded as a result of COM object creation calls, but which are no longer in use, and that, when loaded, were specified to be freed automatically. Client applications can call this function periodically to free up resources:

```
COMSessionManager freeUnusedLibraries.
```

It is most efficient to call freeUnusedLibraries either at the top of a message loop or in some idle-time task. DLLs that are to be freed automatically have been loaded with the bAutoFree parameter of the CoLoadLibrary function set to TRUE. The method freeUnusedLibraries internally calls DllCanUnloadNow for DLLs that implement and export that function.

Instances of COM interface pointers must always be released. For more information COM Connect Basics.

What to Do With an IDispatch

Some operations answer an IDispatch interface. What are you supposed to do with that interface? This section also applies to dual interfaces that are interfaces subclassed from COMDualInterface. Where this document refers to IDispatch, it is also referring to any dual interface subclasses.

The IDispatch interface pointer is represented by an instance of IDispatch. You typically acquire an IDispatch interface pointer in one of the following ways:

1 By specifying IID_IDispatch as the initial interface, when creating a new object using the IClassFactory createInstance:[…] class method. The IID can be that of a dual interface, as well.

2 As the result of a queryInterface: call.

3 As a return value from an Automation object's method or property invocation.

This happens all the time in Excel, for example. Actually, Excel 7.0 has 39 different dispatch interfaces. In Excel, those interfaces act as functionality groups, where a lot of dispinterfaces have similar entries. All Excel dispinterfaces have an 'Application' and a 'Parent' function. The 'Application' function answers a dispinterface pointer. Some dispinterfaces only give you access to an object's properties, while others can have any number of functions.

The default behavior is to automatically wrap IDispatch answers with a COMDispatchDriver (through the default lookup policy). When your starting point is an IDispatch, the simplest is to use the on: message:

```
aDispatchDriver := COMDispatchDriver on: anIDispatch.
```

This creates a COMDispatchDriver with the default specification policy. You can also create a COMDispatchDriver with a specific specification policy:

```
aDispatchDriver := COMDispatchDriver
    on: anIDispatch
    specificationPolicy: COMSpecificationPolicy newTypeCompilerPolicy.
```

The on:specificationTable: method creates a COMDispatchDriver with a 'complete' specification policy specified by the argument. No lookups are performed if a name is not found, and an error is raised. This method is explained in the section on specification policies.

Get the Methods and Properties of an Object

If the object you are using does not come with its own documentation, you can use the COMAutomationTypeAnalyzer class to help identify the method and properties an Automation object has, as well as what arguments each methods expects. These tools can work on a live object or a type library. Examples of using this tool can be found in the class comment of some of the sample classes: AutomationAllDataTypes, AutomationSmalltalkCommander, ExcelApplicationController, Word95BasicController.

The Automation Browser tool (**Tools > Com > Automation Browser**) also allows you to explore type libraries.

Microsoft also provides the OLE/COM Object Viewer tool that lets you browse COM objects and type libraries.

Using Type Libraries

Methods in the COMAutomationTypeAnalyzer class and methods that you write when publishing a COM object work with instances of a COMTypeLibrary. This section describes how to create and use a COMTypeLibrary.

A COMTypeLibrary instance can be used to work in the Windows Registration Database to:

- Register a type library.
- Update the registration for a type library.
- Unregister a type library.

A COMTypeLibrary is also used to get type information interfaces:

- Get an ITypeLib interface from a type library.
- Get an ITypeInfo interface from a type library.
- Get an ITypeInfo interface for a specific GUID from a type library.

Creating an Instance of a COMTypeLibrary

You create an instance of a COMTypeLibrary with one of the class messages or by creating and configuring a new instance of the class. The class methods are:

libraryID: *aLibID*

Answer a new instance of the receiver representing the library identified by the *aLibID* GUID.

new

Answer a new initialized instance of the receiver.

pathName: *aPathName*

Answer a new instance of the receiver representing the type library in the file named *aPathName*.

For example, the following expression creates a COMTypeLibrary and passes it to the COMAutomationTypeAnalyzer to generate a literal specification. Note that you must release the type library.

```
| anITypeLib |
anITypeLib := COMTypeLibrary pathName:
      'c:\vw30\com\examples\comauto\stcom\typelibrary\vwstcom.tlb'.
[ COMAutomationTypeAnalyzer
 generateTypeLibrarySpecificationsFromUser: anITypeLib ]
      ensure: [ anITypeLib release ].
```

Configuring a COMTypeLibrary for a Server Application

More complex examples are found in the AutomationAllDataTypes, AllDataTypesCOMObject, AutomationSmalltalkCommander, and SmalltalkCommanderCOMObject example COM server classes. The AutomationSmalltalkCommander class defines the newTypeLibraryEnglish class method, as follows:

```
newTypeLibraryEnglish
   "Answer a type library for the English language for the application."

   ^COMTypeLibrary new
      libraryID: self typeLibraryID;
      lcid: COMTypeLibrary lcidEnglish;
      directoryName: COMSessionManager absoluteCOMDirectoryName,
         'Examples\COMAuto\StCom\TypeLibrary';
      fileName: 'VwStCom.tlb';
      majorVersion: 1
      minorVersion: 0
```

This complete specification for a COMTypeLibrary permits the COMTypeLibrary updateRegistration method to perform the following, when the example server image is started as an Automation server:

- If the type library has never been registered on the system, the library is registered, which requires the full pathname of the library file. The directoryName: and fileName: methods are used, respectively, to set the absolute directory and the filename.
- If the type library is present on the system, the registry database is updated with dispatch interface information from the type library.

The other messages of interest are:

createRegistration

Makes sure that a library is properly registered. Load the type library from its file name and register the dispatch interfaces. This is normally done once when the application is installed.

removeRegistration

Removes type library information from the system registry. Use this message to allow applications to properly uninstall themselves. This service is not supported on the GA version of Windows 95.

The example servers call updateRegistration every time the server is started. While this can incur more overhead, it ensures that the type libraries are present and that the Registration Database contains the proper interface information. If the type libraries are not present, an error is raised. An alternative is to register the type libraries only on installation with createRegistration, or by adding entries yourself to the **.reg** for your server.

Automation Object Constants

If you are familiar with writing VBA code in Excel, you are familiar with the various constants prefixed with xl. It is far more easier to use these constants than hard-coded numbers. The COMAutomationTypeAnalyzer class lets you create a Pool Dictionary for each constant enumeration defined in a type library.

The ExcelApplicationController class used the following expression to generate a pool dictionary for the Excel constants:

```
"Utilities For Pool Dictionaries
===================="
"Create a text window describing the constants."
COMAutomationTypeAnalyzer describeConstants: self typeLibrary.

"Answer a dictionary of pool dictionaries."
COMAutomationTypeAnalyzer makePoolDictionaries: self
    typeLibrary.

"Show me the pool dictionaries you want to add before you do it."
COMAutomationTypeAnalyzer promptAndDefinePoolDictionaries:
    self typeLibrary.

"Or, add the pool dictionaries to Smalltalk automatically."
```

```
COMAutomationTypeAnalyzer definePoolDictionaries: self
    typeLibrary promptUser: true.

"Inspect the results."
Smalltalk at: #ExcelConstants.
```

These expressions are contained in the class comment. Here is a brief reference of the methods used:

definePoolDictionaries: *aCOMTypeLibrary* **promptUser:** *promptUser*

Add to the image a PoolDictionary for each enumeration constants defined in the *aCOMTypeLibrary* type library. Each dictionary name is the concatenation of the library name and an enumeration name. If *promptUser* is true, and an entry already exists in the Smalltalk dictionary for a given pool dictionary name, ask the user if they want to overwrite the entry. If promptUser is false, the pool dictionary is always added to Smalltalk.

describeConstants: *aCOMTypeLibrary*

Describe in a text window all the enumeration constants defined in the *aCOMTypeLibrary* type library.

makePoolDictionaries: *aCOMTypeLibrary*

Make a PoolDictionary for each enumeration constants defined in the *aCOMTypeLibrary* type library. Answer a dictionary where each key is the concatenation of the library name and an enumeration name and each value is a PoolDictionary.

promptAndDefinePoolDictionaries: *aCOMTypeLibrary*

Show the user a multiple selection list box for all of the pool dictionary name that can be made from the *aCOMTypeLibrary* library. The user chooses which Pool Dictionaries to generate. Add to the image a PoolDictionary for each enumeration constants chosen. Each pool dictionary name is the concatenation of the library name and an enumeration name. If an entry already exists in the Smalltalk dictionary for a given pool dictionary name, ask the user if they want to overwrite the entry.

Accessing Objects with IClassFactory

The Smalltalk class IClassFactory is used to a create single uninitialized object of the class associated with a specified CLSID. Call a createInstance method when you want to create only one object on the local system. To create a single object on a remote system, call a method with a serverName: argument.

To create multiple objects based on a single CLSID, refer to the getClassObject method. The IClassFactory services are used by the COMDispatchDriver class to create Automation objects. When you need to create objects for the purpose of Automation, you do not need to use the IClassFactory class directly. The class COMDispatchDriver provides class messages that wrap IClassFactory services for you. Here is an example of using IClassFactory.This example uses the IID_IDispatch constant from the COMConstants pool. If this pool is not in your evaluation context, add it or replace IID_IDispatch with (COMConstants at: #IID_IDispatch).

```
" Get an IDispatch from MS Excel. The argument CLSCTX_SERVER or
CLSCTX_LOCAL_SERVER must be used with MS Excel."

anIDispatch := IClassFactory
   createInstance: 'Excel.Application'
   iid: IID_IDispatch
   controllingUnknown: nil
   context: CLSCTX_SERVER.

"… do some work …"

anIDispatch release.
```

The first argument is the CLSID of the object to create. It is recommended that you use the COM Server's Application object or whatever object is at the root of the application's Automation object hierarchy.

The second argument is the interface ID, or IID, of the interface to request for the created object. In the case of Automation controllers, the object's dispatch interface is always requested, defined by the constant IID_IDispatch.

The third argument is the controlling unknown, if any. The controlling unknown is part of a reuse technique known as aggregation and is only of interest when implementing a new object.

The fourth argument is the context in which to create the object. This argument is explained later in this section.

Inside the Dispatch Driver

A COMDispatchDriver's behavior is governed by three attributes: an IDispatch interface, a specification table and a specification policy. For a dispatch driver to properly construct the arguments to the IDispatch::Invoke function, it must know about the method or property it is calling. These three attributes are defined as follows:

- The IDispatch interface makes the function calls to the Automation object.
- The Specification Table defines one Member Specification for each method and property that can be used on the Automation object. In general, each Member Specification contains a name, a dispatch ID, type information for a return value and the names and data types of any parameters.
- The Specification Policy defines the algorithm for dynamically defining Member Specifications for methods and properties not found in the Specification Table. The algorithms defined in the policy classes reflect certain speed and space tradeoffs as well as ease of use for the programmer. A Specification Policy contains a Specification Lookup Policy that defines where to look for and how to create Member Specifications.

There are two styles for creating a COMDispatchDriver:

- By specifying a Specification Table
- By specifying a Specification Policy

A Specification Table can be defined at design time and reused at runtime with an IDispatch to create a COMDispatchDriver. With a Specification Table, when a method or property is used that is not defined in the Specification Table, an error occurs. If a Specification Table is not used to create a COMDispatchDriver, unknown properties and methods need to have a Member Specification created for the IDispatch::Invoke call by a Specification Lookup Policy.

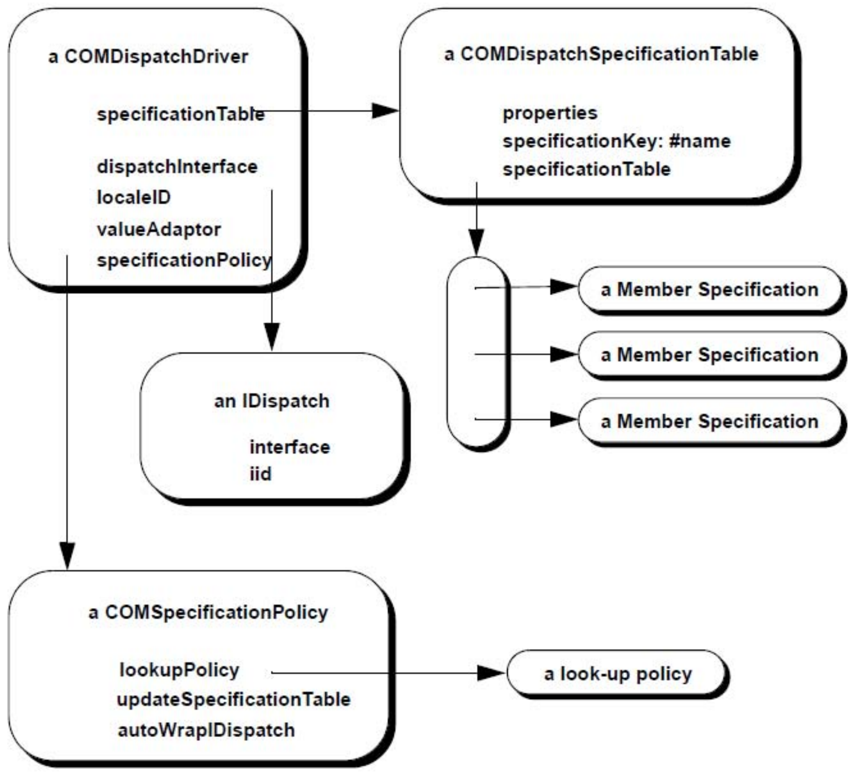
There are various ways Member Specifications can be created at run-time by a Specification Lookup Policy:

- From a type library

- From a type compiler

Another way is not to look anywhere and just use the generic Variant Automation data type familiar to Visual Basic programmers. The generic Variant method is used by the default Specification Policy, it always works since it does not rely on a type library or type compiler being around. The following figure shows the internal layout of a Dispatch Driver.

The internal layout of a Dispatch Driver



What Specification Policy to Use

The first question to answer is: can I rely on the controlled Automation object’s type information to be present at runtime? Microsoft strongly recommends that all applications ship with a type library, and again, MS Word 7 from Office 95 does not. A type library can be a stand-alone file or be included as a resource in a DLL or

EXE file. The documentation for the Automation object you want to use should tell you if a type library is shipped. The typed specification policies (type compiler and type library, described below) require that a type library be present. The untyped specification policies (variant and complete, described below) do not require that a type library be present.

Performance Tradeoffs

The tradeoff in dealing with specification policies and specification tables is as follows: Populating a specification table at run-time by querying the dispatch interface's type information can be slow if you use many of the dispatch interface's methods and properties. For example, Microsoft Word 7 has a monolithic object model (Word.Basic), with hundreds of methods exposed in one object, but this is an extreme case. The upside to the load on demand approach is that a controller does not use memory needed to store a literal array defining a specification table. On the other hand, using literal arrays to create specification tables at run-time is very fast. The downside is that the literal arrays take up space in memory; in the case of Word, a lot of it.

If you really need to use a large specification table, a remedy is to generate and use the entire literal array at development time and then manually prune the array of all unused methods and properties for run time use. Another approach would be to load on demand the methods and properties individually. For the latter, you are using a different specification policy (type library or type compiler).

Currently there are two policies that use the type library: TypeLibraryPolicy and LazyInializationPolicy. The first loads each individual function specification on demand while the latter loads all of them when it is first accessed. This approach is required to benefit from the Automation Inspector extensions.

Using the Default Specification Policy

The default specification policy is used automatically when a new COMDispatchDriver is created. The default is set LazyInitializationPolicy on installation. For example:

```
aDispatchDriver := COMDispatchDriver
   createObject: 'Excel.Application'.
```

or:

```
aDispatchDriver := COMDispatchDriver on: anIDispatch.
```

or:

```
aDispatchDriver := anIDispatch asDispatchDriver.
```

The default policy is set to the lazy initialization policy by the system but can be queried and reset by the programmer with the COMSpecificationPolicy classes defaultPolicy and defaultPolicy: messages. For example, you can query for the default specification policy with the expression:

```
COMSpecificationPolicy defaultPolicy
```

You can set the default Specification Policy with the expression:

```
COMSpecificationPolicy defaultPolicy: <aSymbol>
```

where aSymbol is one of the following:

- #newTypeCompilerPolicy
- #newTypeLibraryPolicy
- #newLazyInitializationPolicy
- #newVariantPolicy

Setting a Specification Policy

The COMDispatchDriver class on:specificationPolicy: message creates a new dispatch driver from an IDispatch and a specification policy. The dispatch driver uses the specification policy instance (Type Compiler, Type Library, or Variant) to dynamically create specifications for methods and properties the dispatch driver's specification table does not know about. For example:

```
aDispatchDriver := COMDispatchDriver
   on: anIDispatch
   specificationPolicy: COMSpecificationPolicy newTypeCompilerPolicy.
```

One of the following COMSpecificationPolicy class messages can be used when passing a specification policy to the on:specificationPolicy: method:

- #newTypeCompilerPolicy
- #newTypeLibraryPolicy
- #newLazyInitializationPolicy
- #newVariantPolicy

The Type Compiler Policy

This algorithm looks for properties and methods through a dispatch driver's ITypeComp interface, the type compiler interface (obtained from the dispatch driver's ITypeInfo interface). This policy creates complete and typed specifications and caches member specifications in the specification table by default.

This interface is not always supported but is efficient since the lookup is a direct one step process. If the application you are using is not associated with a type library through the Win32 Registry Database, this policy cannot be used.

The Type Library Policy

This algorithm looks for properties and methods in a dispatch driver's type library through its ITypeInfo interface. This policy creates complete and typed specifications and caches them by default. This policy is useful if the Type Compiler policy cannot be employed.

Memory is required to keep track of the name to index maps. If the application you are using is not associated with a type library through the Win32 Registry Database, this policy cannot be used.

The Variant Policy

This algorithm looks for properties and methods through a dispatch driver's IDispatch::GetIDsOfNames mechanism. This policy creates untyped specifications using VT_VARIANT as the generic data type. During method invocation it is possible that VT_VARIANT be rejected as the return type, in which case VT_VOID is used.

In Automation, a method is defined to have a return type, which can be generically processed by asking for a VT_VARIANT return data type. A method can have no return value at all, similar to the difference between a procedure and a function, in which case it must be invoked with the VT_VOID return type. Unlike the parameter passing logic, a return data type of VT_VARIANT cannot be used generically for this purpose. The COMUntypedSpecificationPolicy method invocation logic first attempts a VT_VARIANT return type invocation and upon failure for the above stated reason, attempts a VT_VOID return type invocation. If you know in advance (almost all methods in MS Word 7 from Office 95 are like this) that a method is defined not to return anything, use invokeProcedure: instead of invokeMethod:, which is faster, since no retry on failure takes place.

The Variant policy is a quick way to use a dispatch driver, since no additional COM API calls are necessary to properly construct all data structures associated with a particular invocation, the VT_VARIANT type is used generically to create a new specification every time. The default is to not update the specification table, since a call on the same name might have different parameter names creating different specifications. No additional memory is used since the untyped specifications are not stored, the specification table is always empty.

The Lazy Initialization Policy

This policy is a different kind of Type Library Policy. While the above mentioned only provides information about members (functions, properties) when it is explicitely asked for them, this policy loads all member information when the dispatch driver needs any. The advantage is that subsequent querying of members information is not required. Furthermore, supposing that you don't already have a complete specification table stored in the DispatchDriver, this is the only policy that currently can be used to benefit from the Automation Inspector Extensions, which provide a human-readable representation for all members of a living Automation object.

Dynamically Changing a Specification Policy

You can dynamically change the specification policy of a dispatch driver with the specificationPolicy: message. For example:

```
aDispatchDriver specificationPolicy:
   COMSpecificationPolicy newVariantPolicy.
```

A specification policy can be configured to record new specifications in the dispatch driver’s specification table. The default is to do so for a COMTypedSpecificationPolicy (Type Compiler and Type Library policies) but not for a COMUntypedSpecificationPolicy (Variant policy). The default for the Variant policy is to not update the specification table, since a call on the same name might have different parameters creating different specifications. This option can be toggled with the specification policy’s updateSpecificationTable: message. For example:

```
aDispatchDriver specificationPolicy updateSpecificationTable: true.
```

and:

```
aDispatchDriver specificationPolicy updateSpecificationTable: false.
```

Using a Specification Table

The COMDispatchDriver class on:specificationTable: message creates a new dispatch driver from an IDispatch with a Complete specification policy defined by the specification table argument. No specifications are dynamically added the dispatch driver's specification table using this specification policy. If a method or property is not found in the specification table, a COMError is signaled.

Building Specification Tables

The current choices to build a COMDispatchSpecificationTable include:

- Building a specification table using a type library
- Building a specification table using the type information interface

There are a number of class methods provided by COMDispatchSpecificationTable for constructing a specification table from type information describing an Automation object. In addition, the COMAutomationTypeAnalyzer class provides utilities to describe and generate literal specifications for dispatch interfaces and constant enumerations. See COM Connect Development Tools. The COMAutomationTypeAnalyzer class is not a run-time class.

Building a Specification Table from a Type Library

This technique uses a COM server application's Type Library. In the case where the Type Library is a standalone file, the COM server application is not loaded in memory. Note that while Excel 7 is shipped with a Type Library, Word 7 is not (you have to download it from a Microsoft site).

The following shows how support for a specification table was added to the ExcelApplicationController example. The expressions are found in the class comment of ExcelApplicationController.

```
Run the COMAutomationTypeAnalyzer on the wanted Excel class:
```

When this code is evaluated, a dialog box displays, holding a multiple-select list box containing a list of all available dispatch interface names in the type library. Only the ‘Application’ class is chosen. This example outputs a text window containing the literal notation of the specification tables for the dispatch interfaces selected. You can also name which interface you want directly:

```
COMAutomationTypeAnalyzer
   generateDispatchInterfaceNamed: 'Application'
   typeLibrary: ExcelApplicationController typeLibrary.
```

1 The text in the window was then copied and pasted into a new ExcelApplicationController class method called literalSpecification:

```
literalSpecification
   "Answer a collection of all method and property specification
   literals. This only needs to be here if you want to use the complete
   specification policy #newCompleteSpecification with this
   controller."
   " Dispatch Interface Application "
" Methods and Properties "
   ^#( #COMDispatchSpecificationTable
      #specificationKey: #name
      #name: 'Application'
      #iid: #( #GUID #[16r41 8 2 0 0 0 0 0 16rC0 0 0 0 0 0 0 16r46])
      #lcid: 9

   " Methods "
      #( 'method' 'Acos' 16r4063
         #typeCode: #VT_VARIANT
         )
      #( 'method' 'Acosh' 16r40E9
         #typeCode: #VT_VARIANT
         )

"Etc …"

)
```

2 This process is repeated for all of the Excel Controller classes. When you decide to use specification tables through the #newCompleteSpecificationPolicy for an application's controllers, you must provide a literalSpecification method for all controller classes related to this application. To test the various specification tables, see the class comments of the ExcelExampleMonsterDamage and ExcelExampleFileImport classes. For example:

```
"Test all specification policies."
ExcelApplicationController defaultSpecificationPolicy:
#newVariantPolicy.
ExcelExampleMonsterDamage runInvisible. "or runVisible"
```

```
ExcelApplicationController defaultSpecificationPolicy:
#newTypeLibraryPolicy.
ExcelExampleMonsterDamage runInvisible. "or runVisible"

ExcelApplicationController defaultSpecificationPolicy:
#newTypeCompilerPolicy.
ExcelExampleMonsterDamage runInvisible. "or runVisible"

ExcelApplicationController defaultSpecificationPolicy:
#newCompletePolicy.
ExcelExampleMonsterDamage runInvisible. "or runVisible"
```

If you do not want to use the controller framework to drive an application with a specification table, you need to construct the specification table object before passing it the COMDispatchDriver on:specificationTable: method, as follows:

1 Save the literal specification in a method.

2 Construct the specification table using the decodeAsLiteralArray message.

   For example, this is how the controllers do it:

```
specificationTable
    "Private. Answer the specification table for the receiver. Only
    used by controllers with the complete specification policy."

    ^self literalSpecification decodeAsLiteralArray
```

3 You now have all the elements to build a COMDispatchDriver:

```
aDispatchDriver := COMDispatchDriver
    on: anIDispatch
    specificationTable: self specificationTable
```

Building All Specifications From a Type Library

This is essentially the same as above with the difference that the COMAutomationTypeAnalyzer class is used to generate the specifications for all dispinterfaces in an application. As above, the following expression opens a dialog box on all dispinterfaces, if you click **OK**, since all items are checked, a specification table for all dispinterfaces is generated:

```
COMAutomationTypeAnalyzer
   generateTypeLibrarySpecificationsFromUser:
      ExcelApplicationController typeLibrary.
```

You can also do the same without the list box, using the following expression:

```
( MessageBox confirm: 'This might take a while, proceed?' )
    ifTrue: [ COMAutomationTypeAnalyzer
        generateTypeLibrarySpecifications:
            ExcelApplicationController typeLibrary ].
```

Building Specifications From Type Information

Since not all applications are shipped with a type library, a way is needed to create a specification table without a type library. This can be done by creating an instance of the object you need to control and querying its dispatch interface for type information. There is a one-to-one relationship between a dispatch interface and its type information (ITypeInfo). This is different from a type library, which defines the type information for many dispatch interfaces.

This example uses the COMAutomationTypeAnalyzer to load the dispatch specifications for Word from the type information. The method specifications are loaded from the literal encoding constructed at development time using the utility service:

```
COMAutomationTypeAnalyzer generateForID: 'Excel.Application'.
```

You then create the initialization methods from the resulting information as described above.

Summary

The COMDispatchDriver class is used to instantiate Automation objects, get to existing Automation objects and to access methods and properties of a particular Automation object through it’s dispatch interface (the IDispatch interface).

A dispatch driver is created with one of the following COMDispatchDriver class messages:

createObject: *aProgID*

This creates a COMDispatchDriver from a ProgID or a CLSID. The new instance has the default specification policy. This is equivalent to the Visual Basic CreateObject() function.

pathName: *aFileName*

Answer a new instance of the receiver on the automation object in the file named *aFileName*. This is equivalent to the Visual Basic GetObject(FileName) function.

pathName: *aFileName* **progID:** *aProgID*

Create an new Automation object of the aProgIDOrCLSID class and load *aFileName* into it. Answer an instance of the receiver on the automation object. The *aProgID* automation class must support IPersistFile. This is equivalent to the Visual Basic GetObject(FileName,ProgID) function.

onActiveObject: *aProgID*

Answer a new instance of the receiver on the active object of the *aProgID* automation object class. Raise a COMError if there is no active object. ProgID refers to a String representing ProgID or a GUID representing a CLSID. This is equivalent to the Visual Basic GetObject(,ProgID) function.

on: *anIDispatch*

This creates a COMDispatchDriver to wrap *anIDispatch* with the default specification policy.

on: *anIDispatch* **specificationPolicy:** *aCOMSpecificationPolicy*

This creates a COMDispatchDriver to wrap *anIDispatch* with the supplied specification policy.

on: *anIDispatch* **specificationTable:** *aCOMSpecificationTable*

This creates a COMDispatchDriver to wrap *anIDispatch* with a 'complete' policy specified by the argument. No lookups are performed if a name is not found, and an error is raised.

If your starting point is an IDispatch, the simplest way to create a dispatch driver is to use the on: message. The performance and usability trade-offs of using other specification policies are outlined in this document.

With a COMDispatchDriver you can:

- Invoke a method with the invokeMethod: [with: | withArguments: | withNamedArguments:] message.
- Get a property with the getProperty: message.
- Set a property with the setProperty:value: message.

7

Using ActiveX Controls

ActiveX controls are components that provide an easy way to add functionality to an application, including user interface components. They can be used directly in an application or embedded in other controls.

ActiveX Controls are actually complex COM Objects that utilize a many features provided by COM such as event support, Automation Calls, OLE Data Transfer, and Embedding. They are commonly stored in OCX Files, actually DLLs, which act as a file container for the controls, providing functionality to register and access the controls.

COM Connect includes support for using ActiveX Controls in your application's user interface. Configuration features remove the requirement of dealing with the COM layer as far as possible.

Using ActiveX Controls in a VisualWorks Application

Loading ActiveX Support

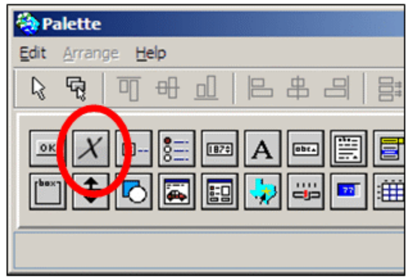
To use ActiveX controls in your VisualWorks application, load the ActiveX-All parcel.

Adding an ActiveX Control to your application

You add an ActiveX Control to your application by adding it to the user interface, using the VisualWorks UI Painter.

1 As usual, click the control button in the UI palette and place the control widget on your canvas as usual. The control is initially

shown only as a black bordered rectangle. This will change once the widget has been configured.



2 Provide an **Aspect** name, which will variable name for accessing the control.

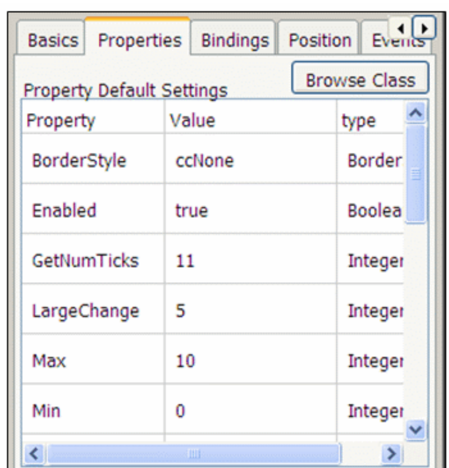
3 Select the Control that shall be embedded from the list of available controls.

4 Click the **Apply** button.

The look of your control in the UI will change to take on the look of the control you chose.

Configuring the Control

While you can use a control with its default settings, you generally will want to configure it for your own needs. For example, you may want to change the range of a slider control. Configuration parameters are provided on **Properties** tab in the GUI Painter Tool.



The tab provides a list of properties and allows assigning values to them which will be used as initial values when the application starts.

The middle row contains input fields in which values can be entered. If possible, a list of suitable values for specific properties is provided in a drop-down menu.

Right to the value row is a row describing the type of the property. In cases where it is not possible to provide a list of suitable values in the drop down menu, this row may help you to guess suitable values.

The values are immediately assigned to the control to provide an instant preview of the result. In case you are not satisfied with the result, you may change it back or use the **Cancel** button to undo all changes since the last **Apply**.

Finally **Apply** the change and **Install** the changes into the application UI specification method.

The **Browse Class** button opens an Automation Browser on the Control class.

Extended Configuration

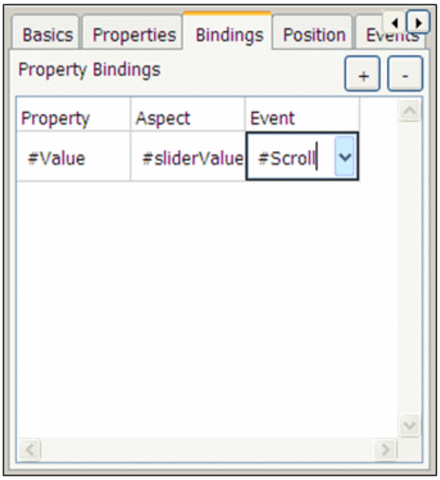
Some extended properties may not be set from the UI Painter. This includes properties which hold other COM objects. These properties need to be set in the source code. Because the control is instantiated using the specification method, this can not be done inside the controls aspect method.

Many controls are only completely operational when they are embedded. The `postOpenWith:` method of the application model is the usual place to fine-tune the control configuration. This can be even simplified by using data bindings.

Configuring Data Bindings

Data Bindings provide a way to bind control properties to application aspects. This is similar to using an `AspectAdaptor` for accessing specific aspects of objects hold by your application. The difference is that the values do not exist in Smalltalk but inside the Active-X contorl.

The Property Bindings Pane allows you to define such adaptors inside the UI Painter for using them in your application.



The advantage is that the UI provides the developer with all available information. The user does not have to know the exact name of the property he wants to access.

The pane contains three columns, titled Property, Aspect and Event. Two buttons, labelled "+" to add a new binding, and "-" to remove a binding.

When you press the add button, a new line is added to the table in which a new binding can be defined. In the first column, which is labelled Property, you should provide the name of the control property which shall be bound to an application aspect.

In the aspect column, provide the name of the application aspect. The name should reflect the respective property. The editor will assist you by suggesting an aspect name based on control and property name.

Many ActiveX Controls do not inform the client of runtime property changes using the PropertyChanged event. In spite of this they raise a custom event which indicates that a specific property has changed.

The **Event** column allows you to specify such an event. When the event appears, the binding will check for a value change and inform interested parties if needed.

Apply any changes and **Install** the UI specification. Also, create the corresponding aspect methods using the **Define** button.

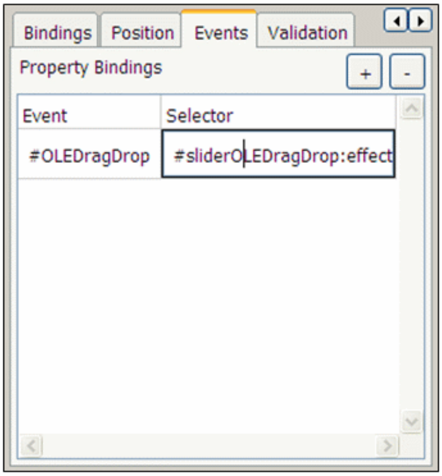
Configuring Events

In addition to binding property-specific events, the UI Painter allows you to configure non-specific events in a convenient way. The **Events** tab provides the functionality needed for mapping control events to application model methods.

To map events to methods:

1. Press the **+** button to add a new line to the table.
2. Select an event from the drop-down menu in the event column.

   The editor will suggest a selector based on control-, event-, and parameter names. You can change it, but the editor will take care that the provided selector will accept the correct number of arguments.



3. **Apply** the changes and **Install** the UI specification, as usual.
4. **Define** the event handler method.
5. Provide the needed code in the method, for example:

```
sliderOLEDragDrop: data effect: effect button: button
    shift: shift x: x y: y
    "This method was generated by UIDefiner for a Control Event.
    Any edits made here may be lost whenever methods are
     automatically defined."
    Dialog warn:
       (#AlthoughItDoesNotMatterSomethingWasDroppedOn
       TheSlider << #examples >> 'Although it does not matter
       something was dropped on the slider').
```

In this example the slider will react when something is dropped on it, such as a file from the Explorer.

Calling Control Methods

For calling control methods the ControlProxy provides three methods: invokeMethod, invokeMethod:with:, and invokeMethod:withArguments:. The first parameter has to be a string containing the name of the method to be called.

Licensing Support

The ActiveX Widget supports runtime licensing model of common container applications. This means, if the widget supports licensing, a valid license key is queried when the UI is created. This key is used at application runtime to verify that the User is allowed to utilize the control. There is no need to configure anything. It will be done for you automatically.

8

Implementing Automation Objects

Overview

COM Automation is a technology that enables application objects to be manipulated by other applications through a standard interface for dispatching client requests. COM Connect enables you to develop applications known as automation controllers, which manipulate objects published by other applications. Automation controller concepts and facilities are described in the chapters Using Automation Objects and Publishing Automation Objects.

VisualWorks COM Connect Automation support also contains facilities that assist you in implementing COM Automation objects within your own application and publishing them to enable other applications to manipulate your application's objects or access services. This section describes the facilities and frameworks that are available for implementing COM Automation objects.

As always when you design a new object, you need to decide what services your Automation object will provide. To publish these capabilities through COM Automation, you must first implement the desired behaviors and then provide a description of how the services appear to a COM Automation client as method and property members of a dispatch interface by creating a type library. Additionally, your COM objects can choose to support a dual interface.

