



ArcGIS® Desktop Developer Guide

ArcGIS® 9.0

PUBLISHED BY
ESRI
380 New York Street
Redlands, California 92373-8100

Copyright © 2004 ESRI
All rights reserved.
Printed in the United States of America.

The information contained in this document is the exclusive property of ESRI. This work is protected under United States copyright law and other international copyright treaties and conventions. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, except as expressly permitted in writing by ESRI. All requests should be sent to Attention: Contracts Manager; ESRI, 380 New York Street, Redlands, California 92373-8100, USA.

The information contained in this document is subject to change without notice.

Contributing Writers

Euan Cameron, Rob Elkins, Shelly Gill, Sean Jones, Allan Laframboise, Glenn Meister, Steve Van Esch

U.S. GOVERNMENT RESTRICTED/LIMITED RIGHTS

Any software, documentation, and/or data delivered hereunder is subject to the terms of the License Agreement. In no event shall the U.S. Government acquire greater than RESTRICTED/LIMITED RIGHTS. At a minimum, use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR §52.227-14 Alternates I, II, and III (JUN 1987); FAR §52.227-19 (JUN 1987) and/or FAR §12.211/12.212 (Commercial Technical Data/Computer Software); and DFARS §252.227-7015 (NOV 1995) (Technical Data) and/or DFARS §227.7202 (Computer Software), as applicable. Contractor/Manufacturer is ESRI, 380 New York Street, Redlands, California 92373-8100, USA.

ESRI, ArcView, ArcIMS, SDE, the ESRI globe logo, ArcGlobe, StreetMap, ArcReader, ArcPad, ArcScan, ArcObjects, ArcGIS, ArcMap, ArcCatalog, ArcScene, ArcInfo, ArcEdit, ArcEditor, ArcToolbox, 3D Analyst, ArcPress, ArcSDE, GIS by ESRI, the ArcGIS logo, and www.esri.com are trademarks, registered trademarks, or service marks of ESRI in the United States, the European Community, or certain other jurisdictions.

Other companies and products mentioned herein are trademarks or registered trademarks of their respective trademark owners.

Contents

CHAPTER 1: INTRODUCING ArcGIS DESKTOP DEVELOPMENT	1
ArcGIS 9 developer overview	2
ArcGIS Desktop developer overview	6
Using this guide	10
ArcGIS developer resources	12
CHAPTER 2: ArcGIS SOFTWARE ARCHITECTURE	17
ArcGIS software architecture	18
ArcGIS application programming interfaces	23
ArcGIS Engine libraries	25
ArcGIS Desktop application libraries	33
CHAPTER 3: DEVELOPING FOR ArcGIS DESKTOP APPLICATIONS	45
Customizing ArcGIS Desktop	46
Storing customizations	49
Customizing ArcGIS Desktop applications with VBA	53
Component development	64
Choosing a component development environment	67
Building an ArcGIS Desktop component	68
CHAPTER 4: LICENSING AND DEPLOYMENT	73
ArcGIS license checking	74
Packing and deploying customizations	86
CHAPTER 5: DEVELOPER SCENARIOS	93
Create a toolbar: Command, Tool, and Menu	94
Extensions	113
Dockable window	119
APPENDIX A: DEVELOPER ENVIRONMENTS	131
The Microsoft Component Object Model	132
Developing with ArcObjects	144
The Visual Basic 6 environment	153
Visual Basic for Applications	166
The Visual Basic 6 development environment	172
Visual C++	179
.NET Application Programming Interface	221

APPENDIX B: READING THE OBJECT MODEL DIAGRAMS	259
Interpreting the object model diagrams	260
APPENDIX C: ILLUSTRATED CODE SAMPLES	263
Locate and execute command on toolbar	265
Draw digitized line on screen	266
Add feature class to ArcMap	268
Add layer to ArcMap using GxDialog	270
Style gallery auto symbol selection	272
Loop through selected area features	274
Spatial query	276
Add map surround to page layout	278
Add text callout to active view	280
Geometry projection	282
Display raster cell value in status bar	284
Export current view	286
Print current view	288
Display map extent in GxView as envelope	290
Edit feature class schema	292
APPENDIX D: ArcOBJECTS PROBLEM-SOLVING GUIDE	295
ArcObjects problem-solving guide	296
APPENDIX E: UICONTROLS	313
UIControl classes	314
UIButtonControl class	315
UIComboBoxControl class	316
UIEditBoxControl class	317
UIToolControl class	318
APPENDIX F: BIBLIOGRAPHY	319
INDEX	323

1

Introducing ArcGIS Desktop development

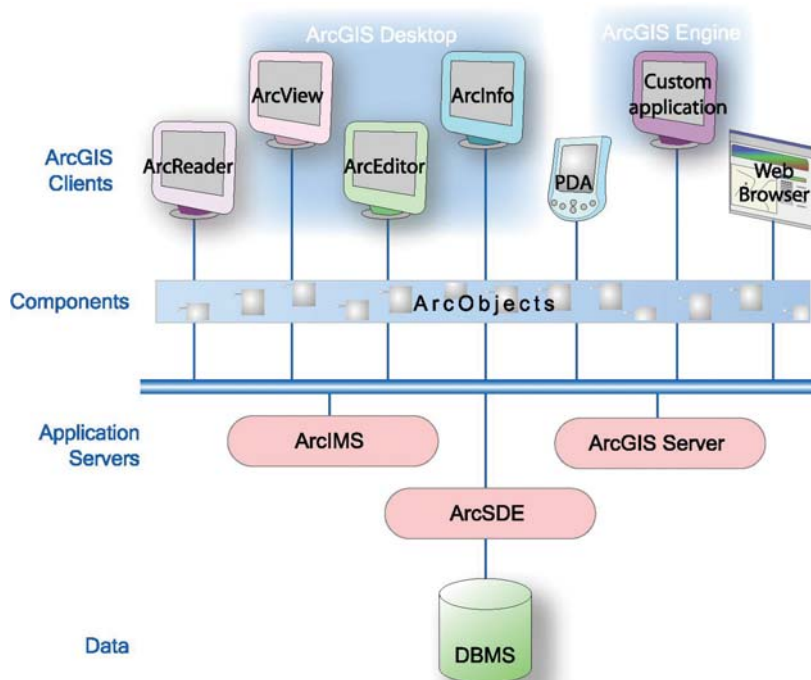
The ESRI® ArcGIS® Desktop Developer Guide is an introduction for anyone who wants to customize or extend ArcGIS Desktop applications, such as ArcMap™ or ArcCatalog™.

This chapter introduces the ArcGIS Desktop development environment in the context of the ArcGIS product family.

Topics covered in this chapter include an ArcGIS 9 developer overview, an ArcGIS Desktop developer overview, using this guide, and ArcGIS developer resources.

ArcGIS 9 is an integrated family of geographic information system (GIS) software products for building a complete GIS. It is based on a common library of shared GIS software components called ArcObjects™. ArcGIS 9 consists of four key parts:

- **Desktop GIS**, an integrated suite of advanced GIS applications.
- **Embedded GIS**, embeddable GIS component libraries for building custom applications using C++, COM, .NET, and Java™.
- **Server GIS**, a shared library of GIS software objects used to build server-side GIS applications in enterprise and Web computing frameworks. Used for building both SOAP-based Web services and Web applications using .NET/ASP and Java/JSP.
- **Mobile GIS**, GIS Web services to publish maps, data, and metadata through open Internet protocols.



ArcSDE® Gateway is an interface for managing geodatabases in numerous relational database management systems (RDBMSs).

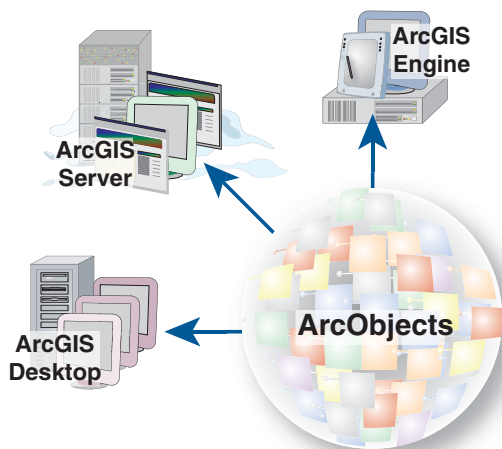
ArcGIS provides a scalable framework for implementing GIS for a single user or for many users on desktops and servers. This book will be of greatest use to developers who want to use the ArcGIS Desktop Developer Kit to customize and extend the ArcView®, ArcEditor™, or ArcInfo™ desktop products. This book provides a general explanation of the options and opportunities available to developers with ArcGIS Desktop. Several scenarios will illustrate with code examples the different types of customization that can be developed with the ArcGIS Desktop Developer Kit.

ArcGIS is a platform for building geographic information systems. ArcGIS 9 will extend the system with major new capabilities in the areas of geoprocessing, 3D visualization, and developer tools. Two new products, ArcGIS Engine and ArcGIS Server, are introduced at this release, making ArcGIS a complete system for application and server development.

There is a wide range of possibilities when developing with ArcGIS, including the following:

- Configure/Customize ArcGIS applications, such as ArcMap and ArcCatalog.
- Extend the ArcGIS architecture and data model.
- Embed maps and GIS functionality in other applications with ArcGIS Engine.
- Build and deploy custom desktop applications with ArcGIS Engine.
- Build Web services and applications with ArcGIS Server.

ArcGIS 9 has a common developer experience across all ArcGIS products (Engine, Server, and Desktop). This book focuses on customizing and extending the ArcGIS Desktop applications. Developers wanting to build custom standalone applications or work with ArcGIS Server should refer to the *ArcGIS Engine Developer Guide* and the *ArcGIS Server Administration and Development Guide*, respectively.



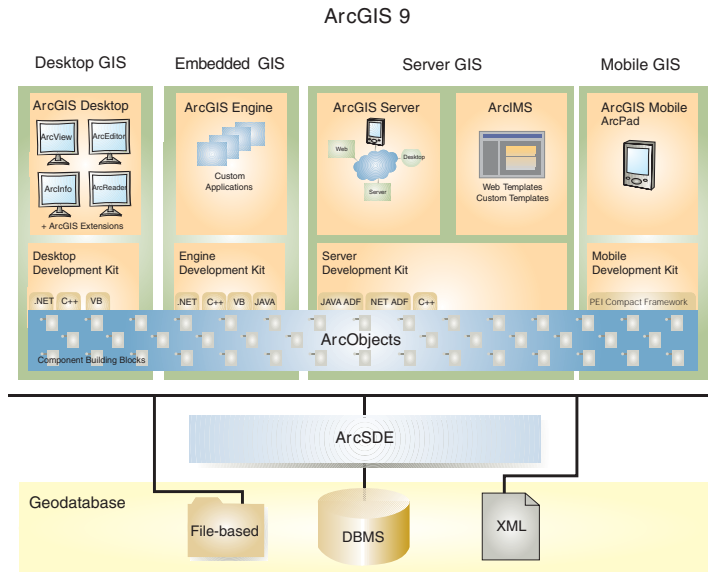
The ArcGIS system is built and extended using ArcObjects software components. ArcObjects includes a wide variety of programmable components, ranging from fine-grained objects (for example, individual geometry objects) to coarse-grained objects (for example, a map object to interact with existing ArcMap documents), which aggregate comprehensive GIS functionality for developers.

Developers work with ArcObjects using standard programming frameworks to extend ArcGIS Desktop, build custom applications with ArcGIS Engine, and implement server GIS applications for various enterprise purposes using ArcGIS Server.

Each of the ArcGIS product architectures built with ArcObjects represents alternative application development containers for GIS software developers, including desktops, embeddable engines, and servers.

ArcGIS SOFTWARE DEVELOPER KITS

The ArcGIS Engine, Server, and Desktop have software developer kits (SDK) for working with ArcObjects. The developer kits provide everything needed to be successful with ArcObjects, including developer documentation, samples, and object model diagrams.



The ArcGIS Desktop Developer Kit

ArcGIS Desktop includes a series of Windows® desktop application frameworks—ArcMap, ArcCatalog, ArcToolbox™—and more, each with user interface components. ArcGIS Desktop is available at three functional levels (ArcView, ArcEditor, and ArcInfo) and can be customized and extended using the ArcGIS Desktop Developer Kit.

The software developer kit for ArcGIS Desktop is included with ArcView, ArcEditor, and ArcInfo and supports the COM and .NET programming frameworks. Many users apply the ArcGIS Desktop Developer Kit to add extended functions, new GIS tools, custom user interfaces (UIs), and full extensions for improving professional GIS productivity of ArcGIS Desktop.

The ArcGIS Engine Developer Kit

ArcGIS Engine is a simple, application-neutral programming environment for ArcObjects. The ArcGIS Engine Developer Kit provides a series of embeddable ArcGIS components that are used outside the ArcGIS Desktop application

framework (for example, mapping objects are managed as part of ArcGIS Engine, rather than in ArcMap). Using the ArcGIS Engine Developer Kit, developers build focused GIS solutions with simple interfaces to access any set of GIS functions or can embed GIS logic in existing user applications to deploy GIS to broad groups of users. ArcGIS Engine has a COM, .NET, Java, and C++ application programming interface (API) for developers.

The ArcGIS Server Developer Kit

ArcGIS Server defines and implements a set of standard GIS Web services (for example, mapping, data access, and geocoding), as well as supports enterprise-level application development based on ArcObjects for the server.

The ArcGIS Server Developer Kit enables developers to build central GIS servers to host GIS functions that are accessed by many users, perform back office processing on large central GIS databases, build and deliver GIS Web applications, and perform distributed GIS computing.

ArcGIS Desktop includes a suite of integrated applications, including ArcMap, ArcCatalog, and ArcToolbox. By using these applications and interfaces in unison, you can perform any GIS task, simple to advanced, including mapping, geographic analysis, data editing and compilation, data management, visualization, and geoprocessing.

WHAT ARE ArcVIEW, ArcEDITOR, AND ArcINFO?

ArcGIS Desktop is the information authoring and usage tool for GIS professionals. It is scalable as three separate software products to meet the needs of many types of users.

ArcView provides comprehensive mapping and analysis tools along with simple editing and geoprocessing.

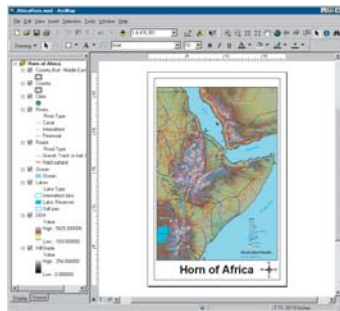
ArcEditor includes advanced editing capabilities for shapefiles and geodatabases in addition to the full functionality of ArcView.

ArcInfo is the flagship ArcGIS Desktop product. It builds on the functionality of ArcEditor with advanced geoprocessing. It also includes the legacy applications for ArcInfo Workstation.

Because ArcView, ArcEditor, and ArcInfo all share a common architecture, users working with any of these GIS desktops can share their work with other users. Maps, data, symbology, map layers, custom tools and interfaces, reports, metadata, and so on, can be accessed interchangeably in all three products. This means that you benefit from using a single architecture, minimizing the need to learn and deploy several different architectures.

New capabilities can be added to all seats through a series of ArcGIS Desktop extensions from ESRI and other organizations. Users can also develop their own custom extensions to ArcGIS Desktop by working with ArcObjects, the ArcGIS software component library. Users develop extensions and custom tools using standard Windows programming interfaces, such as COM and .NET.

ArcObjects is a framework that lets you create domain-specific components from other components. The ArcObjects components collaborate to serve every data management and map presentation function common to most GIS applications. ArcObjects provides an infrastructure for application customization that lets you concentrate on serving the specific needs of your clients.



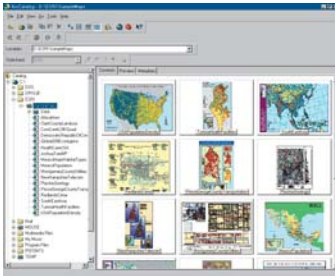
ArcMap is used for mapping and editing tasks as well as map-based analysis.

ArcMap

ArcMap is the central application in ArcGIS Desktop. It is the GIS application used for all map-based tasks, including cartography, map analysis, and editing. In this application, you work with maps. Maps have a page layout containing a geographic window, or a data frame, with a series of layers, legends, scalebars, North arrows, and other elements. ArcMap offers different ways to view a map's geographic data and layout views in which you can perform a broad range of advanced GIS tasks.

ArcCatalog

The ArcCatalog application helps you organize and manage all of your GIS information (maps, globes, datasets, models, metadata, services, and so on). It includes tools to:



ArcCatalog is used for managing your spatial data holdings, defining your geographic data schemas, and recording and viewing metadata.

- Browse and find geographic information.
- Record, view, and manage metadata.
- Define geodatabase schemas and designs.
- Administer an ArcGIS Server.
- Search for and discover GIS data on local networks and the Web.

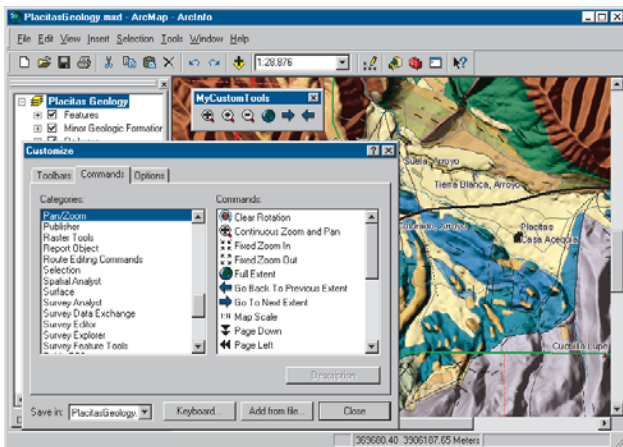
GIS users apply ArcCatalog to organize, find, and use GIS data as well as to document their data holdings using standards-based metadata. A GIS database administrator uses ArcCatalog to define and build geodatabases. A GIS server administrator uses ArcCatalog to administer the GIS server framework.

Customizing ArcGIS Desktop applications

You may want the ArcMap and ArcCatalog interfaces to reflect your own preferences and the way you work.

You can customize ArcMap and ArcCatalog in many ways. Here are some examples:

- Position toolbars in a specific area of the application.
- Group commands in a way that works best for you.
- Add new macros or load custom commands from another source.
- Always work with the same geographic data (via templates).



The Customize dialog box is used to create new toolbars and add or remove controls.

If you work in a larger organization, others may want you to develop a customized work environment for them. You can handle many customization tasks without writing a single line of code. In fact, you may be able to instruct others on how to use the customization environment to create the look and feel they want on their own. You can change or create toolbars, menus, shortcut keys, and so on, to help you do your work in the most efficient way. You can provide additional functionality by linking code you or others have written to menu commands or tools.

Several toolbars are provided with ArcMap and ArcCatalog, but you may want to create new toolbars to organize commands that you often use together or to contain buttons that run your custom scripts.

Writing VBA macros in ArcGIS applications

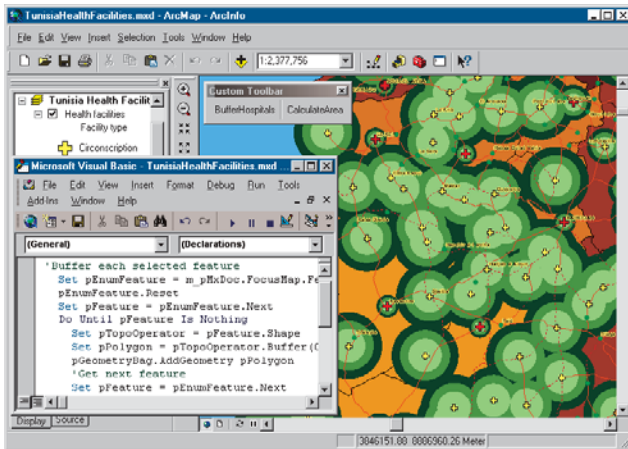
Both ArcMap and ArcCatalog come with Visual Basic® for Applications (VBA). VBA is not a standalone program; it's embedded in the applications. It provides an integrated programming environment, the Visual Basic Editor (VBE), which lets you write a Visual Basic (VB) macro, then debug and test it right away in ArcMap or ArcCatalog. A macro can integrate some or all of VB's functionality, such as using message boxes for input, with the extensive object library that ArcMap and ArcCatalog expose. The ESRI object libraries are always available to you in the VBA environment.

There is an easy way to create custom commands with VBA. You can create a new button, tool, combo box, or edit box (collectively called UIControls), then attach code to the control's events, such as what happens when you click a button. After you have created it, you can drag this new control onto a toolbar.

As mentioned, you should start development by using the VBA environment in one of the existing ArcGIS applications. VBA is a simple programming language with many utilities, such as design time code completion and the Object Browser that will help you assemble code quickly.

Here are more reasons to choose the VBA environment:

- It's fast and easy to create, test, and debug macros inside ArcMap and ArcCatalog.
- The standard ESRI type libraries are already referenced for you.
- Important global variables, such as the Application and Document, are available.
- It's simple to assemble UI forms using VBA and ActiveX® components.
- It's straightforward to integrate VBA code with new ArcObjects UIControls.
- It's relatively easy to migrate VBA code to VB ActiveX Dynamic Link Library (DLL) projects.
- Many code samples available in the help system are macros that can be cut, pasted, and run within the VBA environment.



A buffer command created in VBA

Writing custom components to extend the ArcGIS Desktop applications

You don't have to use VBA to create custom commands and toolbars—in fact, in some cases, your custom commands and toolbars may require you to use another development environment. You can create custom objects in any programming language that supports the Microsoft® Component Object Model. Custom commands or toolbars created outside VBA are often distributed as ActiveX DLLs. If you have created some custom commands and toolbars, or someone else has given you an ActiveX DLL containing custom commands and toolbars, you can easily add these objects to ArcMap or ArcCatalog. After adding a custom object to ArcMap or ArcCatalog, you can use it as you would any built-in command or toolbar.

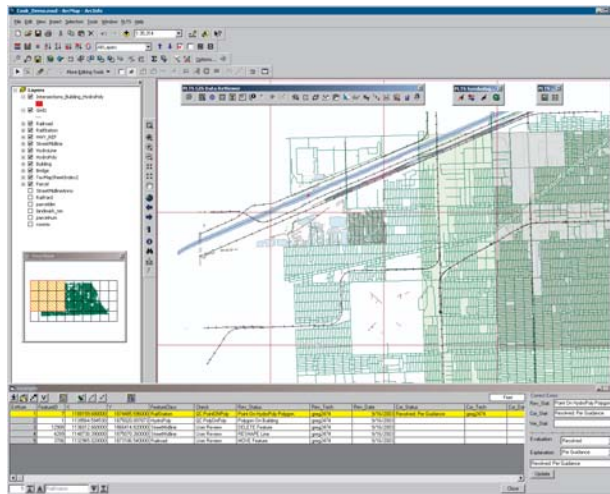
If you want to use a programming language other than VBA, or if you want to package ArcObjects functionality into a COM DLL, EXE, OCX, or .NET assembly, you will have to work outside the VBA development environment. This approach generally requires creating a project, referencing the ArcObjects type libraries required, implementing ArcObjects interfaces, adding code, then compiling the source into a binary file.

Following are some advantages of building custom components:

- They can be easily delivered to end users via custom setup programs.
- You can hide your code and logic in a binary file.
- You can extend and customize virtually every aspect of the ArcGIS technology.

Components can be broadly categorized into two areas of customization: those that reside at the application level, such as custom buttons, toolbars, windows, and extensions, and those that reside at the geodatabase level, such as custom feature class extensions and features. Some of these more advanced customizations cannot be accomplished through the VBA environment.

The Production Line Tool Set (PLTS) is an example of a complex application extension created with ArcObjects.



The *ArcGIS Desktop Developer Guide* is an introduction for anyone who wants to customize or extend the ArcGIS Desktop applications, such as ArcMap or ArcCatalog.

This guide will help you become a desktop developer by walking you through numerous VBA code samples and providing a problem-solving guide and developer scenarios. Although the samples documented in this guide may not solve your immediate problem, they will serve as a framework or template on which you can build a more specific or complex solution.

To serve the greatest base of developers, most of the code samples in this guide are written in VBA. As necessary, some code samples are written in Microsoft Visual Basic, VB .NET, or Visual C++®.

CHAPTER GUIDE

Chapter 1, 'Introducing ArcGIS Desktop development', gives you an overview of the ArcGIS 9 product family, the desktop developer framework, the types of customizations that can be made, and additional resources.

Chapter 2, 'ArcGIS software architecture', describes the underlying technology used to create the ArcGIS Desktop products, ArcObjects, and the software architecture common to the ArcGIS product family.

Chapter 3, 'Developing for ArcGIS Desktop applications', guides you through the basics of ArcGIS Desktop development. The chapter begins with some desktop application framework theory, then walks you through some small tutorials for toolbar customization and VBA macro writing. The chapter concludes with the theory and steps in creating custom components that extend the ArcGIS Desktop framework.

Chapter 4, 'Licensing and deployment', discusses license considerations when developing your customizations and how to package and deploy your customizations, including VBA macros and components, to other users.

Chapter 5, 'Developer scenarios', guides you through the creation of several types of components that can plug into the ArcGIS Desktop applications.

Appendix A, 'Developer environments', describes in more detail the Microsoft Component Object Model, the foundation for ArcObjects, and continues with a language reference guide for each API supported by ArcGIS. These guides discuss issues and considerations while developing ArcGIS with a particular API.

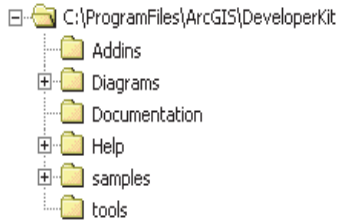
Appendix B, 'Reading the object model diagrams', discusses how to read and interpret the ArcGIS object model diagrams. Understanding the structure and relationships of ArcObjects represented in these diagrams is essential in ArcGIS development.

Appendix C, 'Illustrated code samples', shows numerous VBA code snippets for some common desktop applications. These are illustrated with components from the ArcGIS object model to help you understand the structure and relationships.

Appendix D, 'ArcObjects problem-solving guide,' presents a methodology to help you solve real-world ArcObjects programming tasks that customize or extend the ArcGIS Desktop applications.

Appendix E, 'UIControls', shows class and interface diagrams for UIControls, which are VBA-based commands with interfaces available only in VBA.

Appendix F, 'Bibliography', is not intended as a complete resource, but it does contain many of the everyday references that ESRI developers use when developing Visual Basic, Visual C++, Visual Studio .NET code, and ArcObjects.



A typical SDK installation

The following topics describe some of the additional resources available to you as a developer. These include books and guides and various help systems.

ArcGIS SOFTWARE DEVELOPER KIT

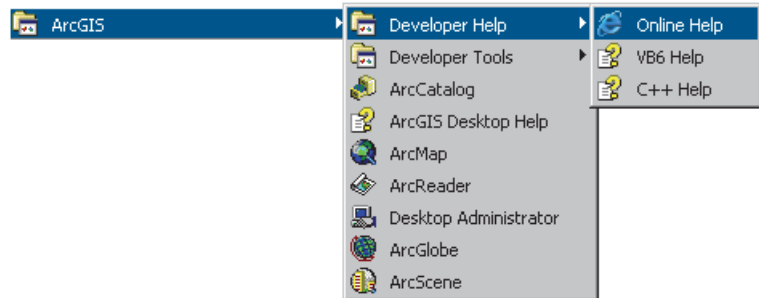
The ArcGIS SDK is a collection of diagrams, utilities, add-ins, samples, and documentation geared to help developers implement custom ArcGIS functionality.

ArcGIS Developer Help system

The ArcGIS Developer Help system is the gateway to all SDK documentation, including help for the add-ins, developer tools, and samples; in addition, it serves as the complete syntactical reference for all object libraries.

Each supported API has a version of the help system that works in concert with it. Regardless of the API you choose to use, you will see the appropriate library reference syntax and have a help system that is integrated with your development environment. For example, if you are a Visual Basic 6 developer, you will use ArcGISDevHelp.chm, which has the VB6 syntax and integrates with the VB6 integrated development environment (IDE), thereby providing F1 help support in the Code window.

The help systems reside in the DeveloperKit\Help folder but are typically launched from the Start menu or F1 Help in Visual Basic 6 and Visual Studio .NET 2003. The graphic below shows the Start menu options for opening the help systems.

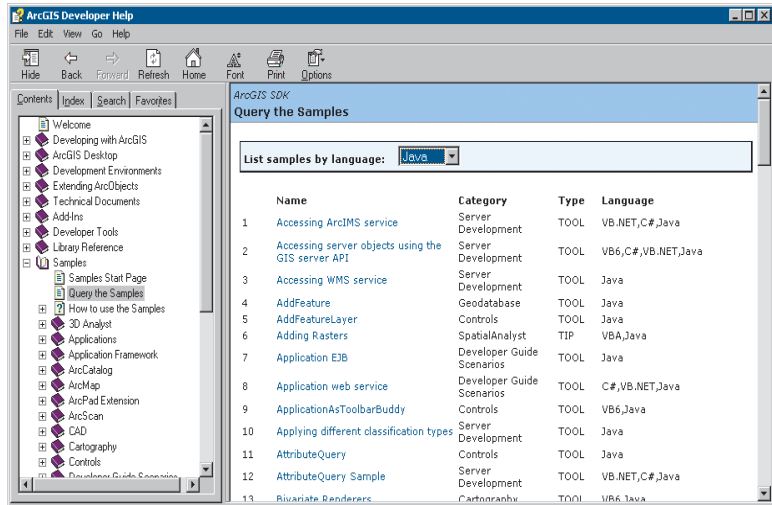


Samples

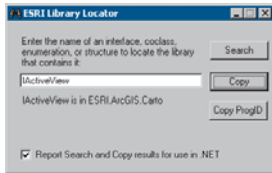
The ArcGIS developer kit contains more than 600 samples, many of which are written in several languages. The samples are described in the help system, and source code and project files are installed in the DeveloperKit\samples folder. The help system's table of contents for the samples section mirrors the samples directory structure.

The help system organizes samples by functionality. For example, all the geodatabase samples are grouped under Samples\Geodatabase. Most first-tier groupings are further subdivided. You can also find samples in the SDK using the 'Query the Samples' topic in the help system, which lists all the samples alphabetically; in addition, you can sort the list by language. For example, you can elect to only list the available VB6 samples.

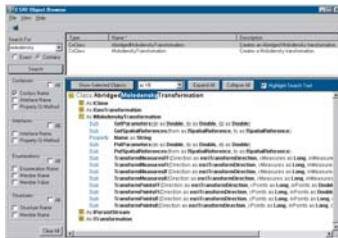
You can use the 'Query the Samples' topic in the help system to find specific samples that interest you.



Installing the sample source code and project files is an option in the Desktop Developer Kit install. The samples are installed under the ArcGIS\DeveloperKit\samples folder. If you don't have this folder on your computer, you can rerun the install program and check Samples under Developer Kit.



ESRI Library Locator



ESRI Object Browser

Developer tools

The ArcGIS developer tools are executables that ESRI has provided to facilitate your ArcObjects development. For example, if you are a Visual Basic 6 desktop developer, you will likely use the Categories.exe tool to register components in component categories.

The list below features some of the more important developer tools available with ArcGIS Desktop. Refer to the help system for more developer tool details and instructions.

- Component Categories Manager—Registers components within a specific component category
- Library Locator—Identifies an object library containing a specified interface, class, enumeration, or structure
- ESRI Object Browser—Lets you explore the structure of ArcObjects, overcoming certain limitations of other standard object browsers

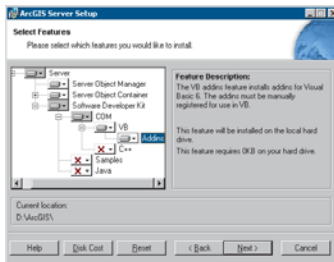
The developer tools are installed in the DeveloperKit\tools folder. There is one exception—the Component Category Manager is located in the ArcGIS\bin folder.

Add-ins

The ESRI add-ins automate some of the tasks performed by the software engineer when developing with ArcObjects, as well as provide tools that make debugging code easier. ESRI provides add-ins for the Visual Basic 6 IDE and the Visual Studio .NET IDE. Listed below are the add-ins available for these development environments.

Visual Basic 6

- ESRI Align Controls With Tab Index—Ensures control creation order matches tab index
- ESRI Automatic References—Automatically adds ArcGIS library references
- ESRI Code Converter—Converts projects from ArcGIS 8.x to ArcGIS 9.x
- ESRI Command Creation Wizard—Facilitates the creation of commands and tools
- ESRI Compile and Register—Aids in compiling components and registering these in desired component categories
- ESRI ErrorHandler Generator—Automates the generation of error handling code
- ESRI ErrorHandler Remover—Removes the error handlers from the source files
- ESRI Interface Implementer—Automatically stubs out implemented interfaces



Visual Basic 6 add-ins are only installed if you select them on the install.

Visual Studio .NET

- ESRI Component Category Registrar—Stubs out registration functions to enable self-component category registration
- ESRI .NET Code Converter—Converts projects from ArcGIS 8.x to ArcGIS 9.x
- ESRI GUID Generator—Inserts a Global Unique Identifier (GUID) attribute

The .NET add-ins are automatically installed during setup if a version of Visual Studio .NET 2003 is detected; the Visual Basic 6 add-ins are only installed if you select them on the install. Once installed the add-ins have to be registered manually using the .bat files in the Addins directory.

THE ARC GIS DEVELOPER DOCUMENTATION SERIES

This guide is part of the ArcGIS Developer documentation series.

The *ArcGIS Engine Developer Guide* provides information for developers who want to create applications based on ArcGIS Engine. ArcGIS Engine allows you to embed GIS functionality within other applications and create desktop-like applications using the supplied ArcGIS controls, such as MapControl, Toolbar, and PageLayout controls. ArcGIS Engine is also based on ArcObjects components and may be programmed through a number of APIs.

The *ArcGIS Server Administrator and Developer Guide* is for developers who will create ArcGIS Server applications and customizations. At the core of ArcGIS Server is a rich ArcObjects object library that can be consumed in Web applications and Web services to deliver advanced GIS functionality to a wide range of users who interact with the server through Web browsers and other thin client applications. ArcGIS Server may also be programmed through a number of APIs.

ARC GIS DEVELOPER ONLINE WEB SITE

ArcGIS Developer Online is the place to find the most up-to-date ArcGIS 9 developer information, including sample code, technical documents, object model diagrams, and the complete object library reference.

The Web site is a reflection of the ArcGIS Developer Help system, except it is online and, therefore, more current. The Web site has some additional features including an advanced search utility that enables you to control the scope of your searches. For example, you can create a search that only scans the library reference portion of the help system.

Visit the site at <http://arcgisdeveloperonline.esri.com>.





ESRI SUPPORT CENTER

The ESRI Support Center at <http://support.esri.com> contains software information, technical documents, samples, forums, and a knowledge base for all ArcGIS products.

ArcGIS developers can take advantage of the forums, knowledge base, and samples sections in particular to aid in development of their ArcGIS applications.

TRAINING

ESRI offers a number of instructor-led and Web-based training courses for the ArcGIS Desktop developer. These courses range from the introductory level for VBA to more advanced courses in component development with specific APIs.

For more information visit <http://www.esri.com> and click the Training and Events tab.

The ESRI Virtual Campus can also be found directly at <http://campus.esri.com/>.

2

ArcGIS software architecture

The architecture of ArcGIS has evolved over several releases of the technology to be a modular, scalable, cross-platform architecture implemented by a set of software components called ArcObjects.

This chapter focuses on the main themes of this evolution at ArcGIS 9 and introduces the reader to the libraries that comprise the ArcGIS system.

The ArcGIS software architecture supports a number of products, each with its unique set of requirements. ArcObjects, the components that make up ArcGIS, are designed and built to support this. This chapter introduces ArcObjects.

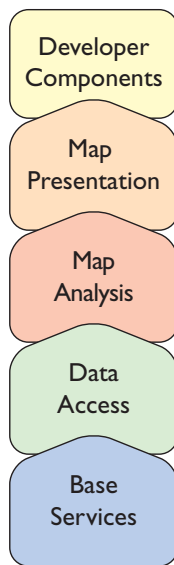
ArcObjects is a set of platform-independent software components, written in C++, that provides services to support GIS applications on the desktop in the form of thick and thin clients and on the server.

As stated, the language chosen to develop ArcObjects was C++; in addition to this language, ArcObjects makes use of the Microsoft Component Object Model. COM is often thought of as simply specifying how objects are implemented and built in memory and how these objects communicate with one another. While this is true, COM also provides a solid infrastructure at the operating system level to support any system built using COM. On Microsoft Windows operating systems, the COM infrastructure is built directly into the operating system. For operating systems other than Microsoft Windows, this infrastructure must be provided for the ArcObjects system to function.

For a detailed explanation of COM see the COM section of Appendix A, 'Developer environments'.

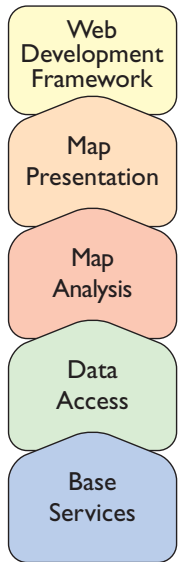
Not all ArcObjects components are created equally. The requirements of a particular object, in addition to its basic functionality, vary depending on the final end use of the object. This end use broadly falls into one of the three ArcGIS product families:

- **ArcGIS Engine**—Use of the object is within a custom application. Objects within the Engine must support a variety of uses; simple map dialog boxes, multithreaded servers, and complex Windows desktop applications are all possible uses of Engine objects. The dependencies of the objects within the Engine must be well understood. The impact of adding dependencies external to ArcObjects must be carefully reviewed, since new dependencies may introduce undesirable complexity to the installation of the application built on the Engine.
- **ArcGIS Server**—The object is used within the server framework, where clients of the object are most often remote. The remoteness of the client can vary from local, possibly on the same machine or network, to distant, where clients can be on the Internet. Objects running within the server must be scalable and thread safe to allow execution in a multithreaded environment.
- **ArcGIS Desktop**—Use of the object is within one of the ArcGIS Desktop applications. ArcGIS Desktop applications have a rich user experience, with applications containing many dialog boxes and property pages that allow end users to work effectively with the functionality of the object. Objects that contain properties that are to be modified by users of these applications should have property pages created for these properties. Not all objects require property pages.

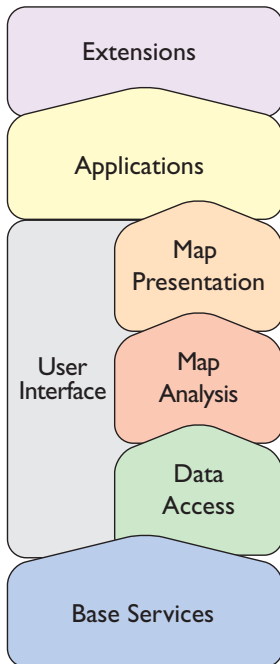


ArcGIS Engine

Many of the ArcObjects components that make up ArcGIS are used within all three of the ArcGIS products. The product diagrams on these pages show that the objects within the broad categories of base services, data access, map analysis, and map presentation are contained in all three products. These four categories contain the majority of the GIS functionality exposed to developers and users in ArcGIS.



ArcGIS Server



ArcGIS Desktop

This commonality of function among all the products is important for developers to understand, since it means that when working in a particular category, much of the development effort can be transferred between the ArcGIS products with little change to the software. After all, this is exactly how the ArcGIS architecture is developed. Code reuse is a major benefit of building a modular architecture, but code reuse does not simply come from creating components in a modular fashion.

The ArcGIS architecture provides rich functionality to the developer, but it is not a closed system. The ArcGIS architecture is extendable by developers external to ESRI. Developers have been extending the architecture for a number of years, and the ArcGIS 9 architecture is no different; it, too, can be extended. However, ArcGIS 9 introduces many new possibilities for the use of objects created by ESRI and you. To realize these possibilities, components must meet additional requirements to ensure that they will operate successfully within this new and significantly enhanced ArcGIS system. Some of the changes from ArcGIS 8 to ArcGIS 9 appear superficial, an example being the breakup of the type libraries into smaller libraries. That, along with the fact that the objects with their methods and properties that were present at 8.3 are still available at 9, masks the fact that internally ArcObjects has undergone some significant work.

The main focus of the changes made to the ArcGIS architecture at 9 revolves around four key concepts:

- **Modularity**—A modular system where the dependencies between components are well-defined in a flexible system.
- **Scalability**—ArcObjects must perform well in all intended operating environments, from single user desktop applications to multiuser/multithreaded server applications.
- **Multiple Platform Support**—ArcObjects for the Engine and Server should be capable of running on multiple computing platforms.
- **Compatibility**—ArcObjects 9 should remain equivalent, both functionally and programmatically, to ArcObjects 8.3.

MODULARITY

The esriCore object library, shipped as part of ArcGIS 8.3, effectively packaged all of ArcObjects into one large block of GIS functionality; there was no distinction between components. The ArcObjects components were divided into smaller groups of components, these groups being packaged in DLLs. The one large library, while simplifying the task of development for external developers, prevented the software from being modular. Adding the type information to all the DLLs, while possible, would have greatly increased the burden on external developers and, hence, was not an option. In addition, the DLL structure did not always reflect the best modular breakup of software components based on functionality and dependency.

There is always a trade-off in performance and manageability when considering architecture modularity. For each criteria, thought is given to the end use and the modularity required for support. For example, the system could be divided into many small DLLs with only a few objects in each. Although this provides a flexible system for deployment options, at minimum memory requirements, it

ESRI has developed a modular architecture for ArcGIS 9 by a process of analyzing features and functions and matching those with end user requirements and deployment options based on the three ArcGIS product families. Developers who have extended the ArcGIS 8 architecture with custom components are encouraged to go through the same process to restructure their source code into similar modular structures.

An obvious functionality split to make is user interface and nonuser interface code. UI libraries tend to be included only with the ArcGIS Desktop products.

would affect performance due to the large number of DLLs being loaded and unloaded. Conversely, one large DLL containing all objects is not a suitable solution either. Knowing the requirements of the components allows them to be effectively packaged into DLLs.

The ArcGIS 9 architecture is divided into a number of libraries. It is possible for a library to have any number of DLLs and EXEs within it. The requirements that components must meet to be within a library are well-defined. For instance, a library, such as `esriGeometry` (from the base services set of modules), has the requirements of being thread safe, scalable, without user interface components, and deployable on a number of computing platforms. These requirements are different from libraries, such as `esriArcMap` (from the applications category), which does have user interface components and is a Windows-only library.

All the components in the library will share the same set of requirements placed on the library. It is not possible to subdivide a library into smaller pieces for distribution. The library defines the namespace for all components within it and is seen in a form suitable for your chosen API.

- Type Library—COM
- .NET Interop Assembly—.NET
- Java Package—Java
- Header File—C++

SCALABILITY

The `ArcObjects` components within ArcGIS Engine and ArcGIS Server must be scalable. Engine objects are scalable because they can be used in many different types of applications; some require scalability, while others do not. Server objects are required to be scalable to ensure that the server can handle many users connecting to it, and as the configuration of the server grows, so does the performance of the `ArcObjects` components running on the server.

The scalability of a system is achieved using a number of variables involving the hardware and software of the system. In this regard, `ArcObjects` supports scalability with the effective use of memory within the objects and the ability to execute the objects within multithreaded processes.

There are two considerations when multithreaded applications are discussed: thread safety and scalability. It is important for all objects to be thread safe, but simply having thread-safe objects does not automatically mean that creating multithreaded applications is straightforward or that the resulting application will provide vastly improved performance.

The `ArcObjects` components contained in the base services, data access, map analysis, and map presentation categories are all thread safe. This means that application developers can use them in multithreaded applications; however, programmers must still write multithreaded code in such a way as to avoid application failures due to deadlock situations and so forth.

In addition to the `ArcObjects` components being thread safe for ArcGIS 9, the apartment threading model used by `ArcObjects` was analyzed to ensure that

Thread safety refers to concurrent object access from multiple threads.

ArcObjects could be run efficiently in a multithreaded process. A model referred to as “Threads in Isolation” was used to ensure that the ArcObjects architecture is used efficiently.

This model works by reducing cross-thread communication to an absolute minimum or, better still, removing it entirely. For this to work, the singleton objects at ArcGIS 9 were changed to be singletons per thread and not singletons per process. The resource overhead of hosting multiple singletons in a process was outweighed by the performance gain of stopping cross-thread communication where the singleton object is created in one thread (normally the Main single-threaded apartment [STA]) and the accessing object is in another thread.

ArcGIS is an extensible system, and for the Threads in Isolation model to work, all singleton objects must adhere to this rule. If you are creating singleton objects as part of your development, you must ensure that these objects adhere to the rule.

The classic singleton per process model means that all threads of an application will still access the main thread hosting the singleton objects. This effectively reduces the application to a single-threaded application.

MULTIPLE PLATFORM SUPPORT

As stated earlier, ArcObjects components are C++ objects, meaning that any computing platform with a C++ compiler can potentially be a platform for ArcObjects. In addition to the C++ compiler, the platform must also support some basic services required by ArcObjects.

Although many of the platform differences do not affect the way in which ArcObjects components are developed, there are areas where differences do affect the way code is developed. The byte order of different computing architectures varies between little endian and big endian. This is most readily seen when objects read and write data to disk. Data written using one computing platform will not be compatible if read using another platform, unless some decoding work is performed. All the ArcGIS Engine and ArcGIS Server objects support this multiple platform persistence model. ArcObjects components always persist themselves using the little endian model; when the objects read persisted data, it is converted to the appropriate native byte order. In addition to the byte order differences, there are other areas of functionality that differ between platforms; the directory structure, for example, uses different separators for Windows and UNIX®—“\” and “/”, respectively. Another example is the platform-specific areas of functionality, such as OLE DB.

Microsoft Windows is a little endian platform, while Sun™ Solaris™ is a big endian platform.

COMPATIBILITY

Maintaining compatibility of the ArcGIS system between releases is important to ensure that external developers are not burdened with changing their code to work with the latest release of the technology. Maintaining compatibility at the object level was a primary goal of the ArcGIS 9 development effort. Although this object-level compatibility has been maintained, there are some changes between the ArcGIS 8 and ArcGIS 9 architectures that will affect developers, mainly related to the compilation of the software.

Although the changes required for software created for use with ArcGIS 8 to work with ArcGIS 9 are minimal, it is important to understand that to realize any existing investment in the ArcObjects architecture at ArcGIS 9, you must review your developments with respect to ArcGIS Engine, ArcGIS Server, and ArcGIS Desktop.

While the aim of ArcGIS releases is to limit the change in the APIs, developers should still test their software thoroughly with later releases.

ESRI understands the importance of a unified software architecture and has made numerous changes for ArcGIS 9 so the investment in ArcObjects can be realized on multiple products. If you have been involved in creating extensions to the ArcGIS architecture for ArcGIS 8, you should think about how the new ArcGIS 9 architecture affects the way your components are implemented.

The functionality of ArcObjects can be accessed using four APIs: COM, .NET, Java, and C++. The choice of which API to use is not a simple one and will depend on a number of factors, including the ArcGIS product that you are developing with, the end user functionality that you are developing, and your development experience with particular languages. ArcGIS Desktop supports the following APIs:

- COM—Any COM-compliant language (for example, Visual Basic, Visual C++, Delphi) can be used with this API.
- .NET—Visual Basic .NET and C# are supported by this API.

When working with ArcObjects, developers can consume functionality exposed by ArcObjects or extend the functionality of ArcObjects with their own components. When referring to these APIs, there are differences with respect to consuming and extending the ArcObjects architecture.

CONSUMING THE API

The APIs support consuming the functionality of the ArcObjects components; however, not all interfaces implemented by ArcObjects are supported on all platforms. In some cases, interfaces make use of data types that are not compatible with an API. In situations like this, an alternative implementation of the interface is provided for developers to use. The naming convention of a “GEN” postfix on the interface name is used to signify this kind of interface; IFoo would have an IFooGEN interface. This alternative interface is usable by all APIs; however, if the nongeneric interface is supported by the API, it is possible to continue to use the API-specific interface.

Since ArcObjects are developed in C++, there are some cases in which data types compatible with C++ have been used for performance reasons. These performance considerations mostly affect the internals of ArcObjects; hence, using one of the generic interfaces should not adversely affect performance or your ArcObjects developments.

EXTENDING THE API

Extending ArcObjects entails creating your own objects and adding them to the ArcObjects architecture. ArcObjects is written to be extensible in almost all areas. Support for extending the architecture varies among the APIs and, in some cases, varies among languages of an API.

The COM API provides the most possibilities for extending the system. The limitation within this API is with the Visual Basic language. Visual Basic does not support the implementation of interfaces that have one or more of the following characteristics:

- The interface inherits from an interface other than *IUnknown* or *IDispatch*. For example, *ICurve*, which inherits from *IGeometry*, cannot be implemented in VB for this reason.
- Method names on an interface start with an underscore (“_”). You will not find functions beginning with “_” in ArcObjects.
- A parameter of a method uses a data type not supported by Visual Basic. *IActiveView* cannot be implemented in Visual Basic for this reason.

In addition to the limitations on the interfaces supported by VB, the binary reuse technique of COM aggregation is not supported by VB. This means that certain parts of the architecture cannot be extended; Custom Features is one such example. In reality, the above limitations of Visual Basic have little effect on the

vast majority of developers, since the percentage of ArcObjects affected is small, and for this small percentage, it is unlikely that developers will have a need to extend the architecture. Other COM languages, such as Visual C++, do not have any of these limitations.

The .NET API supports extending ArcObjects fully, with the one exception being interfaces that make use of non-OLE automation-compliant data types (see the table below for a complete list of all OLE automation-compliant data types).

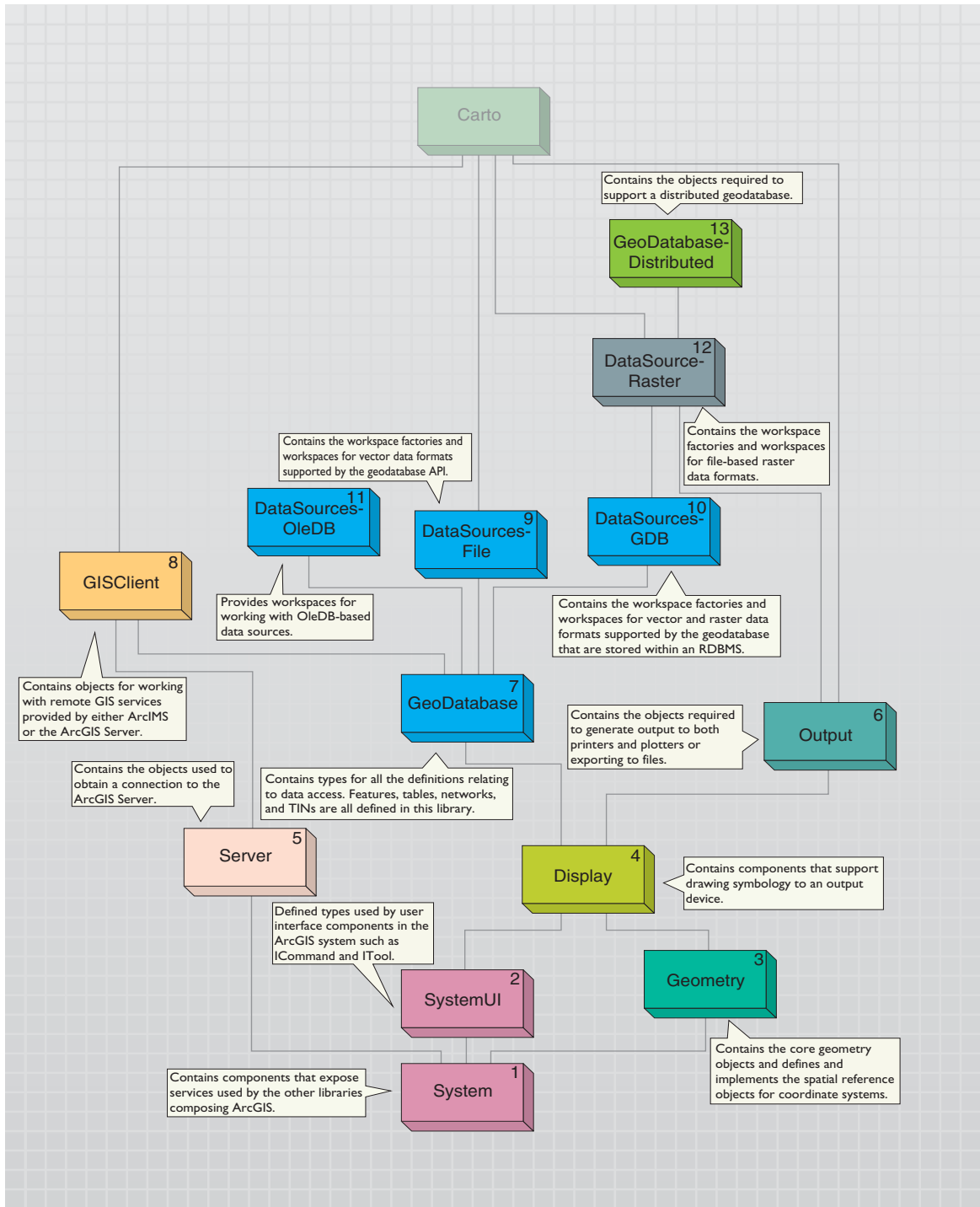
The majority of differences between the API's support for ArcObjects revolves around data types. All APIs fully support the automation-compliant data types shown on the right. Differences occur with data types that are not OLE automation compliant.

Type	Description
Boolean	Data item that can have the value True or False.
unsigned char	8-bit unsigned data item.
double	64-bit IEEE floating-point number.
float	32-bit IEEE floating-point number.
int	Signed integer, whose size is system dependent.
long	32-bit signed integer.
short	16-bit signed integer.
BSTR	Length-prefixed string.
CURRENCY	8-byte, fixed-point number.
DATE	64-bit, floating-point fractional number of days since Dec 30, 1899.
SCODE	For 16-bit systems - Built-in error that corresponds to VT_ERROR.
Typedef enum myenum	Signed integer, whose size is system dependent.
Interface IDispatch *	Pointer to the IDispatch interface.
Interface IUnknown *	Pointer to an interface that does not derive from IDispatch.
dispinterface Typename *	Pointer to an interface derived from IDispatch.
Coclass Typename *	Pointer to a coclass name (VT_UNKOWN).
[oleautomation] interface Typename *	Pointer to an interface that derives from IDispatch.
SAFEARRAY (Typename)	Typename is any of the above types. Array of these types.
Typename*	Typename is any of the above types. Pointer to a type.
Decimal	96-bit unsigned binary integer scaled by a variable power of 10. A decimal data type that provides a size and scale for a number (as in coordinates).

OLE automation data types

The ArcGIS Desktop applications are rich, professional GIS applications with a lot of functionality, but if viewed simply, the applications can be broken down into a series of toolbars, along with a table of contents (TOC) and map viewing area. The desktop applications are all extended by adding new commands and tools. In a similar way, developers can build applications with rich functionality using any of the four ArcGIS Engine APIs.

The COM and .NET APIs are only supported on the Microsoft Windows platform.



For a comprehensive discussion on each library the library overviews, refer to the ArcGIS Developer Help system.

The libraries contained within the ArcGIS Engine are also available in ArcGIS Desktop and are summarized below. The diagrams that accompany this section indicate the library architecture of the ArcGIS Engine. Understanding the library structure, dependencies, and basic functionality will help you as a developer navigate through the components of ArcGIS Engine.

The libraries are discussed in dependency order. The diagrams show this with sequential numbers in the upper right corner of the library block. For example, System, as the library at the base of the ArcGIS architecture, is numbered one, while GeoDatabase, numbered seven, depends on the six libraries that precede it in the diagram—System, SystemUI, Geometry, Display, Server, and Output.

SYSTEM

The System library is the lowest level library in the ArcGIS architecture. The library contains components that expose services used by the other libraries comprising ArcGIS. There are a number of interfaces defined within the System library that can be implemented by the developer. The *AoInitializer* object is defined in System; all developers must use this object to initialize and uninitialize the ArcGIS Engine in applications that make use of Engine functionality. The developer does not extend this library but can extend the ArcGIS system by implementing interfaces contained within this library.

SYSTEMUI

The SystemUI library contains the interface definitions for user interface components that can be extended within the ArcGIS Engine. These include the *ICommand*, *ITool*, and *IToolControl* interfaces. The developer uses these interfaces to extend the UI components that the ArcGIS Engine developer components use. The objects contained within this library are utility objects available to the developer to simplify some user interface developments. The developer does not extend this library but can extend the ArcGIS system by implementing interfaces contained within this library.

GEOMETRY

The Geometry library handles the geometry, or shape, of features stored in feature classes or other graphical elements. The fundamental geometry objects with which most users will interact are *Point*, *MultiPoint*, *Polyline*, and *Polygon*. Beside those top-level entities are geometries that serve as building blocks for *Polylines* and *Polygons*. Those are the primitives that compose the geometries. They are *Segments*, *Paths*, and *Rings*. *Polylines* and *Polygons* are composed of a sequence of connected *Segments* that form a *Path*. A *Segment* consists of two distinguished points, the start and the endpoint, and an element type that defines the curve from beginning to end. The kinds of segments are *CircularArc*, *Line*, *EllipticArc*, and *BezierCurve*. All geometry objects can have Z, M, and IDs associated with their vertices. The fundamental geometry objects all support geometric operations, such as *Buffer* and *Clip*. The geometry primitives are not meant to be extended by developers.

Entities within a GIS refer to real-world features; the location of these real-world features is defined by a geometry along with a spatial reference. Spatial

Knowing the library dependency order is important, since it affects the way in which developers interact with the libraries as they develop software. For example, C++ developers must include the type libraries in the library dependency order to ensure correct compilation. Understanding the dependencies also helps when deploying your developments.

reference objects for both projected and geographic coordinate systems are included in the Geometry library. Developers can extend the spatial reference system by adding new spatial references and projections between spatial references.

DISPLAY

The Display library contains objects used for the display of GIS data. In addition to the main display objects responsible for the actual output of the image, the library contains objects that represent symbols and colors used to control the properties of entities drawn on the display. The library also contains objects that provide the user with visual feedback when interacting with the display. Developers most often interact with the display through a view like the ones provided by the *Map* or *PageLayout* objects. All parts of the library can be extended, with the commonly extended areas being symbols, colors, and display feedbacks.

SERVER

The Server library contains objects that allow you to connect and work with ArcGIS Servers. Developers gain access to an ArcGIS Server using the *GISServerConnection* object. The *GISServerConnection* object gives access to the *ServerObjectManager*. Using this object a developer works with *ServerContext* objects to manipulate ArcObjects running on the server. The Server library is not extended by developers. Developers can also use the GISClient library when interacting with the ArcGIS Server.

OUTPUT

The Output library is used to create graphical output to devices, such as printers and plotters, and hardcopy formats, such as enhanced metafiles and raster image formats (JPG, BMP, and so forth). The developer uses the objects in the library with other parts of the ArcGIS system to create graphical output. Commonly, these would be objects in the Display and Carto libraries. Developers can extend the output library for custom devices and export formats.

GEODATABASE

The GeoDatabase library provides the programming API for the geodatabase. The geodatabase is a repository of geographic data built on standard industry relational and object relational database technology. The objects within the library provide a unified programming model for all supported data sources within ArcGIS. The GeoDatabase library defines many of the interfaces that are implemented by data source providers higher in the architecture. The geodatabase can be extended by developers to support specialized types of data objects (Features, Classes, and so forth); in addition, it can have custom vector data sources added using the *PlugInDataSource* objects. The native data types supported by the geodatabase cannot be extended.

GISCLIENT

The GISClient library allows developers to consume Web services; these Web services can be provided by ArcIMS and ArcGIS Server. The library includes objects for connecting to GIS servers to make use of Web services. There is

support for ArcIMS Image Services and Feature Services. The library provides a common programming model for working with ArcGIS Server objects in a stateless manner, either directly or through a Web service catalog. ArcObjects running on the ArcGIS Server is not accessible through the *GISClient* interface. To gain direct access to ArcObjects running on the server, you should use functionality in the Server library.

DATASOURCESFILE

The DataSourcesFile library contains the implementation of the GeoDatabase API for file-based data sources. These file-based data sources include shapefile, coverage, TIN, CAD, SDC, ArcGIS StreetMap™, and VPF. The DataSourcesFile library is not extended by developers.

DATASOURCESGDB

The DataSourcesGDB library contains the implementation of the GeoDatabase API for the database data sources. These data sources include Microsoft Access and SDE® software-supported RDBMSs. The DataSourcesGDB library is not extended by developers.

DATASOURCESOLEDB

The DataSourcesOleDB library contains the implementation of the GeoDatabase API for the Microsoft OLE DB data sources. This library is only available on the Microsoft Windows operating system. These data sources include any OLE DB-supported data provider and text file workspaces. The DataSourcesOleDB library is not extended by developers.