After implementing your automation objects, you need to package your application as a COM object server and publish it so that your automation objects can be accessed from other applications. Creating a COM object server application requires a few steps to provide class factory support, so that clients can create new instances of your object, and a small amount of COM-specific application startup and termination logic.

To complete publishing a COM Automation object server, you need to create an executable application EXE containing your application code and register the server application with COM. The steps involved in creating an object server EXE and registering the application and its type library information with COM are also described in this documentation.

To summarize, publishing a COM Automation object involves the following steps:

1. Implementing the Automation objects
2. Creating a type library describing each Automation object
3. Mapping the COM interface functions to your class
4. Providing class factory support
5. Creating a **`.reg`** file to register the object server application
6. Implementing the object server application logic
7. Creating an object server application EXE

Installing the Automation Server Samples

The topics discussed in the document use several sample COM Automation objects to demonstrate the concepts and techniques.

The automation object server samples are installed with the COM Connect software and can be found in the 'COM-Automation-Server Samples' category in your VisualWorks system browser.

Basic Concepts of Automation Object Implementation

The basic notion of COM Automation is pretty simple. A COM Automation object is a COM object that supports the IDispatch interface, which clients use to invoke methods and access properties supported by your object through the IDispatch::Invoke function. A dispatch member invocation is similar to performing a message in Smalltalk: a dispatch invocation request contains the name of the operation the receiving object is to perform and the argument values for the operation. An automation object uses the information in the dispatch invocation request to perform the requested service and return a result value to the client.

Your task as an Automation object developer is to decide what operations and properties to support for your object, then make them available through an IDispatch interface binding to a COM object. The VisualWorks Smalltalk Automation support in Smalltalk provides facilities to assist in the mechanics of providing the IDispatch support, so that you can focus of the specifics of your own application.

When you implement a COM Automation object, you must define the methods and properties that your object supports. Each dispatch member has a unique integer name, referred to as its DISPID, as well as a string name. You must assign a unique DISPID to each method and property that your automation object supports. Choose positive integer values; negative values are reserved for predefined system uses and should be used only when implementing dispatch members that comply with the published standards. (You can obtain more information on standard COM Automation objects from Microsoft's OLE Programmer's Reference documentation.)

In addition to assigning unique DISPID's, specify a string name for each method and property. This enables use of your automation object from macro scripting clients such as Visual Basic. (Not to mention making it easier to describe your automation object and talk about its services!)

Finally, you must specify the type signature of each method and property. COM automation provides a well-defined set of data types, including such basic data types as integers, floating point numbers, and strings.

Even though this dispatch member information can all be specified inside a Smalltalk COM server, following standards and publishing this information through a COM type library is recommended. The MIDL Compiler takes an IDL text file to produce a type library and automatically assign DISPIDs for you.

Automation Object Implementation Techniques

VisualWorks provides an implementation framework and a configurable automation object server that you can use to easily publish objects through COM Automation. Your implementation options include:

- Publishing a Smalltalk object using the general-purpose automation server (IDispatch)
- Publishing a Smalltalk object using a custom Automation server (dual interface)

The COMAutomationServer class provides a generalized implementation of dispatch support, which can be used to publish the services of Smalltalk objects through COM Automation. A COMAutomationServer is configured by providing it with the object whose capabilities are being exposed through COM Automation and a specification table describing the methods and properties of the automation object. The dispatch specification table is indexed by the DISPID and contains a COMDispatchMemberSpecification entry for each method and property defining the string name of the member, its DISPID, and the automation data types of the value and any parameters needed to invoke the member. Each entry must also include the message selector, which is sent to the automation object being published to invoke the method or to get or set the property. A literal notation for dispatch member specifications is supported to enable a compact specification notation from which the execution dispatch specifications can be rapidly constructed.

A dual interface object supports the same functionality as an object published through an IDispatch with a COMAutomationServer with the added feature that all methods and properties can also be accessed through custom written interface functions. The COMDualInterfaceObject class provides a dual interface framework, which you can use to publish the services of Smalltalk objects through COM Automation. Implementing support for a dual interface is more complex but can yield to higher performance objects.

The configurable COMAutomationServer class is the easiest approach to use for most automation objects that you might wish to implement. However, you might encounter cases in more sophisticated applications where the standard automation object server facilities do not solve your particular problem. To develop customized automation objects, you can subclass the COMAutomationObject or COMDualInterfaceObject implementation framework to provide the desired IDispatch services or support additional interfaces.

To give an example of exposing a Smalltalk object through Automation, a subclass of Object called AutomationAllDataTypes is created and published through an IDispatch. The purpose of this example is to show how to use all of the Automation data types. Using all of the data types is easy when publishing an object through an IDispatch, but it is more complicated for a dual interface, since more code has to be written.

Under Publishing Automation Objects, the AutomationAllDataTypes example class is upgraded so that it can be published through a dual interface.

The code examples in this document for publishing a Smalltalk object through an IDispatch interface come from the example class AutomationAllDataTypes. The code examples in this document for publishing a Smalltalk class through a dual interface class come from the example classes AutomationAllDataTypes and AllDataTypesCOMObject.

Exposing a Smalltalk Class

As always when you design a new object, you need to decide what services your Automation object will provide. To publish these capabilities through COM Automation, you must first implement the desired behaviors and then provide a description of how the services appear to a COM Automation client as method and property members of a dispatch interface by creating a type library.

The tasks described in this section are common to exposing objects through an IDispatch or through a dual interface.

Publishing a class consists of the following tasks.

- Create or choose a Smalltalk class to publish. If your class is going to keep a copy of any interfaces, follow the rules for interface reference counting in this section.
- Create GUIDs to identify your classes, type libraries, and interfaces.
- Create an IDL file describing each class to publish from the image. Compile the IDL file into a type library.
- If you choose to implement a dual interface, implement the interface class, the interface implementation class and the COM class. Also implement the interface pointer class if you want to access the object from VisualWorks.
- Create a registration file (.reg) that describes to COM where to find your application.
- Make a run-time image.

You can publish any class in the hierarchy. To demonstrate the techniques of publishing a COM object, a subclass of Object called AutomationAllDataTypes is defined. This example shows how to support all of the standard Automation data types.

```
Object subclass: #AutomationAllDataTypes
   instanceVariableNames: 'propertyLONGValue propertyBYTEValue
      propertySHORTValue propertyFLOATValue propertyDOUBLEValue
      propertyVARIANT_BOOLValue propertySCODEValue
      propertyDATEValue propertyBSTRValue
      propertyIUnknownReference  propertyIDispatchReference
      propertyVARIANTValue propertyCURRENCYValue
      propertySAFEARRAY_I4Value
      propertySAFEARRAY_DISPATCHValue
      propertySAFEARRAY_UNKNOWNValue
      propertySAFEARRAY_BSTRValue '
   classVariableNames: 'TypeLibraries '
   poolDictionaries: 'COMConstants COMAutomationConstants '
   category: 'COM-Automation-Server Samples'
```

The example class defines an instance variable for each data type that can be used for automation.

Example Instance variable	COM data type	COM Type Code	Smalltalk Class
propertyLONGValue	longLONG	VT_I4	Integer
propertyBYTEValue	unsigned char BYTE	VT_UI1	Integer
propertySHORTValue	shortSHORT	VT_I2	Integer
propertyFLOATValue	floatFLOAT	VT_R4	Float
propertyDOUBLEValue	doubleDOUBLE	VT_R8	Double
propertyVARIANT_BOOLValue	booleanBOOLEAN	VT_BOOL	Boolean
propertySCODEValue	SCODE	VT_ERROR	Integer
propertyDATEValue	DATE	VT_DATE	Timestamp
propertyBSTRValue	BSTR	VT_BSTR	String
propertyIUnknownReference	IUnknown*	VT_UNKNOWN	IUnknown
propertyIDispatchReference	IDispatch*	VT_DISPATCH	IDispatch
propertyVARIANTValue	VARIANT	VT_VARIANT	An object
propertyCURRENCYValue	CURRENCY	VT_CY	FixedPoint w/ scale of 4

Example Instance variable	COM data type	COM Type Code	Smalltalk Class
propertySAFEARRAY_I4Value	SAFEARRAY (LONG)	VT_ARRAY | VT_I4	Array of Integers
propertySAFEARRAY_BSTRValue	SAFEARRAY (BSTR)	VT_ARRAY | VT_BSTR	Array of Strings
propertySAFEARRAY_DISPATCHValue	SAFEARRAY (IDispatch*)	VT_ARRAY | VT_DISPATCH	Array of IDispatches
propertySAFEARRAY_UNKNOWNValue	SAFEARRAY (IUnknown*)	VT_ARRAY | VT_UNKNOWN	Array of IUnknowns

Notes:

- An Integer can be a large negative integer or a large positive integer.
- A COM DATE holds its time part in two second intervals.
- A VARIANT can hold an object of any of the types in the table above.
- A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in the above table.

Implementing Properties

By convention, the selector of the get method, which returns the value of the property, is the same as the name of the property function defined in the type library (which was compiled from the IDL file) and is prefixed with get. The selector of the set method, which sets the value of the property, is the same as the name of the property function defined in the type library (which was compiled from the IDL file) and is prefixed with set.

For example, the accessor and mutator methods for the propertyBSTRValue instance variable are:

```
getBSTRValue
   "Answer the BSTRValue property. Answer a Smalltalk object that
   has been mapped from its Automation counterpart."
   ^propertyBSTRValue

setBSTRValue: aValue
   "Set the BSTRValue property. <aValue> is a Smalltalk object that has
   been mapped from its Automation counterpart."
   propertyBSTRValue := aValue
```

In the example class, the TypeLibraries class variable is used to keep track of the type libraries defined by the application. The type library in the example defines one object with one interface for the English language. Multiple type libraries are used when multiple languages are supported. The code associated with managing this variable is described later in the sections Application Startup and Type Library Management

Rules for Handling Interfaces

The data types that necessitate special handling are the interface types IDispatch and IUnknown. In COM, interfaces have special reference counting rules that must be closely followed. For more details, see COM Connect Basics in this document.

If an object returns an interface through an out parameter or as the function return value, then it must give away a separately counted reference of the interface. This is achieved by sending the message separateReference to the interface and giving away the answer.

The following is an example of giving away an IDispatch reference:

```
getIDispatchReference
   "Answer the IDispatchReference property. Answer a Smalltalk object
   that has been mapped from its Automation counterpart. The answer
   is a  separate reference of the interface that must be released by the
   caller."
   ^propertyIDispatchReference isNil
      ifTrue: [ propertyIDispatchReference ]
      ifFalse: [ propertyIDispatchReference separateReference ]
```

If an object retains a copy of an interface reference passed to it as a function in parameter for longer then the duration of the function call, then it must keep a separately counted reference of the interface. This is achieved by sending the message separateReference to the interface argument. The following is an example of keeping an IDispatch reference. Remember that the object keeping the interface is also responsible for releasing its reference to the interface when it is done using the interface.

```
setIDispatchReference: aValue
   "Set the IDispatchReference property. <aValue> is a Smalltalk object
   that has been mapped from its Automation counterpart."
   | anInterface |
   "Release the current interface if present."
   propertyIDispatchReference isCOMInterface
      ifTrue: [
         anInterface := propertyIDispatchReference.
```

```
            propertyIDispatchReference := nil.
            anInterface release ].
    aValue isCOMInterface
        ifTrue: [
            "To keep a separate reference of the interface passed in."
            propertyIDispatchReference := aValue separateReference ]
        ifFalse: [
            propertyIDispatchReference := aValue ]
```

Saving a reference to an IUnknown interface follows the same pattern as is shown here for IDispatch.

The same care must be applied when dealing with SAFEARRAYs of IDispatches and IUnknowns as illustrated by the following methods. Again, only the IDispatch version is shown.

```
getSAFEARRAY_DISPATCHValue
    "Answer the SAFEARRAY_DISPATCHValue property. Answer a
    Smalltalk object that has been mapped from its Automation
    counterpart. The answer is made of separate references of the
    interfaces that must each be released by the caller."
    ^propertySAFEARRAY_DISPATCHValue isNil
        ifTrue: [ propertySAFEARRAY_DISPATCHValue ]
        ifFalse: [ propertySAFEARRAY_DISPATCHValue collect:
            [ : anInterface | anInterface separateReference ] ]

setSAFEARRAY_DISPATCHValue: aValue
    "Set the SAFEARRAY_DISPATCHValue property. <aValue> is a
    Smalltalk object that has been mapped from it's Automation
    counterpart."
    | interfaces |
    "Release the current interface if present."
    interfaces := propertySAFEARRAY_DISPATCHValue.
    propertySAFEARRAY_DISPATCHValue := Array new.
    interfaces notNil
        ifTrue: [ COMSafeArray releaseInterfacesIn: interfaces ].
    COMSafeArray acquireInterfacesIn: aValue.
    propertySAFEARRAY_DISPATCHValue := aValue.
```

Implementing a Method

By convention, the selector for a method called by COM is the same as the function name defined in the type library (which was compiled from the IDL file). For example, a Reset method is defined.

```
Reset
    "Reset the values in the receiver to the initialized state."
    self initialize
```

Terminating an Application

When you create a deployment image for delivery as a COM object server application, you are responsible for providing the application startup logic that checks the startup conditions and registers the class factories for the COM object classes that your application supports with COM. Your application is also responsible for deciding how to shut itself down and when this occurs.

Continue with this section only if you arranged for a COM object server application that never shuts down. For more information see Publishing Automation Objects

This setting is useful when you want your server to always be executing in anticipation of client requests. Once started, the object server application continues running indefinitely, with your class factories registered with COM and everything prepared to handle client object creation requests immediately.

The server session termination service revokes the class factories registered by your application with COM and shuts down the Smalltalk process. Determining when to actually terminate the application is your responsibility.

If you are implementing an Automation application object following the guidelines defined by Automation for the standard Application object, it is required that your application object support a Quit method, which exits the application and closes all open documents. To implement the Quit command for an application object, the Quit method in your application should conform to the following pattern:

```
Quit "Quit the application."
    " ... close all open documents ... "
    COMSessionManager terminateServerSessionDeferred.
```

Note the use of the terminateServerSessionDeferred message in this case. The deferred termination service is necessary here because the Quit method is invoked by a client, and you need to ensure that your function returns to the caller before the Smalltalk process is shut down.

Creating Class Identifiers

When you are ready to publish your COM object so that it can be used by other applications, you must have a unique CLSID that identifies the COM object class. CLSIDs are universally unique identifiers (UUIDs, also called globally unique identifiers, or GUIDs) that identify class objects to COM. The CLSID is included in an application, and must be registered with the operating system when an application is installed. The CLSID is how other applications can create or access instances of your object and must be registered with the operating system when an application is installed.

You can create a new GUID value to assign as the CLSID of your automation object by evaluating the expression:

```
GUID new
```

A new GUID value created is guaranteed to be unique and can be used to name your automation object class.

When you are ready to assign a CLSID to your object class, implement in your automation object class an accessing method that returns the CLSID.

The CLSID of the AutomationAllDataTypes class is defined in its class method clsid as follows:

```
clsid
    "Answer the CLSID under which a the receiver is published as an
    Automation object."
    " '{DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}' asGUID storeString "

    ^GUID fromBytes: #[ 16rE3 16rE8 16r5D 16rDB 16r1F 16rAD 16rD0
     16r11 16rAC 16rBE 16r5E 16r86 16rB1 0 0 0 ]
```

While the string representation of the CLSID is more readable, as you can see by the variation in the comment, it is more efficient to construct the CLSID from the byte encoding. You can obtain this expression by pasting the result of evaluating the following expression into the clsid class method of your automation object class.

```
GUID new storeString
```

Note that while the clsid message can be sent to any COMObject class, not all COM objects have a CLSID. A COM object without a CLSID is not available to clients independently, in that clients cannot arbitrarily request such an object to be created. Such objects typically occur as part of implementing cooperating objects as part of a running application. This is actually quite common; for example, the COMDataTransferObject, which you can use to implement COM data transfer support for an object you are implementing, does not have a CLSID, since it is only instantiated within the implementation of your Smalltalk COM application.

The AutomationAllDataTypes example generated the following GUIDs.

- a CLSID {DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}
- a Type library ID for **vwAllDT.tlb** {DB5DE8E1-AD1F-11d0-ACBE-5E86B1000000}
- an Interface ID for IAllDataTypesDisp {DB5DE8E2-AD1F-11d0-ACBE-5E86B1000000}

Creating the Type Library

Create a type library for each set of exposed objects. Because VTBL references are bound at compile time, exposed objects that support VTBL binding must be described in a type library.

Type libraries provide these important benefits:

- Type checking can be performed at compile time. This might help developers of ActiveX clients to write fast, correct code to access objects.
- Visual Basic applications can create objects with specific interface types, rather than the generic Object type, to take advantage of early binding.
- VisualWorks and VisualSmalltalk applications can create objects with specific interface types, rather than the generic IDispatch type, to take advantage of early binding.
- ActiveX clients that do not support VTBLs can read and cache DISPIDs at compile time, improving run-time performance.
- Type browsers can scan the library, allowing others to see the characteristics of objects.
- The RegisterTypeLib function can be used to register exposed objects in the registration database. This operation is performed by the COMTypeLibrary createRegistration method.
- The UnRegisterTypeLib function can be used to completely uninstall an application from the system registry. This operation is performed by the COMTypeLibrary removeRegistration method.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.

Type Libraries and the Object Description Language

When you expose ActiveX objects, type libraries allow interoperability with the programs of other vendors. For vendors to use these objects, they must have access to the characteristics of the objects (properties and methods). To make this information available developers must:

- Publish object and type definitions (for example, as printed documentation).
- Code objects into so they can be accessed using `IDispatch::GetTypeInfo` or implementations of the ITypeInfo and ITypeLib interfaces.
- Use the Microsoft Interface Definition Language (MIDL) compiler or the MkTypLib utility to create a type library that contains the object descriptions, then make the type library available.

The MIDL compiler and the MkTypLib utility both compile scripts that are written in the Object Description Language (ODL). Microsoft has expanded the Interface Definition Language (IDL) to contain the complete ODL syntax. Use the MIDL compiler in preference to MkTypLib, since support for MkTypLib is being phased out.

For more information about the MIDL compiler, refer to the *MIDL Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK).

Generating a Type Library With MIDL

Microsoft's Interface Definition Language (IDL) now includes the complete Object Definition Language (ODL) syntax. This allows you to use the 32-bit MIDL compiler instead of `MKTYPLIB.EXE` to generate a type library and optional header files for a COM application.

Note: When the documentation refers to an ODL file, this means a file that MKTYPLIB can parse. When it refers to an IDL file, this means a file that MIDL parses. This is strictly a naming convention. The MIDL compiler parses an input file regardless of its filename extension.

The top-level element of the ODL syntax is the library statement (library block). Every other ODL statement, with the exception of the attributes that are applied to the library statement, must be defined within the library block. When the MIDL compiler sees a library block it generates a type library in much the same way as MKTYPLIB does. With a few exceptions, the statements within the library block should follow the same syntax as in the ODL language and MKTYPLIB.

You can apply ODL attributes to elements that are defined either inside or outside the library block. These attributes have no effect outside the library block unless the element they are applied to is referenced from within the library block. Statements inside the library block can reference an outside element either by using it as a base type, inheriting from it, or by referencing it on a line as shown:

```
<IDL definitions including definitions for interface IFoo and struct bar>
[<some attributes>]
library a
{
interface IFoo;
struct bar;
...
};
```

If an element defined outside the library block is referenced within the library block, then its definition is put into the generated type library.

The MIDL compiler treats the statements outside of a library block as a typical IDL file and parses those statements as it has always done. Normally, this means generating C-language stubs for an RPC application.

The Win32 SDK contains full documentation for MIDL and the IDL language at:http://www.microsoft.com

Automation Data Types

In the .idl file used to define a type library, the oleautomation attribute indicates that an interface is compatible with COM Automation. The parameters and return types specified for its members must be compatible with COM Automation, as listed in the following table. Keep in mind that the set of data types you can use must come from this table or be equivalent to them for the MIDL compiler.

Automation Data Types

Type	Description	Maps to Class
boolean	Data item that can have the value TRUE or FALSE. In MIDL, the size corresponds to unsigned char.	Smalltalk Boolean
BSTR	Length-prefixed string, as described in the COM Automation topic BSTR.	Smalltalk String

Type	Description	Maps to Class
Type	Description	Maps to Class
DATE	64-bit floating-point fractional number of days since December 30, 1899.	Smalltalk Timestamp
double	64-bit IEEE floating-point number.	Smalltalk Double
CY	(Formerly CURRENCY) A currency number stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range of money, or for any fixed-point calculation where accuracy is particularly important.	Smalltalk FixedPoint (with a scale of 4)
enum	Signed integer, whose size is system-dependent. In remote operations, enum objects are treated as 16-bit unsigned entities. Applying the v1_enum attribute to an enum type definition allows enum objects to be transmitted as 32-bit entities.	Smalltalk Integer
float	32-bit IEEE floating-point number.	Smalltalk Float
IDispatch *	Pointer to IDispatch interface (VT_DISPATCH).	Smalltalk IDispatch
int	Integer whose size is system dependent. On 32-bit platforms, MIDL treats int as a 32-bit signed integer.	Smalltalk Integer
Type	Description	Maps to Class
IUnknown *	Pointer to interface that is not derived from IDispatch (VT_UNKNOWN). (Any COM interface can be represented by its IUnknown interface.)	Smalltalk IUnknown
long	32-bit signed integer.	Smalltalk Integer
SCODE	Built-in error type that corresponds to HRESULT.	Smalltalk Integer
short	16-bit signed integer.	Smalltalk Integer
unsigned char	8-bit unsigned data item.	Smalltalk Integer

A parameter is compatible with COM Automation if its type is a COM Automation-compatible type, a pointer to a COM Automation-compatible type, or a SAFEARRAY of a COM Automation-compatible type. A SAFEARRAY maps to a Smalltalk Array.

A return type is compatible with COM Automation if its type is an HRESULT, SCODE or void. However, MIDL requires that interface methods return either HRESULT or SCODE. Returning void generates a compiler error.

A member is compatible with COM Automation if its return type and all its parameters are COM-Automation compatible.

An interface is compatible with COM Automation if it is derived from IDispatch or IUnknown, it has the **oleautomation** attribute, and all of its VTBL entries are COM-Automation compatible. For 32-bit platforms, the calling convention for all methods in the interface must be STDCALL. For 16-bit systems, all methods must have the CDECL calling convention.

Every dispinterface is implicitly COM Automation-compatible. Therefore, do not use the **oleautomation** attribute on dispinterfaces.

The **oleautomation** attribute is not available when you compile using the MIDL compiler /osf switch.

Type Libraries and the MIDL compiler are discussed in further detail later in this document.

Creating the Programmable Interface

An object's programmable interface comprises the properties, methods, and events that it defines. Organizing the objects, properties, and methods that an application exposes is like creating an object-oriented framework for an application. The chapter Standard Automation Objects and Naming Guidelines discusses some of the concepts behind naming and organizing the programmable elements that an application can expose.

Creating Methods

A method is an action that an object can perform, such as a request to perform a debit or credit transaction. Methods can take any number of arguments (including optional arguments), and they can be passed either by value or by reference. A method might or might not return a value.

Creating Properties

A property is a member function that sets or returns information about the state of the object, such as a social security number or birthday. Most properties have a pair of accessor functions; a function to get the property value and a function to set the property value. Properties defined to be read-only or write-only, however, have only one accessor function.

Property Accessor Functions

The accessor functions for a single property have the same dispatch identifier (DISPID). The purpose of each function is indicated by attributes that are set for the function. These attributes are set in the **`.idl`** file description of the function, and are passed in the wFlags parameter to IDispatch::Invoke in order to set the context for the call. The attributes and flags are shown in the following table.

Purpose of function	ODL attribute	wFlags
Returns a value.	propget	DISPATCH_PROPERTYGET
Sets a value.	propput	DISPATCH_PROPERTYPUT
Sets a reference.	propputref	DISPATCH_PROPERTYPUTREF

> **Note:** VisualWorks encapsulates the IDispatch interface gore through the COMDispatchDriver class.

The **`propget`** attribute designates the accessor function that gets the value of a property. When a COM Connect client needs to get the value of a property, it calls the COMDispatchDriver method getProperty:. The argument is the property name.

The **`propput`** attribute designates the accessor function that sets the value of a property. When a COM Connect client needs to set the value of a property, it calls the COMDispatchDriver method setProperty:value:. The first argument is the property name and the second argument is the property value.

The **`propputref`** attribute indicates that the property should be set by reference, rather than by value. When a COM Connect client needs to set the reference of a property, it calls the COMDispatchDriver method setProperty:value:. The first argument is the property name and the second argument is the property reference.

Implementing the Value Property

The Value property defines the default behavior of an object when no property or method is specified. It is typically used for the property that users associate most closely with the object. For example, a cell in a spreadsheet might have many properties (Font, Width, Height, and so on), but its Value property defines the value of the cell. To refer to this property in Visual Basic, a user does not need to specify the property name Cell(1,1).Value, but can simply use Cell(1,1). The Value property is identified by the standard DISPID named DISPID_VALUE. In an **`.idl`** file, the Value property for an object has the attribute **`id(0)`**.

Handling Events

In addition to supporting properties and methods, ActiveX objects can be a source of events. In Automation, an event is a method that is called by an ActiveX object, rather than implemented by the object. For example, an object might include an event method named Button that retrieves clicks of the mouse button. Instead of being implemented by the object, the Button method returns an object that is a source of events.

In Automation, you use the **`source`** attribute to identify a member that is a source of events. The **`source`** attribute is allowed on a member of a co-class, property, or method. See the discussion of COM event support under Implementing COM Objects.

Creating the Type Library IDL File

To create a type library, you must write an IDL file to describe the methods and properties of the objects you want to publish. An IDL file describes one type library. A type library can hold definitions for multiple objects, each object with multiple interfaces. The example type library contains one object with one interface.

By convention, all interface names have an 'I' prefix, and dispatch interface names have a 'Disp' suffix. The dispatch interface supported by the AutomationAllDataTypes object is called IAllDataTypesDisp. For this example, the following GUIDs are defined:

- a CLSID {DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}
- a Type library ID for vwAllDT.tlb {DB5DE8E1-AD1F-11d0-ACBE-5E86B1000000}
- an Interface ID for IAllDataTypesDisp {DB5DE8E2-AD1F-11d0-ACBE-5E86B1000000}

For example, view the IDL file for the example in **`COM\Examples\COMAuto\AllDataT\TypeLibrary\VWAllDT.idl`**.

When the example is upgraded to a dual interface under, as described in Publishing Automation Objects, a slightly different IDL file will be used. A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in this example.

Building the Type Library

The IDL file for the example is compiled by running the **`COM\Examples\COMAuto\AllDataT\TypeLibrary\MakeLibrary.bat`** batch file, which contains the following command:

```
MIDL @vwAllDT.rsp
```

The response file **`vwAIIDT.rsp`** contains:

```
/win32
/tlb vwAllDt.tlb
/iid iid_vwAllDT.cpp
/h midl_vwAllDT.h
/o vwAllDT.log
/proxy vwAllDT_p.c
```

This example response file for the MIDL compiler contains directives to perform the following operations:

```
/win32
```

1 The target environment is Microsoft Windows 32-bit (NT).

```
/iid iid_vwAllDT.cpp
```

2 Specify interface UUID file name. The name of this output file is set with the ‘iid_’ prefix to hint at its content and indicate that it was produced from the MIDL compiler.

```
/h midl_vwAllDT.h
```

3 Specify header file name. The name of this output file is set with the ‘midl_’ prefix to indicate that it was produced from the MIDL compiler.

```
/o vwAllDT.log
```

4 Redirects output from the screen to a file.

```
vwAllDT.idl
```

The input file.

The examples use the /iid and /h and directives in order to avoid a potential file name collision with Visual C++. With Visual C++, files can be automatically generated from a type library to create files containing C++ utilities.

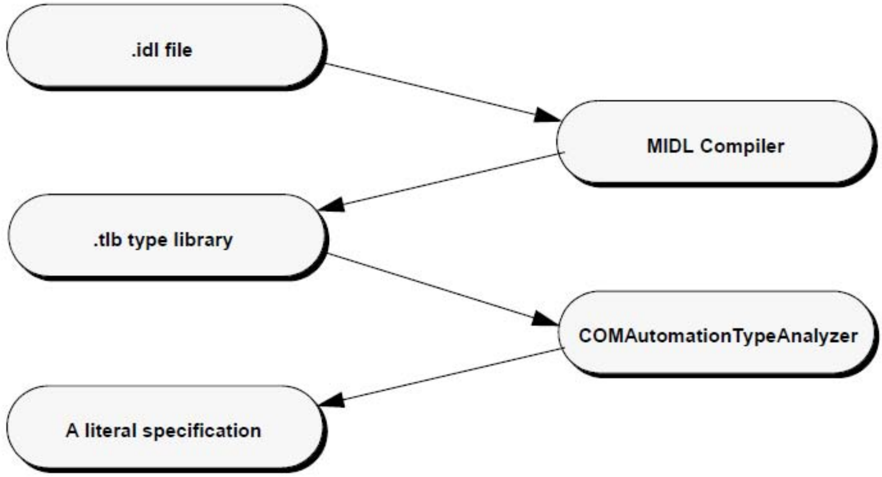
Mapping COM Interface Functions to a Class

A Specification Table for a COM Automation server object is used to map incoming requests to methods implemented by the published class. A Specification Table defines one Member Specification for each method and property that can be used on the Automation object. In general, each Member Specification contains a name, a dispatch ID, type information for a return value and the names and data types of any parameters. The type information is used to translate objects between their COM and Smalltalk representation. For a COM Automation server, this table is indexed by DISPID.

A specification table is created in the examples by decoding its literal representation which is defined by a method, typically implemented in the class being published. A literal specification for a specification table is created from the information in the type library by the COMAutomationTypeAnalyzer development utility. The next expression can be found in the class comment for AutomationAllDataTypes.

The following figure shows how a literal specification is derived.

Deriving a literal specification




```
"Create the specification for all dispatch interfaces in the type library."
COMAutomationTypeAnalyzer
   generateTypeLibrarySpecifications: AutomationAllDataTypes
typeLibraryEnglish
   forRole: #server.
```

In the preceding example, the text generated by the COMAutomationTypeAnalyzer class is cut and pasted into a method called literalSpecification. Note the following about the literal specification:

- The specification table is indexed by member ID:

  ```
  #specificationKey: #memberID
  ```

- No selectors are explicitly defined in the literal specification, so the default selectors are used.
- For a method, the default selector is the method name itself and arguments keywords are _:.
- For properties, the default set selector is set<Proper-tyName>: and the default get selector is get<PropertyName>.

```
literalSpecification
   " Type Library VWALLDT Dispatch Interfaces "
   " Generated by COMAutomationTypeAnalyzer on June 19, 1997
   17:30:52.000 "
   " From VisualWorks(R), Release 2.5.2 of September 26, 1995 "
   " This is the specification table literal for the dispatch Interface:
```

```
    Name:IAllDataTypesDisp
    Locale ID:1033
    IID:     {DB5DE8E2-AD1F-11D0-ACBE-5E86B1000000}
    Methods:3
    Properties:18
 This interface is indexed for use by a server. "
" Specification Table Header "
^#( #COMDispatchSpecificationTable
  #specificationKey: #memberID
  #name: 'IAllDataTypesDisp'
  #iid: #( #GUID #[ 16rE2 16rE8 16r5D 16rDB 16r1F 16rAD 16rD0
    16r11 16rAC 16rBE 16r5E 16r86 16rB1 0 0 0 ] )
  #lcid: 1033
  " Methods (3) "
  " The selector sent to the published object is by default the
    method name itself. "
  " The keyword for method arguments is by default #with: "
  " The selector can be set manually in each methodspecification
    by using the pattern: "
  "   #selector: #mySelector "
  #( 'method' 'Quit' 16r60020024
    )
  #( 'method' 'Reset' 16r60020025
    )
  #( 'method' 'ManyArguments' 16r60020026
    #typeCode: #VT_VARIANT
    #parameterTypes: #( #VT_DISPATCH #VT_BSTR #VT_I4 )
    #parameterNames: #( 'AnIDispatch' 'PropertyName' 'Number' )
    )
  "  Properties (18)  "
  " The selector sent to the published object to set a property is
    by default set<PropertyName>: "
  " The selector sent to the published object to get a property is
    by default get<PropertyName>: "
  " The selector can be set manually in each property
    specification by using the pattern: "
  "   #setSelector: #mySelector: "
  "   #getSelector: #mySelector "
  #( 'property' 'LONGValue' 16r60020000
    #typeCode: #VT_I4
    #parameterTypes: #( #VT_I4 )
    )
  #( 'property' 'BYTEValue' 16r60020002
    #typeCode: #VT_UI1
    #parameterTypes: #( #VT_UI1 )
    )
  #( 'property' 'SHORTValue' 16r60020004
```

```
        #typeCode: #VT_I2
        #parameterTypes: #( #VT_I2 )
        )       #( 'property' 'FLOATValue' 16r60020006
        #typeCode: #VT_R4
        #parameterTypes: #( #VT_R4 )
        )
    #( 'property' 'DOUBLEValue' 16r60020008
        #typeCode: #VT_R8
        #parameterTypes: #( #VT_R8 )
        )
    #( 'property' 'VARIANT_BOOLValue' 16r6002000A
        #typeCode: #VT_BOOL
        #parameterTypes: #( #VT_BOOL )
        )
    #( 'property' 'SCODEValue' 16r6002000C
        #typeCode: #VT_ERROR
        #parameterTypes: #( #VT_ERROR )
        )
    #( 'property' 'DATEValue' 16r6002000E
        #typeCode: #VT_DATE
        #parameterTypes: #( #VT_DATE )
        )
    #( 'property' 'BSTRValue' 16r60020010
        #typeCode: #VT_BSTR
        #parameterTypes: #( #VT_BSTR )
        )
    #( 'property' 'IUnknownReference' 16r60020012
        #typeCode: #VT_UNKNOWN
        #parameterTypes: #( #VT_UNKNOWN )
        )
    #( 'property' 'IDispatchReference' 16r60020014
        #typeCode: #VT_DISPATCH
        #parameterTypes: #( #VT_DISPATCH )
        )
    #( 'property' 'VARIANTValue' 16r60020016
        #typeCode: #VT_VARIANT
        #parameterTypes: #( #VT_VARIANT )
        )
    #( 'property' 'CURRENCYValue' 16r60020018
        #typeCode: #VT_CY
        #parameterTypes: #( #VT_CY )
        )
    #( 'property' 'SAFEARRAY_I4Value' 16r6002001A
        #typeCode: #( #VT_ARRAY #VT_I4 )
        #parameterTypes: #( #( #VT_ARRAY #VT_I4 ) )
        )
    #( 'property' 'SAFEARRAY_DISPATCHValue' 16r6002001C
```

```
        #typeCode: #( #VT_ARRAY #VT_DISPATCH )
        #parameterTypes: #( #( #VT_ARRAY #VT_DISPATCH ) )
        )
    #( 'property' 'SAFEARRAY_UNKNOWNValue' 16r6002001E
        #typeCode: #( #VT_ARRAY #VT_UNKNOWN )
        #parameterTypes: #( #( #VT_ARRAY #VT_UNKNOWN ) )
        )
    #( 'property' 'SAFEARRAY_BSTRValue' 16r60020020
        #typeCode: #( #VT_ARRAY #VT_BSTR )
        #parameterTypes: #( #( #VT_ARRAY #VT_BSTR ) )
        )
    #( 'property' 'SAFEARRAY_VARIANTValue' 16r60020022
        #typeCode: #( #VT_ARRAY #VT_VARIANT )
        #parameterTypes: #( #( #VT_ARRAY #VT_VARIANT ) )
        )
    ).
" End of specification "
```

Note: A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in this example.

Mapping DISPID Requests to Your Class

When an invocation request comes into VisualWorks, a DISPID is supplied to identify which method or property is to be invoked on behalf of the client. This input DISPID is used as an the index into the specification table for the object being invoked. Once the corresponding member specification is found in the specification table, it identifies which method to call on the published object. The member specification also indicates how to decode the arguments and return value.

Mapping a DISPID to a Method

The selector used to call a method on a published object is by default the same as the method name, case included. For example, the example Reset method is mapped to the published object (an instance of AutomationAllDataTypes) method Reset. The method to call can be overridden in the method specification with the #selector: specifier. For example, you could change the definition of Reset to invoke the initialize method. The current definition is as follows:

```
#( 'method' 'Reset' 16r60020025 )
```

A new definition might be as follows:

```
#( 'method' 'Reset' 16r60020025
#selector: #initialize )
```

Mapping a DISPID to a Method With Arguments

The selector used to call a method on the published object is by default the same as the method name, case included. For each argument selector the default keyword _: is used. For example, the ManyArguments method is mapped to the published object (an instance of AutomationAllDataTypes) method ManyArguments:_:_:. This method takes three input argument and answers a value. At the interface level, answers are actually placed in the last argument which is marked in the IDL file by **[out, retval]**. The method to call can be overridden in the method specification with the #selector: specifier. For example, you could change the definition of ManyArguments to invoke another method selector. The current definition is as follows:

```
#( 'method' 'ManyArguments' 16r60020026
   #typeCode: #VT_VARIANT
   #parameterTypes: #( #VT_DISPATCH #VT_BSTR #VT_I4 )
   #parameterNames: #( 'AnIDispatch' 'PropertyName' 'Number' )
)
```

A new definition might be as follows:

```
#( 'method' 'ManyArguments' 16r60020026
   #typeCode: #VT_VARIANT
   #parameterTypes: #( #VT_DISPATCH #VT_BSTR #VT_I4 )
   #parameterNames: #( 'AnIDispatch' 'PropertyName' 'Number' )
   #selector: #SomeNewName:argNumber2:argNumber3:
)
```

Mapping a DISPID to a Property

For properties, two messages can be used, a get message (or accessor) and a set message (or mutator). By default, the get selector is defined as #get concatenated to the property name. The set selector is defined as #set concatenated to the property name. In the example, the LONGValue property is defined as follows:

```
#( 'property' 'LONGValue' 16r60020000
   #typeCode: #VT_I4
   #parameterTypes: #( #VT_I4 ))
```

This definition indicates the return value for the property is a COM long (VT_I4 which maps to a Smalltalk Integer). The parameter types shows that when the property is set the argument is also a COM long.

When a property get invocation comes into Smalltalk, the method getLONGValue is invoked on the published object and answers an Integer. When a property set invocation comes into Smalltalk, the method setLONGValue: is invoked on the published object with an Integer argument.

To change either methods invoked on the published object, the literal member specification can be redefined with custom selectors. For example, you can change the definition to invoke a pair of methods myLongValue, myLongValue:

```
#( 'property' 'LONGValue' 16r60020000
   #typeCode: #VT_I4
   #parameterTypes: #( #VT_I4 )
   #getSelector: #myLongValue
   #setSelector: #myLongValue:)
```

Exposing Classes Through IDispatch

Publishing a class through an IDispatch consists of the following tasks:

- Create or choose a Smalltalk class to publish. If your class is going to keep a copy of any interfaces, follow the rules for interface reference counting in this section.
- Implement the class initialization code. This code is meant to be executed when the class is installed in an image. It should not cause COM APIs to be called.
- Create an IDL file describing each class to publish from the image. Compile the IDL file in a type library.
- Implement the application startup logic. This code hooks up your class to COM when the run-time image starts up.
- Create a **.reg** File that describes to COM where to find your application.
- Make a run-time image.

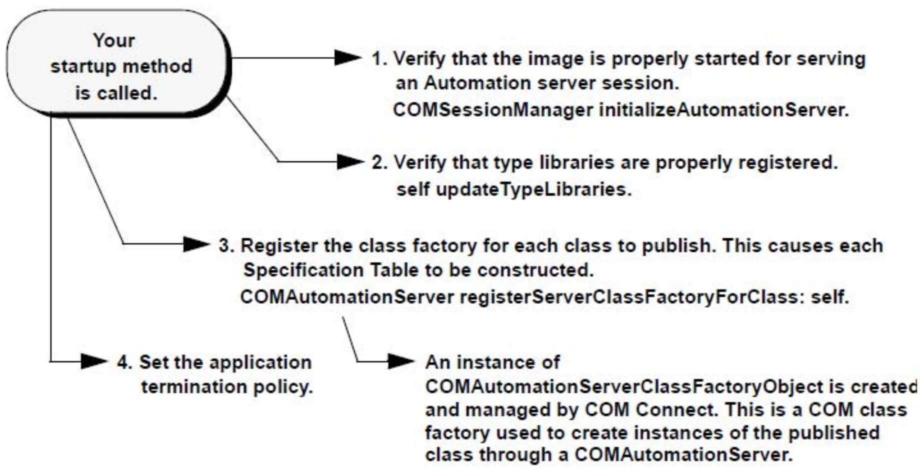
The Big Picture

This section presents an overview of the process of image startup, object creation and object function invocation.

Image Startup

When a object server application image startup method is called for a published object, the steps shown in the following figure should occur.

Image startup sequence

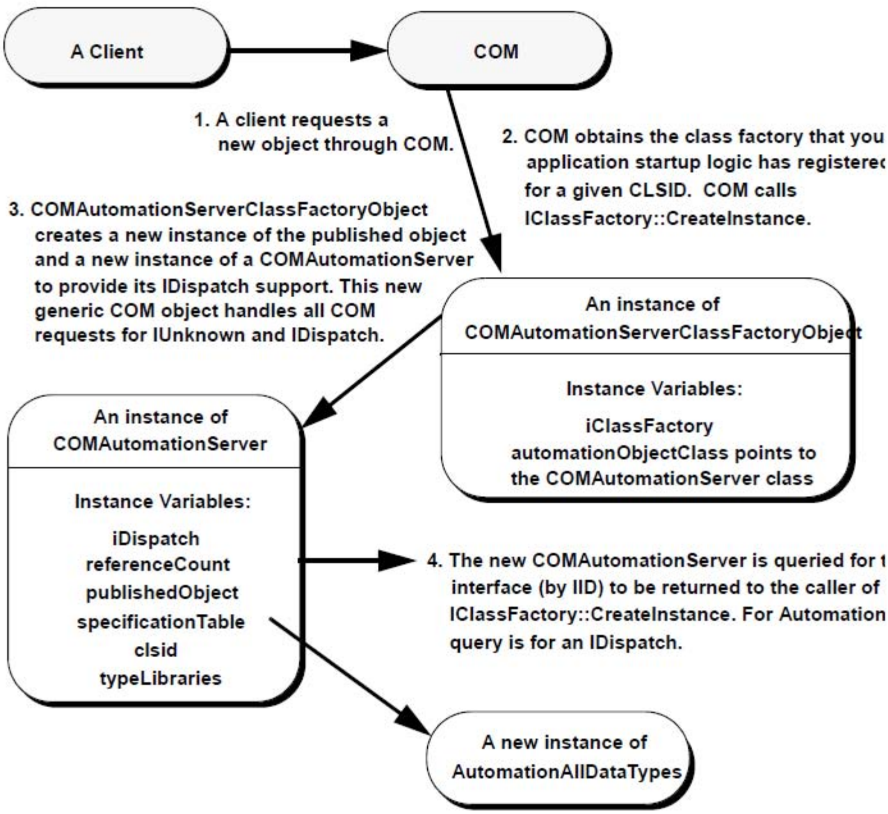


In step 3 an instance of COMAutomationServerClassFactoryObject is created by your application startup logic and managed by COM Connect. This is a COM class factory that is used to create instances of the published class through a COMAutomationServer. You configure the class factory with the class to be published (your automation class) and the dispatch specifications describing the capabilities of the published object.

Object Creation

When an object server application image is started as a result of a client request to COM to create an instance of your COM object, the steps shown in following figure take place.

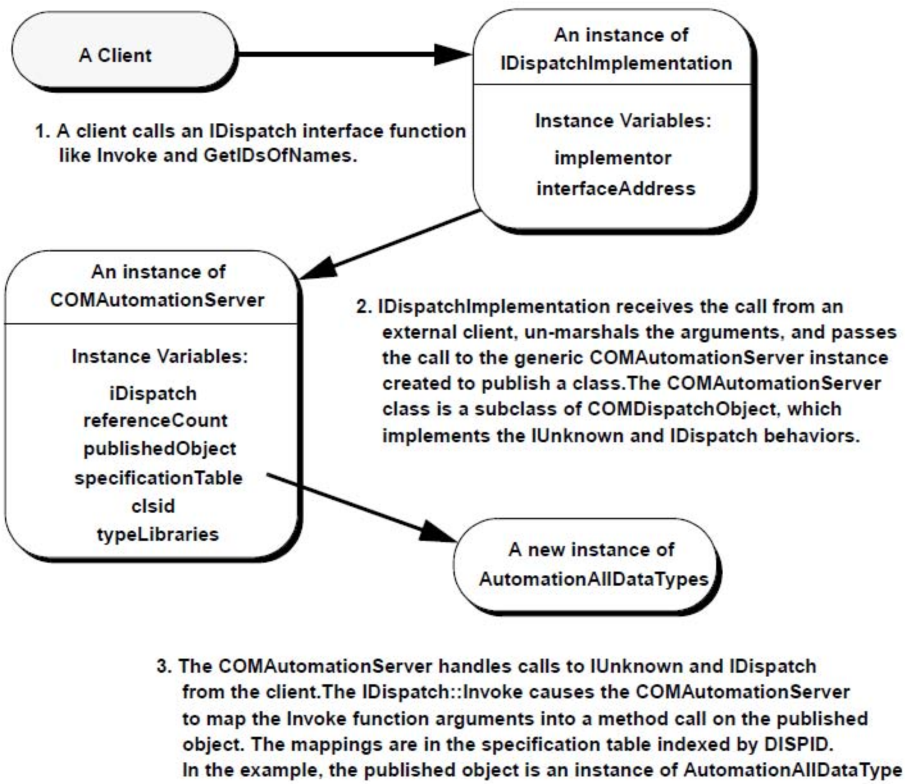
Object creation sequence



Object Function Invocation

When the published object is running and a dispatch invocation call comes in from the client, the steps in following figure take place.

Object function invocation sequence



Class Initialization

For a class used in Automation to be properly managed in the image, the class must:

- Ask to be notified of a run-time image startup in order to connect the Smalltalk class to the COM world.
- Ask to be notified of image shutdown and image save in order to release type libraries.

These tasks are accomplished by implementing initialization code meant to be executed when the class is installed in the image. The example includes a ClassInitializer class method:

```
ClassInitializer
    "This method is run at COM Connect installation time."
    " self ClassInitializer "
    self registerSessionEventHandlers.
registerSessionEventHandlers
    "Install the event handlers for the receiver."
    self removeSessionEventHandlers.  " always safe "
    COMSessionManager
        when: #shutdownImage
        send: #releaseTypeLibraries to: self.
    COMSessionManager
        when: #confirmSaveImage
        send: #releaseTypeLibraries to: self.
    ImageManager
        when: #deploymentStartup
        send: #startUpApplication to: self.
```

The registerSessionEventHandlers method performs the following:

- Call removeSessionEventHandlers as a safety.
- Ask to be notified of image shutdown and image save in order to release type libraries.
- Ask to be notified of a run-time image startup in order to connect the class to COM.

Note: A ClassInitializer method must not cause any COM APIs to be called.

The methods implementing these functions are described in the sections Type Library Management and Application Startup.

Application Startup

In the example, the message startUpApplication is sent to the class when a run-time image is started. On application startup, your 'startUpApplication' method must perform the following tasks:

- Verify that the image is properly started for serving an Automation server session.
- Verify that type libraries are properly registered.

- Register the class factory for each class to publish. This causes each Specification Table to be constructed.
- Set the termination policy.

A startUpApplication method is typically defined on the class side as follows:

```
startUpApplication
    "Start up the Automation object server."
    "Initialize COM and verify that the application is being run as an object
    server."
    COMSessionManager initializeAutomationServer.
    "Make sure the type libraries are ok."
    self updateTypeLibraries.
    "Register the class factory for the object server application."
    COMAutomationServer registerServerClassFactoryForClass: self.
    "Arrange for server application termination "
    COMSessionManager exitIfNotInUse: false.
```

The steps in the startUpApplication methods are explained below.

Verify Startup for an Automation Server

The COMSessionManager class method initializeAutomationServer initializes COM and verifies that the application is being run as an Automation object server. The session is terminated if the necessary conditions for an object server application are not satisfied.

```
COMSessionManager initializeAutomationServer.
```

If the server EXE is not started with the **/Automation** command line argument, a dialog displays instructions to set the command argument, then terminates.

Verify Type Library Registration

On application startup, the class should make sure that type libraries are properly registered:

```
self updateTypeLibraries.
```

The example method updateTypeLibraries updates the registry for each type library defined by the application. For each type library, this method makes sure that the type library is properly registered by performing the following tasks:

1. Try to load the library as registered. If OK, then

2 Update the registry with dispatch interface information from the type library.

3 If (1) fails, then try to load the type library from its file name and register the dispatch interfaces.

Register the Class Factory

Register the object's class factory with COM so other applications can use it to create new objects:

```
COMAutomationServer registerServerClassFactoryForClass: self.
```

A look at the COMAutomationServer class method registerServerSessionClassFactoryForClass shows that your class must implement the class methods clsid, specificationTable and typeLibraries in order to use this class factory registration service.

```
registerClassFactoryForClass: aClass
   " Register a class factory to create instances of <aClass>.  Answer the
   class factory.  Answer nil if class factory registration failed. "
   ^self registerClassFactoryForClass: aClass
      clsid: aClass clsid
      specificationTable: aClass specificationTable
      typeLibraries: aClass typeLibraries
```

You can call the COMAutomationServer registerClassFactoryForClass:clsid:specificationTable:typeLibraries: method and provide the necessary argument values explicitly, if the pattern of the protocol expected of the class argument does not suit your application.The example, using the registerClassFactoryForClass:, defines the methods clsid, specificationTable and typeLibraries as follows:

```
clsid
   "Answer the CLSID under which a the receiver is published as an
   Automation object."
   " '{DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}' asGUID storeString "

   ^GUID fromBytes: #[ 16rE3 16rE8 16r5D 16rDB 16r1F 16rAD 16rD0
   16r11 16rAC 16rBE 16r5E 16r86 16rB1 0 0 0 ]

specificationTable
   "Private. Answer the specification table for the receiver."

   ^self literalSpecification decodeAsLiteralArray

typeLibraries
   "Answer the type library dictionary. The dictionary keys are LCIDs and
   the values are instances of COMTypeLibrary."| aTypeLibrary |
```

```
TypeLibraries isNil ifTrue: [
    TypeLibraries := Dictionary new.
    aTypeLibrary := self newTypeLibraryEnglish.
    TypeLibraries at: aTypeLibrary lcid put: aTypeLibrary.
    "Other type libraries can be added for additional languages...
    aTypeLibrary := self typeLibraryLanguageX.
    TypeLibraries at: aTypeLibrary lcid put: aTypeLibrary."
 ].
^TypeLibraries
```

Type Library Management

Each class should be described in a type library to properly play the Automation game. When an Automation class is used, at least one type library should be registered. There can be an additional type library for each additional language supported.

You must create and compile an IDL file for the objects you want to publish. Type libraries are discussed in further detail elsewhere in this document. An IDL file defines a type library with a type library ID, a locale ID, a library name, a major and minor version.

The most important method in the example is newTypeLibraryEnglish, which defines the COMTypeLibrary instance used with the COMAutomationServer class. The newTypeLibraryEnglish method answer a new instance of a COMTypeLibrary, which is used to create the type library entries in the registration database at installation time and to update the type library entries at runtime.

```
newTypeLibraryEnglish
    "Answer a type library for the English language for the
    application."
    ^COMTypeLibrary new
        libraryID: self typeLibraryID;
        lcid: COMTypeLibrary lcidEnglish;
        directoryName: COMSessionManager absoluteCOMDirectoryName,
        'Examples\COMAuto\AllDataTypes\TypeLibrary';
        fileName: 'VwAllDT.tlb';
        majorVersion: 1;
        minorVersion: 0
```

The example implements type library management as listed below. (Note that when this example is upgraded to use a dual interface, most of this code is unnecessary, since the dual interface framework handles type library management.)

```
registerTypeLibraries
    "Register the type libraries."
    self typeLibraries do: [ : aTypeLibrary |
        aTypeLibrary createRegistration ]

releaseTypeLibraries
    "Release the type libraries."
    | toBeReleased |
    "This test does not use self to avoid loading."
    TypeLibraries isNil ifTrue: [ ^self ].
    toBeReleased := TypeLibraries.
    TypeLibraries := nil.
    toBeReleased do: [ : aTypeLibrary | aTypeLibrary release ].

typeLibraries
    "Answer the type library dictionary. The dictionary keys are
    LCIDs and the values are instances of COMTypeLibrary."
    | aTypeLibrary |
    TypeLibraries isNil ifTrue: [
        TypeLibraries := Dictionary new.
        aTypeLibrary := self newTypeLibraryEnglish.
        TypeLibraries at: aTypeLibrary lcid put: aTypeLibrary.
        "Other type libraries can be added for additional languages...
        aTypeLibrary := self typeLibraryLanguageX.
        TypeLibraries at: aTypeLibrary lcid put: aTypeLibrary."
     ].
    ^TypeLibraries

typeLibraryEnglish
    "Answer a type library for the English language for the application."
    ^self typeLibraries at: COMTypeLibrary lcidEnglish

typeLibraryID
    "Answer the IID of the receiver's type library."
    " '{DB5DE8E1-AD1F-11d0-ACBE-5E86B1000000}' asGUID storeString "
     ^GUID fromBytes: #[ 16rE1 16rE8 16r5D 16rDB 16r1F 16rAD
    16rD016r11 16rAC 16rBE 16r5E 16r86 16rB1 0 0 0 ]

unregisterTypeLibraries
    "Unregister the type libraries."
    self typeLibraries do: [ : aTypeLibrary |
        aTypeLibrary removeRegistration ]

updateTypeLibraries
    "Update the registry for the type libraries."
    self typeLibraries do: [ : aTypeLibrary |
        aTypeLibrary updateRegistration ]
```

Run-Time Installation

"Publishing Automation Objects describes the steps involved in saving an image for run-time deployment. For a Smalltalk class published with an IDispatch interface, your class should implement an installRuntime method to provide some housekeeping hooks for your type libraries.

The example class AutomationAllDataTypes class method installRuntime is defined as follows:

```
installRuntime
   " Prepare the receiver for deployment in a run-time image
   configuration.  You can extend this method and place installation
   code in it. "
   " self installRuntime "
   self
      releaseTypeLibraries;
      registerTypeLibraries.
```

Supporting Multiple National Languages

In order to support multiple languages, you must create a type library for each language. Each type library must then be wrapped by an instance of COMTypeLibrary in your type library management code for the published class. If you are implementing a dual interface object, this would be done through your COMDualInterfaceObject subclass getTypeLibraries class method.

Implementing IDispatch for Multilingual Applications

When creating applications that support multiple languages, you need to create separate type libraries for each supported language, as well as for versions of the IDispatch member functions that include dependencies for each language.

Creating Separate Type Libraries

For each supported language, write and register a separate type library. The type libraries use the same DISPIDs and globally unique identifiers, but localize names and Help strings based on the language. You must also define the LCIDs for the supported languages.

The following registration file example includes entries for U.S. English and German.

```
// Type library registration information.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826
   -00DD01103DE1}

HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-
   00DD01103DE1}\2.0 =Hello 2.0 Type Library

HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-
   00DD01103DE1}\2.0\HELPDIR
=
// U.S. English.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-
   00DD01103DE1}\2.0\409\win32 = helloeng.tlb
// German.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-
   00DD01103DE1}\2.0\407\win32 = helloger.tlb
```

Passing Formatted Data Using IDataObject

Often, an application needs to accept formatted data as an argument to a method or property. Examples include a bitmap, formatted text, or a spreadsheet range. When handling formatted data, the application should pass an interface reference to an object that implements the COM IDataObject interface.

By using this interface, applications can retrieve the data of any Clipboard format. Because a COM object that supports the IDataObject interface can provide data of more than one format, a caller can provide data in several formats and let the called object choose which format is most appropriate.

If the data object implements IDispatch, it should be passed using the VT_DISPATCH flag. If the data object does not support IDispatch, it should be passed with the VT_UNKNOWN flag.

For more information on the IDataObject interface, see Implementing COM Objects.

Implementing the IEnumVARIANT Interface

Automation defines the IEnumVARIANT interface to provide a standard way for ActiveX clients to iterate over collection of values. Every collection object should expose a read-only property named _NewEnum to let ActiveX clients know that the object supports iteration. The _NewEnum property returns an enumerator object that supports IEnumVARIANT.

The IEnumVARIANT interface provides a way to iterate through the items contained by a collection object. This interface is supported by an enumerator object that is returned by the _NewEnum property of the collection object, as in the following figure.

Implementing the IEnumVARIANT interface

The IEnumVARIANT interface defines these member functions:

- Next. Retrieves one or more elements in a collection, starting with the current element.
- Skip. Skips over one or more elements in a collection.
- Reset. Resets the current element to the first element in the collection.
- Clone. Copies the current state of the enumeration so you can return to the current element after using Skip or Reset.

IEnumVARIANT can be supported by using the COMVariantEnumerator object provided with COM Connect. For more information, see Implementing COM Objects.

The _NewEnum property identifies an object as supporting iteration through the IEnumVARIANT interface. This property has the following requirements:

- Must be named _NewEnum and must not be localized.
- Must return a reference to the enumerator object's IUnknown interface.

- Must use the reserved DISPID for _NewEnum: DISPID_NEWENUM (-4). This constant is defined in COMAutomationConstants.

Returning an Error

ActiveX objects typically return rich contextual error information, including an error number, a description of the error, and the path of a Help file that supplies more information. Objects that do not need to return detailed error information can simply return an HRESULT that indicates the nature of the error.

Passing Exceptions Through IDispatch

When an error occurs, objects invoked through IDispatch can return DISP_E_EXCEPTION and pass the details in the pexcepinfo parameter (an EXCEPINFO structure) to IDispatch::Invoke.

Troubleshooting Q & A

Problem:

The client gets the error RPC_E_SERVERFAULT when invoking virtual function table methods for a dual interface object. From a COM Connect client, the walkback title would read: “Unhandled Exception: The server threw an exception (HRESULT RPC_E_SERVERFAULT)”.

Solution 1:

The dual interface classes need to be re-initialized. For example:

```
IAllDataTypesDispPointer ClassInitializer.
IAllDataTypesDispImplementation ClassInitializer.
IAllDataTypesDisp ClassInitializer.
```

Solution 2:

This usually means that the server object has not been created. This can happen when you attempt to register a class factory for a CLSID that is already in use. Since the CLSID is in use for another class factory, the new class is not registered. For example, when you upgrade the sample class

AutomationAllDataTypes from being an Automation only object to AllDataTypesCOMObject, which supports a dual interface, make sure to un-install the old class.

Solution 3:

Does the **`.reg`** file point to the image and object engine that your server application needs to use? Did you update your server image but forgot to copy it to the location indicated by the **`.reg`** file?

Problem:

The COM server does not start, causing an error.

Solution:

If you deleted your COM server **`.exe`** file, COM changed the Registration Database to reflect the deletion or the move to the deleted file space. If you copy your COM server to the same place, the Registration Database is not modified. You must re-register the COM server by running its **`.reg`** file.

9

Publishing Automation Objects

Creating a Registration File

Before an application can use COM and Automation, the COM objects must be registered with the user's system registration database. Sample registration files to perform this task are provided for the COM objects and the sample applications. Registration makes the following possible:

- ActiveX clients can create instances of the objects through CoCreateInstance. In COM Connect, this is encapsulated in the classes IClassFactory and COMDispatchDriver.
- Automation tools can find the type libraries that are installed on the user's computer.
- COM can find code for the dealing with interfaces remotely.

Registering the Application

Registration maps the ProgID of the application to a unique CLSID, so that you can create instances of the application by name, rather than by CLSID. For example, registering Microsoft Excel associates a CLSID with the ProgID Excel.Application. In COM Connect, you use the ProgID to create an instance of the application as follows:

```
aDispatchDriver := COMDispatchDriver createObject: 'Excel.Application'.
```

You can pass a ProgID or GUID with the following COMDispatchDriver class messages:

- pathName: aFileName
- onActiveObject: aCLSIDOrProgID

- pathName: aFileName progID: aCLSIDOrProgID
- createObject: aCLSIDOrProgID serverName: serverName

Only applications whose objects are created via their ProgID or CLSID need to be registered.

The registration file uses the following syntax for the application:

```
\AppName.ObjectName[.VersionNumber] = human_readable_string
\AppName.ObjectName\CLSID = {UUID}
```

Where the names are defined as:

AppName

The name of the application.

ObjectName

The name of the object to be registered.

VersionNumber

The optional version number of the object.

human_readable_string

A string that describes the application to users. The recommended maximum length is 40 characters.

GUID

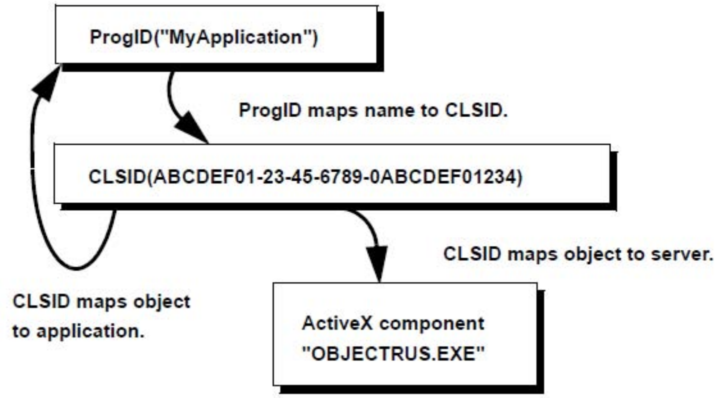
The universally unique identifier for the application CLSID. To generate a UUID for your class, evaluate the expression: GUID new

Registering Classes

Objects that can be created through Automation must be registered with the system. For these objects, registration maps a CLSID to the Automation component file (**`.dll`** or **`.exe`**) in which the object resides. The CLSID also maps an ActiveX object back to its application and ProgID.

The following figure shows how registration connects ProgIDs, CLSIDs, and ActiveX components.

ProgIDs and Clsids in the registry



The type library can be obtained from its CLSID using the following syntax:

```
\CLSID\TypeLib = {UUID of type library}
```

The following syntax indicates that the server is an ActiveX component:

```
\CLSID\Programmable
```

The following example shows required COM class registry keys:

```
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826
   -00DD01103DE1}
= Hello 2.0 Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-
   00DD01103DE1}
\ProgID = Hello.Application.2
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826
   00DD01103DE1}
\VersionIndependentProgID = Hello.Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-
   00DD01103DE1}
\LocalServer32 = hello.exe /Automation
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-
   00DD01103DE1}
\TypeLib = {F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\CLSID\
   {F37C8061-4AD5-101B-B826-00DD01103DE1}\Programmable
```

The registration file uses the following syntax for each class of each object that the application exposes:

```
\CLSID\{UUID} = human_readable_string
\CLSID\{UUID}\ProgID = AppName.ObjectName.VersionNumber
\CLSID\{UUID}\VersionIndependentProgID = AppName.ObjectName
\CLSID\{UUID}\LocalServer[32] = filepath[/Automation]
\CLSID\{UUID}\InProcServer[32] = filepath[/Automation]
```

Where the names are defined as:

human_readable_string

A string that describes the object to users. The recommended maximum length is 40 characters.

AppName

The name of the application, as specified in the application registration string.

ObjectName

The name of the object to be registered.

VersionNumber

The version number of the object.

UUID

The universally unique identifier for the application `CLSID`. To generate a `UUID` for your class, evaluate the expression: `GUID new`

filepath

The full path and name of the file that contains the object. The optional **/Automation** switch tells the application it was launched for Automation purposes. Specify the switch for the Application object's class.

The `ProgID` and `VersionIndependentProgID` are used by other programmers to gain access to the objects you expose. These identifiers (IDs) should use consistent naming guidelines across all your applications as follows:

- Can contain up to 39 characters.
- Must not contain any punctuation (except for the period).
- Must not start with a digit.

Version-independent names consist of an AppName.ObjectName, without a version number. For example, Word.Document or Excel.Chart.

Version-dependent names consist of an AppName.ObjectName.VersionNumber, such as Excel.Application.5.

LocalServer[32]

Indicates that the ActiveX component is an **.exe** file and runs in a separate process from the ActiveX client. The optional 32 specifies a server intended for use on 32-bit Windows systems.

InProcServer[32]

Indicates that the ActiveX component is a DLL and runs in the same process space as the ActiveX client. The optional 32 specifies a server intended for use on 32-bit Windows systems. The filepath you register should give the full path and name. Applications should not rely on the MS-DOS PATH variable to find the object.

Registering a Type Library

Tools and applications that expose type information must register the information so that it is available to type browsers and programming tools. The correct registration entries for a type library can be generated by calling the RegisterTypeLib function on the type library. This operation is performed by the COMTypeLibrary method createRegistration. The **regedit.exe** file supplied with the Win32 SDK, as well as with Windows NT and Windows 95, can then be used to write the registration entries from a text file into the system registration database.

The following information is registered for a type library:

```
\TypeLib\{libUUID}
\TypeLib\{libUUID}\major.minor = human_readable_string
\TypeLib\{libUUID}\major.minor\HELPDIR = [helpfile_path]
\TypeLib\{libUUID}\major.minor\Flags = typelib_flags
\TypeLib\{libUUID}\major.minor\lcid\platform =
localized_typelib_filename
```

Where the names are defined as:

libUUID

The universally unique ID of the type library.

major.minor

The two-part version number of the type library. If only the minor version number increases, all the features of the previous type library are supported in a compatible way. If the major version number changes, code that compiled against the type library must be recompiled. The version number of the type library might differ from the version number of the application.

human_readable_string

A string that describes the type library to users. The recommended maximum length is 40 characters.

helpfile_path

The location of the Help file for types in the type library. If the application supports type libraries for multiple languages, the libraries might refer to different filenames in the Help file directory.

typelib_flags

The hexadecimal representation of the type library flags for this type library. These are the values of the LIBFLAGS enumeration, and are the same flags specified in the uLibFlags parameter to `ICreateTypeLib::SetLibFlags`. These flags cannot have leading zeros or an 0x prefix.

lcid

The hexadecimal string representation of the locale identifier (LCID). It is one to four hexadecimal digits with no 0x prefix and no leading zeros. The LCID might have a neutral sublanguage ID.

platform

The target operating system platform: 16-bit Windows, 32-bit Windows, or Apple® Macintosh®.

localized_typelib_filename

The full name of the localized type library.

Using the LCID specifier, an application can explicitly register the file names of type libraries for different languages. This allows the application to find the desired language without having to open all type libraries with a given name.

For example, to find the type library for Australian English (309), the application first looks for it. If that fails, the application looks for an entry for standard English (a primary identifier of 0x09). If there is no entry for standard English, the application looks for LANG_SYSTEM_DEFAULT (0). For more information on locale support, refer to your operating system manuals for the national language support (NLS) interface.

Note: The registration of type libraries described in this section can be performed automatically when your application uses a `COMTypeLibrary` object with the framework.

Registering Interfaces

Applications that add interfaces need to register the interfaces in the registration database so COM can find the appropriate marshaling code for interprocess communication. By default, Automation registers dispinterfaces that appear in the .odl or .idl file. It also registers remote Automation-compatible interfaces that are not registered elsewhere in the system registry under the label ProxyStubClsid32 (or ProxyStubClsid on 16-bit systems).

The syntax of the information registered for an interface is:

```
\Interface\{UUID} = InterfaceName
\Interface\{UUID}\Typelib = {LIBID}
\Interface\{UUID}\ProxyStubClsid[32] = {CLSID}
```

Where the tags are defined as follows:

UUID

The universally unique ID of the interface.

InterfaceName

The name of the interface.

LIBID

The universally unique ID associated with the type library in which the interface is described.

CLSID

The universally unique ID associated with the proxy/stub implementation of the interface, used internally by COM for interprocess communication. ActiveX objects use the proxy/stub implementation of IDispatch.

Note: The registration of interfaces described in this section can be performed automatically when your application uses a COMTypeLibrary object with the framework.

Example

Here is the example from COM\Examples\COMAuto\ AllDataT\VWAllDT.reg

```
REGEDIT
/
*********************************************************
*
/* VisualWorks COM All Data Types Example
/* Copyright (c) 1997 ObjectShare
/*
/* Summary of GUIDs:
/*
/* CLSID_VWAllDataTypes:{DB5DE8E3-AD1F-11d0-ACBE-
5E86B1000000}
/* Type library: vwAllDT.tlb{DB5DE8E1-AD1F-11d0-ACBE-
5E86B1000000}
/*
/* Created by Gary Gregory
/* Last update: May 19-1997.
/*
/
*********************************************************
*
; Version independent registration. Points to version 1.0
HKEY_CLASSES_ROOT\VisualWorks.AllDataTypes = VisualWorks
   All Data Types
HKEY_CLASSES_ROOT\VisualWorks.AllDataTypes\Clsid =
   {DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}

; Version 1.0 registration
HKEY_CLASSES_ROOT\VisualWorks.AllDataTypes.1 = VisualWorks
   All DataTypes 1.0
HKEY_CLASSES_ROOT\VisualWorks.AllDataTypes.1\Clsid =
   {DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
   5E86B1000000} = VisualWorks All Data Types 1.0
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
   5E86B1000000}\ProgID = VisualWorks.AllDataTypes.1
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
   5E86B1000000}\VersionIndependentProgID =
VisualWorks.AllDataTypes
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
```

```
    5E86B1000000}\LocalServer32 = C:\vw30\Bin\vwnt.exe
C:\vw30\Com\Examples\Automation\vwComSrv.im /Automation
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
    5E86B1000000}\TypeLib = {DB5DE8E1-AD1F-11d0-ACBE-
5E86B1000000}
HKEY_CLASSES_ROOT\CLSID\{DB5DE8E3-AD1F-11d0-ACBE-
    5E86B1000000}\Programmable
; About Type Library and interface registrations.
; All interfaces that support virtual function table binding must be
registered.
; The RegisterTypeLib and LoadTypeLib APIs do this automatically
through
; the COMTypeLibrary services.
; eof
```

In this example, note that object engine executable file name is **`vwnt.exe`** and the image file name is **`vwComSrv.im`**. These names must match the object engine and image name to publish.

Creating a Run-Time Image

This section describes how to prepare a run-time image for an object published with an IDispatch and a dual interface.

Publishing an Object Through IDispatch

When you are ready to make a run-time image to publish an object through IDispatch, you must do the following:

1. Perform any clean up.
2. Reset the COMSessionManager to install run-time image configuration settings.
3. Configure the COMSessionManager with run-time settings appropriate for your application.
4. Install your Smalltalk classes.

The example of publishing the AutomationAllDataTypes class through an IDispatch is installed as follows:

```
"Un-register the other example to avoid CLSID clash since second
example is an upgrade from the first"
AllDataTypesCOMObject unregister.
"Always"
COMSessionManager installRuntime.
```

```
"Set the directory where COM Connect is installed so type libraries can be
located."
COMSessionManager defaultCOMDirectoryName: 'C:\vw30\COM'.
"Your application run-time installation, for example:"
AutomationAllDataTypes installRuntime.
```

Publishing an Object Through a Dual Interface

When you are ready to make a run-time image to publish an Object through a dual interface, you must do the following:

1. Perform any clean up.
2. Reset the COMSessionManager to install run-time image configuration settings.
3. Configure the COMSessionManager with run-time settings appropriate for your application.
4. Install your COMDualInterfaceObject subclasses.