Raster Data Objects (RDO) is a COM API that provides display and analysis support for file-based raster data.

DATASOURCESRASTER

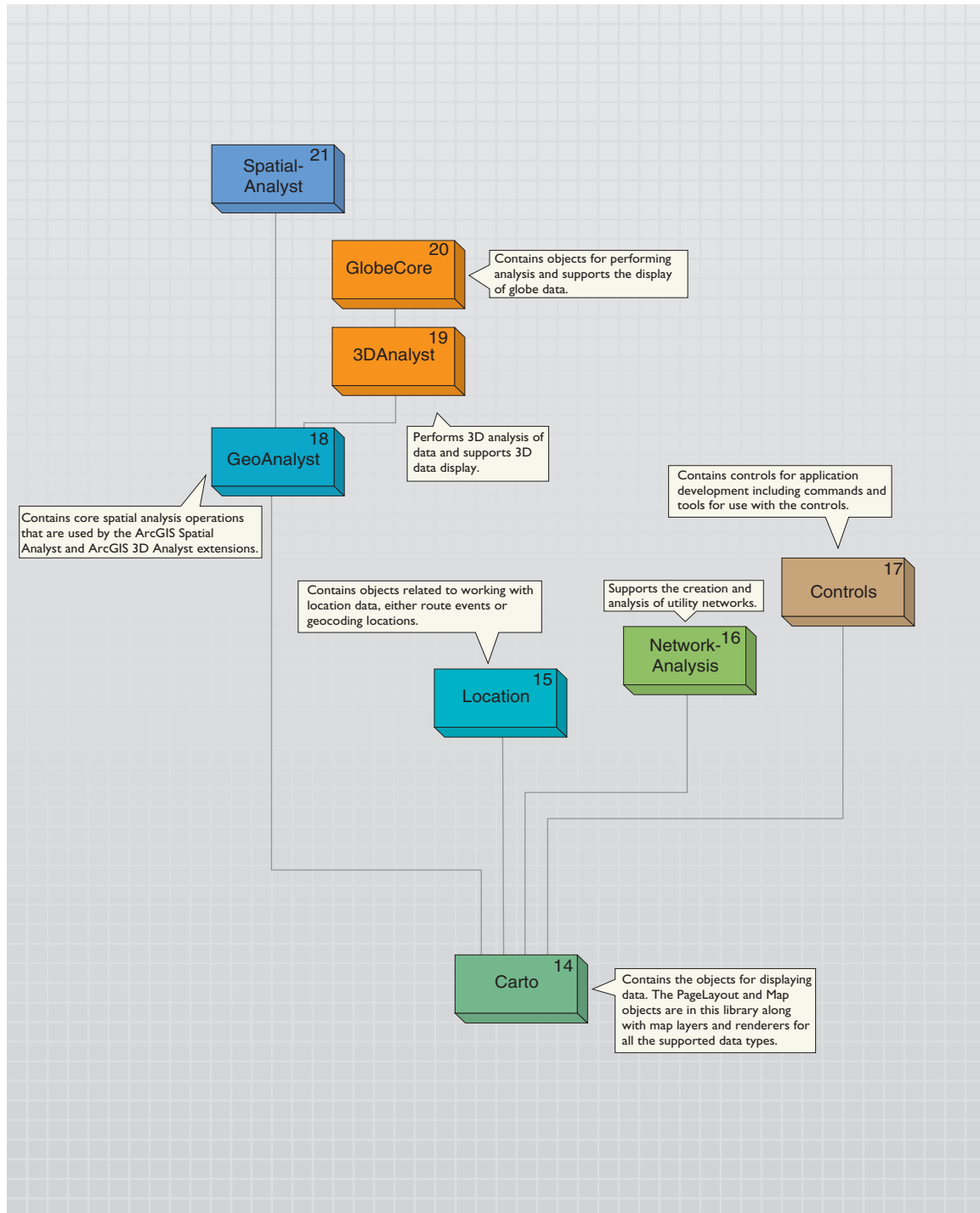
The DataSourcesRaster library contains the implementation of the GeoDatabase API for the Raster data sources. These data sources include SDE software-supported RDBMSs, along with all the supported RDO raster file formats. Developers do not extend this library when support for new raster formats is required; rather, they extend RDO. The DataSourcesRaster library is not extended by developers.

GEODATABASEDISTRIBUTED

The GeoDatabaseDistributed library supports distributed access to an enterprise geodatabase by providing tools for importing data into and exporting data out of a geodatabase. The GeoDatabaseDistributed library is not extended by developers.

CARTO

The Carto library supports the creation and display of maps; these maps can consist of data in one map or a page with many maps and associated marginalia. The *PageLayout* object is a container for hosting one or more maps and their associated marginalia: North arrows, legends, scalebars, and so forth. The *Map* object is a container of layers. The *Map* object has properties that operate on all layers within the map: spatial reference, map scale, and so forth, along with methods that manipulate the map's layers. There are many different types of



The ArcGIS Server uses the *MapServer* object for its *MapService*.

layers that can be added to a map. Different data sources often have an associated layer responsible for displaying the data on the map; vector features are handled by the *FeatureLayer* object, raster data by the *RasterLayer*, TIN data by the *TinLayer*, and so forth. Layers can, if required, handle all the drawing operations for their associated data, but it is more common for layers to have an associated *Renderer* object. The properties of the *Renderer* object control how the data is displayed in the map. Renderers commonly use symbols from the *Display* library for the actual drawing; the renderer simply matches a particular symbol with the properties of the entity that is to be drawn. A *Map*, along with a *PageLayout*, can contain elements. An element has geometry to define its location on the map or page, along with behavior that controls the display of the element. There are elements for basic shapes, text labels, complex marginalia, and so forth. The *Carto* library also contains support for map annotation and dynamic labeling.

Although developers can directly make use of the *Map* or *PageLayout* objects in their applications, it is more common for developers to use a higher-level object, such as the *MapControl*, *PageLayoutControl*, or *ArcGIS Application*. These higher-level objects simplify some tasks, although they always provide access to the lower-level *Map* and *PageLayout* objects, allowing the developer fine control of the objects.

The *Map* and *PageLayout* objects are not the only objects in *Carto* that expose the behavior of *Map* and *Page* drawing. The *MxdServer* and *MapServer* objects both support the rendering of *Maps* and *Pages*, but instead of rendering to a window, these objects render directly to a file.

Using the *MapDocument* object, developers can persist the state of the *Map* and *PageLayout* within a *Map Document (MXD)*, which can be used in *ArcMap*, or one of the *ArcGIS* controls.

The *Carto* library is commonly extended in a number of areas. Custom renderers, layers, and so forth, are common. A custom layer is often the easiest method of adding custom data support to a mapping application.

LOCATION

The *Location* library contains objects that support geocoding and working with route events. The geocoding functionality can be accessed through fine-grained objects for full control, or the *GeocodeServer* objects offer a simplified API. Developers can create their own geocoding objects. The linear referencing functionality provides objects for adding events to linear features and rendering these events using a variety of drawing options. The developer can extend the linear reference functionality.

NETWORK ANALYSIS

The *NetworkAnalysis* library provides objects for populating a geodatabase with network data and objects to analyze the network when it is loaded in the geodatabase. Developers can extend this library to support custom network tracing. The library is meant to work with utility networks—gas lines, electricity supply lines, and so forth.

The contents of the Map and PageLayout controls can be specified programmatically, or they can load Map Documents.

The ReaderControl only supports Published Map Files (PMF).

ArcGIS Engine comes with more than 150 commands.

CONTROLS

The Controls library is used by developers to build or extend applications with ArcGIS functionality. The ArcGIS Controls simplify the development process by encapsulating ArcObjects and providing a coarser-grained API. Although the controls encapsulate the fine-grained ArcObjects, they do not restrict access to them. The MapControl and PageLayoutControl encapsulate the Carto library's Map and PageLayout objects, respectively. The ReaderControl encapsulates both the Map and PageLayout objects and provides a simplified API when working with the control. If the map publisher has granted permission, the developer can access the internal objects in a similar way to the Map and PageLayout controls. The library also contains the TOCControl that implements a table of contents and a ToolbarControl for hosting commands and tools that work with a suitable control.

Developers extend the Controls library by creating their own commands and tools for use with the controls. To support this the library has the HookHelper object. This object makes it straightforward to create a command that works with any of the controls, in addition to ArcGIS applications, such as ArcMap.

GEOANALYST

The GeoAnalyst library contains objects that support core spatial analysis functions. These functions are used within both the ArcGIS Spatial Analyst and ArcGIS 3D Analyst™ libraries. Developers can extend the library by creating a new type of raster operation. An ArcGIS Spatial Analyst or 3D Analyst license is required to make use of the objects in this library.

3DANALYST

The 3DAnalyst library contains objects for working with three-dimensional scenes in a similar way that the Carto library contains objects for working with two-dimensional maps. The Scene object is one of the main objects of the library, since it is the container for data similar to the Map object. The Camera and Target objects specify how the scene is viewed regarding the positioning of the features relative to the observer. A scene consists of one or more layers; these layers specify the data in the scene and how the data is drawn.

The 3DAnalyst library has a developer control along with a set of commands and tools to use with this control. This control can be used in conjunction with the objects in the Controls library. It is not common for developers to extend this library beyond the creation of commands and tools. A 3D Analyst license is required to work with objects in this library.

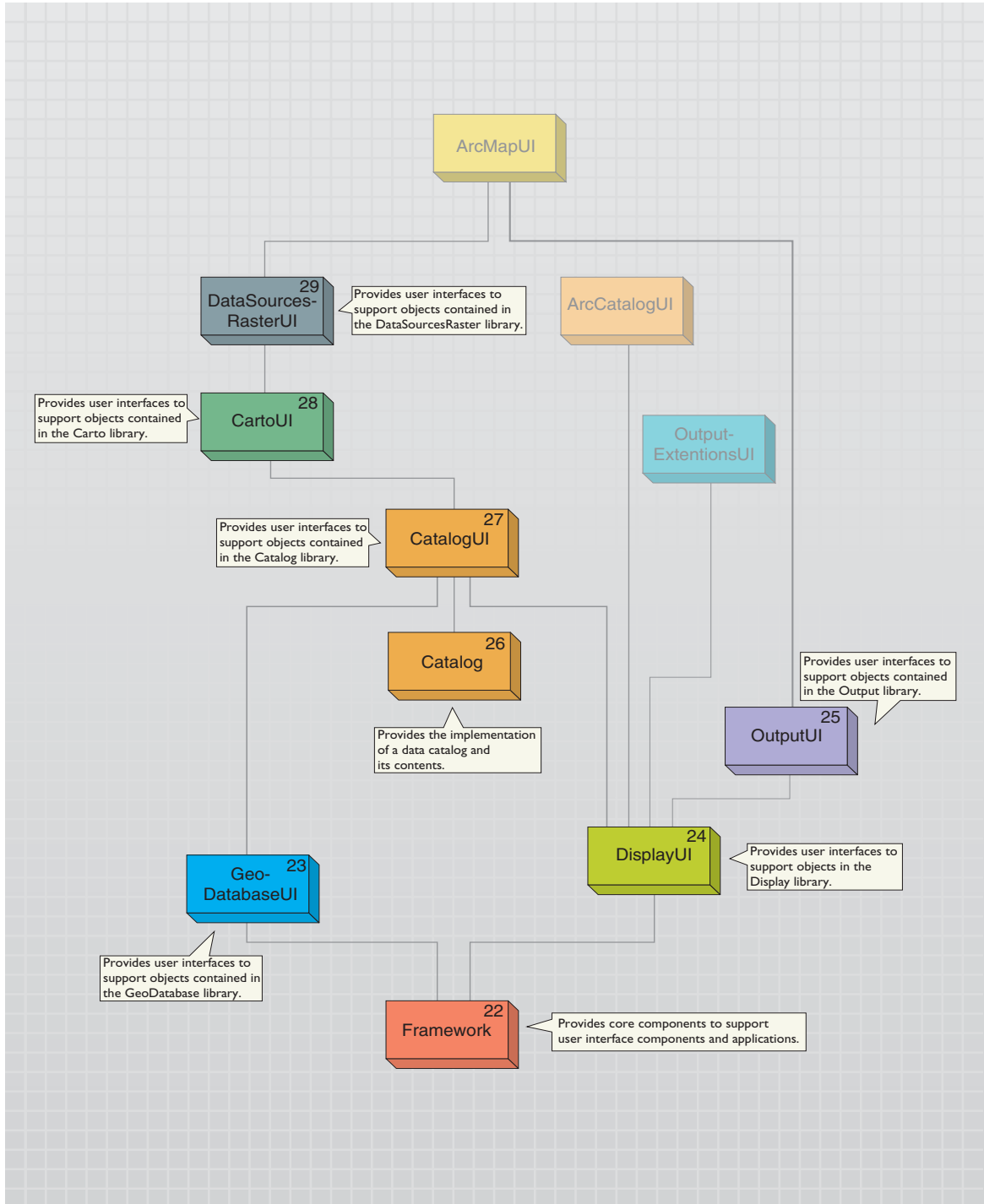
GLOBECORE

The GlobeCore library contains objects for working with globe data similar to the way that the Carto library contains objects for working with two-dimensional maps. The Globe object is one of the main objects of the library, since it is the container for data similar to the Map object. The GlobeCamera object specifies how the Globe is viewed regarding the positioning of the globe relative to the observer. The Globe can have one or more layers; these layers specify the data on the Globe and how the data is drawn.

The GlobeCore library has a developer control along with a set of commands and tools to use with this control. This control can be used in conjunction with the objects in the Controls library. It is not common for developers to extend this library beyond the creation of commands and tools. A 3D Analyst license is required to work with objects in this library.

SPATIALANALYST

The SpatialAnalyst library contains objects for performing spatial analysis on raster and vector data. Developers most commonly consume the objects within this library and do not extend it. An ArcGIS Spatial Analyst license is required to work with objects in this library.



The libraries contained within ArcGIS Desktop are summarized below. The diagrams that accompany this section indicate the library architecture of the ArcGIS Desktop applications. Understanding the library structure, their dependencies, and basic functionality will help you as a developer navigate through the components of the ArcGIS Desktop applications. The libraries are discussed in dependency order. The diagrams show this with a number in the upper right corner of the library block.

The modular architecture of ArcObjects makes a split between UI and non-UI code. The objects that have the GIS functionality do not have UI in the same DLL. The UI is provided by a complementary UI library. This UI library uses framework components, as well as components from its corresponding non-UI library.

FRAMEWORK

The Framework library provides core components and software interfaces to support user interface components and the ArcGIS applications. A number of the objects in the Framework library are used internally by the ArcGIS applications to support their customization environment. There are a number of helper objects in Framework that developers can use when creating user interfaces for inclusion in one of the ArcGIS applications—ComPropertySheet, ModelessFrame, and MouseCursor are three examples—along with a set of dialog boxes that support common UI operations within an ArcGIS application—ColorSelector and NumberDialog are two commonly used dialog boxes. The Framework library defines the software interfaces that developers use when creating user interfaces for extending the ArcGIS system using property pages and dockable windows. The Framework library is not extended by the developer, but by implementing interfaces defined in the library, developers can extend the ArcGIS architecture with UI components.

GEODATABASEUI

The GeoDatabaseUI library provides user interfaces, including property pages, to support objects contained in the GeoDatabase library. The library supports a number of dialog boxes that developers can use; TableView, Calculator, and the version dialog boxes are all defined in the library. It is not common for developers to extend this library.

DISPLAYUI

The DisplayUI library provides user interfaces, including property pages, to support objects contained in the Display library. All the symbols defined in the Display library have their property pages defined in this library. There are dialog boxes to manage styles and symbols in this library. Developers extend this library when they create UIs for corresponding components they have created in the Display library.

OUTPUTUI

The OutputUI library provides user interfaces, including property pages, to support objects contained in the Output library. In addition to the property pages, there are a number of dialog boxes, including the PrintDialog and ExportDialog, available for developers to use. Developers extend this library when they create UIs for corresponding components they have created in the Output library.

CATALOG

The Catalog library contains objects and defines interfaces to support data cata-

logs. The catalog is a representation of persistent data. The data can be both local and remote. By using the objects within the catalog, developers can browse data holdings and, if required, obtain connections to the data. Many of the objects defined in Catalog are referred to as “GX” objects. These GX objects all implement the interface IGxObject. Objects that implement this interface can be manipulated within a catalog. GxFilters, which allow developers to browse for certain types of data, are also defined in this library. Developers commonly extend this library when they want to add catalog support for a data type not already supported by the ArcGIS system.

CATALOGUI

The CatalogUI library provides user interfaces, including property pages, to support objects contained in the Catalog library. In addition to the property pages, there are a number of dialog boxes, including the GxDialog, that can be used when interacting with catalogs and their contents. The GxDialog object supports the “Add Data” functionality of the ArcGIS applications. The FindDialog is also implemented by this library. Many of the commands and context menus seen in the ArcCatalog application are defined in this library. Developers extend this library when they create UIs for corresponding components they have created in the Catalog library.

Although implemented in libraries, commands are not exposed directly to developers. Developers obtain references to commands through the hosting application.

CARTOUI

The CartoUI library provides user interfaces, including property pages, to support objects contained in the Carto library. In addition to the property pages, there are a number of dialog boxes, including the IdentifyDialog, available for developers to use, although many of the dialog boxes contained in this library are commonly accessed through a property page. Developers extend this library when they create UIs for corresponding components they have created in the Carto library.

DATASOURCESRASTERUI

The DataSourcesRasterUI library provides user interfaces, including property pages, to support objects contained in the DataSourcesRaster library. In addition to the property pages, there are dialog boxes, including RasterSdeLoader and SidEncoder, available for developers to use. Developers extend this library when they create UIs for corresponding components they have created in the DataSourcesRaster library.

ARCCATALOGUI

The ArcCatalogUI library provides user interface components specific to the ArcCatalog application. The dialog box IDs for specific ArcCatalog dialog boxes are found in this library. Developers do not extend this library.

ARCCATALOG

The ArcCatalog library contains the ArcCatalog application, including the Application and Document objects. Some of the interfaces, such as IGxApplication, are defined in the ArcCatalogUI library. This is because the objects in the ArcCatalogUI library use the IGxApplication interface to interact with the ArcCatalog application. The GxDocument object fires various events during the

As earlier noted, it is advantageous for developers to develop their commands and tools for use within the various ArcGIS controls, as well as the ArcMap application.

lifetime of the ArcCatalog application that can be used by developers to synchronize with ArcCatalog events. Developers do not extend this library; instead, they create commands and tools for use within the ArcCatalog application.

ARCMapUI

The ArcMapUI library provides user interface components specific to the ArcMap application. The components contained in this library cannot be used outside the context of ArcMap. The IMxApplication and IMxDocument interfaces are defined in this library, although they are implemented in the ArcMap library. The ArcMap table of contents is implemented in this library, along with many of the commands present in ArcMap. Developers extend this library by creating custom commands and tools for use within the ArcMap application.

EDITOR

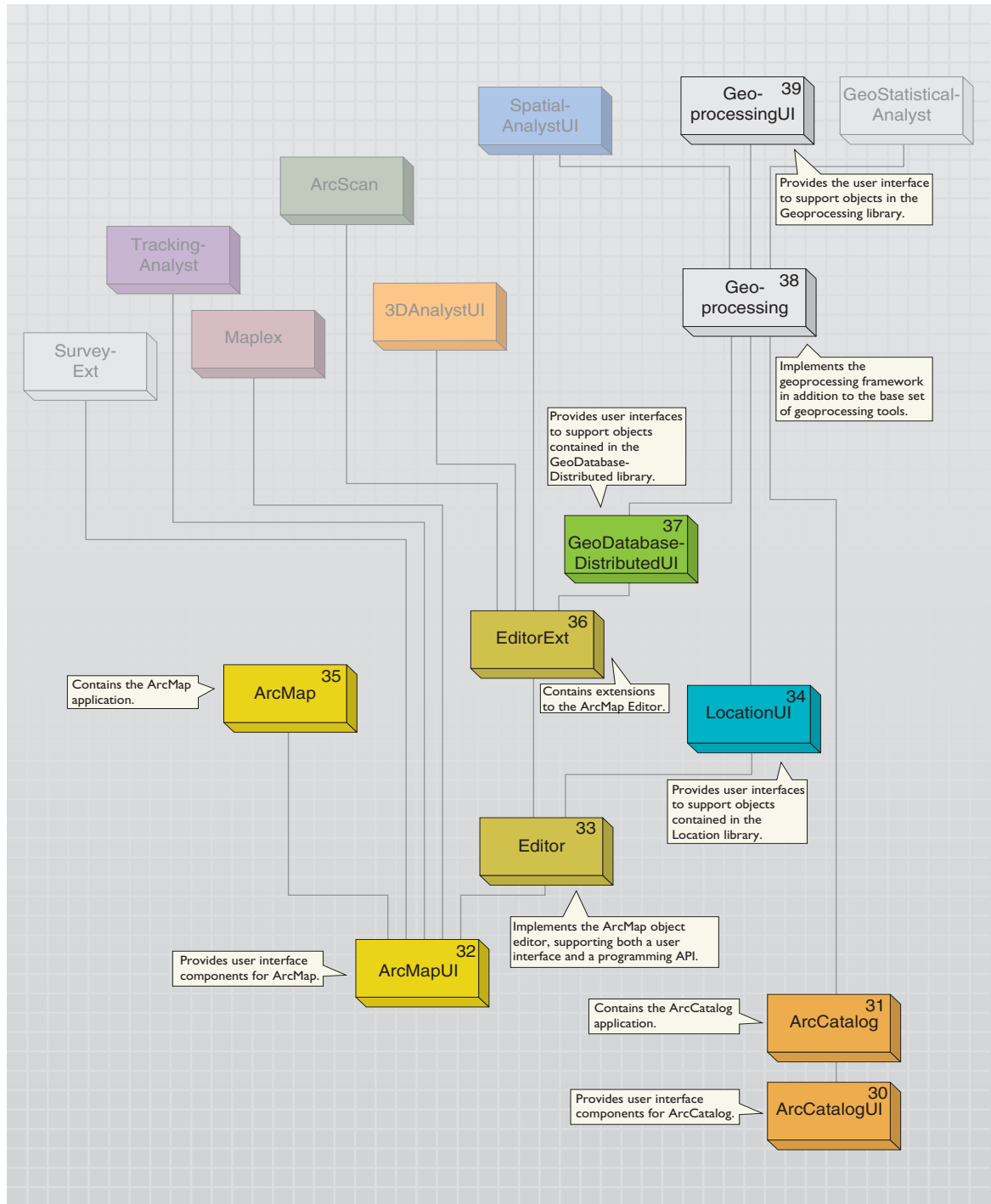
The Editor library implements the ArcMap object editor. The editor supports the editing of simple features, network features, annotations, and topological features, along with attributes for all these features. The library supports both a user interface and a programming API. The API provided by the Editor is a higher-level API than that of the GeoDatabase library. Developers are encouraged to use the Editor API when editing and manipulating features in the geodatabase. Developers can extend the library with their own editing commands, edit tasks, and snap agents; and for more advanced customization, developers can create extensions to the editor. The object inspector interface is implemented by the Editor library; however, to extend this user interface, a Class extension is implemented by extending the GeoDatabase library for the appropriate data source.

LOCATIONUI

The LocationUI library provides user interfaces, including property pages, to support objects contained in the Location library. In addition to the property pages, there are a number of dialog boxes, including the EventFinder and AddressLocatorUI, available for developers to use. This library also contains objects that extend other core libraries of the ArcGIS system, such as Catalog, CatalogUI, and CartoUI. Developers extend this library when they create UIs for corresponding components they have created in the Location library.

ARCMap

The ArcMap library contains the ArcMap application, which is implemented by the Application object. Similar to the ArcCatalog library, the Application object implements interfaces from other libraries; namely, ArcMapUI. The ArcMap application can be programmatically controlled either by developers who write new commands and tools that are included in the application or through OLE automation. When interacting with ArcMap and OLE automation, it is important that all objects used by ArcMap are created within the context of ArcMap. To support this programming model, the application implements the IObjectFactory interface. Developers also use the Application object to work with ArcMap documents, dockable windows, extensions, and the various data windows supported by ArcMap. Developers do not extend this library; instead, they create commands and tools for use within the ArcMap application.



EDITOREXT

The EditorExt library contains extensions to the ArcMap Editor and components dependent on the Editor. The functionality supported by this library is diverse, with the commonality being its reliance on the Object Editor. The library has functionality and associated UI to support network tracing, database loading, ArcPad® integration, edge matching, and managing map topology within ArcMap. Developers do not commonly extend this library; rather, they create their own editor extensions in their own library.

GEODATABASEDISTRIBUTEDUI

The GeoDatabaseDistributedUI library provides user interfaces, including property pages and dialog boxes, to support objects contained in the GeoDatabaseDistributed library. Developers do not extend this library.

GEOPROCESSING

The Geoprocessing library contains the objects that implement the unified geoprocessing framework. This framework supports the execution of geoprocessing tools using Dialogs, Models, Scripts, Command Line, and the ArcObjects COM or .NET APIs. In addition to the core framework, the library contains more than 200 geoprocessing tools. Developers can programmatically interact with the framework using the objects in this library. More commonly, developers will extend this library with new geoprocessing tools for subsequent use within the geoprocessing framework. Other libraries within the ArcGIS system implement geoprocessing tools so their functionality is exposed to users through the unified framework.

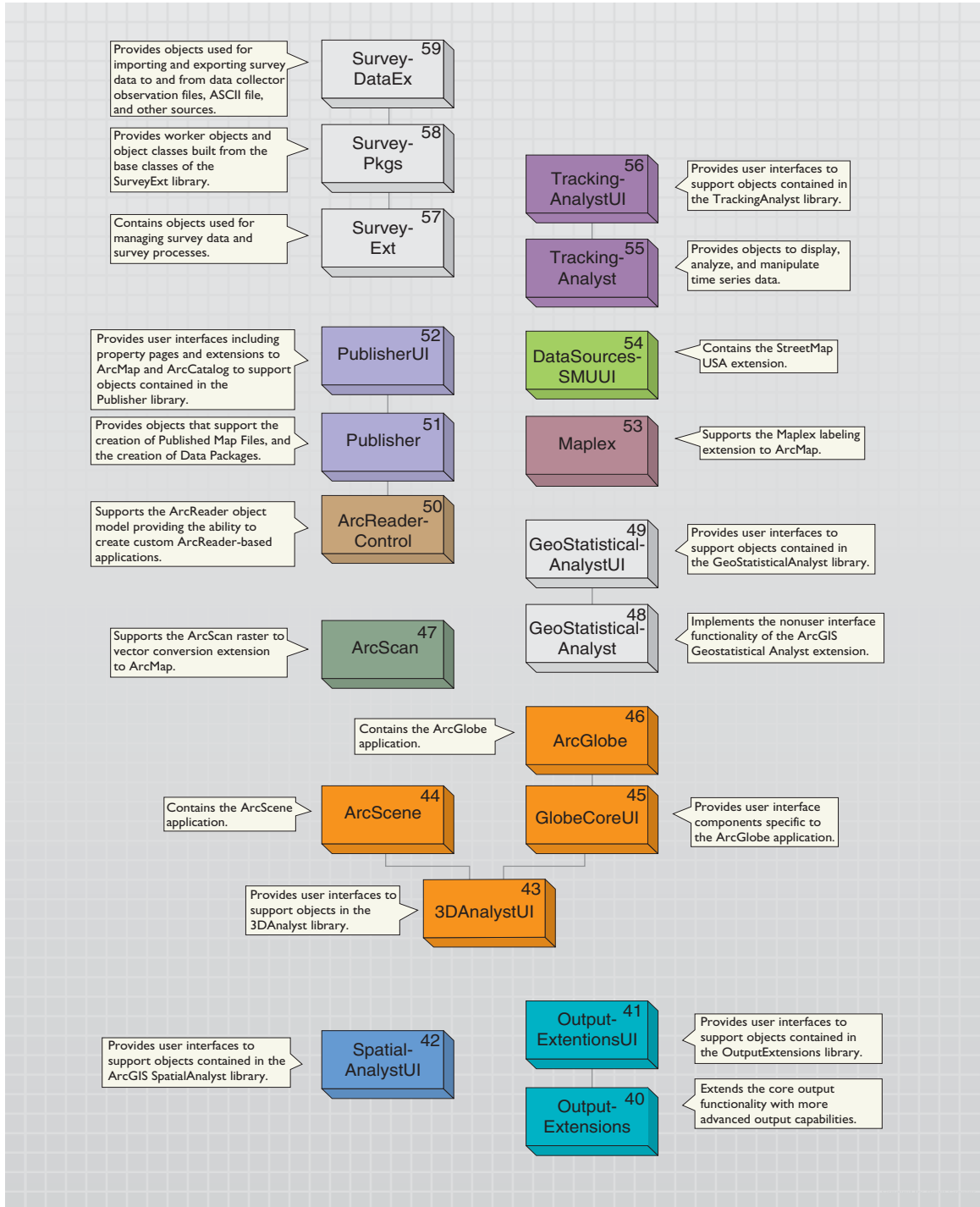
GEOPROCESSINGUI

The GeoprocessingUI library provides user interfaces, including property pages, to support objects contained in the Geoprocessing library. In addition to the property pages, there are a number of dialog boxes available for developers to use. Developers extend this library when they create UIs for corresponding components they have created in the Geoprocessing library. In addition to the normal property pages created to support objects in a non-UI library, it is possible that new ActiveX controls will be required to support data types used by geoprocessing tools. The ActiveX controls are used by the geoprocessing dialog boxes when requesting the parameters for a geoprocessing tool from users. This library contains all the parameter controls required by the geoprocessing tools defined in the Geoprocessing library.

OUTPUTEXTENSIONS

The OutputExtensions library extends the core output functionality with more advanced output capabilities. The ArcPress™ printing engine is implemented by this library. This library is not extended by developers. Depending on the components used from the library, an extension license may be required.

As earlier noted, it is advantageous for developers to develop their commands and tools for use within the various ArcGIS controls, as well as the ArcMap application.



Within ArcGIS Desktop it is not uncommon to see extensions broken up into non-UI and UI libraries; OutputExtensions and Publisher are two such examples.

OUTPUTEXTENSIONSUI

The OutputExtensionsUI library provides user interfaces, including property pages, to support objects contained in the OutputExtensions library. Developers do not extend this library. Depending on the components used from the library, an extension license may be required.

SPATIALANALYSTUI

The SpatialAnalystUI library provides user interfaces, including property pages, to support objects contained in the SpatialAnalyst library. The library also contains a comprehensive set of geoprocessing tools that expose the SpatialAnalyst library functionality for use within the geoprocessing framework. The SpatialAnalyst Extension object is implemented by this library. Developers extend this library when they create UIs or geoprocessing functions for corresponding components they have created in the SpatialAnalyst library. An ArcGIS Spatial Analyst license is required to work with objects in this library.

3DANALYSTUI

The 3DAnalystUI library provides user interfaces, including property pages, to support objects contained in the 3DAnalyst library. The library also contains a comprehensive set of geoprocessing tools that expose the 3DAnalyst library functionality for use within the geoprocessing framework. The 3DAnalyst Extension object is implemented by this library. Developers extend this library when they create UIs or geoprocessing functions for corresponding components they have created in the 3DAnalyst library. A 3D Analyst license is required to work with objects in this library.

ARCSCENE

The ArcScene library contains the ArcScene™ application, along with its associated user interface components, commands, and tools. The ArcScene Application and SxDocument objects are both defined and implemented by this library. Developers can use the Application object when customizing the ArcScene application or working with one of the ArcScene extensions. Developers extend this library by creating commands, tools, and extensions for use within the ArcScene application. A 3D Analyst license is required to work with objects in this library.

Similar to developing tools for ArcMap, developers should develop commands and tools that work in both the ArcScene application and the SceneControl developer component.

GLOBECOREUI

The GlobeCoreUI library provides user interface components specific to the ArcGlobe™ application. The library also provides property pages for objects contained in the GlobeCore library. It is not common for developers to extend this library. A 3D Analyst extension license is required to work with objects in this library.

ARC GLOBE

The ArcGlobe library contains the ArcGlobe application, along with its associated user interface components, commands, and tools. The ArcGlobe Application and GMxDocument objects are both defined and implemented by this library. Developers can use the Application object when customizing the ArcGlobe application

Similar to developing tools for ArcScene, developers should develop commands and tools that work in both the ArcGlobe application and the GlobeControl developer component.

or working with one of the ArcGlobe extensions. Developers extend this library by creating commands, tools, and extensions for use within the ArcGlobe application. A 3D Analyst license is required to work with objects in this library.

ARCSCAN

The ArcScan library supports the ArcScan™ raster-to-vector conversion extension to ArcMap, along with its associated user interface components, commands, and tools. The ArcScan extension object, Vectorization, is implemented by this library. Developers do not extend this library. An ArcScan extension license is required to work with objects in this library.

GEOSTATISTICALANALYST

The GeoStatisticalAnalyst library implements the non-UI functionality of the Geostatistical Analyst extension to ArcMap. The geostatistics engine, along with the GeoStatistical Layer and its associated renderers, are implemented by this library. Developers do not extend this library. A Geostatistical Analyst extension license is required to work with objects in this library.

GEOSTATISTICALANALYSTUI

The GeoStatisticalAnalyst UI library provides user interfaces, including property pages, to support objects contained in the GeoStatisticalAnalyst library. In addition to the property pages, there are a number of dialog boxes available for developers to use. The library also contains a set of geoprocessing tools that expose the GeoStatisticalAnalyst library functionality for use within the geoprocessing framework. A Geostatistical Analyst extension license is required to work with objects in this library.

ARCREADERCONTROL

The ArcReaderControl library contains the ArcReaderControl along with the objects that make up the ArcReaderControl API. This API supports the creation of custom ArcReader™ applications through a simplified API. The encapsulated ArcObjects are not accessible using this control. The functionality supported by the control is similar to that of ArcReader with the addition of query features. There is no cost to deploy applications built using the ArcReaderControl because it uses the free ArcReader application to provide the run-time environment. The ArcReaderControl is not extended by developers.

PUBLISHER

The Publisher library implements the non-UI functionality of the Publisher extension to ArcGIS. The PublisherEngine and PackagerEngine objects support the publishing of PMF files and the subsequent packaging of the published map files. Developers do not extend this library. An ArcGIS Publisher extension license is required to work with objects in this library.

PUBLISHERUI

The PublisherUI library provides user interfaces, including property pages along with ArcCatalog and ArcMap commands, to support objects contained in the Publisher library. Developers do not extend this library. An ArcGIS Publisher extension license is required to work with objects in this library.

MAPLEX

The Maplex library implements the non-UI functionality of the Maplex extension to ArcMap. Developers do not extend this library. A Maplex extension license is required to work with objects in this library.

TRACKINGANALYST

The TrackingAnalyst library implements the non-UI functionality of the Tracking Analyst extension to ArcGIS. The Tracking Analyst extension supports the display, analyses, and manipulation of time series data within ArcGIS. Developers do not extend this library. A Tracking Analyst extension license is required to work with objects in this library.

TRACKINGANALYSTUI

The TrackingAnalystUI library provides user interfaces, including property pages, along with commands and tools, to support objects contained in the TrackingAnalyst library. Developers do not extend this library. A Tracking Analyst extension license is required to work with objects in this library.

SURVEYEXT

The SurveyExt (Survey Extension) library handles the core objects used to manage survey data and survey processes. The system allows angle and distance measurements observed using field survey equipment to be processed in order to generate computed coordinates. The objects in this library are data objects and data management objects. Data management objects are manifested through ArcCatalog as survey-specific datasets, projects, and folders and through ArcMap as survey layers and a Survey Explorer UI. The fundamental survey data objects with which users interact are survey points, coordinates, simple measurements, composite measurements, and computations. These objects are persisted as rows in a set of tables that represent the survey dataset. These tables are called survey classes. Each row in the table is called a survey object. A computation is a survey process that is persisted in a computation survey class. Each has a signature behavior, makes reference to survey points and measurements, and generates coordinates based on the observation values held by the reference objects. Two examples of a computation are a traverse and a least-squares adjustment.

The SurveyExt library is the set of primitive objects that forms the foundation for the objects in the SurveyPkgs and SurveyDataEx libraries.

SURVEYPKGS

The SurveyPkgs (Survey Packages) library provides a set of concrete worker objects and object classes that are built from the base classes of the SurveyExt

library. There are three survey packages provided with Survey Analyst in ArcGIS 9. They are the Point, TPS (total station), and COGO packages. The existing survey packages can be extended and customized, and new survey packages can be created. To create your own survey package, you need to aggregate the core Survey Analyst objects from the SurveyExt library. To do this, a programming language that supports aggregation is required. VB does not support aggregation, but C++ does. Survey packages may depend on each other in a hierarchical manner. For instance, instead of defining their own point and coordinate data types, the COGO and TPS packages use the point and coordinate data types of the point package.

SURVEYDATAEX

The SurveyDataEx (Survey Data Exchange) library handles the core objects used to import and export survey data to and from data collector observation files, ASCII files, and other sources. The objects in this library are used to ensure data integrity when merging imported data with preexisting data in a survey dataset. Data exchange objects may be used through ArcCatalog and ArcMap or instantiated and executed through standalone applications.

The fundamental survey data exchange objects with which users interact are survey converters. These objects are registered in two component categories: ESRI Survey Analyst Survey Import Converters and ESRI Survey Analyst Survey Export Converters. Custom survey converters must also be registered in these component categories.

The core ArcGIS Survey Analyst extension supports the following import converters: Configurable ASCII coordinate importer, Geodimeter, Geo Serial Interface, Tripod Data Systems, and Sokkia SDR. In addition, its export converters allow the transfer of coordinates into these same formats.

3

Developing for ArcGIS Desktop applications

The most common way developers customize ArcGIS Desktop applications is through Visual Basic for Applications, which is embedded within each application. The application framework also provides for the creation of plug-in components to extend the applications.

This chapter discusses the application framework, customization options, customization of the applications with VBA, and the creation of plug-in components.

In the beginning of this book, you were introduced to the development possibilities provided by the ArcGIS Desktop applications. This chapter will outline in greater detail those customization options and introduce you to their development.

COMMON APPLICATION FRAMEWORK

In Chapter 2, 'ArcGIS software architecture', you learned that the ArcGIS product family shared a similar architecture based on ArcObjects. The desktop applications also share a common application framework, which you should understand before undertaking any desktop development.

The ArcGIS Desktop applications are developed using ArcObjects. When you use an application, such as ArcMap, most of the time you are simply looking at or working with ArcObjects.

The graphical user interface in each application is also developed using the same objects, such that in each application you will find the interface contains toolbars, menus, commands, and tools that have the same look and feel. The interfaces can also be easily manipulated in terms of adding and removing toolbars, docking toolbars, adding and removing commands, and so on. This nonprogrammatic manipulation of the interface is actually the first of the customization options that will be described later.

It's important to remember that every command or tool you use, for adding data, editing, or performing some GIS analysis, is simply running some code that includes ArcObjects behind the scenes. The application framework of the desktop applications allows you, as a developer, to write your own code using ArcObjects to perform some customization.

Each desktop application includes VBA. The VBA development environment is integrated within the application and provides the Customize dialog box for user interface manipulation, as described above, and the Visual Basic Editor, which provides an interface for creating forms and writing ArcObjects code. The integration between the application and VBA allows you to create your own controls and work with the application and current document.

The application framework also provides for the creation of components, such as new commands, tools, or extensions, that plug in to one or more of the desktop applications. These components are typically created with development environments, such as Visual Basic 6 or .NET.

Recall that all the desktop applications are built using the same objects, share a common interface, and allow for the creation and integration of your own ArcObjects code. The way you develop a customization for applications, such as ArcMap, is no different from creating a customization for another desktop application, such as ArcGlobe or ArcCatalog. The ArcObjects that you consume will be different, but the implementation is essentially the same due to the common application framework that these applications share.

Exactly how you perform these customizations is the focus of this chapter and will be described later.

Customization options

The types of customization possible within ArcGIS Desktop are summarized in the following list and described below:

- User interface customization
- VBA macros
- VBA UI controls
- Framework components
- Extensions
- Custom layers, features, and symbology

User interface customizations

User interface customizations involve a developer or user altering the application graphical user interface. As mentioned previously, this may be as simple as adding or removing toolbars, but more commonly involves adding, removing, or rearranging controls on existing toolbars. This is typically done to de-clutter or simplify the user interface in certain work flow situations. For example, an edit work flow task may make use of several controls on two or more existing toolbars. A customization here may involve the creation of a new toolbar and the moving of existing controls or controls not usually displayed with the new toolbar. This enables operators to perform the work flow task with all the required tools on one toolbar so they can focus on the task at hand and not spend time searching for the next control, thus increasing productivity.

User interface customization is perhaps the easiest customization because they do not involve programming. All the modifications are performed via the Customize dialog box, part of the VBA development environment, or click-and-drag operations. The user interface can, of course, be altered programmatically, but this is usually done in conjunction with delivering other controls (commands, tools, and so forth), which will be described later.

VBA macros

VBA offers a quick and easy entry point for ArcGIS Desktop development. The simplest customization through code is the creation of VBA macros, also called procedures, that can be run from within the VBA development environment or by calling the macro from the desktop application. The macro contains VBA code, which uses ArcObjects to perform operations and can also include calls to existing ArcGIS Desktop commands. Macros often become the foundation for the more advanced customizations as they provide a good prototyping environment.

VBA UI controls

Through VBA you can also create your own commands, tools, and menus and place them on the interface via the Customize dialog box. These controls contain code that runs when you, for example, click a button or apply a tool. The code contained within macros is often converted to controls, or alternatively, controls may call existing macros.

Framework components

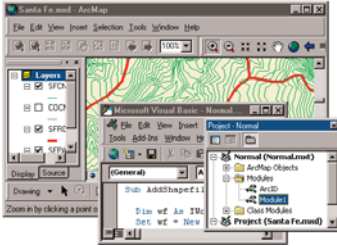
Although VBA may meet most of your development needs, there are some situations in which you may want to write ArcObjects code using a separate development environment, such as Visual Basic 6, Visual C++, Visual Studio .NET, and so forth. These offer richer development environments and the ability to create more advanced customizations that are not possible in VBA. Through these environments you can create components that plug in to the desktop framework as controls, such as commands, tools, menus, and toolbars. These types of components offer more advantages for the developer but often take longer and are more complicated to develop.

Extensions

Extensions provide users with additional GIS functionality. Typically, GIS functions that perform a specific task are grouped into an extension. 3D Analyst and Tracking Analyst are examples of extensions provided by ESRI. Extensions are often used by developers who require the license and management functions provided by the extension framework. Extensions cannot be created in VBA.

Custom objects

Advanced developers create custom objects for specific applications. Custom layers, features, and symbology are examples of such objects and can only be created in development environments, such as Visual C++ or .NET. Developing custom objects is beyond the scope of this book.



map document

- data references** The ArcMap table of contents manages the geographic data referenced in the map.
- map layout** A map can be composed with data frames and cartographic elements and saved.
- user interface** ArcMap has a standard user interface which can be customized and saved in a document.
- VBA project** A Visual Basic for Applications project contains forms, modules, and classes.

DOCUMENTS AND TEMPLATES

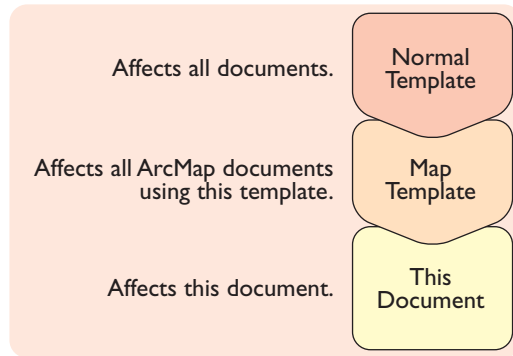
Understanding documents and templates is an essential part in understanding customization with ArcObjects in ArcGIS Desktop applications.

Whenever you are using ArcMap, ArcGlobe, or ArcScene, you are working with a document, usually referred to as a map, globe, or scene document. This document stores the map state, the state of the user interface, custom user interface settings, a Visual Basic for Applications project, and other application-specific information, such as cartographic layouts for ArcMap documents.

Templates are kinds of documents that serve as starting points for new documents. All the desktop applications have a template known as the Normal template, which stores the default or original state of the application. ArcCatalog is a special case because it only uses the Normal template and has no documents.

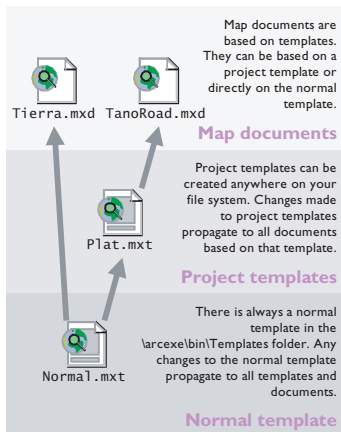
As a desktop developer you can store your customizations in either the current document or the Normal template. ArcMap developers can also store the customizations in another map template usually established for different cartographic layouts.

When you open a desktop document, the corresponding desktop application first reads the customizations from the Normal template, then the map template, if applicable, then finally the document itself. The graphic below illustrates how customizations are read from top to bottom to incorporate customizations from all levels.

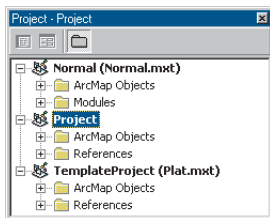


The order in which these are read is important because changes in one template can affect other templates and the desktop document. For example, an ArcMap Normal template may have the AddData command turned off, but a map template or document may turn the command back on.

The structure and function of documents and templates are further explained below in the context of each respective application.



ArcMap automatically creates a Normal template if one does not exist. If you have applied unintended customizations, such as removing toolbars and command items, you can simply remove the Normal.mxt file, and a new one with the standard user interface will be generated. This is easier than undoing a set of unintended customizations.



This is how the three loaded templates in ArcMap—normal, project (current document), and project template—appear in the VBA project explorer.

CUSTOMIZING ArcMAP

You can customize ArcMap in several ways:

- You can add references to geographic data and define how the data is displayed.
- You can create a map layout with a spatial reference and ancillary cartographic elements.
- You can add, remove, or rearrange elements of the standard user interface.
- You can write code in a Visual Basic for Applications project.

All customization in ArcMap is stored in a map document or a map template.

The changes you make to the ArcMap table of contents, the layout of a map, the toolbars and their command items, and the VBA code you write all get saved to the map document.

A map document can reside anywhere on your file system; it has a file extension of .mxd.

Map templates

You can use map templates to disseminate customization throughout an organization—globally, by project, or by document.

A map template is a kind of map document. In nearly every respect, map templates are structurally identical to map documents. The functional difference is that ArcMap recognizes and uses templates as a starting point to create new map documents. This is similar to how you work with templates in Microsoft Office applications.

Any customization of the user interface or the VBA project becomes part of the newly created map document. Furthermore, any changes to a template will propagate to template-based documents when they are next loaded.

There are three levels of templates and documents in ArcMap. You can save changes to any level to control how widely your customizations are used.

Custom map documents

When you are working with a map, you are setting references to data, designing a map layout, customizing the user interface, and writing VBA code, all for the lifetime of the document.

Selective customization with project templates

Other projects and other users can share the customizations that you make through templates. A template is a kind of map document that is specified to be a starting point for a new map document. The new map document will inherit all the customizations from the template (data references, map layout, user interface state, and VBA project).

Global customizations with the Normal template

The Normal template in ArcMap stores any personal settings you have made to the user interface that you want loaded every time you start ArcMap. Any customizations that you save to the Normal template, including code and control customizations, will get propagated to all the other map documents when they are next opened.

In ArcMap the Normal template is also referred to as the Normal.mxt file.

When you first start ArcMap after installing the software, a Normal template is automatically created and put in your profiles location, which is one of the following folders, depending on your operating system.

For Windows NT®:

`C:\WINNT\Profiles\\Application Data\ESRI\ArcMap\Templates\`

For Windows 2000 and XP:

`C:\Documents and Settings\\Application Data\ESRI\ArcMap\Templates\`

This is the default Normal template that contains all the standard toolbars and commands and places the toolbars and the table of contents in their default positions. Any customizations that you save in your Normal template get saved to this file.

If you want to make changes that appear every time you open ArcMap, save them in the Normal template.

Suppose your administrator has custom toolbars or tools to which he or she would like everyone in your organization to have access. Your administrator could create a customized Normal template and allow everyone in your organization to use that Normal template instead of the default Normal template. To accomplish this, your administrator would customize the Normal template and copy that Normal.mxt file to the \ArcGIS\bin\Templates folder. Everyone would then start with this Normal template instead of the default Normal template. The following is an explanation of how this works.

If there is no Normal.mxt file in your profiles location when you start ArcMap, the application will look in the \ArcGIS\bin\Templates folder. If a Normal.mxt file exists in the \ArcGIS\bin\Templates folder, that file will be copied to your profiles location and will be treated as your personal Normal template. Therefore, you start off with a copy of your organization's customized Normal template, but from that point on you can save your own customizations to it.

If a Normal.mxt file is not found in your Profiles location or in the \ArcGIS\bin\Templates folder, then a new default Normal.mxt file will be created and placed in your Profiles location.

CUSTOMIZING ArcSCENE AND ArcGLOBE

ArcScene and ArcGlobe can be customized in the following ways:

- You can add references to geographic data and define how the data is displayed.
- You can add, remove, or rearrange elements of the standard user interface.
- You can write code in a Visual Basic for Applications project.

All customizations are stored as documents. For ArcScene the document extension is .sxd; for ArcGlobe it is .3dd.

The changes you make to the table of contents, the toolbars and their command items, and the VBA code you write all get saved to these documents.

Like ArcMap, these applications also have a Normal template that behaves the same way. For ArcScene the file is called Normal.sxd; for ArcGlobe it is Normal.3dt. You can find these files at the following locations, depending on your operating system:

For Windows NT:

`C:\WINNT\Profiles\\Application Data\ESRI\ArcScene (or ArcGlobe)`

For Windows 2000 and XP:

`C:\Documents and Settings\\Application Data\ESRI\ArcScene (or ArcGlobe)`

CUSTOMIZING ArcCATALOG

You can customize ArcCatalog in several ways:

- Add, remove, or rearrange elements of the standard user interface.
- Write code in a Visual Basic for Applications project.

ArcCatalog does not employ the full structure of documents and templates like ArcMap does. The ArcCatalog application does not use documents or base templates; it only uses a Normal template. Therefore, all customizations to the ArcCatalog user interface are stored in the Normal template.

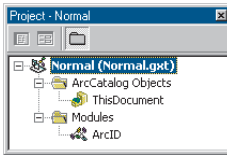
When you first start ArcCatalog after installing the software, a Normal template called Normal.gxt is automatically created and put in your Profiles location, which is one of the following folders, depending on your operating system.

For Windows NT:

`C:\WINNT\Profiles\\Application Data\ESRI\ArcCatalog\`

For Windows 2000:

`C:\Documents and Settings\\Application Data\ESRI\ArcCatalog\`



This is how the ArcCatalog Normal template appears in the VBA project explorer.

VBA is available in all the ArcGIS Desktop applications such as ArcMap, ArcCatalog, and ArcGlobe. The examples in this quick-start tutorial work within ArcMap, but the process of creating macros and commands for the other applications is the same.

This section details the various VBA customizations available to desktop developers. Specifically, it will cover user interface customizations, VBA macros, and VBA UI controls in the form of quick-start tutorials.

Before attempting these tutorials, however, you should have at least a basic understanding of ArcObjects, Microsoft COM, and the Visual Basic 6/VBA syntax. This information can be found in Chapter 2, ‘ArcGIS software architecture’, and Appendix A, ‘Developer environments’, the language reference appendix found in this guide.

After completing this section you should also review Appendix C, ‘Illustrated code samples’, and Appendix D, ‘Problem-solving guide’. The illustrated code samples show numerous ArcObjects code examples that can be used in VBA utilizing the techniques shown in this section. The problem-solving guide walks you through a typical customization problem in ArcGIS Desktop using ArcObjects and VBA.

1. To start this tutorial, click the Windows Start button, point to Programs, point to ArcGIS, and click ArcMap.
2. On the startup dialog box, in the ‘Start ArcMap using’ options, click A new empty map. Click OK.
3. Add some sample data or your own data to the map.
4. Save the map document.

USER INTERFACE CUSTOMIZATIONS

The following tutorials describe nonprogrammatic user interface customizations through the use of the Customize dialog box.

Showing and hiding toolbars using the Customize dialog box

1. Click the Tools menu and click Customize.

The Customize dialog box appears.

You can also double-click any unoccupied area of any toolbar to display the Customize dialog box.

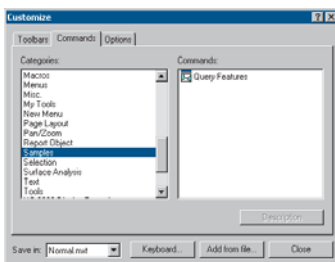
2. If it is not visible, click the Toolbars tab.

The presence or absence of a check mark next to the toolbar name indicates its visible state.

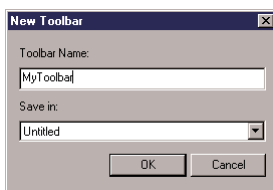
3. Check and uncheck the check boxes.

Creating a new toolbar

1. On the Toolbars tab of the Customize dialog box, click New.
2. On the dialog box that appears, type “MyToolbar” as the name of the new toolbar or use the default setting.
3. Store the toolbar in the document by changing its name on the Save in dropdown list from Normal.mxt to the name of the current project you saved.



The Customize dialog box



The New Toolbar dialog box

4. Click OK.

The newly created toolbar appears near the top of the application window.

Adding buttons to a toolbar

1. Make sure the toolbar you just created, MyToolbar, is visible.
2. Open the Customize dialog box.
3. Click the Commands tab on the Customize dialog box.
4. Click Pan/Zoom from the Categories list on the left of the dialog box.
5. Scroll to the bottom of the Commands list on the right of the dialog box.
6. Click the Zoom in command and drag it to the MyToolbar toolbar. Release the command when the arrow cursor with a small box below it appears.
7. Continue adding commands from the Pan/Zoom category until you have your own version of the built-in Tools toolbar.

NOTE: You may switch to other categories to select commands.

You can dock the toolbar or drag it to any of the toolbar drop sites on the application window.



Dragging a toolbar



Your toolbar might look like this.

Renaming a toolbar

1. On the Toolbars tab, click the name of the toolbar whose name you want to change. In this case, click MyToolbar.
2. Click the Rename button.
3. In the dialog box that appears, type “My Own Tools” for the new name.

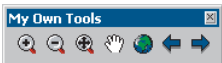
NOTE: You can only rename toolbars you’ve created.

4. Click OK.

If you decide not to rename the toolbar, click Cancel.

Removing buttons from a toolbar

1. Make sure the toolbar you just renamed, My Own Tools, is visible.
2. Open the Customize dialog box.



My Own Tools toolbar

3. Drag some of the commands off the toolbar.

Even though you've removed the buttons from the toolbar, they are still available in the Customize dialog box.



Selection menu on the My Own Tools toolbar

Documents and templates are explained in more detail in the previous section.

Adding a menu to a toolbar

1. Make sure the My Own Tools toolbar is visible.
2. Open the Customize dialog box.
3. Click the Commands tab and choose the Menus category from the Categories list on the left side of the dialog box.
4. In the Commands list on the right side of the dialog box, click Selection.
5. Drag and drop it to the left of the Zoom In button on the My Own Tools toolbar.
6. Click Close in the Customize dialog box.
7. Click Selection on the My Own Tools toolbar and note the menu that appears.

Saving changes to a template

You can save your work to a document or template. Changes saved to a document are specific to the document, whereas changes saved to a template will be reflected in all documents based on the template.

1. Click the File menu and click Save. This saves the current document.
2. Click the File menu and click Save As.
3. Navigate to the Templates folder of the <installation directory>\bin folder.
4. Click the Create New Folder button.
Type a new name for the folder and double-click it. You'll see the folder name as a tab the next time you create a document from a template.
5. Type the template name in the filename text box, click ArcMap Templates (*.mxt) from the Save as type dropdown menu, then click Save.
6. Reopen the original document you saved in Step 1.

VBA MACROS

You can use the VBA IDE to create macros to help you automate tasks you perform repeatedly or to extend the application's built-in functionality.

Before starting the tutorial you need to be aware of the following framework properties.

Preset ArcObjects Variables

In the ArcGIS Desktop applications you have two preset variables that will serve as the starting point for much of your code: Application and ThisDocument. These object variables are always available as soon as you launch the desktop application. While the number of methods and properties that are available on these variables is fairly limited, they serve as a stepping stone to other objects you might want to program with, such as maps, layers, and files.

To see all the methods and properties available with the Application variable, see the Application entry in the ArcGIS Developer help.

Application Variable

Application is a preset variable that points you to the current application. In ArcMap, Application refers to the ArcMap application, while in ArcCatalog, it refers to ArcCatalog, and so forth. In either environment, the Application variable will always have the same methods and properties.

Below is a sample of some methods and properties available on Application.

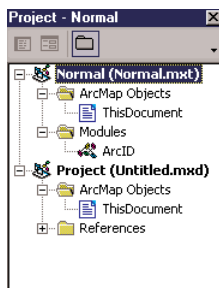
- Caption: A read/write property to get or set the text that appears on the applications title bar
- Name: A read-only property that will always return the name of the application as a string (ArcMap, ArcCatalog, and so forth)
- RefreshWindow: A method to redraw the application window

ThisDocument

The ThisDocument preset variable points to the document that is currently open in the application. In ArcCatalog this always refers to the Normal template, while in the other desktop applications it refers to the current, map, scene, or globe document. Unlike the Application preset variable, there are some differences in the methods and properties available on ThisDocument in ArcCatalog versus the other desktop applications.

Here are some methods and properties you will find on ThisDocument:

- AddLayer: ArcMap only. A method to add a new layer to a document. A layer is required as an argument.
- Title: A read-only property to get the name of the document (for example, normal.mxt, WestAust.mxd).
- Type: A read-only property that describes the type of current document (for example, Normal template, map template, or document).



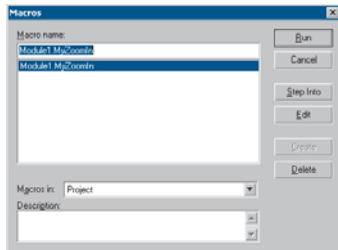
The VBA project explorer displays the available VBA projects.

VBA project organization

When working within the VBA editor, the project explorer window displays two or more projects. The Normal project refers to the Normal template associated with the application. If you write macros within this project, they will be saved to the Normal template and, thus, will be available to all documents for this application. If you write macros in the document project, however, the code will only be available in that current document. A third project may be displayed if you are working with a map template. Any code saved to a template project will be available to documents based on that template.

ArcID module

Each VBA project contains a module that is automatically added called ArcID. This module, stored in the Normal template, contains a reference to all existing



The Macros dialog box

controls (commands, tools, and so forth) within the current application. Using this information you can call or run an existing control from within your code. This procedure is described in the tutorials.

Creating a macro

With the Visual Basic Editor, you can create and edit macros, copy macros from one module to another, rename the modules that store the macros, or rename the macros.

1. Click the Tools menu, point to Macros, then click Macros.
2. In the Macros dialog box, type “MyZoomIn” in the Macro name text box and click Create.

The application creates a new module named Module1 and stubs out the Sub procedure.

3. Enter the following code for *MyZoomIn*:

```
Sub MyZoomIn()
    '
    ' macro: MyZoomIn
    '
    Dim pDoc As IMxDocument
    Dim pEnv As IEnvelope
    Set pDoc = ThisDocument
    Set pEnv = pDoc.ActiveView.Extent
    pEnv.Expand 0.5, 0.5, True
    pDoc.ActiveView.Extent = pEnv
    pDoc.ActiveView.Refresh
End Sub
```

The first line of the macro declares a variable that represents the ArcMap document. At this point, the coding techniques that are used with the ArcInfo COM-based object model will not be addressed. These techniques are discussed in greater detail later in this guide.

The second line declares a variable that represents a rectangle with sides parallel to a coordinate system defining the extent of the data. You'll use *pEnv* to define the visible bounds of the map.

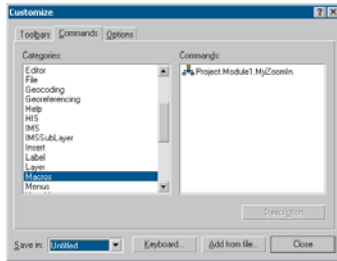
The predefined variable, *ThisDocument*, is the *IDocument* interface to the *Mx-Document* object that represents the ArcMap document.

The *ActiveView* property provides an *IActiveView* interface that links the document data to the current screen display of that data.

By reducing the size of the envelope that represents the extent of the map, the macro zooms in on the map's features once the screen display is refreshed.