The example of publishing AutomationAllDataTypes through a dual interface is installed as follows:

```
"Un-register other example to avoid CLSID clash since second example is
an upgrade from the first"
AutomationAllDataTypes unregister.
"Always"
COMSessionManager installRuntime.
"Set the directory where COM Connect is installed so type libraries can be
located."
COMSessionManager defaultCOMDirectoryName: 'C:\vw30\COM'.
"Your application run-time installation, for example:"
AllDataTypesCOMObject installRuntime.
```

These expressions are stored in the **`COM\Examples\COMAuto\servers.txt`** file.

Creating the Deployment Image

You can now save your image with a new name, these examples use the image name **`VwComSrv.im`**. For these examples, the image is then copied to its destination directory:

`COM\Examples\ComAuto`

You must make sure that the object engine image file locations matches with the registration file (**`.reg`**).

Object Server Application Termination Considerations

When you create a deployment image for delivery as a COM object server application, you are responsible for providing the application startup logic that checks the startup conditions and registers the class factories for the COM object classes that your application supports with COM. Your application is also responsible for deciding how and when to shut itself down.

In the simplest case, your object server application should run for as long as it supports objects that were created by COM clients. When the last object created for a client by one of your class factories has been released by the last client using it, your application should shut down by terminating the Smalltalk process. This application termination policy is the common case and is made easy for you to support by facilities provided by COMSessionManager. To specify that the termination policy for your application is that it should shut down when the last object being supported is released, simply include the following expression in your application startup logic:

```
COMSessionManager exitIfNotInUse: true.
```

This configures your image so that the release logic for COM objects that are manufactured by a class factory you registered in your startup logic causes the process to be terminated when the application no longer supports any objects.

To arrange for a COM object server application that never shuts down, use the following expression to configure your image during application startup:

```
COMSessionManager exitIfNotInUse: false.
```

This setting is useful when you want your server to always be executing in anticipation of client requests. Once started, the object server application continues running indefinitely, with your class factories registered with COM and everything prepared to handle client object creation requests immediately.

You might find that you require a more complex application termination policy than the simple case of simply shutting down when there are no longer any COM objects supported. For example, this can occur if your application provides a user interface in addition to manufacturing objects for COM clients. In this case, the application termination policy is something along the lines of “shut down the application when there are no longer any COM objects being supported and any open windows have been closed by the user.”

You are responsible for implementing complex termination policies in whatever manner is suitable for your application. The COMSessionManager class provides you with some services to assistance in writing such application specific termination policies. To determine whether your application is currently supporting any COM objects that have been created by clients, evaluate the expression:

```
COMSessionManager isServerInUse
```

To shut down the object server application, evaluate the expression:

```
COMSessionManager terminateServer
```

The server session termination service revokes the class factories registered by your application with COM and shuts down the Smalltalk process. Determining when to actually terminate the application is your responsibility.

If you are implementing an automation application object following the guidelines defined by Automation for the standard Application object, it is required that your application object support a Quit method, which exits the application and closes all open documents. To implement the Quit command for an application object, the Quit method in your application should conform to the following pattern:

```
Quit
   "Quit the application."
   " ... close all open documents ... "
   COMSessionManager terminateServerDeferred.
```

Note the use of the terminateServerDeferred message in this case. The deferred termination service is necessary here because the Quit method is invoked by a client, and you need to ensure that your function returns to the caller before the Smalltalk process is shut down.

Testing an Object Server Application EXE

After you create the deployment image for your object server application EXE, you need to verify that your application installs and works correctly as a COM object server application.

To test your object server, create (or update) a directory in which the deployed server application files are placed. At a minimum, this typically includes the VisualWorks object engine **vwnt.exe** and the **.im** file of the deployment image you created. Optionally, you might want to include the sources or change files of the image, to enable debugging by a developer. You might also include other files that you intend to deliver with the server, such as the type library and any other supporting files needed by the deployed application, by placing the files either in your deployment directory or in subdirectories. The files and directory structure of your delivery directory is, of course, entirely under your control.

To verify that you correctly installed the image configuration settings and your application startup logic in the object server you are delivering, try running the application directly. As always, you can create a shortcut invocation, or simply double-click on the **`.im`** deployment image if your image is properly configured to run the VisualWorks object engine on image files. Alternatively, in an Explorer view you can drag the **`.im`** deployment image and drop it on the **`vwnt.exe`**.

If you built the deployment image correctly and hooked up your application startup logic, you should now see the expected startup behavior of your application. In most cases, this results in a dialog informing you that your application has not been started with the expected flags, which indicates that it is being executed by COM as an object server application and that your application is therefore immediately terminating itself. This is good. It is what you asked for in this case.

To fully test your object server, you need to register it with COM. If you have not already registered the application you have created, do so now.

First, verify that the **`.reg`** file you created for your application has the correct pathnames to the deployment files you created and placed in the desired delivery directory structure. Make any necessary corrections, then double-click on the **`.reg`** file to register the object server application.

The standard Win32 **`regedit.exe`** program processes the information in the **`.reg`** file and updates the system registry information. If you correctly specified the directory names, the COM library service manager now has all the information that it needs to run your object server application when a client application asks to create a new instance of the object you are publishing.

To verify that your object server application is correctly configured and registered with COM, run a client application to create and exercise your automation object through the standard COM mechanisms. The simplest test case is to start up your VisualWorks development image and try to create an instance of your class using the standard IClassFactory createInstance: message. Specify either the CLSID or the ProgID of the class you are publishing in the server you have just created.

If your server is correctly registered with COM and your application startup logic correctly initializes the application and registers the class factory for each object class supported by the server component, executing your object creation request should result in a short delay while the COM library starts your object server and then creates and returns an interface reference to the new instance of your object, which has been created from the local process server running your server application.

If you have other test drivers available to exercise your object, or you have a code fragment that opens a COMAutomationEditor to exercise your object through a COMDispatchDriver that you construct on the newly created object, you might want to run them now. If you want to see a tracing report to verify that the dispatch functions are being invoked by external function callouts to the object in another process, you can open a COMTraceViewer before creating and testing the objects you are creating in the object server process.

When you are done testing the object, release the interface(s) in the usual fashion. Depending on the server termination policy you implemented in your object server application, you should now verify that the server process has been terminated, if that is what you expected at this point. You can use the Windows task manager to inspect your system and confirm whether the object server application is still running.

Troubleshooting an Object Server Application EXE

Server Startup Problems

Problems with creating objects in your object server usually reflect one of two configuration problems.

The first kind of problem that typically occurs is an error message from COM indicating that the server can not be started when you try to create a new object. This typically reflects either a configuration problem, where some file that is needed to run the executable application is not available, or a problem with the pathnames in the **.reg** file. Check that you have assembled the necessary files in your delivery directory and that the information you installed in the Windows system registry is correct.

The other problem that typically occurs when deploying an object server application is that your object creation requests returns with an error message from COM indicating that the server was started but did not register a class factory. In this case, you probably have not correctly installed your application startup logic, or have failed to properly register the necessary class factory. You can verify your basic startup logic, as described in a previous section, by double-clicking directly on the object server and verifying whether your startup displays the expected message, that it is shutting down the application immediately because it was not run by COM. If you observe the expected behavior but the creation request failed because COM could not obtain the class factory, review your application startup logic to determine why the class factory that you are attempting to use is not registered correctly.

Server Termination Problems

Problems with your object server not terminating as expected might reflect a number of conditions.

The simplest case is when you have failed to specify a server application termination policy for your application, either by using `COMSessionManager` services that configure the image settings to specify process termination when the server is no longer in use, or in some other suitable fashion ensuring that your application terminates at some point. If this is the case, simply correct your defective application logic and rebuild the deployment image.

Other cases of server termination problems are typically more complex to analyze. For example, failure of the object server to terminate when you expect might reflect a reference counting bug in either your object implementation or the client code that you are using to exercise the object. COM interface reference counting bugs are typically difficult to detect and correct. They are usually best tackled by fully testing your object implementation and client code in a Smalltalk development image, using tools such as the `COMTraceViewer` and `COMResourceBrowser`, which are provided with COM Connect to assist in analyzing COM application behavior, before attempting to deploy an object server application.

Stripping an Object Server Application Using RTP

Once your object server is mature and well tested you may wish to remove unnecessary objects from the image to reduce its deployment size. As a first step, save the image with all development and example parcels unloaded. To remove additional classes, shared variables, and methods from your image use RuntimePackager as described here. For more information on use of RuntimePackager refer to the VisualWorks Application Developer's Guide.

1 Load the RuntimePackager parcel into an image with the COM classes and server application ready.

2 Load the Headless parcel if you wish to remove support for windows in the image.

3 Open RuntimePackager (**Tool > Runtime Packager**)

4 Load the parameters file **`comserver.rtp`** located in the VisualWorks COM directory. This file specifies the following:

- All COM classes, shared variables, and methods are retained, except COM examples and development tools. Certain COM example and development classes are retained, if RuntimePackager determines that they are used. You may need to retain additional examples.

- The runtime image will not shutdown when its last window is closed. The default option to shut down the runtime image on last window close would prevent a windowless COM runtime server image from starting up. If you decide your COM server should continue only while any of its windows are open, select the option to shutdown the image on last window close. Otherwise, be certain the server application method startUpApplication specifies

  ```
  COMSessionManager exitIfNotInUse: true
  ```

- This will ensure that the server is shutdown when all its clients have released all references.

- A single step save procedure will be used.

- The stripped image will be saved named as “vwcomsvr”. Rename this entry as you please. The startup class and method specified in RuntimePackager should remain blank.

5 Specify which of your application classes to keep in the deployed image.

6 Scan for unreferenced items.

7 Set the `COMSessionManager` default COM directory string and install the server runtime (see section "Creating a Run-Time Image").

8 Before continuing with stripping the image, we stongly recommend that you save the current image and RuntimePackager parameters file. If the final stripped image later fails due to removal of an unexpected object, the strip procedure can resume from this image.

There are two choices at this step: one can either test the server before stripping or proceed directly to strip the image. The procedure continues through the next two section. You may skip the testing steps if you want to proceed directly to stripping the image.

Test the COM Server Application for Dynamic References

Since the COM server image must be saved and registered in order for it to be used by a client, the RuntimePackager test step cannot be performed without first saving the image before stripping. This test step is important to discover any dynamic references to objects that may otherwise be stripped from the final runtime image.

1 Save the image with the RuntimePackager window open and quit the image.

2 Register the image above with the Windows registry using its registration file (.reg). Ensure the registration file entry for LocalServer32 has the correct path information entered.

3 From the command line, start up the image. Include the `/Automation` flag. For an example see section Starting a Deployed Image Manually.

4 Begin tests in RuntimePackager by pressing its **Begin Test** button.

5 Exercise the server by opening a client on its interfaces then invoking all its methods and accessing its properties. For example, this can be done by a combination of:

- inspecting the COM server interfaces using the OLE/COM Object Viewer, and
- starting another VisualWorks COM image, opening a

`COMAutomationEditor` on the COM server image's automation interfaces, and interactively evaluating Smalltalk expressions that access server properties and invoke functions.

6. Release all client references to the COM server interfaces. In the RuntimePackager, accept dynamic references and end the test.

7. Optionally save the image and RuntimePackager parameters file. Accept the notifier that opens warning that open interfaces will be rebuilt on image startup. Note that unless you specify the path information, by default the image will be saved to the Windows `system32` directory.

Strip the Image

1. Close any unnecessary windows in the server image and select the RuntimePackager option to strip the image. Accept the notifier that opens warning that open interfaces will be rebuilt on image startup.

2. Register the image with the Windows registry using its registration file (.reg). Ensure the registration file entry for LocalServer32 has the correct path information entered.

Once the stripped image is complete be sure to test it thoroughly before distribution.

10

Publishing using the Automation Wizard

The IAAutomationWizard is a tool for publishing COM automation objects. This section briefly describes how to use the tool and assumes you are familiar with creating, implementing, and publishing COM automation objects, as described in the preceding chapters.

To review, there are a few steps that you follow when publishing a COM automation object:

1. Implement the automation object.
2. Create a type library describing each automation object.
3. Map the COM interface functions to your class.
4. Provide class factory support.
5. Create a .reg file to register the object server application.
6. Implement the object server application logic.
7. Create an object server application.

What the Automation Wizard Does

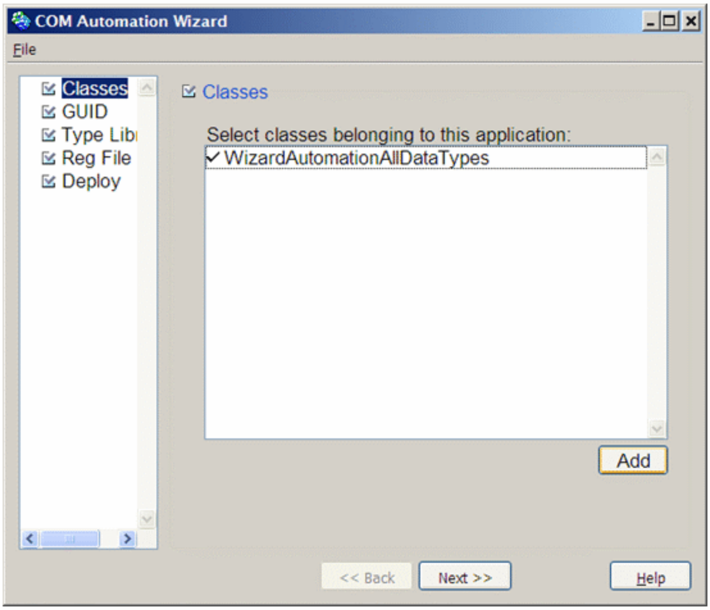
The IAAutomationWizard uses method pragmas to help generate the IDL and type library for the classes you want to publish. See the class and class comment of Examples. WizardAutomationAllDataTypes for examples of using those pragmas. This class is a variation of the AutomationAllDataTypes example, so it is advisable to become familiar with that example before using this tool.

The IAAutomationWizard streamlines the steps listed above, simplifying building an object server. To open up the IAAutomationWizard, select **Tools > COM > Open the Automation Wizard...** from the VisualLauncher.

The wizard walks you through several steps, to identiry classes, GUIDs, the Type Library, the Reg File, and Deployment. The following describes each step and the information required in each.

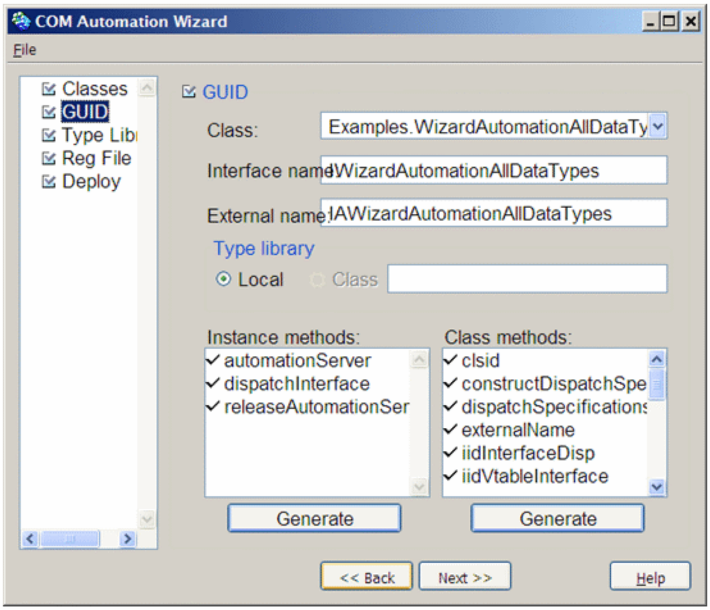
The Classes Step

In this step you select the classes that you want to publish. You can use the Class Finder (by clicking on the magnifying glass button) to find the classes that you want. After you have found a class, select OK from the ClassFinder and click on the Add button to add the class into the selected classes list. Next, select the class(es) that you want to publish.



The GUIDs Step

This step is where the class identifier, support instance, and class methods are created for the class you have selected. There will be default values in the various widgets and those values do not need normally to be modified.



Class: A combo box with the classes that you have added to the wizard.

Interface name: The name of the dispatch interface with which a Smalltalk object is published as a COM automation object.

External name: The name of the COM object as it is known outside of VisualWorks (must be different than Interface name).

Type Library

- **Local:** The selected class that holds the type library support code.
- **Class:** Specify another class to hold the type library support code.

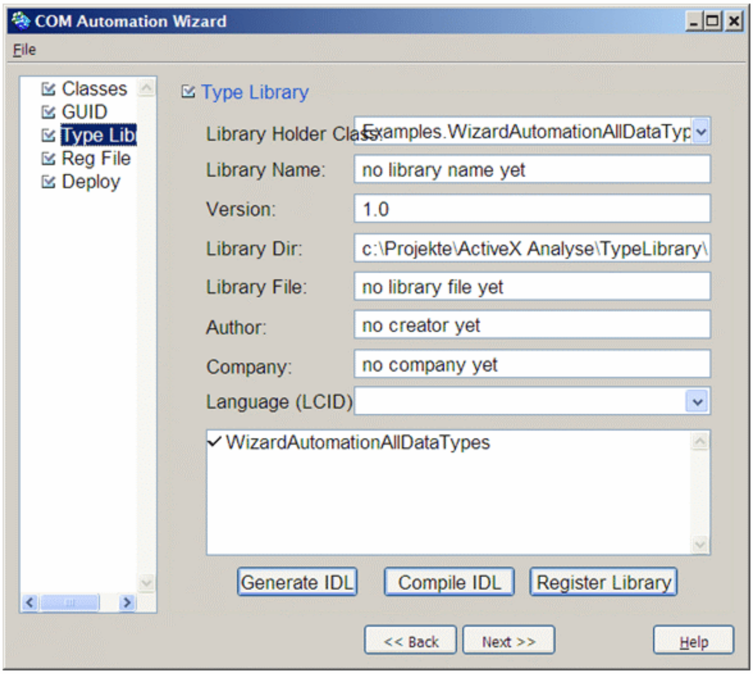
Instance methods: The instance side methods to be auto-generated for the selected class. See Chapter 7, “Implementing Automation Objects.”

Class methods: The class side support methods to be auto-generated for the selected class. See Implementing Automation Objects.

There are two **Generate** buttons. The left one generates the instance methods, and the right one generates the class methods.

The Type Library Step

This step is where the IDL and type library will be auto-generated based on the pragmas that you used in the class to be published.



Library Holder Class: The selected class that will hold the Type Library information.

LibraryName: The name you want to give the library.

Version: Defaults to 1.0 (supply a different version if you prefer).

Library Dir: The full path name to where the library file will be stored.

LibraryFile: The name of the library file that will be generated.

Author: Your name.

Company: Your company name.

There are also three buttons:

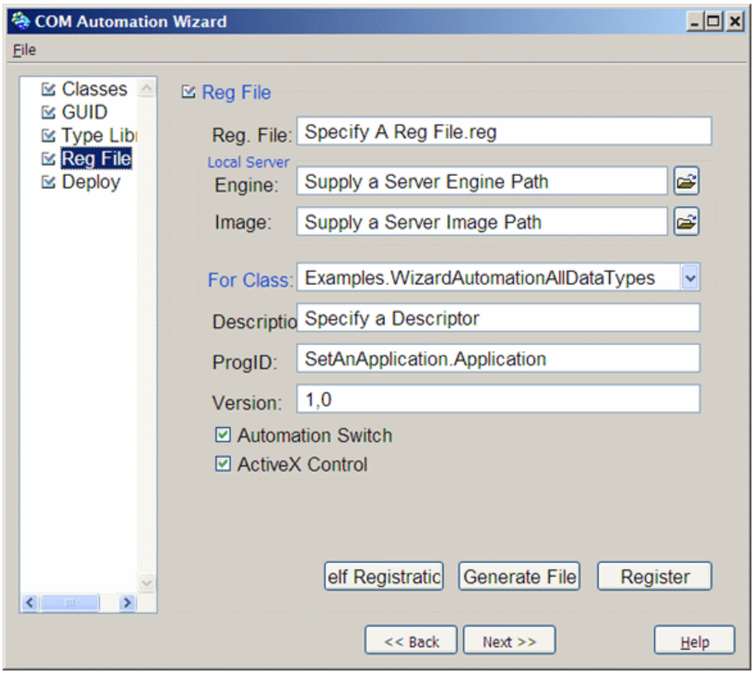
Generate IDL - This button will generate the IDL file that will be used by the MIDL compiler. After it is generated, a TextWindow will open on the file so you can review it for any errors.

Compile IDL - This button will invoke the MIDL compiler to create the library file. A console window will be displayed while the compilation occurs. Then notepad will open to display the log file that was generated by the compilation processes. This log file would contain information on any errors encountered during compilation.

Register Library - This button will register the library with the operating system.

The Reg File Step

This step is where the registration file is created and registered with the operating system.



For Class: The class for which you are generating the registration file.

Reg. File: The name of the registration file (must end with .reg).

Description: See Publishing Automation Objects

ProgID: See Publishing Automation Objects

Version: See Publishing Automation Objects

LocalServer: See Publishing Automation Objects

InProcServer: See Publishing Automation Objects

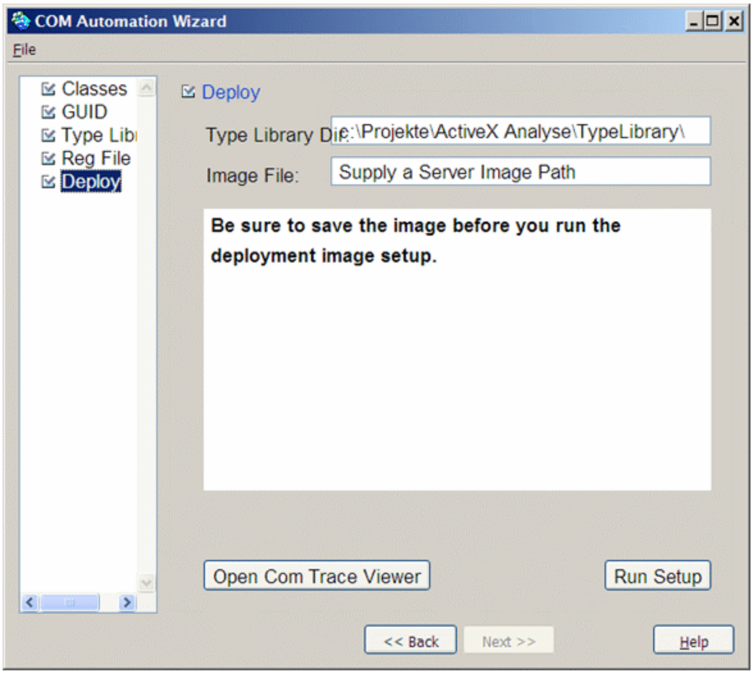
Automation Switch: Check this if you want the server image to start up with the **/AUTOMATION** flag.

ActiveX Control: See Publishing Automation Objects

There are two buttons here: **Generate File** and **Register**. Generate File will generate the file that you can use to register the object in the registry. Register will merge the .reg file with the registry.

The Deploy Step

This step helps you prepare the deployment image. The fields reflect the both the type library directory and the image file as you've defined during the setup.



Saving and loading Settings

The Automation Wizard allows you to save settings and load these settings again. This allow you to repeat the publishing process. Loading and saving settings is done using the respective menu items.

Example of Using the IAAutomationWizard

This is an example of using the IAAutomationWizard, with an illustration showing the properly completed fields in each step.

The Classes Step

1. Use the class finder (click the magnifying glass button) to find the class WizardAutomationAllDataTypes (it is in the examples workspace) and click OK.
2. The fully qualified name of the class will now be in the input field. Click Add.
3. Select the class in the list pane.

The GUIDs Step

1. Select Local.
2. Click on the Generate button under the instance methods list.
3. Click on the Generate button under the class methods list.

The Type Library Step

1. Supply a library name.
2. Set the proper library directory, be sure it is a full path.
3. Set the name to use for the library file that is going to be generated.
4. Set the Author.
5. Set the Company.
6. Click Generate IDL.
7. Click Compile IDL.
8. Click Register Library.

The Reg File Step

1. Supply a name for the reg file.
2. Supply a string for the description.
3. Supply a Prog ID.

4 Set the LocalServer to be something like (be sure to replace with your path to the engine and image):

```
e:\visualworks\7\bin\visual.exe
e:\visualworks\7\image\vwcomsrv.im
```

5 Set the InProcServer to be something like (be sure to replace with your path to the engine and image):

```
e:\visualworks\5i\bin\visual.exe
e:\visualworks\5i\image\vwcomsrv.im
```

6 Click Generate File.

7 Click Register.

The Deploy Step

1 Save your image with the same name as in the Image File input field.

2 Click Run Setup, you should be told that you can now save the image

3 Save the image.

4 Quit the image.

The Test

1 Start up a VisualWorks image and load COM Connect.

2 Open up a class hierarchy browser on the WizardAutomationAllDataTypes.

3 Look at the class comment of the class and follow the instructions.

11

Exposing Classes Through Dual Interfaces

Although Automation allows you to implement an IDispatch interface, a VTBL interface, or a dual interface (which encompasses both), it is strongly recommended that you implement dual interfaces for all exposed ActiveX objects. Dual interfaces have significant advantages over IDispatch-only or VTBL-only interfaces.

- Binding can take place at compile time through the VTBL interface, or at run time through IDispatch.
- ActiveX clients that can use the VTBL interface might benefit from improved performance.
- Existing ActiveX clients that use the IDispatch interface continue to work.
- The VTBL interface is easier to call from C++.

Note: The example class used to illustrate this section is `AllDataTypesCOMObject`, which publishes the class `AutomationAllDataTypes`. While reusing the `AutomationAllDataTypes` class, be aware that there are no requirements on this class. The class methods created for the IDispatch publication example are now unnecessary, since the COMObject framework provides all of the necessary behavior.

Exposing Objects

Publishing a class through a dual interface consists of the following tasks.

- Create or choose a Smalltalk class to publish. If your class is going to keep a copy of any interface pointers, follow the rules for interface reference counting defined under Implementing Automation Objects.
- Create an IDL file describing each class to publish out of the image. Compile the IDL file to a type library.
- Create a data type method for your dual interface in the COMExternalInterface class.
- Create a dual interface virtual function table method in the COMInterfaceVTableSignatures class.
- Create a subclass of COMDualInterfaceObject, this is the object that is actually published to COM. Implement all of the subclass responsibilities.
- Implement a subclass of COMDispatchInterface. This public class is the interface definition for both incoming and outgoing calls.
- Implement a subclass of COMDispatchInterfaceImplementation. This private class defines the low-level virtual function table for incoming interface calls from the COM world.
- Implement a subclass of COMDispatchInterfacePointer if you want to access the COM object from Smalltalk. This private class defines the low-level virtual function table for outgoing interface calls to the COM world.
- Create a **`.reg`** File that describes to COM where to find your application.
- Make a run-time image.

Note: You can use the interface class generator tools provided with COM Connect to assist you in creating the interface wrapper classes from the VTable definition of your dual interface. See COM Connect Development Tools.

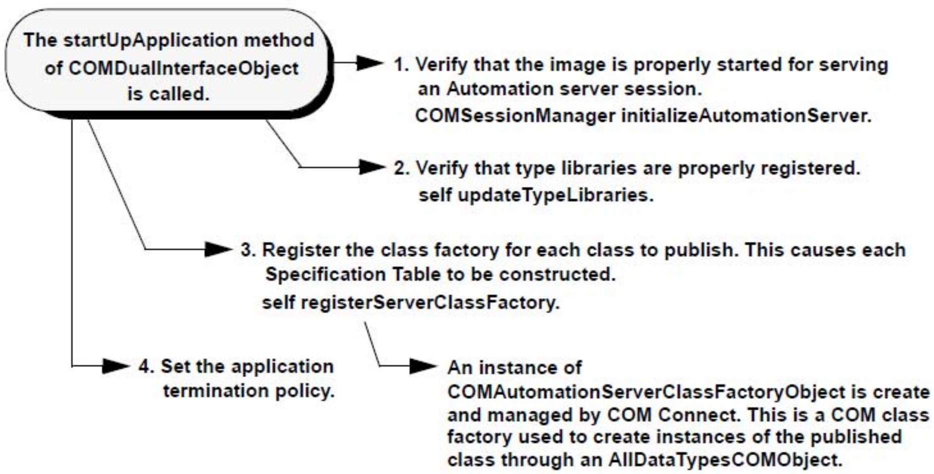
The Big Picture

This section presents an overview of the process of image startup, object creation and object function invocation.

Image Startup

When an object server application image starts up, the startUpApplication method of COMDualInterfaceObject is called, as shown in the following figure:

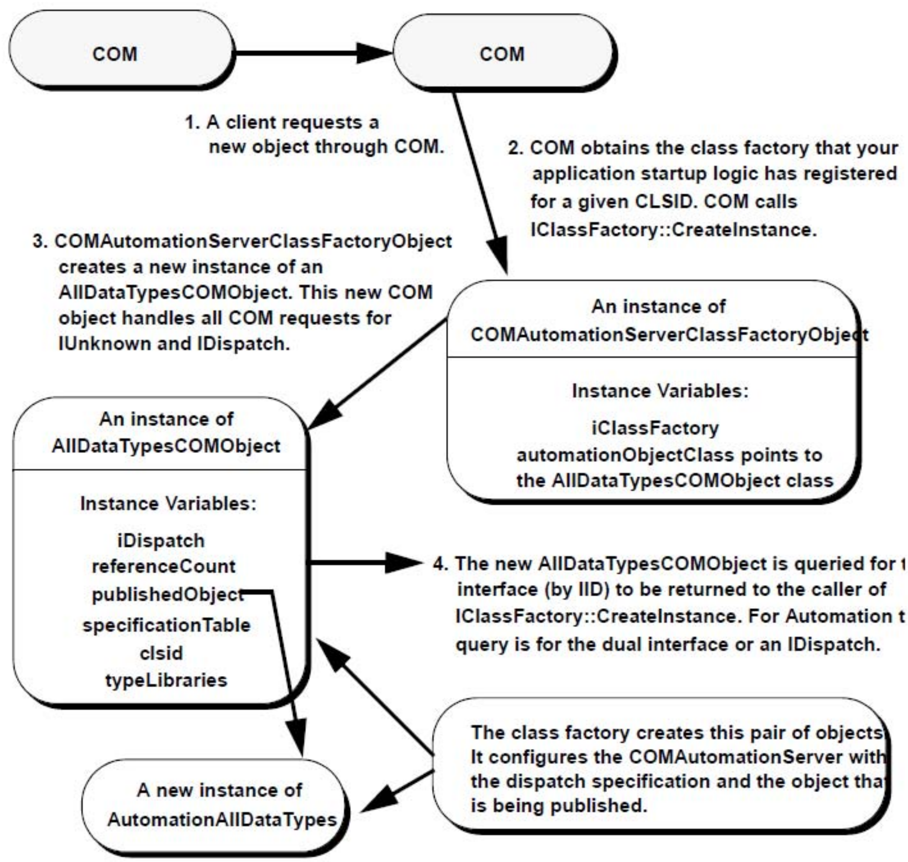
Image startup sequence



Object Creation

When an object server application image is started as a result of a client request to COM to create a new instance of your COM dual interface object, the steps shown in the following figure take place.

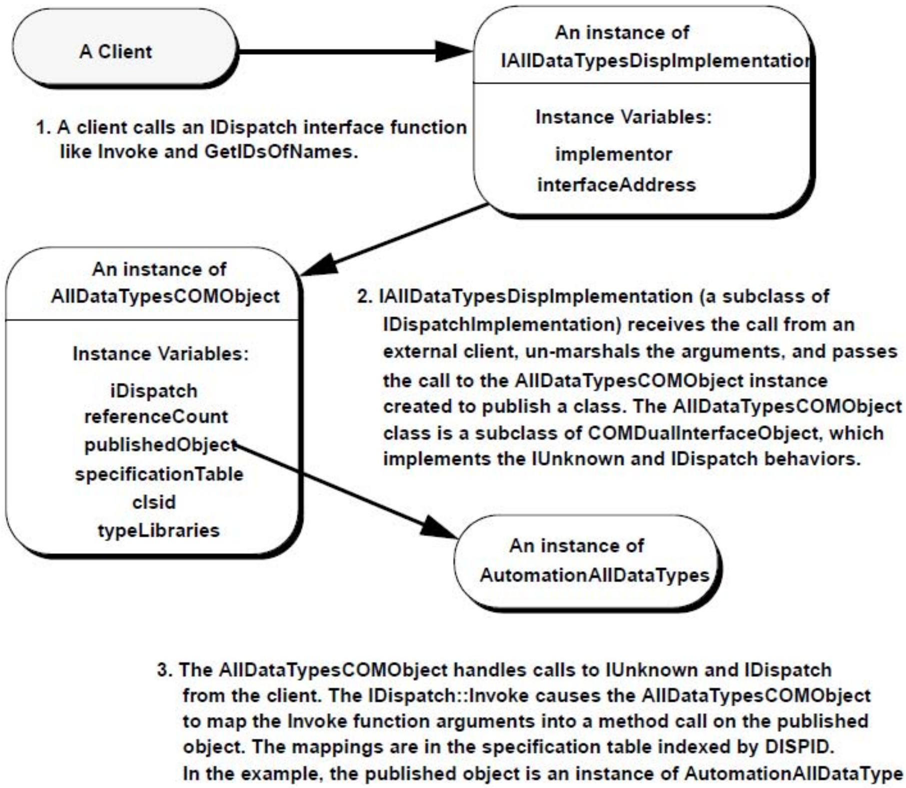
Object creation sequence



Object Function Invocation

When the published object is running and a dispatch invocation call comes in from a client, the steps in following figure occur.

Object function invocation sequence



The Published Class

In the example, the class AutomationAllDataTypes is published. This class is upgraded from being published with an IDispatch to being published with the IAllDataTypesDisp custom dual interface.

IDL Requirements

The IDL file must declare the interface as a dual interface with the **dual** keyword, as shown in the Dual_vwAllDT.idl file:

```
// -------------------------------------------------------------------------------
//
// vwAllDT.idl: IDL source for VisualWorks Automation All Data Types
// Example.
// Publish IAllDataTypesDisp as a dual interface (not just a dispinterface).
//
//**
//** Differences between a dispinterface and dual interface are marked
// with '//**'
//**
//
// This file will be processed by the MIDL compiler to
// produce the type library (vwAllDT.tlb) and marshalling code.
//
// CLSID_VWAllDataTypes:{DB5DE8E3-AD1F-11d0-ACBE-
// 5E86B1000000}
// Type library: vwAllDT.tlb   {DB5DE8E1-AD1F-11d0-ACBE-
// 5E86B1000000}
// Interface:IAllDataTypesDisp{DB5DE8E2-AD1F-11d0-ACBE-
5E86B1000000}
//
// For the legal data types permited on an [oleautomation] interface see
// Microsoft Developer Network Library -- Visual Studio 97 CD:
// mk:@ivt:pdapp/native/sdk/rpc/src/mi-laref_100.htm
//
// -------------------------------------------------------------------------------
cpp_quote("//---------------------------------------------------------------------
")
cpp_quote("//")
cpp_quote("//  VisualWorks Automation: All Data Types Example")
cpp_quote("//  Created by Gary Gregory")
cpp_quote("//  Copyright (C) ObjectShare, 1997.")
cpp_quote("//")
cpp_quote("//---------------------------------------------------------------------
")
//**
//** Declare the dual interface IAllDataTypesDisp
//**
[
   object,
   uuid(DB5DE8E2-AD1F-11d0-ACBE-5E86B1000000), //
      DIID_IAllDataTypesDisp
```

```
    helpstring("VisualWorks All Data Types dispatch interface"),
    pointer_default(unique),
    dual,//** Mark this interface as a dual interface.
    oleautomation
]
interface IAllDataTypesDisp : IDispatch
{
    import "oaidl.idl";
//
// Properties
//
    [propput, helpstring("Sets or returns the LONGValue property
(VT_I4).")]
    HRESULT LONGValue([in] LONG Value);
    [propget]
    HRESULT LONGValue([out, retval] LONG* Value);
//
//... All other property and method definitions...
//
// A method with many arguments, just for show
    [helpstring("Fancy method with many arguments.")]
    HRESULT ManyArguments(
        [in] IDispatch* AnIDispatch,
        [in] BSTR PropertyName,
        [in] LONG Number,
        [out,retval] VARIANT* Value );
};
//
// Component and type library descriptions
//
[
    uuid(DB5DE8E1-AD1F-11d0-ACBE-5E86B1000000), //
        LIBID_VWALLDT
    lcid(0x0409),
    version(1.0),
    helpstring("VisualWorks All Data Types")
]
library VWALLDT
{
    importlib("stdole32.tlb");
    //  Class information for VWAllDataTypes
    [
        uuid(DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000), //
            CLSID_VWAllDataTypes
        helpstring("VisualWorks All Data Types Class."),
    ]
    coclass VWAllDataTypes
```

```
    {
        //**
        //** This next line surfaces the dual interface from the class.
        //**
        [default] interface IAllDataTypesDisp;
    };
};
// eof
```

Creating the Dual Interface Data Type

In the COMExternalInterface class, create a method to define your new dual interface data type. By convention, the name of the method is the same as the interface name. This example defines a type called IAllDataTypesDisp in a method called IAllDataTypesDisp in the framework class COMExternalInterface.

```
IAllDataTypesDisp
    "Define the interface data type. Using __IAnonymous instead of
    __IAllDataTypesDisp is a space optimization that avoids defining
extraneous
    data types that are not used by the COM Connect runtime."

    <C: typedef struct __IAnonymous IAllDataTypesDisp>
```

Note: The data type definition used in the Smalltalk binding for an interface type declaration is always mapped to the standard __IAnonymous structure definition, which defines a structure whose first member is a pointer to a Vtable. Interface structure types are not defined for each interface, because the extra level of Ctype definitions to connect the actual Vtable definition is not needed to support the run-time facilities of the COM Connect interface binding machinery.