4. Switch back to ArcMap by clicking the File menu, then clicking Close and Return to ArcMap.
5. Click the Tools menu, point to Macros, then click Macros.

Some macros will not run correctly if the VBA editor window is still open.



The Customize dialog box

You can only change the properties of an object, such as a macro, command, or tool, while the Customize dialog box is visible.

6. Select the *Module1.MyZoomIn* macro and click Run.

The display zooms in.

Adding a macro to a toolbar

You'll want convenient access to the macros you write. You can add a macro to built-in toolbars or toolbars you've created.

1. Click the Tools menu and click Customize.
2. In the Toolbars tab, ensure that the toolbar you created is visible.
3. Click the Commands tab and click the Macros category.
4. Click the name of your project in the Save in dropdown menu.

The Commands pane to the right of the dialog box lists Project.Module1.MyZoomIn.
5. Drag the macro name to the My Own Tools toolbar you created.

The macro appears as a button on the toolbar with a default icon.
6. To change its properties, right-click the icon.
7. In the context menu that appears, click Change Button Image and choose a button from the palette of icons.
8. Close the Customize dialog box.
9. Click the button on the toolbar to run the macro.

Invoking the Visual Basic Editor directly

As an alternative to the Create button in the Macros dialog box, you can navigate directly to the Visual Basic Editor and create procedures on your own. In this section, you'll create a macro named *MyZoomOut* in the *Module1* module that will zoom out from the display. You can use the same code that you used for *MyZoomIn*, with only a minor modification to one line.

1. Press Alt+F11, which is the Visual Basic Editor keyboard accelerator.
2. Click Project Explorer in the Visual Basic Editor View menu.
3. In the Project Explorer, click the Project entry, then Modules, then Module1.
4. In the Code window, copy the *MyZoomIn* code from the beginning of the Sub to the End Sub.
5. Paste the *MyZoomIn* Sub code below the existing code.
6. Change the name of the copied Sub to *MyZoomOut*.
7. Change the line:

```
pEnv.Expand 0.5, 0.5, True
to:
pEnv.Expand 2.0, 2.0, True
```

- Follow Steps 1–9 of the ‘Adding a macro to a toolbar’ section above to add and run your second macro.

Getting help in the Code window

The two macros you’ve just completed perform operations similar to the Fixed Zoom In and Fixed Zoom Out commands on the Tools toolbar. You didn’t really add any new functionality, but you’ve perhaps learned something about the object model and how to start to write some useful code. You can learn more about these methods you’ve worked with by making use of the help that’s available in the Object Browser or in the Code window.

- Click the Tools menu, point to Macros, then click Visual Basic Editor.
- Locate the *Module1* module. In the *MyZoomIn* Sub, click the method name *Expand* in the line:
`pEnv.Expand 0.5, 0.5, True`
- Press F1.

The ArcGIS Developer Help window displays the help topic for *Expand*. In addition, consult the ArcGIS Developer Help, which you can also start from the ArcGIS program group, for object model diagrams, samples, tips, and tricks.

Calling built-in commands

If you’ve read any of the ArcGIS user guides, you know that the code you’ll be writing will add functionality to an already rich environment. There may be instances in which you want to make use of several built-in commands executed in sequence or combine built-in commands with your own code.

Calling existing commands involves working with the *ArcID* module. Using the *ArcID* module you can get the unique ID (UID) of a particular command. By using its UID you can *Find* a command in ArcMap and run it. If you want to look at the *ArcID* module in greater detail, it’s in the Normal template of your application.

The following steps outline how to write a macro that calls existing commands:

- If you are not in the Visual Basic Editor, click the Tools menu, point to Macros, then click Visual Basic Editor.
- In the *Module1* module, create a Sub procedure with the following code:

```
Sub FullExtentPlus()
    '
    ' macro: FullExtentPlus
    '
    Dim intAns As Integer
    Dim pItem As ICommandItem
    With ThisDocument.CommandBars
        Set pItem = .Find(ArcID.PanZoom_FullExtent)
        pItem.Execute
        intAns = MsgBox("Zoom to previous extent?", vbYesNo)
        If intAns = vbYes Then
```

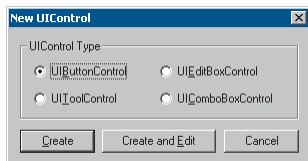
The Name of a command in the ArcID module can be derived using the following formula: Category In Customize Categories List + “_” + Command Caption in Customize Commands List. Any spaces are removed from the name.

```

Set pItem = .Find(ArcID.PanZoom_ZoomToLastExtentBack)
pItem.Execute
End If
End With
End Sub

```

3. Add the *FullExtentPlus* macro to a toolbar or menu.
4. Run the *MyZoomIn* macro, then run *FullExtentPlus*.



The New UIControl dialog box

Creating a command in VBA

Up to this point in the tutorial, you've only created macros. A command is similar to a macro but allows more customization in the way it interacts with the user and provides ToolTips, descriptions, and so on. Once invoked, a command usually performs some direct action without user intervention. A command is a type of UIControl. You can read more about all the UIControls in Appendix E, 'UI Controls'.

1. Click the Tools menu and click Customize.
2. In the Customize dialog box, click the Commands tab and click the Save in dropdown menu to navigate to your project.
3. In the Categories list, click UIControls.
4. Click New UIControl.
5. On the dialog box that appears, choose UIButtonControl as the UIControl Type, then click Create and Edit.

The following code assumes the UIButtonControl was named UIButtonControl1. If another name was used because this name was already in use, make the necessary changes to the code snippets.

The code in the Click event procedure will run when you click the button.

Adding code for the UIButtonControl

The application adds an entry in the Object box for the *UIButtonControl* and stubs in an event procedure for the *UIButtonControl's Click* event. You'll add code to this event to zoom the display to the extents of the dataset.

1. Add the following code to the *Click* event:

```

Private Sub UIButtonControl1_Click()
    Dim pDoc As IMxDocument
    Set pDoc = ThisDocument

    pDoc.ActiveView.Extent = pDoc.ActiveView.FullExtent
    pDoc.ActiveView.Refresh
End Sub

```

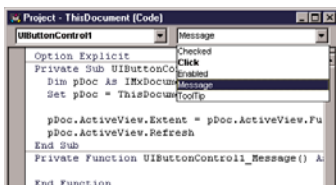
So far there is no difference from the macros you developed earlier. You will now add a ToolTip and message for the command.

2. Click Message in the Procedure combo box. This creates a stub function, to which you should add the following code:

```

Private Function UIButtonControl1_Message() As String
    UIButtonControl1_Message = _
        "Zooms the display to the full dataset extents"
End Function

```



VBA Editor showing the Code window, Object and Procedure combo box.

3. Click ToolTip in the Procedure combo box. This creates a stub function, to which you should add the following code:

```
Private Function UIButtonControl1_ToolTip() As String
    UIButtonControl1_ToolTip= "Full Extent"
End Function
```

4. Click the Visual Basic Editor's File menu, click Close and click Return to ArcMap.
5. Click the Tools menu, click Customize, then click the Commands tab.
6. In the Customize dialog box, click the Commands tab and click the Save in dropdown menu to navigate to the name of your saved project or to Untitled.
7. In the Categories list, choose UIControls and drag the UIButtonControl you created to a toolbar. Close the Customize dialog box.

Try the new command by zooming in on the map and clicking the button. Also, test the ToolTip and description properties. The ToolTip will display if you pause the cursor over the button, while the description, the code in the *message* procedure, will display in the status bar as the cursor moves over the button.

Creating a tool in VBA

As you've seen in the built-in toolbars and menus, users interact with other controls in addition to commands. As part of the customization environment, you can add sophisticated controls to toolbars and menus. A tool is a type of control that allows you to interact with the display. The Identify tool, in ArcMap, is an example of a tool control that shows the attributes of geographic features that you click on in the ArcMap display. In this section of the tutorial, you'll create a UIToolControl to interact with the ArcMap display.

1. Click the Tools menu and click Customize.
2. Click the Commands tab and click in the Save in combo box to locate your project.
3. Choose UIControls from the Categories list.
4. Click New UIControl.
5. In the dialog box that appears, choose UIToolControl as the UIControl Type, then click Create and Edit.

Adding code for the UIToolControl

The application adds an entry in the Object box for the *UIToolControl* and stubs in an event procedure for the *UIToolControls.Select* event. You won't add any code to the *Select* event procedure at this time; instead, select the *MouseDown* event in the Procedures combo box on the right side of the Code window. You'll add code to this event to enable you to drag a rectangle on the screen display; the application will zoom to the rectangle's extent.

1. Add the following code to the *MouseDown* event procedure:

The code on the right assumes the `UIToolControl` was named `UIToolControl1`. If another name was used because this name was already in use, make the necessary changes to the code snippets.

```
Private Sub UIToolControl1_MouseDown(ByVal button As Long, ByVal shift As Long, ByVal x As Long, ByVal y As Long)
    Dim pDoc As IMxDocument
    Dim pScreenDisp As IScreenDisplay
    Dim pRubber As IRubberBand
    Dim pEnv As IEnvelope
    Set pDoc = ThisDocument
    Set pScreenDisp = pDoc.ActiveView.ScreenDisplay
    Set pRubber = New RubberEnvelope
    Set pEnv = pRubber.TrackNew(pScreenDisp, Nothing)
    pDoc.ActiveView.Extent = pEnv
    pDoc.ActiveView.Refresh
End Sub
```

The key line of this procedure is the one that contains the `TrackNew` method, which rubberbands a new shape on the specified screen. The code uses the `Envelope` object that the method returns to set the new extent for the map.

When you selected the `MouseDown` event to add code to, you may have noticed that `UIToolControl` supports several other events. The customization framework handles many of the details of coding for you, so you only have to code the event procedures you need. Later in this chapter, you'll find that this is in contrast to what is required when implementing a tool as part of an ActiveX DLL. A tool is not appropriate for all occasions. You can control when a tool or command is available by adding code to its `Enabled` event.

The code in the `Mouse Down` event will run when you click the mouse down in the display when the tool is active.

2. Add the following code to the `Enabled` event procedure of `UIToolControl1`:

```
Private Function UIToolControl1_Enabled() As Boolean
    Dim pDoc As IMxDocument
    Set pDoc = ThisDocument
    UIToolControl1_Enabled = (pDoc.FocusMap.LayerCount <> 0)
End Function
```

3. Add the following code to the `CursorID` event procedure to control the cursor that appears when you use the tool:

```
Private Function UIToolControl1_CursorID() As Variant
    UIToolControl1_CursorID = 3 ' Crosshair
End Function
```

4. Add a `ToolTip` and message for the tool control as you did for `UIButtonControl` in the steps above.
5. Click the Visual Basic Editor File menu, click Close, then click Return to ArcMap.
6. Click the Tools menu, click Customize, then click the Commands tab.
7. In the Customize dialog box, click the Commands tab and click the Save in dropdown menu to navigate to the name of your project or to Untitled.
8. In the Categories list, choose `UIControls` and drag the `UIToolControl` that you created to a toolbar. Close the Customize dialog box.

Try out the tool by selecting it and dragging a rectangle on the display. You can also see the *Enabled* event procedure code in action if you remove all layers from the map. Once you add data back to the map, the tool will be enabled again.

Changing button properties

You can change the image on any toolbar button or menu command, except for a button that displays a list or a menu when you click it. You can display text, an icon, or both on a toolbar button. You can also display either an icon and text or text only on a menu command. You can change the image that represents the tool and other properties by right-clicking the button.

1. Right-click any toolbar to determine whether a context menu is available and click *Customize* in the context menu that appears.
2. Right-click the button whose properties you want to change.
3. In the context menu that appears, click *Image and Text*. The button now displays the image and the name of the button.
4. Close the *Customize* dialog box.

Congratulations! You now have the basic knowledge to tackle the example code samples located in the appendixes. Along with each of these code samples is a hint about where best to develop the code—in a macro, command, or tool.

Additional information on the VBA environment and developing with VBA and ArcObjects can be found in the appendixes.

VBA is an ideal prototyping environment that provides the means for deploying customizations. In previous sections, you have seen that you can make toolbar customizations and quickly develop modest applications. Many of the developer samples available with ArcGIS are simply VBA code snippets and procedures that you can copy, paste, and run in the VBA development environment.

Developing with VBA does have some disadvantages, however:

- To deploy your customization or code, you need to ship the .mxd file or export the code to text files. This is not very flexible for the end user.
- Your VBA code is also exposed by default. Although you can lock the VBA environment to prevent other users from seeing your code, this also means that they cannot extend your customizations or make any other changes in VBA.

ArcGIS applications are COM-based. This means to extend them, you must use a development language capable of creating COM components.

You can overcome these disadvantages and get access to a richer development experience by using a COM-compliant environment, such as Visual Basic 6, Visual C++ with Active Template Library (ATL), or Visual Studio .NET. These environments can all create COM-compliant components, typically as DLLs or OCXs that contain your customizations. These components simply plug into the ArcGIS Desktop framework, revealing your customizations and making them available for use.

Following are some advantages of building custom components:

- You can integrate a wider range of third party controls and code into your customizations.
- Your code is hidden within the binary-compiled component.
- You can extend and customize virtually every aspect of the ArcGIS Desktop applications.
- They can be easily delivered to end users via custom setup programs.

The disadvantage, of course, is that you must obtain one of these development environments, which are not included with ArcGIS, and learn the particular language syntax, increasing development time.

Before looking at how these components are developed, see the explanation below on what COM is and how these customizations are possible.

COM REVIEW

Of the set of components that make up the ArcGIS Desktop applications, ArcObjects is platform independent and written in C++, which makes use of Microsoft's COM. COM is a standard or protocol that connects one software component, or module, with another. With this protocol it is possible to build reusable software components that can be dynamically interchanged in a distributed system.

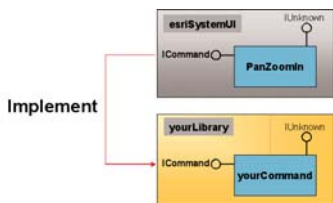
ArcObjects classes based on this model have interfaces that organize the various properties and methods for an object or component. Developing with ArcObjects means developing with interfaces, since all communication between components happens via their interfaces. Interfaces are standardized by COM rules, so it does not matter what language is used to create the component, as long as that language is COM compliant.

See Appendix A, 'Developer environments', for more information on COM.

If you are an existing ArcGIS VBA developer or have worked through the quick-start tutorials in the previous section, you have already been programming with ArcObjects components based on this model and can see how these components work together.

Plugging into ArcGIS Desktop

Using a component in ArcGIS is similar to adding a new CD player to your stereo system. If the plug on your CD player fits the plug on your stereo receiver, the components will be able to work together. Both the stereo and the CD player can be ignorant of how the other works, as long as the communication between them occurs as expected.



Implement ICommand to create your own command.

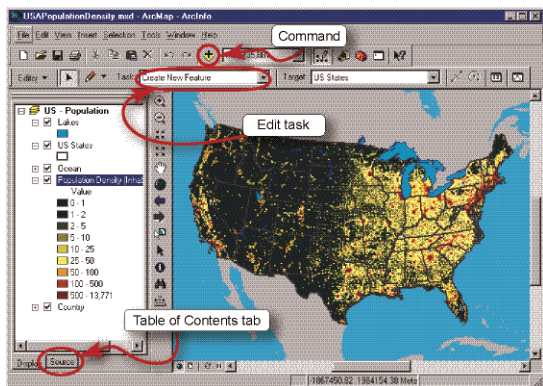
When designing a new ArcGIS component, such as a command or tool, the interface you choose to implement is the equivalent of the type of plug used by a stereo component. If you want your component to plug in as a new command in ArcMap, for example, you must implement the interface that ArcMap expects for commands, *ICommand*.

How does ArcMap know the new command exists?

ArcMap knows what commands are available via component categories. A component category is an operating system registry key that contains component class IDs. Every component within ArcGIS, including components you create, has a globally unique identifier, also called a GUID. Once a component is registered with the operating system, it can be registered to a particular component category. ArcMap and the other desktop applications look for these component categories to work out which commands, tools, edit tasks, and so forth, are available. ArcGIS Desktop applications include the component category manager to help you manage this task. This step can also be automated with setup programs and developer add-ins for the various development environments.

Where can components plug in?

Here are some examples of components that can be added to ArcMap:



ArcMap interface showing where components can plug in

- **Command**—A button, tool, or menu choice. To create this component you must implement *ICommand*, and you may also need to implement *ITool*, *IToolCommand*, or both.
- **Edit Task**—A task that works with the ArcMap Editor in conjunction with the Sketch tool. It must implement *IEditTask*.
- **Table of Contents tab**—An additional tab on the ArcMap table of contents that implements the *IContentsView* interface.
- **Class extension**—A component that works with a dataset (feature class or table) to provide limited custom behavior (attribute calculation or validation, for example). A class extension must implement *IClassExtension*, and it may also implement *IPropertyInspector*, *IObjectClassValidation*, or several others.

What can components do?

Anything you can do in VBA, you can do with a component. However, the reverse is not true, which is one advantage of using a custom component. You can take any of the tutorial samples you have seen so far or any of the illustrated code samples in the appendixes and easily create a component, such as a command or tool, using one of the development languages. In some cases you may only have to make small changes to the VBA code to make it into a component. For example, if you're developing in Visual Basic 6, the syntax is almost identical to VBA, requiring you only to make minor changes and implement the required interfaces.

It's important to reiterate that if you don't intend to take advantage of the benefits that components offer, such as using the code in other documents or deploying to other users, you should continue to make your customizations within VBA. In most cases it will be less complicated and quicker to develop.

WHICH COMPONENT DEVELOPMENT ENVIRONMENT?

The choice of development environment is not a simple task, and it is influenced by many factors. Many developers will select either Visual Basic 6 or Visual C++, while others will use Visual Studio .NET, Delphi™, C++ Builder, and so on. The primary driving force behind the decision is the experience and skill level of the developers who will write the code. Other issues worth considering are the application requirements, performance, development process, and security of code.

For more information on development environments, see Appendix A, 'Developer environments'.

Performance differences between development languages are not as significant as you might think. Since the majority of the work will be performed within the ArcObjects components, which are all written in C++, the developer's customization language is, for the most part, used to control the program flow and user interface interaction. Since Visual Basic 6 uses the same optimized back-end compiler technology that Visual C++ uses, the generated machine code performs at a comparable level. Tests have shown that when performing typical actions on features contained within a database (drawing, querying, editing, and so on), Visual Basic 6 is approximately two percent slower than optimized Visual C++ code, and Visual Basic for Applications is two percent slower than Visual Basic 6.

Performance in Visual Studio .NET can be slightly slower than the other languages due to the intermediate layer, run-time call wrapper (RCW), that .NET code goes through to communicate with ArcGIS COM objects. If you suspect that performance is being hindered by this process, you may be able to restructure your code to make it more efficient. For more information see the 'Performance of ArcObjects in .NET' section in the ArcGIS Developer Help under the .NET development environment.

Visual Basic 6 is a productive tool, especially for user interface development, but there are limitations to what can be done in Visual Basic. In the majority of cases, these limitations will not affect a developer's ability to customize and extend ArcObjects. Many of the limitations that do exist are directly associated with the development environment itself. Visual Basic does not support COM aggregation, for example, so it cannot be used to create custom features. In addition, debugging Visual Basic code is not as flexible as Visual C++. Using Visual Basic 6 in a large development environment with many developers is not as productive as Visual C++ since partial compilations of projects are not supported. If one file is changed in a Visual Basic project, all the files must be recompiled. Since Visual Basic 6 hides much of the interaction with COM inside the Visual Basic Virtual Machine, low-level COM plumbing code cannot be written in Visual Basic.

Visual Studio .NET overcomes many of the limitations of its predecessor, Visual Studio 6, providing a productive IDE, a large supporting class library, more object-oriented functionality, better support for data types, and so forth.

In this section you will learn how to create a component that you can plug into the ArcGIS Desktop applications.

The emphasis here is not how to program in a particular language or how to solve a particular ArcObjects problem; there are many code samples as well as the problem-solving guide in the appendixes that illustrate these. This section provides an overview on how to wrap the solution in a component and make it available in ArcGIS. Chapter 5, 'Developer scenarios', shows you how to create specific components using a development environment.

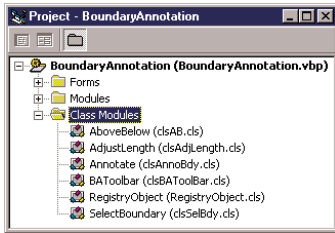
BASIC STEPS IN BUILDING A COMPONENT

The following basic steps are used in building a component. These are explained in more detail in the subsequent paragraphs.

1. Create a new project.
To create a custom component you must use a standalone programming environment. This section will illustrate Visual Basic 6, but you could also use any language that supports COM components.
2. Reference the ArcObject libraries.
Unlike VBA within the desktop applications, Visual Basic will not know about the ArcObject libraries.
3. Implement the required interfaces.
To ensure that your component works with the existing architecture, you must implement the proper interfaces that ArcGIS will expect.
4. Write the implementation code.
Write the code to accomplish the component's purpose.
5. Compile the component as a DLL.
Write your component out as a Dynamic Link Library. You may need to perform some debugging before your component compiles without error.
6. Plug the component into ArcGIS Desktop.
Register the component with the operating system and make it available to ArcGIS through the component categories.
7. Test/Debug/Recompile.
Test the component in the intended ArcGIS Desktop application. If it does not work as expected, you may need to return to your project to fix bugs and recompile your DLL.

1. Create a new project.

To design new ArcGIS Desktop components, you need to make sure you create a new ActiveX DLL Visual Basic project as opposed to a standard executable.



Each class module will become a component or control.

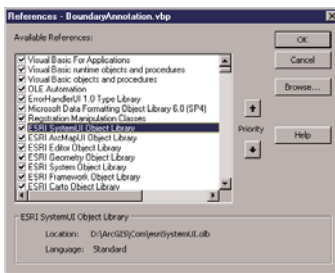
When creating your new VB project, be sure to provide meaningful names for the project and for each class that you create. The name of the project will also be the default name of the DLL when it is compiled. Each class in your project will be an individual component in your DLL. If you create a class called Class1 that is going to be a custom control, for example, 'my new control', it would also be called Class1 inside the DLL.

Components that you want your user to create, such as commands on the user interface, should have their instancing property set to MultiUse.

A single DLL can contain several classes, or components. It is not necessary, therefore, to create a new ActiveX DLL project for each component you want to create. If needed, you could deliver dozens of commands, toolbars, and other components in a single DLL.

2. Reference the ArcObjects libraries.

When you are programming within the ArcGIS Desktop applications with VBA, you do not normally need to explicitly reference the ArcObjects libraries, since most of the ones you will use are already referenced for you. This is not the case when programming in a standalone development environment, such as Visual Basic. Any libraries that are referenced by your code, beyond the standard Visual Basic libraries, need to be explicitly brought into your project.



The Visual Basic References dialog box. Standard and ESRI object libraries have been checked.

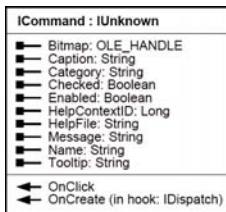
To reference additional class libraries, choose References from the Project menu. Almost all projects need a reference to esriSystemUI since this library contains the plug-in interfaces for commands and tools (for example, *ICommand*, *ITool*).

You will also need to add ArcObjects libraries throughout your development, depending on which parts of the ArcGIS object model you are using.

Visual Basic users may also use the ESRI Automatic References add-in, which provides an easier way to reference the ArcObject libraries. For more information see the section on add-ins within the ArcGIS Developer Help.

3. Determine the required interfaces.

To make sure your component will be understood by the ArcGIS Desktop applications, you need to implement an interface, or interfaces, appropriate for your component.

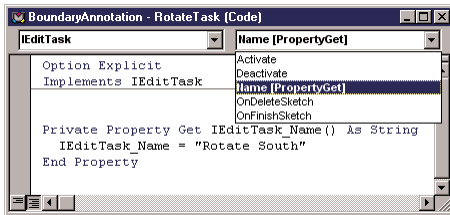


The ICommand interface in esriSystemUI

If ArcMap is to place a new control on its user interface, it needs to be able to apply to the control some basic properties, such as what image should be displayed on the control (Bitmap property), what text should be displayed when the user hovers over the control with the mouse (ToolTip), and most important, what it should do when the control is clicked (*OnClick* event procedure). As a programmer, you address these by writing code for the proper methods and properties. This is performed in the next step.

4. Write component code.

After deciding which interface your component needs to implement, the next step is to write code for every method and property on each interface. To satisfy the rules of COM, this simply means having all the method and property procedure stubs in your class module.



Visual Basic code module showing the implementation of IEditTask and its members

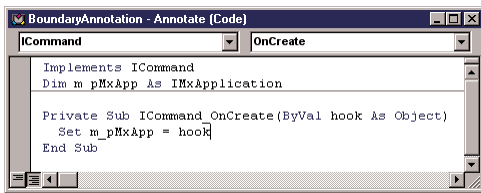
Visual Basic users may also use the Interface Implementer add-in, which automatically adds an interface and stubs out all the required methods and properties.

Upon entering the *implements* statement in the general declarations (top) portion of the class module, you will be able to choose the interface from the object list (upper left) and see all its associated methods and properties in the procedure list (upper right). You may only need to provide code for a handful of these procedures to make your component work as desired. You must, however, have stub code for at least each member of your implemented interfaces before your DLL will compile.

You can now write the code to perform some action, the purpose of the command or tool. For commands, your start point will typically be in the *OnClick* event property; for tools, it may be one of the *OnMouse* event properties. Your code may consist of only a few lines, or it may call other procedures or VB forms as part of a larger application. During this step you will probably need to reference additional ArcObject libraries, as described in Step 2, to support your code.

Referencing the application

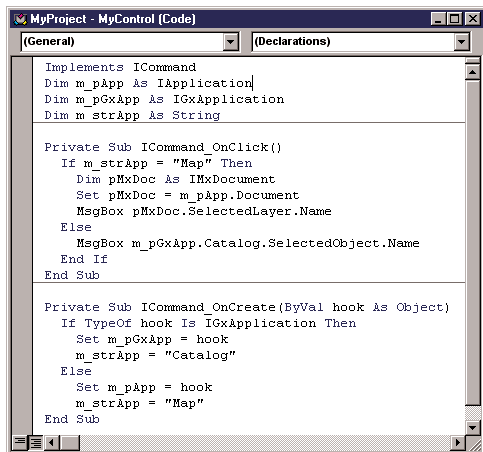
A major difference you will find when programming in a standalone environment, such as Visual Basic as opposed to VBA, is the lack of the available preset variables. When programming in ArcMap or ArcCatalog using VBA, you can jump into your code using the preset Application or ThisDocument variables. To get access to the current application or document in VB, you must use another technique.



The Application object is passed to your component as hook.

The ArcObjects interfaces you will most commonly implement on your components, for example, *ICommand* and *IExtension*, pass a reference to the current Application object into one of their interface members. On the *ICommand* interface, the *OnCreate* event passes in a parameter called *hook* that is referred to as *Object*. This object is the equivalent of the Application preset variable. To use this object throughout your component, you will need to store it as a module-level variable.

The *OnCreate* event fires when a user drags the Custom command from the Customize dialog box to a toolbar or menu or when a map containing the control is opened.



```

MyProject - MyControl (Code)
(General) (Declarations)

Implements ICommand
Dim m_pApp As IApplication
Dim m_pGxApp As IGxApplication
Dim m_strApp As String

Private Sub ICommand_OnClick()
    If m_strApp = "Map" Then
        Dim pMxDoc As IMxDocument
        Set pMxDoc = m_pApp.Document
        MsgBox pMxDoc.SelectedLayer.Name
    Else
        MsgBox m_pGxApp.Catalog.SelectedObject.Name
    End If
End Sub

Private Sub ICommand_OnCreate(ByVal hook As Object)
    If TypeOf hook Is IGxApplication Then
        Set m_pGxApp = hook
        m_strApp = "Catalog"
    Else
        Set m_pApp = hook
        m_strApp = "Map"
    End Sub
    
```

Different tasks are performed in ArcMap and ArcCatalog. TypeOf is used to see where the control was added.

Creating an all-purpose command

It is possible to create a command that works in all the desktop applications. The class shown on the left implements the *ICommand* interface, which is required for all desktop commands.

When the *OnCreate* event fires, the *TypeOf* statement is used to see if the hook is the ArcMap or ArcCatalog application in this example. A string variable is set to record which application the control is being used in.

The *OnClick* event uses the string variable (*m_StrApp*) to see if the control is being used in ArcMap or ArcCatalog. Depending on the application, the control will serve a different purpose: reporting the selected layer in ArcMap or reporting the selected layer in ArcCatalog.

5. Compile your component DLL.

Once you have written all the required code to make your control work inside the desktop applications, you need to compile the project to a DLL on disk. Choose 'Make <project name>' to compile your DLL, specifying an output file location. If there are no syntax errors in your code, the DLL will compile; otherwise, VB will report an error.

Common errors encountered when compiling

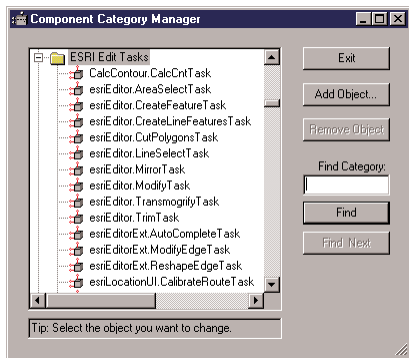
- 'Object module needs to implement <member name> for interface <interface name>'—You did not *stub* all the methods and properties on the interface that you implemented.
- 'User-defined type not defined'—You probably forgot to reference a required library.
- 'Permission denied: <DLL filename>'—The DLL is likely being used, by ArcMap or ArcCatalog, for example. This error might be encountered when you are in the testing/debugging phase of developing your component.

6. Plug the component into ArcGIS Desktop.

Before you can use your custom component in one of the ArcGIS Desktop applications, you need to make sure it is registered with your operating system and registered to the correct component category.

There are several ways in which you can do this:

- **Customize dialog box**—When you use the Add from file button to bring in a component, your component DLL is automatically registered. You cannot use this method for DLLs created with Visual Studio .NET.
- **Component Category Manager**—The Component Category Manager allows you to add and remove ArcGIS components. Components are organized into various categories. By adding your DLL to the proper category, your components will be incorporated into ArcGIS. To add a new edit task that you have created, for example, add your component to the ESRI Edit Tasks category.



The Component Category Manager

This task may be automated to some extent, depending on which development environment you use. In VB6, for example, ESRI provides a Compile and Registry add-in that compiles your project and creates a Windows registry merge file that will place the component in the appropriate component category. For more information about this add-in and others, see the add-ins help in the ArcGIS Developer Help.

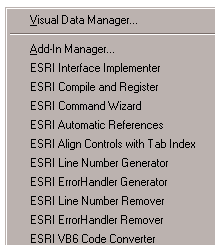
7. Test/Debug/Recompile.

Test your component inside the desktop application. If it does not work as expected or produces an error, debug and recompile the code and test again. The ease with which you can debug your code depends on the development environment and the complexity of your code. Debugging techniques for each development language supported by ArcGIS are discussed in detail in Appendix A, 'Developer environments'.

SUMMARY

The ArcGIS Desktop applications are built using component object technology. This allows you, as a developer, to create your own components that can easily plug into the existing architecture as new controls, tools, menus, and applications, for example.

This section provided an overview of component development options and outlined the basic steps in creating a component to plug into ArcGIS. The developer scenarios in Chapter 5 will show in more detail the steps to create specific components using a development environment.



Visual Basic developers can make use of several add-ins to aid debugging.

4

Licensing and deployment

Some of your customizations may be for personal use on your PC, while others may have been developed for a wider audience. This chapter describes ArcGIS license considerations when developing your applications and illustrates some simple ways to package and deploy your customizations, ranging from VBA macros to more complex applications.

Developing ArcGIS Desktop applications should not be undertaken in isolation from the deployment of the final application. Currently, there are three possible deployments or software products with ArcGIS Desktop: ArcView, ArcEditor, and ArcInfo. There may be more deployment options in the future. As a developer you may need to know what product a user has installed so your code can be robust enough to work on all deployments or at least have the appropriate error checking. This maximizes the potential number of users for your components.

This section outlines how to write code in a way that requires only one code base to support all possible deployment options, both at present and in the future.

DEPLOYMENT OBJECT MODELS

The object models for ArcView, ArcEditor, and ArcInfo are identical. All classes, interfaces, methods, and properties are present in all products. This means that the same DLLs containing the same components with the same GUIDs are installed for all deployments; in other words, code written on one deployment will successfully compile on another. What will differ for the various deployments is the behavior of certain method calls.

All the ESRI-developed components handle the possible deployment options in a unified manner. The functionality available with these different deployments is controlled via a license. This means that if a user installs a new license, the software does not require a reinstallation to access the functionality permitted under the new license.

ArcObjects performs several types of license checking:

- **Application:** Each ArcGIS Desktop application requires a valid application license to run.
- **Extension:** Extension products also have licenses associated with them.
- **Component:** The components within ArcObjects perform license checking.
- **Functional:** When methods are executed, the behavior of the method varies depending on the available licenses.

It is likely that you will be interacting with more than one of these license-checking mechanisms. For instance, you may check for the appropriate component-level license; then, when working with individual methods, you will have to be aware of the license restrictions associated with these methods.

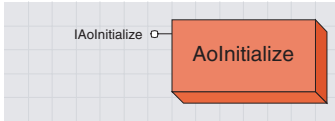
ESRI engineers use the same coding techniques outlined in this chapter to write code that works with the various deployment options of ArcGIS. Using these techniques, you will be able to write your code to handle the various deployment models.

For more information on the ArcGIS license manager, view the license manager reference guide under the ArcGIS > license manager Programs menu.

When executing code, no query interface call will fail because of a license issue, for this would break the rules of COM. If license checking were implemented at the query interface level, depending on licenses being checked in and out, the query interface call may succeed the first time but fail the next or vice versa.

Application license checking

The simplest form of license checking for you to deal with is when your components are running within an ArcGIS Desktop application, since there is little for you to do. The license-checking procedures are contained within the ArcGIS applications, and the fact that your components are initialized means that the user has a valid ArcGIS license. However, determining what license is currently checked out by the user can be useful for working with licensing at the functional level.



IAoInitialize : IUnknown	
← CheckInExtension (in extensionCode: esriLicenseExtensionCode): esriLicenseStatus	Check in an extension.
← CheckOutExtension (in extensionCode: esriLicenseExtensionCode): esriLicenseStatus	Check out an extension.
← Initialize (in ProductCode: esriLicenseProductCode): esriLicenseStatus	This must be called before any other ArcObjects components are created to initialize product Code. If called a second time during the lifetime of an executable with a new product code, it will return esriLicenseAlreadyInitialized.
← InitializeProduct: esriLicenseProductCode	Retrieves the product code where the application has been initialized.
← IsExtensionCheckedOut (in extensionCode: esriLicenseExtensionCode): Boolean	Specifies if the extension is checked out.
← IsExtensionCodeAvailable (in ProductCode: esriLicenseProductCode, in extensionCode: esriLicenseExtensionCode): esriLicenseStatus	Determines if the product code is available. If it is, it then determines if the extension code is available.
← IsProductCodeAvailable (in ProductCode: esriLicenseProductCode): esriLicenseStatus	Determines if the product code is available.
← Shutdown	The Shutdown method. This should be the last call to ArcObjects in an application.

To determine the level of license currently in use, use the *IAoInitialize* coclass and the *Initialized* method on its *IAoInitialize* interface.

Private Function GetLicenseLevel() As String

```

Dim pLicense As IAoInitialize
Set pLicense = New AoInitialize

Select Case pLicense.InitializedProduct
Case esriLicenseProductCodeArcView
    GetLicenseLevel = "ArcView"
Case esriLicenseProductCodeArcEditor
    GetLicenseLevel = "ArcEditor"
Case esriLicenseProductCodeArcInfo
    GetLicenseLevel = "ArcInfo"
End Select
End Function
    
```

Extension license checking

ESRI desktop extension products, such as 3D Analyst or Spatial Analyst, provide additional functionality and components for ArcGIS Desktop users and developers. To use an extension, a valid extension license must be available. If you are using the desktop interface, a license is checked out when you select an extension in the Extension Manager dialog box. When you are developing, however, you must ensure the appropriate extension is checked out before making calls to

If you are developing an ArcGIS Engine component, you may use `AoInitialize` to check out an extension instead.

objects within that extension, since you cannot guarantee the user has checked out an extension before running your code. For example, if the following lines of code are executed in a VBA macro, an error will be raised when the *StartExporting* method is executed, since an ArcPress license has not been checked out. The method calls do not attempt to check one out; they only ensure that one already has been checked out. This gives you license usage control.

```
Dim pExporter As IExporter
Set pExporter = New ArcPressExporterJPEG
...
Dim hDc As OLE_HANDLE
hDc = pExporter.StartExporting
```

For the above code to execute without the license error, the ArcPress extension must be initialized before the call to the *StartExporting* method. The function below shows how to check out an ArcPress license:

```
Public Function GetArcPressLicense() As Boolean
    Dim pUid As UID
    Set pUid = New UID
    pUid.Value = "esriOutputExtensions.ArcPressExtension"
```

```
    Dim pExtAdmin As IExtensionManagerAdmin
    Set pExtAdmin = New ExtensionManager
    'Necessary in standalone application
    pExtAdmin.AddExtension pUid, 0
```

```
    Dim pExtManager As IExtensionManager
    Set pExtManager = pExtAdmin
    Dim pExtConfig As IExtensionConfig
    Set pExtConfig = pExtManager.FindExtension(pUID)
```

```
    If (Not pExtConfig.State = esriESUnavailable) Then
        On Error Resume Next
        'Check the license out. Enabling the extension checks out a license.
        pExtConfig.State = esriESEnabled
        'Return TRUE if the license was checked out successfully
        GetArcPressLicense = (pExtConfig.State = esriESEnabled)
    End If
```

```
    If (Not GetArcPressLicense) Then _
        MsgBox "No ArcPress licenses available"
End Function
```

Assuming that the process of exporting only requires access to the license for a short time, the license should be released upon completion of the export. Releasing the license means that another user can export using the same license; the only restriction is that the other user cannot export at the same time, assuming you only have one ArcPress license.

```
Public Sub ReleaseArcPressLicense()
    Dim pUid As UID
    Set pUid = New UID
    pUid.Value = "esriOutputExtensions.ArcPressExtension"
```

A license will fail to check out if you have exceeded the number of licenses available for a product.

Note that license checking is per machine rather than per process. Checking out the same license in different processes on the same machine will only check out one license.


```

Dim pExtManager As IExtensionManager
Set pExtManager = New ExtensionManager

Dim pExtConfig As IExtensionConfig
Set pExtConfig = pExtManager.FindExtension(pUID)

If (Not pExtConfig.State = esriESUnavailable) Then
    pExtConfig.State = esriESDisabled
End If
End Sub

```

If you are developing an extension to which you want to add license checking in a way similar to ESRI, you must follow certain rules when dealing with the configuration state of your extension. These rules are outlined in the 'Framework Library' reference within the ArcGIS Developer Help.

Component license checking

When embedding ArcObjects components within another application, careful thought must be given to license issues.

Before calling any other ArcObjects code, you must first initialize the application with a suitable license in order for it to run successfully. Failure to do so will result in application errors.

Initialization is performed with the *Initialize* method on the *IAoInitialize* interface and establishes the product level—for example, ArcView, ArcEditor, or ArcInfo—for the duration of the application. The product license determines the functionality the application will be able to access. Once the product license has been initialized, it cannot be changed for the duration of the application's life, as it is not possible to reinitialize the application.

The following Visual Basic 6 code shows an example of initializing an application during the form load procedure.

```

Option Explicit
Private m_pAoInitialize As IAoInitialize

Private Sub Form_Load()
    'This sample is designed to perform license initialization on a system
    'that may have access to a floating license. An ArcEditor license will be
    'used.

    Dim licenseStatus As esriLicenseStatus
    licenseStatus = CheckOutLicenses(esriLicenseProductCodeArcEditor)

    'Take a look at the licenseStatus to see if it failed
    'Not licensed
    If (licenseStatus = esriLicenseNotLicensed) Then
        MsgBox "You are not licensed to run this product"
        Unload Form1
    'The licenses needed are currently in use
    ElseIf (licenseStatus = esriLicenseUnavailable) Then
        MsgBox "There are insufficient licenses to run"
        Unload Form1
    End If
End Sub

```

For more information on embedding ArcObjects components within other applications or to create standalone applications, see the 'ArcGIS Engine Developer Guide'.

The AoInitialize help topic in the ArcGIS Developer help contains more information and examples.

```

'The licenses unexpectedly failed.
ElseIf (licenseStatus = esriLicenseFailure) Then
    MsgBox "Unexpected license failure please contact your administrator"
    Unload Form1
'Already initialized (Initialization can only occur once)
ElseIf (licenseStatus = esriLicenseAlreadyInitialized) Then
    MsgBox "Your license has already been initialized please check your
implementation"
    Unload Form1
'Everything was checked out successfully.
ElseIf (licenseStatus = esriLicenseCheckedOut) Then
    MsgBox "Licenses checked out successfully"
End If
End Sub

Private Function CheckOutLicenses(productCode As esriLicenseProductCode) As
esriLicenseStatus

    Dim licenseStatus As esriLicenseStatus
    Set m_pAoInitialize = New AoInitialize
    CheckOutLicenses = esriLicenseUnavailable

    'Check the productCode
    licenseStatus = m_pAoInitialize.IsProductCodeAvailable(productCode)
    If (licenseStatus = esriLicenseAvailable) Then
        'Initialize the license
        licenseStatus = m_pAoInitialize.Initialize(productCode)
    End If

    CheckOutLicenses = licenseStatus
End Function

```

Before an application is shut down, the AOInitialize object must be shut down via the *Shutdown* method. This ensures that any ESRI libraries that have been used are unloaded in the correct order. Failure to do this may result in random crashes on exit due to the operating system unloading the libraries in the incorrect order.

The following Visual Basic 6 code shows an example of the *Shutdown* method being called in the form unload procedure associated with the example above.

```

Private Sub Form_Unload(Cancel As Integer)
    'Shutdown
    m_pAoInitialize.Shutdown
End Sub

```

Functional license checking

Interaction with the three previous forms of license checking in ArcObjects is relatively straightforward. Depending on the functionality accessed, the functional license checking is more involved.

A personal geodatabase is stored in the Microsoft Access .mdb format. An enterprise geodatabase is stored within an RDBMS.

The differences between ArcObjects software-based functionality available through ArcGIS deployments are centered on the geodatabase. ArcEditor and ArcInfo products have the same capabilities, and ArcView has reduced functionality.

ArcView can view all supported ArcGIS data sources, but only shapefiles and personal geodatabases can be edited. Geodatabase functionality is further refined to provide a user read access to all geodatabases. What can be created and edited within a personal geodatabase is further refined to prohibit the following:

- Geometric networks
- Feature classes using nonsimple classes (for example, network feature classes and dimension classes), except annotation
- Feature classes with subtypes
- Feature classes participating in a relationship class (for example, feature-linked annotation)
- Tables with subtypes
- Tables participating in a relationship class

Knowing this list of supported functionality will help you make decisions on whether licensing issues are of concern for the components you are developing.

As a developer, you have the choice to write proactive or reactive code when dealing with these functional license checks. Proactive code determines the license that is currently in use, which dictates the flow through the program. Reactive code does not perform up-front checking, but it does perform checks after the methods with license behavior are called. In reality, you will most often employ a mixture of both techniques.

An example of proactive code might involve an application that will display and edit data from a variety of data sources. You might choose to limit the data that a user can add to the application based on the license in use. This can be achieved in conjunction with the *GxDialog* coclass and a selection of *GxObject* filters, as illustrated below:

```
Private Function SelectLicensedEditClasses() As IEnumGxObject
    Dim pGxDialog As IGxDialog
    Set pGxDialog = New GxDialog

    Dim pFilters As IGxObjectFilterCollection
    Set pFilters = pGxDialog
    pFilters.RemoveAllFilters
    'Add filters common to all products
    pFilters.AddFilter New GxFilterShapefiles, False
    pFilters.AddFilter New GxFilterPGDBFeatureClasses, False
    pFilters.AddFilter New GxFilterPGDBFeatureDatasets, False
    pFilters.AddFilter New GxFilterPGDBTables, False

    Dim pLicInfo As IAoInitialize
    Set pLicInfo = New AoInitialize
```

```
'Add filters based on product level - ArcEditor, ArcInfo
If ((pLicInfo.InitializedProduct = esriLicenseProductCodeArcEditor) Or _
(pLicInfo.InitializedProduct = esriLicenseProductCodeArcInfo)) Then
pFilters.AddFilter New GxFilterCoverageAnnotationClasses, False
pFilters.AddFilter New GxFilterCoverageFeatureClasses, False
pFilters.AddFilter New GxFilterCoverages, False
pFilters.AddFilter New GxFilterDimensionFeatureClasses, False
pFilters.AddFilter New GxFilterGeometricNetworks, False
pFilters.AddFilter New GxFilterInfoTables, False
pFilters.AddFilter New GxFilterRelationshipClasses, False
pFilters.AddFilter New GxFilterSDEFeatureClasses, False
pFilters.AddFilter New GxFilterSDEFeatureDatasets, False
pFilters.AddFilter New GxFilterSDETables, False
End If
```

```
With pGxDialog
.AllowMultiSelect = True
.Title = "Select Editable data"
.DoModalOpen 0, SelectLicensedEditClasses
End With
End Function
```

Functional changes take two forms. A method either returns an appropriate error HRESULT to signal that there is not an appropriate license available to successfully execute the method, or it returns a successful HRESULT, but the behavior of the method changes to reflect the available licenses.

As an example of the first kind of functional license check, the *Delete* method on the *IDataset* interface may return the HRESULT FDO_E_NO_OPERATION_LICENSE to say that you did not have the correct license to complete the operation. This type of error can be easily found reactively, then reported to the user using an informative message box.

```
Private Function DeleteDataset(pDataset As IDataset) As Boolean
On Error GoTo ErrorHandler

pDataset.Delete
DeleteDataset = True

Exit Function
ErrorHandler:
If (Err.Number = FDO_E_NO_OPERATION_LICENSE) Then
MsgBox "You do not have a license that enables you to delete _
dataset " & pDataset.Name, vbCritical
Else
MsgBox "Error Deleting Dataset " & pDataset.Name & vbCrLf & _
"Error Description : " & Err.Description, vbCritical
End If
End Function
```

The alternative is to determine the license in use and the type of dataset that the user wants to delete, then decide whether or not to allow the *DeleteDataset* function to be called.

The more difficult scenario is when the behavior of a method changes depending on the available licenses. For instance, assume that the user has defined a personal geodatabase using ArcEditor and has a number of classes defined. Two of these feature classes have a relationship class. This means that as long as an ArcEditor or ArcInfo license is used to edit the database, all classes are editable. If an ArcView user starts editing on the database, the start edit operation will succeed for all the classes except the two with the relationship. The method's behavior has changed, but there was no failure HRESULT returned from the method call since it successfully started editing all the other classes. In this case, you must perform another step after calling *StartEdit* to determine whether or not the start edit operation was successful on all classes. If you find that it was not successful, you can retrieve the reason from the database and present that information to the user or perhaps configure your tools accordingly.

```
Private Sub StartEditWithCheck(pWorkspace As IWorkspace)
    Dim pWorkspaceEdit As IWorkspaceEdit
    Set pWorkspaceEdit = pWorkspace
    pWorkspaceEdit.StartEditing True
    Dim pDatasets As IEnumDataset

    Set pDatasets = pWorkspace.Datasets(esriDTFeatureClass)
    pDatasets.Reset

    Dim pDataset As IDataset
    Dim pDatasetEdit As IDatasetEdit
    Set pDatasetEdit = pDatasets.Next

    Dim failedClasses As String
    Do Until (pDatasetEdit Is Nothing)
        If (Not pDatasetEdit.IsBeingEdited) Then
            Set pDataset = pDatasetEdit
            failedClasses = failedClasses & pDataset.Name & vbCrLf
        End If

        Set pDatasetEdit = pDatasets.Next
    Loop
    If (failedClasses <> "") Then _
        MsgBox "Start edit failed for the following classes : " & _
            failedClasses, vbCritical
End Sub
```

The above function can be changed slightly to perform the checking proactively. In the following function, the class is checked to see if it can be edited using its *IDatasetEditInfo* interface. This is the preferred method of checking since there are a number of reasons, in addition to the license issues discussed here, that a user may not be able to start editing a feature class. For more information, see the Geodatabase library overview in the ArcGIS Developer Help.

```
Private Function AllOrNothingStartEdit(pWorkspace As IWorkspace) As Boolean
    Dim pDatasets As IEnumDataset
    Set pDatasets = pWorkspace.Datasets(esriDTFeatureClass)
    pDatasets.Reset
    Dim pDatasetEditInfo As IDatasetEditInfo
    Set pDatasetEditInfo = pDatasets.Next

    Do Until (pDatasetEditInfo Is Nothing)
        If (Not pDatasetEditInfo.CanEdit) Then Exit Function
        Set pDatasetEditInfo = pDatasets.Next
    Loop

    Dim pWorkspaceEdit As IWorkspaceEdit
    Set pWorkspaceEdit = pWorkspace

    pWorkspaceEdit.StartEditing True
    AllOrNothingStartEdit = True
End Function
```

When designing your functionality, being aware of these license issues will help you create a solid application that will work on any deployment of the ArcGIS functionality.

In general, in any application you should always:

- Check for product licensing in custom extensions that depend on other extensions.
- Decide if you want to check out a license for the duration of the application or just for the function use.

Using the following tables will help you decide when it is appropriate to check for license-related HRESULTs. You should not treat this as a fixed list of method calls since changes in ArcGIS deployments may result in changes to the functional license-checking routines.

The following table lists the license-related HRESULTS:

Name	Hexidecimal value	Decimal value
E_GEOSTAT_LICENSENOTAVAILABLE	0x80040301	-2147220735
E_LICENSENOTAVAILABLE	0x80040302	-2147220734
E_RASTER_FILE_LZW_FAILED	0x80041006	-2147217402
E_RASTERENCODER_NO_LICENSE	0x80041001	-2147217407
E_SPATIAL_ANALYST_LICENSENOTAVAILABLE	0x80041068	-2147217304
E_SPATIAL_ANALYST_SHAREDLICENSENOTAVAILABLE	0x8004106A	-2147217302
E_TIN_LICENSE_NOT_AVAILABLE	0x80042B65	-2147210395
FDO_E_LICENSE_FAILURE	0x80040212	-2147220974
FDO_E_NO_EDIT_LICENSE	0x8004021E	-2147220962
FDO_E_NO_OPERATION_LICENSE	0x80040220	-2147220960
FDO_E_NO_SCHEMA_LICENSE	0x8004021F	-2147220961
FDO_E_SE_LICENSE_EXPIRED	0x80041542	-2147216062
FDO_E_SE_LICENSE_FAILURE	0x8004152B	-2147216085
FDO_E_SE_OUT_OF_LICENSES	0x8004152C	-2147216084
GEOCODING_E_NO_LICENSE	0x80040101	-2147221247
LOCATION_E_NO_LICENSE	0x80040210	-2147220976
ROUTEVENT_E_NOT_LICENSED	0x80040224	-2147220956

The following tables list the method calls that can return license-related HRESULTs:

Interface	Method	HRESULT
IArcInfoWorkspace	CreateCoverage	FDO_E_NO_SCHEMA_LICENSE
	CreateInfoTable	FDO_E_NO_SCHEMA_LICENSE
ICheckIn	CheckInFromDeltaFile	FDO_E_NO_OPERATION_LICENSE
	CheckInFromGDB	FDO_E_NO_OPERATION_LICENSE
ICheckInDataSynchronizer	Synchronize	FDO_E_NO_OPERATION_LICENSE
ICheckOut	CheckOutData	FDO_E_NO_OPERATION_LICENSE
	CheckOutSchema	FDO_E_NO_OPERATION_LICENSE
Iclass	AddField	FDO_E_NO_SCHEMA_LICENSE
IDatasetContainer	AddDataset	FDO_E_NO_OPERATION_LICENSE
IDeltaDataChangesInit	Init	FDO_E_NO_OPERATION_LICENSE
IDeltaDataChangesInit2	Init2	FDO_E_NO_OPERATION_LICENSE
Ieditor	StartEditing	FDO_E_NO_EDIT_LICENSE
Iexport	StartExporting	E_LICENSENOTAVAILABLE
IExportDataChanges	ExportDataChanges	FDO_E_NO_OPERATION_LICENSE
IFeatureDataset	CreateFeatureClass	FDO_E_NO_SCHEMA_LICENSE
	CreateGeometricNetwork	FDO_E_NO_SCHEMA_LICENSE
	CreateRelationshipClass	FDO_E_NO_SCHEMA_LICENSE
	Delete	FDO_E_NO_OPERATION_LICENSE
IFeatureWorkspace	CreateAnnotationClass	FDO_E_NO_SCHEMA_LICENSE
	CreateFeatureClass	FDO_E_NO_SCHEMA_LICENSE
	CreateFeatureDataset	FDO_E_NO_SCHEMA_LICENSE
	CreateRelationshipClass	FDO_E_NO_SCHEMA_LICENSE
	CreateTable	FDO_E_NO_SCHEMA_LICENSE
IGeoDBDataTransfer	GenerateNameMapping	FDO_E_NO_SCHEMA_LICENSE
IGPFunction	Execute	FDO_E_NO_OPERATION_LICENSE
	Validate	FDO_E_NO_OPERATION_LICENSE
IImportDataChanges	ImportDataChanges	FDO_E_NO_OPERATION_LICENSE
ILocatorAttach2	AttachLocator	GEOCODING_E_NO_LICENSE
ILocatorExtension	AddLocator	GEOCODING_E_NO_LICENSE
ILocatorLibrary	ReBuildIndexes	GEOCODING_E_NO_LICENSE
ILocatorWorkspace	AddLocatorStyle	GEOCODING_E_NO_LICENSE
	DeleteWorkspace	GEOCODING_E_NO_LICENSE
	UpdateLocator	GEOCODING_E_NO_LICENSE
INetworkCollection2	CreateGeometricNetworkEx	FDO_E_NO_SCHEMA_LICENSE
IObjectClass	DeleteField	FDO_E_NO_SCHEMA_LICENSE
IPluginLicense	CheckExtensionLicense	FDO_E_NO_OPERATION_LICENSE
Iprinter	StartPrinting	E_LICENSENOTAVAILABLE
IRasterBandCollection	SaveAs	E_RASTER_FILE_LZW_FAILED
IRasterWorkspaceEx	CreateRasterCatalog	FDO_E_NO_SCHEMA_LICENSE
IReplicaDataChangesInit	Init	FDO_E_NO_OPERATION_LICENSE
IReplicaValidation	ValidateReplicaPair	FDO_E_NO_OPERATION_LICENSE
IRouteLocatorOperations	LocateLineFeatures	ROUTEEVENT_E_NOT_LICENSED
	LocatePointEvents	ROUTEEVENT_E_NOT_LICENSED
	LocatePointFeatures	ROUTEEVENT_E_NOT_LICENSED
	LocatePolygonFeatures	ROUTEEVENT_E_NOT_LICENSED
IRouteMeasureCalibrator	CalibrateRoutesByDistance	ROUTEEVENT_E_NOT_LICENSED
	CalibrateRoutesByMs	ROUTEEVENT_E_NOT_LICENSED
IRouteMeasureCreator	CreateUsingPointS	ROUTEEVENT_E_NOT_LICENSED
IRouteMeasureCreator2	CreateUsing2Fields2	ROUTEEVENT_E_NOT_LICENSED
	CreateUsingCoordinatePriority2	ROUTEEVENT_E_NOT_LICENSED
IRouteMeasureEvent-Geoprocessor2	Concatenate2	ROUTEEVENT_E_NOT_LICENSED
	Dissolve2	ROUTEEVENT_E_NOT_LICENSED
	Intersect2	ROUTEEVENT_E_NOT_LICENSED
	Union2	ROUTEEVENT_E_NOT_LICENSED
ISceneGraph	AddSimpleActor	E_TIN_LICENSE_NOT_AVAILABLE

Interface	Method	HRESULT
ISubtypes	AddSubtype	FDO_E_NO_SCHEMA_LICENSE
	DeleteSubtype	FDO_E_NO_SCHEMA_LICENSE
	put_DefaultSubtypeCode	FDO_E_NO_SCHEMA_LICENSE
	put_DefaultValue	FDO_E_NO_SCHEMA_LICENSE
	putref_Domain	FDO_E_NO_SCHEMA_LICENSE
ISurface	AsPolygons	E_TIN_LICENSE_NOT_AVAILABLE
	Contour	E_TIN_LICENSE_NOT_AVAILABLE
	ContourList	E_TIN_LICENSE_NOT_AVAILABLE
	ConvertToPolygons	E_TIN_LICENSE_NOT_AVAILABLE
	GetContour	E_TIN_LICENSE_NOT_AVAILABLE
	GetLineOfSight	E_TIN_LICENSE_NOT_AVAILABLE
	GetPartialVolumeAndArea	E_TIN_LICENSE_NOT_AVAILABLE
	GetProjectedArea	E_TIN_LICENSE_NOT_AVAILABLE
	GetSteepestPath	E_TIN_LICENSE_NOT_AVAILABLE
	GetSurfaceArea	E_TIN_LICENSE_NOT_AVAILABLE
	GetVolume	E_TIN_LICENSE_NOT_AVAILABLE
	GetVolumeAndArea	E_TIN_LICENSE_NOT_AVAILABLE
	QueryPixelBlock	E_TIN_LICENSE_NOT_AVAILABLE
ITinAdvanced	ConvertToVoronoiRegions	E_TIN_LICENSE_NOT_AVAILABLE
	MakeEdgeEnumerator	E_TIN_LICENSE_NOT_AVAILABLE
	MakeNodeEnumerator	E_TIN_LICENSE_NOT_AVAILABLE
	MakeTriangleEnumerator	E_TIN_LICENSE_NOT_AVAILABLE
ITinEdit	InitNew	E_TIN_LICENSE_NOT_AVAILABLE
	StartEditing	E_TIN_LICENSE_NOT_AVAILABLE
ITopologyContainer2	CreateTopologyEx	FDO_E_NO_SCHEMA_LICENSE
IVersion	CreateVersion	FDO_E_NO_OPERATION_LICENSE
	Delete	FDO_E_NO_OPERATION_LICENSE
	put_Access	FDO_E_NO_OPERATION_LICENSE
	put_Description	FDO_E_NO_OPERATION_LICENSE
	put_VersionName	FDO_E_NO_OPERATION_LICENSE
IVersionDataChangesInit	Init	FDO_E_NO_OPERATION_LICENSE
IWorkspaceEdit	StartEditing	FDO_E_NO_SCHEMA_LICENSE
	StartEditing	FDO_E_NO_EDIT_LICENSE
IWorkspaceLicense	putref_Domain	FDO_E_NO_SCHEMA_LICENSE

This section describes how you can package your developments and deploy them to other users.

Exactly what must be packaged depends on the type of development—VBA or DLLs; but you should also consider including the following:

- **Object Diagrams**—Since you have developed your code using the same open and extensible architecture that ESRI uses, other developers are free to work with your components in the same way you work with ArcObjects. Object diagrams and help within the DLLs are good ways of supplying developers with information.
- Other files to package can include data files, help files, documentation, and so on.

It is important not to package any of the files within the ArcGIS installation. If you did this and the user uninstalled your software, there would be a danger that some of the files ArcGIS requires to function correctly might be removed.

VBA DEVELOPMENTS

For VBA developments, there are several ways to distribute the code and UI customizations. Recall from Chapter 3, the ‘Storing customizations’ section, that all the code and UI customizations may be packaged in either the document file—for example, a .mxd file for ArcMap—or a template file.

The first method to deploy this information is to simply copy the document and/or template to the appropriate directory on the target machine, as described in Chapter 3.

This method has some disadvantages, however. If your document contains an enabled extension and the target machine does not have that extension installed, the document will not open on the target machine, and the user cannot extract any code from that document. Another disadvantage is that if you are distributing a Normal template, it will overwrite any customizations that already exist on the target machine.

Also, keep in mind that when you save a document, you have a choice of storing absolute or relative pathnames. If you intend to distribute a .mxd file and data to other users, try to use relative pathnames such that when they open the document it will look for the data in a pathname relative to the .mxd file rather than a hard-coded path that may not exist on the target machine.

The second method is to export the VBA code modules to text files, copy those files to the target machine, then import the files into an application’s VBA session.

The disadvantages with this second method include the manual process of exporting and importing potentially large numbers of files and the fact that UI customizations cannot be transferred this way.

You can change the data storage options under the File > Document Properties menu in ArcMap, ArcGlobe, and ArcScene.

DLL DEVELOPMENTS

How you package and deploy your DLL development depends on the API used to create the DLL. For this section .NET DLLs created with the .NET framework and COM DLLs created by languages such as Visual Basic 6 and Visual C++ will be considered.