Creating the Dual Interface Virtual Function Table Definition

In the class COMInterfaceVTableSignatures, create a method to define the virtual function table layout for your interface. It is important for your virtual function table method to follow the pattern defined below.

The Vtable structure definition must conform to the C declaration of the interface, in which the object that supports the function is explicitly declared in a leading ‘This’ parameter. (In contrast, the receiver is implicit in the equivalent C++ declaration.)

Note that the custom interface function definitions start after the IUnknown and IDispatch function definitions since, by definition, a dual interface implements the IUnknown and IDispatch function.

```
__ISomeInterfaceNameVtbl

<C: struct __ISomeInterfaceNameVtbl {

HRESULT ( __stdcall * QueryInterface)(IAllDataTypesDisp * This, const
   IID * const riid, void * * ppvObject);
ULONG ( __stdcall * AddRef)(IAllDataTypesDisp * This);
ULONG ( __stdcall * Release)(IAllDataTypesDisp * This);

HRESULT ( __stdcall * GetTypeInfoCount)(IAllDataTypesDisp * This,
   UINT * pctinfo);
HRESULT ( __stdcall * GetTypeInfo)(IAllDataTypesDisp * This, UINT
   itinfo, LCID lcid, ITypeInfo * * pptinfo);
HRESULT ( __stdcall * GetIDsOfNames)(IAllDataTypesDisp * This, const
   IID * const riid, LPOLESTR * rgszNames, UINT cNames, LCID lcid,
   DISPID * rgdispid);

HRESULT ( __stdcall * Invoke)(IAllDataTypesDisp * This, DISPID
   dispidMember, const IID * const riid, LCID lcid, WORD wFlags,
   DISPPARAMS * pdispparams, VARIANT * pvarResult, EXCEPINFO *
   pexcepinfo, UINT * puArgErr);

/* The custom function definitions start here */
}>
```

Even though you can write the virtual function table definition from scratch, it is better to use the header file generated by the MIDL compiler. The example IDL processing generated the COM\Examples\COMAuto\AllDataT\TypeLibrary\ midl_VWAllDT.h header file. The relevant section for this task is the C (not C++) definition of the header file.

```
#if defined(__cplusplus) && !defined(CINTERFACE)
(snip)
#else /* C style interface */

   typedef struct IAllDataTypesDispVtbl
   {
      BEGIN_INTERFACE

      HRESULT(STDMETHODCALLTYPE__RPC_FAR *QueryInterface)(
         IAllDataTypesDisp __RPC_FAR * This,
         /* [in] */ REFIID riid,
         /* [iid_is][out] */ void __RPC_FAR *__RPC_FAR *ppvObject);
```

```
ULONG ( STDMETHODCALLTYPE __RPC_FAR *AddRef )(
    IAllDataTypesDisp __RPC_FAR * This);

ULONG ( STDMETHODCALLTYPE __RPC_FAR *Release )(
    IAllDataTypesDisp __RPC_FAR * This);

HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *GetTypeInfoCount )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [out] */ UINT __RPC_FAR *pctinfo);

HRESULT ( STDMETHODCALLTYPE __RPC_FAR *GetTypeInfo )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ UINT iTInfo,
    /* [in] */ LCID lcid,
    /* [out] */ ITypeInfo __RPC_FAR *__RPC_FAR *ppTInfo);

HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *GetIDsOfNames )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ REFIID riid,
    /* [size_is][in] */ LPOLESTR __RPC_FAR *rgszNames,
    /* [in] */ UINT cNames,
    /* [in] */ LCID lcid,
    /* [size_is][out] */ DISPID __RPC_FAR *rgDispId);

/* [local] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *Invoke )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ DISPID dispIdMember,
    /* [in] */ REFIID riid,
    /* [in] */ LCID lcid,
    /* [in] */ WORD wFlags,
    /* [out][in] */ DISPPARAMS __RPC_FAR *pDispParams,
    /* [out] */ VARIANT __RPC_FAR *pVarResult,
    /* [out] */ EXCEPINFO __RPC_FAR *pExcepInfo,
    /* [out] */ UINT __RPC_FAR *puArgErr);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR *put_LONGValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ LONG Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_LONGValue )(
    IAllDataTypesDisp __RPC_FAR * This,
```

```
    /* [retval][out] */ LONG __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR *put_BYTEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ BYTE Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_BYTEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ BYTE __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR *put_SHORTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SHORT Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SHORTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SHORT __RPC_FAR *Value);
    /* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
        __RPC_FAR
        *put_FLOATValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ FLOAT Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_FLOATValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ FLOAT __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR *put_DOUBLEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ DOUBLE Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_DOUBLEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ DOUBLE __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_VARIANT_BOOLValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ VARIANT_BOOL Value);
```

```
/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_VARIANT_BOOLValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ VARIANT_BOOL __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SCODEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SCODE Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SCODEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SCODE __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_DATEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ DATE Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_DATEValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ DATE __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_BSTRValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ BSTR Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_BSTRValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ BSTR __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_IUnknownReference )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ IUnknown __RPC_FAR *Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_IUnknownReference )(
```

```
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ IUnknown __RPC_FAR *__RPC_FAR
        *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_IDispatchReference )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ IDispatch __RPC_FAR *Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_IDispatchReference )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ IDispatch __RPC_FAR *__RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_VARIANTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ VARIANT Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_VARIANTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ VARIANT __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_CURRENCYValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ CURRENCY Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_CURRENCYValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ CURRENCY __RPC_FAR *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SAFEARRAY_I4Value )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SAFEARRAY __RPC_FAR * Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SAFEARRAY_I4Value )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SAFEARRAY __RPC_FAR * __RPC_FAR
```

```
*Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SAFEARRAY_DISPATCHValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SAFEARRAY __RPC_FAR * Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SAFEARRAY_DISPATCHValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SAFEARRAY __RPC_FAR * __RPC_FAR
        *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SAFEARRAY_UNKNOWNValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SAFEARRAY __RPC_FAR * Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SAFEARRAY_UNKNOWNValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SAFEARRAY __RPC_FAR * __RPC_FAR
        *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SAFEARRAY_BSTRValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SAFEARRAY __RPC_FAR * Value);

/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SAFEARRAY_BSTRValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SAFEARRAY __RPC_FAR * __RPC_FAR
        *Value);

/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
    __RPC_FAR
    *put_SAFEARRAY_VARIANTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [in] */ SAFEARRAY __RPC_FAR * Value);
/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
    *get_SAFEARRAY_VARIANTValue )(
    IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ SAFEARRAY __RPC_FAR * __RPC_FAR
```

```
            *Value);
        /* [helpstring] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
            *Quit )(
            IAllDataTypesDisp __RPC_FAR * This);
        /* [helpstring] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
            *Reset )(
            IAllDataTypesDisp __RPC_FAR * This);
        /* [helpstring] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
            *ManyArguments )(
            IAllDataTypesDisp __RPC_FAR * This,
            /* [in] */ IDispatch __RPC_FAR *AnIDispatch,
            /* [in] */ BSTR PropertyName,
            /* [in] */ LONG Number,
            /* [retval][out] */ VARIANT __RPC_FAR *Value);

        END_INTERFACE
    } IAllDataTypesDispVtbl;
(snip)
```

By convention the method defined in COMInterfaceVTableSignatures is named after the interface name prefixed with two underscores '__' and terminated with 'Vtbl'. For the IAllDataTypesDisp interface, a method named __IAllDataTypesDispVtbl was created.

```
__IAllDataTypesDispVtbl

<C: struct __IAllDataTypesDispVtbl {

HRESULT ( __stdcall * QueryInterface)(IAllDataTypesDisp * This, const
    IID * const riid, void * * ppvObject);
ULONG ( __stdcall * AddRef)(IAllDataTypesDisp * This);
ULONG ( __stdcall * Release)(IAllDataTypesDisp * This);

HRESULT ( __stdcall * GetTypeInfoCount)(IAllDataTypesDisp * This,
    UINT * pctinfo);
HRESULT ( __stdcall * GetTypeInfo)(IAllDataTypesDisp * This, UINT
    itinfo, LCID lcid, ITypeInfo * * pptinfo);
HRESULT ( __stdcall * GetIDsOfNames)(IAllDataTypesDisp * This, const
    IID * const riid, LPOLESTR * rgszNames, UINT cNames, LCID lcid,
    DISPID * rgdispid);

HRESULT ( __stdcall * Invoke)(IAllDataTypesDisp * This, DISPID
    dispidMember, const IID * const riid, LCID lcid, WORD wFlags,
    DISPPARAMS * pdispparams, VARIANT * pvarResult, EXCEPINFO *
    pexcepinfo, UINT * puArgErr);

HRESULT ( __stdcall * put_LONGValue )( IAllDataTypesDisp * This,
    LONG Value);
```

```
HRESULT ( __stdcall * get_LONGValue )( IAllDataTypesDisp * This,
   LONG *Value);
HRESULT ( __stdcall * put_BYTEValue )( IAllDataTypesDisp * This, BYTE
   Value);
HRESULT ( __stdcall * get_BYTEValue )( IAllDataTypesDisp * This, BYTE
   *Value);
HRESULT ( __stdcall * put_SHORTValue )( IAllDataTypesDisp * This,
   SHORT Value);
HRESULT ( __stdcall * get_SHORTValue )( IAllDataTypesDisp * This,
   SHORT *Value);
HRESULT ( __stdcall * put_FLOATValue )( IAllDataTypesDisp * This,
   FLOAT Value);
HRESULT ( __stdcall * get_FLOATValue )( IAllDataTypesDisp * This,
   FLOAT *Value);
HRESULT ( __stdcall * put_DOUBLEValue )( IAllDataTypesDisp * This,
   DOUBLE Value);
HRESULT ( __stdcall * get_DOUBLEValue )( IAllDataTypesDisp * This,
   DOUBLE *Value);
HRESULT ( __stdcall * put_VARIANT_BOOLValue )( IAllDataTypesDisp *
   This, VARIANT_BOOL Value);
HRESULT ( __stdcall * get_VARIANT_BOOLValue )( IAllDataTypesDisp *
   This, VARIANT_BOOL *Value);
HRESULT ( __stdcall * put_SCODEValue )( IAllDataTypesDisp * This,
   SCODE Value);
HRESULT ( __stdcall * get_SCODEValue )( IAllDataTypesDisp * This,
   SCODE *Value);
HRESULT ( __stdcall * put_DATEValue )( IAllDataTypesDisp * This,
   DATE Value);
HRESULT ( __stdcall * get_DATEValue )( IAllDataTypesDisp * This, DATE
   *Value);
HRESULT ( __stdcall * put_BSTRValue )( IAllDataTypesDisp * This,
   BSTR Value);
HRESULT ( __stdcall * get_BSTRValue )( IAllDataTypesDisp * This,
   BSTR *Value);
HRESULT ( __stdcall * put_IUnknownReference )( IAllDataTypesDisp *
   This, IUnknown *Value);
HRESULT ( __stdcall * get_IUnknownReference )( IAllDataTypesDisp *
   This, IUnknown **Value);
HRESULT ( __stdcall * put_IDispatchReference )( IAllDataTypesDisp *
   This, IDispatch *Value);
HRESULT ( __stdcall * get_IDispatchReference )( IAllDataTypesDisp *
   This, IDispatch **Value);
HRESULT ( __stdcall * put_VARIANTValue )( IAllDataTypesDisp * This,
   VARIANT Value);
HRESULT ( __stdcall * get_VARIANTValue )( IAllDataTypesDisp * This,
   VARIANT *Value);
HRESULT ( __stdcall * put_CURRENCYValue )( IAllDataTypesDisp *
```

```
    This, CURRENCY Value);
HRESULT ( __stdcall * get_CURRENCYValue )( IAllDataTypesDisp *
    This, CURRENCY *Value);
HRESULT ( __stdcall * put_SAFEARRAY_I4Value )( IAllDataTypesDisp *
    This, SAFEARRAY *Value);
HRESULT ( __stdcall * get_SAFEARRAY_I4Value )( IAllDataTypesDisp *
    This, SAFEARRAY **Value);
HRESULT ( __stdcall * put_SAFEARRAY_DISPATCHValue )(
    IAllDataTypesDisp * This, SAFEARRAY *Value);
HRESULT ( __stdcall * get_SAFEARRAY_DISPATCHValue )(
    IAllDataTypesDisp * This, SAFEARRAY **Value);
HRESULT ( __stdcall * put_SAFEARRAY_UNKNOWNValue )(
    IAllDataTypesDisp * This, SAFEARRAY *Value);
HRESULT ( __stdcall * get_SAFEARRAY_UNKNOWNValue )(
    IAllDataTypesDisp * This, SAFEARRAY **Value);
HRESULT ( __stdcall * put_SAFEARRAY_BSTRValue )(
    IAllDataTypesDisp * This, SAFEARRAY *Value);
HRESULT ( __stdcall * get_SAFEARRAY_BSTRValue )(
    IAllDataTypesDisp * This, SAFEARRAY **Value);
HRESULT ( __stdcall * put_SAFEARRAY_VARIANTValue )(
    IAllDataTypesDisp * This, SAFEARRAY *Value);
HRESULT ( __stdcall * get_SAFEARRAY_VARIANTValue )(
    IAllDataTypesDisp * This, SAFEARRAY **Value);
HRESULT ( __stdcall *Quit )( IAllDataTypesDisp * This);
HRESULT ( __stdcall *Reset )( IAllDataTypesDisp * This);
HRESULT ( __stdcall *ManyArguments )( IAllDataTypesDisp * This,
        /* [in] */ IDispatch *AnIDispatch,
        /* [in] */ BSTR PropertyName,
            /* [in] */ LONG Number,
        /* [retval][out] */ VARIANT *Value);
    }>
```

Note: A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in this example.

Modifying Existing Virtual Function Table Definition

Any time during development that you modify the virtual function table method for an interface, make sure to reinitialize your interface binding classes.

The interface binding classes contain class state derived from the Vtable definition, which is used to support the run-time mechanisms of the COM interface binding. This information must be reset if you modify the interface function definitions while developing your dual interface class.

For example, when modifying the method __IAllDataTypesDispVtbl in the COMExternalInterface framework class, the expression below must be evaluated in order to re-initialize the affected dual interface classes.

```
IAllDataTypesDispPointer ClassInitializer.
IAllDataTypesDispImplementation ClassInitializer.
```

Note: Modifying an interface definition should occur only during initial development. Once you publish the interface, do not change its signature or semantics, since there might be clients relying on your commitment to the “contract” represented by the published COM interface.

Creating the Dual Interface Classes

Two classes must be created: an interface class as a subclass of COMDispatchInterface and an implementation class as a subclass of COMDispatchInterfaceImplementation. To access this object from Smalltalk, you must also create a pointer class as a subclass of COMDispatchInterfacePointer. COM Connect provides tools to assist the creation of necessary wrapper classes for a COM interface. See COM Connect Development Tools.

The AllDataTypes example defines all three interface classes and uses all Automation data types to show you all of the code patterns you use to implement a dual interface object. While most example methods show function calls with only one argument, multiple arguments are handled by simply following the same method patterns defined in this section for each additional argument.

Creating the Interface Class

The IAllDataTypesDisp class is defined as a subclass of COMDispatchInterface. The class IAllDataTypesDisp inherits the basic IDispatch behavior from COMDispatchInterface.

```
Object
   COMInterface ('interface')
      IUnknown
         COMDispatchInterface
            IAllDataTypesDisp
COMDispatchInterface subclass: #IAllDataTypesDisp
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: 'COMStatusCodeConstants
COMAutomationConstants
      COMConstants '
   category: 'COM-Automation-Server Samples'
```

By convention, you always define the interface operations method category for the methods in an interface class. These methods provide a Smalltalk presentation of interface functions.

By convention, you match the function names from the header file generated from the MIDL compiler for interface function names. The interface functions defined for properties are prefixed with get_ for property get functions and put_ for property put functions. There is no modification for method names.

Automatically Generating the Interface Class

A prototype of the interface class of a COM interface can be generated automatically using tools provided with COM Connect. For example,

```
COMInterfaceClassGenerator
   generateInterfacePrototypeFor: #IAllDataTypesDisp.
```

This generates code that you must review.

The COMInterfaceClassGenerator class is discussed under COM Connect Development Tools. You need to review and customize the prototype class, but the tool provides an initial “rough draft” to get you started.

General Pattern for Getting Output Parameter Values

The general method pattern for getting output parameter values for subclasses of COMDispatchInterface is defined as follows:

```
<Get PropertyName Function>
   "Answer the <PropertyName> property."
   | resultReference |
   resultReference := <Value Reference Object>.
   interface <Get PropertyName Function>: resultReference.
   ^resultReference value
```

The Value Reference Object is used to get the output parameter from the function. The expression to create this object depends on the data type of the parameter and is discussed in more detail below.

The interface instance variable holds an instance of the interface pointer class. In the example, this instance variable holds an instance of IAllDataTypesDispPointer.

While this method pattern is defined for properties, it also applies for any argument that needs to be returned through an output parameter. Output parameters are marked in the IDL file with the **out** tag. In addition when they are meant to be a return value, as is the case for returning a property value, they are also marked with the **retval** tag.

Getting Scalar Output Values

To get a scalar output parameter value, the method pattern is as follows:

```
<Get PropertyName Function>
   "Answer the <PropertyName> property."
   | resultReference |
   resultReference := nil asValueReference.
   interface <Get PropertyName Function>: resultReference.
   ^resultReference value
```

This method pattern can be applied for the following data types:

Example instance variable	COM data type	COM Type Code	Smalltalk Class
propertyLONGValue	longLONG	VT_I4	Integer
propertyBYTEValue	unsigned charBYTE	VT_UI1	Integer
propertySHORTValue	shortSHORT	VT_I2	Integer
propertyFLOATValue	floatFLOAT	VT_R4	Float

Example instance variable	COM data type	COM Type Code	Smalltalk Class
propertyDOUBLE-Value	doubleDOUBLE	VT_R8	Double
propertyVARIANT_BOOLValue	booleanBOOLEAN	VT_BOOL	Boolean
propertySCODEValue	SCODE	VT_ERROR	Integer
propertyDATEValue	DATE	VT_DATE	Timestamp
propertyBSTRValue	BSTR	VT_BSTR	String
propertyCURRENCYValue	CURRENCY	VT_CY	FixedPoint with a scale of 4
propertySAFEARRAY_I4Value	SAFEARRAY (LONG)	VT_ARRAY | VT_I4	Array of Integers
propertySAFEARRAY_ BSTRValue	SAFEARRAY (BSTR)	VT_ARRAY | VT_BSTR	Array of Strings
propertySAFEARRAY_ DISPATCHValue	SAFEARRAY (IDispatch*)	VT_ARRAY | VT_DISPATCH	Array of IDispatches
propertySAFEARRAY_ UNKNOWNValue	SAFEARRAY (IUnknown*)	VT_ARRAY | VT_UNKNOWN	Array of IUnknowns
propertySAFEARRAY_ DISPATCHValue	SAFEARRAY (IDispatch*)	VT_ARRAY | VT_DISPATCH	Array of IDispatch

Note: A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in the above table.

By convention, all of the method names match the names from the C header file. For example, the BSTRValue property is accessed by defining the following interface method.

```
get_BSTRValue
    "Answer the BSTRValue property."
    | resultReference |
    resultReference := nil asValueReference.
    interface get_BSTRValue: resultReference.
    ^resultReference value
```

Getting Interface Output Arguments

To get an interface output parameter, the method pattern is as follows:

```
<Get PropertyName Function>
    "Answer the <PropertyName> property."
```

```
    | resultReference |
    resultReference := <Interface Class> new asValueReference.
    interface <Get PropertyName Function>: resultReference.
    ^resultReference value
```

This method pattern can be applied for the following data types:

Example Instance variable	COM data type	COM Type Code	Value Reference Class
propertyIUnknownReference	IUnknown*	VT_UNKNOWN	IUnknown
propertyIDispatchReference	IDispatch*	VT_DISPATCH	IDispatch

By convention, all of the method names match the names from the C header file. For example, the IDispatchReference property is accessed by defining the following interface method:

```
get_IDispatchReference
    "Answer the IDispatchReference property."
    | resultReference |
    resultReference := IDispatch new asValueReference.
    interface get_IDispatchReference: resultReference.
    ^resultReference value
```

Getting VARIANT Output Values

To get a VARIANT output parameter, the method pattern is:

```
<Get PropertyName Function>
    "Answer the <PropertyName> property."
    | resultReference |
    resultReference := COMVariantValueReference new.
    interface <Get PropertyName Function>: resultReference.
    ^resultReference value
```

This method pattern can be applied for the following data types:

Example Instance variable	COM data type	COM Type Code	Smalltalk Class
propertyVARIANT-Value	VARIANT	VT_VARIANT	An object

By convention, all of the method names match the names from the C header file. For example, the propertyVARIANTValue property is accessed by defining the following interface method:

```
get_VARIANTValue
    "Answer the VARIANTValue property."
    | resultReference |
    resultReference := COMVariantValueReference new.
    interface get_VARIANTValue: resultReference.
    ^resultReference value
```

Passing Input Parameter Values

To set an input parameter value, the method pattern is as follows:

```
<Put PropertyName>: aValue
    "Set the <PropertyName> property."
    interface <Put PropertyName Function>: aValue
```

This method pattern can be applied for all data types.

By convention, all of the method names match the names from the C header file. For example, the propertyBSTRValue property is set by defining the following interface method:

```
put_BSTRValue: aValue
    "Set the BYTEValue property."
    interface put_BSTRValue: aValue
```

Calling a Method

To call a COM method, a method is implemented to pass the call to the interface binding that performs the low-level Vtable function call. In an interface class, method names should be “civilized” to follow the Smalltalk convention by starting with a lower-case letter.

For example, the example defines the reset method as:

```
reset
    "Invole the IAllDataTypesDisp::Reset function.."
    interface Reset
```

Calling a Method With Arguments

To call a COM method, a method is implemented to pass the call to the interface binding that performs the low-level Vtable function call. Arguments are handled in the same fashion as defined for properties. In an interface class, method names should be ‘civilized’ to follow the Smalltalk convention by starting with a lower-case letter. The argument selectors should be Smalltalk-like as well, since the anonymous argument selector _: is used only at the function level in the interface pointer and implementation binding classes.

For example, the following defines the manyArguments: propertyName: aLongValue: method.

```
manyArguments: anIDispatch propertyName: aPropertyName
 aLongValue: aLong
    "Call ManyArguments and answer a VARIANT value."
    | resultReference |
    resultReference := COMVariantValueReference new.
    interface ManyArguments: anIDispatch _: aPropertyName _: aLong _:
        resultReference.
    ^resultReference value
```

Class Initialization

The class method used to initialize the class is as follows:

```
ClassInitializer
    " self ClassInitializer "
    self iid: IAllDataTypesDispPointer iid.
    self initialize.
```

Class initialization comprises the following steps:

1. Setting the class interface identifier with iid:.
2. Calling super initialize.

Creating the Interface Implementation Binding Class

Implementing an interface implementation binding class is required when you want a Smalltalk object to support a COM interface. An interface implementation class is used when a client calls into VisualWorks.

The IAllDataTypesDispImplementation class is defined as a subclass of COMDispatchImplementation. The IAllDataTypesDispImplementation class inherits the IDispatch behavior. It is through these methods that COM Connect is invoked from the external world.

```
Object
   COMInterfaceBinding
      COMInterfaceImplementation ('implementor' 'interfaceAddress')
         IUnknownImplementation
            COMDispatchImplementation
               IAllDataTypesDispImplementation

COMDispatchImplementation subclass:
    #IAllDataTypesDispImplementation
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: 'COMConstants COMAutomationConstants
      COMStatusCodeConstants '
   category: 'COM-Automation-Server Samples'
```

By convention, you always define two method categories for the methods in an interface pointer class:

interface operations

These methods provide an optimization for Smalltalk clients that call the interface from the same image. They are also useful for testing your object.

private-invocation

These methods define the interface function callback entry points. These methods receive the external client calls.

The convention used by COM Connect for incoming call is to call a method with a selector prefix of invoke followed by the interface function name. Following the COM Connect argument convention, the anonymous keyword _: is used for argument selectors. The first argument of all interface functions is always noted as the this, which is familiar to C++ programmers.The ‘This’ argument of an external function invocation is provided by the Object Engine COM callback machinery. It is always the interface binding that is processing the callback; it is not used by the image binding.

Automatically Generating the Interface Implementation Class

A prototype of the interface implementation binding class of a COM interface can be generated automatically using tools provided with COM Connect. For example:

```
COMInterfaceImplementationClassGenerator
   generateInterfacePrototypeFor: #IAllDataTypesDisp.
```

This generates code that you must review.

The COMInterfaceImplementationClassGenerator class is discussed under COM Connect Development Tools. You need to review and customize the prototype class, but the tool provides an initial “rough draft” to get you started.

General Pattern for Returning a Value in an Output Parameter

The general method pattern used to return a value in an output parameter is defined as follows:

```
invoke<Get SomeName Function>: this _: pvarResult
"Private. Invoke the <InterfaceName>::<Get SomeName Function>
 function for an external caller. "

"An additional comment with the C function definition from the header
file. "

^[   " terminate exception stack unwind at external callin boundary "

| resultReference hresult |

self reportExternalFunctionEntry.
pvarResult isValid
    ifFalse: [ ^E_INVALIDARG ].
resultReference := <Value Reference Object>.
hresult := implementor <Get SomeName Function>: resultReference.
( HRESULT succeeded: hresult )
    ifTrue: [ <Copy 'resultReference value' to the memory location
        'pvarResult'> ].
hresult
] on: self rootExceptions
    do: ( self exceptionHandlerForHRESULTReturnValue: #externalCallin )
```

The Value Reference Object is used to get the output parameter from the implementor object. The expression to create this object depends on the data type of the parameter. In most cases, the expression ‘nil as Value Reference’ is used in an external function invocation.

The instance variable implementor holds an instance of the published COM class. In the example, this instance variable holds an instance of AllDataTypesCOMObject.

Copying Output Values to External Memory

The following table lists the variations in the method pattern for copying a COM representation of a Smalltalk object into external memory.

COM data type	Value reference class	Setting the result value pointer
BSTR	nil	self bstrResultAtAddress: pvarResult put: resultReference value
BYTE	nil	self scalarResultAtAddress: pvarResult put: resultReference value
CURRENCY	nil	self currencyResultAtAddress: pvarResult put: resultReference value
DATE	nil	self dateResultAtAddress: pvarResult put: resultReference value
DOUBLE	nil	self scalarResultAtAddress: pvarResult put: resultReference value
FLOAT	nil	self scalarResultAtAddress: pvarResult put: resultReference value
IDispatch*	nil	self interfaceResultAtAddress: pvarResult put: resultReference value
IUnknown*	nil	self interfaceResultAtAddress: pvarResult put: resultReference value
LONG	nil	self scalarResultAtAddress: pvarResult put: resultReference value
SAFEARRAY of BSTR	nil	self safeArrayResultPointerAtAddress: pvarResult put: resultReference value elementType: VT_BSTR
SAFEARRAY of DISPATCH	nil	self safeArrayResultPointerAtAddress: pvarResult put: resultReference value elementType: VT_DISPATCH
SAFEARRAY of UNKNOWN	nil	self safeArrayResultPointerAtAddress: pvarResult put: resultReference value elementType: VT_UNKNOWN
SAFEARRAY of VT_I4	nil	self safeArrayResultPointerAtAddress: pvarResult put: resultReference value elementType: VT_I4
SCODE	nil	self scalarResultAtAddress: pvarResult put: resultReference value
SHORT	nil	self scalarResultAtAddress: pvarResult put: resultReference value
VARIANT_BOOL	nil	self variantBoolResultAtAddress: pvarResult put: resultReference value
VARIANT	COMVariant Value-Reference	selfvariantResultAt Address: PvarResult put:resultReference value

A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in above table. The selectors in above table that are sent to self are methods implemented in the COM Connect framework superclass COMInterfaceImplementation.

For example, to answer a BSTR value to a client, in the IAllDataTypesDisp dual interface class a function called invokeget_BSTRValue:_: is defined. The first argument is noted as the this, which is familiar to C++ programmers. The second argument is a pointer to a BSTR.

```
invokeget_BSTRValue: this _: pvarResult
    "Private. Invoke the IAllDataTypes::get_BSTRValue function for an
    external caller. "

    "/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
        *get_BSTRValue )(
        IAllDataTypesDisp __RPC_FAR * This,
        /* [retval][out] */ BSTR __RPC_FAR *Value);"
        ^[   " terminate exception stack unwind at external callin boundary "
    | resultReference hresult |
    self reportExternalFunctionEntry.
    pvarResult isValid
        ifFalse: [ ^E_INVALIDARG ].
    resultReference := nil asValueReference.
    hresult := implementor get_BSTRValue: resultReference.
    ( HRESULT succeeded: hresult )
        ifTrue: [ self bstrResultAtAddress: pvarResult
            put: resultReference value ].
    hresult

    ] on: self rootExceptions
        do: ( self exceptionHandlerForHRESULTReturnValue:
            #externalCallin )
```

General Pattern for Getting Values From Input Parameters

The general method pattern used to get a value from an input parameter is defined as follows:

```
invoke<Put SomeName Function>: this _: aValue
    "Private. Invoke the <InterfaceName>:: <Put SomeName Function>
     function for  an external caller. "
    "An additional comment with the C function definition from the header
     file. "
    ^[   " terminate exception stack unwind at external callin boundary "
    implementor <Put SomeName Function>:
```

```
            ( <Convert 'aValue' to its Smalltalk  representation> ).
        ] on: self rootExceptions
            do: ( self exceptionHandlerForHRESULTReturnValue: #externalCallin
    )
```

This table lists the variations in the method pattern.

COM data type	Getting the value
BSTR	self stringAtBSTRPointer: aValue
BYTE	aValue
CURRENCY	self currencyValueAtAddress: aValue
DATE	self dateValueAtAddress: aValue
DOUBLE	aValue
FLOAT	aValue
IDispatch*	self interfaceAtAddress: aValue type: IDispatch
IUnknown*	self interfaceAtAddress: aValue type: IDispatch
LONG	aValue
SAFEARRAY of BSTR	self safeArrayValueAtAddress: pvarResult put: aValue elementType: VT_BSTR
SAFEARRAY of DISPATCH	self safeArrayValueAtAddress: pvarResult put: aValue elementType: VT_DISPATCH
SAFEARRAY of UNKNOWN	self safeArrayValueAtAddress: pvarResult put: aValue elementType: VT_UNKNOWN
SAFEARRAY of VT_I4	self safeArrayValueAtAddress: pvarResult put: aValue elementType: VT_I4
SCODE	aValue
SHORT	aValue
VARIANT	self variantValueFrom: aValue
VARIANT_BOOL	self booleanFromVariantBool: aValue

Note: A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in above table. The selectors in above table that are sent to self are methods implemented in the framework class COMInterfaceImplementation.

For example, to set a BSTR value in an object, a function called invokeput_BSTRValue:_: is defined in the IAllDataTypesDisp dual interface class. The first argument is noted as the this familiar to C++ programmers. The second argument is a BSTR.

```
invokeput_BSTRValue: this _: aValue
    "Private. Invoke the IAllDataTypes::put_BSTRValue function for an
    external caller. "

    "/* [helpstring][propput] */ HRESULT ( STDMETHODCALLTYPE
        __RPC_FAR
        *put_BSTRValue )(
        IAllDataTypesDisp __RPC_FAR * This,
        /* [in] */ BSTR Value);"

    ^[   " terminate exception stack unwind at external callin boundary "
    implementor put_BSTRValue: ( self stringAtBSTRPointer: aValue ).
    ] on: self rootExceptions
        do: ( self exceptionHandlerForHRESULTReturnValue:
            #externalCallin )
```

Optimizing Same Image Clients

You can publish COM objects from many places, including the image you are running from. When a COM object is implemented in the same image it is called from, it is possible to optimize the call path by avoiding going out to COM and back in again. This optimization is enabled by implementing interface methods on the interface implementation class. Invoking interface functions internally is also useful for testing your object during development. By convention, these methods are placed in a category called ‘interface operations.’

The method pattern is as follows:

```
<Interface Function Name>
    "Invoke the IAllDataTypes:: <Interface Function Name> function. Raise
    an exception if an error occurs. Answer the result code."

    | hresult |
    hresult := [  " terminate exception stack unwind at function invocation
        boundary "
        implementor <Interface Function Call>
        ] on: self rootExceptions
            do: ( self exceptionHandlerForHRESULTReturnValue:
                #internalCallin ).
    self checkHresult: hresult.
    ^hresult
```

For example, the example IAllDataTypesDispImplementation method get_BSTRValue: is defined as follows:

```
get_BSTRValue: resultReference
    "Invoke the IAllDataTypes::get_BSTRValue function. Raise an exception
    if anerror occurs. Answer the result code."
    | hresult |
    hresult := [  " terminate exception stack unwind at function invocation
    boundary "
        implementor get_BSTRValue: resultReference
        ] on: self rootExceptions
            do: ( self exceptionHandlerForHRESULTReturnValue:
                #internalCallin ).
    self checkHresult: hresult.
    ^hresult
```

Class Initialization

The class method used to initialize the class is as follows:

```
ClassInitializer
    " self ClassInitializer "
    self iid: IAllDataTypesDispPointer iid.
    self vtableSignatureTypeName: #__IAllDataTypesDispVtbl.
    self initialize.
```

Class initialization comprises the following steps:

1. Set the class interface identifier with iid:.
2. Specify the name of the virtual function table with the vtableSignatureTypeName: method. The argument is a selector for the virtual function table method defined for the dual interface in the COMInterfaceVTableSignatures class.
3. Call super initialize.

Creating the Interface Pointer Binding Class

Implementing an interface pointer class is required when you want a Smalltalk client to access a COM object implemented by another application. An interface pointer class is used to call out of VisualWorks.

The class IAllDataTypesDispPointer is defined as a subclass of COMDispatchPointer.The class IAllDataTypesDispPointer inherits IDispatch behavior from COMDispatchInterfacePointer. It is through these methods that COM Connect calls out to the COM world.

```
Object
   COMInterfaceBinding
      COMInterfacePointer ('interfaceAddress')
         IUnknownPointer
            COMDispatchInterfacePointer
               IAllDataTypesDispPointer

COMDispatchInterfacePointer subclass: #IAllDataTypesDispPointer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: 'COMStatusCodeConstants
      COMAutomationConstants  COMConstants '
   category: 'COM-Automation-Server Samples'
```

By convention, you always define two method categories for the methods in an interface pointer class:

interface operations

These methods convert as necessary any arguments for calling the functions.

private-invocation

These methods define the function entry points for the external function call.

Automatically Generating the Interface Pointer Class

A prototype of the interface pointer class of a COM interface can be generated automatically using tools provided with COM Connect. For example,

```
COMInterfacePointerClassGenerator
   generateInterfacePrototypeFor: #IAllDataTypesDisp.
```

This generates code that you must review.

The COMInterfacePointerClassGenerator class is documented under COM Connect Development ToolsCOM Connect Development Tools. You need to review and customize the prototype class, but the tool provides an initial “rough draft” to get you started.

Getting Output Parameter Values

The general pattern for methods getting output parameter values in the ‘interface operations’ category is as follows:

```
<Get Function name>: resultReference
    "Invoke the IAllDataTypesDisp::<Get Function name> function. Raise an
    exception if an error occurs. Answer the result code."
    | resultBuffer hresult |
    resultBuffer := <Create result value buffer>.
    hresult := self invoke<Get Function name>: resultBuffer
        asPointerParameter.
    resultReference value: ( <Get value from buffer> ).
    ^hresult
```

The following table lists variations in the method pattern.

COM data type	Creating the result value buffer	Getting the value from the buffer
BSTR	BSTR resultValueBuffer.	resultBuffer contents
BYTE	COMExternalInterface scalarResultBufferFor: #BYTE.	resultBuffer contents
CURRENCY	COMStructure resultValueBuffer: #CY.	resultBuffer contents
DATE	COMDate resultValueBuffer.	resultBuffer contents
DOUBLE	COMExternalInterface scalarResultBufferFor: #DOUBLE.	resultBuffer contents
FLOAT	COMExternalInterface scalarResultBufferFor: #FLOAT	resultBuffer contents
IDispatch*	IDispatchPointer resultValueBuffer.	resultBuffer contents
IUnknown*	IUnknownPointer resultValueBuffer.	resultBuffer contents
LONG	COMExternalInterface scalarResultBufferFor: #LONG.	resultBuffer contents
SAFEARRAY of BSTR	COMSafeArray resultValueBufferFor: VT_BSTR	resultBuffer contents
SAFEARRAY of DISPATCH	COMSafeArray resultValueBufferFor: VT_DISPATCH	resultBuffer contents
SAFEARRAY of UNKNOWN	COMSafeArray resultValueBufferFor: VT_UNKNOWN	resultBuffer contents
SAFEARRAY of VT_I4	COMSafeArray resultValueBufferFor: VT_I4	resultBuffer contents
SCODE	COMExternalInterface scalarResultBufferFor: #SCODE	resultBuffer contents
SHORT	COMExternalInterface scalarResultBufferFor: #SHORT.	resultBuffer contents

COM data type	Creating the result value buffer	Getting the value from the buffer
VARIANT	COMStructure resultValueBufferFor: #VARIANT.	resultBuffer contents
VARIANT_BOOL	COMExternalInterface scalarResultBufferFor: #VARIANT_BOOL	COMExternalInterface booleanFromVARIANT_BOOL: resultBuffer contents

A SAFEARRAY can be of any COM Automation-compatible type. Not all combinations for SAFEARRAYs are shown in above table.

The general pattern for methods in the 'private-invocation' category is defined below. By convention all interface methods are prefixed with invoke.

```
invoke<Function Name>
   "Private."
   " The C header file interface Function definition is included here "
   <COM: HRESULT <Function definition>= <Virtual function table
      position>>
   ^self externalAccessFailedWith: _errorCode
```

Note: The virtual function table position is a 0-based index, with 0 predefined for IUnknown::QueryInterface, 1 for IUnknown::AddRef and so on until 6 for IDispatch::Invoke. Therefore, the first available index for a dual interface entry is always 7.

For example, the IAllDataTypesDispPointer class implements the get_BSTRValue: and invokeget_BSTRValue: methods.

```
get_BSTRValue: resultReference
   "Invoke the IAllDataTypesDisp::get_BSTRValue function. Raise an
   exception if an error occurs. Answer the result code."
   | resultBuffer hresult |
   resultBuffer := BSTR resultValueBuffer.
   hresult := self invokeget_BSTRValue: resultBuffer asPointerParameter.
   resultReference value: resultBuffer contents.
   ^hresult

invokeget_BSTRValue: Value
   "Private."
   "/* [propget] */ HRESULT ( STDMETHODCALLTYPE __RPC_FAR
      *get_BSTRValue )(
   IAllDataTypesDisp __RPC_FAR * This,
    /* [retval][out] */ BSTR __RPC_FAR *Value);"
   <COM: HRESULT __stdcall get_BSTRValue(BSTR *Value) = 24>
   ^self externalAccessFailedWith: _errorCode
```

Setting Input Parameter Values

The following sections introduce method patterns for converting a Smalltalk object into its COM representation suitable for an external interface function call.

Setting Input Parameters for Scalar Values

This section applies to following COM Automation data types:

- BYTE
- DOUBLE
- FLOAT
- LONG
- SCODE
- SHORT

The method pattern to convert one of these values from a Smalltalk object into COM bits is as follows. There is actually no work to do for scalar types, in the pattern and example the value in aValue is just passed to the interface function entry point.

```
<Put Function Name>: aValue
    "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
    exception if an error occurs. Answer the result code."
    ^self invoke<Put Function Name>: aValue
```

For example, the IAllDataTypesDispPointer class defines the put_BYTEValue: method to deal with a BYTE input parameter, as follows:

```
put_BYTEValue: aValue
    "Invoke the IAllDataTypesDisp::put_BYTEValue function. Raise
    an exception if an error occurs. Answer the result code."
    ^self checkHresult: ( self invokeput_BYTEValue: aValue )
```

Setting Input Parameters for BSTR Values

The method pattern to convert a Smalltalk String into COM bits is as follows:

```
<Put Function Name>: aValue
    "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
        exception if an error occurs. Answer the result code."
    | aBSTR hresult |
    aBSTR := BSTR allocateString: aValue.
    [
```

```
        hresult := self invoke<Put Function Name>:
            aBSTR asPointerParameter.
        ] ensure: [ aBSTR release ].
        ^hresult
```

For example, the IAllDataTypesDispPointer class defines the put_BSTRValue: method to deal with a BSTR input parameter, as follows:

```
    put_BSTRValue: aValue
        "Invoke the IAllDataTypesDisp::put_BSTRValue function. Raise an
        exception if an error occurs. Answer the result code."
        | aBSTR hresult |
        aBSTR := BSTR allocateString: aValue.
        [
        hresult := self invokeput_BSTRValue: aBSTR asPointerParameter.
        ] ensure: [ aBSTR release ].
        ^hresult
```

Setting Input Parameters for CURRENCY Values

The method pattern to convert a Smalltalk FixedPoint representing a CURRENCY value into COM bits is as follows:

```
    <Put Function Name>: aNumber
        "Invoke the <InterfaceName>:: <Put Function Name>function. Raise an
            exception if an error occurs. Answer the result code.
            <aNumber> represents a CURRENCY with a scale of 4."

        ^self invoke<Put Function Name>:
            ( COMExternalInterface asCYParameter: aNumber )
```

For example, the IAllDataTypesDispPointer class defines the put_CURRENCYValue: method to deal with a CURRENCY input parameter, as follows:

```
    put_CURRENCYValue: aFixedPoint
        "Invoke the IAllDataTypesDisp::put_CURRENCYValue function. Raise
        an  exception if an error occurs. Answer the result code. <aFixedPoint>
        represents a CURRENCY with a scale of 4."

        ^self invokeput_CURRENCYValue:
            ( COMExternalInterface asCYParameter: aNumber )
```

Setting Input Parameters for DATE Values

The method pattern to convert a Smalltalk Date, Timestamp or LimitedPrecisionReal into COM bits is as follows:

```
<Put Function Name>: aValue
   "Invoke the <InterfaceName>::<Put Function Name> function with the
   argument <aValue> is a Timestamp or a Date. Raise an exception if an
   error occurs. Answer the result code."

   ^ self invoke<Put Function Name>:
      ( COMExternalInterface asDATEParameter: aValue ) )
```

For example, the IAllDataTypesDispPointer class defines the put_DATEValue: method to deal with a DATE input parameter, as follows:

```
put_DATEValue: aValue
   "Invoke the IAllDataTypesDisp::put_DATEValue function with the
   argument <aValue> is a Timestamp or a Date. Raise an exception if an
   error occurs. Answer the result code."

   ^self invokeput_DATEValue:
      ( COMExternalInterface asDATEParameter: aValue )
```

Setting Input Parameters for Interface Values

The method pattern to convert a Smalltalk interface into COM bits is as follows:

```
<Put Function Name>: aValue
   "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
   exception if an error occurs. Answer the result code."
   ^self invoke<Put Function Name>: aValue asPointerParameter
```

For example, the IAllDataTypesDispPointer class defines the put_IDispatchReference: method to deal with an IDispatch input parameter, as follows:

```
put_IDispatchReference: aValue
   "Invoke the IAllDataTypesDisp::put_IDispatchReference function. Raise
   an exception if an error occurs. Answer the result code."

   ^self invokeput_IDispatchReference: aValue asPointerParameter
```

Setting Input Parameters for SAFEARRAY Values

The method pattern to convert an Array into COM bits is as follows:

```
<Put Function Name>: aValue
    "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
        exception if an error occurs. Answer the result code."
    | hresult aComSA |
    aComSA := COMSafeArray fromCollection: aValue elementType: <An
        Automation VT Constant>
    [
    hresult := self invoke<Put Function Name>:
        aComSA asPointerParameter.
    ] ensure: [ aComSA release ].
    ^hresult
```

For example, the IAllDataTypesDispPointer class defines the put_SAFEARRAY_DISPATCHValue: method to deal with a SAFEARRAY of IDispatch references input parameter, as follows:

```
put_SAFEARRAY_DISPATCHValue: aValue
    "Invoke the IAllDataTypesDisp::put_SAFEARRAY_DISPATCHValue
function.
        Raise an exception if an error occurs. Answer the result code."
    | hresult aComSA |
    aComSA := COMSafeArray fromCollection: aValue elementType:
        VT_DISPATCH.
    [
    hresult := self invokeput_SAFEARRAY_DISPATCHValue: aComSA
        asPointerParameter.
    ] ensure: [ aComSA release ].
    ^hresult
```

Setting Input Parameters for VARIANT Values

The method pattern to convert any Smalltalk object whose class is compatible with a COM Automation data type into COM bits is as follows:

```
<Put Function Name>: aValue
put_VARIANTValue: aValue
    "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
    exception if an error occurs. Answer the result code."

    ^self invoke<Put Function Name>:
    aValue asCOMVariant asStructureParameter
```

For example, the IAllDataTypesDispPointer class defines the put_VARIANTValue: method to deal with a VARIANT input parameter, as follows:

```
put_VARIANTValue: aValue
    "Invoke the IAllDataTypesDisp::put_VARIANTValue function. Raise an
    exception if an error occurs. Answer the result code."

    ^self invokeput_VARIANTValue: aValue asCOMVariant
asStructureParameter
```

Setting Input Parameters for VARIANT_BOOL Values

The method pattern to convert a Smalltalk Boolean into COM bits is as follows:

```
<Put Function Name>: aValue
    "Invoke the <InterfaceName>::<Put Function Name> function. Raise an
    exception if an error occurs. Answer the result code."

    ^self invoke<Put Function Name>:
        ( COMExternalInterface asVARIANT_BOOL: aValue )
```

For example, the IAllDataTypesDispPointer class defines the put_DATEValue: method to deal with a DATE input parameter, as follows:

```
put_VARIANT_BOOLValue: aValue
    "Invoke the IAllDataTypesDisp::put_VARIANT_BOOLValue function.
    Raise an exception if an error occurs. Answer the result code."

    ^self invokeput_VARIANT_BOOLValue:
        ( COMExternalInterface asVARIANT_BOOL: aValue )
```

Class Initialization

The class method used to initialize the class is as follows:

```
ClassInitializer
    " self ClassInitializer "
    " '{DB5DE8E2-AD1F-11d0-ACBE-5E86B1000000}' asGUID storeString "

    self iid: ( GUID fromBytes: #[ 16rE2 16rE8 16r5D 16rDB 16r1F 16rAD
        16rD016r11 16rAC 16rBE 16r5E 16r86 16rB1 0 0 0 ] ).
    self initialize.
```

Class initialization comprises the following steps:

1. Setting the class interface identifier with iid:.
2. Calling super initialize.

Create a COMDualInterfaceObject Subclass

Creating a COMDualInterfaceObject subclass involves implementing the following:

- Class initialization
- Methods and properties
- Class factory creation
- Type library management
- Run-Time installation

Subclassing the COMDualInterfaceObject class allows you to inherit default implementation for IDispatch and IUnknown. The example defines the AllDataTypesCOMObject subclass, as follows:

```
Object
   COMObject ('referenceCount' 'controllingUnknown' 'innerUnknown')
      COMAutomationObject ('iDispatch' 'valueAdaptor')
         COMDispatchObject ('publishedObject' 'registrationToken')
            COMDualInterfaceObject
               AllDataTypesCOMObject

COMDualInterfaceObject subclass: #AllDataTypesCOMObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: 'COMStatusCodeConstants '
   category: 'COM-Automation-Server Samples'
```

COMDualInterfaceObject Subclass Responsibilities

The COMDualInterfaceObject subclass must implement the following class methods:

ClassInitializer

Initialize the class when loaded.

newClassFactory

Answer a new class factory that creates instances of the receiver.

installRuntime

Install this class in the image for delivery.

getTypeLibraries

Answer a Collection of the type libraries used for the application.

These methods and their implementation are discussed next.

Implementing Methods and Properties

Implementing Automation Objects explains how to create the link between the incoming COM interface functions calls and your COMDualInterfaceObject subclass. Once the calls arrive in your COMDualInterfaceObject subclass, your application takes over. In the case of the AllDataTypes example, all non-COM-specific work is delegated to the AutomationAllDataTypes class. For example, the Reset method is implemented to call on the AutomationAllDataTypes method of the same name as follows:

```
Reset
   "Implement ISmalltalkCommanderDisp::Reset."
   self publishedObject Reset.
   ^S_OK
```

The return value of each method must be an HRESULT. If your code does not detect any error conditions, the answer should be the constant S_OK.

By convention, the names of the methods in a COMDualInterfaceObject subclass are identical to the names of the interface in the header file generated by the MIDL compiler.

When an object is called to get one of its properties, a value reference is passed as an argument for the object to fill in. The message value: is used to set the answer to a Smalltalk object.

For example, the example defines the get_BSTRValue: message, as follows:

```
get_BSTRValue: resultReference
   "Implement IAllDataTypesDisp::get_BSTRValue."
   resultReference value: ( self publishedObject getBSTRValue ).
   ^S_OK
```

When an object is called to set one of its properties, a Smalltalk object is passed as an argument for the object to fill save. For example, the example defines the put_BSTRValue: message, as follows:

```
put_BSTRValue: aValue
    "Implement IAllDataTypesDisp::setBSTRValue."
    self publishedObject setBSTRValue: aValue.
    ^S_OK
```

The keyword used for passing multiple arguments is the anonymous _: , as the next example illustrates.

```
ManyArguments: anIDispatch _: aPropertyName _: aLong _:
resultReference
    "Implement ISmalltalkCommanderDisp::ManyArguments."
    resultReference value: ( self publishedObject
        ManyArguments: anIDispatch
    propertyName: aPropertyName aLong: aLong ).
    ^S_OK
```

Implementing Class Initialization

When the class AllDataTypesCOMObject is loaded in the system it must be initialized with the CLSID and dual interface class for the object it represents. This is done in the ClassInitilizer method.

```
ClassInitializer
    "This method is run at COM Connect installation time."
    " self ClassInitializer "
    self initialize.

initialize
    "This method is run at COM Connect installation time."
    " self ClassInitializer "
    super initialize.
    self clsid: AutomationAllDataTypes clsid.
    self dualInterfaceClass: IAllDataTypesDisp.
```

Class initialization comprises the following steps:

1 Calling super initialize.

2 Setting the class identifier (CLSID) with clsid:. In the example, the definition from AutomationAllDataTypes class was reused.

3 Setting the dual interface class with dualInterfaceClass: to the custom dual interface class. The class is IAllDataTypesDisp in the example.

Providing Class Factory Support

A class factory is constructed by the newClassFactory method, as follows:

```
newClassFactory
   "Answer a new class factory that creates instances of the receiver."
   ^self newClassFactoryForClass: AutomationAllDataTypes
      clsid: self clsid
      specificationTable: AutomationAllDataTypes specificationTable
      typeLibraries: self typeLibraries
```

The newClassFactory method publishes the AutomationAllDataTypes class. The published class can be anywhere in the hierarchy. There are no requirements on the published class. The only requirements are on your COMDualInterfaceObject subclass presented in this section. For this example, the existing specification table method from the AutomationAllDataTypes class was not duplicated in the AllDataTypesCOMObject class.

In this example, the newClassFactoryForClass:clsid:specificationTable:typeLibraries: message answers a new class factory that creates instances of the receiver. This COMDualInterfaceObject subclass publishes a new instance of itself as a COM Automation object with the CLSID ‘self clsid’. The dispatch specifications for the properties and methods that are published for the subclass are defined by AutomationAllDataTypes specificationTable, which is indexed by DISPID. The type libraries for this object are specified by self typeLibraries, a dictionary of COMTypeLibrary objects indexed by locale IDs (LCIDs). The example has one type library for the English language.

Summary

The published COM class AllDataTypesCOMObject inherits IUnknown and IDispatch behavior from the COMObject framework. The AllDataTypesCOMObject class implements the interface for the additional members of the IAllDataTypesDisp dual interface. The AllDataTypesCOMObject class passes all non-grunt-COM work to its published Smalltalk class AutomationAllDataTypes, which can reside anywhere in the hierarchy.

Implementing Type Library Management

The getTypeLibraries method answers a collection of COMTypeLibrary objects. Have one type library for each language your application supports.

```
getTypeLibraries
    "Answer a Collection of the type libraries used for the application. The
locale ID must be specified for each COMTypeLibrary since the
framework uses this field as an index."
    | myTypeLibraries |
    myTypeLibraries := OrderedCollection new.
    myTypeLibraries
        add: self newTypeLibraryEnglish.
    ^myTypeLibraries

newTypeLibraryEnglish
    "Answer a type library for the English language for the application."
    ^COMTypeLibrary new
        libraryID: AutomationAllDataTypes typeLibraryID;
        lcid: COMTypeLibrary lcidEnglish;
        directoryName: COMSessionManager absoluteCOMDirectoryName,
            'Examples\COMAuto\AllDataT\TypeLibrary';
        fileName: 'VwAllDT.tlb';
        majorVersion: 1;
        minorVersion: 0
```

Implementing Run-Time Installation

When ready to make a deployment image, you must first send the installRuntime message to your class.

```
installRuntime
    " Prepare the receiver for deployment in a run-time image
    configuration.  You can extend this method and place installation code
     in it. "

    " self installRuntime "
    super installRuntime.
    "You can override the default for server application termination:"
    "self exitIfNotInUse: true."
    "You can also change the adaptor binding policy:"
    "self useAdaptorBinding: true."
```

Converting Existing Objects to Dual Interfaces

If you already implemented exposed objects that support only the IDispatch interface, it is recommended that you convert them to support dual interfaces. Do the following:

1 Edit the `.odl` or `.idl` file to declare a dual interface instead of an IDispatch-only interface.

2 Rearrange the parameter lists so that the methods and properties of your exposed objects return an HRESULT and pass their return values in a parameter.

12

Using Distributed COM

Microsoft's distributed COM (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers—on a LAN, a WAN, or the Internet. With DCOM, your application can be distributed at locations that make the most sense to your customer and to the application.

Because DCOM is an evolution of COM, you can take advantage of your existing investment in COM-based applications, components, tools, and knowledge to move into the world of distributed computing. As you do so, DCOM handles low-level details of network protocols so you can focus on your real business.

Locating a Remote Object

With the advent of COM for distributed systems, COM uses the basic model for object creation, and adds more than one way to locate an object that may reside on another system in a network, without overburdening the client application.

DCOM has added registry keys that permit a server to register the name of the machine on which it resides, or the machine where an existing storage is located. Thus, client applications, as before, need know only the CLSID of the server.

However, for cases where it is desired, DCOM lets you use a structure called COSERVERINFO through a `serverName:` argument to the IClassFactory object creation services, which allows a client to specify the location of a server. Another important value is the class context (CLSCTX), which specifies whether the expected object is to be run in-process, out-of-process local, or out-of-process remote. Taken together, these two values and the values in the registry determine how and where the object is to be run. Instance creation calls, when they specify a server location, can override a registry setting. The algorithm COM uses for doing this is described in the description of the CLSCTX enumeration in Using Automation Objects.

The client and server machines must both be members of domains with a trust relationship for all types of remote activation.

Accessing Objects on Remote Machines

While it is possible to monkey with the Registry Database or use DCOMCnfg.exe and turn a local server into a remote server, you can also programmatically specify that you want to access a remote server. For an example of using IClassFactory to create a remote component, see Optimizing Querying Interfaces.

The following is an example of using createObject:onServer: to create a remote component:

```
aDispatchDriver := COMDispatchDriver
   createObject: 'MyApp.SomeObjectClass'
   serverName: 'MyRemoteServerName'.
```

The serverName: argument is used to construct a DCOM COSERVERINFO structure and describes the machine on which to instantiate the object. This argument might be nil, in which case the object is instantiated on the current machine or at the machine specified in the registry under the class's RemoteServerName named-value, according to the interpretation of the context flags parameter. See the CLSCTX documentation for details.

The Remote Server Name Key

The server name is used to identify a remote system in object creation functions. Machine resources are named using the naming scheme of the network transport. By default, all UNC (\\server or server) and DNS names (server.com, www.foo.com, or 135.5.33.19) names are allowed.

A server can install the RemoteServerName named-value on client machines to configure the client to request that the object be run at a particular machine whenever an activation function is called, for which a server name is not specified. A RemoteServerName Registry is defined as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID\{AppID_valu}
   \RemoteServerName = server_name
```

As described in the section on CLSCTX enumeration and the COSERVERINFO structure, one of the parameters of distributed COM activation is a pointer to a COSERVERINFO structure. When this value is not nil, the information in the COSERVERINFO structure overrides the setting of the RemoteServerName key for the function call.

RemoteServerName allows simple configuration management of client applications - they might be written without hard-coded server names, and can be configured by modifying the RemoteServerName registry values of the classes of objects they use.

You can specify a context for object creation with class messages from IClassFactory and COMDispatchDriver. The context: keyword is used to denote the class context parameter.

In Depth: The COSERVERINFO Structure

The object creation messages currently presented by COMDispatchDriver and IClassFactory let you set only the server name attribute of the DCOM COSERVERINFO structure. This assumes that you are using the default NTLMSSP security package, for which the pAuthInfo parameter is set to zero.

The COSERVERINFO structure identifies a remote machine resource to the new or enhanced activation functions. The structure is defined in the Wtypes.h header file, as follows:

```
typedef struct  _COSERVERINFO
   {
   DWORD dwReserved1;
   LPWSTR pwszName;
   COAUTHINFO  *pAuthInfo;
“   DWORD dwReserved2;”   }   COSERVERINFO;
```

When using NTLMSSP, the pAuthInfo value must be set to zero. A non-zero value, which is a pointer to a COAUTHINFO structure, is used only when a security package other than NTLMSSP is being used.

If you are a vendor supporting another security package, refer to COAUTHINFO documentation from Microsoft. The mechanism described there is intended to allow DCOM activations to work correctly with security providers other than NTLMSSP, or to specify additional security information used during remote activations, for interoperability with alternate implementations of DCOM. If pAuthInfo is set, those values are used to specify the authentication settings for the remote call. These settings are passed to RpcBindingSetAuthInfoEx.

If the pAuthInfo field is not specified, any values in the AppID section of the registry are used to override the following default authentication settings:

```
dwAuthnSvc RPC_C_AUTHN_WINNT
dwAuthzSvc RPC_C_AUTHZ_NONE
pszServerPrincName NULL
dwAuthnLevel RPC_C_AUTHN_LEVEL_CONNECT
dwImpersonationLevel RPC_C_IMP_LEVEL_IMPERSONATE
pvAuthIdentityData NULL
dwCapabilities RPC_C_QOS_CAPABILITIES_DEFAULT
```

Optimizing Querying Interfaces

To in-process components, queryInterface: calls are very fast. To components in local servers, the queryInterface: is still pretty fast. But when you need to move across a network, the overhead of calling a function increases greatly. It is not inconceivable for an application to grind to a halt as it repeatedly makes any function calls, including queryInterface: calls. Therefore, to reduce the impact of calling queryInterface:, DCOM has created a new structure named MULTI_QI. The MULTI_QI structure allows you to query for several interfaces at the same time. This can save considerable overhead. In Smalltalk, you can create a component with IClassFactory and request multiple interfaces at the same time:

```
requestedInterfaceIIDs := Array with: IID_IDispatch with: IID_IUnknown.
interfaces := IClassFactory
    createInstance: aCLSIDOrProgID
    iids: requestedInterfaceIIDs
    controllingUnknown: nil
    context: CLSCTX_SERVER
```

```
    serverName: serverName.
"Work with the interfaces
…
and release them when you are done."
interfaces do: [ : anInterface | anInterface release ].
```

This example requests two interface pointers by passing the desired IIDs in an Array. If the call succeeds, the answer is an array of interfaces. In this example, instances of IDispatch and IUnknown.

When the controllingUnknown: parameter is non-nil, this indicates the instance is being created as part of an aggregate, and the parameter is to be used as the new instance's controlling IUnknown. Aggregation is currently (in Windows NT 4.0) not supported cross-process or cross-machine. When instantiating an object out of process, CLASS_E_NOAGGREGATION is returned if this parameter is non-nil.

The COMDispatchDriver methods with a serverName: parameter use this IClassFactory facility. For further information, consult the Microsoft documentation for the CoCreateInstan-ceEx API.

Determining Whether DCOM Is Available

To determine whether DCOM services are enabled, first check to see whether OLE32.DLL supports free threading:

```
COMSessionManager isFreeThreadingAvailable.
```

The isFreeThreading method checks if the OLE32.DLL library implements the CoInitializeEx API.

After determining whether your system supports free threading, check to see whether DCOM is enabled:

```
COMSessionManager isDCOMEnabled.
```

The isDCOMEnabled method checks if the Registration Database has the value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole\EnableDCOM
```

set to **Y** or **y**.

You can check for both with the isDCOMAvailable method:

```
COMSessionManager isDCOMAvailable.
```

Making VisualWorks COM Server a Windows NT 4.0 Service

This section describes making a VisualWorks COM server image a Windows NT 4.0 service, using as an example, the COM example class VisualWorks.SmalltalkCommander.

System Requirements

The Windows NT Resource Kit provides two utilities that allow you to create a Windows NT user-defined service for an application (executable) or batch file. This section uses two Windows NT Resource Kit utilities, SRVANY.EXE and INSTSRV.EXE, to set up SmalltalkCommander as a Windows NT service. In this section you are instructed to use the REGEDT32.EXE Registry Editor. Using the Registry Editor incorrectly can cause serious, system-wide problems that might require you to reinstall Windows NT to correct them. Cincom or Microsoft cannot guarantee that any problems resulting from the use of Registry Editor can be solved. Use this tool at your own risk.

Before you start, make sure that your VisualWorks image is properly configured as a deployment image.

Configuration Procedure

To configure SmalltalkCommander as a Windows NT service, follow these step-by-step instructions:

1. Copy SRVANY.EXE and INSTSRV.EXE to your system32 directory.
2. Select a name for your service, VWNTOE for instance (you can give it a different name later), and install it as a new service using the following command:

   ```
   INSTSRV VWNTOE c:\tools\srvany.exe
   ```

3. When the new service is created, you see a notice with instructions.

   Note: VWNTOE is the name chosen for this new service; any unused service name suffices.

4. Go to the Control Panel and open the **Services** utility.
5. Select your new service, in this case VWNTOE, and click the **Startup** button.

6 Create a domain account, like MYDOMAIN\Smalltalk, for Smalltalk to use. Configure the VWNTOE service to start up using the MYDOMAIN\Smalltalk login (and appropriate password), and to start either automatically or manually.

   To have your service start every time you boot your machine, select **Automatic** for Startup Type in the Service dialog box. Otherwise, select **Manual** or **Disabled**. If you select **Manual**, you must go into the Services utility, select the service, and click the **Start** or **Stop** button every time you want to start or stop the service.

7 Run REGEDT32.EXE (not regedit.exe).

8 Create a Parameters key under:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
Services\VWNTOE
```

9 Under this key, create an Application value of type REG_SZ and specify the full path of the VisualWorks object engine executable.

10 Create values to specify the command line parameters and the default directory; for example:

```
Application: REG_SZ: C:\vw30\Bin\Vwnt.exe
AppDirectory: REG_SZ: C:\vw30\Bin
AppParameters: REG_SZ:
C:\vw30\Com\Examples\ComAuto\Vwcomsrv.im /Automation
```

   This tells SRVANY what application it should start when SRVANY itself is started as the VWNTOE service.

11 It is useful to also establish a dependency to make sure the RPCSS service is started before the system attempts to start the VisualWorks server. Under the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VWNTOE key, create a DependOnService value of type REG_MULTI_SZ, and give it the value RPCSS, as follows:

```
DependOnService: REG_MULTI_SZ: RPCSS
```

12 At this time, take the opportunity to modify the default display name:

```
DisplayName: REG_SZ: VisualWorks NT Object Engine
```

13 SRVANY is now configured to start up VisualWorks as a service; however, once you have read and understood the contents of the "COM Servers Activation and NT Windows Stations" article (see references at the end of this section), it is necessary to set up the VisualWorks.SmalltalkCommander class to launch as the

MYDOMAIN\Smalltalk user. For this, you need to use the DCOMCNFG.EXE utility included with Windows NT 4.0.

However, for VisualWorks.SmalltalkCommander to be recognized by DCOMCNFG as a LOCAL server (so that you can modify the Run As setting), you must temporarily strip off any command line parameters that got registered in the LocalServer32 value. This appears to be a bug in DCOMCNFG, which somehow gets confused by anything other than <pathname>\<filename>.exe in LocalServer32.

Thus, the steps are:

a Strip off the parameters using REGEDT32.

b Run DCOMCNFG and set up SmalltalkCommander to Run As MYDOMAIN\Smalltalk (with password).

c Restore command line parameters in LocalServer32.

After doing all this, you can reboot your machine, and if you set up the service (it now appears as "VisualWorks NT Object Engine" in the **Services** applet) to be automatic, you have a running instance of VisualWorks that is ready to handle COM and DCOM requests from any user that has the privilege to access the VisualWorks.SmalltalkCommander class.

One last note. When DCOMCNFG was used to set up the SmalltalkCommander class to run as MYDOMAIN\Smalltalk, it created a **RunAs** entry in the registry under the **AppID** key referenced by the SmalltalkCommander's CLSID. To save lots of effort, it is advisable that all classes implemented within your VisualWorks application use the SAME AppID, unless you explicitly want to start a separate instance of the VisualWorks Object Engine executable for some subset(s) of classes (perhaps for security purposes). If so, you could set up the other AppIDs to RunAs different users, with different sets of privileges.

Reference Material

The following Microsoft Knowledge Base articles are very insightful:

- "COM Servers Activation and NT Windows Stations"
- "HOWTO: Run Automation Manager as a Windows NT Service" :
- "How To Create A User-Defined Service"

13

Automation Controller Framework

When writing Automation controller programs, you might find yourself writing the same kind of code over and over again. When you work with a COMDispatchDriver, you must know the name of each object's methods and properties. You must also take care of parameters, named or not, optional and required, as well as some possible data type conversion issues.

It turns out that some applications also define the same kind of standard ActiveX objects like collections, documents and fonts. The Automation Controller framework provides Smalltalk wrappers for these standard Automation objects, as well as abstract classes you can subclass to add support for other Automation objects. You can define controllers to provide a Smalltalk front-end to Automation objects. Clients of a controller use a Smalltalk class without dealing with the details of the Automation object and its COMDispatchDriver.

Examples

The MS Excel Monster Damage Example

This Excel Automation example illustrates using the Automation Controller Framework. This example requires Microsoft Excel 97 or Microsoft Excel for Windows 95, version 7.0 (from Office 95). This sample contains the ExcelExampleMonsterDamage class, which performs the following operations:

1. Start Excel.
2. Fill a spreadsheet with some numbers.
3. Create and format a chart for those numbers.

4 Save the work to a file

5 Quit Excel.

Sample expressions to run this sample can be found in the ExcelExampleMonsterDamage class comment:

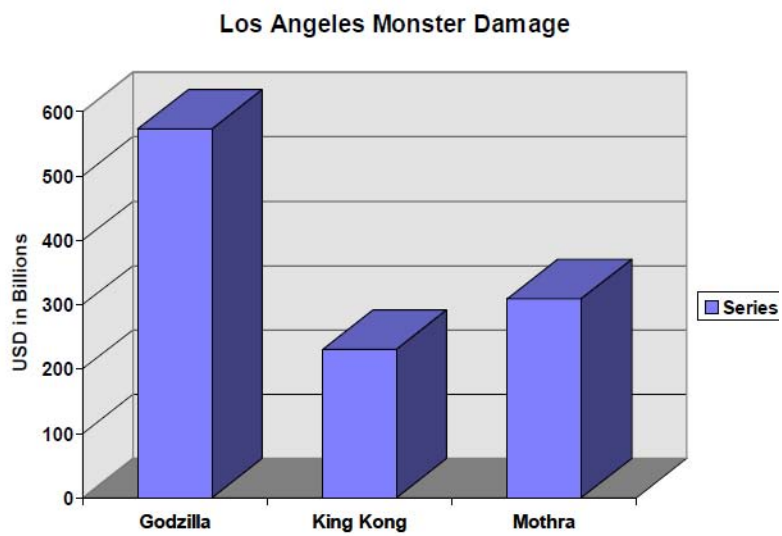
```
"The following examples will create its Monster.xls output file in theCOM\Examples\COMAuto\Output directory."
   "Run and save the report in a background invisible Excel process."
   ExcelExampleMonsterDamage runInvisible.

   "Make Excel visible to run and chart the report in the foreground."
   ExcelExampleMonsterDamage runVisible.

   "Run, save and print the report in a background invisible Excel
    process."
   ExcelExampleMonsterDamage runInvisibleAndPrint.
```

The resulting Monster.xls spreadsheet is shown in following figure:

Result of the Excel Monster Damage Sample



The MS Excel Import Text File Example

This Excel Automation example illustrates using the Automation Controller Framework. This example requires Microsoft Excel 97 or Microsoft Excel for Windows 95, version 7.0 (from Office 95). This sample uses the ExcelExampleFileImport class, which performs the following operations:

1 Start Excel.
2 Import a text file of fixed width column format (COM\Examples\COMAuto\Input\Stocks.txt). This text file was obtained by saving a CompuServe stock quote window to a text file.
3 Format the spreadsheet.
4 Save the work to a file.
5 Print the file.
6 Read the spreadsheet into a Smalltalk Collection from Excel and display it in a text widow.
7 Quit Excel.

Sample expressions to run this sample can be found in the ExcelExampleFileImport class comment:

```
"Run and save the report in a background invisible Excel process."
ExcelExampleFileImport runInvisible.