.NET Framework DLLs

This includes DLLs created with either VB .NET or C# .NET. To utilize these DLLs on a target machine, the .NET framework must be installed. This is available as a free download from Microsoft. In addition, the ESRI interop assemblies must be present, but these are installed with ArcGIS.

The two popular methods to package and deploy .NET customizations are described below.

Just the binaries

When you compile a .NET project, the resulting DLL will be contained within a folder called bin in the project directory. This directory will also contain the Type library file for the project with the extension .tlb.

It is possible to simply give the user a copy of these compiled binaries with instructions on how to register them on the system. This is often the easiest method if you're sharing customizations with colleagues. Many of the developer samples within ArcGIS Developer Help are provided this way.

Before you can use the DLL in a desktop application, you must register the DLL with the operating system, then register the classes it contains with the appropriate component category.

To register the DLL with the operating system, use the following command line syntax:

```
regasm.exe MyCustom.DLL
```

To unregister the DLL, the command is run with the /u option:

```
regasm.exe MyCustom.DLL /u
```

If you have used the ESRI Component Category Registrar add-in for Visual Studio .NET, the component category information is embedded within the DLL, and the classes will be automatically registered with the component category when you register the DLL.

If you did not use the add-in during development, the user will have to manually register the classes using the Component Category Manager located at <ArcGIS Install>\bin\categories.exe.

To use the Component Category Manager with .NET DLLs, follow these steps:

1. Open the Component Category Manager.
2. Select the component category to which you want to add the class.
3. Click Add Object, then browse to and select the compiled .tlb file, not the DLL.
4. Select the classes to be registered in the selected component category and click OK.

The .NET framework must be installed for the ESRI interop assemblies to be installed.

You do not have to register a .NET DLL with the operating system on the machine on which you build it since that occurs during the build process.

Repeat these steps for additional classes within the DLL that need to be registered in other component categories.

Using an installation program

A setup package is useful when you want to deploy a large application consisting of several DLLs, associated documents, and data. These packages usually create simple executable files that can be easily deployed and installed by users. These packages also make the uninstallation of your customization much easier.

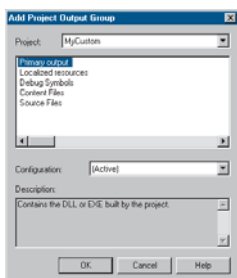
There are many commercial installation packages available that vary in functionality. Visual Studio .NET also includes a straightforward Setup and deployment option that will be shown in the following example.

To create a setup program for your .NET customization use the following steps:

1. Open your customization solution in Visual Studio .NET.
2. From the File menu, select New Project and click the Setup and Deployment projects folder in the New Project dialog box. Click the Setup Project template, accept the default name and location, and click the Add to Solution button.



Step 2



Step 3

This creates a new project in your solution that builds your setup files.

3. Right-click the application folder, point to Add, then click Project Output. In the Add Project Output Group dialog box, click your customization project in the Project dropdown list. Click Primary output in the pane below. Leave the configuration as Active and click OK.

This will add the primary output, the DLL, to the setup project and any dependents, in this case the referenced libraries.

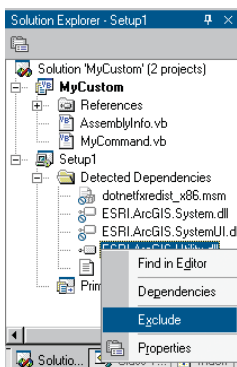
4. Exclude the ESRI-referenced libraries from the Detected Dependencies folder in the Solution Explorer by right-clicking each library name and clicking Exclude.

This ensures that the ESRI libraries do not get deployed with the setup. You may also like to add any files, such as release notes, at this stage by right-clicking the setup project and clicking Add.

5. Finally, create the setup files by right-clicking the setup project in the Solution Explorer and clicking Build. This creates the setup files in the setup directory created in Step 2 under the active configuration name, debug or release.

The three files in the Setup directory—setup.exe and the .msi and .ini files—can then be given to users for a seamless install.

For more information on this process and additional options, search for and review ‘Setup, deployment methods’ within Microsoft Developer Network (MSDN) Help, provided with Visual Studio .NET.



Step 4

Com DLLs

These include DLLs created with Visual Basic 6, Visual C++, Delphi, and so forth. The only requirement for installation on the target machine is that it must have ArcGIS Desktop installed.

There are two popular methods for packaging and deploying COM DLLs.

Just the DLL

It is possible to simply give the user a copy of the DLL with instructions on how to register the DLL on the system. Normally, this involves the use of the Windows Utility RegSvr32.EXE. To register a DLL, the user must type a command line similar to that below.

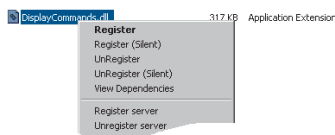
`RegSvr32 MyServer.DLL`

To unregister a server, the command is run with the /U switch.

`RegSvr32 /U MyServer.DLL`

A dialog box appears when the operation completes. When running regsvr32 on several files, it is advisable to run it in silent mode with the /S switch—this disables the dialog box.

Depending on how the DLL was developed, registering the DLL may not be the only task. The coclasses contained with the DLL may have to be added to the appropriate component categories. If ATL was used, as shown in the ATL section, this can be made automatic on server registration. Other alternatives include the facility in the applications for commands; the Category Manager utility application; and the *ComponentCategoryManager* coclass, which is part of the framework subsystem or the creation of a registry script.



Included in the Tools directory of the ArcGIS Developer Kit folder is a small registry script called `reg_in_menu.reg`. The registry script adds options to the Windows Explorer context menu when DLL, EXE, OLB, and OCX files are selected. The five options provide support for registering and unregistering the files. The context menu is shown in the figure to the left.

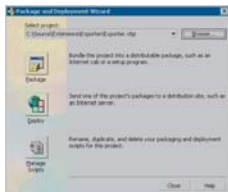
Using registry scripts

After the server is registered on the system, registry scripts provide a good mechanism for adding supplemental information about the server to the registry, including the component category information. These registry scripts can either be written by hand or generated from the Compile and Register Visual Basic add-in. A sample script is shown below. The lines beginning “[HKEY” must all be on one line in the file.

`REGEDIT4`

`; This Registry Script enters CoClasses Into their appropriate Component Category ; Use this script during installation of the components`

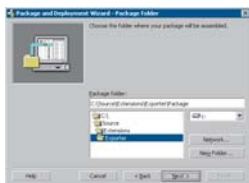
```
; Coclass: prjDisplay.ZoomIn ; CLSID: {FC7EC05F-6B1B-4A59-B8A2-37CE33738728}
; Component Category: ESRI Mx Commands
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{FC7EC05F-6B1B-4A59-B8A2-37CE33738728}\Implemented Categories\{B56A7C42-83D4-11D2-A2E9-080009B6F22B}]
```



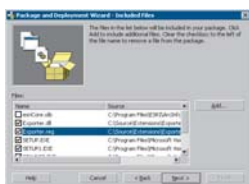
Step 1



Step 2



Step 3



Step 4



Step 5



Step 6

```
; Coclass: prjDisplay.ZoomOut ; CLSID: {2C120434-0248-43DB-AD8E-BD4523A93DF8} ; Component Category: ESRI Mx Commands [HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{2C120434-0248-43DB-AD8E-BD4523A93DF8}\ImplementedCategories\{B56A7C42-83D4-11D2-A2E9-080009B6F22B}]
```

Using an installation program

Most setup packages work well with registry scripts. For example, the Visual Basic Package and Deployment wizard provides a straightforward way of creating setup programs. To create a setup program for your server, follow these steps:

1. Click the Start menu and click the Package and Deployment Wizard. The dialog box to the left is displayed. Click the Visual Basic project to be packaged and choose the package option. This will build the setup program and gather all files required by the setup program into a support directory for easy regeneration of the package. The wizard then performs some checks to ensure that the server created by the Visual Basic project is up-to-date with its source files. If not, you are given the option to recompile the project.
2. Next, choose the package type; this will normally be a Standard Setup Package.
3. The next step allows you to specify the folder where the package is created. This folder will contain the Setup executable and cabinet files and a supporting folder with all the files used to build the package.
4. Ensure that the files list shown doesn't include any ArcGIS installed files and that any other files required by the installation are added. The additional files normally include a registry script to perform advanced registration, along with help files, and so on.
5. The next panel depends on whether a registry file was added in the previous step. If the file was added, the dialog box to the left is shown. If no file was added, go to Step 6. The simplest option is to accept the default. This will cause the registry script to be executed when the setup program has registered the servers on the target machine but will not copy the registry script to the machine.
6. The wizard then asks if one or multiple cabinet files will be created. This depends on whether or not the setup program will span multiple floppy disks.



Step 7



Step 8



Step 9



Step 10

7. Next, follow a couple of panels asking for the Installation Screen title and where on the Windows Start menu the setup program should group files. Often when installing DLLs it is not appropriate to define an entry on the Start menu. Sometimes, even with DLLs, it may be desirable to add access to documents containing help information.
8. The next panel allows the user to define the location of the various files after they have been installed. Various macros are defined that will point to different locations, depending on the configuration of the target machine.
9. The next panel allows files to be marked as shared. Any files of the installation that will be used by other programs or installations must be marked as shared. This ensures that the uninstall program does not remove them automatically, which would break the other programs.
10. Finally, the Finish panel is displayed. Click Finish to assemble the package. The three files in the package directory—setup, cabinet, and list files—can then be given to third parties for a seamless install.

This is just one method of packaging COM developments. Whatever method you use, the setup procedure must be as simple as possible and involve as few decisions as possible to avoid user frustration.

5

Developer scenarios

Throughout this book, you have been introduced to several programming concepts and patterns, as well as some APIs. This chapter contains examples of developer scenarios that build applications using ArcGIS Desktop that apply these concepts and use these APIs. Each scenario is available, in a completed state, with the ArcGIS developer samples installed as part of the ArcGIS Desktop Developer Kits.

PROJECT DESCRIPTION

This scenario is for ArcGIS Desktop developers who need to create and deploy a simple toolbar containing a command, tool, and menu that can plug into the ArcGIS Desktop application framework.

The scenario develops these controls and adds them in stages to a toolbar. If your customization only requires one of these controls, for example—a command, you can look for it under the appropriate heading.

The emphasis in this scenario is how you create the components that plug into the framework rather than any particular ArcObjects solution.

CONCEPTS

There are some important concepts you should understand before following this scenario. These are described below.

Commands

All the user interface controls in the ArcGIS Desktop applications are commands. The simplest type of command is a button that generally appears as an icon on a toolbar, but it may also be placed on menus by the user. In both cases an action occurs when the button is clicked, running the code within the *OnClick* procedure.

Commands are simply components within a Dynamic Link Library that implement the *ICommand* interface referenced from the *esriSystemUI* library. Commands are registered in the component category for the application in which they are designed to run. In ArcMap they run as ESRI Mx Commands, in ArcCatalog is ESRI Gx Commands, in ArcScene is ESRI Sx Commands, and in ArcGlobe is ESRI GMx Commands.

Tools

Tools are similar to button commands, but they are designed to interact with the application's display. The *Zoom In* command is a good example of a tool—you click or drag a rectangle over a map before the display is redrawn to show the map contents in more detail. Tools can react to a number of events, including mouse up/down, key up/down, and a double click.

Tools are simply components within a Dynamic Link Library that implement the *ITool* and *ICommand* interfaces referenced from the *esriSystemUI* library. Tools, like commands, are registered in the component category for the application in which they are designed to run. In ArcMap they run as ESRI Mx Commands, in ArcCatalog as ESRI Gx Commands, in ArcScene as ESRI Sx Commands, and in ArcGlobe as ESRI GMx Commands.

Menus

Menus are an alternative to commands for performing an action. As an interface, they contain calls to existing commands that may be defined in the same or other DLLs. Menus can be stacked, employing a pull-right feature, by calling a command item that is on another menu. The *File*, *Edit*, and *View* menus on the ArcMap Standard toolbar are examples of menus.

Menus are simply components within a Dynamic Link Library that implement the *IMenuDef* interface referenced from the *esriSystemUI* library. Menus may also implement the *IRootLevelMenu* interface to appear under the menu's category in the Customize dialog box. Menus are registered in the component category for the application in which they are designed to run. In ArcMap this is ESRI Mx CommandBars, ArcCatalog as ESRI Gx CommandBars, ArcScene as ESRI Sx CommandBars, and ArcGlobe as ESRI GMx CommandBars.

Context menus are slightly different in that they implement the *IShortCutMenu* interface to distinguish them from other menus. They are not discussed in this scenario, however.

Toolbars

Toolbars are simply components within a Dynamic Link Library that implement the *IToolBarDef* interface referenced from the *esriSystemUI* library. They simply act as containers for other controls. You can programmatically or manually add commands, tools, and menus to toolbars. Toolbars are registered in the component category for the application in which they are designed to run. In ArcMap they run as ESRI Mx CommandBars, in ArcCatalog as ESRI Gx CommandBars, in ArcScene as ESRI Sx CommandBars, and in ArcGlobe as ESRI GMx CommandBars.

You can also enable toolbars to display when the desktop application is first run, so the user does not have to manually display it. For more information, see the COM category registration functions step in this section and the ArcGIS Developer Help *IToolBarDef* interface.

REQUIREMENTS

The requirements for working through this scenario are that you have ArcGIS Desktop installed and running.

The IDE used for this example is Visual Basic Studio .NET 2003, and all IDE-specific steps will assume this is the IDE you are using.

It is also recommended that you read the .NET language section in Appendix A, 'Developer environments'.

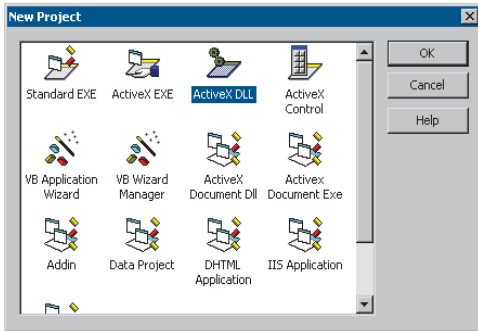
ADDITIONAL RESOURCES

The completed code for this scenario can be found in ArcGIS Developer Help under ArcGIS Desktop > Developer Guide > Developer Scenarios and on disk in the `\DeveloperKit\samples\Developer_Guide_Scenarios\ArcGIS Desktop` directory.

IMPLEMENTATION

In this example you will create a command for ArcMap that toggles the visibility of the selected layer. Later, this command will be added to a toolbar. The code for this example is written in Visual Basic .NET.

Creating a new project

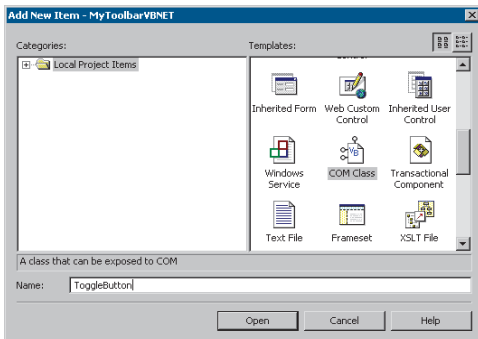


1. Start Visual Studio .NET.
2. Click File, then New, and click Project.
3. In the New Project dialog box, click Visual Basic Projects in the left pane.
4. Click Class Library in the right pane.
5. Type “MyToolbarVBNET” for the Name and browse to the location to which you want to save the project.
6. Click OK, and the new project will be created for you.

The DLL you compile will have the same name as the project, so choose a name representative of the application you are creating. Remember, though, that you can have many commands as classes within the project.

Creating a new COM class for the command

The new project will contain a default Visual Basic class, Class1.vb. You must, however, create and work with a COM class to communicate with the ArcObjects COM framework within ArcGIS Desktop.

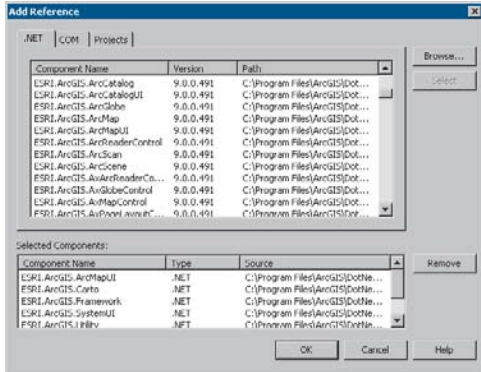


1. In the Solution Explorer, right-click MyToolbarVBNET project, click Add, then click Add New Item.
2. In the Add New Item dialog box, scroll down the right pane and click COM Class. In the Name text box, type “ToggleButton”. Click Open to add the new class to the project.
3. In the Solution Explorer, right-click the existing class (Class1.vb), and click Delete.

The new COM class will have a new GUID and an empty subroutine called New. The project will also be registered for COM interop.

Referencing ESRI libraries in your project

To program using ArcObjects, you will need to add references to the ESRI libraries.



1. In the Solution Explorer, right-click References and click Add References.
2. In the Add Reference dialog box, click the .NET tab, then double-click the following assemblies:
 - ESRI.ArcGIS.ArcMapUI
 - ESRI.ArcGIS.Carto
 - ESRI.ArcGIS.Framework
 - ESRI.ArcGIS.SystemUI
 - ESRI.ArcGIS.Utility
 - System.Drawing.dll
3. Click OK to close the dialog box and add the libraries to your solution.

Adding Imports statements

1. In the Solution Explorer, double-click ToggleButton.vb to open its Code window.
2. At the top of the Code window, before the beginning of the class declaration, add the following lines of code:

```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.ArcMapUI
Imports ESRI.ArcGIS.Framework
Imports ESRI.ArcGIS.Utility.BaseClasses
Imports ESRI.ArcGIS.Utility.CATIDs
```

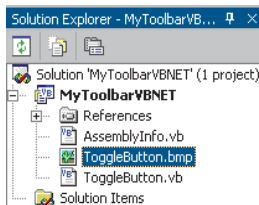
```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.ArcMapUI
Imports ESRI.ArcGIS.Framework
Imports ESRI.ArcGIS.Utility.BaseClasses
Imports ESRI.ArcGIS.Utility.CATIDs
```

You will notice that the *Imports* statements work with IntelliSense.

Adding a bitmap

Add a bitmap file to your project to be used as the icon for the toggle button command, which will be set in the next step.

1. In the Solution Explorer window, right-click the MyToolbarVBNET project, click Add, then click Add Existing Item.
2. In the Add Existing Item dialog box, click the Files of Type pulldown menu and click Image Files.
3. Browse to the \DeveloperKit\Samples\Developer Guide Scenarios\ArcGIS Desktop\Toolbar under your ArcGIS install, click ToggleButton.bmp, then click OK to copy it to your project directory.
4. In the Solution Explorer window, make sure you've selected the new bitmap, and in the Properties window below the Solution Explorer, change the Build Action property to Embedded Resource.



Inheriting the BaseCommand abstract class

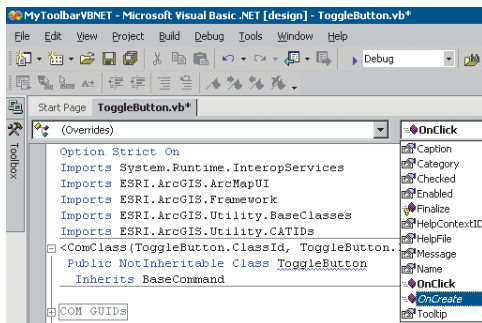
The BaseCommand class provided by ESRI in the Utility library allows you to create commands quicker and simpler than by directly implementing the *esri.SystemUI.ICommand* interface.

1. In the ToggleButton.vb Code window, specify that the ToggleButton class inherits from the BaseCommand abstract class as shown below, then add the NotInheritable class modifier.

This code replaces the existing public class definition.

```
Public NotInheritable Class ToggleButton
    Inherits BaseCommand
```

The NotInheritable class modifier states that a class cannot be inherited from. As you are not designing a class for this purpose, it is prudent to add this modifier to prevent other classes from inheriting this class.



Overriding members of the BaseCommand ICommand interface

1. In the Wizard bar at the top of the ToggleButton class Code window, click the left pulldown menu, and click Overrides.
2. In the right pulldown menu, click OnCreate. A code stub for the overridden OnCreate method is added to the Code window.
3. Repeat Steps 1 and 2, but this time click OnClick and Enabled.
4. Press Ctrl + Shift + S to save all the files in your project, or click File > Save All.

Adding code to the overridden members of BaseCommand

You will now start adding VB .NET code to the methods you have overridden.

1. Add the following member variables to the class as shown.

```
Public NotInheritable Class ToggleButton
    Inherits BaseCommand
    Private m_pApp As IApplication
    Private m_pMxDoc As IMxDocument
```

2. In the ToggleButton class Code window, scroll down to find the overridden OnCreate method, and add the code shown below.

```
Public Overrides Sub OnCreate(ByVal hook As Object)
    If Not (hook Is Nothing) Then
        If TypeOf (hook) Is IApplication Then
            m_pApp = CType(hook, IApplication)
            m_pMxDoc = CType(m_pApp.Document, IMxDocument)
        End If
    End If
End Sub
```

3. Scroll down again to find the *OnClick* method, and add the following code:

```
Public Overrides Sub OnClick()
    m_pMxDoc.SelectedLayer.Visible = Not (m_pMxDoc.SelectedLayer.Visible)
    m_pMxDoc.ActiveView.Refresh()
    m_pMxDoc.UpdateContents()
End Sub
```

4. Scroll down to find the Enabled property, and add the following code.

```
Public Overrides ReadOnly Property Enabled() As Boolean
    Get
        Return (Not (m_pMxDoc.SelectedLayer Is Nothing))
    End Get
End Property
```

5. Scroll the Code window upward to find the constructor, sub *New*, for the *ToggleButton* class as shown, and add the following code to the constructor:

```
Public Sub New()
    MyBase.New()
    MyBase.m_caption = "Toggle Layer On/Off"
    MyBase.m_category = "Custom Controls"
    MyBase.m_message = "Toggles the visibility of the selected layer"
    MyBase.m_toolTip = "Layer On/Off"
    MyBase.m_name = "ToggleButton"
    MyBase.m_bitmap = New _
        System.Drawing.Bitmap(GetType(ToggleButton) _
            .Assembly.GetManifestResourceStream("MyCommandVBNET.globe_1.bmp"))
End Sub
```

Adding COM category registration functions

The .NET framework allows you to place COM registry functions within the class code. When the DLL is registered the class will be placed within the specified COM component category. ESRI provides an add-in to add this code.

1. Enable the add-in.
 - Click the Tools menu and click Add-in Manager.
 - Check ESRI ComponentCategoryRegistrar if it is not checked.
 - Click OK to close the dialog box.
2. Scroll the Code window to near the top of the class declaration, and put the cursor just below the COM GUIDs region.
3. Click the Tools menu and click ESRI ComponentCategoryRegistrar.
4. Check the MxCommand box and click OK to add the code. If you have not added a reference to the ESRI.ArcGIS.Utility assembly, the add-in will add the reference. If you have not declared using the namespace of System.Runtime.InteropServices or ESRI.ArcGIS.Utility.CATIDs, the add-in will add the declaration.

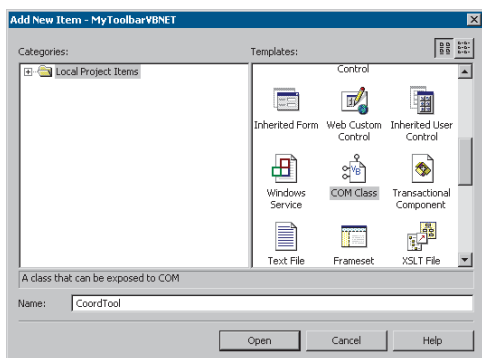
IMPLEMENTATION

In this example you will create a tool for ArcMap that displays the coordinates where the mouse is clicked. The tool also responds to other events, such as right-click and mouse click. Later, this tool will be added to a toolbar. The code for this example is written in Visual Basic .NET.

Opening the toolbar project

If you previously created the MyToolbarVBNET project, open it now.

1. Start Visual Studio .NET.
2. Click File, click Open, and click Project.
3. In the Open Project dialog box, click MyToolbarVBNET.sln from the location to which you saved it.



Creating a new COM class for the tool

1. In the Solution Explorer, right-click MyToolbarVBNET project, click Add, then click Add New Item.
2. In the Add New Item dialog box, scroll down the right pane and click COM Class. In the Name text box below, type “CoordTool”. Click Open to add the new class to the project.

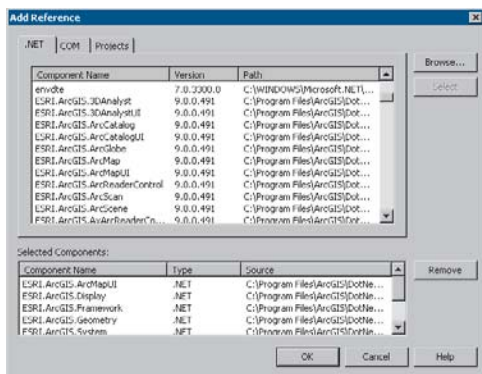
The new COM class will have a new GUID and an empty subroutine called New. The project will also be registered for COM interop.

Referencing ESRI libraries in your project

To program using ArcObjects, you will need to add references to the ESRI libraries.

1. In the Solution Explorer, right-click References and click Add References.
2. In the Add Reference dialog box, click the .NET tab, and double-click the following assemblies:

- ESRI.ArcGIS.ArcMapUI
- ESRI.ArcGIS.Display
- ESRI.ArcGIS.Framework
- ESRI.ArcGIS.Geometry
- ESRI.ArcGIS.System
- ESRI.ArcGIS.SystemUI
- ESRI.ArcGIS.Utility
- System.Drawing.dll
- System.Windows.Forms.dll



3. Click OK to close the dialog box and add the libraries to your solution.

Adding Imports statement

1. In the Solution Explorer, double-click `CoordTool.vb` to open its Code window.
2. At the top of the Code window, before the beginning of the class declaration, add the following lines of code:

```
Option Explicit On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.ArcMapUI
Imports ESRI.ArcGIS.Framework
Imports ESRI.ArcGIS.Geometry
Imports ESRI.ArcGIS.esriSystem
Imports ESRI.ArcGIS.SystemUI
Imports ESRI.ArcGIS.Utility.BaseClasses
Imports ESRI.ArcGIS.Utility.CATIDs
```

You will notice that the *Imports* statements work with IntelliSense.

Adding a bitmap and cursor

Add a bitmap file and a cursor to your project to be used as the icon and cursor for the tool, which will be set in the next step.

1. In the Solution Explorer window, right-click the `MyToolbarVBNET` project, click Add, then click Add Existing Item.
2. In the Add Existing Item dialog box, click the Files of Type pulldown menu and choose Image Files.
3. Browse to the `\DeveloperKit\Samples\Developer Guide Scenarios\ArcGIS Desktop\Toolbar` under your ArcGIS install, and click `CoordTool.bmp`, then click OK to copy it to your project directory.
4. In the Solution Explorer window, make sure you've selected the new bitmap, and in the Properties window below the Solution Explorer, change the Build Action property to Embedded Resource.
5. Repeat Steps 3 and 4 for the `3dsmove.cur` file.

Inheriting the BaseTool abstract class

The `BaseTool` class provided by ESRI in the Utilities library allows you to create commands more quickly and simply than directly implementing the *esriSystemUI.ITool* interface.

1. In the `CoordTool.vb` Code window, specify that the *CoordTool* class inherits from the *BaseCommand* abstract class as shown below and add the *NotInheritable* class modifier.

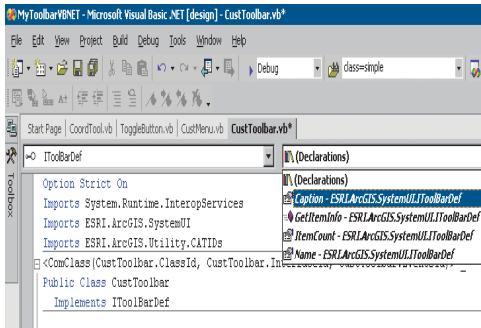
This code replaces the existing public class definition.

```
Public NotInheritable Class CoordTool
    Inherits BaseTool
```

The *NotInheritable* class modifier states that a class cannot be inherited from. As you are not designing a class for this purpose, it is prudent to add this modifier to prevent other classes from inheriting this class.

Overriding members of the BaseTool ITool interface

1. In the Wizard bar at the top of the MyTool class Code window, click the left pulldown menu, and click Overrides.



2. In the right pulldown menu, click Deactivate. A code stub for the overridden *Deactivate* method is added to the Code window.

3. Repeat Steps 1 and 2, but this time for each of the following:

- Enabled
- OnClick
- OnContextMenu
- OnCreate
- OnDbClick
- OnMouseDown

4. Press **Ctrl + Shift + S** to save all the files in your project, or click **File > Save All**.

Adding code to the overridden members of BaseCommand

You will now start adding VB .NET code to the methods you have overridden.

1. Add the following member variables to the class as shown:

```
Public NotInheritable Class CoordTool
    Inherits BaseCommand
    Private m_pApp As IApplication
```

2. In the CoordTool class Code window, use the pulldown menus to find the overridden *Deactivate* method, and add the code shown below.

```
Public Overrides Function Deactivate() As Boolean
    ' Deactivate the tool. If set to False (the default),
    ' you cannot interact with any other tools because this tool
    ' cannot be interrupted by another tool
    Return True
End Function
```

3. Find the overridden *Enabled* property, and add the code shown below.

```
Public Overrides ReadOnly Property Enabled() As Boolean
    ' Add some logic here to specify when the command should
    ' be enabled. In this example, the command is always enabled.
    Get
        Return True
    End Get
End Property
```

4. Find the overridden *OnClick* method, and add the code shown below.

```
Public Overrides Sub OnClick()
    ' Add some code to do some action when the command is clicked. In this
    ' example, a message box is displayed.
    MsgBox("Clicked on the Coordinate display tool.")
End Sub
```

The `OnContextMenu` method allows you to create context menus for your tools.

5. Find the overridden `OnContextMenu` method, and add the code shown below.

```
Public Overrides Function OnContextMenu(ByVal X As Integer, ByVal Y As Integer) As Boolean
    ' Add some code to show a custom context menu when there is a right
    ' click. This example creates a new context menu with one macro item.
    Dim pShortcut As ICommandBar
    Dim pItem As ICommandItem
    ' Create a new context menu.
    pShortcut = m_pApp.Document.CommandBars.Create("MyShortcut",
    esriCmdBarType.esriCmdBarTypeShortcutMenu)
    ' Add an item to it.
    pItem = pShortcut.CreateMacroItem("MyMacro", 4)
    ' Display the menu.
    pShortcut.Popup()
    ' Let the application know that you handled the OnContextMenu event.
    ' If you don't do this, the standard context menu will be displayed
    ' after this custom context menu.
    Return True
End Function
```

6. Find the overridden `OnCreate` method, and add the code shown below.

```
Public Overrides Sub OnCreate(ByVal hook As Object)
    ' The hook argument is a pointer to Application object.
    ' Establish a hook to the application.
    m_pApp = hook
End Sub
```

7. Find the overridden `OnDoubleClick` method, and add the code shown below.

```
Public Overrides Sub OnDoubleClick()
    ' Add some code to do some action on double-click.
    ' This example makes the built-in Select Graphics Tool the active tool.
    Dim pSelectTool As ICommandItem
    Dim pCommandBars As ICommandBars
    ' The identifier for the Select Graphics Tool
    Dim u As New UID
    u.Value = "{C22579D1-BC17-11D0-8667-0000F8751720}"
    'Find the Select Graphics tool.
    pCommandBars = m_pApp.Document.CommandBars
    pSelectTool = pCommandBars.Find(u)
    'Set the current tool of the application to be the Select Graphics tool.
    m_pApp.CurrentTool = pSelectTool
End Sub
```

8. Find the overridden *OnMouseDown* method, and add the code shown below.

```
Public Overrides Sub OnMouseDown(ByVal Button As Integer, ByVal Shift As Integer, ByVal X As Integer, ByVal Y As Integer)
    ' Add some code to do some action when the mouse button is pressed.
    ' This example displays the X and Y coordinates of the
    ' left mouse button click in the statusbar message in ArcMap.
    ' Button, X, and Y are passed in as arguments to this subprocedure.

    ' Check to see if left button is pressed
    If Button = 1 Then
        ' Convert x and y to map units. m_pApp is set in ICommand_OnCreate.
        Dim pPoint As IPoint
        Dim pMxApp As IMxApplication
        pMxApp = m_pApp
        pPoint = pMxApp.Display.DisplayTransformation.ToMapPoint(X, Y)
        ' Set the statusbar message.
        m_pApp.StatusBar.Message(0) = Str(pPoint.X) & ", " & Str(pPoint.Y)
    End If
End Sub
```

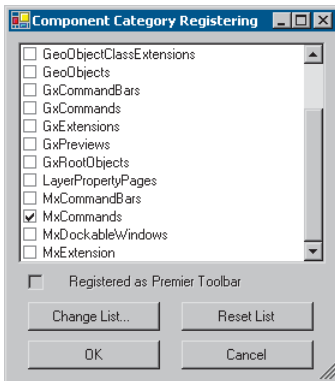
9. Find the Sub New constructor and add the code shown below.

```
Public Sub New()
    MyBase.New()
    MyBase.m_caption = "Display Coordinates"
    MyBase.m_message = "Returns coordinates"
    MyBase.m_toolTip = "Display Coordinates"
    MyBase.m_name = "CoordTool"
    MyBase.m_category = "Custom Controls"
    MyBase.m_bitmap = New
    System.Drawing.Bitmap((GetType(CoordTool).Assembly.GetManifestResourceStream("MyToolbar\BNET.bmp")))
    MyBase.m_cursor = New
    System.Windows.Forms.Cursor((GetType(CoordTool).Assembly.GetManifestResourceStream("MyToolbar\BNET.3dmove.cur")))
End Sub
```

Adding COM category registration functions

The .NET framework allows you to place COM registry functions within the class code. When the DLL is registered the class will be placed within the specified COM component category. ESRI provides an add-in to add this code.

1. Enable the add-in.
 - Click the Tools menu and click Add-in Manager.
 - Check the ESRI ComponentCategoryRegistrar box if it is not checked.
 - Click OK to close the dialog box.
2. Scroll the Code window to near the top of the class declaration, and put the cursor just below the COM GUIDs region.
3. Click the Tools menu and click ESRI ComponentCategoryRegistrar.
4. Check the MxCommand box and click OK to add the code. If you have not added a reference to the ESRI.ArcGIS.Utility assembly, the add-in will add the reference.



IMPLEMENTATION

In this example you will create a menu for ArcMap that contains several existing ArcMap commands. This menu will be later added to a toolbar. The code for this example is written in Visual Basic .NET.

Opening the toolbar project

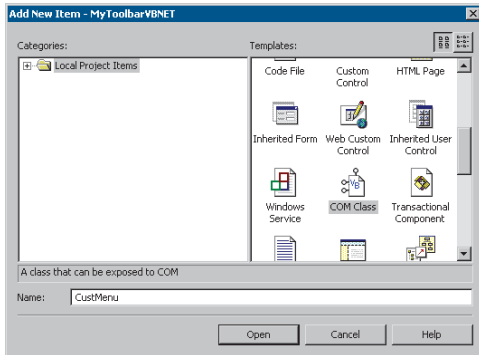
Open the previously created MyToolbarVBNET project.

1. Start Visual Studio .NET.
2. Click File, click Open, then click Project.
3. In the Open Project dialog box, click MyToolbarVBNET.sln from the location in which you saved it.

Creating a new COM class for the menu

1. In the Solution Explorer, right-click the MyToolbarVBNET project, click Add, then click Add New Item.
2. In the Add New Item dialog box, scroll down the right pane and click COM Class. In the Name text box, type “CustMenu”. Click Open to add the new class to the project.

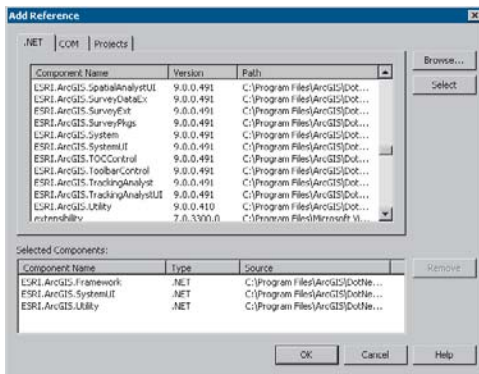
The new COM class will have a new GUID and an empty subroutine called New.



Referencing ESRI libraries in your project

To program using ArcObjects, you will need to add references to the ESRI libraries.

1. In the Solution Explorer, right-click References and click Add References.
2. In the Add Reference dialog box, click the .NET tab and double-click the following assemblies:
 - ESRI.ArcGIS.Framework
 - ESRI.ArcGIS.SystemUI
 - ESRI.ArcGIS.Utility
3. Click OK to close the dialog box and add the libraries to your solution.



Adding Imports statements

1. In the Solution Explorer, double-click *CustMenu.vb* to open its Code window.
2. At the top of the Code window, before the beginning of the class declaration, add the following lines of code:

```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.SystemUI
Imports ESRI.ArcGIS.Framework
Imports ESRI.ArcGIS.Utility.CATIDs
```

```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.SystemUI
Imports ESRI.ArcGIS.Framework
Imports ESRI.ArcGIS.Utility.CATIDs
```

You will notice that the *Imports* statements work with IntelliSense.

Implementing the IMenuDef interface

1. In the *CustMenu.vb* Code window, specify that the *CustMenu* class implements the *IMenuDef* and *IRootLevelMenu* interfaces as shown below. Implementing *IRootLevelMenu* allows this menu to be added to the Menus category in the Customize dialog box.

```
Public Class CustMenu
    Implements IMenuDef
    Implements IRootLevelMenu
```

2. In the Wizard bar at the top of the *CustMenu* class Code window, click the left pulldown menu and click *IMenuDef*.



3. In the right pulldown menu, click the first member of *IMenuDef*, *Caption*. A code stub for the *Caption* property is added to the Code window.

The interface implementer will fully reference the implemented method types by default, as shown below, even though you have added *Imports* statements.

```
Public ReadOnly Property Caption() As String
    Implements ESRI.ArcGIS.SystemUI.IMenuDef.Caption
```

If you want, you can remove the unnecessary namespace details to make your code more readable, as shown below, although you do not have to.

```
Public ReadOnly Property Caption() As String
    Implements IMenuDef.Caption
```

4. Repeat Steps 2 and 3 until you have added code stubs for all the members of the *IMenuDef* interface—*Caption*, *GetItemInfo*, *ItemCount*, and *Name*.
5. Press **Ctrl + Shift + S** to save all the files in your project.

Adding code to the members of IMenuDef

The menu items are defined within the *GetItemInfo* subroutine. Each item is a call to an existing command that is referenced by program ID and the class name. For example, to reference a command in this project, you would use *MyToolbarVBNET.MyCommand*.

1. In the CustMenu class Code window, scroll down to find the *GetItemInfo* method, and add the code shown below.

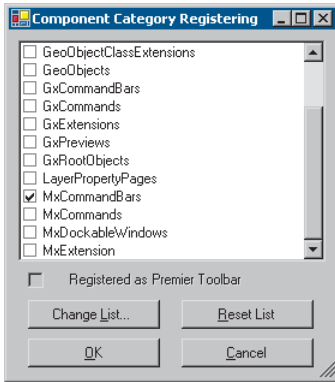
```
Public Sub GetItemInfo(ByVal pos As Integer, ByVal itemDef As
ESRI.ArcGIS.SystemUI.IItemDef)
    Implements ESRI.ArcGIS.SystemUI.IMenuDef.GetItemInfo
    Select Case pos
        Case 0
            itemDef.ID = "esriArcMapUI.AddDataCommand"
            itemDef.Group = False
        Case 1
            itemDef.ID = "esriArcMapUI.FullExtentCommand"
            itemDef.Group = True
        Case 2
            itemDef.ID = "esriArcMapUI.ZoomInFixedCommand"
            itemDef.Group = False
    End Select
End Sub
```

2. Edit the code stubs as shown to complete the *Caption*, *ItemCount*, and *Name* methods as shown to return information about the menu.

```
Public ReadOnly Property Caption() As String Implements IMenuDef.Caption
    Get
        Return "CustMenu-VBNET"
    End Get
End Property

Public ReadOnly Property ItemCount() As Integer Implements
ESRI.ArcGIS.SystemUI.IMenuDef.ItemCount
    Get
        Return 3
    End Get
End Property

Public ReadOnly Property Name() As String Implements
ESRI.ArcGIS.SystemUI.IMenuDef.Name
    Get
        Return "CustMenu-VBNET"
    End Get
End Property
```



Adding COM category registration functions

The .NET Framework allows you to place COM registry functions within the class code. When the DLL is registered the class will be placed within the specified COM component category. ESRI provides an add-in to add this code.

1. Enable the add-in.
Click the Tools menu and click Add-In Manager.
Check the ESRI ComponentCategoryRegistrar box if it is not checked.
Click OK to close the dialog box.
2. Scroll the Code window to near the top of the class declaration, and put the cursor just below the COM GUIDs region.
3. Click the Tools menu and click ESRI ComponentCategoryRegistrar.
4. Check MxCommandBar and click OK to add the code. If you have not added a reference to the ESRI.ArcGIS.Utility assembly, the add-in will add the reference. If you have not declared using the namespace of System.Runtime.InteropServices or ESRI.ArcGIS.Utility.CATIDs, the add-in will add the declaration.

IMPLEMENTATION

In this example you will create a toolbar for ArcMap that contains some existing ArcMap commands, in this case the previous three controls in the project. The code for this example is written in Visual Basic .NET.

Opening the toolbar project

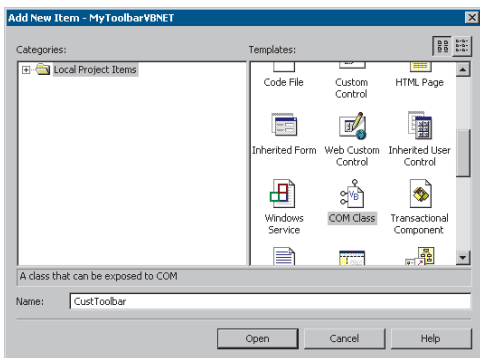
Open the previously created MyToolBarVBNET project.

1. Start Visual Studio .NET.
2. Click File, click Open, then click Project.
3. In the Open Project dialog box, click MyToolBarVBNET.sln from the location to which you saved it.

Creating a new COM class for the toolbar

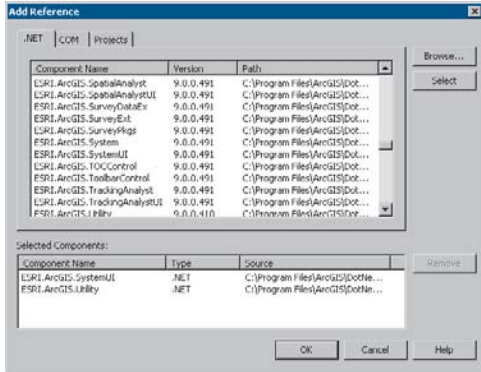
1. In the Solution Explorer, right-click MyToolBarVBNET project, click Add, then click Add New Item.
2. In the Add New Item dialog box, scroll down the right pane and click COM Class. In the Name text box, type "CustToolBar". Click Open to add the new class to the project.

The new COM class will have a new GUID and an empty subroutine called New.



Referencing ESRI libraries in your project

To program using ArcObjects, you will need to add references to the ESRI libraries.



1. In the Solution Explorer, right-click References and click Add References.
2. In the Add Reference dialog box, click the .NET tab and double-click the following assemblies:
 - ESRI.ArcGIS.SystemUI
 - ESRI.ArcGIS.Utility
3. Click OK to close the dialog box and add the libraries to your solution.

Adding Imports statements

1. In the Solution Explorer, double-click CustToolBar.vb to open its Code window.
2. At the top of the Code window, before the beginning of the class declaration, add the following lines of code:

```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.SystemUI
Imports ESRI.ArcGIS.Utility.CATIDs
```

```
Option Strict On
Imports System.Runtime.InteropServices
Imports ESRI.ArcGIS.SystemUI
Imports ESRI.ArcGIS.Utility.CATIDs
```

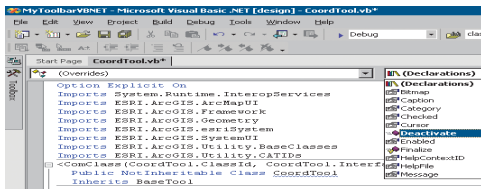
You will notice that the *Imports* statements work with IntelliSense.

Implementing the IToolBarDef interface

1. In the CustToolBar.vb Code window, specify that the *CustToolBar* class implements the *IToolBarDef* as shown below.

```
Public Class CustToolBar
    Implements IToolBarDef
```

2. In the Wizard bar at the top of the CustToolBar class Code window, click the left pulldown menu and click IToolBarDef.
3. In the right pulldown menu, click the first member of IToolBarDef, Caption. A code stub for the Caption property is added to the Code window.



The interface implementer will fully reference the implemented method types by default, as shown below, even though you have added *Imports* statements.

```
Public ReadOnly Property Caption() As String
    Implements ESRI.ArcGIS.SystemUI.IToolBarDef.Caption
```

If you want, you can remove the unnecessary namespace details to make your code more readable, as shown below, although you do not have to.

```
Public ReadOnly Property Caption() As String
    Implements IToolBarDef.Caption
```

- Repeat Steps 2 and 3 until you have added code stubs for all the members of the *IToolBarDef* interface: *Caption*, *GetItemInfo*, *ItemCount*, and *Name*.
- Press Ctrl + Shift + S to save all the files in your project.

Adding code to the members of IToolBarDef

The Toolbar items are defined within the *GetItemInfo* subroutine. Each item is a call to an existing command that is referenced by program ID and class name. For example, to reference a command in this project, you would use *MyToolBarVBNET.MyCommand*. In this example, you will add the three previous examples—a command, tool, and menu—as well as two existing ArcMap commands.

- In the *CustToolBar* class Code window, scroll down to find the *GetItemInfo* method, and add the code shown below.

```
Public Sub GetItemInfo(ByVal pos As Integer, ByVal itemDef As
ESRI.ArcGIS.SystemUI.IItemDef) Implements
ESRI.ArcGIS.SystemUI.IToolBarDef.GetItemInfo
    Select Case pos
        Case 0
            itemDef.ID = "MyToolBarVBNET.CustMenu"
            itemDef.Group = False
        Case 1
            itemDef.ID = "MyToolBarVBNET.ToggleButton"
            itemDef.Group = False
        Case 2
            itemDef.ID = "MyToolBarVBNET.CoordTool"
            itemDef.Group = False
        Case 3
            itemDef.ID = "esriArcMapUI.ZoomInTool"
            itemDef.Group = True
        Case 4
            itemDef.ID = "esriArcMapUI.ZoomOutTool"
            itemDef.Group = False
    End Select
End Sub
```

- Edit the code stubs as shown to complete the *Caption*, *ItemCount*, and *Name* methods as shown to return information about the toolbar.

```
Public ReadOnly Property Caption() As String Implements
IToolBarDef.Caption
    Get
        Return "CustToolBar-VBNET"
    End Get
End Property
```

```
Public ReadOnly Property ItemCount() As Integer Implements
ESRI.ArcGIS.SystemUI.IToolBarDef.ItemCount
```

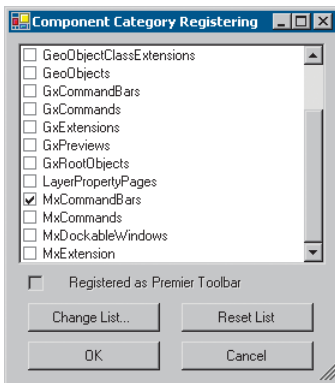
```

Get
    Return 5
End Get
End Property
    
```

Public ReadOnly Property Name() As String Implements
ESRI.ArcGIS.SystemUI.IToolBarDef.Name

```

Get
    Return "CustToolbar-VBNET"
End Get
End Property
    
```

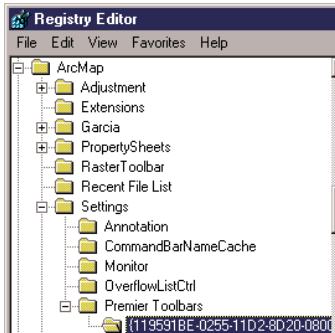


Adding COM category registration functions

The .NET Framework allows you to place COM registry functions within the class code. When the DLL is registered the class will be placed within the specified COM component category. ESRI provides an add-in to add this code.

1. Enable the add-in.
Click the Tools menu and click Add-in Manager.
Check the ESRI ComponentCategoryRegistrar box if it is not checked.
Click OK to close the dialog box.
2. Scroll the Code window to near the top of the class declaration, and put the cursor just below the COM GUIDs region.
3. Click the Tools menu and click ESRI ComponentCategoryRegistrar.
4. Check the MxCommandBar box and click OK to add the code. If you have not added a reference to the ESRI.ArcGIS.Utility assembly, the add-in will add the reference. If you have not declared using the namespace of System.Runtime.InteropServices or ESRI.ArcGIS.Utility.CATIDs, the add-in will add the declaration.

If you want the toolbar to be displayed the first time the desktop application is run after you have installed the toolbar, check the Registered as Premier Toolbar check box. This creates a new registry key name under HKEY_CURRENT_USER\Software\ESRI\ArcMap\Settings\PremierToolbars. The key name is the ClassID of the toolbar class and there is no value.



Compiling the project

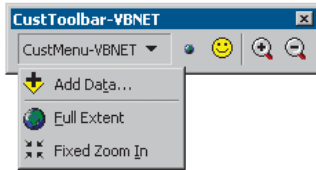
Now you are ready to build your project.

1. Click Build and click Build Solution.
2. Look at the Output window at the bottom of the Visual Studio .NET IDE.
If your project is compiled correctly, you should find a report stating Build succeeded.
3. If your build operation did not succeed, click the Task List window to see what errors are present, and correct the errors as indicated.
4. You can check the results of the Build operation by looking in the subdirectories of your project. By default, you will build a Debug version of your project, and the DLL that results from the Build operation will be stored in the \Bin directory. This directory should also contain debug information

(.pdb) and a type library (.tlb) file, produced by the Assembly Registration tool.

Examining the toolbar components in ArcMap

Before testing and using the toolbar and the controls it contains, it is worthwhile examining where the controls can be found in ArcMap.



1. Start ArcMap.
2. Open the Customize dialog box by clicking Tools > Customize.
3. The Toolbars tab displays the toolbars in ArcMap. You should see CustToolbar-VBNET, which you can turn on and off via the check box.
4. Click the Commands tab and open the Custom Controls category. You should see the command and tool you created. These can be dragged onto other toolbars as required.
5. Open the Menus category. You should see the CustMenu-VBNET menu in the list of available menus. Like the controls above, the menu can be dragged onto an existing toolbar as required.

Using the toolbar components in ArcMap

You can now test the toolbar and the controls it contains.

1. Start ArcMap.
2. Add some vector data.
3. Add the Custom toolbar by either going through the Customize dialog box as described above or by right-clicking an empty area of the ArcMap frame and clicking the CustToolbar-VBNET item.
4. Test the menu by clicking some items. These are standard ArcMap commands called via the menu.
5. Click a layer in the Table of Contents. The toggle layer visibility command (ToggleButton) should become enabled. Click the button to toggle the selected layer's visibility.
6. Click the Coord tool. A message box should be displayed, then be replaced by a cursor when you click OK on the dialog box. If you click a location on the map with the tool active, it should display the coordinates in the ArcMap status bar. Right-click to display a context menu for the tool. Double-click to enable the Select Graphics tool.
7. Test the other standard ArcMap tools on the toolbar.

You can remove all the components the DLL provides by simply unregistering it.

Deployment

The resulting DLL can easily be deployed to other machines using the methods described in Chapter 4, 'Licensing and deployment'.

PROJECT DESCRIPTION

This scenario is for ArcGIS Desktop developers who want to create an extension. The example used in this scenario is a simple extension that enables some tools on an existing toolbar.

The emphasis in this scenario is how you create the components that plug into the framework rather than any particular ArcObjects solution.

CONCEPTS

Extensions provide the developer with a powerful mechanism for extending the core functionality of the ArcGIS applications. An extension can provide a toolbar with new tools, listen for and respond to events, perform feature validation, and so on.

Extensions act as a central point of reference for developers when they are creating commands and tools for use within the applications. Often these commands and tools must share data or access common UI components. An extension is the logical place to store this data and develop the UI components. The main reason for this is that there is only ever one instance of an extension per running application and, given an *IApplication* interface, it is always possible to locate the extension and work with it.

Any extension that is registered with an application is automatically loaded and unloaded by the application; the end user does nothing to load or unload. For example, an extension that has been added to the ESRI Mx Extensions component category will be started when ArcMap is started and will be shut down when ArcMap is shut down.

To create your own extension, implement the *IExtension* interface. This interface allows you to set the name of the extension and specify what action takes place when the extension is started or shut down.

If you want your extension to be exposed in the Extensions dialog box, you should implement the *IExtensionConfig* interface. The Extensions dialog box allows users to turn extensions on and off. The *IExtensionConfig* interface provides the Extension dialog box with the name of the extension and a description of the extension; it also specifies the state of the extension.

The *IExtensionConfig* interface is independent of ESRI's licensing approach, so as a developer, you can incorporate a custom licensing solution. Alternatively, if your extension doesn't work with a license manager, you don't have to worry about requesting and releasing a license. You can implement *IExtensionConfig* to enable and disable the tools on your extension's toolbar accordingly.

For additional information on extensions and their use in extending the desktop applications, see 'Extending ArcObjects, Extending the User Interface', in the ArcGIS Developer Help system.

REQUIREMENTS

The requirements for working through this scenario are that you have ArcGIS Desktop installed and running.

The IDE used for this example is Visual Basic 6, and all IDE-specific steps will assume this is the IDE you are using.

It is also recommended that you read the Visual Basic language section within Appendix A, 'Developer environments'.

ADDITIONAL RESOURCES

The completed code for this scenario can be found in ArcGIS Developer Help under ArcGIS Desktop > Developer Guide > Developer Scenarios and on disk in the \DeveloperKit\samples\Developer_Guide_Scenarios\ArcGIS Desktop directory.

IMPLEMENTATION

In this example you will create an extension for ArcMap that enables controls on a toolbar. The code for the toolbar, controls, and extension all reside in a single project. The project for containing the toolbar has already been provided; this scenario will add the extension code.

The code for this example is written in Visual Basic 6.

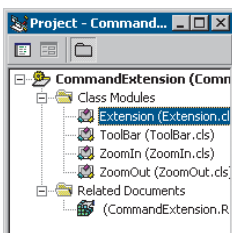
Opening the existing project

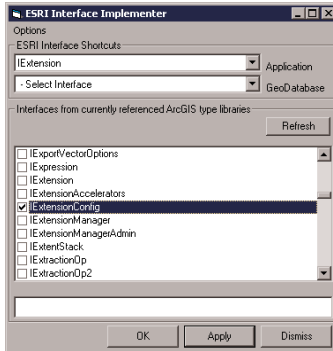
1. Start Visual Basic 6.
2. Open the CommandsExtension project in the \DeveloperKit\samples\Developer_Guide_Scenarios\ArcGIS_Desktop directory.
3. Take some time to examine the project and the classes that create the toolbar, zoom in, and zoom out tools. Compare this with the VB .NET code from the previous scenario.

Creating a new class for the extension

You need to create a new class for the extension code. All components that plug into the desktop framework are required to be classes. The UI controls that you see in ArcGIS Desktop are instances of these classes.

1. In the project window, right-click and click Add > Class Module. The Add Class Module dialog box appears.
2. On the New tab in the Add Class Module dialog box, click Class Module and click OK. A new empty class called class1 is created.
3. Display the project properties for this class by clicking it in the Project window and pressing F4.
4. Change the name of the class to Extension and make sure the instancing property is set to 5—Multiuse.
5. Save the extension class by right-clicking the class in the Project window and clicking Save.





Implementing extension interfaces

You now need to implement the *IExtension* and *IExtensionConfig* interfaces in the Extension class. The easiest way to do this is to use the ESRI Interface Implementer add-in. For help with add-ins, see the Add-Ins folder under the Contents tab within the ArcGIS Developer Help.

1. In VB6, click to display the empty Extension class Code window.
2. Click Add-Ins > ESRI Interface Implementer. The ESRI Interface Implementer dialog box is displayed.
3. In the dialog box, click Options and uncheck the Generate Error Handlers item.
4. Click IExtension from the Application pulldown menu.
5. Click IExtensionConfig from the lower window and click OK to close the dialog box.

The ESRI Interface Implmenter adds the *Implements* statement and stubs out all the members of those interfaces in the Code window. It will also add the appropriate library references to the project if they don't exist.

If you don't use the add-in, you will have to manually add the project references, type in the Implements statement, and manually stub out each interface member in the Code window.

Adding to the Extension class

Add the following code to the procedures in the Extension class.

Option Explicit

Implements IExtension

Implements IExtensionConfig

Private m_pApp As IApplication

Private m_ExtensionState As esriExtensionState

Private Property Get IExtension_Name() As String

' Internal name of the extension

IExtension_Name = "CommandExtension"

End Property

Private Sub IExtension_Startup(ByRef initializationData As Variant)

' Set the ArcMap application interface

If (TypeOf initializationData Is IMxApplication) Then

Set m_pApp = initializationData ' Is ArcMap

End If

End Sub

Private Sub IExtension_Shutdown()

' Release interface pointers

Set m_pApp = Nothing

End Sub

```

Private Property Get IExtensionConfig_ProductName() As String
    ' Name in Extension Manager Dialog
    If (m_pApp Is Nothing) Then Exit Property
    IExtensionConfig_ProductName = "ArcMap Command Extension"
End Property

Private Property Get IExtensionConfig_Description() As String
    ' Description in Extension Manager Dialog
    If (m_pApp Is Nothing) Then Exit Property
    IExtensionConfig_Description = "ArcMap Command Extension Version 1.0 " &
    vbCrLf & _
    "Copyright/Company/Date" & vbCrLf & vbCrLf & "Controls the enabled property
of zoom in and out commands."
End Property
Private Property Get IExtensionConfig_State() As esriExtensionState
    ' Get the extension state
    If (m_pApp Is Nothing) Then Exit Property
    IExtensionConfig_State = m_ExtensionState
End Property

Private Property Let IExtensionConfig_State(ByVal ExtensionState As
esriExtensionState)
    ' Set the extension state according to the check box in the
ExtensionManager Dialog
    If (m_pApp Is Nothing) Then Exit Property
    m_ExtensionState = ExtensionState
End Property

```

Enabling the ZoomIn tool with the extension

The `ZoomIn` class needs to be modified to listen to the extension configuration state. When the extension is turned on, the tool will be enabled.

1. In the `ZoomIn` class, declare a module-based variable to reference the extension state.

```
Private m_pCommandExtension As IExtensionConfig
```

2. Set the variable to `Extension` in the command's `OnCreate` method.

```

Private Sub ICommand_OnCreate(ByVal hook As Object)
    If (TypeOf hook Is IMxApplication) Then ' ArcMap
        Set m_pApp = hook
        Set m_pCommandExtension =
m_pApp.FindExtensionByName("CommandExtension")
    End If
End Sub

```

3. Set the `Enabled` property based on the extension state.

```

Private Property Get ICommand_Enabled() As Boolean
    ' Check box in Extension Manager dialog box controls command enabled
property
    If (Not m_pCommandExtension Is Nothing) Then
        If (m_pCommandExtension.State = esriESEnabled) Then
            ICommand_Enabled = True

```



```

Else
    ICommand_Enabled = False
End If
Else
    ICommand_Enabled = False
End If
End Property

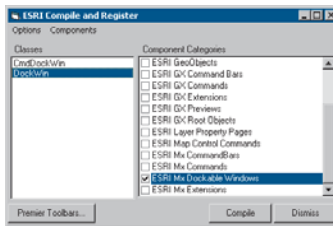
```

The same steps should be applied to the ZoomOut class to enable the ZoomOut tool based on the extension state.

Compiling the project

You are now ready to compile your project.

For help using the Component Category Manager, see Appendix A, 'Developer environments'.



1. In VB6, click File, then click Make CommandsExtensions.dll.

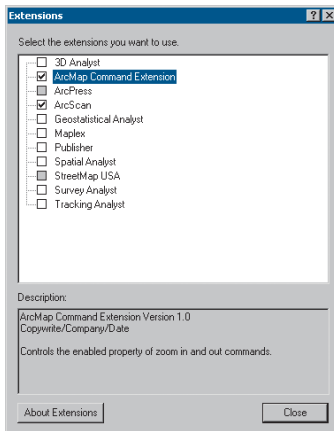
This creates a single DLL that you can deploy. The next step is to register the DLL with the operating system, then use the Component Category Manager to place the individual components within the DLL into the correct component categories, for example, the ZoomIn and ZoomOut classes are registered with the MxCommands component category, and so forth.