"Make Excel visible to run and chart the report in the foreground."
ExcelExampleFileImport runVisible.

"Run, save and print the report in a background invisible Excel
process."
ExcelExampleFileImport runInvisibleAndPrint.
```

The MS Word Class Formatter Example

The WordExampleClassFormatter class is an example of using a non-standard Automation object: the ‘Word.Basic’ object from Microsoft Word for Windows 95 version 7.0, which is required for this example. This example takes a class and creates a word document containing a description of the class including tables of class, class instance and instance variables; class method and instance method categories.

Sample expressions to run this sample can be found in the WordExampleClassFormatter class comment:

```
"Show Word running the formatting but do not save."
WordExampleClassFormatter showClass: WordExampleClassFormatter.

"Run the formatter in an invisible background Word process.
The formatted document is saved in the directory
COM\Examples\COMAuto\Output with the class name as a file name."

WordExampleClassFormatter saveClass: WordExampleClassFormatter.
```

```
"Run the formatter and print the document in an invisible backgroung
Word process. Do not save."
WordExampleClassFormatter printClass: WordExampleClassFormatter.
```

Documentation on Word 7 (Office 95) objects, methods and properties can be found in the file **Wrdbasic.hlp**. This Microsoft Word 7 file can be found on the Microsoft CD, it is not normally copied during installation.

Creating New Controller Classes

To create a new controller class for an Automation object you must follow this recipe:

1. Subclass the right framework class
2. Define a class message to match your controller to an Automation class.
3. Implement methods to wrap the Automation object's methods and properties.
4. In general you should create a controller class to provide a customized wrapper for each automation object supported by the application that you want to use in your program.

Next are some examples with the Microsoft Excel 7 objects: the Application object, the Workbooks Collection object, and the Range object.

Subclassing a Framework Class

For the Excel Application object, a subclass of COMAutomation-ApplicationController was created called ExcelApplicationController. For the Workbooks collection, a subclass of COMAutomationCollectionController was first created for all of the Excel collection objects, called ExcelCollectionController. Under ExcelCollectionController a subclass called ExcelWorkbooksController was created. For the Range object, a subclass of COMAutomationObjectController was created for all of the Excel objects. Under COMAutomationObjectController a subclass called ExcelRangeController was created.

The Excel sample controllers included are:

```
Object
  COMAutomationController
    COMAutomationObjectController
      ExcelObjectController
        ExcelChartAreaController
        ExcelChartController
        ExcelRangeController
        ExcelWorkbookController
        ExcelWorksheetController
      COMAutomationApplicationController
        ExcelApplicationController
      COMAutomationCollectionController
        ExcelCollectionController
          ExcelChartsController
          ExcelSheetsController
          ExcelWorkbooksController
        COMAutomationDocumentsController
      COMAutomationDocumentController
      COMAutomationFontController
```

A subclass of COMAutomationController should re-implement one of the following class methods in order for the instance creation method to access Automation objects:

clsid

Answer the receiver's CLSID (a GUID). If not re-implemented by a subclass, the #progID is used to find the CLSID. If #progID is not re-implemented, the #versionIndependentProgID is used and must then be re-implemented by the subclass.

progId

Answer the receiver's ProgID. This can be re-implemented by concrete subclasses.

versionIndependentProgID

Answer the receiver's Version Independent ProgID. This can be re-implemented by concrete subclasses if #progID is not.

Even though you can access an application from any of its objects, in general, it is recommended that you start from the program's Application object. This means you can implement these messages in any concrete subclass of COMAutomationController, it is recommended that you do so for your application objects as subclasses of COMAutomationApplicationController. See the next section of this document for an example.

The COMAutomationController defines methods to ease access to its dispatch driver by providing the following methods:

```
getProperty:
invokeMethod:
invokeMethod:with:
invokeMethod:with:with:
invokeMethod:with:with:with:
invokeMethod:withArguments:
invokeProcedure:
invokeProcedure:with:
invokeProcedure:with:with:
invokeProcedure:with:with:with:
invokeProcedure:withArguments:
setProperty:value
```

These methods are not documented here, because they are covered in the COMDispatchDriver reference. Other methods of interest are:

dispatchDriver

Answer the receiver's dispatch driver.

releaseResources

Release any resources associated with receiver. This method is called by release just before the dispatch driver is released. It can be reimplemented by a subclass. Default implementation does nothing.

isMethod: *aMethodName*

Answer if there is a method named *aMethodName*.

isProperty: *aPropertyName*

Answer if there is a property named *aPropertyName*.

isValid

Answer true if the receiver is valid for use, false otherwise.

All other classes in this framework are derived from COMAutomationController.

Rules for Adding an Application Object

By convention, the Application object is the "root" object of the application and provides access to its basic capabilities and major domain objects.

To add support for an Application object, create a subclass of COMAutomationApplicationController. In general, it is recommended that you access an application's objects through its Application object, even though you can start an application from any of its objects. To identify the Application object, define one of the following class methods:

- clsid
- progId
- versionIndependentProgId

For example, the following ExcelApplicationController class method is defined:

```
versionIndependentProgID
    "Answer the ProgID describing the receiver in the registration
    database."
    ^'Excel.Application'
```

In addition, a subclass of COMAutomationApplicationController must define a class method called controllerClasses to answer all of the controller classes in the application's domain. The ExcelApplicationController class defines the following class methods:

```
controllerClasses
    "Answer the collection of classes that work with this application
    controller."
    ^( Array with: self ),
        ExcelObjectController allSubclasses,
        ExcelCollectionController allSubclasses
```

The controllerClasses method permits the framework to create a mapping between the name of COM objects (like Application) and a controller class (like Excel97ApplicationController). This mapping is used by the controllerFor: method to answer a controller class from a dispatch driver argument.

```
ClassInitializer
    "This method should be called on application installation."
    self initialize.
```

Implementing the controllerClasses methods permits the controllerFor: method to work. This mechanism allows a dispatch driver returned from a method or property invocation to be matched to its controller class.

Rules for Adding a Standard Object

To add support for a standard object, create a subclass of COMAutomationObjectController. Standard objects are defined in and always define two properties that help you navigate an application's objects: parent and application.

All subclasses of COMAutomationObjectController should answer appropriately to the class messages:

applicationClass

Answer the Application controller class for this class. This supports the application and the controllerFor: methods.

automationClassName

Answer the Automation name of the receiver. This is the #name of the COMDispatchDriver.

For example, in abstract ExcelControllerObject abstract class is defined:

```
applicationClass
   "Answer the Application controller class for this class."
   ^ExcelApplicationController
```

Each of the Excel controller classes define an automationClassName method, for example in ExcelRangeObject:

```
automationClassName
   "Answer the name of the receiver."
   ^'Range'
```

The COMAutomationObjectController class defines the controllerFor: controller utility method as:

```
controllerFor: aValue
   "If <aValue> is a dispatch driver then wrap <aValue> with a new
   controller and answer the new controller. Otherwise answer <aValue>.
   If no class is registered for a dispatch driver, answer the dispatch
   driver (aValue)."

   aValue class == self class dispatchDriverClassifTrue:
      [ ^self class newControllerObject: aValue ].^aValue
```

This method knows what controller to use for an application's dispatch driver, because application controller classes register which classes are part of their domain at startup.

Adding Behavior to a Standard Object

When a controller class is defined in the framework, you can define instance methods to access properties and invoke methods. The messages for doing so are defined in the root controller class COMAutomationController. For a set property example, the ExcelRangeController defines the value: method as:

```
value: anObject
   "Set the Value property to <anObject>."
   ^self setProperty: 'Value' value: anObject
```

For a get property example, the ExcelWorkbookController defines the name method as:

```
name
   "Answer a String for the Name property."
   ^self getProperty: 'Name'
```

For a method invocation example, the ExcelWorksheetController defines the getRange: method as:

```
getRange: aRangeString
   "Answer a controller for the Range property defined by the range of
   cells <aRangeString>."
   ^self controllerFor: ( self invokeMethod: 'Range' with: aRangeString )
```

You might choose to hard-code a controller class to use with a particular method for one of two reasons:

First, you might want to associate a stock framework controller to an answer. Exactly that is done in the ExcelChartArea font method:

```
font
   "Answer the Font property."

   ^COMAutomationFontController on: ( self getProperty: 'Font' )
```

Second, you do not or cannot rely on the run-time use of controllerFor:. For example, we define the ExcelApplicationController charts method as:

```
charts
   "Answer the Charts collection Controller."
   "Implementation note: This method refers to the class name
   ExcelChartsController directly instead of using the #controllerFor:
   service since it seems Excel 7 uses the same class name. You could
   add behavior to a charts collection vs. Worksheets for a fancier
   sample."
   "some caching added for this sample."
```

```
^self cachedControllers at: 'Charts'
   ifAbsentPut: [ ExcelChartsController on: ( self invokeMethod:
   'Charts' ) ].
```

Adding a Collection Controller

To add support for a standard collection object, create a subclass of COMAutomationCollectionController. Standard collection objects are defined under Automation Controller Framework.

Just like other controller classes, you must define the applicationClass and automationClassName methods. In the ExcelCollectionObjects abstract class is defined:

```
applicationClass
   "Answer the Application controller class for this class."
   ^ExcelApplicationController
```

Concrete subclasses of ExcelCollectionObjects define automationClassName; for example, ExcelSheetsController defines:

```
automationClassName
   "Answer the name of the receiver."
   ^'Sheets'
```

The collection controller has an additional level of control that lets you define which controller class to use for each item in the collection (if the items are represented by a dispatch interface). The COMAutomationCollectionController class method itemControllerClass might be re-implemented to answer the controller class to be used for each item in the collection. If not re-implemented by a subclass, the controllerFor: service is used. For example, the ExcelChartsController defines the itemControllerClass class method as:

```
itemControllerClass
   "Answer the Controller class to be used for each item in the
   collection."
   ^ExcelChartController
```

For the Excel Chart Collection, the item: and add methods answer a controller on a chart object.

Releasing Return Values

You are responsible for releasing any resources returned from a method invocation or a get property call. The only resources that you need be concerned with are marked VT_DISPATCH or VT_UNKNOWN. A VT_DISPATCH return value gets mapped to an IDispatch, a COMDispatchDriver, or an Automation Controller class.

A VT_UNKNOWN return value gets mapped to an IUnknown. All can be released with a release message. Any memory used by a VT_BSTR value is handled automatically.

It is possible for Automation objects to return entire Arrays of objects (VT_ARRAY combined with another type), in which case you should be careful to release the contents of the array, if necessary. For example, the ExcelApplicationController class defines its setRange:to: method as:

```
setRange: aRangeString to: anObject
   "Set the range defined by <aRangeString> to <anObject>."

   | range |
   "Get a range controller for <aRangeString>."
   range := self getRange: aRangeString.

   "Set the range to <aValue> and release the range."
   [ range value: anObject ]
      ensure: [ range release ]
```

To release an Array of interfaces, you can use the COMSafeArray utility releaseInterfacesIn: method.

Using Non-Standard Objects

Some Automation objects and applications do not organize their objects according to the Microsoft guidelines presented in this section. For example, Microsoft Word for Windows 95 version 7.0 does not have any standard objects. In fact, it is a monolithic application; it has only one object, Word.Basic, which contains hundreds of procedures. Since it is an Automation application nonetheless, it can still be controlled through a dispatch driver.

To support a nonstandard object, you can subclass COMAutomationController and add Smalltalk behavior to your subclass. For example, the Word 95 sample class WordBasicController is a subclass of COMAutomationController.

Using the Default Specification Policy

Specification policies and specification tables are described in Using Automation Objects.

While a default specification policy for each automation controller class is tracked to account for non-standard classes, you can control an application through only one specification policy set on the application's Application class. The default specification policy is used automatically when new controllers are created. The default is set to the Variant policy on installation.

Setting the Default Specification Policy

The default policy is set to the Variant policy but can be queried and reset with the COMAutomationController defaultSpecificationPolicy and defaultSpecificationPolicy: class messages. This message must be sent to your application's class, not to COMAutomationController. For example, you can query for the default Specification Policy with the expression:

```
ExcelApplicationController defaultSpecificationPolicy.
```

You can set the default Specification Policy with the expression:

```
ExcelApplicationController defaultSpecificationPolicy: <aSymbol>.
```

Where aSymbol is one of the following:

- #newCompletePolicy
- #newTypeCompilerPolicy
- #newTypeLibraryPolicy
- #newVariantPolicy

If you choose the #newCompletePolicy, each controller class for your application must define a literalSpecification class method.

14

Standard Automation Objects and Naming Guidelines

This chapter describes the standard ActiveX objects defined by Microsoft, and discusses naming guidelines for creating and using objects that are unique to applications, especially user-interactive applications that support a multiple-document interface (MDI). If an ActiveX object is not user-interactive or supports only a single-document interface (SDI), the standards and guidelines should be adapted as appropriate.

Standard objects comprise a set of objects defined by Automation. You can use them as appropriate to your application. The objects described in this chapter are oriented toward document-based, user-interactive applications. Other applications (such as non-interactive database servers) may have different requirements.

Naming guidelines are recommendations meant to improve consistency across applications.

The standards and guidelines are subject to change according to Microsoft.

The classes the COM Connect Automation Controller framework defines are:

```
Object
   COMAutomationController
      COMAutomationObjectController
         COMAutomationApplicationController
         COMAutomationCollectionController
            COMAutomationDocumentsController
         COMAutomationDocumentController
         COMAutomationFontController
```

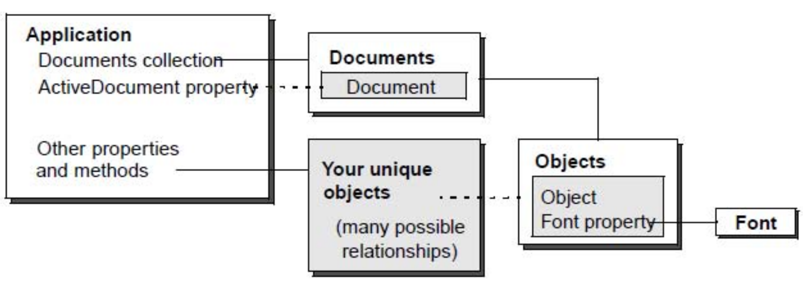
Using Standard Objects

The following table lists the Automation standard objects. Although none of these objects are required, user-interactive applications with subordinate objects should include an Application object.

Class name Automation object name	Description
COMAutomationApplicationController Application	Top-level object. Provides a standard way for ActiveX clients to retrieve and navigate an application's subordinate objects.
COMAutomationDocumentController Document	Provides a way to open, print, change, and save an application document.
COMAutomationDocumentsController Documents	Provides a way to iterate over and select open documents in MDI applications.
COMAutomationFontController Font	Describes fonts that are used to display or print text.

The following figure shows how the standard objects fit into the organization of objects provided by an application.

An organization of objects in an application



The following sections describe the standard properties and methods for all objects, all collection objects, and each of the standard objects. These sections list the standard methods and properties for each object, as well as the standard arguments for those properties and methods. You can define additional application-specific properties and methods for each object. You can also provide additional optional arguments for any of the listed properties or methods; however, the optional arguments should follow the standard arguments in a positional argument list.

Accessing Objects

The abstract root class COMAutomationController defines the following instance creation methods:

new

Answer a new instance of the receiver on a new Automation object defined by the receiver's identity (CLSID, ProgID or Version Independent ProgID).

on: *aDispatchDriver*

Answer a new instance of the receiver initialized with *aDispatchDriver*.

onActiveObject

Answer a new instance of the receiver on the active Automation object defined by the receiver's identity (CLSID, ProgID or Version Independent ProgID).

serverName: *aServerName*

Answer a new instance of the receiver on a new Automation object on *aServerName*. If *aServerName* is nil, the object will be created on the same machine as the receiver.

pathName: *aFileName*

Answer a new instance of the receiver on the Automation object in *aFileName*.

You must make sure to send the release message to any controller you create.

Method and Property Names

Automation names are not case sensitive. The name of an Automation method or property is usually given in upper case. The Smalltalk wrapper methods in the controller classes for Automation methods and properties always start with a lower case letter, as usual in Smalltalk. The first column of the tables in this section contains the method and property names.

When an Automation object defines a property called MyName for example, the controller class will define a set method called myName: and a get method myName.

If an Automation object defines a boolean property called Visible for example, the controller class will define a set method called isVisible: and a get method isVisible. If the property is read-only, there will be only a get method.

Standard Object Properties

All objects, including the Application object and collection objects are subclasses of the COMAutomationObjectController abstract class.

All objects, including the Application object and collection objects, must provide the properties listed in the following table:

Property method name	Answer class Automation type	Description
application	COMAutomationApplication-Controller VT_DISPATCH	Returns the Application object; read only.
parent	COMAutomationApplication-Controller VT_DISPATCH	Returns the creator of the object; read only.

The Application and Parent properties of the Application object return the Application object.

Collection Object Properties

Collection objects are wrapped by the `COMAutomationCollectionController` class.

A collection provides a set of objects over which iteration can be performed. All collection objects must provide the properties listed in the following table:

Property method name	Answer class Automation type	Description
count	IntegerVT_I4	Returns the number of items in the collection; read only. Required.

Collection Methods

Methods for collections are described in following table. The item: method is required; other methods are optional.

Method name	Answer class Automation type	Description
add	A Controller class or undefinedVT_DISPATCH or VT_EMPTY	Adds an item to a collection. Returns a COMDispatchDriver (VT_DISPATCH) if the object is created (object cannot exist outside the collection) or VT_EMPTY if no object is created (object can exist outside the collection).
item:	A Controller class. Varies with type of collection	Returns the indicated item in the collection. Required. The Item method may take one or more arguments to indicate the element within the collection to return. This method is the default member (DISPID_VALUE) for the collection object.
remove:	undefinedVT_EMPTY	Removes an item from a collection. Uses indexing arguments in the same way as the Item method.

The class COMAutomationCollectionController also defines utility methods like at:, do: and size to provide a more Smalltalk-like protocol.

All collection objects must provide at least one form of indexing through the Item method. The dispatch identifier (DISPID) of the Item method is DISPID_VALUE. It can be used in the following form:

```
thirdObject = myWords at: 3.
```

The Item method takes one or more arguments to indicate the index. Indexes can be numbers, strings, or other types. For example:

```
dogObject = myWords at: 'dog'.
```

Kinds of Collections

The standard for collections lets you describe two kinds of collections, depending on whether it makes sense for the collected objects to exist outside the collection.

In some cases, it is not logical for an object to exist independently of its collection. For example, an application's Documents collection contains all Document objects currently open. Opening a document means adding it to the collection, and closing the document means removing it from the collection. All open documents are part of the collection. The application cannot have open documents that are not part of the collection. The relationship between the collection and the members of the collection can be shown in the following ways:

The Documents collection add method creates an object (an open document) and adds it to the collection. Because an object is created, a reference to it is returned.

```
myDocController := myDocuments add.
```

The Document close method removes an object from the collection.

```
someDocController := myDocuments at: 3.
someDocController close.
someDocController release.
```

In other cases, it is logical for the objects to exist outside the collection. For example, a Mail application might have Name objects, and many collections of these Name objects. Each Name object would have a user's e-mail name, full name, and possibly other information. The e-mail name and full name would likely be properties named EmailName and FullName.

Additionally, the application might have the following collections of Name objects.

- A collection for the "To" list on each piece of e-mail.
- A collection of the names of the people to whom a user has sent e-mail.

The collections of Name objects could be indexed by using either EmailName or FullName.

For these collections, the add method does not create an object because the object already exists. Therefore, the add method should take an object as an argument, and should not return a value.

Assuming the existence of two collections (AddressBook and ToList), a user might execute the following code to add a Name object to the ToList collection:

```
aCollection := myAddressBook names.
aNameRef := aCollection at: 'John Smith'.
aToList := aMessage toList.
aToList add: aNameRef.
```

The Name object already exists and is contained in the AddressBook collection. The first line of code obtains a reference to the Name object for "John Smith" and points to aNameRef. The second line of code adds a reference to the object to the toList collection. No new object is created, so no reference is returned from the add method.

Unlike the relationship between Documents and Document, there is no way for the collected object (the Name) to know how to remove itself from the collections in which it is contained.

To remove an item from a collection, use the Remove method, as follows:

```
aToList := aMessage toList.
aToList remove: 'John Smith'.
```

This line of code removes the Name object that has the FullName "John Smith." The "John Smith" object might exist in other collections, but they are unaffected.

Using the Application Object in a Type Library

Application objects are wrapped by the COMAutomationApplicationController class.

If you use a type library, the Application object should be the object that has the appobject attribute. Because some ActiveX clients use the type information to allow unqualified access to the Application object’s members, it is important to avoid overloading the Application object with too many members.

The Application object should have the properties listed in the following table. The Application, FullName, Name, Parent, and Visible properties are required; other properties are optional.

Property method name	Answer class Automation type	Description
activeDocument	COMAutomationDocumentController VT_DISPATCH, VT_EMPTY	Returns the active document object or VT_EMPTY if none; read only.
application	COMAutomationApplicationController VT_DISPATCH	Returns the Application object; read only. Required.
caption caption:	StringVT_BSTR	Sets or returns the title of the application window; read/write. Setting the Caption to VT_EMPTY returns control to the application.
defaultFilePath defaultFilePath:	StringVT_BSTR	Sets or returns the default path specification used by the application for opening files; read/write.
documents	COMAutomationDocumentController VT_DISPATCH	Returns a collection object for the open documents; read only.
fullName	StringVT_BSTR	Returns the file specification for the application, including path; read only. For example, C:\Drawdir\Scribble. Required.
height height:	FloatVT_R4	Sets or returns the distance between the top and bottom edge of the main application window; read/write.
isInteractiveisInteractive:	BooleanVT_BOOL	Sets or returns True if the application accepts actions from the user, otherwise False; read/write.
leftleft:	FloatVT_R4	Sets or returns the distance between the left edge of the physical screen and the main application window; read/write.
name	StringVT_BSTR	Returns the name of the application, such as "Microsoft Excel"; read only. The Name property is the default member (DISPID_VALUE) for the Application object. Required.
parent	COMAutomationApplicationController VT_DISPATCH	Returns the Application object; read only. Required.

Property method name	Answer class Automation type	Description
path	StringVT_BSTR	Returns the path specification for the application's executable file; read only. For example, C:\Drawdir if the .exe file is C:\Drawdir\Scribble.exe.
statusBarstatusBar:	StringVT_BSTR	Sets or returns the text displayed in the status bar; read/write.
top top:	FloatVT_R4	Sets or returns the distance between the top edge of the physical screen and main application window; read/write.
isVisibleisVisible:	BooleanVT_BOOL	Sets or returns whether the application is visible to the user; read/write. The default is False when the application is started with the /Automation command-line switch. Required.
width width:	FloatVT_R4	Sets or returns the distance between the left and right edges of the main application window; read/write.

The Application object should have the methods listed in the following table. The quit method is required; other methods are optional.

Application object methods

Method name	Answer class Automation type	Description
help	undefinedVT_EMPTY	Displays online Help. May take three optional arguments: helpfile (String, VT_BSTR), helpcontextID (Integer, VT_I4), and helpstring (String, VT_BSTR). The helpfile argument specifies the Help file to display; if omitted, the main Help file for the application is displayed. The helpcontextID and helpstring arguments specify a Help context to display; only one of them can be supplied. If both are omitted, the default Help topic is displayed.
quit	undefinedVT_EMPTY	Exits the application and closes all open documents. Required.
repeat	undefinedVT_EMPTY	Repeats the previous action in the user interface.
undo	undefinedVT_EMPTY	Reverses the previous action in the user interface.

Document Object Properties

Document objects are wrapped by the COMAutomationDocumentController class.

If the application is document based, it should provide a Document object named Document. Use a different name only if Document is inappropriate (for example, if the application uses highly technical or otherwise specialized terminology within its user interface).

The Document object should have the properties listed in the following table. The properties Application, FullName, Name, Parent, Path, and Saved are required; other properties are optional.

Document object properties

Property method name	Answer class Automation type	Description
application	COMAutomation ApplicationController VT_DISPATCH	Returns the Application object; read only. Required.
author author:	StringVT_BSTR	Sets or returns summary information about the document's author; read/ write.
comments comments:	StringVT_BSTR	Sets or returns summary information comments for the document; read/ write.
fullName	StringVT_BSTR	Returns the file specification of the document, including the path; read only. Required.
keywords keywords:	StringVT_BSTR	Sets or returns summary information keywords associated with the document; read/write.
name	StringVT_BSTR	Returns the file name of the document, not including the file's path specification; read only.
parent	COMAutomationApplicationController VT_DISPATCH	Returns the Parent property of the Document object; read only. Required.
path	StringVT_BSTR	Returns the path specification for the document, not including the file name or file name extension; read only. Required.
readOnly	BooleanVT_BOOL	Returns True if the file is read only, otherwise False; read only.

Property method name	Answer class Automation type	Description
saved	BooleanVT_BOOL	Returns True if the document has never been saved, but has not changed since it was created. Returns True if it has been saved and has not changed since last saved. Returns False if it has never been saved and has changed since it was created; or if it was saved, but has changed since last saved. Read only; required.
subject subject:	StringVT_BSTR	Sets or returns summary information about the subject of the document; read/write.
title title:	StringVT_BSTR	Sets or returns summary information about the title of the document; read/write.

The Document object should have the methods listed in the following table. The methods Activate, Close, Print, Save, and SaveAs are required; other methods are optional.

Document object methods

Method name	Answer class Automation type	Description
activate	undefined VT_EMPTY	Activates the first window associated with the document. Required.
close	undefined VT_EMPTY	Closes all windows associated with the document and removes the document from the Documents collection. Required. Takes two optional arguments, saveChanges (Boolean, VT_BOOL) and filename (String, VT_BSTR). The filename argument specifies the name of the file in which to save the document.
newWindow	undefined VT_EMPTY	Creates a new window for the document.
print	undefined VT_EMPTY	Prints the document. Required. Takes three optional arguments: from (Integer, VT_I2), to (Integer, VT_I2), and copies (Integer, VT_I2). The from and to arguments specify the page range to print. The copies argument specifies the number of copies to print.

Method name	Answer class Automation type	Description
printOut	undefined VT_EMPTY	Same as Print method, but provides an easier way to use the method in Visual Basic, because Print is a Visual Basic keyword.
printPreview	undefined VT_EMPTY	Previews the pages and page breaks of the document. Equivalent to clicking Print Preview on the File menu.
revertToSaved	undefined VT_EMPTY	Reverts to the last saved copy of the document, and discards any changes.
save	undefined VT_EMPTY	Saves changes to the file specified in the document's FullName property. Required.
saveAs	undefined VT_EMPTY	Saves changes to a file. Required. Takes one optional argument, filename (String, VT_BSTR). The filename argument can include an optional path specification.

Documents Collection Object

Documents objects are wrapped by the COMAutomationDocumentsController class.

If an application supports a multiple-document interface (MDI), it should provide a Documents collection object. The collection name should be Documents, unless the name is inappropriate for the application. The Documents collection object should have all of the properties listed in the following table.

Documents collection object properties

Property method name	Answer class Automation type	Description
application	COMAutomation ApplicationController VT_DISPATCH	Returns the Application object; read only. Required.
count	Integer, VT_I4	Returns the number of items in the collection; read only. Required.
parent	COMAutomationApplicationController VT_DISPATCH	Returns the parent of the Documents collection object; read only. Required.

The Documents collection object should have all of the methods listed in the following table:

Documents collection object methods

Method name	Answer class Automation type	Description
add	COMDispatchDriver VT_DISPATCH	Creates a new document and adds it to the collection. Returns the document that was created. Required.
close	undefined VT_EMPTY	Closes all documents in the collection. Required.
item:	COMDispatchDriver VT_DISPATCH or VT_EMPTY	Returns a Document object from the collection or returns VT_EMPTY if the document does not exist. Takes an optional argument, index, which may be a string (String, VT_BSTR) indicating the document name, a number (Integer, VT_I4) indicating the ordered position within the collection, or either (VT_VARIANT). If index is omitted, returns the Document collection. The Item method is the default member (DISPID_VALUE). Required.
open	COMDispatchDriver VT_DISPATCH or VT_EMPTY	Opens an existing document and adds it to the collection. Returns the document that was opened, or nil (VT_EMPTY) if the object could not be opened. Takes one required argument, filename, and one optional argument, password. Both arguments are Strings (VT_BSTR). Required.

The Font Object

Font objects are wrapped by the COMAutomationFontController class.

The Font object may be appropriate for some applications. The properties Application, Bold, Italic, Parent, and Size are required; other properties are optional. The Font object should have the properties listed in the following table:

Font object properties

Property name	Answer class Automation type	Description
application	COMAutomationApplicationController VT_DISPATCH	Returns the Application object; read only. Required.
isBold isBold:	BooleanVT_BOOL	Sets or returns True if the font is bold, otherwise False; read/write. Required.
color color:	Integer, VT_I4	Sets or returns the RGB color of the font; read/write.
isItalic isItalic:	BooleanVT_BOOL	Sets or returns True if the font is italic; otherwise False, read/write. Required.
name	StringVT_BSTR	Returns the name of the font; read only.

Property name	Answer class Automation type	Description
isOutlineFont isOutlineFont:	BooleanVT_BOOL	Sets or returns True if the font is scaleable, otherwise False. For example, bitmapped fonts are not scaleable, whereas TrueType® fonts are scaleable; read/write.
parent	COMAutomationApplicationController VT_DISPATCH	Returns the parent of the Font object; read only. Required.
isShadow isShadow:	BooleanVT_BOOL	Sets or returns True if the font appears with a shadow, otherwise False; read/write.
size size:	FloatVT_R4	Sets or returns the point size of the font; read/write. Required.
isStrikethrough isStrikethrough:	BooleanVT_BOOL	Sets or returns True if the font appears with a line running through it, otherwise False; read/write.
isSubscript isSubscript:	BooleanVT_BOOL	Sets or returns True if the font is subscripted, otherwise False; read/write.
isSuperscript isSuperscript:	BooleanVT_BOOL	Sets or returns True if the font is superscripted, otherwise False; read/write.

15

Under the Hood

This section presents additional information that is not essential to learning how to use VisualWorks COM Connect Automation classes but helps you understand COM Connect technology.

Using AutomationObject With COMDispatchDriver

This section introduces the nuts and bolts used to control automation objects and the two objects it needs to accomplish this task: the IDispatch interface and the specification table.

The Dispatch Interface

The core of automation is the dispatch interface, or dispinterface for short, which is a specific implementation of the interface named IDispatch, which responds only to certain DISPIDs. In C++, the interface IDispatch is defined as follows:

```
interface IDispatch : public IUnknown
   {
   public:
   HRESULT GetTypeInfoCount(
      /* [out] */ UINT __RPC_FAR *pctinfo);

   HRESULT GetTypeInfo(
      /* [in] */ UINT itinfo,
      /* [in] */ LCID lcid,
      /* [out] */ ITypeInfo * *pptinfo);

   HRESULT GetIDsOfNames(
      /* [in] */ REFIID riid,
      /* [size_is][in] */ LPOLESTR *rgszNames,
      /* [in] */ UINT cNames,
```

```
        /* [in] */ LCID lcid,
        /* [size_is][out][in] */ DISPID *rgdispid);

    HRESULT Invoke(
        /* [in] */ DISPID dispidMember,
        /* [in] */ REFIID riid,
        /* [in] */ LCID lcid,
        /* [in] */ WORD wFlags,
        /* [unique][in] */ DISPPARAMS *pdispparams,
        /* [unique][out][in] */ VARIANT *pvarResult,
        /* [out] */ EXCEPINFO *pexcepinfo,
        /* [out] */ UINT *puArgErr);

    };
```

In COM Connect, the IDispatch class is used to invoke the functions in this interface. Client code typically uses an IDispatch through an instance of COMDispatchDriver.

Included with this release are sample controller applications for Word and Excel that demonstrate the use of these classes.

Through a COMDispatchDriver, a controller can retrieve the object's type information for the dispinterface, map names to DISPIDs, and invoke methods and properties. The latter happens through IDispatch::Invoke. This function has a fixed compile-time signature by which it can accept any number of arguments for the invocation of a method call, including named and optional arguments. In return, Invoke can provide any type of return value as well as rich error information.

Passing Arguments to a Dispatch Interface

In C and C++, arguments and return values handled through Invoke use the types VARIANTARG and VARIANT.