You can make this step semiautomatic, making deployment much easier for you and users, by utilizing the ESRI Compile and Register add-in.

1. In VB6, click Add-Ins > ESRI Compile and Register. The ESRI Compile and Register dialog box appears.
2. In the Classes pane, click Extension; in the Component Categories pane, click ESRI Mx Extensions.
3. In the Classes pane, click Toolbar; in the Component Categories pane, click ESRI Mx CommandBars.
4. In the Classes pane, click ZoomIn; in the Component Categories pane, click ESRI Mx Commands.
5. In the Classes pane, click ZoomOut; in the Component Categories pane, click ESRI Mx Commands.
6. Click Compile, click Read, and click OK on the dialog boxes that follow.

This add-in creates a Windows registry merge file, CommandsExtensions.reg, that can register the components in the correct component categories.

To register the components using the merge file, first register the DLL with the operating system if you have not done so already, then right-click the .reg file and click Merge.



Using the extension in ArcMap

You are now ready to use the extension in ArcMap.

1. Start ArcMap.
2. Add the ArcMap Command Extension toolbar either through the Customize dialog box or by right-clicking an empty area of the ArcMap frame and clicking the ArcMap Command Extension toolbar item. The toolbar will appear with the two tools. These will be disabled.
3. Open the Extensions dialog box via Tools > Extensions. The Extensions dialog box will be displayed.
4. Click the ArcMap Command Extension item and read the description in the lower pane.
5. Check the item and close the dialog box.

You should now see the two tools enabled.

Deployment

The resulting DLL can easily be deployed to other machines using the methods described in Chapter 4, 'Licensing and deployment'.

PROJECT DESCRIPTION

This scenario is for ArcGIS Desktop developers who want to create a dockable window as a component that can plug into the ArcGIS Desktop application framework.

For this scenario the dockable window will contain a *MapControl* to display an overview of the layers in the current document. This functionality is similar to the standard ArcMap Overview window, except the Overview window created in this scenario will be dockable.

CONCEPTS

Dockable windows

A dockable window is a window that can exist in a floating state or be attached to the main application window. The Table of Contents in ArcMap and the Tree View in ArcCatalog are examples of dockable windows.

Dockable windows are created by implementing the *IDockableWindowDef* interface within a class. You use the *ChildHWND* property on this interface to define what the window will consist of by passing in an *hWnd* of a control, such as a form or listbox.

The class you create is a definition for a dockable window; it is not actually a dockable window object. Once your class is registered in one of the dockable window component categories, the application uses the definition of the dockable window in your class to create the actual dockable window. This window is then treated as a modeless child window of the application.

You can access a particular dockable window through *IDockableWindowManager*, an interface implemented by the application. The *GetDockableWindow* method finds a dockable window using the UID of the dockable window. The following code illustrates this:

```
Dim pDocWinMgr As IDockableWindowManager
Set pDocWinMgr = Application 'QI
Dim u As New UID
u.Value = "OverviewDockWin.DockWin"
Dim pDockWin as IDockableWindow
Set pDockWin = pDocWinMgr.GetDockableWindow(u)
```

MapControl

The ESRI map control is a component that provides for the display of data similar to the data view in ArcMap. The *MapControl* encapsulates the *MapCoClass* and provides additional properties, methods, and events for managing the general appearance, display properties, and map properties of the control; adding and managing data layers within the control; loading map documents into the control; dropping data onto the control from other applications; and tracking shapes and drawing to the display.

The dockable window component categories include:
 ArcMap—ESRI Mx DockableWindows
 ArcCatalog—ESRI Gx DockableWindows

REQUIREMENTS

The requirements for working through this scenario are that you have ArcGIS Desktop installed and running.

The IDE used for this example is Visual Basic 6, and all IDE-specific steps will assume this is the IDE you are using.

It is also recommended that you read the Visual Basic language section within Appendix A, ‘Developer environments’.

ADDITIONAL RESOURCES

The completed code for this scenario can be found in ArcGIS Developer Help under ArcGIS Desktop > Developer Guide > Developer Scenarios and on disk in the \DeveloperKit\samples\Developer_Guide_Scenarios\ArcGIS Desktop directory.

IMPLEMENTATION

In this example you will create a dockable window for ArcMap. A dockable window can contain a variety of data, such as a form or listbox. In this example the window will contain a form with MapControl that behaves similar to the ArcMap Overview window.

The project with which you will work will contain a form with MapControl to display the overview data, a class for the dockable window, and a class for the command to display the dockable window.

The code for this example is written in Visual Basic 6.

Creating a Visual Basic 6 project

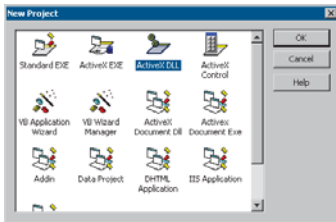
You will create and work with a Visual Basic 6 project to create an ActiveX DLL that will contain the components for the dockable window.

1. Start Visual Basic 6.
2. Click File > New Project. The New Project dialog box opens.
3. Click ActiveX DLL from the New Project dialog box and click OK.

This creates a project, Project1, with an empty class, Class1. For the moment delete Class1 by right-clicking the class in the project window and clicking Delete. Do not save Class1 when prompted.

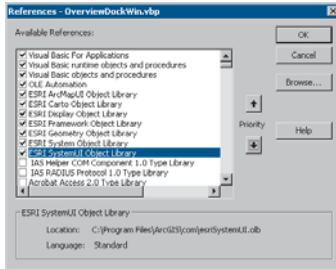
Change the name of the default project to OverviewDockWin.

1. Right-click the project name in the Project window and click Project1 Properties. The project properties dialog box is displayed.
2. In the Name text box, type “OverviewDockWin”. Click OK.



Referencing ESRI libraries in your project

To program using ArcObjects, you will need to add references to the ESRI libraries.

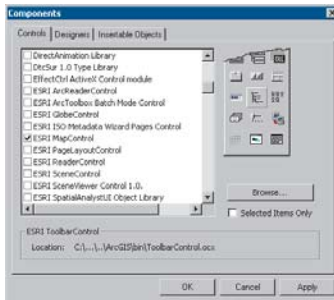


1. Click Project > References. The References dialog box appears.
2. In the References dialog box, click the following libraries:
 - ESRI ArcMapUI Object Library
 - ESRI Carto Object Library
 - ESRI Display Object Library
 - ESRI Framework Object Library
 - ESRI Geometry Object Library
 - ESRI System Object Library
 - ESRI SystemUI Object Library
3. Click OK.

Adding the MapControl component to the project

The ESRI MapControl provides a window similar to the data view in ArcMap. Although provided as a component with ArcGIS Desktop, it is mostly used in ArcGIS Engine applications as the primary display.

The ESRI MapControl component will be used to display the overview information on the form.



1. Click Project > Components. The Components dialog box is displayed.
2. In the Components dialog box, check the ESRI MapControl box and click OK.

ESRI MapControl should be added to the VB toolbox (View > Toolbox).

Save the current project. In the Save dialog box, create a directory for the project and save the project vbp file, for example, c:\overviewdockwin\overviewdockwin.vbp.

Creating the dockable window form

Create a form to contain the MapControl display.



1. In the project window, right-click and add a new form.
2. Open the properties for the form, press F4, and change the name to frmDockWin.
3. Change the border style to 0—None.
4. Change the height to 3270 and width to 3570.
5. Click ESRI MapControl from the VB toolbox and drag the control just inside the form. A white box labeled ESRI MapControl should appear in the form.

Adding code to the dockable window form

Open the code view for the form by right-clicking the form in the project win-

dow and click View Code. Enter the following code into the form:

```
Option Explicit
```

```
Private m_pArcMapAV As IActiveView 'Use this ActiveView to draw on
Dim m_pExtent As IEnvelope        'Extent rectangle
Dim m_pFillSym As IFillSymbol     'Rectangle draw symbol
```

```
Public Property Set ArcMapActiveView(ByVal pAV As IActiveView)
'Property for handling the ActiveView of ArcMap locally
If (Not pAV Is Nothing) Then
    Set m_pArcMapAV = pAV
Else
    Set m_pArcMapAV = Nothing
End If
End Property
```

```
Private Sub Form_Load()
'Turn off scrollbars.
MapControl1.ShowScrollbars = False

'Set up the symbol for the envelope.
Set m_pFillSym = New SimpleFillSymbol
'Transparent color for fill
Dim pColor As IColor
Set pColor = New RgbColor
pColor.Transparency = 0
m_pFillSym.Color = pColor

'Set up symbol for the outline.
Dim pLineSym As ILineStyle
Set pLineSym = New SimpleLineStyle
'Color for outline
Set pColor = New RgbColor
pColor.RGB = vbRed
pLineSym.Color = pColor
pLineSym.Width = 1.5

'Assign outline to fill symbol.
m_pFillSym.Outline = pLineSym
End Sub
```

It is a good practice to remove your module variables when terminating a class or unloading a form.

```
Private Sub Form_Unload(Cancel As Integer)
'Free memory.
Set m_pExtent = Nothing
Set m_pFillSym = Nothing
Set m_pArcMapAV = Nothing
End Sub
```

```
Private Sub MapControl1_OnAfterDraw(ByVal display As Variant, ByVal
viewDrawPhase As Long)
```

```

Dim phase As esriViewDrawPhase
phase = viewDrawPhase

'Draw a red rectangle indicating the current extent of the overview.
If phase = esriViewForeground Then
    If (Not m_pExtent Is Nothing And Not m_pFillSym Is Nothing And
m_pArcMapAV.FocusMap.LayerCount > 0) Then
        MapControl1.DrawShape m_pExtent, m_pFillSym
    End If
End If
End Sub

Private Sub MapControl1_OnMouseDown(ByVal button As Long, ByVal shift As
Long, ByVal X As Long, ByVal Y As Long, ByVal mapX As Double, ByVal mapY As
Double)
    If (m_pArcMapAV.FocusMap.LayerCount = 0) Then Exit Sub
    If (m_pExtent Is Nothing) Then Set m_pExtent = m_pArcMapAV.Extent

    'Track a rectangle representing the new extent of the ActiveView in
ArcMap.
    If (button = vbLeftButton) Then
        Set m_pExtent = MapControl1.TrackRectangle
        m_pArcMapAV.Extent = m_pExtent
    ElseIf (button = vbRightButton) Then
        'Zoom out to full extent.
        m_pArcMapAV.Extent = m_pArcMapAV.FullExtent
        Set m_pExtent = m_pArcMapAV.FullExtent
    End If

    'Redraw the new extent of the map in ArcMap and draw the red rectangle.
m_pArcMapAV.Refresh
MapControl1.Refresh esriViewForeground
End Sub

```

Save your project. Save the form inside the previous Save directory.

Add the dockable window class

A dockable window class defines the dockable window. This class implements *IDockableWindowDef*.

Add a new class to the project.

1. Right-click in the project window and click Add > Class Module. A new empty class is added to the project.
2. Open the properties for the new class, press F4, and rename the class as DockWin.

Add the following code to the DockWin class:

```
Option Explicit
```

```
'Implementation of dockable window
Implements IDockableWindowDef
```

```
Private Const EXPAND_FACTOR As Double = 1.2 ' Make the display a little
bigger to see the full extent.
```

```
Dim m_pApp As IApplication
Dim m_pMXDoc As IMxDocument
Dim WithEvents m_pDocEvent As MxDocument 'Listen for the MxDocument events.
Dim WithEvents m_pMapEvent As Map 'Listen for the Map events.
Dim WithEvents m_pLayoutEvent As PageLayout 'Listen for the PageLayout events.
Private m_pfrmDockWin As frmDockWin
```

```
Private Sub Class_Initialize()
    Set m_pfrmDockWin = New frmDockWin
End Sub
```

```
Private Sub Class_Terminate()
    Set m_pfrmDockWin = Nothing
End Sub
```

```
Private Property Get IDockableWindowDef_Caption() As String
    IDockableWindowDef_Caption = "Dockable Overview Window"
End Property
```

```
Private Property Get IDockableWindowDef_ChildHWND() As esriSystem.OLE_HANDLE
    'Pass back the hWnd of the picturebox that contains the MapControl.
    Load m_pfrmDockWin
    IDockableWindowDef_ChildHWND = m_pfrmDockWin.MapControl1.hwnd
End Property
```

The ChildHWND property defines the contents of the dockable window by passing in the hWnd of a control, such as a form or listbox.

```
Private Property Get IDockableWindowDef_Name() As String
    IDockableWindowDef_Name = "Overview Window"
End Property
```

```
Private Sub IDockableWindowDef_OnCreate(ByVal hook As Object)
    'The hook argument is a pointer to Application object.
    Set m_pApp = hook
    Set m_pMXDoc = m_pApp.Document

    'Set event handlers.
    Set m_pDocEvent = m_pApp.Document
    Set m_pMapEvent = m_pMXDoc.FocusMap

    'Let the mapcontrol know about the current ActiveView.
    Set m_pfrmDockWin.ArcMapActiveView = m_pMXDoc.FocusMap

    'Add overview layer to map control.
    AddOverviewLayer
End Sub
```

```
Private Sub IDockableWindowDef_OnDestroy()
```



```
'Free memory
Set m_pMapEvent = Nothing
Set m_pDocEvent = Nothing
Set m_pLayoutEvent = Nothing
Set m_pMXDoc = Nothing
Set m_pApp = Nothing
End Sub

Private Property Get IDockableWindowDef_UserData() As Variant
    'Not implemented
End Property

Private Function m_pDocEvent_BeforeCloseDocument() As Boolean
    Set m_pfrmDockWin.ArcMapActiveView = Nothing
End Function

Private Function m_pDocEvent_MapsChanged() As Boolean
    'Clear the mapcontrol and reset the reference to the map when a new
    dataframe is added.
    Set m_pMapEvent = m_pMXDoc.FocusMap

    'Let the mapcontrol know about the current ActiveView.
    Set m_pfrmDockWin.ArcMapActiveView = m_pMXDoc.FocusMap

    'Add overview layer to map control.
    AddOverviewLayer
End Function

Private Function m_pDocEvent_NewDocument() As Boolean
    'Clear the mapcontrol and reset the reference to the Map and PageLayout.
    Set m_pMapEvent = m_pMXDoc.FocusMap
    Set m_pLayoutEvent = m_pMXDoc.PageLayout

    m_pfrmDockWin.MapControl1.ClearLayers
    m_pfrmDockWin.MapControl1.Refresh

    'Let the mapcontrol know about the current ActiveView.
    Set m_pfrmDockWin.ArcMapActiveView = m_pMXDoc.FocusMap
End Function

Private Function m_pDocEvent_OpenDocument() As Boolean
    'Set the event handlers for map and page layout
    Set m_pMapEvent = m_pMXDoc.FocusMap
    Set m_pLayoutEvent = m_pMXDoc.PageLayout

    'Let the mapcontrol know about the current ActiveView.
    Set m_pfrmDockWin.ArcMapActiveView = m_pMXDoc.FocusMap
```

```

'Add overview layer to map control.
AddOverviewLayer
End Function

Private Sub m_pLayoutEvent_FocusMapChanged()
'Handle a different map.
Set m_pMapEvent = m_pMXDoc.FocusMap
Set m_pfrmDockWin.ArcMapActiveView = m_pMXDoc.FocusMap
End Sub

Private Sub m_pMapEvent_ItemAdded(ByVal Item As Variant)
'Add new overview layer to map control.
AddOverviewLayer
End Sub

Private Sub m_pMapEvent_ItemDeleted(ByVal Item As Variant)
'Add new overview layer to map control.
AddOverviewLayer
End Sub

Private Sub m_pMapEvent_ItemReordered(ByVal Item As Variant, ByVal toIndex
As Long)
'Add new overview layer to map control.
AddOverviewLayer
End Sub

Private Function AddOverviewLayer()
'Remove existing layer from map control.
m_pfrmDockWin.MapControl1.Map.ClearLayers
Set m_pfrmDockWin.MapControl1.SpatialReference =
m_pMXDoc.FocusMap.SpatialReference
m_pfrmDockWin.MapControl1.Refresh

'Get the best overview layer from the layers present in the focus map.
If (m_pMXDoc.FocusMap.LayerCount > 0) Then
Dim pLayer As ILayer
Set pLayer = GetBestOverviewLayer(m_pMXDoc.FocusMap)

'Add the layer to the map control and set the extent to see rectangle.
If (Not pLayer Is Nothing) Then
m_pfrmDockWin.MapControl1.AddLayer pLayer
Dim pEnv As IEnvelope
Set pEnv = pLayer.AreaOfInterest
pEnv.Expand EXPAND_FACTOR, EXPAND_FACTOR, True
m_pfrmDockWin.MapControl1.FullExtent = pEnv
m_pfrmDockWin.MapControl1.Refresh
End If
End If
End Function

```

The AddOverviewLayer function finds the layer with the largest extent in the map document and uses that in the Overview window.

```

Private Function GetBestOverviewLayer(pMap As IMap) As ILayer
'Get polygon layer with largest extent.
Dim pLayer As ILayer
Dim pFeatureLayer As IFeatureLayer
Dim pBestLayer As ILayer
Dim pEnumLayer As IEnumLayer
Dim pMaxEnv As IEnvelope
Dim pArea As IArea
Dim pBigArea As IArea

Set pMaxEnv = New Envelope
Set pEnumLayer = pMap.Layers
Set pLayer = pEnumLayer.Next
Set pBestLayer = pLayer
Set pBigArea = pLayer.AreaOfInterest
Set pArea = pLayer.AreaOfInterest

'Find the layer that is the greatest area.
While (Not pLayer Is Nothing)
  If (pArea.Area > pBigArea.Area) Then
    Set pBestLayer = pLayer
    Set pBigArea = pArea
  End If

  Set pLayer = pEnumLayer.Next
  If (Not pLayer Is Nothing) Then Set pArea = pLayer.AreaOfInterest
Wend

pBestLayer.Visible = True
Set GetBestOverviewLayer = pBestLayer
End Function

```

Save your project. Save the class inside the previous Save directory.

Adding the resource file

The resource file for the project contains a bitmap that will be used as the button icon for the dockable Overview Window command.

1. Right-click in the project window and click Add > Resource File.
2. In the Open a Resource File dialog box, browse to the `\DeveloperKit\samples\Developer_Guide_Scenarios\ArcGIS_Desktop\OverviewDocWin` directory and click `OverViewDockWin.res`.

You may browse the resource file and examine the bitmap.

Adding the overview dockable window command class

The final module to be added is the class that creates the command button to display the dockable window. This class implements *ICommand*.

Add a new class to the project.

1. Right-click in the project window and click Add > Class Module. A new empty class is added to the project.
2. Open the properties for the new class, press F4, and rename the class as CmdDockWin.

Add the following code to the CmdDockWin class:

Option Explicit

'Implementation of ICommand

Implements ICommand

Dim m_pApp As IApplication

Dim m_pDockWinMgr As IDockableWindowManager

Dim m_pDockWin As IDockableWindow

Dim m_pBitmap As IPictureDisp

Private Sub Class_Initialize()

Set m_pBitmap = LoadResPicture(101, 0)

End Sub

Private Sub Class_Terminate()

Set m_pBitmap = Nothing

End Sub

Private Property Get ICommand_Bitmap() As esriSystem.OLE_HANDLE

ICommand_Bitmap = m_pBitmap

End Property

Private Property Get ICommand_Caption() As String

ICommand_Caption = "Overview Dockable Window"

End Property

Private Property Get ICommand_Category() As String

ICommand_Category = "Developer Samples"

End Property

Private Property Get ICommand_Checked() As Boolean

'If the dockable window is visible, check the command

ICommand_Checked = m_pDockWin.IsVisible

End Property

Private Property Get ICommand_Enabled() As Boolean

ICommand_Enabled = True

End Property

Private Property Get ICommand_HelpContextID() As Long

'Not implemented

End Property

Private Property Get ICommand_HelpFile() As String

```

'Not implemented
End Property

Private Property Get ICommand_Message() As String
    ICommand_Message = "Display the dockable overview window"
End Property

Private Property Get ICommand_Name() As String
    ICommand_Name = "DeveloperSamples_OverviewDockableWindow"
End Property

Private Sub ICommand_OnClick()
    'Toggle the visibility of the dockable window.
    Dim pMxdoc As IMxDocument
    Set pMxdoc = m_pApp.Document
    Set frmDockWin.ArcMapActiveView = pMxdoc.FocusMap
    m_pDockWin.Show Not m_pDockWin.IsVisible
End Sub

Private Sub ICommand_OnCreate(ByVal hook As Object)
    Set m_pApp = hook
    Set m_pDockWinMgr = m_pApp 'QI for IDockableWindowManager

    'Get a reference to the dockable window with map control.
    Dim u As New UID
    u.Value = "OverviewDockWin.DockWin"
    Set m_pDockWin = m_pDockWinMgr.GetDockableWindow(u)
End Sub

Private Property Get ICommand_Tooltip() As String
    ICommand_Tooltip = "Overview Window"
End Property

```

The `GetDockableWindow` method finds the dockable window in the application.

Save your project. Save the class inside the previous Save directory.

Compiling the project

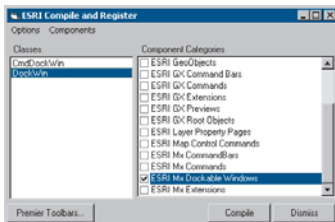
You are now ready to compile your project.

1. In VB6, click File, then click Make OverviewDockWin.dll.

This creates a single DLL that you can deploy. The next step is to register the DLL with the operating system, then use the Component Category Manager to place the individual components within the DLL into the correct component categories; for example, the DockWin class is registered with the Mx Dockable Windows component category and the CmdDockWin Class with Mx Commands.

You can make this step semiautomatic, making deployment much easier for you and users by using the ESRI Compile and Register add-in.

1. In VB6, click Add-Ins > ESRI Compile and Register. The ESRI Compile and Register dialog box appears.

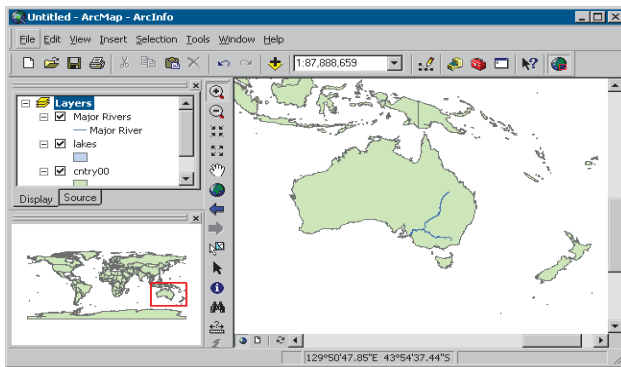


2. In the Classes pane click CmdDockWin; in the Component Categories pane, click ESRI Mx Commands.
3. In the Classes pane click DockWin; in the Component Categories pane, click ESRI Mx Dockable Windows.

Click Compile, click Read, and click OK in the dialog boxes that follow.

This add-in creates a Windows registry merge file, OverviewDockWin.reg, that can register the components in the correct component category.

To register the components using the merge file, first register the DLL with the operating system if you have not done so already, then right-click the .reg file and click Merge.



To move any dockable window without docking it, hold down the Ctrl key while dragging the window frame.

Using the dockable window in ArcMap

You are now ready to use the dockable window in ArcMap. You will first need to drag the Overview Dockable Window command to a toolbar.

1. Start ArcMap.
2. Add two or more vector layers in the map document.
3. Open the Customize dialog box (Tools > Customize).
4. In the 'Save in' dropdown menu, click the current project, unless you want to have a global customization.
5. Click the Commands tab and click the Developer Samples category. You should see the Overview Dockable Window command in the Commands pane.
6. Drag the Overview Dockable Window command to an existing toolbar.
7. Close the Customize dialog box.
8. Zoom into the display window.
9. Click the Overview Dockable Window button. The Overview dockable window should display.

The dockable window displays the layer with the largest extent. Spend some time experimenting with the window by docking it to the ArcMap frame. You can also drag a rectangle within the Overview window to change the extent in the ArcMap display.

Deployment

The resulting DLL can easily be deployed to other machines using the methods described in Chapter 4, 'Licensing and deployment'.



A

Developer environments

ArcObjects is based on Microsoft's Component Object Model. End users of ArcGIS applications don't necessarily have to understand COM, but if you're a developer intent on developing applications based on ArcObjects or extending the existing ArcMap and ArcCatalog applications using ArcObjects, an understanding of COM is a requirement. The level of understanding required depends on the depth of customization or development you wish to undertake.

Although this appendix does not cover the entire COM environment, it provides Visual Basic, Visual C++, and .NET developers with sufficient knowledge to be effective in using ArcObjects. There are many coding tips and guidelines that should make your work with ArcObjects more effective.

Before discussing COM specifically, it is worth considering the wider use of software components in general. There are a number of factors driving the motivation behind software components, but the principal one is the fact that software development is a costly and time-consuming venture.

In an ideal world, it should be possible to write a piece of code once and reuse it again and again using a variety of development tools, even in circumstances that the original developer did not foresee. Ideally, changes to the code's functionality made by the original developer could be deployed without requiring existing users to change or recompile their code.

Early attempts at producing reusable chunks of code revolved around the creation of class libraries, usually developed in C++. These early attempts suffered from several limitations, notably difficulty of sharing parts of the system (it is difficult to share binary C++ components—most attempts have only shared source code), problems of persistence and updating C++ components without recompiling, lack of good modeling languages and tools, and proprietary interfaces and customization tools.

To counteract these and other problems, many software engineers have adopted component-based approaches to system development. A software component is a binary unit of reusable code.

Several different but overlapping standards have emerged for developing and sharing components. For building interactive desktop applications, Microsoft's COM is the de facto standard. On the Internet, JavaBeans™ is viable technology. At a coarser grain appropriate for application-level interoperability, the Object Management Group (OMG) has specified the common object request broker architecture (CORBA).

To understand COM—and therefore all COM-based technologies—it's important to realize that it isn't an object-oriented language but a protocol or standard. COM is more than just a technology; it is a methodology of software development. COM defines a protocol that connects one software component, or module, with another. By making use of this protocol, it's possible to build reusable software components that can be dynamically interchanged in a distributed system.

COM also defines a programming model, known as interface-based programming. Objects encapsulate the manipulation methods and the data that characterize each instantiated object behind a well-defined interface. This promotes structured and safe system development since the client of an object is protected from knowing any of the details of how a particular method is implemented. COM doesn't specify how an application should be structured. As an application programmer working with COM, language, structure, and implementation details are left up to you.

COM does specify an object model and programming requirements that enable COM objects to interact with other COM objects. These objects can be within a single process, in other processes, or even on remote machines. They can be written in other languages and may have been developed in very different ways. That is why COM is referred to as a binary specification or standard—it is a standard that applies after a program has been translated to binary machine code.

ESRI chose COM as the component technology for ArcGIS because it is a mature technology that offers good performance, many of today's development tools support it, and there are a multitude of third-party components that can be used to extend the functionality of ArcObjects.

The key to the success of components is that they implement, in a very practical way, many of the object-oriented principles now commonly accepted in software engineering. Components facilitate software reuse because they are self-contained building blocks that can easily be assembled into larger systems.

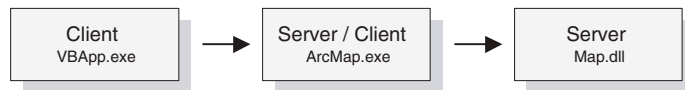
COM allows these objects to be reused at a binary level, meaning that third-party developers do not require access to source code, header files, or object libraries to extend the system even at the lowest level.

COMPONENTS, OBJECTS, CLIENTS, AND SERVERS

Different texts use the terms components, objects, clients, and servers to mean different things (to add to the confusion, various texts refer to the same thing using all of these terms). Therefore, it is worthwhile to define the terminology that this book will use.

COM is a client/server architecture. The server (or object) provides some functionality, and the client uses that functionality. COM facilitates the communication between the client and the object. An object can at the same time be a server to a client and be a client of some other object's services.

Objects are instances of COM classes that make services available for use by a client. Hence it is normal to talk of clients and objects instead of clients and servers. These objects are often referred to as COM objects and component objects. This book will refer to them simply as objects.



The client and its servers can exist in the same process or in a different process space. In-process servers are packaged in DLL form, and these DLLs are loaded into the client's address space when the client first accesses the server. Out-of-process servers are packaged in executables (EXE) and run in their own address space. COM makes the differences transparent to the client.

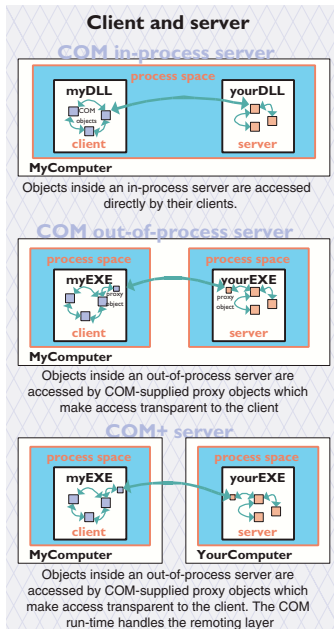
When creating COM objects, the developer must be aware of the type of server that the objects will sit inside, but if the creator of the object has implemented them correctly the packaging does not affect the use of the objects by the client.

There are pros and cons to each method of packaging that are symmetrically opposite. DLLs are faster to load into memory, and calling a DLL function is faster. EXEs, on the other hand, provide a more robust solution (if the server fails, the client will not crash), and security is better handled since the server has its own security context.

In a distributed system, EXEs are more flexible, and it does not matter if the server has a different byte ordering than the client. The majority of ArcObjects servers are packaged as in-process servers (DLLs). Later, you will see the performance benefits associated with in-process servers.

In a COM system, the client, or user of functionality, is completely isolated from the provider of that functionality, the object. All the client needs to know is that the functionality is available; with this knowledge, the client can make method calls to the object and expect the object to honor them. In this way, COM is said to act as a contract between client and object. If the object breaks that contract, the behavior of the system will be unspecified. In this way, COM development is based on trust between the implementer and the user of functionality.

In the ArcGIS applications there are many objects that provide, via their interfaces, thousands of properties and methods. When you use the ESRI object libraries you can assume that all these properties and interfaces have been fully implemented, and if they are present on the object diagrams, they are there to use.



CLASS FACTORY

Within each server there is an object called a class factory that the COM run time interacts with to instantiate objects of a particular class. For every corresponding COM class there is a class factory. Normally, when a client requests an object from a server, the appropriate class factory creates a new object and passes out that object to the client.

SINGLETON OBJECTS

While this is the normal implementation, it is not the only implementation possible. The class factory can also create an instance of the object the first time and, with subsequent calls, pass out the same object to clients. This type of implementation creates what is known as a singleton object since there is only one instance of the object per process.

GLOBALLY UNIQUE IDENTIFIERS

A distributed system potentially has many thousands of interfaces, classes, and servers, all of which must be referenced when locating and binding clients and objects together at run time. Clearly, using human-readable names would lead to the potential for clashes, hence COM uses GUIDs, 128-bit numbers that are virtually guaranteed to be unique in the world. It is possible to generate 10 million GUIDs per second until the year 5770 A.D, and each one would be unique.

The COM API defines a function that can be used to generate GUIDs; in addition, all COM-compliant development tools automatically assign GUIDs when appropriate. GUIDs are the same as Universally Unique Identifiers (UUIDs), defined by the Open Group's Distributed Computing Environment (DCE) specification. Below is a sample GUID in registry format.

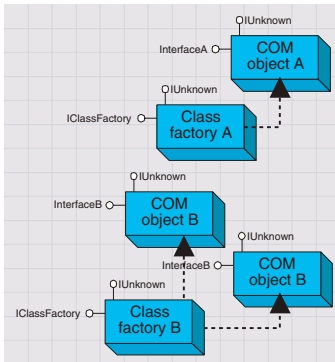
{E6BDA76-4D35-11D0-98BE-00805F7CED21}

COM CLASSES AND INTERFACES

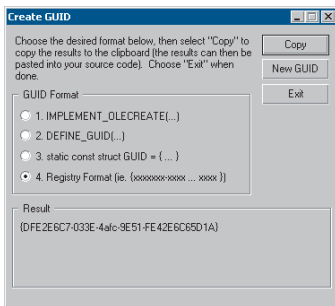
Developing with COM means developing using interfaces, the so-called interface-based programming model. All communication between objects is made via their interfaces. COM interfaces are abstract, meaning there is no implementation associated with an interface; the code associated with an interface comes from a class implementation. The interface sets out what requests can be made of an object that chooses to implement the interface.

How an interface is implemented differs between objects. Thus the objects inherit the type of interface, not its implementation, which is called type inheritance. Functionality is modeled abstractly with the interfaces and implemented within a class implementation. Classes and interfaces are often referred to as the "What" and "How" of COM. The interface defines what an object can do, and the class defines how it is done.

COM classes provide the code associated with one or more interfaces, thus encapsulating the functionality entirely within the class. Two classes can both have the same interface, but they may implement them quite differently. By implementing these interfaces in this way, COM displays classic object-oriented polymorphic behavior. COM does not support the concept of multiple inheritance; however,

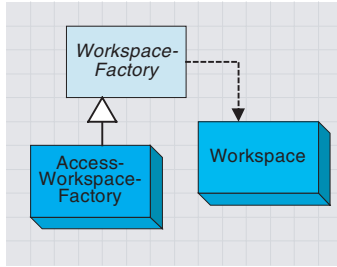


A server is a binary file that contains all the code required by one or more COM classes. This includes both the code that works with COM to instantiate objects into memory and the code to perform the methods supported by the objects contained within the server.



GUIDGEN.EXE is a utility that ships with Microsoft's Visual Studio and provides an easy-to-use user interface for generating GUIDs. It can be found in the directory <VS Install Dir>\Common\Tools.

The acronym GUID is commonly pronounced "gwid".



This is a simplified portion of the geodatabase object model showing type inheritance among abstract classes and coclasses and instantiation of classes.

this is not a shortcoming since individual classes can implement multiple interfaces. See the diagram to the left on polymorphic behavior.

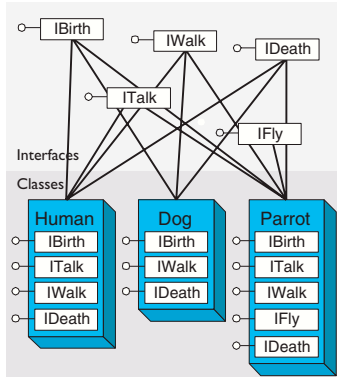
Within ArcObjects are three types of classes that the developer must be aware of: abstract classes, coclasses, and classes. An abstract class cannot be created; it is solely a specification for instances of subclasses (through type inheritance). ArcObjects Dataset or Geometry classes are examples of abstract classes. An object of type Geometry cannot be created, but an object of type Polyline can. This Polyline object in turn implements the interfaces defined within the Geometry base class, hence any interfaces defined within object-based classes are accessible from the coclass.

A coclass is a publicly creatable class. In other words, it is possible for COM to create an instance of that class and give the resultant object to the client in order for the client to use the services defined by the interfaces of that class. A class cannot be publicly created, but objects of this class can be created by other objects within ArcObjects and given to clients to use.

To the left is a diagram that illustrates the polymorphic behavior exhibited in COM classes when implementing interfaces. Notice that both the *Human* and *Parrot* classes implement the *ITalk* interface. The *ITalk* interface defines the methods and properties, such as *StartTalking*, *StopTalking*, or *Language*, but clearly the two classes implement these differently.

INSIDE INTERFACES

COM interfaces are how COM objects communicate with each other. When working with COM objects, the developer never works with the COM object directly but gains access to the object via one of its interfaces. COM interfaces are designed to be a grouping of logically related functions. The virtual functions are called by the client and implemented by the server; in this way an object's interfaces are the contract between the client and object. The client of an object is holding an interface pointer onto that object. This interface pointer is referred to as an opaque pointer since the client cannot gain any knowledge of the implementation details within an object or direct access to an object's state data. The client must communicate through the member functions of the interface. This allows COM to provide a binary standard through which all objects can effectively communicate.



This diagram shows how common behavior, expressed as interfaces, can be shared among multiple objects, animals in this example, to support polymorphism.

Interfaces allow developers to model functionality abstractly. Visual C++ developers see interfaces as collections of pure virtual functions, while Visual Basic developers see interfaces as collections of properties, functions, and sub routines.

The concept of the interface is fundamental in COM. The COM Specification (Microsoft, 1995) emphasizes these four points when discussing COM interfaces:

1. An interface is not a class. An interface cannot be instantiated by itself since it carries no implementation.
2. An interface is not an object. An interface is a related group of functions and is the binary standard through which clients and objects communicate.

3. Interfaces are strongly typed. Every interface has its own interface identifier, thereby eliminating the possibility of a collision between interfaces of the same human-readable name.
4. Interfaces are immutable. Interfaces are never versioned. Once defined and published, an interface cannot be changed.

An interface's permanence is not restricted to simply its method signatures, but extends to its semantic behavior as well. For example, an interface defines two methods, A and B, with no restrictions placed on their use. It breaks the COM contract if at a subsequent release Method A requires that Method B be executed first. A change like this would force possible recompilations of clients.

Once an interface has been published, it is not possible to change the external signature of that interface. It is possible at any time to change the implementation details of an object that exposes an interface. This change may be a minor bug fix or a complete reworking of the underlying algorithm; the clients of the interface do not care since the interface appears the same to them. This means that when upgrades to the servers are deployed in the form of new DLLs and EXEs, existing clients need not be recompiled to make use of the new functionality. If the external signature of the interface is no longer sufficient, a new interface is created to expose the new functions. Old or deprecated interfaces are not removed from a class to ensure all existing client applications can continue to communicate with the newly upgraded server. Newer clients will have the choice of using the old or new interfaces.

THE IUNKNOWN INTERFACE

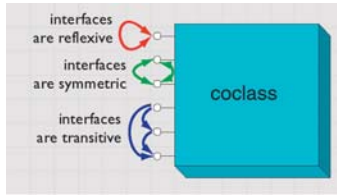
All COM interfaces derive from the *IUnknown* interface, and all COM objects must implement this interface. The *IUnknown* interface performs two tasks: it controls object lifetime and provides run-time type support. It is through the *IUnknown* interface that clients maintain a reference on an object while it is in use—leaving the actual lifetime management to the object itself.

Object lifetime is controlled with two methods, *AddRef* and *Release*, and an internal reference counter. Every object must have an implementation of *IUnknown* to control its own lifetime. Anytime an interface pointer is created or duplicated, the *AddRef* method is called, and when the client no longer requires this pointer, the corresponding *Release* method is called. When the reference count reaches zero, the object destroys itself.

Clients also use *IUnknown* to acquire other interfaces on an object. *QueryInterface* is the method that a client calls when another interface on the object is required. When a client calls *QueryInterface*, the object provides an interface and calls *AddRef*. In fact, it is the responsibility of any COM method that returns an interface to increment the reference count for the object on behalf of the caller. The client must call the *Release* method when the interface is no longer needed. The client calls *AddRef* explicitly only when an interface is duplicated.

When developing a COM object, the developer must obey the rules of *QueryInterface*. These rules dictate that interfaces for an object are symmetric, transitive, and reflexive and are always available for the lifetime of an object. For the client this means that, given a valid interface to an object, it is always valid to ask the object, via a call to *QueryInterface*, for any other interface on that object including itself. It is not possible to support an interface and later deny access to that interface, perhaps because of time or security constraints. Other mechanisms

The name IUnknown came from a 1988 internal Microsoft paper called Object Architecture: Dealing with the Unknown – or – Type Safety in a Dynamically Extensible Class Library.



The rules of *QueryInterface* dictate that interfaces of an object are reflexive, symmetric, and transitive. It is always possible, holding a valid interface pointer on an object, to get any other interface on that object.

The method *QueryInterface* is often referred to by the abbreviation *QI*.

Since *IUnknown* is fundamental to all COM objects, in general there are no references to *IUnknown* in any of the *ArcObjects* documentation and class diagrams.

Smart pointers are a class-based smart type and are covered in detail later in this appendix.

MIDL is commonly referred to simply as *IDL*.

The *IDL* defines the public interface that developers use when working with *ArcObjects*. When compiled, the *IDL* creates a type library.

must be used to provide this level of functionality. Some classes support the concept of optional interfaces. Depending on the coclass, they may optionally implement an interface; this does not break this rule since the interface is either always available or always not available on the class.

When requested for a particular interface, the *QueryInterface* method can return an already assigned piece of memory for that requested interface, or it can allocate a new piece of memory and return that. The only case when the same piece of memory must be returned is when the *IUnknown* interface is requested. When comparing two interface pointers to see if they point to the same object, it is important that a simple comparison not be performed. To correctly compare two interface pointers to see if they are for the same object, they both must be queried for their *IUnknown* interface, and the comparison must be performed on the *IUnknown* pointers. In this way, the *IUnknown* interface is said to define a COM object's identity.

It's good practice in Visual Basic to call *Release* explicitly by assigning an interface equal to *Nothing* to release any resources it's holding. Even if you don't call *Release*, Visual Basic will automatically call it when you no longer need the object—that is, when it goes out of scope. With global variables, you must explicitly call *Release*. In Visual Basic, the system performs all these reference-counting operations for you, making the use of COM objects relatively straightforward.

In C++, however, you must increment and decrement the reference count to allow an object to correctly control its own lifetime. Likewise, the *QueryInterface* method must be called when asking for another interface. In C++ the use of smart pointers simplifies much of this. These smart pointers are class-based and hence have appropriate constructors, destructors, and overloaded operators to automate much of the reference counting and query interface operations.

INTERFACE DEFINITION LANGUAGE

Microsoft Interface Definition Language (*MIDL*) is used to describe COM objects including their interfaces. This *MIDL* is an extension of the *IDL* defined by the Distributed Computing Environment (*DCE*), where it used to define remote procedure calls between clients and servers. The *MIDL* extensions include most of the Object Definition Language (*ODL*) statements and attributes. *ODL* was used in the early days of *OLE* automation for the creation of type libraries.

TYPE LIBRARY

A type library is best thought of as a binary version of an Interface Definition Language (*IDL*) file. It contains a binary description of all coclasses, interfaces, methods, and types contained within a server or servers.

There are several COM interfaces provided by Microsoft that work with type libraries. Two of these interfaces are *ITypeInfo* and *ITypeLib*. By utilizing these standard COM interfaces, various development tools and compilers can gain information about the coclasses and interfaces supported by a particular library.

To support the concept of a language-independent development set of components, all relevant data concerning the *ArcObjects* libraries is shipped inside type libraries. There are no header files, source files, or object files supplied or needed by external developers.

INBOUND AND OUTBOUND INTERFACES

Interfaces can be either inbound or outbound. An inbound interface is the most common kind—the client makes calls to functions within the interface contained on an object. An outbound interface is one where the object makes calls to the client—a technique analogous to the traditional callback mechanism.

There are differences in the way these interfaces are implemented. The implementer of an inbound interface must implement all functions of the interface; failure to do so breaks the contract of COM. This is also true for outbound interfaces. If you use Visual Basic, you don't have to implement all functions present on the interface since it provides stub methods for the methods you don't implement. On the other hand, if you use C++ you must implement all the pure virtual functions to compile the class.

Connection points is a specific methodology for working with outbound COM interfaces. The connection point architecture defines how the communication between objects is set up and taken down. Connection points are not the most efficient way of initializing bidirectional object communication, but they are in common use because many development tools and environments support them.

Dispatch event interfaces

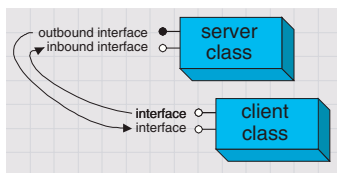
There are some objects with ArcObjects that support two outbound event interfaces that look similar to the methods they support. An example of two such interfaces are the *IDocumentEvents* and the *IDocumentEventsDisp*. The “Disp” suffix denotes a pure Dispatch interface. These dispatch interfaces are used by VBA when dealing with certain application events, such as loading documents. A VBA programmer works with the dispatch interfaces, while a developer using another development language uses the nonpure dispatch interface. Since these dispatch event interfaces are application specific, the details are discussed in the application chapters of the book, not the framework chapter.

Default interfaces

Every COM object has a default interface that is returned when the object is created if no other interface is specified. All the objects within the ESRI object libraries have *IUnknown* as their default interface, with a few exceptions.

The default interface of the *Application* object for both ArcCatalog and ArcMap is the *LApplication* interface. These uses of non-*IUnknown* default interfaces are a requirement of Visual Basic for Applications and are found on the ArcMap and ArcCatalog application-level objects.

This means that variables that hold interface pointers must be declared in a certain way. For more details, see the coding sections later in this appendix. When COM objects are created, any of the supported interfaces can be requested at creation time.



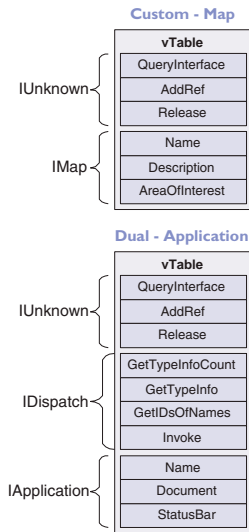
In the diagrams in this book and the ArcObjects object model diagrams, outbound interfaces are depicted with a solid circle on the interface jack.

The reason for making IUnknown the default interface is because the VB object browser hides information for the default interface. The fact that it hides IUnknown is not important for VB developers.

Binding is the term given to the process of matching the location of a function given a pointer to an object.

Binding type	In process DLL	Out of process DLL
Late binding	22,250	5,000
Custom vTable binding	825,000	20,000

This table shows the number of function calls that can be made per second on a typical Pentium® III machine.



These diagrams summarize the custom and IDispatch interfaces for two classes in ArcObjects. The layout of the vTable displays the differences. It also illustrates the importance of implementing all methods—if one method is missing, the vTable will have the wrong layout, and hence the wrong function pointer would be returned to the client, resulting in a system crash.

IDispatch interface

COM supports three types of binding:

1. Late. This is where type discovery is left until run time. Method calls made by the client but not implemented by the object will fail at execution time.
2. ID. Method IDs are stored at compile time, but execution of the method is still performed through a higher-level function.
3. Custom vTable (early). Binding is performed at compile time. The client can then make method calls directly into the object.

The *IDispatch* interface supports late- and ID-binding languages. The *IDispatch* interface has methods that allow clients to ask the object what methods it supports.

Assuming the required method is supported, the client executes the method by calling the *IDispatch::Invoke* method. This method, in turn, calls the required method and returns the status and any parameters back to the client on completion of the method call.

Clearly, this is not the most efficient way to make calls on a COM object. Late binding requires a call to the object to retrieve the list of method IDs; the client must then construct the call to the *Invoke* method and call it. The *Invoke* method must then unpack the method parameters and call the function.

All these steps add significant overhead to the time it takes to execute a method. In addition, every object must have an implementation for *IDispatch*, which makes all objects larger and adds to their development time.

ID binding offers a slight improvement over late binding in that the method IDs are cached at compile time, which means the initial call to retrieve the IDs is not required. However, there is still significant call overhead because the *IDispatch::Invoke* method is still called execute the required method on the object.

Early binding, often referred to as custom vTable binding, does not use the *IDispatch* interface. Instead, a type library provides the required information at compile time to allow the client to know the layout of the server object. At run time, the client makes method calls directly into the object. This is the fastest method of calling object methods and also has the benefit of compile-time type checking.

Objects that support both *IDispatch* and custom vTable are referred to as dual interface objects. The object classes within the ESRI object libraries do not implement the *IDispatch* interface; this means that these object libraries cannot be used with late-binding scripting languages, such as JavaScript™ or VBScript, since these languages require that all COM servers accessed support the *IDispatch* interface.

Careful examination of the ArcGIS class diagrams indicates that the *Application* objects support *IDispatch* because there is a requirement in VBA for the *IDispatch* interface.

Interfaces that directly inherit from an interface other than IUnknown cannot be implemented in VB.

All ActiveX controls support *IDispatch*. This means it is possible to use the various ActiveX controls shipped with ArcObjects to access functionality from within scripting environments.

INTERFACE INHERITANCE

An interface consists of a group of methods and properties. If one interface inherits from another, then all of the methods and properties in the parent are directly available in the inheriting object.

The underlying principle here is interface inheritance, rather than the implementation inheritance you may have seen in languages such as SmallTalk and C++. In implementation inheritance, an object inherits actual code from its parent; in interface inheritance, it's the definitions of the methods of the object that are passed on. The coclass that implements the interfaces must provide the implementation for all inherited interfaces.

Implementation inheritance is not supported in a heterogeneous development environment because of the need to access source and header files. For reuse of code, COM uses the principles of aggregation and containment. Both of these are binary-reuse techniques.

AGGREGATION AND CONTAINMENT

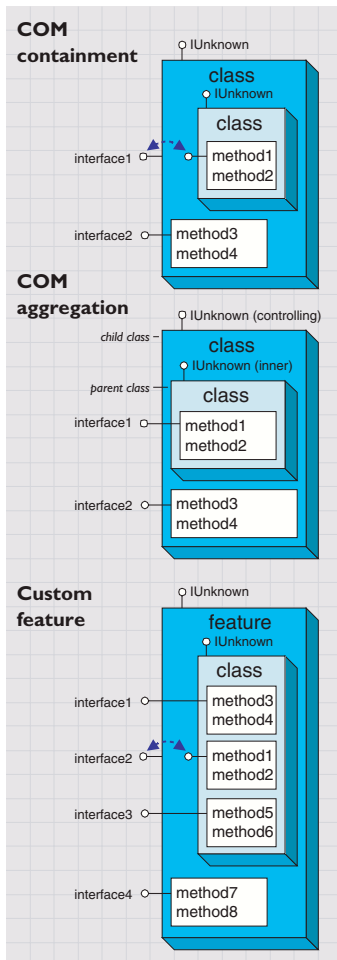
For a third-party developer to make use of existing objects, using either containment or aggregation, the only requirement is that the server housing the contained or aggregated object is installed on both the developer and target release machines. Not all development languages support aggregation.

The simplest form of binary reuse is containment. Containment allows modification of the original object's method behavior but not the method's signature. With containment, the contained object (inner) has no knowledge that it is contained within another object (outer). The outer object must implement all the interfaces supported by the inner. When requests are made on these interfaces, the outer object simply delegates them to the inner. To support new functionality, the outer object can either implement one of the interfaces without passing the calls on or implement an entirely new interface in addition to those interfaces from the inner object.

COM aggregation involves an outer object that controls which interfaces it chooses to expose from an inner object. Aggregation does not allow modification of the original object's method behavior. The inner object is aware that it is being aggregated into another object and forwards any *QueryInterface* calls to the outer (controlling) object so that the object as a whole obeys the laws of COM.

To the clients of an object using aggregation, there is no way to distinguish which interfaces the outer object implements and which interfaces the inner object implements.

Custom features make use of both containment and aggregation. The developer aggregates the interfaces where no customizations are required and contains those that are to be customized. The individual methods on the contained interfaces can then either be implemented in the customized class, thus providing custom functionality, or the method call can be passed to the appropriate method on the contained interface.



Aggregation is important in this case since there are some hidden interfaces defined on a feature that cannot be contained.

Visual Basic 6 does not support aggregation, so it can't be used to create custom features.

THREADS, APARTMENTS, AND MARSHALLING

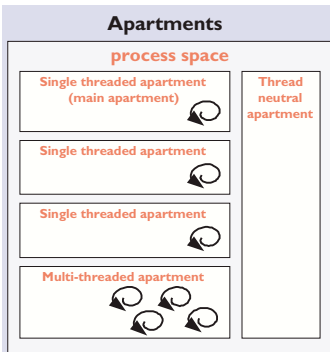
A thread is a process flow through an application. There are potentially many threads within Windows applications. An apartment is a group of threads that work with contexts within a process. With COM+, a context belongs to one apartment. There are potentially many types of context; security is an example of a type of context. Before successfully communicating with each other, objects must have compatible contexts.

Although an understanding of apartments and threading is not essential in the use of ArcObjects, basic knowledge will help you understand some of the implications with certain development environments highlighted later in this appendix.

COM supports two types of apartments: single-threaded apartment and multithreaded apartment (MTA). COM+ supports the additional thread-neutral apartment (TNA). A process can have any number of STAs; each process creates one STA called the main apartment. Threads that are created as apartments are placed in an STA. All user interface code is placed in an STA to prevent deadlock situations. A process can only have one MTA. A thread that is started as multi-threaded is placed in the MTA. The TNA has no threads permanently associated with it; rather, threads enter and leave the apartment when appropriate.

In-process objects have an entry in the registry, the ThreadingModel, that informs the COM Service Control Manager (SCM) into which apartment to place the object. If the object's requested apartment is compatible with the creator's apartment, the object is placed in that apartment; otherwise, the SCM will find or create the appropriate apartment. If no threading model is defined, the object will be placed in the main apartment of the process. The ThreadingModel registry entry can have the following values:

1. *Apartment.* Object must be executed within the STA. Normally used by UI objects.
2. *Free.* Object must be executed within the MTA. Objects creating threads are normally placed in the MTA.
3. *Both.* Object is compatible with all apartment types. The object will be created in the same apartment as the creator.
4. *Neutral.* Objects must execute in the TNA. Used by objects to ensure there is no thread switch when called from other apartments. This is only available under COM+.



Think of the SCM (pronounced scum) as the COM run-time environment. The SCM interacts with objects, servers, and the operating system and provides the transparency between clients and the objects that they work with.

Marshalling enables a client to make interface-function calls to objects in other apartments transparently. Marshalling can occur between COM apartments on different machines, between COM apartments in different process spaces, and between COM apartments in the same process space (STA to MTA, for example). COM provides a standard marshaller that handles function calls that use automation-compliant data types (see table below). Nonautomation data types can be handled by the standard marshaller as long as proxy stub code is generated; otherwise, custom marshalling code is required.

Type	Description
Boolean	Data item that can have the value True or False
unsigned char	8-bit unsigned data item
double	64-bit IEEE floating-point number
float	32-bit IEEE floating-point number
int	Signed integer, whose size is system dependent
long	32-bit signed integer
short	16-bit signed integer
BSTR	Length-prefixed string
CURRENCY	8-byte, fixed-point number
DATE	64-bit, floating-point fractional number of days since Dec 30, 1899
SCODE	For 16-bit systems - Built-in error that corresponds to VT_ERROR
Typedef enum myenum	Signed integer, whose size is system dependent
Interface IDispatch *	Pointer to the IDispatch interface
Interface IUnknown *	Pointer to an interface that does not derive from IDispatch
disinterface Typename *	Pointer to an interface derived from IDispatch
Coclass Typename *	Pointer to a coclass name (VT_UNKNOWN)
[oleautomation] interface Typename *	Pointer to an interface that derives from IDispatch
SAFEARRAY(Typename)	Typename is any of the above types. Array of these types
Typename*	Typename is any of the above types. Pointer to a type
Decimal	96-bit unsigned binary integer scaled by a variable power of 10. A decimal data type that provides a size and scale for a number (as in coordinates)

COMPONENT CATEGORY

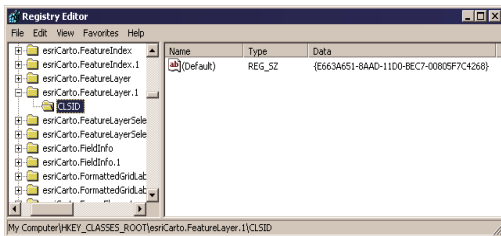
Component categories are used by client applications to find all COM classes of a particular type that are installed on the system efficiently. For example, a client application may support a data export function in which you can specify the output format—a component category could be used to find all the data export classes for the various formats. If component categories are not used, the application has to instantiate each object and interrogate it to see if it supports the required functionality, which is not a practical approach. Component categories support the extensibility of COM by allowing the developer of the client application to create and work with classes that belong to a particular category. If at a later date a new class is added to the category, the client application need not be changed to take advantage of the new class; it will automatically pick up the new class the next time the category is read.

COM AND THE REGISTRY

COM makes use of the Windows system registry to store information about the various parts that compose a COM system. The classes, interfaces, DLLs, EXEs, type libraries, and so forth, are all given unique identifiers (GUIDs) that the SCM

uses when referencing these components. To see an example of this, run regedit, then open HKEY_CLASSES_ROOT. This opens a list of all the classes registered on the system.

COM makes use of the registry for a number of housekeeping tasks, but the most important and most easily understood is the use of the registry when instantiating COM objects into



ESRI keys in the Windows system registry

memory. In the simplest case, that of an in-process server, the steps are as follows:

1. Client requests the services of a COM object.
2. SCM looks for the requested objects registry entry by searching on the class ID (a GUID).
3. DLL is located and loaded into memory. The SCM calls a function within the DLL called *DllGetClassObject*, passing the desired class as the first argument.
4. The class object normally implements the interface *IClassFactory*. The SCM calls the method *CreateInstance* on this interface to instantiate the appropriate object into memory.
5. Finally, the SCM asks the newly created object for the interface that the client requested and passes that interface back to the client. At this stage, the SCM drops out of the equation, and the client and object communicate directly.

From the above sequence of steps, it is easy to imagine how changes in the object's packaging (DLL versus EXE) make little difference to the client of the object. COM handles these differences.

AUTOMATION

Automation is the technology used by individual objects or entire applications to provide access to their encapsulated functionality via a late-bound language. Commonly, automation is thought of as writing macros, where these macros can access many applications for a task to be done. ArcObjects, as already stated, does not support the IDispatch interface; hence it cannot be used alone by an automation controller.

It is possible to instantiate an instance of ArcMap by cocreating the document object and making calls into ArcMap via the document object or one of its connected objects. There are, however, problems with this approach since the automation controller instance and the ArcMap instance are running in separate processes. Many of the objects contained within ArcObjects are process dependent, and therefore simple Automation will not work.

The function DllGetClassObject is the function that makes a DLL a COM DLL. Other functions, such as DllRegisterServer and DllUnregisterServer, are nice to have but not essential for a DLL to function as a COM DLL.

ArcGIS applications are built using ArcObjects and can be developed via several APIs. These include COM (VB, VC++, Delphi, MainWin), .NET (VB .NET and C#), Java, and C++. Some APIs are more suitable than others for developing certain applications. This is briefly discussed later, but you should also read the appropriate developer guide for the product you are working with for more information and recommendations on which API to use.

The subsequent sections of this appendix cover some general guidelines and considerations when developing with ArcObjects regardless of the API. Some of the more common API languages each then have a section describing the development environment, programming techniques, resources, and any other issues you must consider when developing with ArcObjects.

CODING STANDARDS

Each of the language-specific sections begins with a section on coding standards for that language. These standards are used internally at ESRI and are followed by the samples that ship with the software.

To understand why standards and guidelines are important, consider that in any large software development project, there are many backgrounds represented by the team members. Each programmer has personal opinions concerning how code should look and be built. If each programmer engineers code differently, it becomes increasingly difficult to share work and ideas. On a successful team, the developers adapt their coding styles to the tone set by the group. Often, this means adapting one's code to match the style of existing code in the system.

Initially, this may seem burdensome, but adopting a uniform programming style and set of techniques invariably increases software quality. When all the code in a project conforms to a standard set of styles and conventions, less time is wasted learning the particular syntactic quirks of individual programmers, and more time can be spent reviewing, debugging, and extending the code. Even at a social level, uniform style encourages team-oriented, rather than individualist, outlooks—leading to greater team unity, productivity and, ultimately, better software.

GENERAL CODING TIPS AND RESOURCES

This section on general coding tips will benefit all developers working with ArcObjects no matter what language they are using. Code examples are shown in VBA, however.

Class diagrams

Getting help with the object model is fundamental to successfully working with ArcObjects. Appendix B, 'Reading the object model diagrams', provides an introduction to the class diagrams and shows many of the common routes through objects. The class diagrams are most useful if viewed in the early learning process in printed form. This allows developers to appreciate the overall structure of the object model implemented by ArcObjects. When you are comfortable with the overall structure, the PDF files included with the software distribution can be more effective to work with. The PDF files are searchable; you can use the Search dialog box in Adobe® Acrobat® Reader® to find classes and interfaces quickly.

For simplicity, some samples will not follow the coding standards. As an example, it is recommended that when coding in Visual Basic, all types defined within an ESRI object library are prefixed with the library name, for example, `esriGeometry.IPolyline`. This is only done in samples in which a name clash will occur. Omitting this text makes the code easier to understand for developers new to ArcObjects.

Object browsers

In addition to the class diagram PDF files, the type library information can be viewed using a number of object browsers depending on your development platform.

Visual Basic and .NET have built-in object browsers; OLEView (a free utility from Microsoft) also displays type library information. The best object viewer to use in this environment is the ESRI object viewer. This object viewer can be used to view type information for any type library that you reference within it. Information on the classes and interfaces can be displayed in Visual Basic, Visual C++, or object diagram format. The object browsers can view coclasses and classes but cannot be used to view abstract classes. Abstract classes are only viewable on the object diagrams, where their use is solely to simplify the models.