```
/* VARIANT STRUCTURE
 *
 *  VARTYPE vt;
 *  WORD wReserved1;
 *  WORD wReserved2;
 *  WORD wReserved3;
 *  union {
 *    LONG  VT_I4
 *    BYTE VT_UI1
 *    SHORT VT_I2
 *    FLOAT VT_R4
 *    DOUBLE VT_R8
```

```
 *    VARIANT_BOOL VT_BOOL
 *    SCODE VT_ERROR
 *    CY VT_CY
 *    DATE VT_DATE
 *    BSTR VT_BSTR
 *    IUnknown * VT_UNKNOWN
 *    IDispatch * VT_DISPATCH
 *    SAFEARRAY * VT_ARRAY
 *    BYTE * VT_BYREF|VT_UI1
 *    SHORT * VT_BYREF|VT_I2
 *    LONG * VT_BYREF|VT_I4
 *    FLOAT * VT_BYREF|VT_R4
 *    DOUBLE * VT_BYREF|VT_R8
 *    VARIANT_BOOL * VT_BYREF|VT_BOOL
 *    SCODE * VT_BYREF|VT_ERROR
 *    CY * VT_BYREF|VT_CY
 *    DATE * VT_BYREF|VT_DATE
 *    BSTR * VT_BYREF|VT_BSTR
 *    IUnknown ** VT_BYREF|VT_UNKNOWN
 *    IDispatch ** VT_BYREF|VT_DISPATCH
 *    SAFEARRAY ** VT_BYREF|VT_ARRAY
 *    VARIANT * VT_BYREF|VT_VARIANT
 *    PVOID VT_BYREF (Generic ByRef)
 *    CHAR VT_I1
 *    USHORT VT_UI2
 *    ULONG VT_UI4
 *    INT VT_INT
 *    UINT VT_UINT
 *    DECIMAL * VT_BYREF|VT_DECIMAL
 *    CHAR * VT_BYREF|VT_I1
 *    USHORT * VT_BYREF|VT_UI2
 *    ULONG * VT_BYREF|VT_UI4
 *    INT * VT_BYREF|VT_INT
 *    UINT * VT_BYREF|VT_UINT
 *  }
 */
struct  tagVARIANT
   {
   union
      {
      struct  __tagVARIANT
         {
         VARTYPE vt;
         WORD wReserved1;
         WORD wReserved2;
         WORD wReserved3;
         union
```

```
            {
            LONG lVal;
            BYTE bVal;
            SHORT iVal;
            FLOAT fltVal;
            DOUBLE dblVal;
            VARIANT_BOOL boolVal;
            _VARIANT_BOOL bool;
            SCODE scode;
            CY cyVal;
            DATE date;
            BSTR bstrVal;
            IUnknown __RPC_FAR *punkVal;
            IDispatch __RPC_FAR *pdispVal;
            SAFEARRAY __RPC_FAR *parray;
            BYTE __RPC_FAR *pbVal;
            SHORT __RPC_FAR *piVal;
            LONG __RPC_FAR *plVal;
            FLOAT __RPC_FAR *pfltVal;
            DOUBLE __RPC_FAR *pdblVal;
            VARIANT_BOOL __RPC_FAR *pboolVal;
            _VARIANT_BOOL __RPC_FAR *pbool;
            SCODE __RPC_FAR *pscode;
            CY __RPC_FAR *pcyVal;
            DATE __RPC_FAR *pdate;
            BSTR __RPC_FAR *pbstrVal;
            IUnknown __RPC_FAR *__RPC_FAR *ppunkVal;
            IDispatch __RPC_FAR *__RPC_FAR *ppdispVal;
            SAFEARRAY __RPC_FAR *__RPC_FAR *pparray;
            VARIANT __RPC_FAR *pvarVal;
            PVOID byref;
            CHAR cVal;
            USHORT uiVal;
            ULONG ulVal;
            INT intVal;
            UINT uintVal;
            DECIMAL __RPC_FAR *pdecVal;
            CHAR __RPC_FAR *pcVal;
            USHORT __RPC_FAR *puiVal;
            ULONG __RPC_FAR *pulVal;
            INT __RPC_FAR *pintVal;
            UINT __RPC_FAR *puintVal;
            }   __VARIANT_NAME_3;
        }   __VARIANT_NAME_2;
    DECIMAL decVal;
    }   __VARIANT_NAME_1;
```

```
  };
typedef VARIANT __RPC_FAR *LPVARIANT;
```

Both types, which are structurally identical, contain a type identifier (VARTYPE) and a value appropriate to that type, whether it is a pointer, an integer, a string pointer, a date or currency value, and so on. The value is stored in one field of a large union of types within the VARIANT. Two of these types are used frequently in Automation: the BSTR (Basic string) and the Safe Array (an array that carries its bounds with it). COM provides services to coerce a VARIANT of one type into another, compatible, type if the conversion is at all possible.

The VARIANT data type is not only used by IDispatch::Invoke but can also be used to define the types of arguments and return values for an Automation object's methods and properties. In fact, if you look at the member specifications for Excel 7, you see that the VT_VARIANT type is used everywhere. What this means is a controller does not have to worry about passing a number or a string argument, it just passes whatever suits it. It is the job of the server object to coerce the argument. On the other hand, Word 7 is not so flexible, instead of VT_VARIANT, you see types like VT_I4, VT_BSTR, etc. While COM Connect makes appropriate conversions between objects and Automation types, using the VT_VARIANT type gives the controller the most degree of flexibility.

In COM Connect the class BSTR is used to wrap the OLE Basic String type. All low-level Automation services are accessed through the OLEAutomationDLL class.

Specification Policies

The policy classes define various algorithms for method and property specification lookup and invocation. The algorithms defined in the policy classes reflects speed and space tradeoffs, as well as convenience for the programmer.

Class Hierarchy

The following defines specification class hierarchies:

Object
  COMSpecificationPolicy (*updateSpecificationTable lookupPolicy*)
    COMTypedSpecificationPolicy
    COMUntypedSpecificationPolicy
  COMLookupSpecificationPolicy
    COMNoLookupSpecificationPolicy

COMTypeCompilerLookupSpecificationPolicy
COMTypeLibraryLookupSpecificationPolicy
COMVariantLookupSpecificationPolicy

COMSpecificationPolicy

This is an abstract class that is subclassed by all other policy classes. A specification policy controls the method invocation mechanism and delegates lookup of unknown method and property specifications to its lookup policy.

Once created, a policy can be queried with canSupportIDispatch: to find out whether or not it can perform its job for a given IDispatch. A controller can then gracefully replace its choice of policy from say, a Type Compiler policy to a Variant policy.

A policy is created with one of the following class messages:

- newDefaultPolicy
- newCompletePolicy
- newTypeCompilerPolicy
- newTypeLibraryPolicy
- newVariantPolicy

The proper subclass is created and is configured with an appropriate lookup policy.

If a method or property specification is not present in the driver's specification table, the look-up policy creates a new method or property specification that is used for invocation. This new specification can be added to the dispatch driver's specification table (or not) depending on the setting of the updateSpecificationTable Boolean instance variable.

COMTypedSpecificationPolicy

An instance of this class is created for the following class messages:

- newDefaultPolicy (if it is one of the following)
- newTypeCompilerPolicy
- newTypeLibraryPolicy

COMUntypedSpecificationPolicy

An instance of this class is created for the class message newVariantPolicy.

This class message overrides the default method invocation of its superclass COMSpecificationPolicy. In Automation, a method is defined to have a return type, which can be generically processed by asking for a VT_VARIANT return data type. A method can have no return value at all, similar to the difference between a procedure and a function, in which case it must be invoked with the VT_VOID return type. Unlike the parameter passing logic, a return data type of VT_VARIANT cannot be used generically for this purpose. The COMUntypedSpecificationPolicy method invocation logic first attempts a VT_VARIANT return type invocation, and upon failure for the above stated reason, attempts a VT_VOID return type invocation. Word 7 is an example, when using the COMDispatchDriver invokeMethod: method. The invokeProcedure: method can be used to invoke a method with a VT_VOID return type directly.

COMLookupSpecificationPolicy

This is an abstract superclass for all lookup policy classes. A lookup policy is associated with a COMSpecificationPolicy subclass to create method and property specifications that the specification does not know about when invoking a method or property.

COMNoLookupSpecificationPolicy

This lookup policy is used when all method and property specifications are supplied when the instance of the COMDispatchDriver is instantiated with on:specificationTable:. If a specification is not present when a method or property is invoked, an error is raised. Specifications are created manually, with the COMSpecificationTable class or with the COMAutomationTypeAnalyzer tool class.

This is a fast way to use a dispatch driver since all specifications are already supplied, therefore no additional OLE API calls are necessary in order to properly construct all data structures associated with a particular invocation. The tradeoff is that a specification table containing all of the methods and properties that a driver will ever use must be built and kept around (usually as a literal array), therefore using memory. This can be very large in the case of Microsoft Word 7, for example.

COMTypeCompilerLookupSpecificationPolicy

This lookup policy is used when a COMDispatchDriver is instantiated with on:specificationPolicy: and the policy is specified with COMSpecificationPolicy newTypeCompilerPolicy. This algorithm looks for properties and methods through a dispatch driver's ITypeComp interface, the type compiler interface (obtained from the dispatch driver's ITypeInfo interface). This policy creates complete and typed specifications.

The ITypeComp interface is not always supported but is efficient since the lookup is a direct one-step process. If the application you are using is not associated with a type library, this policy cannot be used.

COMTypeLibraryLookupSpecificationPolicy

This lookup policy is used when a COMDispatchDriver is instantiated with on:specificationPolicy: and the policy is specified with COMSpecificationPolicy newTypeLibraryPolicy. This algorithm looks for properties and methods in a dispatch driver's type library through its ITypeInfo interface. This policy creates complete and typed specifications. This policy is useful if the Type Compiler policy cannot be employed.

Memory is required to keep track of the name to index maps. If the application you are using is not associated with a type library, this policy cannot be used.

COMVariantLookupSpecificationPolicy

This lookup policy is used when a COMDispatchDriver is instantiated with on:specificationPolicy: and the policy is specified with COMSpecificationPolicy newTypeVariantPolicy. This algorithm looks for properties and methods through a dispatch driver's GetIDsOfNames mechanism. This policy creates untyped specifications using VT_VARIANT as the generic data type. During invocation it is possible that VT_VARIANT gets rejected as the return type, in which case VT_VOID is used; see COMUntypedSpecificationPolicy.

This is a very fast way to use a dispatch driver since no additional COM function calls are necessary to properly construct all data structures associated with a particular invocation; the VT_VARIANT type is used generically to create a new specification every time. The default is to *not* update the specification table, since a call on the same name might have different parameters creating different specifications. No additional memory is used, the untyped specifications are not stored, the specification table is always empty.

16

COM Connect Server Examples

This section describes two examples, which are provided with COM Connect, of publishing COM Automation objects. The first example is AllDataTypes, which is used throughout this chapter to demonstrate the code patterns needed to work with all of the Automation data types. The second example is SmalltalkCommander, which lets an ActiveX client evaluate any Smalltalk expression and obtain an answer by using the Smalltalk Compiler class. Examples are also provided of client interaction with these Automation objects implemented in the following environments:

- COM Connect
- Visual Basic
- Visual Java
- Visual C++

Registering the Example COM Server

Both of the VisualWorks COM Server examples are saved and configured in the `Com\Examples\COMAuto\vwComSrv.im` image, which publishes both the AllDataTypes example and the SmalltalkCommander example through dual interfaces. The image was saved with the following setting:

```
COMSessionManager defaultCOMDirectoryName: 'C:\vw30\COM'.
```

This directory setting permits the start up code in the image to register the application type libraries with full pathnames. Full pathnames are required when registering type libraries.

Registration files for the server example expect the following:

- The object engine is located at:

  `C:\vw30\Bin\vwnt.exe`
- The example server image is located at:

  `C:\vw30\Com\Examples\COMAuto\vwComSrv.im`
- The SmalltalkCommander type library is located in:

  `C:\vw30\Com\Examples\COMAuto\StCom\TypeLibrary\vwstcom.tlb`
- The AllDataTypes type library is located in:

  `C:\vw30\Com\Examples\COMAuto\AllDataT\TypeLibrary\vwAllDT.tlb`

Do not depend on any PATH settings; all file references must match exactly. Setting up your system to run with a different directory structure is discussed under Modifying the Examples to Match Your Directory Structure.

To run the examples, both registration files for the examples must be run. These files are located at:

`COM\Examples\COMAuto\AllDataT\vwStCom.reg`

and

`COM\Examples\COMAuto\StCom\vwAllDT.reg.`

To register a file, double-click on the .reg file or run REGEDIT.EXE and follow these steps:

1. Choose **Start\Run...** from the Windows taskbar, and type REGEDIT.EXE.
2. Choose **Registry\Import registry file...** from the REGEDIT menu.
3. Choose the **`.reg`** file for the example.
4. Quit REGEDIT.

Once the **`.reg`** files are run, you are ready to access the COM Server objects.

How to Publish the COM Automation Server Example Image

This section shows how to make a deployment image for the COM Automation server examples. If you installed COM Connect in **`C:\vw30\COM`**, you do not need to make a new example image to reflect a different directory structure.

The following steps can be found in **`COM\Examples\COMAuto\Servers.txt`**:

1. Start with a clean image in which the COM Connect software is installed.
2. Make sure the COM and Automation examples are installed.
3. Change the directory pathname in the following code to specify the name of the directory containing your COM installation directory, if necessary, and then evaluate to configure the image with run-time settings for an object server application deployment image:

```
" Install the COM Automation dual interface servers "
"NOTE: Modify the following to specify the name of your COM
 installation directory."

COMSessionManager defaultCOMDirectoryName: 'C:\vw30\COM'.
COMSessionManager installRuntime.
AutomationAllDataTypes unregister.
AllDataTypesCOMObject installRuntime.
AutomationOnlySmalltalkCommander unregister.
SmalltalkCommanderCOMObject installRuntime.
```

4. Make a deployment image for the object server application EXE. Remember to position your open windows or close your open windows. You can also resize the Transcript window.

   At this point, the following deployment image options exist:

   Option A: Unload development parcels. If desired, strip the system using RuntimePackager. Use the RuntimePackager parameter file comserver.rtp, located in the COM directory, as a starting basis to strip the image. In RuntimePackager, be sure to specify to *retain* all relevant COM example classes (*do not* strip out the examples).

   Option B: Make a headless image.

   Option C: Save the image with a NEW name, for example VwComSrv.

5. If necessary, copy the new image to the location designated by your **`.reg`** file.

   The example COM Connect image is copied to:

   `COM\Examples\COMAuto`

Restore your image configuration to the original development settings by evaluating the following:

```
" Unregister the COM Automation dual interface servers. "
ImageConfiguration isDevelopment: true.
AllDataTypesCOMObject unregister.
SmalltalkCommanderCOMObject unregister.
```

Modifying the Examples to Match Your Directory Structure

The **`.reg`** files shipped with COM Connect are based on a directory structure with a **`C:\vw30\COM`** root, as described previously under Registering the Example COM Server.

For each example server you want to run, modify the registration file to reflect your directory structure.

The registration file for the AllDataTypes example is located in **`COM\Examples\COMAuto\AllDataT\vwAllDt.reg`**. The following listing shows which entry must be changed to suit your directory structure:

```
HKEY_CLASSES_ROOT
   CLSID
      {DB5DE8E3-AD1F-11d0-ACBE-5E86B1000000}
         LocalServer32 = C:\vw30\Bin\vwnt.exe
            C:\vw30\Com\Examples\COMAuto\vwComSrv.im /Automation
```

The registration file for the SmalltalkCommander example is in Com\Examples\COMAuto\SmalltalkCommander\vwStCom.reg. The following listing shows which entry must be changed to suit your directory structure:

```
HKEY_CLASSES_ROOT
   CLSID
      {5FD2D2B1-95A8-11d0-ACAB-E80467000000}
         LocalServer32 = C:\vw30\Bin\vwnt.exe
            C:\vw30\Com\Examples\COMAuto\vwComSrv.im /Automation
```

The following class methods must be adapted to your directory structure only if you change the internal directory structure for the example directories. For the AllDataTypes example to publish the AutomationAllDataTypes class with an IDispatch, the newTypeLibraryEnglish class method must be modified.

```
newTypeLibraryEnglish
    "Answer a type library for the English language for the application."
    ^COMTypeLibrary new
        libraryID: self typeLibraryID;
        lcid: COMTypeLibrary lcidEnglish;
        directoryName: COMSessionManager absoluteCOMDirectoryName,
            'Examples\COMAuto\AllDataTypes\TypeLibrary';
        fileName: 'VwAllDT.tlb';
        majorVersion: 1;
        minorVersion: 0
```

For the AllDataTypes example to publish the AutomationAllDataTypes class with a dual interface, the newTypeLibraryEnglish class method in AllDataTypesCOMObject must be modified.

```
newTypeLibraryEnglish
    "Answer a type library for the English language for the application."
    ^COMTypeLibrary new
        libraryID: AutomationAllDataTypes typeLibraryID;
        lcid: COMTypeLibrary lcidEnglish;
        directoryName: COMSessionManager absoluteCOMDirectoryName,
            'Examples\COMAuto\AllDataTypes\TypeLibrary';
        fileName: 'VwAllDT.tlb';
        majorVersion: 1;
        minorVersion: 0
```

The SmalltalkCommander example also has similar methods in the AutomationSmalltalkCommander and SmalltalkCommanderCOMObject classes.

Starting a Deployed Image Manually

There might some debugging situations when you want to start an image that has been saved to serve COM and Automation objects instead of making a new deployment image.

To start an image that has been saved to serve COM and Automation objects, start it with an equivalent command line similar to this one, which includes the /Automation flag.

```
C:\vw30\Bin\vwnt.exe C:\vw30\Com\Examples\
COMAuto\vwComSrv.im /Automation
```

This should be similar to the command in the **.reg** file.

The Smalltalk Commander Examples

The SmalltalkCommander example makes for great demonstrations, because it allows you to evaluate any Smalltalk expression from any COM compliant client. This example is located in **COM\Examples\COMAuto\StCom**.

This example is comprised of a COM Connect server image and various client applications used to demonstrate flexibility in choosing client environments. This example lets a client evaluate any Smalltalk expression and get an answer either as a string or as an Automation object. This server example can be published through an IDispatch interface or through a custom dual interface called ISmalltalkCommanderDisp.

The following table lists **COM\Examples\COMAuto\StCom** subdirectories.

Subdirectory	Description
VB4	Contains a Visual Basic 4 client application.
CStCom	Contains a Visual C++ 5.0 client application.
VJ++	Contains a Visual J++ 1.1 client application.
TypeLibrary	Contains the .idl source files and header files produced by the MIDL compiler.
Help	Contains the help source and .hlp file for this example used by the various clients.

COM Connect example expressions are in the comments for the AutomationSmalltalkCommander and SmalltalkCommanderCOMObject classes.The server example uses the VisualWorks class Compiler and might be subject to restrictions for distribution. Consult the VisualWorks documentation on this topic.

To run this sample, you must run the registration file located at **COM\Examples\COMAuto\StCom\vwStCom.reg**.

To register the file, double-click on the .reg file or run REGEDIT.EXE and follow these steps:

1. Choose **Start\Run...** from the Windows taskbar, and type REGEDIT.EXE.
2. Choose **Registry\Import registry file...** from the REGEDIT menu.

3 Choose the .reg file for the example.

4 Quit REGEDIT.

COM Connect Client Example: The Smalltalk Commander

The class comments for the AutomationSmalltalkCommander and SmalltalkCommanderCOMObject classes contain example expressions for running the COM server from a COM Connect image.

Accessing With the Standard IDispatch

To start the server and access its services through the standard Automation IDispatch interface, evaluate:

```
"Run the server from here. Inspect the expression:"
COMDispatchDriver createObject: 'VisualWorks.SmalltalkCommander'.
```

Copy and paste the expressions into the inspector from the AutomationSmalltalkCommander class comment, and evaluate:

```
self invokeMethod: 'Evaluate' with: '3+4'.
```

which answers the Smalltalk Integer 7.

Evaluating the following:

```
self invokeMethod: 'EvaluateAsString' with: '100 factorial'.
```

answers the following Smalltalk string:

```
‘9332621544394415268169923885626670049071596826438162146859
2963895217599993229915608941463976156518286253697920827223
758251185210916864000000000000000000000000’
```

To releases the client resources, evaluate

```
self release
```

Accessing With the Dual Interface ISmalltalkCommanderDisp

To start the server and access its services through the custom ISmalltalkCommanderDisp interface, inspect:

```
"Get a dispatch driver running on the dual interface. Inspect: "
( IClassFactory
   createInstance: SmalltalkCommanderCOMObject clsid
   iid: ISmalltalkCommanderDisp iid ) asDispatchDriver.
```

Note that in this example, we wrap the IClassFactory answer with a COMDispatchDriver. Since ISmalltalkCommanderDisp is a dual interface, it supports IDispatch. This lets you use the expressions from the previous section to insure that the same results are obtained, whether a client uses an IDispatch or the custom ISmalltalkCommanderDisp interface.

To work with an ISmalltalkCommanderDisp, inspect:

```
( IClassFactory
   createInstance: SmalltalkCommanderCOMObject clsid
   iid: ISmalltalkCommanderDisp iid ).
```

Copy the expressions from the SmalltalkCommanderCOMObject class comment and paste them in the inspector. When you are using a COMDispatchDriver, you get the dual interface with the dispatchInterface message.

```
self dispatchInterface evaluate: '3+4'.
self dispatchInterface evaluateAsString: '100 factorial'.
```

If you have an ISmalltalkCommanderDisp interface, send the messages directly.

```
self evaluate: '3+4'.
self evaluateAsString: '100 factorial'.
self release.    "release the client resources. "
```

Releasing the Interface

The release message releases client resources. The example image is set up to stay alive even when the last reference to a server object is released. If this is not the case, the server image quits. The termination policy for a server image is discussed under Publishing Automation Objects. If you really want to terminate the server, invoke the Quit method.

Visual Basic Client Example: The Smalltalk Commander

The Visual Basic 4 example opens a window and lets you evaluate any Smalltalk expression from a text box. The answer of the Smalltalk expression is displayed in another text box. Smalltalk errors are reported in the answer text box, and a stack trace from expression evaluation errors can be viewed in a separate window.

This example is located in **`COM\Examples\COMAuto\StCom\ VB4`** and includes (among others) the files listed in the following table:

Filename	Description
vbStCom.vbp	The Visual Basic 4 project file. Double-click on this file to start Microsoft Visual Basic.
VbStCom.exe	A run-time version of this Visual Basic example. Run this EXE if you have Visual Basic 4 or the Visual Basic 4 runtime installed on your system.
Installer\SetUp.exe	An installation program for this example client. Run this EXE if you do not have Visual Basic 4 or the Visual Basic 4 run-time installed on your system. This installer also copies the Visual Basic 4 run-time files into your system; thus, you do not need a copy of Visual Basic to run this example.

Visual Basic Client Example: The Class Hierarchy Browser

This example uses the SmalltalkCommander to implement a simple class hierarchy browser in Visual Basic 4. This example is located in **`COM\Examples\COMAuto\CHB`** and includes (among others) the following files:

Filename	Description
ChbVw.vbp	The Visual Basic project file. Double-click on this file to start Microsoft Visual Basic.
ChbVw.exe	A run-time version of this Visual Basic example. Run this EXE if you have Visual Basic 4 or the Visual Basic 4 runtime installed on your system.
Installer\SetUp.exe	An installation program for this example client. Run this EXE if you do not have Visual Basic 4 or the Visual Basic 4 run-time installed on your system. This installer also copies the Visual Basic 4 run-time files to your system; thus, you do not need a copy of Visual Basic to run this example.

Visual C++ Client Example: The Smalltalk Commander

The C++ 5.0 example opens a window and lets you evaluate any Smalltalk expression from a text box. The answer of the Smalltalk expression is displayed in another text box. Smalltalk errors are reported in the answer text box.

This example is in **`COM\Examples\COMAuto\StCom\CStCom`** and includes (among others) the following files:

Filename	Description
CstCom.dsw	The Visual C++ 5.0 workspace file. Double-click on this file to start Microsoft Visual C++.
Debug\CstCom.exe	A debug version of the executable file.
Release\CstCom.exe	A release version of the executable file.

Note that the compiler environment for this project is set to include "..\TypeLibrary" in the INCLUDE search path.

Visual J++ Client Example: The Smalltalk Commander

The Visual J++ 1.1 example opens a window and lets you evaluate any Smalltalk expression from a text box. The answer of the Smalltalk expression is displayed in another text box. Smalltalk errors are reported in the answer text box.

This example is in **`COM\Examples\COMAuto\StCom\VJ++`** and includes (among others) the following files:

Filename	Description
VJStCom.dsw	The Microsoft Visual J++ 1.1 project workspace. Double-click on this file to start Microsoft Visual J++.
VJStCom.java	The main Java source file.
VJStCom.html	An HTML file to run the applet.

If you run **`VJStCom.html`** by itself (not from Microsoft Visual J++) and the applet does not run, your system might not be configured properly. This can happen even if the applet runs when Microsoft Visual J++ launches your web browser. The Internet Explorer might place an error message in the status bar, when the mouse is over the applet area.

The AllDataTypes Examples

This example is comprised of a COM Connect server image and client example expressions. This example is used throughout this document to illustrate the use of all Automation compatible data types. This server example can be published through an IDispatch interface or through a custom dual interface called IAllDataTypesDisp.

This example is in **COM\Examples\COMAuto\AllDataT** and its subdirectories.

Subdirectory	Description
TypeLibrary	Contains the .idl source files and header files produced by the MIDL compiler.

The AllDataTypes Example Server

The AllDataTypes example is used throughout this chapter to illustrate the use of all Automation data types to demonstrate all of the code patterns you might need in creating an Automation object, in particular an Automation object that implements a dual interface. The files for this example are located in **COM\Examples\COMAuto\AllDataT**. To run this sample, the **COM\Examples\COMAuto\AllDataT\vwAllDT.reg** registration file for the example must be run.

To register a file, double-click on the **.reg** file or run **REGEDIT.EXE** and follow these steps:

1 Choose **Start\Run**... from the Windows taskbar, and type REGEDIT.EXE.

2 Choose Registry\Import registry file... from the REGEDIT menu.

3 Choose the .reg file for the example.

4 Quit REGEDIT.

The COM Connect Example Client

The class comments for the AutomationAllDataTypes and AllDataTypesCOMObject classes contain example expressions to run the COM server from a COM Connect image.

Accessing With the Standard IDispatch

To start the server and access its services through the standard Automation IDispatch interface, inspect:

```
"Run the server from here. Inspect the expression:"
COMDispatchDriver createObject: 'VisualWorks.AllDataTypes'.
```

Copy from the class comment for AutomationAllDataTypes, paste the expressions in the inspector, and evaluate the following:

```
"Setting properties."

| anIUnknown |
self setProperty: 'LONGValue' value: 76000.
self setProperty: 'BYTEValue' value: 1.
self setProperty: 'SHORTValue' value: 2.
self setProperty: 'FLOATValue' value: 0.333.
self setProperty: 'DOUBLEValue' value: 800.001.
self setProperty: 'VARIANT_BOOLValue' value: true.
self setProperty: 'SCODEValue' value: 0.
self setProperty: 'DATEValue' value: Timestamp now.
self setProperty: 'BSTRValue' value: 'Bonjour'.
anIUnknown := self dispatchInterface queryInterface: IUnknown iid.
[ self setProperty: 'IUnknownReference' value: anIUnknown ] ensure:
   [ anIUnknown release ].
self setProperty: 'IDispatchReference' value: self dispatchInterface.
self setProperty: 'VARIANTValue' value: ( Array with: 1 with: 2 with: 3
   with: 4 ).
self setProperty: 'CURRENCYValue' value: 10.

"Getting properties."
self getProperty: 'LONGValue'.
self getProperty: 'BYTEValue'.
self getProperty: 'SHORTValue'.
self getProperty: 'FLOATValue'.
self getProperty: 'DOUBLEValue'.
self getProperty: 'VARIANT_BOOLValue'.
self getProperty: 'SCODEValue'.
self getProperty: 'DATEValue'.
self getProperty: 'BSTRValue'.
self getProperty: 'IUnknownReference'.
self getProperty: 'IDispatchReference'.
self getProperty: 'VARIANTValue'.
self getProperty: 'CURRENCYValue'.
self release.
```

Accessing With the Dual Interface IAllDataTypesDisp

To start the server and access its services through the custom IAllDataTypesDisp interface, inspect:

```
"Get a dispatch driver running on the dual interface. Inspect: "
 ( IClassFactory
   createInstance: AllDataTypesCOMObject clsid
   iid: IAllDataTypesDisp iid ) asDispatchDriver.
```

Note that in this example, we wrap the IClassFactory answer with a COMDispatchDriver. Since IAllDataTypesDisp is a dual interface, it supports IDispatch. This lets you use the expressions from the previous section to insure that the same results are obtained, whether a client uses an IDispatch or the custom IAllDataTypesDisp interface.

Copy the expressions from the AllDataTypesCOMObject class comment and paste them in the inspector. When you are using a COMDispatchDriver, you get the dual interface with the dispatchInterface message.

```
"Setting properties."

| anIUnknown anIDispatch aDualInterface |
self dispatchInterface reset.
self dispatchInterface put_LONGValue: 76000.
self dispatchInterface put_BYTEValue: 1.
self dispatchInterface put_SHORTValue: 2.
self dispatchInterface put_FLOATValue: 0.333.
self dispatchInterface put_DOUBLEValue: 800.001.
self dispatchInterface put_VARIANT_BOOLValue: true.
self dispatchInterface put_SCODEValue: 0.
self dispatchInterface put_CURRENCYValue:
( FixedPoint numerator: 9223372036854775807 denominator: 10000
   scale: 4 ).

   "The largest CY value."
self dispatchInterface put_DATEValue: Timestamp now.
self dispatchInterface put_BSTRValue: 'Bonjour'.
anIUnknown := self dispatchInterface queryInterface: IUnknown iid.
[ self dispatchInterface put_IUnknownReference: anIUnknown ] ensure:
   [ anIUnknown release ].
anIDispatch := self dispatchInterface queryInterface: IDispatch iid.
[ self dispatchInterface put_IDispatchReference: anIDispatch ] ensure:
   [ anIDispatch release ].
self dispatchInterface put_IDispatchReference: self dispatchInterface.
self dispatchInterface put_VARIANTValue: ( Array with: 1 with: 2 with: 3
with: 4 ).
self dispatchInterface put_SAFEARRAY_I4Value: ( Array with: 10 with: 20
with: 30 with: 40 ).
anIDispatch := self dispatchInterface queryInterface: IDispatch iid.
[ aDualInterface := self dispatchInterface
   queryInterface: IAllDataTypesDisp iid.
   self dispatchInterface put_SAFEARRAY_DISPATCHValue:
      ( Array with: anIDispatch with: aDualInterface ) ]
         ensure: [ anIDispatch release. aDualInterface release ].

"Getting properties."
self dispatchInterface get_LONGValue.
```

```
self dispatchInterface get_BYTEValue.
self dispatchInterface get_SHORTValue.
self dispatchInterface get_FLOATValue.
self dispatchInterface get_DOUBLEValue.
self dispatchInterface get_VARIANT_BOOLValue.
self dispatchInterface get_SCODEValue.
self dispatchInterface get_DATEValue.
self dispatchInterface get_BSTRValue.
( self dispatchInterface get_IUnknownReference ) release.
( self dispatchInterface get_IDispatchReference ) release.
self dispatchInterface get_VARIANTValue.
self dispatchInterface get_SAFEARRAY_I4Value.
( self dispatchInterface get_SAFEARRAY_DISPATCHValue do:
    [ : anInterface anInterface release ].
self release.
```

To work with an IAllDataTypesDisp, inspect:

```
( IClassFactory
    createInstance: AllDataTypesCOMObject clsid
    iid: IAllDataTypesDisp iid )
```

If you have an IAllDataTypesDisp interface, you can send the messages directly, as follows:

```
self put_LONGValue: 76000.
self get_LONGValue.
```

Releasing the Interface

The release message releases client resources. The example image is set up to stay alive even when the last reference to a server object is released. If this is not the case, the server image quits. The termination policy for a server image is discussed under Publishing Automation Objects. If you really want to terminate the server, invoke the Quit method.

Glossary

accessor function A function that sets or retrieves the value of a property. Most properties have a pair of accessor functions. Properties that are read-only may have only one accessor function.

ActiveX Microsoft's brand name for the technologies that enable interoperability using the Component Object Model (COM). ActiveX technology includes, but is not limited to, OLE.

class identifier (CLSID) A universally unique identifier (UUID) for an application class that identifies an object. An object registers its class identifier (CLSID) in the system registration database so that it can be loaded and programmed by other applications.

class factory An object that implements the IClassFactory interface, which allows it to create other objects of a specific class.

coclass (component class) Component object class. A top-level object in the object hierarchy.

code page The mapping between character glyphs (shapes) and the 1-byte or 2-byte numeric values that are used to represent them.

collection object A grouping of exposed objects. A collection object can address multiple occurrences of an object as a unit (for example, to draw a set of points).

Component Object Model (COM)

The programming model and binary standard on which OLE is based. COM defines how objects and their clients interact within processes or across process boundaries.

compound document A document that contains data of different formats, such as sound clips, spreadsheets, text, and bitmaps, created by different applications. Compound documents are stored by container applications.

container application An OLE-based application that provides storage, a display site, and access to a compound document object.

custom interface
A user-defined COM interface that is not defined as part of OLE.

Dispatch identifier (DISPID)
The number by which a member function, parameter, or data member of an object is known internally to the IDispatch interface.

dispinterface(dispatch interface)
An IDispatch interface that responds only to a certain fixed set of names. The properties and methods of the dispinterface are not in the virtual function table (VTBL) for the object.

dual interface
An interface that supports both IDispatch and VTBL binding.

event
An action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. In Automation, an event is a method that is called, rather than implemented, by an Automation object.

event sink
A function that handles events. The code associated with a Visual Basic form, which contains event handlers for one or more controls, is an event sink.

event source
A control that experiences events and calls an event handler to dispose of them.

exposed object
See Automation object.

HRESULT
A value returned from a function call to an interface, consisting of a severity code, context information, a facility code, and a status code that describes the result. For 16-bit Windows systems, the HRESULT is an opaque result handle defined to be zero for a successful return from a function, and nonzero if error or status information is to be returned. To convert an HRESULT into a more detailed SCODE (or return value), applications call GetSCode(). See SCODE.

ID binding
The ability to bind member names to dispatch identifiers (DISPIDs) at compile time (for example, by obtaining the IDs from a type library). This approach eliminates the need for calls to IDispatch::GetIDsOfNames, and results in improved performance over late-bound calls. See also late binding and VTBL binding.

in-place activation
The ability to activate an object from within an OLE control and to associate a verb with that activation (for example, edit, play, change). Sometimes referred to as in-place editing or visual editing.

in-process server An object application that runs in the same process space as the Automation controller.

interfaces One or more well-defined base classes providing member functions that, when implemented in an application, provide a specific service. Interfaces may include compiled support functions to simplify their implementation.

late binding The ability to bind member names to dispatch identifiers (IDs) at run time, rather than at compile time. See also ID binding and VTBL binding.

LCID (locale identifier) A 32-bit value that identifies the human language preferred by a user, region, or application.

locale User-preference information, represented as a list of values describing the user's language and sublanguage.

MkTypLib utility A library creation utility that compiles scripts written in the Object Description Language. This utility is obsolete; the Microsoft Interface Definition Language (MIDL) compiler should be used instead of MkTypLib.

marshaling The process of packaging and sending interface parameters across process boundaries.

member function One of a group of related functions that make up an interface. See also method and property.

method A member function of an exposed object that performs some action on the object, such as saving it to disk.

MIDL compiler The Microsoft Interface Definition Library (MIDL) compiler can be used to generate a type library. For information about the MIDL compiler, refer to the Microsoft Interface Definition Language Programmer’s Guide and Reference in the Win32 Software Development Kit (SDK) section of the Microsoft Developer’s Network (MSDN).

multiple-document nterface (MDI) application

An application that can support multiple documents from one application instance. MDI object applications can simultaneously service a user and one or more embedding containers. See also single-document interface (SDI) application.

naming guidelines Recommendations meant to improve consistency and readability across applications.

Object Description Language (ODL)

A scripting language used to describe exposed libraries, objects, types, and interfaces. ODL scripts are compiled into type libraries by the MkTypLib tool.

OLE

Microsoft's object-based technology for sharing information and services across process and machine boundaries (object linking and embedding).

out-of process server

An object application implemented in an executable file that runs in a separate process space from the Automation controller.

programmable object

See Automation object.

programmatic identifier (ProgID)

An application's unique name that is mapped to the system registry by the class identifier (CLSID). For example, registering Microsoft Excel associates a CLSID with the ProgID Excel.Application.

property

A data member of an exposed object. Properties are set or returned by means of get and put accessor functions.

proxy

An interface-specific object that packages parameters for that interface in preparation for a remote method call. A proxy runs in the address space of the sender and communicates with a corresponding stub in the receiver's address space. See also stub, marshaling, and unmarshaling.

running object table (ROT)

A globally accessible table on each computer that keeps track of all COM objects in the running state that can be identified by a moniker. Moniker providers register an object in the table, which increments the object's reference count. Before the object can be destroyed, its moniker must be released from the table.

safe array

An array that contains information about the number of dimensions and the bounds of its dimensions. Safe arrays are passed by IDispatch::Invoke within VARIANTARGs. Their base type is VT_tag | VT_ARRAY.

SCODE

A DWORD value that is used in 16-bit systems to pass detailed information to the caller of an interface member or API function. The status codes for OLE interfaces and APIs are defined in FACILITY_ITF. See HRESULT.

single-document interface (SDI) application

An application that can support only one document at a time. Multiple instances of an SDI application must be started to service both an embedded object and a user. See also multiple-document interface (MDI) application.

standard objects

A set of objects defined by Automation, including the following: Application, Document, Documents, and Font.

stub

An interface-specific object that unpackages the parameters for that interface after they are marshaled across the process boundary, and makes the requested method call. The stub runs in the address space of the receiver and communicates with a corresponding proxy in the sender's address space. See proxy, marshaling, and unmarshaling.

type description

The information used to build the type information for one or more aspects of an application's interface. Type descriptions are written in Object Description Language (ODL), and include both programmable and nonprogrammable interfaces.

type information

Information that describes the interfaces of an application. Type information is created from type descriptions using OLE Automation tools, such as MkTypLib or the CreateDispTypeInfo function. Type information can be accessed through the ITypeInfo interface.

type information element

A unit of information identified by one of these statements in a type description: typedef, enum, struct, module, interface, dispinterface, or coclass.

type library

A file or component within another file that contains type information about exposed objects. Type libraries are created using either the MkTypLib utility or the MIDL compiler, and can be accessed through the ITypeLib interface.

unmarshaling

The process of unpackaging parameters that have been sent across process boundaries.

Value property

The property that defines the default behavior of an object when no other methods or properties are specified. Indicate the Value property by specifying the default attribute in ODL.

virtual function table (VTBL)

A table of function pointers, such as an implementation of a class in C++. The pointers in the VTBL point to the members of the interfaces that an object supports.

VTBL binding

A process that allows an ActiveX client to call a method or property accessor function directly without using the IDispatch interface. VTBL binding is faster than both ID binding and late binding because the access is direct. See also late binding and ID binding.

Index

S

T

U

V

W