Java and C++ developers should refer to the ArcObjects JavaDoc or ArcGIS Developer Help.

Component help

All interfaces and coclasses are documented in the component help file. Ultimately, this will be the help most commonly accessed when you get to know the object models better.

For Visual Basic and .NET developers this is a compiled HTML file that can be viewed by itself or when using an IDE. If the cursor is over an ESRI type when the F1 key is pressed, the appropriate page in the ArcObjects Class Help in the ArcGIS Developer Help system is displayed in the compiled HTML viewer.

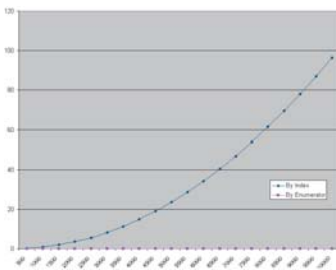
For Java and C++ developers, refer to ArcObjects JavaDoc or ArcGIS Developer Help.

Code wizards

There are a number of Code Generation Wizards available to help with the creation of boilerplate code, both in Visual Basic, Visual C++, and .NET. While these wizards are useful in removing the tediousness in common tasks, they do not excuse you as the developer from understanding the underlying principles of the generated code. The main objective should be to read the accompanying documentation and understand the limitations of these tools.

Indexing of collections

All collection-like objects in ArcObjects are zero-based for their indexing. This is not the case with all development environments; Visual Basic has both zero- and one-based collections. As a general rule, if the collection base is not known, assume that the collection base is zero. This ensures that a run-time error will be raised when the collection is first accessed (assuming the access of the collection does not start at zero). Assuming a base of one means the first element of a zero-based collection would be missed and an error would only be raised if the end of the collection were reached when the code is executed.



This graph shows the performance benefits of accessing a collection using an enumerator opposed to the elements index. As expected, the graph shows a classic power trend line ($y=cx^b$). The client (VB) and Server (VC++) code used to generate these metrics are included in the book samples.

Accessing collection elements

When accessing elements of a collection sequentially, it is best to use an enumerator interface. This provides the fastest method of walking through the collection. The reason for this is that each time an element is requested by index, internally an enumerator is used to locate the element. Hence, if the collection is looped over getting each element in turn, the time taken increases by power ($y=cx^b$).

Enumerator use

When requesting an enumerator interface from an object, the client has no idea how the object has implemented this interface. The object may create a new enumerator, or it may decide for efficiency to return a previously created enumerator. If a previous enumerator is passed to the client, the position of the element pointer will be at the last accessed element. To ensure that the enumerator is at the start of the collection, the client should reset the enumerator before use.

Error handling

All methods of interfaces, in other words, methods callable from other objects, should handle internal errors and signify success or failure via an appropriate HRESULT. COM does not support passing exceptions out of interface method calls. COM supports the notion of a COM exception. A COM exception utilizes the COM error object by populating it with relevant information and returning an appropriate HRESULT to signify failure. Clients, on receiving the HRESULT, can then interrogate the COM *Error* object for contextual information about the error. Languages such as Visual Basic implement their own form of exception handling. For more information, see the API language with which you are developing.

Exception handling is language specific and, since COM is language neutral, exceptions are not supported.

Notification interfaces

There are a number of interfaces in ArcObjects that have no methods. These are known as notification interfaces. Their purpose is to inform the application framework that the class that implements them supports a particular set of functionality. For instance, the Application Framework uses these interfaces to determine if a menu object is a root-level menu (*IRootLevelMenu*) or a context menu (*IShortcutMenu*).

Clientside storage

Some ArcObjects methods expect interface pointers to point to valid objects prior to making the method call. This is known as client storage since the client allocates the memory needed for the object before the method call. Suppose you have a polygon and you want to get its bounding box. To do this, use the *QueryEnvelope* method on *IPolygon*. If you write the following code:

```
Dim pEnv As IEnvelope
pPolygon.QueryEnvelope pEnv
```

you'll get an error because the *QueryEnvelope* method expects you (the client) to create the *Envelope*. The method will modify the envelope you pass in and return the changed one back to you. The correct code is shown below.

```
Dim pEnv As IEnvelope
Set pEnv = New Envelope
pPolygon.QueryEnvelope pEnv
```

How do you know when to create and when not to create? In general, all methods that begin with “Query”, such as *QueryEnvelope*, expect you to create the object. If the method name is *GetEnvelope*, then an object will be created for you. The reason for this clientside storage is performance. Where it is anticipated that the method on an object will be called in a tight loop, the parameters need only be created once and simply populated. This is faster than creating new objects inside the method each time.

Property by value and by reference

Occasionally, you will see a property that can be set by value or by reference, meaning that it has both a *put_XXX* and a *putref_XXX* method. On first appearance this may seem odd—why does a property need to support both? A Visual C++ developer sees this as simply giving the client the opportunity to pass ownership of a resource over to the server (using the *putref_XXX* method). A Visual Basic developer will see this as quite different; indeed, it is likely because of the Visual Basic developer that both *By Reference* and *By Value* are supported on the property.

To illustrate this, assume there are two text boxes on a form, Text1 and Text2. With a *propput*, it is possible to do the following in Visual Basic:

```
Text1.text = Text2.text
```

It is also possible to write this:

```
Text1.text = Text2
```

or this:

```
Text1 = Text2
```

All these cases make use of the *propput* method to assign the text string of text box Text2 to the text string of text box Text1. The second and third cases work because no specific property is stated, so Visual Basic looks for the property with a *DISPID* of 0.

This all makes sense assuming that it is the text string property of the text box that is manipulated. What happens if the actual object referenced by the variable Text2 is to be assigned to the variable Text1? If there was only a *propput* method it would not be possible; hence the need for a *propputref* method. With the *propputref* method, the following code will achieve the setting of the object reference.

```
Set Text1 = Text2
```

Notice the use of the “Set”.

Initializing Outbound interfaces

When initializing an Outbound interface, it is important to only initialize the variable if the variable does not already listen to events from the server object. Failure to follow this rule will result in an infinite loop.

DISPIDs are unique IDs given to properties and methods for the IDispatch interface to efficiently call the appropriate method using the *Invoke* method.

As an example, assume there is a variable *ViewEvents* that has been dimensioned as:

```
Private WithEvents ViewEvents As Map
```

To correctly sink this event handler, you can write code within the *OnClick* event of a UI button control, like this:

```
Private Sub UIButtonControl1_OnClick()  
    Dim pMxDoc As IMxDocument  
    Set pMxDoc = ThisDocument  
  
    ' Check to see that the map is different than what is currently connected  
    If (Not ViewEvents Is pMxDoc.FocusMap) Then  
        ' Sink the event since listener has not been initialized with this map  
        Set ViewEvents = pMxDoc.FocusMap  
    End If  
End Sub
```

Notice in the above code the use of the *Is* keyword to check for object identity.

DATABASE CONSIDERATIONS

When programming against the database, there are a number of rules that must be followed to ensure that the code will be optimal. These rules are detailed below.

If you are going to edit data programmatically, that is, not use the editing tools in ArcMap, you need to follow these rules to ensure that custom object behavior, such as network topology maintenance or triggering of custom-feature-defined methods, is correctly invoked in response to the changes your application makes to the database. You must also follow these rules to ensure that your changes are made within the multiuser editing (long transaction) framework.

Edit sessions

Make all changes to the geodatabase within an edit session, which is bracketed between *StartEditing* and *StopEditing* method calls on the *IWorkspaceEdit* interface found on the *Workspace* object.

This behavior is required for any multiuser update of the database. Starting an edit session gives the application a state of the database that is guaranteed not to change, except for changes made by the editing application.

In addition, starting an edit session turns on behavior in the geodatabase such that a query against the database is guaranteed to return a reference to an existing object in memory if the object was previously retrieved and is still in use.

This behavior is required for correct application behavior when navigating between a cluster of related objects while making modifications to objects. In other words, when you are not within an edit session, the database can create a new instance of a COM object each time the application requests a particular object from the database.

Edit operations

Group your changes into edit operations, which are bracketed between the *StartEditOperation* and *StopEditOperation* method calls on the *IWorkspaceEdit* interface.

You may make all your changes within a single edit operation if so required. Edit operations can be undone and redone. If you are working with data stored in ArcSDE, creating at least one edit operation is a requirement. There is no additional overhead to creating an edit operation.

Recycling and nonrecycling cursors

Use nonrecycling search cursors to select objects or fetch objects that are to be updated. Recycling cursors should only be used for read-only operations, such as drawing and querying features.

Nonrecycling cursors within an edit session create new objects only if the object to be returned does not already exist in memory.

Fetching properties using query filters

Always fetch all properties of the object; query filters should always use “*”. For efficient database access, the number of properties of an object retrieved from the database can be specified. As an example, drawing a feature requires only the *OID* and the *Shape* of the feature, hence the simpler renderers only retrieve these two columns from the database. This optimization speeds up drawing but is not suitable when editing features.

If all properties are not fetched, then object-specific code that is triggered may not find the properties that the method requires. For example, a custom feature developer might write code to update attributes A and B whenever the geometry of a feature changes. If only the geometry was retrieved, then attributes A and B would be found to be missing within the *OnChanged* method. This would cause the *OnChanged* method to return an error, which would cause the *Store* to return an error and the edit operation to fail.

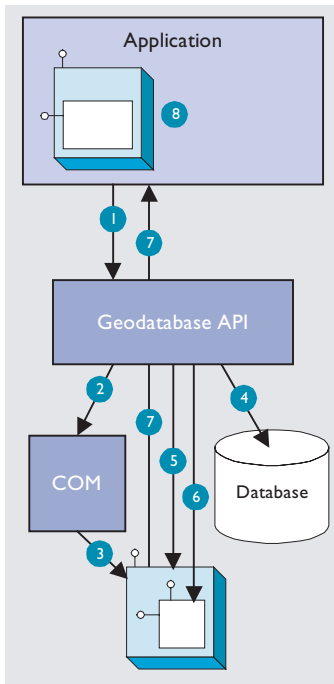
Marking changed objects

After changing an object, mark the object as changed (and guarantee that it is updated in the database) by calling *Store* on the object. Delete an object by calling the *Delete* method on the object. Set versions of these calls also exist and should be used if the operation is being performed on a set of objects to ensure optimal performance.

Calling these methods guarantees that all necessary polymorphic object behavior built into the geodatabase is executed (for example, updating of network topology or updating of specific columns in response to changes in other columns in ESRI-supplied objects). It also guarantees that developer-supplied behavior is correctly triggered.

Update and insert cursors

Never use update cursors or insert cursors to update or insert objects into object and feature classes in an already loaded geodatabase that has active behavior.



The diagram above clearly shows that the Feature, which is a COM object, has another COM object for its geometry. The Shape property of the feature simply passes the IGeometry interface pointer to this geometry object out to the caller that requested the shape. This means that if more than one client requested the shape, all clients point to the same geometry object. Hence, this geometry object must be treated as read-only. No changes should be performed on the geometry returned from this property, even if the changes are temporary. Anytime a change is to be made to a feature's shape, the change must be made on the geometry returned by the ShapeCopy property, and the updated geometry should subsequently be assigned to the Shape property.

Update and insert cursors are bulk cursor APIs for use during initial database loading. If used on an object or feature class with active behavior, they will bypass all object-specific behavior associated with object creation (such as topology creation) and with attribute or geometry updating (such as automatic recalculation of other dependent columns).

Shape and ShapeCopy geometry property

Make use of a Feature object's Shape and ShapeCopy properties to optimally retrieve the geometry of a feature. To better understand how these properties relate to a feature's geometry, refer to the diagram to the left to see how features coming from a data source are instantiated into memory for use within an application.

Features are instantiated from the data source using the following sequence:

1. The application requests a Feature object from a data source by calling the appropriate geodatabase API method calls.
2. The geodatabase makes a request to COM to create a vanilla COM object of the desired COM class (normally this class is *esriCore.Feature*).
3. COM creates the Feature COM object.
4. The geodatabase gets attribute and geometry data from a data source.
5. The vanilla Feature object is populated with appropriate attributes.
6. The Geometry COM object is created, and a reference is set in the Feature object.
7. The Feature object is passed to the application.
8. The Feature object exists in the application until it is no longer required.

USING A TYPE LIBRARY

Since objects from ArcObjects do not implement *IDispatch*, it is essential to make use of a type library for the compiler to early-bind to the correct data types. This applies to all development environments; although for both Visual Basic, Visual C++, and .NET, there are wizards that help you set this reference.

The type libraries required by ArcObjects are located within the ArcGIS install folder. For example the COM type libraries can be found in the COM folder while the .NET Interop assemblies are within the DotNet folder. Many different files can contain type library information, including EXEs, DLLs, OCXs, and OLBs.

COM DATA TYPES

COM objects talk via their interfaces, and hence all data types used must be supported by IDL. IDL supports a large number of data types; however, not all languages that support COM support these data types. Because of this, ArcObjects does not make use of all the data types available in IDL but limits the majority of interfaces to the data type supported by Visual Basic. The following table shows the data types supported by IDL and their corresponding types in a variety of languages.

Language	IDL	Microsoft C++	Visual Basic	Java
Base types	boolean	unsigned char	unsupported	char
	byte	unsigned char	unsupported	char
	small	char	unsupported	char
	short	short	Integer	short
	long	long	Long	int
	hyper	__int64	unsupported	long
	float	float	Single	float
	double	double	Double	double
	char	unsigned char	unsupported	char
	wchar_t	wchar_t	Integer	short
	enum	enum	Enum	int
Extended types	Interface Pointer	Interface Pointer	Interface Ref.	Interface Ref.
	VARIANT	VARIANT	Variant	ms.com.Variant
	BSTR	BSTR	String	java.lang.String
	VARIANT_BOOL	short (-1/0)	Boolean	[true/false]

Note the extended data types at the bottom of the table: *VARIANT*, *BSTR*, and *VARIANT_BOOL*. While it is possible to pass strings using data types such as *char* and *wchar_t*, these are not supported in languages such as Visual Basic. Visual Basic uses *BSTR*s as its text data type. A *BSTR* is a length-prefixed wide character array, in which the pointer to the array points to the text contained within it and not the length prefix. Visual C++ maps *VARIANT_BOOL* values onto 0 and -1 for the *False* and *True* values, respectively. This is different from the normal mapping of 0 and 1. Hence, when writing C++ code, be sure to use the correct macros—*VARIANT_FALSE* and *VARIANT_TRUE*—not *False* and *True*.

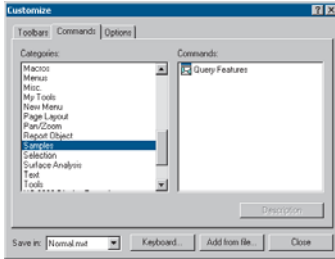
USING COMPONENT CATEGORIES

Component categories are used extensively in ArcObjects so developers can extend the system without requiring any changes to the ArcObjects code that will work with the new functionality.

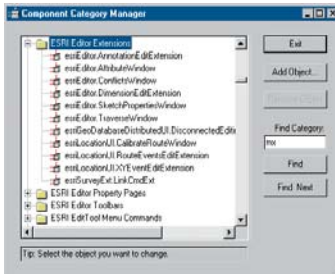
ArcObjects uses component categories in two ways. The first requires classes to be registered in the respective component category at all times, for example, ESRI Mx Extensions. Classes, if present in that component category, have an object that implements the *IExtension* interface and is instantiated when the ArcMap application is started. If the class is removed from the component category, the extension will not load, even if the map document (MXD file) is referencing that extension.

The second use is when the application framework uses the component category to locate classes and display them to a user to allow some user customization to occur. Unlike the first method, the application remembers (inside its map document) the objects being used and will subsequently load them from the map document. An example of this is the commands used within ArcMap. ArcMap reads the ESRI Mx Commands category when the Customization dialog box is displayed to the user. This is the only time the category is read. Once the user selects a command and adds it to a toolbar, the map document is used to determine what commands should be instantiated. Later, when this appendix covers debugging Visual Basic code, you'll see the importance of this.

Now that you've seen two uses of component categories, you will see how to get your classes registered into the correct component category. Development envi-



The Customize dialog box in ArcMap and ArcCatalog



The Component Category Manager

ronments have various levels of support for component categories; ESRI provides two ways of adding classes to a component category. The first can only be used for commands and command bars that are added to either ArcMap or ArcCatalog. Using the Add From File button on the Customize dialog box (shown to the left), it is possible to select a server. All classes in that server are then added to either the ESRI Gx Commands or the ESRI Mx Commands, depending on the application being customized. While this utility is useful, it is limited since it adds all the classes found in the server. It is not possible to remove classes, and it only supports two of the many component categories implemented within ArcObjects.

Distributed with ArcGIS applications is a utility application called the Component Category Manager, shown to the left. This small application allows you to add and remove classes from any of the component categories on your system, not just ArcObjects categories. Expanding a category displays a list of classes in the category. You can then use the Add Object button to display a checklist of all the classes found in the server. You check the required classes, and these checked classes are then added to the category.

Using these ESRI tools is not the only method to interact with component categories. During the installation of the server on the target user's machine, it is possible to add the relevant information to the Registry using a registry script. Below is one such script. The first line tells Windows for which version of regedit this script is intended. The last line, starting with "[HKEY_LOCAL_]", executes the registry command—all the other lines are comments in the file.

REGEDIT4

; This Registry Script enters coclasses into their appropriate Component Category

; Use this script during installation of the components

; Coclass: Exporter.ExportingExtension

; CLSID: {E233797D-020B-4AD4-935C-F659EB237065}

; Component Category: ESRI Mx Extensions

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{E233797D-020B-4AD4-935C-F659EB237065}\Implemented Categories\{B56A7C45-83D4-11D2-A2E9-080009B6F22B}]

The last line in the code above is one continuous line in the script.

The last method is for the self-registration code off the server to add the relevant classes within the server to the appropriate categories. Not all development environments allow this to be set up. Visual Basic has no support for component categories, although there is an add-in that allows this functionality. See the sections on Visual Basic developer add-ins and ATL later in this appendix.

The tables below summarize suggested naming standards for the various elements of your Visual Basic projects.

Module Type	Prefix
Form	frm
Class	cls
Standard	bas
Project	prj

Name your modules according to the overall function they provide; do not leave any with default names (such as "Form1", "Class1", or "Module1"). Additionally, prefix the names of forms, classes, and standard modules with three letters that denote the type of module, as shown in the table above.

Control Type	Prefix
Check box	chk
Combo box	cbo
Command button	cmd
Common dialog	cdl
Form	frm
Frame	fra
Graph	gph
Grid	grd
Image	img
Image list	iml
Label	lbl
List box	lst
List view	lvw
Map control	map
Masked edit	msk
Menu	mnu
OLE client	ole
Option button	opt
Picture box	pic
Progress bar	pbr
Rich text box	rft
Scroll bar	srl
Slider	sld
Status bar	sbr
Tab strip	tab
Text box	txt
Timer	tmr
Tool bar	tbr
Tree view	twv

As with modules, name your controls according to the function they provide; do not leave them with default names since this leads to decreased maintainability. Use the three-letter prefixes above to identify the type of the control.

This section is intended for both VB6 and VBA developers. Differences in the development environments are clearly marked throughout the text.

USER INTERFACE STANDARDS

Consider preloading forms to increase the responsiveness of your application. Be careful not to preload too many (preloading three or four forms is fine).

Use resource files (.res) instead of external files when working with bitmap files, icons, and related files.

Make use of constructors and destructors to set variable references that are only set when the class is loaded. These are the VB functions: *Class_Initialize()* and *Class_Terminate()*, or *Form_Load()* and *Form_Unload()*. Set all variables to *Nothing* when the object is destroyed.

Make sure the tab order is set correctly for the form. Do not add scroll bars to the tabbing sequence; it is too confusing.

Add access keys to those labels that identify controls of special importance on the form (use the *TabIndex* property).

Use system colors where possible instead of hard-coded colors.

Variable declaration

- Always use *Option Explicit* (or turn on Require Variable Declaration in the VB Options dialog box). This forces all variables to be declared before use and thereby prevents careless mistakes.
- Use *Public* and *Private* to declare variables at module scope and *Dim* in local scope. (*Dim* and *Private* mean the same at *Module* scope; however, using *Private* is more informative.) Do not use *Global* anymore; it is available only for backward compatibility with VB 3.0 and earlier.
- Always provide an explicit type for variables, arguments, and functions. Otherwise, they default to *Variant*, which is less efficient.
- Only declare one variable per line unless the type is specified for each variable.

This line causes *count* to be declared as a *Variant*, which is likely to be unintended.

```
Dim count, max As Long
```

This line declares both *count* and *max* as *Long*, the intended type.

```
Dim count As Long, max As Long
```

These lines also declare *count* and *max* as *Long* and are more readable.

```
Dim count As Long
```

```
Dim max As Long
```

Parentheses

Use parentheses to make operator precedence and logic comparison statements easier to read.

```
Result = ((x * 24) / (y / 12)) + 42
```

```
If ((Not pFoo Is Nothing) And (Counter > 200)) Then
```

Use the following notation for naming variables and constants:

[<libraryName.>][<scope.>]<type><name>

<name> describes how the variable is used or what it contains. The <scope> and <type> portions should always be lowercase, and the <name> should use mixed case.

Library Name	Library
esriGeometry	ESRI Object Library
stdole	Standard OLE COM Library
<empty>	Simple variable data type

<libraryName>

Prefix	Variable scope
c	constant within a form or class
g	public variable defined in a class form or standard module
m	private variable defined in a class or form
<empty>	local variable

<scope>

Prefix	Data Type
b	Boolean
by	byte or unsigned char
d	double
fn	function
h	handle
i	int (integer)
l	long
p	a pointer
s	string

<type>

Order of conditional determination

Visual Basic, unlike languages such as C and C++, performs conditional tests on all parts of the condition, even if the first part of the condition is *False*. This means you must not perform conditional tests on objects and interfaces that had their validity tested in an earlier part of the conditional statement.

```
' The following line will raise a run-time error if pFoo is NULL.
If ((Not pFoo Is Nothing) And (TypeOf pFoo.Thing Is IBar)) then
End If
```

```
' The correct way to test this code is
If (Not pFoo Is Nothing) Then
  If (TypeOf pFoo.Thing Is IBar) Then
    ' Perform action on IBar thing of Foo.
  End If
End If
```

Indentation

Use two spaces or a tab width of two for indentation. Since there is only one editor for VB code, formatting is not as critical an issue as it is for C++ code.

Default properties

Avoid using default properties except for the most common cases. They lead to decreased legibility.

Intermodule referencing

When accessing intermodule data or functions, always qualify the reference with the module name. This makes the code more readable and results in more efficient run-time binding.

Multiple property operations

When performing multiple operations against different properties of the same object, use a *With ... End With* statement. It is more efficient than specifying the object each time.

```
With frmHello
  .Caption = "Hello world"
  .Font = "Playbill"
  .Left = (Screen.Width - .Width) / 2
  .Top = (Screen.Height - .Height) / 2
End With
```

Arrays

For arrays, never change *Option Base* to anything other than zero (which is the default). Use *LBound* and *UBound* to iterate over all items in an array.

```
myArray = GetSomeArray
For i = LBound(myArray) To UBound(myArray)
  MsgBox cstr(myArray(i))
Next I
```

Bitwise operators

Since *And*, *Or*, and *Not* are bitwise operators, ensure that all conditions using them test only for Boolean values (unless, of course, bitwise semantics are what is intended).

```
If (Not pFoo Is Nothing) Then
    ' Valid Foo do something with it
End If
```

Type suffixes

Refrain from using type suffixes on variables or function names (such as *myString\$* or *Right\$(myString)*), unless they are needed to distinguish 16-bit from 32-bit numbers.

Ambiguous type matching

For ambiguous type matching, use explicit conversion operators (such as *C.Sng*, *C.Dbl*, and *C.Str*), instead of relying on VB to pick which one will be used.

Simple image display

Use an *ImageControl* rather than a *PictureBox* for simple image display. It is much more efficient.

Error handling

Always use *On Error* to ensure fault-tolerant code. For each function that does error checking, use *On Error* to jump to a single error handler for the routine that deals with all exceptional conditions that are likely to be encountered. After the error handler processes the error—usually by displaying a message—it should proceed by issuing one of the recovery statements shown on the table to the left.

Error handling in Visual Basic is not the same as general error handling in COM (see the section ‘Working with HRESULTs’).

Recovery Statement	Frequency	Meaning
Exit Sub	usually	Function failed, pass control back to caller
Raise	often	Raise a new error code in the caller's scope
Resume	rarely	Error condition removed, reattempt offending statement
Resume Next	very rarely	Ignore error and continue with next statement

Event functions

Refrain from placing more than a few lines of code in event functions to prevent highly fractured and unorganized code. Event functions should simply dispatch to reusable functions elsewhere.

Memory management

To ensure efficient use of memory resources, the following points should be considered:

- Unload forms regularly. Do not keep many forms loaded but invisible since this consumes system resources.
- Be aware that referencing a form-scoped variable causes the form to be loaded.
- Set unused objects to *Nothing* to free up their memory.
- Make use of *Class_Initialize()* and *Class_Terminate()* to allocate and destroy resources.

While Wend constructs

Avoid *While ... Wend* constructs. Use the *Do While ... Loop* or *Do Until ... Loop* instead because you can conditionally branch out of this construct.

```
pFos.Reset
Set pFoo = pFos.Next
Do While (Not pFoo Is Nothing)
  If (pFoo.Answer = "Done") Then Exit Loop
  Set pFoo = pFos.Next
Loop
```

The Visual Basic Virtual Machine

The Visual Basic Virtual Machine (VBVM) contains the intrinsic Visual Basic controls and services, such as starting and ending a Visual Basic application, required to successfully execute all Visual Basic developed code.

The VBVM is packaged as a DLL that must be installed on any machine wanting to execute code written with Visual Basic, even if the code has been compiled to native code. If the dependencies of any Visual Basic compiled file are viewed, the file *msvbvm60.dll* is listed; this is the DLL housing the Virtual Machine.

For more information on the services provided by the VBVM, see the sections 'Interacting with the IUnknown interface' and 'Working with HRESULTs' in this appendix.

Interacting with the IUnknown interface

The section on COM contains a lengthy section on the *IUnknown* interface and how it forms the basis on which all of COM is built. Visual Basic hides this interface from developers and performs the required interactions (*QueryInterface*, *AddRef*, and *Release* function calls) on the developer's behalf. It achieves this because of functionality contained within the VBVM. This simplifies development with COM for many developers, but to work successfully with *ArcObjects*, you must understand what the VBVM is doing.

Visual Basic developers are used to dimensioning variables as follows:

```
Dim pColn as New Collection 'Create a new collection object.
PColn.Add "Foo", "Bar"     'Add element to collection.
```

It is worth considering what is happening at this point. From a quick inspection of the code it looks like the first line creates a collection object and gives the developer a handle on that object in the form of *pColn*. The developer then calls a method on the object *Add*. Earlier in the appendix you learned that objects talk via their interfaces, never through a direct handle on the object itself. Remember, objects expose their services via their interfaces. If this is true, something isn't adding up.

What is actually happening is some "VB magic" performed by the VBVM and some trickery by the Visual Basic Editor in the way that it presents objects and interfaces. The first line of code instantiates an instance of the collection class, then assigns the default interface for that object, *_Collection*, to the variable *pColn*. It is this interface, *_Collection*, that has the methods defined on it. Visual Basic has hidden the fact of interface-based programming to simplify the developer experi-

The VBVM was called the VB Runtime in earlier versions of the software.

ence. This is not an issue if all the functionality implemented by the object can be accessed via one interface, but it is an issue when there are multiple interfaces on an object that provides services.

The Visual Basic Editor backs this up by hiding default interfaces from the IntelliSense completion list and the object browser. By default, any interfaces that begin with an underscore, “_”, are not displayed in the object browser (to display these interfaces turn Show Hidden Member on, although this will still not display default interfaces).

You have already learned that the majority of ArcObjects have *IUnknown* as their default interface and that Visual Basic does not expose any of *IUnknown*'s methods, namely, *QueryInterface*, *AddRef*, and *Release*. Assume you have a class *Foo* that supports three interfaces, *IUnknown* (the default interface), *IFoo*, and *IBar*. This means that if you were to dimension the variable *pFoo* as below, the variable *pFoo* would point to the *IUnknown* interfaces.

```
Dim pFoo As New Foo ' Create a new Foo object
pFoo.??????
```

Since Visual Basic does not allow direct access to the methods of *IUnknown*, you would immediately have to *QI* for an interface with methods on it that you can call. Because of this, the correct way to dimension a variable that will hold pointers to interfaces is as follows:

```
Dim pFoo As IFoo ' Variable will hold pointer to IFoo interface.
Set pFoo = New Foo ' Create Instance of Foo object and QI for IFoo.
```

Now that you have a pointer to one of the object's interfaces, it is an easy matter to request from the object any of its other interfaces.

```
Dim pBar as IBar 'Dim variable to hold pointer to interface
Set pBar = pFoo 'QI for IBar interface
```

By convention, most classes have an interface with the same name as the class with an “I” prefix; this tends to be the interface most commonly used when working with the object. You are not restricted to which interface you request when instantiating an object; any supported interface can be requested, hence the code below is valid.

```
Dim pBar as IBar
Set pBar = New Foo 'CoCreate Object
Set pFoo = pBar 'QI for interface
```

Objects control their own lifetime, which requires clients to call *AddRef* anytime an interface pointer is duplicated by assigning it to another variable and to call *Release* anytime the interface pointer is no longer required. Ensuring that there are a matching number of *AddRefs* and *Releases* is important and, fortunately, Visual Basic performs these calls automatically. This ensures that objects do not “leak”. Even when interface pointers are reused, Visual Basic will correctly call release on the old interface before assigning the new interface to the variable. The following code illustrates these concepts; note the reference count on the object at the various stages of code execution.

See the Visual Basic Magic sample on the disk for this code. You are encouraged to run the sample and use the code. This object also uses an ATL C++ project to define the SimpleObject and its interfaces; you are encouraged to look at this code to learn a simple implementation of a C++ ATL object.

```
Private Sub VBMagic()
    ' Dim a variable to the IUnknown interface on the simple object.
    Dim pUnk As IUnknown

    ' Co Create simpleobject asking for the IUnknown interface.
    Set pUnk = New SimpleObject 'refCount = 1

    ' QI for a useful interface.
    ' Define the interface.
    Dim pMagic As ISimpleObject

    ' Perform the QI operation
    Set pMagic = punk 'refCount = 2

    ' Dim another variable to hold another interface on the object.
    Dim pMagic2 As IAnotherInterface

    ' QI for that interface
    Set pMagic2 = pMagic 'refCount = 3

    ' Release the interface pointer.
    Set pMagic2 = Nothing 'refCount = 2

    ' Release the interface.
    Set pMagic = Nothing 'refCount = 1

    ' Now reuse the pUnk variable - what will VB do for this?
    Set pUnk = New SimpleObject 'refCount = 1, then 0, then 1

    ' Let the interface variable go out of scope and let VB tidy up.
End Sub 'refCount = 0
```

Often interfaces have properties that are actually pointers to other interfaces. Visual Basic allows you to access these properties in a shorthand fashion by chaining interfaces together. For instance, assume that you have a pointer to the *IFoo* interface, and that interface has a property called *Gak* that is an *IGak* interface with the method *DoSomething()*. You have a choice on how to access the *DoSomething* method. The first method is the long-handed way.

```
Dim pGak as IGak
Set pGak = pFoo 'Assign IGak interface to local variable.
pGak.DoSomething 'Call method on IGak interface.
```

Alternatively, you can chain the interfaces and accomplish the same thing on one line of code.

```
pFoo.Gak.DoSomething 'Call method on IGak interface.
```

When looking at the sample code, you will see both methods. Normally the former method is used on the simpler samples, as it explicitly tells you what interfaces are being worked with. More complex samples use the shorthand method.

This technique of chaining interfaces together can always be used to get the value of a property, but it cannot always be used to set the value of a property. Interface chaining can only be used to set a property if all the interfaces in the chain are set by reference. For instance, the code below would execute successfully.

```
Dim pMxDoc As ImxDocument
Set pMxDoc = ThisDocument
pMxDoc.FocusMap.Layers(0).Name = "Foo"
```

The above example works because both the *Layer* of the *Map* and the *Map* of the document are returned by reference. The lines of code below would not work since the *Extent* envelope is set by value on the active view.

```
pMxDoc.ActiveView.Extent.Width = 32
```

The reason that this does not work is that the VBVM expands the interface chain to get the end property. Because an interface in the chain is dealt with by value, the VBVM has its own copy of the variable, not the one chained. To set the *Width* property of the extent envelope in the above example, the VBVM must write code similar to this:

```
Dim pActiveView as IActiveView
Set pActiveView = pMxDoc.ActiveView
```

```
Dim pEnv as IEnvelope
Set pEnv = pActiveView.Extent ' This is a get by value,
```

```
PEnv.Width = 32 ' The VBVM has set its copy of the Extent and not
' the copy inside the ActiveView
```

For this to work the VBVM requires the extra line below.

```
pActiveView.Extent = pEnv ' This is a set by value,
```

Accessing ArcObjects

You will now see some specific uses of the create instance and query interface operations that involve ArcObjects. To use an ArcGIS object in Visual Basic or VBA, you must first reference the ESRI library that contains that object. If you are using VBA inside ArcMap or ArcCatalog, most of the common ESRI object libraries are already referenced for you. In standalone Visual Basic applications or components, you will have to manually reference the required libraries.

You will start by identifying a simple object and an interface that it supports. In this case, you will use a *Point* object and the *IPoint* interface. One way to set the coordinates of the point is to invoke the *PutCoords* method on the *IPoint* interface and pass in the coordinate values.

```
Dim pPt As IPoint
Set pPt = New Point
pPt.PutCoords 100, 100
```

The first line of this simple code fragment illustrates the use of a variable to hold a reference to the interface that the object supports. The line reads the *IID* for the *IPoint* interface from the ESRI object library. You may find it less ambiguous (as per the coding guidelines), particularly if you reference other object libraries in the same project, to precede the interface name with the library name, for example:

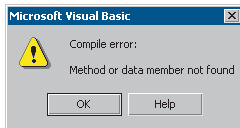
```
Dim pPt As esriCore.IPoint
```

To find out what library an ArcObjects component is in, either review the object model diagrams in the developer help or use the LibraryLocator tool in your development kit tools directory.

IID is short for Interface Identifier, a GUID.

Coclass is an abbreviation of component object class.

A QI is required since the default interface of the object is IUnknown. Since the pPt variable was declared as type IPoint, the default IUnknown interface was QI'd for the IPoint interface.



This is the compilation error message shown when a method or property is not found on an interface.

That way, if there happens to be another *IPoint* referenced in your project, there won't be any ambiguity as to which one you are referring.

The second line of the fragment creates an instance of the object or coclass, then performs a *QI* operation for the *IPoint* interface that it assigns to *pPt*.

With a name for the coclass as common as *Point*, you may want to precede the coclass name with the library name, for example:

```
Set pPt = New esriCore.Point
```

The last line of the code fragment invokes the *PutCoords* method. If a method can't be located on the interface, an error will be shown at compile time.

Working with HRESULTs

So far you have seen that all COM methods signify success or failure via an HRESULT that is returned from the method; no exceptions are raised outside of the interface. You have also learned that Visual Basic raises exceptions when errors are encountered. In Visual Basic, HRESULTs are never returned from method calls and, to confuse you further when errors do occur, Visual Basic throws an exception. How can this be? The answer lies with the Visual Basic Virtual Machine. It is the VBVM that receives the HRESULT; if this is anything other than *S_OK*, the VBVM throws the exception. If it was able to retrieve any worthwhile error information from the COM error object, it populates the Visual Basic *Err* object with that information. In this way, the VBVM handles all HRESULTs returned from the client.

When implementing interfaces in Visual Basic, it is good coding practice to raise an HRESULT error to inform the caller that an error has occurred. Normally, this is done when a method has not been implemented.

```
' Defined in module
Const E_NOTIMPL = &H80004001 'Constant that represents HRESULT
'Added to any method not implemented
On Error GoTo 0
Err.Raise E_NOTIMPL
```

You must also write code to handle the possibility that an HRESULT other than *S_OK* is returned. When this happens, an error handler should be called and the error dealt with. This may mean simply telling the user, or it may mean automatically dealing with the error and continuing with the function. The choice depends on the circumstances. Below is a simple error handler that will catch any error that occurs within the function and report it to the user. Note the use of the *Err* object to provide the user with some description of the error.

```
Private Sub Test()
    On Error GoTo ErrorHandler
    ' Do something here.
Exit Sub ' Must exit sub here before error handler
ErrorHandler:
    MsgBox "Error In Application - Description " & Err.Description
End Sub
```

Working with properties

Some properties refer to specific interfaces in the ESRI object library, and other properties have values that are standard data types, such as strings, numeric expressions, and Boolean values. For interface references, declare an interface variable and use the *Set* statement to assign the interface reference to the property. For other values, declare a variable with an explicit data type or use Visual Basic's *Variant* data type. Then, use a simple assignment statement to assign the value to the variable.

Properties that are interfaces can be set either by reference or by value. Properties that are set by value do not require the *Set* statement.

```
Dim pEnv As IEnvelope
Set pEnv = pActiveView.Extent 'Get extent property of view.
pEnv.Expand 0.5, 0.5, True 'Shrink envelope.
pActiveView.Extent = pEnv 'Set By Value extent back on IActiveView.
```

```
Dim pFeatureLayer as IFeatureLayer
Set pFeatureLayer = New FeatureLayer 'Create New Layer.
Set pFeatureLayer.FeatureClass = pClass 'Set ByRef a class into layer.
```

As you might expect, some properties are read-only, others are write-only, and still others are read-write. All the object browsers and the ArcObjects Class Help (found in the ArcGIS Developer Help system) provide this information. If you attempt to use a property and either forget or misuse the *Set* keyword, Visual Basic will fail the compilation of the source code with a method or data member not found error message. This error may seem strange since it may be given for trying to assign a value to a read-only property. The reason for the message is that Visual Basic is attempting to find a method in the type library that maps to the property name. In the above examples, the underlying method calls in the type library are *put_Extent* and *putref_FeatureClass*.

Working with methods

Methods perform some action and may or may not return a value. In some instances, a method returns a value that's an interface; for example, in the code fragment below, *EditSelection* returns an enumerated feature interface:

```
Dim pApp As IApplication
Dim pEditor As IEditor
Dim pEnumFeat As IEnumFeature 'Holds the selection
Dim pID As New UID
'Get a handle to the Editor extension
pID = "esriEditor.Editor"
Set pApp = Application
Set pEditor = pApp.FindExtensionByCLSID(pID)
'Get the selection
Set pEnumFeat = pEditor.EditSelection
```

In other instances, a method returns a Boolean value that reflects the success of an operation or writes data to a parameter; for example, the *DoModalOpen* method of *GxDialog* returns a value of *True* if a selection occurs and writes the selection to an *IEnumGxObject* parameter.

Be careful not to confuse the idea of a Visual Basic return value from a method call with the idea that all COM methods must return an HRESULT. The VBVM is able to read type library information and set up the return value of the VB method call to be the appropriate parameter of the COM method.

Working with events

Events let you know when something has occurred. You can add code to respond to an event. For example, a command button has a *Click* event. You add code to perform some action when the user clicks the control. You can also add events that certain objects generate. VBA and Visual Basic let you declare a variable with the keyword *WithEvents*. *WithEvents* tells the development environment that the object variable will be used to respond to the object's events. This is sometimes referred to as an "event sink". The declaration must be made in a class module or a form. Here's how you declare a variable and expose the events of an object in the *Declarations* section:

```
Private WithEvents m_pViewEvents as Map
```

Visual Basic only supports one outbound interface (marked as the default outbound interface in the IDL) per coclass. To get around this limitation, the coclasses that implement more than one outbound interface have an associated dummy coclass that allows access to the secondary outbound interface. These coclasses have the same name as the outbound interface they contain, minus the I.

```
Private WithEvents m_pMapEvents as MapEvents
```

Once you've declared the variable, search for its name in the Object combo box at the top left of the Code window. Then, inspect the list of events you can attach code to in the Procedure/Events combo box at the top right of the Code window.

Not all procedures of the outbound event interface need to be stubbed out, as Visual Basic will stub out any unimplemented methods. This is different from inbound interfaces, where all methods must be stubbed out for compilation to occur.

Before the methods are called, the hookup between the event source and sink must be made. This is done by setting the variable that represents the sink to the event source.

```
Set m_pMapEvents = pMxDoc.FocusMap
```

Pointers to valid objects as parameters

Some ArcGIS methods expect interfaces for some of their parameters. The interface pointers passed can point to an instanced object before the method call or after the method call is completed.

For example, if you have a polygon (*pPolygon*) whose center point you want to find, you can write code as follows:

```
Dim pArea As IArea
Dim pPt As IPoint
Set pArea = pPolygon      ' IArea for IArea on pPolygon
Set pPt = pArea.Center
```

You don't need to create *pPt* because the *Center* method creates a *Point* object for you and passes back a reference to the object via its *IPoint* interface. Only methods that use client-side storage require you to create the object prior to the method call.

Passing data between modules

When passing data between modules it is best to use accessor and mutator functions that manipulate some private member variable. This provides data encapsulation, which is a fundamental technique in object-oriented programming. Public variables should never be used.

For instance, you might have decided that a variable has a valid range of 1–100. If you were to allow other developers direct access to that variable, they could set the value to an illegal value. The only way of coping with these illegal values is to check them before they get used. This is both error prone and tiresome to program. The technique of declaring all variables private member variables of the class and providing accessor and mutator functions for manipulating these variables will solve this problem.

In the example below, these properties are added to the default interface of the class. Notice the technique used to raise an error to the client.

```
Private m_lPercentage As Long

Public Property Get Percentage() As Long
    Percentage = m_lPercentage
End Property

Public Property Let Percentage(ByVal lNewValue As Long)
    If (lNewValue >= 0) And (lNewValue <= 100) Then
        m_lPercentage = lNewValue
    Else
        Err.Raise vbObjectError + 29566, "MyProj.MyObject", _
            "Invalid Percentage Value. Valid values (0 -> 100)"
    End If
End Property
```

When you write code to pass an object reference from one form, class, or module to another, for example:

```
Private Property Set PointCoord(ByRef pPt As IPoint)
    Set m_pPoint = pPt
End Property
```

your code passes a pointer to an instance of the *IPoint* interface. This means that you are only passing the reference to the interface, not the interface itself; if you add the *ByVal* keyword (as follows), the interface is passed by value.

```
Private Property Let PointCoord(ByVal pPt As IPoint)
    Set m_pPoint = pPt
End Property
```

In both of these cases the object pointed to by the interfaces is always passed by reference. To pass the object by value, a clone of the object must be made, and that is passed.

Using the `TypeOf` keyword

To check whether an object supports an interface, you can use Visual Basic's `TypeOf` keyword. For example, given an item selected in the ArcMap table of contents, you can test whether it is a `FeatureLayer` using the following code:

```
Dim pDoc As IMxDocument
Dim pUnk As IUnknown
Dim pFeatLyr As IGeoFeatureLayer
Set pDoc = ThisDocument
Set pUnk = pDoc.SelectedItem
If TypeOf pUnk Is IGeoFeatureLayer Then ' can we QI for IGeoFeatureLayer?
    Set pFeatLyr = pUnk ' actually QI happens here
    ' Do something with pFeatLyr.
End If
```

Using the `Is` operator

If your code requires you to compare two interface reference variables, you can use the `Is` operator. Typically, you can use the `Is` operator in the following circumstances:

To check if you have a valid interface—for example, see the following code:

```
Dim pPt As IPoint
Set pPt = New Point
If (Not pPt Is Nothing) Then 'a valid pointer?
    ... ' do something with pPt
End If
```

To check if two interface variables refer to the same actual object—imagine that you have two interface variables of type `IPoint`, `pPt1`, and `pPt2`. Are they pointing to the same object? If they are, then `pPt1 Is pPt2`.

The `Is` keyword works with the COM identity of an object. Below is an example that illustrates the use of the `Is` keyword when finding out if a certain method on an interface returns a copy of or a reference to the same real object.

In the following example, the `Extent` property on a map (`IMap`) returns a copy, while the `ActiveView` property on a document (`IMxDocument`) always returns a reference to the real object.

```
Dim pDoc As IMxDocument
Dim pEnv1 As IEnvelope, pEnv2 as IEnvelope
Dim pActView1 As IActiveView
Dim pActView2 as IActiveView
Set pDoc = ThisDocument
Set pEnv1 = pDoc.ActiveView.Extent
Set pEnv2 = pDoc.ActiveView.Extent
Set pActView1 = pDoc.ActiveView
Set pActView2 = pDoc.ActiveView
' Extent returns a copy,
' so pEnv1 Is pEnv2 returns False
Debug.Print pEnv1 Is pEnv2
' ActiveView returns a reference,
' so pActView1 Is pActView2
Debug.Print pActView1 Is pActView2
```


Enumerators can support other methods, but these two methods are common among all enumerators.

Iterating through a collection

In your work with ArcMap and ArcCatalog, you'll discover that in many cases you'll be working with collections. You can iterate through these collections with an enumerator. An enumerator is an interface that provides methods for traversing a list of elements. Enumerator interfaces typically begin with *IEnum* and have two methods: *Next* and *Reset*. *Next* returns the next element in the set and advances the internal pointer, and *Reset* resets the internal pointer to the beginning.

Here is some VBA code that loops through the selected features (*IEnumFeature*) in a map. To try the code, add the States sample layer to the map and use the Select tool to select multiple features (drag a rectangle to do this). Add the code to a VBA macro, then execute the macro. The name of each selected state will be printed in the debug window.

```
Dim pDoc As IMxDocument
Dim pEnumFeat As IEnumFeature
Dim pFeat As IFeature
Set pDoc = ThisDocument
Set pEnumFeat = pDoc.FocusMap.FeatureSelection
Set pFeat = pEnumFeat.Next
Do While (Not pFeat Is Nothing)
    Debug.Print pFeat.Value(pFeat.Fields.FindField("state_name"))
    Set pFeat = pEnumFeat.Next
Loop
```

Some collection objects, the Visual Basic Collection being one, implement a special interface called *_NewEnum*. This interface, because of the *_* prefix, is hidden, but Visual Basic developers can still use it to simplify iterating through a collection. The Visual Basic *For Each* construct works with this interface to perform the *Reset* and *Next* steps through a collection.

```
Dim pColn as Collection
Set pColn = GetCollection()' Collection returned from some function

Dim thing as Variant      ' VB uses methods on _NewEnum to step through
For Each thing in pColn  ' an enumerator.
    MsgBox Cstr(thing)
Next
```

This section of the appendix discusses how to program in the VBA environment to control ArcGIS Desktop products—such as ArcMap, ArcCatalog, ArcScene, or ArcGlobe—by accessing the objects they expose. Your code manipulates the objects by getting and setting properties on their interfaces, such as setting the *MaximumScale* and *MinimumScale* of a map's *FeatureLayer*, invoking methods on the interfaces, such as adding a vertex to a polyline, or setting a field's value. The code runs when an event occurs, for example, when a user opens a document, clicks a button, or alters data by modifying an edit sketch.

First, though, you'll see the aspects of the VBA development environment in which you'll do your work that are specific to the ESRI applications. Consult the Visual Basic Reference, the online help file that displays when you click Microsoft Visual Basic Help in the Help menu of the VBA Editor for generic help on the user interface, conceptual topics, how-to topics, language reference topics, customizing the Visual Basic Editor, and user forms and controls.

In the VBA development environment you can add modules, class modules, and user forms to the default project contained in every ArcGIS application document. A project can consist of as many modules, class modules, and user forms as your work requires. A project is a collection of items to which you add code. A module is a set of declarations followed by procedures—a list of instructions that your code performs. A class module is a special type of module that contains the definition of a class, including its property and method definitions. A user form is a container for user interface controls, such as command buttons and text boxes.

ArcMap has a default project associated with its document that's listed in the Project Explorer as Project followed by its filename. In addition, you'll see another project listed in the Project Explorer called Normal (Normal.mxt).

Normal is, in fact, a template for all documents. It's always loaded into the document. It contains all the user-interface elements that users see, as well as the class module named ArcID, which contains all the UIDs for the application's commands.

Since any modifications made to Normal will be reflected every time you create or open a document, you should be careful when making changes to Normal.

In ArcMap, users can start by opening a template other than the default template. These templates are available to them in the New dialog box. From a developer's perspective this is a base template, a document that loads an additional project into the document; it is listed in the Project Explorer as the *TemplateProject* followed by its filename. This project can store code in modules, class modules, forms, and any other customizations, such as maps with data or page layout frames. Any modifications or changes made to this base template are reflected only in documents that are derived from it.

In ArcCatalog, Normal (Normal.gxt) is the only project that appears in the Project Explorer. There is no default Project in ArcCatalog, and you can't load any templates. You can, of course, add code to Normal.gxt inside modules, class modules, or forms, but again, be careful when making changes.

Once you've invoked the Visual Basic Editor, you can insert a module, class module, or user form. Then you insert a procedure or enter code for an existing

event procedure in the item's Code window, where you can write, display, and edit code. You can open as many Code windows as you have modules, class modules, and user forms, so you can easily view the code and copy and paste between Code windows. In addition to creating your own modules, you can import other modules, class modules, or user forms from disk.

If your work requires it, you can add an external object library or type library reference to your project. This makes another application's objects available in your code. Once a reference is set, the referenced objects are displayed in the development environment's object browser.

GETTING STARTED WITH VBA

To begin programming with VBA in ArcMap or ArcCatalog, you start the Visual Basic Editor.

To start the Visual Basic Editor

1. Start ArcMap or ArcCatalog.
2. Click the Tools menu, point to Macros, then click Visual Basic Editor. You can also use the shortcut keys Alt+F11 to display the Visual Basic Editor. To navigate among the projects in the Visual Basic Editor, use the Project Explorer. It displays a list of the document's modules, class modules, and user forms.

To add a macro to a module

ArcMap and ArcCatalog both provide a shortcut for creating a simple macro in a module.

1. Click the Tools menu, point to Macros, then click Macros.
2. Type the name of the macro you want to create in the Macro name text box. If you don't specify a module name, the application creates a module called *modulexx* and stores the macro in that module. If no module is specified after you specify a module, and a module is already active, the macro is placed in that module. Preceding a macro's name with a name and a dot stores it in a module with the specified name. If the module doesn't exist, the application creates it.
3. Click the dropdown arrow of the Macros in the combo box and choose the VBA project in which you want to create the macro.
4. Press the Enter key or click Create.
5. The stub for a Sub procedure for the macro appears in the Code window.

Adding modules and class modules

All ArcGIS application documents contain the class module *ThisDocument*, a custom object that represents the specific document associated with a VBA project. The document object is called *MxDocument* in ArcMap and *GxDocument* in ArcCatalog. The *IDocument* interface provides access to the document's title, type, accelerator table, command bars collection, parent application, and Visual Basic project.

Modules and class modules can contain more than one type of procedure: sub,

function, or property. You can choose the procedure type and its scope when you insert a procedure. Inserting a procedure is like creating a code template into which you enter code.

Every procedure has either private or public scope. Procedures with private scope are limited to the module that contains them—only a procedure within the same module can call a private procedure. If you declare the procedure public, other programs and modules can call it.

Variables in your procedures may either be local or global. Global variables exist during the entire time the code executes, whereas local variables exist only while the procedure in which they are declared is running. The next time you execute a procedure, all local variables are reinitialized. However, you can preserve the value of all local variables in a procedure for the code's lifetime by declaring them static, thereby fixing their value.

To add a procedure to an existing module

1. In the Project Explorer, double-click the ArcMap Objects, ArcCatalog Objects, or Modules folder, then choose the name of a module. Ensure that the code view of the module is active by clicking the View Code button.
2. Click the Insert menu and click Procedure.
3. Type the name of the procedure in the Name text box.
4. Click the Type dropdown arrow and click the type of procedure: Sub, Function, or Property.
5. Click the Scope dropdown arrow and click Public or Private.
6. To declare all local variables static, check the All Local variables as Statics check box.
7. Click OK. VBA stubs in a procedure into the item's Code window into which you can enter code. The stub contains the first and last lines of code for the type of procedure you've added.
8. Enter code into the procedure.

For more information about procedures, see the Microsoft Visual Basic online help reference.

Adding user forms

If you want your code to prompt the user for information, or you want to display the result of some action performed when the user invokes an ArcGIS application command or tool or in response to some other event, use VBA's user forms. User forms provide a context in which you can provide access to a rich set of integrated controls. Some of these controls are similar to the *UIControls* that are available as part of the Customize dialog box's Commands tab. In addition to text boxes or command buttons, you have access to a rich set of additional controls. A user form is a container for user-interface controls, such as command buttons and text boxes. A control is a Visual Basic object you place on a user form that has its own properties, methods, and events. You use controls to receive user input, display output, and trigger event procedures. You can set the form to be either modal, in which case the user must respond before using any other part of

the application, or modeless, in which case subsequent code is executed as it's encountered.

To add and start coding in a user form

1. In the Project Explorer, select the Project to which you want to add a user form.
2. Click the Insert menu and click UserForm.
3. VBA inserts a user form into your project and opens the Controls Toolbox.
4. Click the controls that you want to add to the user interface from the Controls Toolbox.
5. Add code to the user form or to its controls.

For more information about adding controls, see the Microsoft Visual Basic online help reference.

To display the Code window for a user form or control, double-click the user form or control. Then, choose the event you want your code to trigger from the dropdown list of events and procedures in the Code window and start typing your code. Or, just as in a module or class module, insert a procedure and start typing your code.

To display the form during an ArcMap or ArcCatalog session in response to some action, invoke its *Show* method, as in this example:

```
UserForm1.Show vbModeless 'show modeless
```

SOME VBA PROJECT MANAGEMENT TECHNIQUES

To work efficiently in the ArcGIS application's VBA development environment and reduce the amount of work you have to do every time you start a new task, make use of several techniques that will streamline your work:

Reusing modules, class modules, and user forms

To add an existing module or form to the Normal template, the Project, or a TemplateProject, click the name of the destination in the Project Explorer, then choose Import File from the File menu. You can choose any VBA module, user form, or class module to add a copy of the file to your project. To export an item from your project so that it is available for importing into other projects, choose the item you want to export in the Project Explorer, click Export File from the File menu, then navigate to where you want to save the file. Exporting an item does not remove it from your project.

Removing project items

When you remove an item, it is permanently deleted from the project list—you can't undo the Remove action; however, this action doesn't delete a file if it exists on disk. Before removing an item, make sure the remaining code in other modules and user forms doesn't refer to code in the removed item. To remove an item, select it in the Project Explorer, then click Remove <Name> from the File menu. Before you remove the item, you'll be asked whether you want to export it. If you click Yes in the message box, the Export File dialog box opens. If you click No, VBA deletes the item.

Protecting your code

To protect your code from alteration and viewing by users, you can lock a Project, a TemplateProject, or even the Normal template. When you lock one of these items, you set a password that must be entered before it can be viewed in the Project Explorer. To lock one of these items, right-click Project, TemplateProject, or Normal in the Project Explorer, then click the Properties item in the context menu that appears. In the Properties dialog box, click the Protection tab and click the option to Lock Project for Viewing. Enter a password and confirm it. Finally, save your ArcMap or ArcCatalog file and close it. The next time you or anyone else opens the file, the project is locked. If anyone wants to view or edit the project, they must enter the password.

Saving a VBA project

VBA projects are stored in a file that can be a base template (*.mxt), the Normal template, or a document (*.mxd). When a user creates a new ArcMap document from a base template, the new document references the base template's VBA project and its items. To save your ArcMap document and your VBA project, click Save from the ArcMap File menu or Save <File Name> from the File menu in the Visual Basic Editor. Both commands save your file with the project and any items stored in it. After saving the file, its filename is displayed in the Project Explorer in parentheses after the project name. To save the document as a template, click Save As from the ArcMap File menu and specify ArcMap Templates (*.mxt) as the File type.

Running VBA code

As you build and refine your code, you can run it within VBA to test and debug it. This section discusses running your code in the Visual Basic Editor during design time. For more information about running and debugging a VBA program, such as adding break points, adding watch expressions, and stepping into and out of execution, see Microsoft Visual Basic online help.

To run your code in the Visual Basic Editor or from the Macros dialog box

1. Click the Tools menu and click Macros.
2. In the Macro list, click the macro you want and click Run.

If the macro you want is not listed, make sure you've chosen the appropriate item: either Normal, Project, or TemplateProject in the Macros In box. Private procedures do not appear in any menus or dialog boxes.

To run only one procedure in the Visual Basic Editor

1. In the Project Explorer, open the module that contains the procedure that you want to run.
2. In the Code window, click an insertion point in the procedure code.
3. Click the Run menu and click Run Sub/UserForm.

Only the procedure in which your cursor is located runs.

After you've finished writing your code

After you have finished writing code, users can run it from ArcMap or ArcCatalog. To do this, click Macros and click it again from the Tools menu. You can also associate the code with a command or tool, or it can run in response to events or in other ways that you design.

USING THE GLOBAL APPLICATION OBJECTS

Since ArcCatalog does not support the use of documents, the ThisDocument global variable is not available to developers. However, the Application variable is available if a developer wishes to access IGxApplication or IApplication.

Application and *ThisDocument* are examples of global system variables that can be accessed by any module or class in the VBA environment while ArcMap is running. This variable is automatically set to reference the current document when ArcMap opens the document. You can use *ThisDocument* as a shortcut when programming in VBA to access the current document. Here is an example of how to use both the *Application* and *ThisDocument*:

```
Dim pMxDoc as IMxDocument
Set pMxDoc = Application.Document
'or
Set pMxDoc = ThisDocument
```

Both methods illustrated above result in a reference being set to the local document.

In the previous section of this appendix, the focus was primarily on how to write code in the VBA development environment embedded within the ArcGIS Desktop applications. This section focuses on particular issues related to creating ActiveX DLLs that can be added to the applications and writing external standalone applications using the Visual Basic development environment.

CREATING COM COMPONENTS

Most developers use Visual Basic to create a COM component that works with ArcMap or ArcCatalog. Earlier in this appendix you learned that since the ESRI applications are COM clients—their architecture supports the use of software components that adhere to the COM specification—you can build components with different languages, including Visual Basic. These components can then be added to the applications easily. For information about packaging and deploying COM components that you’ve built with Visual Basic, see the last section of this appendix.

This section is not intended as a Visual Basic tutorial; rather, it highlights aspects of Visual Basic that you should know to be effective when working with ArcObjects.

In Visual Basic you can build a COM component that will work with ArcMap or ArcCatalog by creating an ActiveX DLL. This section will review the rudimentary steps involved. Note that these steps are not all-inclusive. Your project may involve other requirements.

1. Start Visual Basic. In the New Project dialog box, create an ActiveX DLL Project.
2. In the Properties window, make sure that the Instancing property for the initial class module and any other class modules you add to the Project is set to 5—MultiUse.
3. Reference the ESRI Object Libraries that you will require.
4. Implement the required interfaces. When you implement an interface in a class module, the class provides its own versions of all the public procedures specified in the type library of the interface. In addition to providing mapping between the interface prototypes and your procedures, the *Implements* statement causes the class to accept COM *QueryInterface* calls for the specified interface ID. You must include all the public procedures involved. A missing member in an implementation of an interface or class causes an error. If you don’t put code in one of the procedures in a class you are implementing, you can raise the appropriate error (*Const E_NOTIMPL = &H80004001*). That way, if someone else uses the class, they’ll understand that a member is not implemented.
5. Add any additional code that’s needed.
6. Establish the Project Name and other properties to identify the component. In the Project Properties dialog box, the project name you specify will be used as the name of the component’s type library. It can be combined with the name of each class the component provides to produce unique class names (these names are also called ProgIDs). These names appear in the Component Category Manager. Save the project.

The ESRI VB Add-In interface implementer can be used to automate Steps 3 and 4.

Visual Basic automatically generates the necessary GUIDs for the classes, interfaces, and libraries. Setting binary compatibility forces VB to reuse the GUIDs from a previous compilation of the DLL. This is essential since ArcMap stores the GUIDs of commands in the document for subsequent loading.

7. Compile the DLL.
8. Set the component's Version Compatibility to binary. As your code evolves, it's good practice to set the components to Binary Compatibility so, if you make changes to a component, you'll be warned that you're breaking compatibility. For additional information, see the 'Binary compatibility mode' help topic in the Visual Basic online help.
9. Save the project.
10. Make the component available to the application. You can add a component to a document or template by clicking the Add from file button in the Customize dialog box's Commands tab. In addition, you can register a component in the Component Category Manager.

IMPLEMENTING INTERFACES

You implement interfaces differently in Visual Basic depending if they are inbound or outbound interfaces. An outbound interface is seen by Visual Basic as an event source and is supported through the *WithEvents* keyword. To handle the outbound interface, *IActiveViewEvents*, in Visual Basic (the default outbound interface of the *Map* class), use the *WithEvents* keyword and provide appropriate functions to handle the events.

```
Private WithEvents ViewEvents As Map
```

```
Private Sub ViewEvents_SelectionChanged()  
    ' User changed feature selection update my feature list form  
    UpdateMyFeatureForm  
End Sub
```

Inbound interfaces are supported with the *Implements* keyword. However, unlike the outbound interface, all the methods defined on the interface must be stubbed out. This ensures that the vTable is correctly formed when the object is instantiated. Not all of the methods have to be fully coded, but the stub functions must be there. If the implementation is blank, an appropriate return code should be given to any client to inform them that the method is not implemented (see the section 'Working with HRESULTs'). To implement the *IExtension* interface, code similar to that below is required. Note that all the methods are implemented.

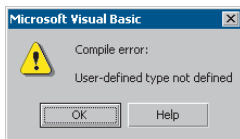
```
Private m_pApp As IApplication  
Implements IExtension  
Private Property Get IExtension_Name() As String  
    IExtension_Name = "Sample Extension"  
End Property
```

```
Private Sub IExtension_Startup(ByRef initializationData As Variant)  
    Set m_pApp = initializationData  
End Sub
```

```
Private Sub IExtension_Shutdown()  
    Set m_pApp = Nothing  
End Sub
```

SETTING REFERENCES TO THE ESRI OBJECT LIBRARIES

The principal difference between working with the VBA development environment embedded in the applications and working with Visual Basic is that the latter environment requires that you load the appropriate object libraries so that any object variables that you declare can be found. If you don't add the reference, you'll get the error message to the left. In addition, the global variables *ThisDocument* and *Application* are not available to you.



To add a reference to an object library

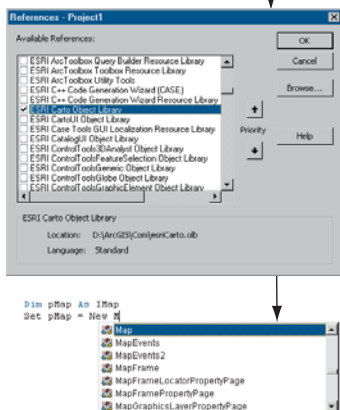
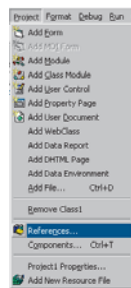
Depending on what you want your code to do, you may need to add several ESRI core object and extension libraries. You can determine what library an object belongs to by reviewing the object model diagrams in the developer help or by using the LibraryLocator tool located in the tools directory of your developer kit.

To display the References dialog box in which you can set the references you need, select References in the Visual Basic Project menu.

After you set a reference to an object library by selecting the check box next to its name, you can find a specific object and its methods and properties in the object browser.

If you are not using any objects in a referenced library, you should clear the check box for that reference to minimize the number of object references Visual Basic must resolve, thus reducing the time it takes your project to compile. You should not remove a reference for an item that is used in your project.

You can't remove the "Visual Basic for Applications" and "Visual Basic objects and procedures" references because they are necessary for running Visual Basic.



After the ESRI Object Library is referenced, all the types contained within it are available to Visual Basic. IntelliSense will also work with the contents of the object library.

REFERRING TO A DOCUMENT

Each VBA project (Normal, Project, TemplateProject) has a class called *ThisDocument*, which represents the document object. Anywhere you write code in VBA you can reference the document as *ThisDocument*. Further, if you are writing your code in the *ThisDocument* Code window, you have direct access to all the methods and properties on *IDocument*. This is not available in Visual Basic. You must first get a reference to the *Application*, then the document. When adding both extensions and commands to ArcGIS applications, a pointer to the *IApplication* interface is provided.

```
Implements IExtension
Private m_pApp As IApplication
```

```
Private Sub IExtension_Startup(ByRef initializationData As Variant)
    Set m_pApp = initializationData ' Assign IApplication.
End Sub
```

```
Implements ICommand
Private m_pApp As IApplication
```

```
Private Sub ICommand_OnCreate(ByVal hook As Object)
    Set m_pApp = hook ' QI for IApplication
End Sub
```

Now that a reference to the application is in an *LApplication* pointer member variable, the document and, hence, all other objects can be accessed from any method within the class.

```
Dim pDoc as IDocument
Set pDoc = m_pApp.Document
MsgBox pDoc.Name
```

GETTING TO AN OBJECT

In the previous example, navigating around the objects within ArcMap is a straightforward process since a pointer to the *Application* object, the root object of most of the ArcGIS application's objects, is passed to the object via one of its interfaces. This, however, is not the case with all interfaces that are implemented within the ArcObjects application framework. There are cases when you may implement an object that exists within the framework and there is no possibility to traverse the object hierarchy from that object. This is because very few objects support a reference to their parent object (the *IDocument* interface has a property named *Parent* that references the *LApplication* interface). To give developers access to the application object, there is a singleton object that provides a pointer to the running application object. The code below illustrates its use.

```
Dim pAppRef As New AppRef
Dim pApp as IApplication
Set pApp = pAppRef
```

You must be careful to ensure that this object is only used where the implementation will always run only within ArcMap and ArcCatalog. For instance, it would not be a good idea to make use of this function from within a custom feature since that would restrict what applications could be used to view the feature class.

RUNNING ARCMAP WITH A COMMAND LINE ARGUMENT

You can start ArcMap from the command line and pass it an argument that is either the pathname of a document (.mxd) or the pathname of a template (.mxt). In the former case, ArcMap will open the document; in the latter case, ArcMap will create a new document based on the template specified.

You can also pass an argument and create an instance of ArcMap by supplying arguments to the Win32 APIs *ShellExecute* function or Visual Basic's *Shell* function as follows:

```
Dim ret As Variant
ret = Shell("d:\arcgis\bin\arcmap.exe _
d:\arcgis\bin\templates\LetterPortrait.mxt", vbNormalFocus)
```

By default, *Shell* runs other programs asynchronously. This means that ArcMap might not finish executing before the statements following the *Shell* function are executed.

To execute a program and wait until it is terminated, you must call three Win32 API functions. First, call the *CreateProcessA* function to load and execute ArcMap. Next, call the *WaitForSingleObject* function, which forces the operating system to wait until ArcMap has been terminated. Finally, when the user has terminated the application, call the *CloseHandle* function to release the application's 32-bit identifier to the system pool.

Singletons are objects that only support one instance of the object. These objects have a class factory that ensures that anytime an object is requested, a pointer to an already existing object is returned.

In Visual Basic, it is not possible to determine the command line used to start the application.

There is a sample on disk that provides this functionality. It can be found at <ArcGIS Developer Kit install>\samples\COM Techniques\Command Line.

DEBUGGING VISUAL BASIC CODE

Visual Basic has a debugger integrated into its development environment. This is in many cases a valuable tool when debugging Visual Basic code; however, in some cases it is not possible to use the VB debugger. The use of the debugger and these special cases are discussed below.

Running the code within an application

It is possible to use the Visual Basic debugger to debug your ArcObjects software-based source code even when ActiveX DLLs are the target server. The application that will host your DLL must be set as the Debug application. To do this, select the appropriate application, ArcMap.exe, for instance, and set it as the Start Program in the Debugging Options of the Project Properties.

Using commands on the Debug toolbar, ArcMap can be started and the DLL loaded and debugged. Break points can be set, lines stepped over, functions stepped into, and variables checked. Moving the line pointer in the left margin can also set the current execution line.

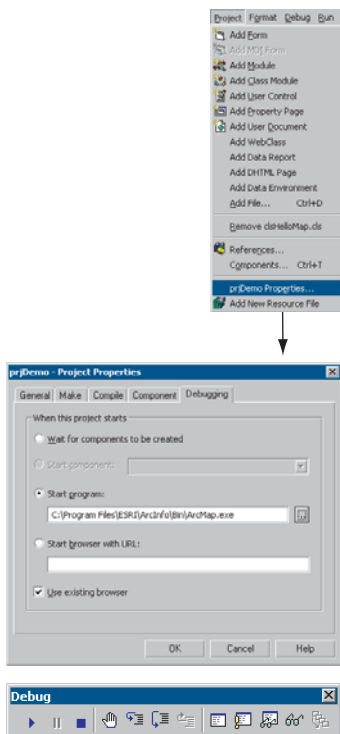
Visual Basic debugger issues

In many cases, the Visual Basic debugger will work without any problems; however, there are two problems when using the debugger that is supplied with Visual Basic 6. Both of these problems exist because of the way that Visual Basic implements its debugger.

Normally when running a tool within ArcMap, the DLL is loaded into ArcMap address space, and calls are made directly into the DLL. When debugging, this is not the case. Visual Basic makes changes to the registry so that the CLSID for your DLL does not point to your DLL but, instead, it points to the Visual Basic Debug DLL (VB6debug.dll). The Debug DLL must then support all the interfaces implemented by your class on the fly. With the VB Debug DLL loaded into ArcMap, any method calls that come into the DLL are forwarded on to Visual Basic, where the code to be debugged is executed. The two problems with this are caused by the changes made to the registry and the cross-process space method calling. When these restrictions are first encountered, it can be confusing since the object works outside the debugger or at least until it hits the area of problem code.

Since the method calls made from ArcMap to the custom tool are across apartments, there is a requirement for the interfaces to be marshalled. This marshalling causes problems in certain circumstances. Most data types can be automatically marshalled by the system, but there are a few that require custom code because the standard marshaller does not support the data types. If one of these data types is used by an interface within the custom tool and there is no custom marshalling code, the debugger will fail with an “Interface not supported error”.

The registry manipulation also breaks the support for component categories. Any time there is a request on a component category, the category manager within COM will be unable to find your component because, rather than asking whether your DLL belongs to the component category, COM is asking whether the VB



debugger DLL belongs to the component category, and it doesn't. What this means is that anytime a component category is used to automate the loading of a DLL, the DLL cannot be debugged using the Visual Basic debugger.

This causes problems for many of the ways to extend the framework. The most common way to extend the framework is to add a command or tool. Previously, it was discussed how component categories were used in this instance. Remember the component category was only used to build the list of commands in the dialog box. This means that if the command to be debugged is already present on a toolbar, the Visual Basic debugger can be used. Hence, the procedure for debugging Visual Basic objects that implement the *ICommand* interface is to ensure that the command is added to a toolbar when ArcMap is executed standalone and, after saving the document, load ArcMap through the debugger.

In some cases, such as extensions and property pages, it is not possible to use the Visual Basic debugger. If you have access to the Visual C++ debugger, you can use one of the options outlined below. Fortunately, there are a number of ESRI Visual Basic Add-ins that make it possible to track down the problem quickly and effectively. The add-ins, described in the ArcGIS Developer Help in the section 'Visual Basic Developer Add-Ins', provide error log information including line and module details. A sample output from an error log is given below; note the call stack information along with line numbers.

Error Log saved on : 8/28/2000 - 10:39:04 AM
 Record Call Stack Sequence - Bottom line is error line.

```
chkVisible_MouseUp C:\Source\MapControl\Commands\frmLayer.frm Line : 196
RefreshMap C:\Source\MapControl\Commands\frmLayer.frm Line : 20
```

Description

Object variable or With block variable not set

Alternatives to the Visual Basic debugger

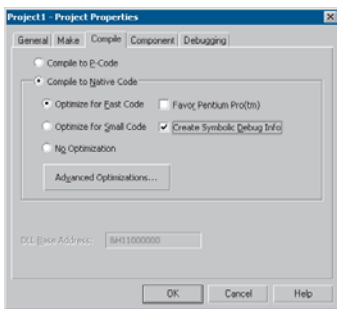
If the Visual Basic debugger and add-ins do not provide enough information, the Visual C++ debugger can be used, either on its own or with C++ ATL wrapper classes. The Visual C++ debugger does not run the object to be debugged out of process from ArcMap, which means that none of the above issues apply. Common debug commands are given in the Visual C++ section 'Debugging tips in Developer Studio'. Both of the techniques below require the Visual Basic project to be compiled with Debug Symbol information.

The Visual C++ debugger can work with this symbolic debug information and the source files.

Visual C++ debugger

It is possible to use the Visual C++ debugger directly by attaching to a running process that has the Visual Basic object to be debugged loaded and setting a break point in the Visual Basic file. When the line of code is reached, the debugger will halt execution and step into the source file at the correct line. The required steps are as follows:

1. Start an appropriate application, such as ArcMap.exe.



Create Debug Symbol information using the Create Symbolic Debug info option on the Compile tab of the Project Properties dialog box.

2. Start Microsoft Visual C++.
3. Attach to the ArcMap process using Menu option Build > Start Debug > Attach to process.
4. Load the appropriate Visual Basic Source file into the Visual C++ debugger and set the break point.
5. Call the method within ArcMap.

No changes can be made to the source code within the debugger, and variables cannot be inspected, but code execution can be viewed and altered. This is often sufficient to determine what is wrong, especially with logic-related problems.

ATL wrapper classes

Using the Active Template Library, you can create a class that implements the same interfaces as the Visual Basic class. When you create the ATL object, you create the Visual Basic object. All method calls are then passed to the Visual Basic object for execution. You debug the contained object by setting a break point in the appropriate C++ wrapper method, and when the code reaches the break point, the debugger is stepped into the Visual Basic code. For more information on this technique, look at the ATL Debugger sample in the Developer Samples of the ArcGIS Developer Help system.

Developing in Visual C++ is a large and complex subject, as it provides a much lower level of interaction with the underlying Windows APIs and COM APIs when compared to other development environments.

While this can be a hindrance for rapid application development, it is the most flexible approach. A number of design patterns, such as COM aggregation and singletons, that are possible in Visual C++ are not possible in Visual Basic 6. By using standard class libraries, such as Active Template Library, the complex COM plumbing code can be hidden. However, it is still important to have a thorough understanding of the underlying ATL COM implementation.

The documentation in this section is based on Microsoft Visual C++ version 6 and provides some guidance for ArcGIS development in this environment. With the release of Visual Studio C++ .NET, (also referred to as VC7), many new enhancements are available to the C++ developer. While VC7 can work with the managed .NET environment, and it is possible to work with the ArcGIS .NET API, this will only add overhead to access the underlying ArcGIS COM objects. So for the purposes of ArcGIS development in VC7, it is recommended to work the “traditional” way—that is, directly with the ArcGIS COM interfaces and objects.

There are many enhancements to ATL in VC7. Some of the relevant changes are covered in the section ‘ATL in Visual C++ .NET’, later in this appendix.

With the addition of the Visual C# .NET language, it is worth considering porting Visual C++ code to this environment and using the ArcGIS .NET API. The syntax of C# is not unlike C++, but the resulting code is generally simpler and more consistent.

This section is intended to serve two main purposes:

1. To familiarize you with general Visual C++ coding style and debugging, beginning with a discussion on ATL
2. To detail specific usage requirements and recommendations for working with the ArcObjects programming platform in Visual C++

WORKING WITH ATL

This section cannot cover all the topics that a developer working with ATL should know to be effective, but it will serve as an introduction to ATL. ATL helps you implement COM objects and saves typing, but it does not excuse you from knowing C++ and how to develop COM objects.

ATL is the recommended framework for implementing COM objects. The ATL code can be combined with Microsoft Foundation Class Library (MFC) code, which provides more support for writing applications. An alternative to MFC is the Windows Template Library (WTL), which is based on the ATL template methodology and provides many wrappers for window classes and other application support for ATL. WTL is available for download from Microsoft; at the time of writing, version 7.1 is the latest and can be used with Visual C++ version 6 and Visual C++ .NET.

ATL in brief

ATL is a set of C++ template classes designed to be small, fast, and extensible, based loosely on the Standard Template Library (STL). STL provides generic template classes for C++ objects, such as vectors, stacks, and queues. ATL also

provides a set of wizards that extend the Visual Studio development environment. These wizards automate some of the tedious plumbing code that all ATL projects must have. The wizards include, but are not limited to, the following:

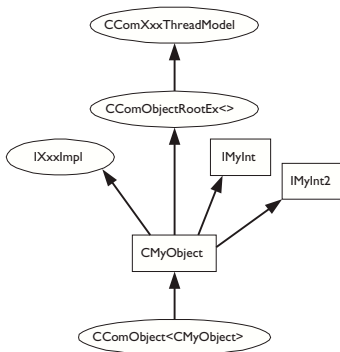
- Application—Used to initialize an ATL C++ project.
- Object—Used to create COM objects. Both C++ and IDL code is generated, along with the appropriate code to support the creation of the objects at run time.
- Property—Used to add properties to interfaces.
- Method—Used to add methods to interfaces; both the Property and Method wizards require you to know some IDL syntax.
- Interface Implementation—Used to implement stub functions for existing interfaces.
- Connection Point Implement—Used to implement outbound events' interfaces.

Typically these are accessed by a right-click on a project, class, or interface in Visual Studio Workspace/Class view.

ATL provides base classes for implementing COM objects as well as implementations for some of the common COM interfaces, including *IUnknown*, *IDispatch*, and *IClassFactory*. There are also classes that provide support for ActiveX controls and their containers.

ATL provides the required services for exposing ATL-based COM objects, including registration, server lifetime, and class objects.

These template classes build a hierarchy that sandwiches your class. These inheritances are shown to the left. The *CComXxxThreadModel* class supports thread-safe access to global, instance, and static data. The *CComObjectRootEx* class provides the behavior for the *IUnknown* methods. The interfaces at the second level represent the interfaces that the class will implement; these come in two varieties. The *IXxxImpl* interface contains ATL-supplied interfaces that also include an implementation; the other interfaces have pure virtual functions that must be fully implemented within your class. The *CComObject* class inherits your class; this class provides the implementation of the *IUnknown* methods along with the object instantiation and lifetime control.



The hierarchical layers of ATL

A more detailed discussion on Direct-To-COM, follows in the section 'Direct-To-COM smart types'.

ATL and DTC

Along with smart types, covered later in this appendix, Direct-To-COM (DTC) provides some useful compiler extensions you can use when creating ATL-based objects. The functions `__declspec` and `__uuidof` are two such functions, but the most useful is the `#import` command.

COM interfaces are defined in IDL, then compiled by the Microsoft IDL compiler (MIDL.exe). This results in the creation of a type library and header files. The project uses these files automatically when compiling software that references these interfaces. This approach is limited in that, when working with interfaces, you must have access to the IDL files. As a developer of ArcGIS, you only have access to the ArcGIS type library information contained in .olb and

.ocx files. While it is possible to engineer a header file from a type library, it is a tedious process. The `#import` command automates the creation of the necessary files required by the compiler. Since the command was developed to support DTC, when using it to import ArcGIS type libraries, there are a number of parameters that must be passed so that the correct import takes place. For further information on this process, see the later section ‘Importing ArcGIS type libraries’.

Handling errors in ATL

It is possible to just return an `E_FAIL HRESULT` code to indicate the failure within a method; however, this does not give the caller any indication of the nature of the failure. There are a number of standard Windows `HRESULT`s available, for example, `E_INVALIDARG` (one or more arguments are invalid) and `E_POINTER` (invalid pointer). These error codes are listed in the Windows header file `winerror.h`. Not all development environments have comprehensive support for `HRESULT`; Visual Basic clients often see error results as “Automation Error – Unspecified Error”. ATL provides a simple mechanism for working with the COM error information object that can provide an error string description, as well as an error code.

When creating an ATL object, the Object wizard has an option to support `ISupportErrorInfo`. If you toggle the option on, when the wizard completes, your object will implement the interface `ISupportErrorInfo`, and a method will be added that looks something like this:

```
STDMETHODIMP MyClass::InterfaceSupportsErrorInfo(REFIID riid)
{
    static const IID* arr[] =
    {
        &IID_IMyClass,
    };

    for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++)
    {
        if (InlineIsEqualGUID(*arr[i], riid))
            return S_OK;
    }

    return S_FALSE;
}
```

It is now possible to return rich error messages by calling one of the ATL error functions. These functions even work with resource files to ensure easy internationalization of the message strings.

Although Visual C++ does support an exception mechanism (try ... catch), it is not recommended to mix this with COM code. If an exception unwinds out of a COM interface, there is no guarantee the client will be able to catch this, and the most likely result is a crash.

```
// Return a simple string
AtlReportError(CLSID_MyClass, _T("No connection to Database."),
IID_IMyClass, E_FAIL);
// Get the Error Text from a resource string
AtlReportError(CLSID_MyClass, IDS_DBERROR, IID_IMyClass, E_FAIL,
_Module.m_hInstResource);
```

To extract an error string from a failed method, use the Windows function `GetErrorInfo`. This is used to retrieve the last `IErrorInfo` object on the current thread and clears the current error state.

Linking ATL code

One of the primary purposes of ATL is to support the creation of small fast objects. To support this, ATL gives the developer a number of choices when compiling and linking the source code. Choices must be made about how to link or dynamically access the C run-time (CRT) libraries, the registration code, and the various ATL utility functions. If no CRT calls are made in the code, this can be removed from the link. If CRT calls are made and the linker switch `_ATL_MIN_CRT` is not removed from the link line, the following error will be generated during the build:

```
LIBCMT.lib(crt0.obj) : error LNK2001: unresolved external symbol _main
ReleaseMinSize/History.d11 : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
```

When compiling a debug build, there will probably not be a problem; however, depending on the code written, there may be problems when compiling a release build. If you receive this error either remove the CRT calls or change the linker switches.

If the utilities code is dynamically loaded at run time, you must ensure that the appropriate DLL (ATL.DLL) is installed and registered on the user's system. The ArcGIS 9 run-time installation will install ATL.dll. The table below shows the various choices and the related linker switches.

	Symbols	CRT	Utilities	Registrar
Debug		yes	static	dynamic
RelMinDepend	<code>_ATL_MIN_CRT</code> <code>_ATL_STATIC_REGISTRY</code>	no	static	static
RelMinSize	<code>_ATL_MIN_CRT</code> <code>_ATL_DLL</code>	no	dynamic	dynamic

By default, there are build configurations for ANSI and Unicode builds. A component that is built with ANSI compilation will run on Windows 9.x; however, considering that ArcGIS is only supported on unicode operating systems (Windows NT, Windows 2000, and Windows XP), these configurations are redundant. To delete a configuration in Visual Studio, click Build / Configurations ...?. Then delete *Win32 Debug*, *Win32 Release MinSize*, and *Win32 Release MinDependency*.

Registration of a COM component

The ATL project wizard generates the standard Windows entry points for registration. This code will register the DLL's type library and execute a registry script file (.rgs) for each COM object within the DLL. Additional C++ code to perform other registration tasks can be inserted into these functions.

```
STDAPI DllRegisterServer(void)
{
    // registers object in .rgs, typelib and all interfaces in typelib
    // TRUE instructs the type library to be registered
    return _Module.RegisterServer(TRUE);
}

STDAPI DllUnregisterServer(void)
{

```

```
return _Module.UnregisterServer(TRUE);
}
```

ATL provides a text file format, .rgs, that is parsed by the ATL's registrar component when a DLL is registered and unregistered. The .rgs file is built into a DLL as a custom resource. The file can be edited to add additional registry entries and contains ProgID, ClassID, and component category entries to place in the registry. The syntax describes keys, values, names, and subkeys to be added or removed from the registry. The format can be summarized as follows:

```
[NoRemove | ForceRemove | val] Name | [ = s 'Value' | d 'Value' | b 'Value' ]
{
    .. optional subkeys for the registry
}
```

NoRemove signifies that the registry key should not be removed on unregistration. *ForceRemove* will ensure the key and subkeys are removed before registering the new keys. The *s*, *d*, and *b* values indicate string (enclosed with apostrophes), double word (32-bit integer value), and binary registry values. A typical registration script is shown below.

HKCR

```
{
    SimpleObject.SimpleCOMObject.1 = s 'SimpleCOMObject Class'
    {
        CLSID = s '{2AFFC10E-ECFB-4697-8B3D-0405650B7CFB}'
    }
    SimpleObject.SimpleCOMObject = s 'SimpleCOMObject Class'
    {
        CLSID = s '{2AFFC10E-ECFB-4697-8B3D-0405650B7CFB}'
        CurVer = s 'SimpleObject.SimpleCOMObject.1'
    }
    NoRemove CLSID
    {
        ForceRemove {2AFFC10E-ECFB-4697-8B3D-0405650B7CFB} = s 'SimpleCOMObject
Class'
        {
            ProgID = s 'SimpleObject.SimpleCOMObject.1'
            VersionIndependentProgID = s 'SimpleObject.SimpleCOMObject'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingMode1 = s 'Apartment'
            }
            'TypeLib' = s '{855DD226-5938-489D-986E-149600FEDD63}'
            'Implemented Categories'
            {
                {7DD95801-9882-11CF-9FA9-00AA006C42C4}
            }
        }
    }
}
```

NoRemove CLSID ensures the registry key *CLSID* is never removed. This is the subkey below which all COM objects use to register their ProgIDs and GUIDs, so

its removal would result in a serious corruption of the registry. *InprocServer32* is the standard COM mechanism that relates a component GUID to a DLL file; ATL will insert the correct module name using the %MODULE% variable. Other entries under the GUID specify the ProgID, threading model, and type library to use with this component.

To register a COM coclass into a component category, there are two approaches. The recommended approach is illustrated above: place GUIDs for component categories beneath an Implemented Categories key, which in turn is under the GUID of the coclass. The second approach is to use ATL macros in an objects header file: `BEGIN_CATEGORY_MAP`, `IMPLEMENTED_CATEGORY`, and `END_CATEGORY_MAP`. However, these macros do not correctly remove registry entries as explained in MSDN article [Q279459](#) *BUG: Component Category Registry Entries Not Removed in ATL Component*. A header file is supplied with the GUIDs of all the component categories used by ArcGIS; this is available in `\Program Files\ArcGIS\include\CatIDs\ArcCATIDs.h`.

If the GUID of a component is changed during development or the type library name is changed, then it is important to keep the .rgs content consistent with these changes; otherwise, the registry will be incorrect and object creation can fail.

Debugging ATL code

In addition to the standard Visual Studio facilities, ATL provides a number of debugging options with specific support for debugging COM objects. The output of these debugging options is displayed in the Visual C++ Output window. The *QueryInterface* call can be debugged by setting the symbol `_ATL_DEBUG_QI`, *AddRef* and *Release* calls with the symbol `_ATL_DEBUG_INTERFACES`, and leaked objects can be traced by monitoring the list of leaked interfaces at termination time when the `_ATL_DEBUG_INTERFACES` symbol is defined. The leaked interfaces list has entries like the following:

```
INTERFACE LEAK: RefCount = 1, MaxRefCount = 3, {Allocation = 10}
```

On its own, this does not tell you much apart from the fact that one of your objects is leaking because an interface pointer has not been released. However, the *Allocation* number allows you to automatically break when that interface is obtained by setting the `m_nIndexBreakAt` member of the *CComModule* at server startup. This in turn calls the function `DebugBreak()` to force the execution of the code to stop at the relevant place in the debugger. For this to work the program flow must be the same.

```
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID /
*lpReserved*/)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        _Module.Init(ObjectMap, hInstance, &LIBID_HISTORYLib);
        DisableThreadLibraryCalls(hInstance);
        _Module.m_nIndexBreakAt = 10;
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        _Module.Term();
    }
    return TRUE;
}
```

Boolean types

Historically, ANSI C did not have a Boolean data type and used int value instead, where 0 represents false and nonzero represents true. However, the bool data-type has now become part of ANSI C++. COM APIs are language independent and define a different Boolean type, VARIANT_BOOL. In addition, Win32 API uses a different bool type. It is important to use the correct type at the appropriate time. The following table summarizes their usage:

Type	True value	False value	Where defined	When to use
bool	true (1)	false (0)	Defined by compiler	This is an intrinsic compiler type so there is more potential for the compiler to optimize its use. This type can also be promoted to an int value. Expressions (e.g., !i==0) return a type of bool. Typically used for class member variables and local variables.
BOOL (int)	TRUE (1)	FALSE (0)	Windows Data Type (defined in windef.h)	Used with windows API functions, often as a return value to indicate success or failure.
VARIANT_BOOL (16 bit short)	VARIANT_TRUE (-1)	VARIANT_FALSE (0)	COM Boolean values (wtypes.h)	Used in COM APIs for boolean values. Also used within VARIANT types; if the VARIANT type is VT_BOOL, then the VARIANT value (boolVal) is populated with a VARIANT_BOOL. Take care to convert a bool class member variable to the correct VARIANT_BOOL value. Often the conditional test "hook - colon" operator is used. For example, where bRes is defined as a bool, then set a result type: *pVal = bRes ? VARIANT_TRUE : VARIANT_FALSE;

String types

Considering that strings (sequences of text characters) are a simple concept, they have unfortunately become a complex and confusing topic in C++. The two main reasons for this confusion are the lack of C++ support for variable length strings combined with the requirement to support ANSI and Unicode character sets within the same code. As ArcGIS is only available on Unicode platforms, it may simplify development to remove the ANSI requirements.

The C++ convention for strings is an array of characters terminated with a 0. This is not always good for performance when calculating lengths of large strings. To support variable length strings, the character arrays can be dynamically allocated and released on the heap, typically using *malloc* and *free* or *new* and *delete*. Consequently, a number of wrapper classes provide this support; CString defined in MFC and WTL is the most widely used. In addition, for COM usage the BSTR type is defined and the ATL wrapper class CComBSTR is available.

To allow for international character sets, Microsoft Windows migrated from an 8-bit ANSI character string (8-bit character) representation (found on Windows 95, Windows 98, and Windows Me platforms) to a 16-bit Unicode character string (16-bit unsigned short). Unicode is synonymous with wide characters (wchar_t). In COM APIs, OLECHAR is the type used and is defined to be wchar_t on Windows. Windows operating systems, such as Windows NT, Windows 2000, and Windows XP, natively support Unicode characters. To allow the same C++ code to be compiled for ANSI and Unicode platforms, compiler switches are used to change Windows API functions (for example, SetWindowText) to resolve to an ANSI version (SetWindowTextA) or a Unicode version (SetWindowTextW). In addition, character-independent types (TCHAR defined in tchar.h) were introduced to represent a character; on an ANSI build this is defined to be a *char*, and on a Unicode build this is a *wchar_t*, a typedef

defined as unsigned short. To perform standard C string manipulation, there are typically three different definitions of the same function; for example, for a case-insensitive comparison, *strcmp* provides the ANSI version, *wscmp* provides the Unicode version, and *_tscmp* provides the TCHAR version. There is also a fourth version, *_mbicmp*, which is a variation of the 8-bit ANSI version that will interpret multibyte character sequences (MBCS) within the 8-bit string.

```
// Initialize some fixed length strings
char*   pNameANSI = "Bill"; // 5 bytes (4 characters plus a terminator)
wchar_t* pNameUNICODE = L"Bill"; // 10 bytes (4 16-bit characters plus a
                                  16-bit terminator)
TCHAR*   pNameTCHAR = _T("Bill"); // either 5 or 10 depending on compiler
                                   settings
```

COM APIs represent variable length strings with a BSTR type; this is a pointer to a sequence of OLECHAR characters, which is defined as Unicode characters and is the same as a wchar_t. A BSTR must be allocated and released with the SysAllocString and SysFreeString windows functions. Unlike C strings, they can contain embedded zero characters although this is unusual. The BSTR also has a count value, which is stored four bytes before the BSTR pointer address. The CComBSTR wrappers are often used to manage the lifetime of a string.

Do not pass a pointer to a C style array of Unicode characters (OLECHAR or wchar_t) to a function expecting a BSTR. The compiler will not raise an error as the types are identical. However, the function receiving the BSTR can behave incorrectly or crash when accessing the string length, which will be random memory values.

```
ipFoo->put_WindowTitle(L"Hello"); // This is bad!
ipFoo->put_WindowTitle(CComBSTR(L"Hello")); // This correctly initializes
                                           and passes a BSTR
```

ATL provides conversion macros to switch strings between ANSI (A), TCHAR (T), Unicode (W), and OLECHAR (OLE). In addition, the types can have a const modifier (C). These macros use the abbreviations shown in brackets with a “2” between them. For example, to convert between OLECHAR (for example, an input BSTR) to const TCHAR (for use in a Windows function), use the OLE2CT conversion macro. To convert ANSI to Unicode, use A2W. These macros require the USES_CONVERSION macro to be placed at the top of a method; this will create some local variables that are used by the conversion macros. When the source and destination character sets are different and the destination type is not a BSTR, the macro allocates the destination string on the call stack (using the _alloca run-time function). It’s important to realize this especially when using these macros within a loop; otherwise, the stack may grow large and run out of stack space.

```
STDMETHODIMP CFoo::put_WindowTitle(BSTR bstrTitle)
{
    USES_CONVERSION;
    if (::SysStringLen(bstrTitle) == 0)
        return E_INVALIDARG;

    ::SetWindowText(m_hWnd, OLE2CT(bstrTitle));

    return S_OK;
}
```

To check if two CComBSTR strings are different, do not use the not equal (“!=”) operator. The “==” operator performs a case-sensitive comparison of the string contents; however, “!=” will compare pointer values and not the string contents, typically returning false.

Implementing noncreatable classes

Noncreatable classes are COM objects that cannot be created by *CoCreateInstance*. Instead, the object is created within a method call of a different object, and an interface pointer to the noncreatable class is returned. This type of object is found in abundance in the geodatabase model. For example, *FeatureClass* is noncreatable and can only be obtained by calling one of a number of methods; one example is the *IFeatureWorkspace::OpenFeatureClass* method.

One advantage of a noncreatable class is that it can be initialized with private data using method calls that are not exposed in a COM API. Below is a simplified example of returning a noncreatable object:

```
// Foo is a cocreatable object.
IFooPtr ipFoo;
HRESULT hr = ipFoo.CreateInstance(CLSID_Foo);

// Bar is a noncreatable object, cannot use ipBar.CreateInstance(CLSID_Bar).
IBarPtr ipBar;
// Use a method on Foo to create a new Bar object.
hr = ipFoo->CreateBar(&ipBar);
ipBar->DoSomething();
```

The steps required to change a cocreatable ATL class into a noncreatable class are shown below:

1. Add “noncreatable” to the .idl file’s coclass attributes.

```
[
    uuid(DCB87952-0716-4873-852B-F56AE8F9BC42),
    noncreatable
]
coclass Bar
{
    [default] interface IUnknown;
    interface IBar;
};
```

2. Change the class factory implementation to fail any cocreate instances of the noncreatable class. This happens via ATL’s object map in the main DLL module.

```
BEGIN_OBJECT_MAP(ObjectMap)
    OBJECT_ENTRY(CLSID_Foo, CFoo) // Creatable object
    OBJECT_ENTRY_NON_CREATEABLE(CLSID_Bar, CBar) // Noncreatable object
END_OBJECT_MAP()
```

3. Optionally, the registry entries can be removed. First, remove the registry script for the object from the resources (Bar.rgs in this example). Then change the class definition `DECLARE_REGISTRY_RESOURCEID(IDR_BAR)` to `DECLARE_NO_REGISTRY()`.
4. To create the noncreatable object inside a method, use the `CCOMObject` template to supply the implementation of `CreateInstance`.

```
// Get NonCreatable object Bar (implementing IBar) from COM object Foo
STDMETHODIMP CFoo::CreateBar(IBar **pVal)
{
```

```

if (pVal==0) return E_POINTER;
// Smart pointer to noncreatable object Bar
IBarPtr ipBar = 0;

// C++ Pointer to Bar, with ATL template to supply CreateInstance
implementation
CComObject<CBar>* pBar = 0;

HRESULT hr = CComObject<CBar>::CreateInstance(&pBar);
if (SUCCEEDED(hr))
{
    // Increment the ref count from 0 to 1 to protect the object
    // from being released in any initialization code.
    pBar->AddRef();

    // Call C++ methods (not exposed to COM) to initialize the Bar object.
    pBar->InitialiseBar(10);

    // QI to IBar and hold a smart pointer reference to the object Bar.
    hr = pBar->QueryInterface(IID_IBar, (void*)&ipBar);

    pBar->Release();
}

// Return IBar pointer to the caller.
*pVal = ipBar.Detach();

return S_OK;
}

```

ATL in Visual C++ .NET

Visual C++ version 6 is used for the majority of this help. However, with the release of Visual C++ .NET, there are enhancements and changes that are relevant to the ArcGIS ATL developer. Some of these are summarized below:

Attribute-based programming—This is a major change introduced in VC7. Attributes are inserted in the source code enclosed in square brackets—for example, [coclass]. Attributes are designed to simplify COM programming and .NET framework common language run-time development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files. There are attributes that aid in the creation of .idl files, interfaces, type libraries, and other COM elements. In the IDE, attributes are supported by the wizards and by the Properties window. The ATL wizards make extensive use of attributes to inject the ATL boilerplate code into the class. Consequently, typical COM coclass header files in VC7 contain much less ATL code than at VC6. As IDL is generated from attributes, there is typically no .idl file present in COM projects as before, and the .idl file is generated at compile time.

Build configurations—There are only two default build configurations in VC7; these are ANSI Debug- and Release-based builds. As ArcGIS is only available on

Unicode platforms, it is recommended to change these by modifying the project properties. The general project properties page has an option for “Character Set”. Change this from “Use Multi-Byte Character Set” to “Use Unicode Character Set”.

Character conversion macros—The character conversion macros (USES_CONVERSION, W2A, W2CT, and so forth) have improved alternative versions. These no longer allocate space on the stack, so they can be used in loops without running out of stack space. The USES_CONVERSION macro is also no longer required. These macros are now implemented as classes and begin with a “C”—for example, CW2A, CW2CT.

Safe array support—This is available with CComSafeArray and CComSafeArrayBound classes.

Module level global—The module level global CComModule _module has been split into a number of related classes, for example, CAtlComModule and CAtlWinModule. To retrieve the resource module instance, use the following code: `_AtlBaseModule.GetResourceInstanceO;`

String support—General variable length string support is now available through CString in ATL. This is defined in the header files atstr.h and cstring.h. If ATL is combined with MFC, this defaults to MFC’s CString implementation.

Filepath handling—A collection of related functions for processing the components of filepaths is available through the CPath class defined in atpath.h.

ATLServer—This is a new selection of ATL classes designed for writing Web applications, XML Web services, and other server applications.

#import issues—When using *#import*, a few modifications are required. For example, the *#import* of *esriSystem* requires an exclude or rename of *GetObject*, and the *#import* of *esriGeometry* requires an exclude or rename of *ISegment*.

ATL REFERENCES

The Microsoft Developer Network (MSDN) provides a wealth of documentation, articles, and samples that are installed with Visual Studio products. ATL reference documentation for Visual Studio version 6 is under:

MSDN Library - October 2001 / Visual Tools and Languages / Visual Studio 6.0 Documentation / Visual C++ Documentation / Reference / Active Template Library

Additional documentation is also available on the MSDN Web site at <http://www.msdn.microsoft.com>.

You may also find the following books to be useful:

Grimes, Richard. *ATL COM Programmer’s Reference*. Chicago: Wrox Press Inc., 1988.

Grimes, Richard. *Professional ATL COM Programming*. Chicago: Wrox Press Inc., 1988.

Grimes, Richard, Reilly Stockton, Alex Stockton, and Julian Templeman. *Beginning ATL 3 COM Programming*. Chicago: Wrox Press Inc. 1999.

King, Brad and George Shepherd. *Inside ATL*. Redmond, WA: Microsoft Press, 1999.

Rector, Brent, Chris Sells, and Jim Springfield. *ATL Internals*. Reading, MA: Addison–Wesley, 1999.

SMART TYPES

Smart types are objects that behave as types. They are C++ class implementations that encapsulate a data type, wrapping it with operators and functions that make working with the underlying type easier and less error prone. When these smart types encapsulate an interface pointer, they are referred to as *smart pointers*. Smart pointers work with the *IUnknown* interface to ensure that resource allocation and deallocation is correctly managed. They accomplish this by various functions, construct and destruct methods, and overloaded operators. There are numerous smart types available to the C++ programmer. The two main smart types covered here are Direct-To-COM and Active Template Library.

Smart types can make the task of working with COM interfaces and data types easier, since many of the API calls are moved into a class implementation; however, they must be used with caution and never without a clear understanding of how they are interacting with the encapsulated data type.

Direct-To-COM smart types

The smart type classes supplied with DTC are known as the Compiler COM Support Classes and consist of:

- `_com_error`—This class represents an exception condition in one of the COM support classes. This object encapsulates the HRESULT and the *IErrorInfo* COM exception objects.
- `_com_ptr_t`—This class encapsulates a COM interface pointer. See below for common uses.
- `_bstr_t`—This class encapsulates the *BSTR* data type. The functions and operators on this class are not as rich as the ATL *CComBSTR* smart type; hence, this is not normally used.
- `_variant_t`—This class encapsulates the *VARIANT* data type. The functions and operators on this class are not as rich as the ATL *CComVariant* smart type; hence, this is not normally used.

To define a smart pointer for an interface, you can use the macro `_COM_SMARTPTR_TYPEDEF` like this:

```
_COM_SMARTPTR_TYPEDEF(IFoo, __uuidof(IFoo));
```

The compiler expands this as follows:

```
typedef _com_ptr_t< _com_IID<IFoo, __uuidof(IFoo)> > IFooPtr;
```

Once declared, it is simply a matter of declaring a variable as the type of the interface and appending *Ptr* to the end of the interface. Below are some common uses of this smart pointer that you will see in the numerous C++ samples.

```
// Get a CLSID GUID constant.
extern "C" const GUID __declspec(selectany) CLSID_Foo = \
    {0x2f3b470c, 0xb01f, 0x11d3, {0x83, 0x8e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}};
```

```

// Declare Smart Pointers for IFoo, IBar, and IGak interfaces.
_COM_SMARTPTR_TYPEDEF(IFoo, __uuidof(IFoo));
_COM_SMARTPTR_TYPEDEF(IBar, __uuidof(IBar));
_COM_SMARTPTR_TYPEDEF(IGak, __uuidof(IGak));

STDMETHODIMP SomeClass::Do()
{
    // Create Instance of Foo class and QueryInterface (QI) for IFoo
    interface.
    IFooPtr ipFoo;
    HRESULT hr = ipFoo.CreateInstance(CLSID_Foo);
    if (FAILED(hr)) return hr;

    // Call method on IFoo to get IBar.
    IBarPtr ipBar;
    hr = ipFoo->get_Bar(&ipBar);
    if (FAILED(hr)) return hr;

    // QI IBar interface for IGak interface.
    IGakPtr ipGak(ipBar);

    // Call method on IGak.
    hr = ipGak->DoSomething();
    if (FAILED(hr)) return hr;

    // Explicitly call Release().
    ipGak = 0;
    ipBar = 0;

    // Let destructor call IFoo's Release.
    return S_OK;
}

```

One of the main advantages of using the DTC smart pointers is that they are automatically generated from the *#import* compiler statement for all interface and coclass definitions in a type library. For more details on this functionality, see the later section ‘Importing ArcGIS type libraries’.

It is possible to create an object implicitly in a DTC smart pointer’s constructor, for example:

```
IFooPtr ipFoo(CLSID_Foo)
```

However, this will raise a C++ exception if there is an error during object creation—for example, if the DLL file containing the object implementation was accidentally deleted. This exception will typically be unhandled and cause a crash. A more robust approach is to avoid exceptions in COM, call `CreateInstance` explicitly, and handle the failure code, for example:

```

IFooPtr ipFoo;
HRESULT hr = ipFoo.CreateInstance(CLSID_Foo);
if (FAILED(hr))
    return hr; // Return object creation failure code to caller.

```

Active Template Library smart types

ATL defines various smart types, as seen in the list below. You are free to combine both the ATL and DTC smart types in your code. However, it is typical to use the DTC for smart pointers, as they are easily generated by importing type libraries. For BSTR and VARIANT types, the ATL versions for CComBSTR, CComVariant are typically used.

ATL smart types include:

- *CComPtr*—encapsulates a COM interface pointer by wrapping the *AddRef* and *Release* methods of the *IUnknown* interface
- *CComQIPtr*—encapsulates a COM interface and supports all three methods of the *IUnknown* interface: *QueryInterface*, *AddRef*, and *Release*
- *CComBSTR*—encapsulates the *BSTR* data type
- *CComVariant*—encapsulates the *VARIANT* data type
- *CRegKey*—provides methods for manipulating Windows registry entries
- *CComDispatchDriver*—provides methods for getting and setting properties and calling methods through an object's *IDispatch* interface
- *CSecurityDescriptor*—provides methods for setting up and working with the Discretionary Access Control List (DACL)

This section examines the first four smart types and their uses. The example code below, written with ATL smart pointers, looks like the following:

```
// Get a CLSID GUID constant.
extern "C" const GUID __declspec(selectany) CLSID_Foo = \
    {0x2F3b470c, 0xb01f, 0x11d3, {0x83, 0x8e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}};

STDMETHODIMP SomeClass::Do ()
{
    // Create Instance of Foo class and QI for IFoo interface.
    CComPtr<IFoo> ipFoo;
    HRESULT hr = CoCreateInstance(CLSID_Foo, NULL, CLSCTX_INPROC_SERVER,
        IID_IFoo, (void **)&ipFoo);
    if (FAILED(hr)) return hr;

    // Call method on IFoo to get IBar.
    CComPtr<IBar> ipBar;
    HRESULT hr = ipFoo->get_Bar(&ipBar);
    if (FAILED(hr)) return hr;

    // IBar interface for IGak interface
    CComQIPtr<IGak> ipGak(ipBar);

    // Call method on IGak.
    hr = ipGak->DoSomething();
    if (FAILED(hr)) return hr;

    // Explicitly call Release().
    ipGak = 0;
}
```

The equality operator ("==") may have different implementations when used during smart pointer comparisons. The COM specification states object identification is performed by comparing the pointer values of IUnknown. The DTC smart pointers will perform necessary QI and comparison when using the "==" operator. However, the ATL smart pointers will not do this, so you must use the ATL IsEqualObject() method.

```

ipBar = 0;

// Let destructor call Foo's Release.
return S_OK;
}

```

The most common smart pointer seen in the Visual C++ samples is the DTC type. In the examples below, which illustrate the *BSTR* and *VARIANT* data types, the DTC pointers are used. When working with *CComBSTR*, use the text mapping `L""` to declare constant *OLECHAR* strings. *CComVariant* derives directly from the *VARIANT* data type, meaning that there is no overloading with its implementation, which in turn simplifies its use. It has a rich set of constructors and functions that make working with *VARIANT*s straightforward; there are even methods for reading and writing from streams. Be sure to call the *Clear* method before reusing the variable.

```

ipFoo->put_Name(CComBSTR(L"NewName"));
if FAILED(hr) return hr;

// Create a VT_I4 variant (signed long).
CComVariant vValue(12);

// Change its data type to a string.
hr = vValue.ChangeType(VT_BSTR);
if (FAILED(hr)) return hr;

```

Some method calls in IDL are marked as being optional and take a variant parameter. However in Visual C++, these parameters still have to be supplied. To signify that a parameter value is not supplied, a variant is passed specifying an error code or type `DISP_E_PARAMNOTFOUND`:

```

CComBSTR documentFilename(L"World.mxd");

CComVariant noPassword;
noPassword.vt = VT_ERROR;
noPassword.scode = DISP_E_PARAMNOTFOUND;
HRESULT hr = ipMapControl->LoadMxFile(documentFilename, noPassword);

```

When working with *CComBSTR* and *CComVariant*, the *Detach()* function releases the underlying data type from the smart type so it can be used when passing a result as an [out] parameter of a method. The use of the *Detach* method with *CComBSTR* is shown below:

```

STDMETHODIMP CFoo::get_Name(BSTR* name)
{
    if (name==0) return E_POINTER;
    CComBSTR bsName(L"FooBar");
    *name = bsName.Detach();
}

```

CComVariant(VARIANT_TRUE) will create a short integer variant (type `VT_I2`) and not a Boolean variant (type `VT_BOOL`) as expected. You can use *CComVariant(true)* to create a Boolean variant.

CComVariant(myVar(ipSmartPointer)) will result in a variant type of Boolean (`VT_BOOL`) and not a variant with an object reference (`VT_UNKNOWN`) as expected. It is better to pass unambiguous types to constructors, that is, types that are not themselves smart types with overloaded cast operators.

```
// Perform QI if IUnknown.
IUnknownPtr ipUnk = ipSmartPointer;
// Ensure IUnknown* constructor of CComVariant is used.
CComVariant myVar2(ipUnk.GetInterfacePtr());
```

A common practice with smart pointers is to use *Detach()* to return an object from a method call. When returning an interface pointer, the COM standard is to increment reference count of the [out] parameter inside the method implementation. It is the caller's responsibility to call *Release* when the pointer is no longer required. Consequently, care must be taken to avoid calling *Detach()* directly on a member variable. A typical pattern is shown below:

```
STDMETHODIMP CFoo::get_Bar(IBar **pVal)
{
    if (pVal==0) return E_POINTER;

    // Constructing a local smart pointer using another smart pointer
    // results in an AddRef (if pointer is not 0).
    IBarPtr ipBar(m_ipBar);

    // Detach will clear the local smart pointer, and the
    // interface is written into the output parameter.
    *pVal = ipBar.Detach();

    // This can be combined into one line
    // *pVal = IBarPtr(m_ipBar).Detach();

    return S_OK;
}
```

The above pattern has the same result as the following code; note that a conditional test for a zero pointer is required before *AddRef* can be called. Calling *AddRef* (or any method) on a zero pointer will result in an access violation exception and typically crash the application:

```
STDMETHODIMP CFoo::get_Bar(IBar **pVal)
{
    if (pVal==0) return E_POINTER;

    // Copy the interface pointer (no AddRef) into the output parameter.
    *pVal = m_ipBar;

    // Make sure interface pointer is nonzero before calling AddRef.
    if (*pVal)
        *pVal->AddRef();

    return S_OK;
}
```

When using a smart pointer to receive an object from an [out] parameter on a method, use the smart pointer “&” dereference operator. This will cause the previous interface pointer in the smart pointer to be released. The smart pointer is then populated with the new [out] value. The implementation of the method will

have already incremented the object reference count. This will be released when the smart pointer goes out of scope:

```
{
    IFooPtr ipFoo1, ipFoo2;
    ipFoo1.CreateInstance(CLSID_Foo);
    ipFoo2.CreateInstance(CLSID_Foo);

    // Initialize ipBar Smart pointer from Foo1.
    IBarPtr ipBar;
    ipFoo1->get_Bar(&ipBar);

    // The "&" dereference will call Release on ipBar.
    // ipBar is then repopulated with a new instance of IBar.
    ipFoo2->get_Bar(&ipBar);
}
// ipBar goes out of scope, and the smart pointer destructor calls
Release.
```

Naming conventions

Type names

All type names (*class*, *struct*, *enum*, and *typedef*) begin with an uppercase letter and use mixed case for the rest of the name:

```
class Foo : public CObject { . . . };
struct Bar { . . . };
enum ShapeType { . . . };
typedef int* FooInt;
```

Typedefs for function pointers (callbacks) append Proc to the end of their names.

```
typedef void (*FooProgressProc)(int step);
```

Enumeration values all begin with a lowercase string that identifies the project; in the case of ArcObjects this is esri, and each string occurs on a separate line:

```
typedef enum esriQuuxness
{
    esriQLow,
    esriQMedium,
    esriQHigh
} esriQuuxness;
```

Function names

Name functions using the following conventions:

- For simple accessor and mutator functions, use Get<Property> and Set<Property>:


```
int GetSize();
void SetSize(int size);
```
- If the client is providing storage for the result, use Query<Property>:


```
void QuerySize(int& size);
```

Here are some suggestions for a naming convention. These help identify the variables' usage and type and so reduce coding errors. This is an abridged Hungarian notation:

[<scope>_]<type><name>

Prefix	Variable scope
m	Instance class members
c	Static class member (including constants)
g	Globally static variable
<empty>	local variable or struct or public class member

<type>

Prefix	Data Type
b	Boolean
by	byte or unsigned char
cx / cy	short used as size
d	double
dw	DWORD, double word or unsigned long
f	float
fn	function
h	handle
i	int (integer)
ip	smart pointer
l	long
p	a pointer
s	string
sz	ASCIIZ null-terminated string
w	WORD unsigned int
x, y	short used as coordinates

<name> describes how the variable is used or what it contains. The <scope> and <type> portions should always be lowercase, and the <name> should use mixed case:

Variable Name	Description
m_hWnd	a handle to a HWND
ipEnvelope	a smart pointer to a COM interface
m_pUnkOuter	a pointer to an object
c_isLoaded	a static class member
g_pWindowList	a global pointer to an object

- For state functions, use Set<State> and Is<State> or Can<State>:

```
bool IsFileDirty();
void SetFileDirty(bool dirty);
bool CanConnect();
```
- Where the semantics of an operation are obvious from the types of arguments, leave type names out of the function names.

Instead of:

```
AddDatabase(Database& db);
```

consider using:

```
Add(Database& db);
```

Instead of:

```
ConvertFoo2Bar(Foo* foo, Bar* bar);
```

consider using:

```
Convert(Foo* foo, Bar* bar)
```

- If a client relinquishes ownership of some data to an object, use Give<Property>. If an object relinquishes ownership of some data to a client, use Take<Property>:

```
void GiveGraphic(Graphic* graphic);
Graphic* TakeGraphic(int itemNum);
```
- Use function overloading when a particular operation works with different argument types:

```
void Append(const CString& text);
void Append(int number);
```

Argument names

Use descriptive argument names in function declarations. The argument name should clearly indicate what purpose the argument serves:

```
bool Send(int messageID, const char* address, const char* message);
```

DEBUGGING TIPS IN DEVELOPER STUDIO

Visual C++ comes with a feature-rich debugger. These tips will help you get the most from your debugging session.

Backing up after failure

When a function call has failed and you'd like to know why (by stepping into it), you don't have to restart the application. Use the Set Next Statement command to reposition the program cursor back to the statement that failed (right-click on the statement to bring up the debugging context menu). Then step into the function.

Edit and Continue

Visual Studio 6 allows changes to source code to be made during a debugging session. The changes can be recompiled and incorporated into the executing code without stopping the debugger. There are some limitations to the type of changes that can be made; in this case, the debug session must be restarted. This feature is enabled by default; the settings are available in the Settings command of the

project menu. Click the C/C++ tab, then choose General from the Category dropdown list. In the Debug info dropdown list, click Program Database for Edit and Continue.

Unicode string display

To set your debugger options to display Unicode strings, click the Tools menu, click Options, click Debug, then check the Display Unicode Strings check box.

Variable value display

Pause the cursor over a variable name in the source code to see its current value. If it is a structure, click it and bring up the QuickWatch dialog box (the Eyeglasses icon or Shift+F9) or drag and drop it into the Watch window.

Undocking windows

If the Output window (or any docked window, for that matter) seems too small to you, try undocking it to make it a real window by right-clicking it and toggling the Docking View item.

Conditional break points

Use conditional break points when you need to stop at a break point only once some condition is reached—for instance, when a for loop reaches a particular counter value. To do so, set the break point normally, then bring up the Breakpoints window (Ctrl+B or Alt+F9). Select the specific break point you just set and click the Condition button to display a dialog box in which you specify the break point condition.

Preloading DLLs

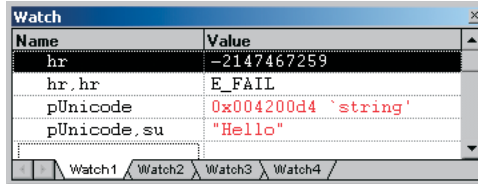
You can preload DLLs that you want to debug before executing the program. This allows you to set break points up front rather than wait until the DLL has been loaded during program execution. To do this, click Project, click Settings, click Debug, click Category, then click Additional DLLs. Then, click in the list area to add any DLLs you want to preload.

Changing display formats

You can change the display format of variables in the QuickWatch dialog box or in the Watch window using the formatting symbols in the following table.

Symbol	Format	Value	Displays
d, i	signed decimal integer	0xF00F065	-268373915
u	unsigned decimal integer	0x0065	101
o	unsigned octal integer	0xF065	0170145
x, X	hexadecimal integer	61541	0x000F065
l, h	long or short prefix for d, l, u, o, x, X	00406042, hx	0x0C22
f	signed floating-point	3./2.	1.500000
e	signed scientific notation	3./2.	1.500000e+00
g	e or f, whichever is shorter	3./2.	1.5
c	single character	0x0065	'e'
s	string	0x0012FDE8	"Hello"
su	Unicode string		"Hello"
hr	string	0	S_OK

To use a formatting symbol, type the variable name followed by a comma and the appropriate symbol. For example, if `var` has a value of `0x0065`, and you want to see the value in character form, type “`var,c`” in the Name column on the tab of the Watch window. When you press Enter, the character format value appears: `var,c = 'e'`. Likewise, assuming that `hr` is a variable holding `HRESULT`s, view a human-readable form of the `HRESULT` by typing “`hr,hr`” in the Name column.



You can use the formatting symbols shown in the following table to format the contents of memory locations.

Symbol	Format	Value
<code>ma</code>	64 ASCII characters	<code>0x0012fac .4...0...7.0W&.. ...1W&0..W..1 ...*.1JO&.1.2 ...1...0y...1</code>
<code>m</code>	16 bytes in hex, followed by 16 ASCII characters	<code>0x0012fac B3 34 CB 00 84 30 94 80 FF 22 8A 30 57 26 00 00 .4...0...".0W&..</code>
<code>mb</code>	16 bytes in hex, followed by 16 ASCII characters	<code>0x0012fac B3 34 CB 00 84 30 94 80 FF 22 8A 30 57 26 00 00 .4...0...".0W&..</code>
<code>mw</code>	8 words	<code>0x0012fac 34B3 00CB 3084 8094 22FF 308A 2657 0000</code>
<code>md</code>	4 double-words	<code>0x0012fac 00CB34B3 80943084 308A22FF 00002657</code>
<code>mu</code>	2-byte characters (Unicode)	<code>0x0012fc60 8478 77f4 ffff ffff 0000 0000 0000 0000</code>

With the memory location formatting symbols, you can type any value or expression that evaluates a location. To display the value of a character array as a string, precede the array name with an ampersand, `&yourname`. A formatting character can also follow an expression:

- `rep+1,x`
- `alps[0],mb`
- `xloc,g`
- `count,d`

To watch the value at an address or the value to which a register points, use the `BY`, `WO`, or `DW` operators:

- `BY` returns the contents of the byte pointed at.
- `WO` returns the contents of the word pointed at.
- `DW` returns the contents of the doubleword pointed at.

Follow the operator with a variable, register, or constant. If the `BY`, `WO`, or `DW` operator is followed by a variable, then the environment watches the byte, word, or doubleword at the address contained in the variable.

You can also use the context operator { } to display the contents of any location.

To display a Unicode string in the Watch window or the QuickWatch dialog box, use the su format specifier. To display data bytes with Unicode characters in the Watch window or the QuickWatch dialog box, use the mu format specifier.

Keyboard shortcuts

There are numerous keyboard shortcuts that make working with the Visual Studio Editor faster. Some of the more useful keyboard shortcuts follow.

The text editor uses many of the standard shortcut keys used by Windows applications, such as Word. Some specific source code editing shortcuts are listed below.

Shortcut	Action
Alt+FB	Correctly indent selected code based on surrounding lines.
Ctrl+]	Find the matching brace.
Ctrl+J	Display list of members.
Ctrl+Spacebar	Complete the word, once the number of letters entered allows the editor to recognize it. Use full when completing function and variable names.
Tab	Indents selection one tab stop to the right.
Shift+Tab	Indents selection one tab to the left.

Below is a table of common keyboard shortcuts used in the debugger.

Shortcut	Action
F9	Add or remove breakpoint from current line.
Ctrl+Shift+F9	Remove all breakpoints.
Ctrl+F9	Disable breakpoints.
Ctrl+Alt+A	Display auto window and move cursor into it.
Ctrl+Alt+C	Display call stack window and move cursor into it.
Ctrl+Alt+L	Display locals window and move cursor into it.
Ctrl+Alt+A	Display auto window and move cursor into it.
Shift+F5	End debugging session.
F11	Execute code one statement at a time, stepping into functions.
F10	Execute code one statement at a time, stepping over functions.
Ctrl+Shift+F5	Restart a debugging session.
Ctrl+F10	Resume execution from current statement to selected statement.
F5	Run the application.
Ctrl+F5	Run the application without the debugger.
Ctrl+Shift+F10	Set the next statement.
Ctrl+Break	Stop execution.

Loading the following shortcuts can greatly increase your productivity with the Visual Studio development environment.

Shortcut	Action
ESC	Close a menu or dialog box, cancel an operation in progress, or place focus in the current document window.
CTRL+SHIFT+N	Create a new file.
CTRL+N	Create a new project.
CTRL+F6 or CTRL+TAB	Cycle through the MDI child windows one window at a time.
CTRL+ALT+A	Display the auto window and move the cursor into it.
CTRL+ALT+C	Display the call stack window and move the cursor into it.
CTRL+ALT+T	Display the document outline window and move the cursor into it.
CTRL+H	Display the find window.
CTRL+F	Display the find window. If there is no current Find criteria, put the word under your cursor in the find box.
CTRL+ALT+I	Display the immediate window and move the cursor into it. Not available if you are in the text editor window.
CTRL+ALT+L	Display the locals window and move the cursor into it.
CTRL+ALT+O	Display the output window and move the cursor into it.
CTRL+ALT+J	Display the project explorer window and move the cursor into it.
CTRL+ALT+P	Display the properties window and move the cursor into it.
CTRL+SHIFT+O	Open a file.
CTRL+O	Open a project.
CTRL+P	Print all or part of the document.
CTRL+SHIFT+S	Save all of the files, projects, or documents.
CTRL+S	Select all.
CTRL+A	Save the current document or selected item or items.

Navigating through online help topics

Right-click a blank area of a toolbar to display a list of all the available toolbars. The Infoviewer toolbar contains up and down arrows that allow you to cycle through help topics in the order in which they appear in the table of contents. The left and right arrows cycle through help topics in the order that you visited them.

IMPORTING ArcGIS TYPE LIBRARIES

To reference ArcGIS interfaces, types, and objects, you will need to import the definitions into Visual C++ types. The *#import* command automates the creation of the necessary files required by the compiler. The *#import* was developed to support Direct-To-Com. When importing ArcGIS library types, there are a number of parameters that must be passed.

```
#pragma warning(push)
#pragma warning(disable : 4192)          /* Ignore warnings for types that are
                                         duplicated in win32 header files.
*/
#pragma warning(disable : 4146)          /* Ignore warnings for use of minus on
                                         unsigned types. */

#import "\\Program Files\\ArcGIS\\com\\esriSystem.olb"
                                         /* Type library to generate C++ wrappers.*/ \
raw_interfaces_only,                    /* Don't add raw_ to method names.    */ \
raw_native_types,                       /* Don't map to DTC smart types.     */ \
no_namespace,                           /* Don't wrap with C++ name space.   */ \
named_guids,                             /* Named guids and declspecs.        */ \
exclude("OLE_COLOR", "OLE_HANDLE", "VARTYPE")
                                         /* Exclude conflicting types.        */ \

#pragma warning(pop)
```

The main use of `#import` is to create C++ code for interface definitions and GUID constants (LIBID, CLSID, and IID) and to define smart pointers. The `exclude` (“OLE_COLOR”, “OLE_HANDLE”, “VARTYPE”) is required because Windows defines these to be unsigned longs, which conflicts with the ArcGIS definition of `long`—this was required to support Visual Basic as a client of ArcObjects, since Visual Basic has no support for unsigned types. There are no issues with excluding these.

You can view the code generated by `#import` in the type library header (.tlh) files, which are similar in format to a .h file. You may also find a type library implementation (.tli) file, which corresponds to a .cpp file. These files can be large but are only regenerated when the type libraries change.

There are many type libraries at ArcGIS 9 for different functional areas. You can start by importing those that contain the definitions that you require. However, `#import` does not automatically include all other definitions that the imported type library requires. For example, when importing the type library `esriGeometry`, it will contain references to types that are defined in `esriSystem`, so `esriSystem` must be imported before `esriGeometry`.

A complete list of library dependencies can be found in the Overview topic for each library.

Choosing the minimum set of type libraries helps reduce compilation time, although this is not always significant. Here are some steps to help determine the minimum number of type libraries required:

1. Do a compilation and look at the “missing type definition” errors generated from code (for example, `ICommand` not found).
2. Place a `#import` statement for the library you need a reference for into your `stdafx.h` file. Use the `LibraryLocator` utility or component help to assist in this task.
3. Compile the project a second time.
4. The compiler will issue errors for types it cannot resolve in the imported type libraries; these are typically type definitions, such as `WKSPoint` or interfaces that are inherited into other interfaces. For example, if working with geometry objects, such as points, start by importing `esriGeometry`. The compiler will issue various error messages, such as:

```
c:\temp\sample\debug\esrigeometry.tlh(869) : error C2061: syntax error :
identifier WKSPoint
```

Looking up the definition of `WKSPoint`, you see it is defined in `esriSystem`. Therefore, importing `esriSystem` before `esriGeometry` will resolve all these issues.

Below is a typical list of imports for working with the ActiveX controls.

```
#pragma warning(push)
#pragma warning(disable : 4192) /* Ignore warnings for types that are
                                duplicated in win32 header files. */
#pragma warning(disable : 4146) /* Ignore warnings for use of minus on
                                unsigned types. */
```

```

    #import "\\Program Files\\ArcGIS\\com\\esriSystem.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids, exclude("OLE_COLOR",
    "OLE_HANDLE", "VARTYPE")
    #import "\\Program Files\\ArcGIS\\com\\esriSystemUI.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriGeometry.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriDisplay.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriOutput.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriGeoDatabase.olb"
    raw_interfaces_only, raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriCarto.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids

    // Some of the Engine controls
    #import "\\Program Files\\ArcGIS\\bin\\TOCControl.ocx" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\bin\\ToolbarControl.ocx"
    raw_interfaces_only, raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\bin\\MapControl.ocx" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\bin\\PageLayoutControl.ocx"
    raw_interfaces_only, raw_native_types, no_namespace, named_guids

    // additionally for 3D controls
    #import "\\Program Files\\ArcGIS\\com\\esri3DAnalyst.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\com\\esriGlobeCore.olb" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\bin\\SceneControl.ocx" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    #import "\\Program Files\\ArcGIS\\bin\\GlobeControl.ocx" raw_interfaces_only,
    raw_native_types, no_namespace, named_guids
    
```

A similar issue arises when writing IDL that contains definitions from other type libraries. In this situation, use `importlib` just after the library definition. For example, writing an external command for ArcMap would require you to create a COM object implementing *ICommand*. This definition is in *esriSystemUI* and is imported into the IDL as follows:

```

library WALKTHROUGH1CPPLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    importlib("\\Program Files\\ArcGIS\\com\\esriSystemUI.olb");

    coclass ZoomIn
    {
        [default] interface IUnknown;
        interface ICommand;
    }
};
    
```

For a general discussion of ATL, see the earlier section 'ATL in brief'.

ATL AND THE ACTIVE X CONTROLS

This section covers how to use ATL to add controls to a dialog box. Although ATL is focused on providing COM support, it also supplies some useful Windows programming wrapper classes. One of the most useful is CWindow, a wrapper around a window handle (HWND). The method names on CWindow correspond to the Win32 API functions. For example:

```
HWND buttonHwnd = GetDlgItem( IDC_BUTTON1 ); // Get window handle of button.
CWindow myButtonWindow( buttonHwnd ); // Attach window handle to CWindow class.
myButtonWindow.SetWindowText( T("Button Title") ); // Win32 function to
                                                    change button caption.
```

CWindow is a generic wrapper for all window handles, so for specific Windows messages to common windows controls, such as buttons, tree views, or edit boxes, one approach is to send window messages directly to the window, for example:

```
// Set button to be checked (pushed in or checkmarked, depending on button
style)
myButtonWindow.SendMessage( BM_SETCHECK, BST_CHECKED );
```

However, there are some wrapper classes for these standard window common controls in a header file *atcontrols.h*. This is available as part of an ATL sample ATLCON supplied in MSDN. See the article "HOWTO: Using Class Wrappers to Access Windows Common Controls in ATL". This header file is an early version of Windows Template Libraries (WTL), available for download from Microsoft.

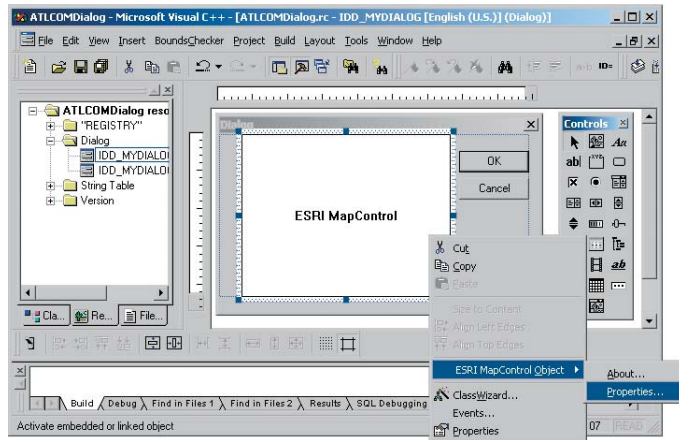
The Visual Studio Resource Editor can be used to design and position common windows and ActiveX controls on a dialog box. To create and manipulate the dialog box, a C++ class is typically created that inherits from *CAXDialogImpl*. This class provides the plumbing to create and manage the ActiveX control on a window. The ATL wizard can be used to supply the majority of the boilerplate code. The steps to create a dialog box and add an ActiveX control in an ATL project are discussed below.

1. Click the menu command Insert/New ATL Object.
2. Click the Miscellaneous category, then click the Dialog object.
3. A dialog box resource and a class inheriting from *CAXDialogImpl* will be added to your project.
4. Right-click the dialog box in resource view and click Insert ActiveX Control. This will display a list of available ActiveX controls.
5. Double-click a control in the list to add that control to the dialog box.

- Right-click the control and click Properties to set the control's design-time properties.

Make sure dialog boxes that host ActiveX controls inherit from `CXDialogImpl` and not `CDialogImpl`. If this mistake is made, the `DoModal` method of the dialog box simply exits with no obvious cause.

Make sure applications that use common window controls, such as treeview, correctly call `InitCommonControlsEx` to load the window class. Otherwise, the class will not function correctly.



Accessing a control on a dialog box through a COM interface

To retrieve a handle to the control that is hosted on a form, use the `GetDlgControl` ATL method that is inherited from `CXDialogImpl` to take a resource ID and return the underlying control pointer:

Make sure applications using COM objects call `CoInitialize`. This initializes COM in the application. Without this call, any `CoCreate` calls will fail.

```
ITOCControlPtr ipTOCControl;
GetDlgControl(IDC_TOCCONTROL1, IID_ITOCControl, (void**) &ipTOCControl);
ipTOCControl->AboutBox();
```

Listening to events from a control

The simplest way to add events is to use the class wizard. Simply right-click the control and choose Events. Next, click the resource ID of the control, then click the event (for example, `OnMouseDown`). Next click Add Handler. Finally, ensure the dialog box begins listening to events by adding `AtlAdviseSinkMap(this,TRUE)` to the `OnInitDialog`. To finish listening to events, add a message handler for `OnDestroy` and add a call to `AtlAdviseSinkMap(this, FALSE)`.

For a detailed discussion on handling events in ATL, see the later section 'Handling COM events in ATL'.

Creating a control at run time

The `CXWindow` class provides a mechanism to create and host ActiveX controls in a similar manner to any other window class. This may be desirable if the parent window of the control is also created at run time.

```
AtlAxWinInit();
CXWindow wnd;
//m_hwnd is the parent window handle.
//rect is the size of ActiveX control in client coordinates.
//IDC_MYCTL is a unique ID to identify the controls window.
RECT rect = {10,10,400,300};
wnd.Create(m_hwnd, rect, _T("esriReaderControl.ReaderControl"),
WS_CHILD|WS_VISIBLE, 0, IDC_MYCTL);
```


Setting the buddy control property

The *ToolBarControl* and *TOCCControl* need to be associated with a “buddy” control on the dialog box. This is typically performed in the *OnInitDialog* windows message handler of a dialog box.

```
LRESULT CEngineControlsDlg::OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM
lParam, BOOL& bHandled)
{
    // Get the Control's interfaces into class member variables.
    GetDlgControl(IDC_TOOLBARCONTROL, IID_IToolBarControl, (void **)
&m_ipToolBarControl);
    GetDlgControl(IDC_TOCCONTR0L, IID_ITOCCControl, (void **) &m_ipTOCCControl);
    GetDlgControl(IDC_PAGELAYOUTCONTROL, IID_IPageLayoutControl, (void **)
&m_ipPageLayoutControl);

    // Connect to the controls.
    AtlAdviseSinkMap(this, TRUE);

    // Set buddy controls.
    m_ipTOCCControl->SetBuddyControl(m_ipPageLayoutControl);
    m_ipToolBarControl->SetBuddyControl(m_ipPageLayoutControl);

    return TRUE;
}
```

Known limitations of Visual Studio C++ Resource Editor and ArcGIS ActiveX controls

Disabled buddy property on property page

In Visual Studio C++ you cannot set the ‘Buddy’ property of the *TOCCControl* and the *ToolBarControl* through the General property page. Visual C++ does not support controls finding other controls at design time. However, this step can be performed in code in the *OnInitDialog* method.

ToolBarControl not resized to the height of one button

In other environments (Visual Basic 6, .NET) the *ToolBarControl* will automatically resize to be one button high. However, in Visual Studio C++ 6 it can be any size. In MFC and ATL the ActiveX host classes do not allow controls to determine their own size.

Design-time property pages disappearing when displaying context-sensitive help

When viewing the controls property page at design time, right-clicking and clicking “What’s This?” will cause the help tip to display; however, the property pages will then close. This is a limitation of the Visual Studio floating windows combined with the floating tip window from HTML help. Clicking the Help button provides the same text for the whole property page.

MFC AND THE ACTIVE X CONTROLS

There are many choices for how to work with ArcGIS ActiveX Controls in Visual

C++, the first of which is what framework to use to host the controls (for example, ATL or MFC). A second decision is where the control will be hosted (Dialog, MDI app, and so forth). This section discusses MFC and hosting the control on a dialog box.

Creating an MFC dialog box-based application

If you do not have a dialog box in your application or component, here are the steps to create an MFC dialog box application.

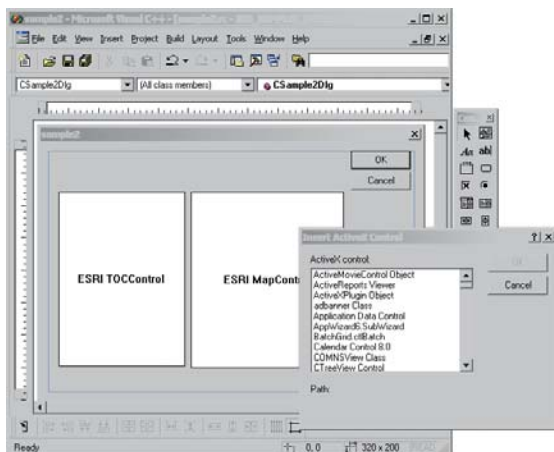
1. Launch Visual Studio C++ 6 and click New.
2. Click the Projects tab and choose MFC AppWizard (exe). Enter the project name and location and click OK.
3. For Step 1 of the wizard: From the radio buttons, change the application type to Dialog Based. Click Next.
4. For Step 2 of the wizard: The default project features are fine, although you can uncheck AboutBox to simplify the application. Ensure that the option to support ActiveX Controls is checked. Click Next.
5. For Step 3 of the wizard: The default settings on this page are fine. The MFC DLL is shared. Click Next.
6. For Step 4 of the wizard: This shows you what the wizard will generate. Click Finish.

You should now have a simple dialog box-based application. In the resource view, you will see “TODO: Place Dialog Controls Here”. You can place buttons, list boxes, and so forth in this dialog box. The dialog box can also host ActiveX controls; there are two approaches to doing this, as discussed below. You can also compile and run this application.

Inserting ActiveX controls on a dialog box in Visual Studio C++ design time. The TOCControl and MapControl have been added to the dialog box. The ToolbarControl is next.

Hosting controls on an MFC dialog box and accessing them using IDispatch

1. Right-click the MFC dialog box and click Insert ActiveX control.
2. Double-click a control from the list box. The control appears on the dialog box with a default size.
3. Size and position the control as required.
4. Repeat Steps 1 through 3 for each control.
5. You can right-click the control and choose Properties to set the control’s design-time properties.
6. To access the control in code, you will need ArcGIS interface definitions for IMapControl, for example. To do this, use the *#import* command in your stdafx.h file. See the section ‘Importing ArcGIS type libraries’ on how to do this.



- MFC provides control hosting on a dialog box; this will translate Windows messages, such as WM_SIZE, into appropriate control method calls. However, to be able to make calls on a control, there are a few steps you must perform to go from a resource ID to a controls interface. The following code illustrates setting the TOCControl's Buddy to be the MapControl:

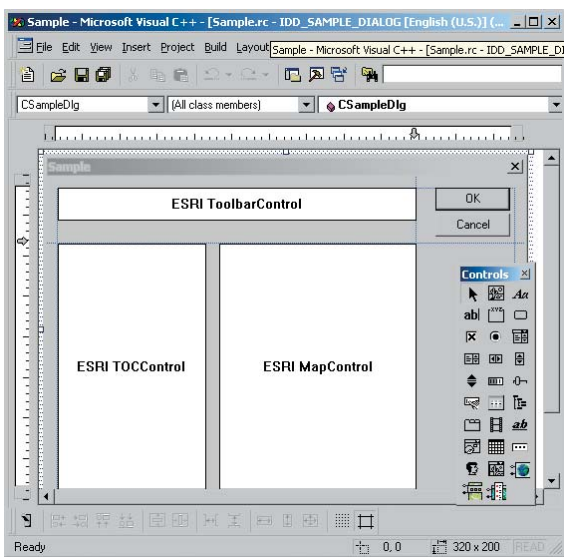
```
// Code to set the Buddy property of the TOCControl to be the MapControl
// Get a pointer to the PageLayoutControl and TOCControl.
IPageLayoutControlPtr ipPageLayoutControl;
GetDlgControl(IDC_PAGELAYOUTCONTROL1, IID_IPageLayoutControl, (void**)
&ipPageLayoutControl);

ITOCControlPtr ipTOCControl;
GetDlgControl(IDC_TOCCONTROL1, IID_ITOCControl, (void**) &ipTOCControl);

// Get the IDispatch of the PageLayoutControl.
IDispatchPtr ipBuddyDisp = ipPageLayoutControl;

// Set the TOCControl's Buddy to the map control.
ipTOCControl->putref_Buddy(ipBuddyDisp);
```

- To catch events from the controls, double-click the control on the form and supply the name of a method to be called. By default, the wizard will add an extra word "On" to the beginning of the event handler. Remove this to avoid the event handler's name from becoming "OnOnMouseDownMapcontrol". The wizard will then automatically generate the necessary MFC sink map macros to listen to events.



The design environment showing the TOCControl, MapControl, and ToolbarControl has been added to the Controls toolbar and to the dialog box.

Adding controls to an MFC dialog box using IDispatch wrappers

As all ActiveX controls support *IDispatch*, this is the typical approach to add an ActiveX control to an MFC project:

- Click Project, click Add, then click Components and Controls.
- Click Registered ActiveX Controls.
- Double-click to select a control (for example, *ESRI TOCControl*), then click OK to insert a component. Click OK to generate wrappers. This will add an icon for the control to the Controls toolbar in Visual Studio.
- Additional source files are added to your project (for example, *toccontrol.cpp* and *toccontrol.h*). These files contain a wrapper class (for example, *CTOCControl*) to provide methods and properties to access the control. This class will invoke the control through the *IDispatch* calling mechanism. Note that *IDispatch* does incur some performance overhead to package parameters when making method and property calls. The wrapper class inherits from a MFC *CWnd* class that hosts an ActiveX control.
- Repeat Steps 1 through 4 to add each control to the project's Controls toolbar.

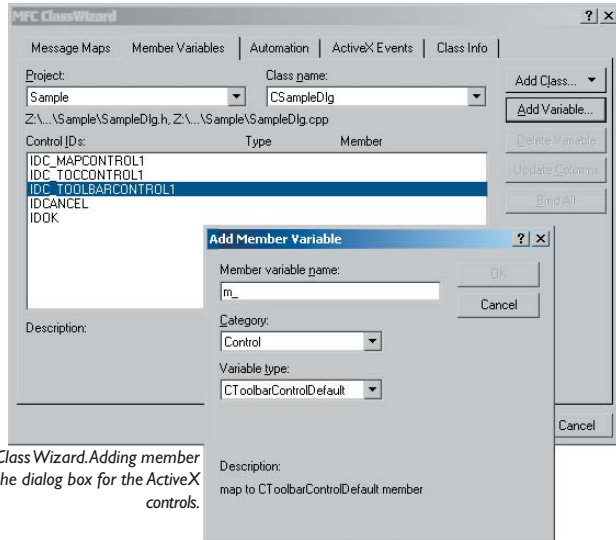
6. Select a control from the Controls toolbar and drag it onto the dialog box.
7. Right-click the control and click Properties. This will allow design-time properties to be set on the control. NOTE: in Visual Studio C++, you cannot set the Buddy property of the *TOCCControl* and the *ToolBarControl*.

This environment does not support controls finding other controls at design time. However, this step can be performed in code using the *OnInitDialog* method.

```
// Note no addref performed with GetControlUnknown, so no need to release
// this pointer.
LPUNKNOWN pUnk = m_mapcontrol.GetControlUnknown();
LPDISPATCH pDisp =
0;pUnk->QueryInterface(IID_IDispatch, (void **) &pDisp);

// Set TOCControl's buddy to be MapControl.
m_toccontrol.SetRefBuddy(pDisp);
pDisp->Release();
```

8. Right-click the control and choose Class Wizard to launch the class wizard. Click the Member Variables tab and click the resource ID corresponding to the control to give the control member variable name. The dialog box class member variable can now be used to invoke methods and properties on the control.



Visual Studio C++ Class Wizard. Adding member variables to the dialog box for the ActiveX controls.

Do not use the method *GetDispatch* (inherited from MFC's *CCmdTarget*) on the wrapper classes; it is intended for objects implementing *IDispatch* and not the wrapper classes that are calling *IDispatch*. Instead, to get a control's *IDispatch* use *m_mapcontrol.GetControlUnknown()* and *QueryInterface* to *IDispatch*. See the above example of setting the *Buddy* property.

9. To catch control events, click the Message Maps tab of the class wizard and choose the resource ID of the control. In the list of messages click the event to catch—for example, *OnBeginLabelEdit*. Double-click this event and a handler for it will be added to your dialog box class. By default, the wizard will add an extra word “On” to the beginning of the event handler. Remove this to avoid the event handler name becoming *OnOnBeginLabelEditTocontrol1*.

HANDLING COM EVENTS IN ATL

Here is a summary of terminology used here when discussing COM events in Visual C++ and ATL.

Inbound interface—This is the normal case where a COM object implements a predefined interface.

Outbound interface—This is an interface of methods that a COM object will fire at various times. For example, the *MapCoClass* will fire an event on the *LActiveViewEvents* in response to changes in the map.

Event source—The source COM object will fire events to an outbound interface when certain actions occur. For example, the *MapCoClass* is a source of *LActiveViewEvents* and will fire the *LActiveViewEvents::ItemAdded* event when a new layer is added to the map. The source object can have any number of clients, or *event sink objects*, listening to events. Also, a source object may have more than one outbound interface; for example, the *MapCoClass* also fires events on an *IMapEvents* interface. An event source will typically declare its outbound interfaces in IDL with the *[source]* tag.

Event sink—A COM object that listens to events is said to be a “sink” for events. The sink object implements the outbound interface; this is not always advertised in the type libraries because the sink may listen to events internally. An event sink typically uses the connection point mechanism to register its interest in the events of a source object.

Connection point—COM objects that are the source of events typically use the connection point mechanism to allow sinks to hook up to a source. The connection point interfaces are the standard COM interfaces *IConnectionPointContainer* and *IConnectionPoint*.

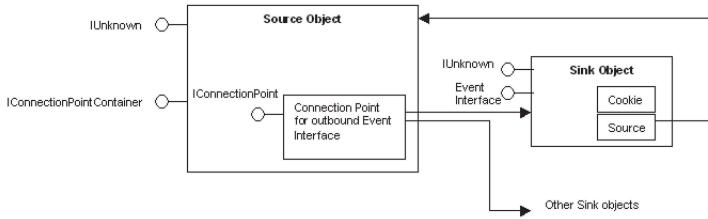
Fire event—When a source object needs to inform all the sinks of a particular action, the source is said to “fire” an event. This results in the source iterating all the sinks and making the same method call on each. For example, when a layer is added to a map, the *Map* coclass is said to fire the *ItemAdded* event. So all the objects listening to the *Map*’s outbound *LActiveViewEvents* interface will be called on their implementation of the *ItemAdded* method.

Advise and unadvise events—To begin receiving events a sink object is said to “advise” a source object that it needs to receive events. When events are no longer required, the sink will “unadvise” the source.

The ConnectionPoint mechanism

The source object implements the *IConnectionPointContainer* interface to allow sinks to query a source for a specific outbound interface. The following steps are performed to begin listening to an event. ATL implements this with the *AtlAdvise* method.

1. The sink will QI the source object’s *IConnectionPointContainer* and call *FindConnectionPoint* to supply an interface ID for outbound interfaces. To be able to receive events, the sink object must implement this interface.
2. The source may implement many outbound interfaces and will return a pointer to a specific connection point object implementing *IConnectionPoint* to represent one outbound interface.



Connection point mechanism for hooking source to sink objects

3. The sink calls *IConnectionPoint::Advise*, passing a pointer to its own *IUnknown* implementation. The source will store this with any other sinks that may be listening to events. If the call to *Advise* was successful, the sink will be given an identifier—a simple unsigned long value, called a cookie—to give back to the source at a later point when it no longer needs to listen to events.

The connection is now complete; methods will be called on any listening sinks by the source. The sink will typically hold onto an interface pointer to the source, so when a sink has finished listening it can be released from the source object by calling *IConnectionPoint::Unadvise*. This is implemented with *AtlUnadvise*.

IDispatch events versus pure COM events

An outbound interface can be a pure dispatch interface. This means instead of the source calling directly onto a method in a sink, the call is made via the

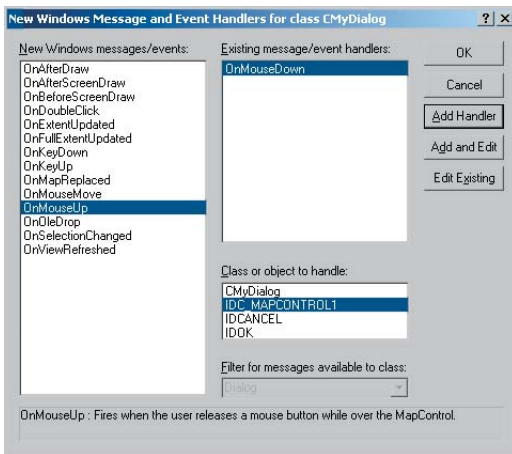
IDispatch::Invoke mechanism. The *IDispatch* mechanism has a performance overhead to package parameters compared to a pure vtable COM call. However, there are some situations where this must be used. ActiveX controls must implement their default outbound interface as a pure *IDispatch* interface; for example, *IMapControlEvents2* is a pure dispatch interface. Second, Microsoft Visual Basic 6 can only be a source of pure *IDispatch* events. The connection point mechanism is the same as for pure COM mechanisms, the main difference being in how the events are fired.

ATL provides some macros to assist with listening to *IDispatch* events; this is discussed on MSDN under ‘Event Handling and ATL’. There are two templates available, *IDispEventImpl* and *IDispEventSimpleImpl*, that are discussed in the following sections.

Using IDispEventImpl to listen to events

The ATL template *IDispEventImpl* will use a type library to “crack” the *IDispatch* calls and process the arguments into C++ method calls. The Visual Studio Class wizard can provide this mechanism automatically when adding an ActiveX control to a dialog box. Right-click the Control and click Events. In the Class wizard choose the resource ID of the control, choose the event, then click Add Handler.

The following code illustrates the event handling code added by the wizard, with some modifications to ensure advise and unadvise are performed.



Visual Studio C++ Class Wizard. Adding event handler to an ActiveX control on a dialog box

There is a bug in the wizard: it does not add the advise and unadvise code to the dialog box.

To fix this issue, add a message handler for *OnDestroy*. Then in the *OnInitDialog* handler, call *AtlAdviseSinkMap* with a *TRUE* second parameter to begin listening to events. Place a corresponding call to *AtlAdviseSinkMap* (with *FALSE* as the second parameter) in the *OnDestroy* handler. This is discussed further in the MSDN article “BUG: ActiveX Control Events Are Not Fired in ATL Dialog (Q190530)”.

```

#pragma once

#include "resource.h" // Main symbols
#include <atlhost.h>

/////////////////////////////////////////////////////////////////
// CMyDialog
class CMyDialog :
public CxDialogImpl<CMyDialog>,
public IDispEventImpl<IDC_MAPCONTROL1, CMyDialog>
{
public:

enum { IDD = IDD_MYDIALOG };

BEGIN_MSG_MAP(CMyDialog)
MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)

// Add a handler to ensure event unadvise occurs.
MESSAGE_HANDLER(WM_DESTROY, OnDestroy)

COMMAND_ID_HANDLER(IDOK, OnOK)
COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
END_MSG_MAP()

LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
// Calls IConnectionPoint::Advise() for each control on the dialog box
with sink map entry
AtlAdviseSinkMap(this, TRUE);
return 1; // Let the system set the focus.
}

LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
// Calls IConnectionPoint::Unadvise() for each control on the dialog box
with sink map entry
AtlAdviseSinkMap(this, FALSE);
return 0;
}

LRESULT OnOK(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
{
EndDialog(wID);
return 0;
}

LRESULT OnCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
{
EndDialog(wID);
}

```

The following issues with events are documented on the MSDN Knowledge Base when using `IDispEventImpl`. Fixes to ATL code are shown in MSDN for these issues; however, it is not always desirable to modify or copy ATL header files. In this case, the `IDispEventSimpleImpl` can be used instead.

BUG: Events Fail in ATL Containers when Enum Used as Event Parameter (Q237771)
 BUG: IDispEventImpl Event Handlers May Give Strange Values for Parameters (Q241810)

```

    return 0;
}

// ATL callback from SinkMap entry
VOID __stdcall OnMouseDownMapControl1(LONG button, LONG shift, LONG x,
LONG y, DOUBLE mapX, DOUBLE mapY)
{
    MessageBox(_T("MouseDown!"));
}

BEGIN_SINK_MAP(CMyDialog)
// Make sure the Event Handlers have __stdcall calling convention.
// The 0x1 is the Dispatch ID of the OnMouseDown method.
    SINK_ENTRY(IDC_MAPCONTROL1, 0x1, OnMouseDownMapControl1)
END_SINK_MAP()
};
    
```

Using IDispEventSimpleImpl to listen to events

As the name of this template suggests, it is a simpler version of *IDispEventImpl*. The type library is no longer used to turn the *IDispatch* arguments into a C++ method call. While this may be a simpler implementation, it now requires the developer to supply a pointer to a structure describing the format of the event parameters. This structure is typically placed in the .cpp file. For example, here is the structure describing the parameters of an *OnMouseDown* event for the *MapControl*:

```

_ATL_FUNC_INFO g_ParamInfo_MapControl_OnMouseDown =
{
    CC_STDCALL,                // Calling convention
    VT_EMPTY,                 // Return type
    6,                        // Number of arguments
    {VT_I4, VT_I4, VT_I4, VT_I4, VT_R8, VT_R8} // VariantArgument types
};
    
```

The header file now inherits from *IDispEventSimpleImpl* and uses a different macro, *SINK_ENTRY_INFO*, in the *SINK_MAP*. Also, the events interface ID is required; *#import* can be used to define this symbol. Note that a dispatch interface is normally prefixed with *DIID* instead of *IID*.

```

#pragma once

#include "resource.h" // Main symbols
#include <atlhost.h>

// reference to structure defining event parameters
extern _ATL_FUNC_INFO g_ParamInfo_MapControl_OnMouseDown;

////////////////////////////////////
// CMyDialog2
class CMyDialog2 :
    public CAxDialogImpl<CMyDialog2>,
    public IDispEventSimpleImpl<IDC_MAPCONTROL1, CMyDialog2,
    &DIID_IMapControlEvents2>
    
```

See the 'Importing ArcGIS type libraries' section earlier in this appendix for an explanation of `#import`.


```

{
public:

// Message handler code removed, it is the same as CMyDialog using
IDispEventSimple

BEGIN_SINK_MAP(CMyDialog2)
// Make sure the Event Handlers have __stdcall calling convention.
// The 0x1 is the Dispatch ID of the OnMouseDown method.
SINK_ENTRY_INFO(IDC_MAPCONTROL1, // ID of event source
                DIID_IMapControlEvents2, // interface to listen to
                0x1, // dispatch ID of MouseDown
                OnMapControlMouseDown, // method to call when event arrives
                &g_ParamInfo_MapControl_OnMouseDown) // parameter info for method
call

END_SINK_MAP()
};

```

Listening to more than one IDispatch event interface on a COM object

If a single COM object needs to receive events from more than one *IDispatch* source, then this can cause compiler issues with ambiguous definitions of the *DispEventAdvise* method. This is not normally a problem in a dialog box, as *AtlAdviseSinkMap* will handle all the connections. The ambiguity can be avoided by introducing different typedefs each time *IDispatchSimpleImpl* is inherited. The following example illustrates a COM object called *CListen*, which is a sink for dispatch events from a *MapControl* and a *PageLayoutControl*.

```

#pragma once

#include "resource.h" // Main symbols

// This is the parameter information
extern _ATL_FUNC_INFO g_ParamInfo_MapControl_OnMouseDown;
extern _ATL_FUNC_INFO g_ParamInfo_PageLayoutControl_OnMouseDown;

//
// Define some typedefs of the dispatch template.
//
class CListen; // Forward definition

typedef IDispatchSimpleImpl<0, CListen, &DIID_IMapControlEvents2>
    IDispatchSimpleImpl_MapControl;

typedef IDispatchSimpleImpl<1, CListen, &DIID_IPageLayoutControlEvents>
    IDispatchSimpleImpl_PageLayoutControl;

////////////////////////////////////
// CListen

class ATL_NO_VTABLE CListen :

```

```

public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CListen,&CLSID_Listen>,
public IDispEventSimpleImpl_MapControl,
public IDispEventSimpleImpl_PageLayoutControl,
public IListen
{
public:
    CListen()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_LISTEN)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CListen)
    COM_INTERFACE_ENTRY(IListen)
END_COM_MAP()

// Associated source and dispatchID to a method call
BEGIN_SINK_MAP(CListen)
    SINK_ENTRY_INFO(0, // ID of event source
                    DIID_IMapControlEvents2, // Interface to listen to
                    0x1, // Dispatch ID to receive
                    OnMapControlMouseDown, // Method to call when event arrives
                    &g_ParamInfo_MapControl_OnMouseDown) // Parameter info for
                                                    // method call

    SINK_ENTRY_INFO(1,
                    DIID_IPageLayoutControlEvents,
                    0x1,
                    OnPageLayoutControlMouseDown,
                    &g_ParamInfo_PageLayoutControl_OnMouseDown)
END_SINK_MAP()

// IListen
public:
    STDMETHOD(SetControls)(IUnknown* pMapControl, IUnknown*
pPageLayoutControl);
    STDMETHOD(Clear)();

private:
    void __stdcall OnMapControlMouseDown(long button, long shift, long x, long
y, double mapX, double mapY);
    void __stdcall OnPageLayoutControlMouseDown(long button, long shift, long
x, long y, double pageX, double pageY);

    IUnknownPtr m_ipUnkMapControl;
    IUnknownPtr m_ipUnkPageLayoutControl;
};

```

The implementation of `CListen` contains the following code to start listening to the controls; the typedef avoids the ambiguity of the `DispEventAdvise` implementation.

```
// Start listening to the MapControl.
IUnknownPtr ipUnk = pMapControl;
HRESULT hr = IDispEventSimpleImpl_MapControl::DispEventAdvise(ipUnk);
if (SUCCEEDED(hr))
    m_ipUnkMapControl = ipUnk; // Store pointer to MapControl for Unadvise.

// Start listening to the PageLayoutControl.
ipUnk = pPageLayoutControl;
hr = IDispEventSimpleImpl_PageLayoutControl::DispEventAdvise(ipUnk);
if (SUCCEEDED(hr))
    m_ipUnkPageLayoutControl = ipUnk; // Store pointer to PageLayoutControl
// for Unadvise.
```

The implementation of `CListen` also contains the following code to `UnAdvise` and stop listening to the controls.

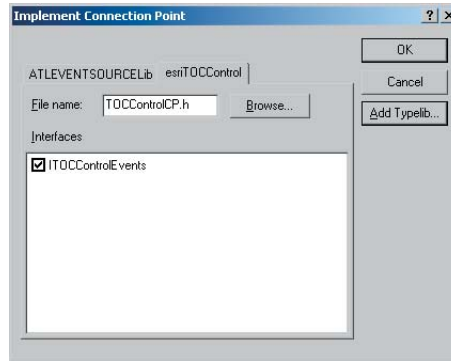
```
// Stop listening to the MapControl.
if (m_ipUnkMapControl!=0)
    IDispEventSimpleImpl_MapControl::DispEventUnadvise(m_ipUnkMapControl);
m_ipUnkMapControl = 0;

if (m_ipUnkPageLayoutControl!=0)
    IDispEventSimpleImpl_PageLayoutControl::DispEventUnadvise(m_ipUnkPageLayoutControl);
m_ipUnkPageLayoutControl= 0;
```

Creating a COM events source

For an object to be a source of events, it will need to provide an implementation of `IConnectionPointContainer` and a mechanism to track which sinks are listening to which `IConnectionPoint` interfaces. ATL provides this through the `IConnectionPointContainerImpl` template. In addition, ATL provides a wizard to generate code to fire `IDispatch` events for all members of a given dispatch events interface. Below are the steps to modify an ATL COM coclass to support a connection point:

1. First ensure that your ATL coclass has been compiled at least once. This will allow the wizard to find an initial type library.
2. In Class view, right-click the COM object and click Implement Connection Point.
3. Either use a definition of events from the IDL in the project or click Add Type Lib to browse for another definition.
4. Check the outbound interface to be implemented in the coclass.



5. Clicking OK will modify your ATL class and generate the proxy classes in a header file, with a name ending in CP, for firing events.

If the wizard fails to run, use the following example, which illustrates a coclass that is a source of *ITOCControlEvents*, a pure dispatch interface.

```
#pragma once

#include "resource.h" // Main symbols
#include "TOCControlCP.h" // Include generated connection point class
// for firing events.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CMyEventSource
class ATL_NO_VTABLE CMyEventSource :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CMyEventSource,&CLSID_MyEventSource>,
public IMyEventSource,
public CProxyITOCControlEvents< CMyEventSource >, // Generated
// ConnectionPoint class
public IConnectionPointContainerImpl< CMyEventSource > // Implementation
// of Connection point Container
{
public:
    CMyEventSource()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_MYEVENTSOURCE)

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    BEGIN_COM_MAP(CMyEventSource)
        COM_INTERFACE_ENTRY(IMyEventSource)
        COM_INTERFACE_ENTRY(IConnectionPointContainer) // Allow QI to this
// interface.
    END_COM_MAP()
}
```

```
// List of available connection points
BEGIN_CONNECTION_POINT_MAP(CMyEventSource)
    CONNECTION_POINT_ENTRY(DIID_ITOCControlEvents)
END_CONNECTION_POINT_MAP()
};
```

The connection point class (*TOCControlEventsCP.h* in the above example) contains code to fire an event to all sink objects on a connection point.

There is one method in the class for each event beginning “Fire_”. Each method will build a parameter list of variants to pass as an argument to the dispatch Invoke method. Each sink is iterated, and a pointer to the sink is stored in a vector `m_vec` member variable inherited from *IConnectionPointContainerImpl*. Note that `m_vec` can contain pointers to zero; this must be checked before firing the event.

```
template <class T>
class CProxyITOCControlEvents : public IConnectionPointImpl<T,
&DIID_ITOCControlEvents, CComDynamicUnkArray>
{
public:
    VOID Fire_OnMouseDown(LONG button, LONG shift, LONG x, LONG y)
    {
        // Package each of the parameters into an IDispatch argument list.
        T* pT = static_cast<T*>(this);
        int nConnectionIndex;
        CComVariant* pvars = new CComVariant[4];
        int nConnections = m_vec.GetSize();

        // Iterate each sink object.
        for (nConnectionIndex = 0; nConnectionIndex < nConnections;
nConnectionIndex++)
        {
            pT->Lock();
            CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
            pT->Unlock();
            IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);

            // Note m_vec can contain 0 entries so it is important to check for this.
            if (pDispatch != NULL)
            {
                // Build up the argument list.
                pvars[3] = button;
                pvars[2] = shift;
                pvars[1] = x;
                pvars[0] = y;
                DISPPARAMS disp = { pvars, NULL, 4, 0 };

                // Fire the dispatch method, 0x1 is the DispatchId for MouseDown.
                pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT,
DISPATCH_METHOD, &disp, NULL, NULL, NULL);
            }
        }
    }
};
```

```

    }
}
delete[] pvars; // Clean up the parameter list.

}
VOID Fire_OnMouseUp(LONG button, LONG shift, LONG x, LONG y)
{
    // ... Other events

```

To fire an event from the source, simply call the *Fire_OnMouseDown* when required.

A similar approach can be used for firing events to a pure COM (non IDispatch) interface. The wizard will not generate the connection point class, so this must be written by hand; the following example illustrates a class that will fire an *ITOCBuddyEvents::ActiveViewReplaced* event; *ITOCBuddyEvents* is a pure COM, non-IDispatch interface. The key difference is that there is no need to package the parameters. A direct method call can be made.

```

template < class T >
class CProxyTOCBuddyEvents : public IConnectionPointImpl< T,
&IID_ITOCBuddyEvents, CComDynamicUnkArray >
{
    // This class based on the ATL-generated connection point class
public:
    void Fire_ActiveViewReplaced(IActiveView* pNewActiveView)
    {
        T* pT = static_cast< T* >(this);
        int nConnectionIndex;
        int nConnections = this->m_vec.GetSize();
        for (nConnectionIndex = 0; nConnectionIndex < nConnections;
nConnectionIndex++)
        {
            pT->Lock();
            CComPtr< IUnknown > sp=this->m_vec.GetAt(nConnectionIndex);
            pT->Unlock();
            ITOCBuddyEvents* pTOCBuddyEvents = reinterpret_cast< ITOCBuddyEvents*
>(sp.p);
            if (pTOCBuddyEvents)
                pTOCBuddyEvents->ActiveViewReplaced(pNewActiveView);
        }
    }
};

```

IDL declarations for an object that supports events

When an object is exported to a type library, the event interfaces are declared by using the *[source]* tag against the interface name. For example, an object that fires *ITOCBuddyEvents* declares

```
[source] interface ITOCBuddyEvents;
```

If the outbound interface is a dispatch events interface, *dispinterface* is used instead of *interface*. Additionally, a coclass can have a default outbound interface;

this is specified with the *[default]* tag. Default interfaces are identified by some design environments (for example, Visual Basic 6). Following is the declaration for the default outbound events interface:

```
[default, source] dispinterface IMyEvents2;
```

Event circular reference issues

After a sink has performed an advise on the source, there is typically a COM circular reference. This occurs because the source has an interface pointer to a sink to fire events, and this keeps the sink alive. Similarly, a sink object has a pointer back to the source so it can perform the unadvise at a later point. This keeps the source alive. Therefore, these two objects will never be released and may cause substantial memory leaks. There are a number of ways to tackle this issue:

1. Ensure the advise and unadvise are made on a method or windows message that is guaranteed to happen in pairs and is independent of an object's life cycle. For example, in a coclass that is also receiving windows messages, use the Windows messages *OnCreate* (WM_CREATE) and *OnDestroy* (WM_DESTROY) to advise and unadvise.
2. If an ATL dialog box class needs to listen to events, one approach is to make the dialog box a private COM class and implement the events interface directly on the dialog box. ATL allows this without much extra coding. This approach is illustrated below. The dialog box class creates a *CustomizeDialog* coclass and listens to *ICustomizeDialogEvents*. The *OnInitDialog* and *OnDestroy* methods (corresponding to window messages) are used to advise and unadvise on the *CustomizeDialog*.

```
class CEngineControlsDlg :
    public CxDialogImpl<CEngineControlsDlg>,
    public CComObjectRoot, // Make Dialog Class a COM Object as well.
    public ICustomizeDialogEvents // Implement this interface directly on
this object.

    CEngineControlsDlg() : m_dwCustDlgCookie(0) {} // initialize cookie for
event listening

    // ... Event handlers and other standard dialog code has been removed ...

BEGIN_COM_MAP(CEngineControlsDlg)
    COM_INTERFACE_ENTRY(ICustomizeDialogEvents) // Make sure QI works for
// this event interface.
END_COM_MAP()

    // ICustomizeDialogEvents implementation to receive events on this
// dialog box.
    STDMETHOD(OnStartDialog)();
    STDMETHOD(OnCloseDialog)();

    ICustomizeDialogPtr    m_ipCustomizeDialog; // The source of events
    DWORD                  m_dwCustDlgCookie; // Cookie for
// CustomizeDialogEvents
}
```

The dialog box needs to be created like a noncreatable COM object, rather than on the stack as a local variable. This allocates the object on the heap and allows it to be released through the COM reference counting mechanism.

```
// Create dialog class on the heap using ATL CComObject template.
```

```
CComObject<CEngineControlsDlg> *myDlg;
```

```
CComObject<CEngineControlsDlg>::CreateInstance(&myDlg);
```

```
myDlg->AddRef(); // Keep dialog box alive until you're done with it.
```

```
myDlg->DoModal(); // Launch the dialog box; when method returns, dialog box  
// has exited.
```

```
myDlg->Release(); // Typically, the refcount now goes to 0 and frees the  
// dialog object.
```

3. Implement an intermediate COM object for use by the sink; this is sometimes called a listener or event helper object. This object typically contains no implementation but simply uses C++ method calls to forward events to the sink object. The listener has its reference count incremented by the source, but the sink's reference count is unaffected. This breaks the cycle, allowing the sink's reference count to reach 0 when all other references are released. As the sink executes its destructor code, it instructs the listener to unadvise and release the source.

An alternative to using C++ pointers to communicate between listener and sink is to use an interface pointer that is a weak reference. That is, the listener contains a COM pointer to the sink but does not increment the sink's reference count. It is the responsibility of the sink to ensure that this pointer is not accessed after the sink object has been released.

WHAT IS THE .NET FRAMEWORK?

The .NET Framework is an integral Windows component that supports building and running the next generation of applications and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- Provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- Provide a code execution environment that minimizes software deployment and versioning conflicts.
- Provide a code execution environment that guarantees safe execution of code, including code created by an unknown or semitrusted third party.
- Provide a code execution environment that eliminates the performance problems of scripted or interpreted environments.
- Make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- Build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language run time and the .NET Framework class library. The common language run time is the foundation of the .NET Framework. You can think of the run time as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness. In fact, the concept of code management is a fundamental principle of the run time. Code that targets the run time is known as managed code, while code that does not target the run time is known as unmanaged code. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language run time into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework not only provides several run-time hosts but also supports the development of third-party run-time hosts.

For example, ASP.NET hosts the run time to provide a scalable, server-side environment for managed code. ASP.NET works directly with the run time to enable ASP.NET applications and XML Web services, both of which are discussed later in this topic.

Internet Explorer is an example of an unmanaged application that hosts the run time (in the form of a MIME type extension). Using Internet Explorer to host the run time enables you to embed managed components or Windows Forms controls in HTML documents. Hosting the run time in this way makes managed

mobile code (similar to Microsoft ActiveX controls) possible, but with significant improvements that only managed code can offer, such as semitrusted execution and secure isolated file storage.

The following sections describe the main components and features of the .NET Framework in greater detail.

Features of the common language run time

The common language run time manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These

features are intrinsic to the managed code that runs on the common language run time.

Regarding security, managed components are awarded varying degrees of trust, depending on a number of factors that includes their origin, such as the Internet, enterprise network, or local computer. This means that a managed component might or might not be able to perform file access operations, registry access operations, or other sensitive functions, even if it is being used in the same active application.

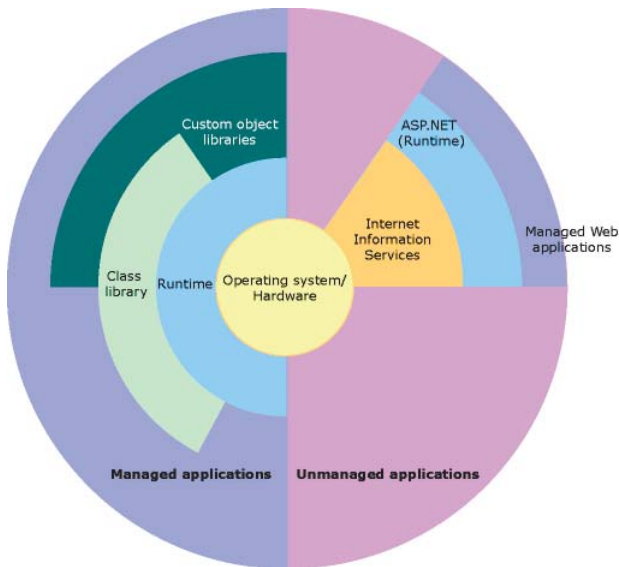
The run time enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation onscreen or sing a song but cannot access their personal data, file system, or network. The security features of the run time thus enable legitimate Internet-deployed software to be exceptionally feature rich.

The run time also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The

various Microsoft and third-party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the run time eliminates many common software issues. For example, the run time automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors: memory leaks and invalid memory references.

The run time also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the run time, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the run time can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.



This diagram shows the relationship of the common language run time and the class library to your applications and to the overall system. It also illustrates how managed code operates within a larger architecture.

While the run time is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The run time is designed to enhance performance. Although the common language run time provides many standard run-time services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the run time can be hosted by high-performance, server-side applications, such as Microsoft SQL Server™ and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support run-time hosting.

.NET Framework class library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language run time. The class library is object-oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

- Console applications
- Windows GUI applications (Windows Forms)
- ASP.NET applications
- XML Web services
- Windows services

For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form application, you can use the Windows Forms classes.

Client application development

Client applications are the closest to a traditional style of application in Windows-based programming. These are the types of applications that display windows or forms on the desktop, enabling a user to perform a task. Client applications include applications such as word processors and spreadsheets, as well as custom business applications such as data entry and reporting tools. Client applications usually employ windows, menus, buttons, and other GUI elements, and they likely access local resources, such as the file system, and peripherals such as printers.

Another kind of client application is the traditional ActiveX control (now replaced by the managed Windows Forms control) deployed over the Internet as a Web page. This application is much like other client applications: it is executed natively, has access to local resources, and includes graphical elements.

In the past, developers created such applications using C or C++ in conjunction with the Microsoft Foundation Classes or with a rapid application development (RAD) environment such as Microsoft Visual Basic. The .NET Framework incorporates aspects of these existing products into a single, consistent development environment that drastically simplifies the development of client applications.

The Windows Forms classes contained in the .NET Framework are designed to be used for GUI development. You can easily create command windows, buttons, menus, toolbars, and other screen elements with the flexibility necessary to accommodate shifting business needs.

For example, the .NET Framework provides simple properties to adjust visual attributes associated with forms. In some cases the underlying operating system does not support changing these attributes directly, and in these cases the .NET Framework automatically re-creates the forms. This is one of many ways in which the .NET Framework integrates the developer interface, making coding simpler and more consistent.

Unlike ActiveX controls, Windows Forms controls have semitrusted access to a user's computer. This means that binary or natively executing code can access some of the resources on the user's system, such as GUI elements and limited file access, without being able to access or compromise other resources. Because of code access security, many applications that once needed to be installed on a user's system can now be safely deployed through the Web. Your applications can implement the features of a local application while being deployed like a Web page.

Server application development

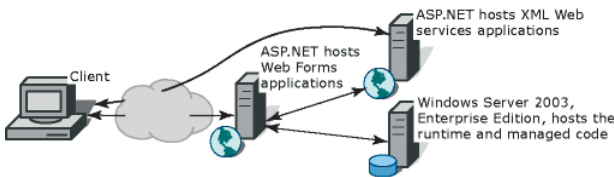
Server-side applications in the managed world are implemented through run-time hosts. Unmanaged applications host the common language run time, which allows your custom managed code to control the behavior of the server. This model provides you with all the features of the common language run time and class library while gaining the performance and scalability of the host server.

Server-side managed code

ASP.NET is the hosting environment that enables developers to use the .NET Framework to target Web-based applications. However, ASP.NET is more than a run-time host; it is a complete architecture for developing Web sites and Internet-distributed objects using managed code. Both Web Forms and XML Web services use IIS and ASP.NET as the publishing mechanism for applications, and both have a collection of supporting classes in the .NET Framework.

XML Web services, an important evolution in Web-based technology, are distributed, server-side application components similar to common Web sites. However, unlike Web-based applications, XML Web services components have no UI and are not targeted for browsers, such as Internet Explorer and Netscape Navigator.

Instead, XML Web services consist of reusable software components designed to be consumed by other applications, such as traditional client applications, Web-based applications, or even other XML Web services. As a result, XML Web services technology is rapidly moving application development and deployment into the highly distributed environment of the Internet.



This diagram illustrates a basic network schema with managed code running in different server environments. Servers, such as IIS and SQL Server, can perform standard operations while your application logic executes the managed code.

If you have used earlier versions of ASP technology, you will immediately notice the improvements that ASP.NET and Web Forms offer. For example, you can develop Web Forms pages in any language that supports the .NET Framework. In addition, your code no longer needs to share the same file with your HTTP text (although it can continue to do so if you prefer). Web Forms pages execute in native machine language because, like any other managed application, they take full advantage of the run time. In contrast, unmanaged ASP pages are always scripted and interpreted. ASP.NET pages are faster, more functional, and easier to develop than unmanaged ASP pages because they interact with the run time like any managed application.

The .NET Framework also provides a collection of classes and tools to aid in development and consumption of XML Web services applications. XML Web services are built on standards such as SOAP, a remote procedure-call protocol; XML, an extensible data format; and WSDL, the Web Services Description Language. The .NET Framework is built on these standards to promote interoperability with non-Microsoft solutions.

For example, the Web Services Description Language tool included with the .NET Framework SDK can query an XML Web service published on the Web, parse its WSDL description, and produce C# or Visual Basic source code that your application can use to become a client of the XML Web service. The source code can create classes derived from classes in the class library that handle all the underlying communication using SOAP and XML parsing. Although you can use the class library to consume XML Web services directly, the Web Services Description Language tool and the other tools contained in the SDK facilitate your development efforts with the .NET Framework.

If you develop and publish your own XML Web service, the .NET Framework provides a set of classes that conform to all the underlying communication stan-

dards, such as SOAP, WSDL, and XML. Using those classes enables you to focus on the logic of your service, without concerning yourself with the communications infrastructure required by distributed software development.

Finally, like Web Forms pages in the managed environment, your XML Web service will run with the speed of native machine language using the scalable communication of IIS.

INTEROPERATING WITH COM

Code running under the .NET Framework's control is called managed code; conversely, code executing outside the .NET Framework is termed unmanaged code. COM is one example of unmanaged code. The .NET framework interacts with COM via a technology known as COM Interop.

For COM Interop to work, the CLR requires metadata for all the COM types. This means that the COM type definitions normally stored in the type libraries need to be converted to .NET metadata. This is easily accomplished with the Type Library Importer utility (tlbimp.exe), which ships with the .NET Framework SDK. This utility generates interop assemblies containing the metadata for all the COM definitions in a type library. Once metadata is available, .NET clients can seamlessly create instances of COM types and call its methods as though they were native .NET instances.

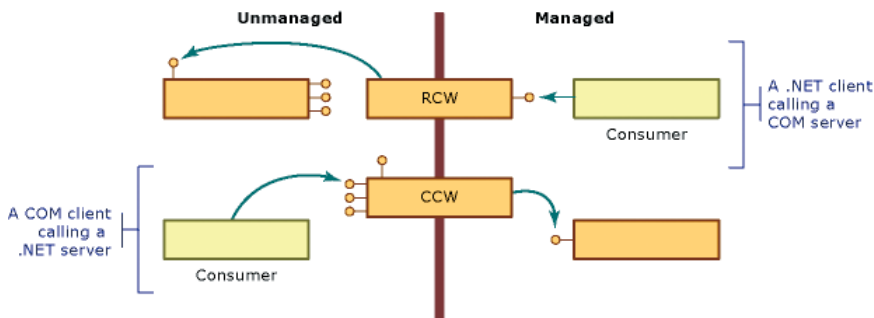
Primary interop assemblies

Primary interop assemblies (PIAs) are the official, vendor-supplied, .NET type definitions for interoperating with underlying COM types. Primary interop assemblies are strongly named by the COM library publisher to guarantee uniqueness.

ESRI provides primary interop assemblies for all the ArcObjects type libraries that are implemented with COM. ArcGIS .NET developers should only use these primary interop assemblies that are installed in the Global Assembly Cache (GAC) during install if version 1.1 of the .NET Framework is detected. ESRI only supports the interop assemblies that ship with ArcGIS. You can identify a valid ESRI assembly by its public key (8FC3CC631E44AD86).

COM wrappers

The .NET run time provides wrapper classes to make both managed and



unmanaged clients believe they are communicating with objects within their respective environment. When managed clients call a method on a COM object, the run time creates a run-time callable wrapper (RCW) that handles the marshalling between the two environments. Similarly, the .NET run time creates COM callable wrappers for the reverse

case, COM clients communicating with .NET components. The illustration above outlines this process.

Exposing .NET components to COM

When creating .NET components that COM clients will make use of, follow the guidelines listed below to ensure interoperability.

- Avoid using parameterized constructors.
- Avoid using static methods.
- Define event source interfaces in managed code.
- Include HRESULTs in user-defined exceptions.
- Supply GUIDs for types that require them.
- Expect inheritance differences.

For more information, review 'Interoperating with Unmanaged Code' in the MSDN help collection.

Performance considerations

COM Interop clearly adds a new layer of overhead to applications, but the overall cost of interoperating between COM and .NET is small and often unnoticeable. However, the cost of creating wrappers and having them marshal between environments does add up; if you suspect COM Interop is the bottleneck in your application's performance, try creating a COM worker class that wraps all the chatty COM calls into one function that managed code can invoke. This improves performance by limiting the marshalling between the two environments.

COM to .NET type conversion

Generally speaking, the type library importer imports types with the same name they originally had in COM. All imported types are additionally added to a namespace that has the following naming convention: ESRI.ArcGIS plus the name of the library. For example, the namespace for the Geometry library is ESRI.ArcGIS.Geometry. All types are identified by their complete namespace and type name.

Classes, Interfaces, and Members

All COM coclasses are converted to managed classes; the managed classes have the same name as the original with 'Class' appended. For example, the Point coclass is PointClass.

All classes additionally have an interface with the same name as the coclass that corresponds to the default interface for the coclass. For example, the PointClass has a Point interface. The type library importer adds this interface so clients can register event sinks.

The .NET classes additionally have class members that .NET supports but COM does not. Each member of each interface the class implements is added as a class member. Any property or method a class implants can be accessed directly from the class rather than having to cast to a specific interface. Since interface member names are not unique, name conflicts are resolved by prefixing the interface name and an underscore to the name of each conflicting member. When member names conflict, the first interface listed with the coclass remains unchanged.

Properties in C# that have by-reference or multiple parameters are not supported with the regular property syntax. In these cases, it is necessary to use the accessor methods instead. The following code excerpt shows an example.

```
ILayer layer = mapControl.get_Layer(0);  
MessageBox.Show(layer.Name);
```

Events

The type library importer creates several types that enable managed applications to sink to events fired by COM classes. The first type is a delegate that is named after the event interface plus an underscore followed by the event name, then the word `EventHandler`. For example, the *SelectionChanged* event defined on the *ActiveViewEvents* interface has the following delegate defined: *ActiveViewEvents_SelectionChangedEventHandler*. The importer additionally creates an event interface with a ‘_Event’ suffix added to the end of the original interface name. For example, *ActiveViewEvents* generates *ActiveViewEvents_Event*. Use the event interfaces to set up event sinks.

Non-OLE Automation Compliant Types

COM types that are not OLE automation compliant generally do not work in .NET. ArcGIS contains a few noncompliant methods and these cannot be used in .NET. However, in most cases, supplemental interfaces have been added that have the offending members rewritten compliantly. For example, when defining an envelope via a point array, you can't use *IEnvelope::DefineFromPoints*; instead, you must use *IEnvelopeGEN::DefineFromPoints*.

[VB.NET]

```
Dim pointArray(1) As IPoint  
pointArray(0) = New PointClass  
pointArray(1) = New PointClass  
pointArray(0).PutCoords(0, 0)  
pointArray(1).PutCoords(100, 100)  
  
Dim env As IEnvelope  
Dim envGEN As IEnvelopeGEN  
env = New EnvelopeClass  
envGEN = New EnvelopeClass  
  
'Won't compile  
'env.DefineFromPoints(2, pointArray)  
  
'Doesn't work  
env.DefineFromPoints(2, pointArray(0))  
  
'Works  
envGEN.DefineFromPoints(pointArray)
```

[C#]

```
IPoint[] pointArray = new IPoint[2];  
pointArray[0] = new PointClass();  
pointArray[1] = new PointClass();  
pointArray[0].PutCoords(0,0);
```



```
pointArray[1].PutCoords(100,100);

IEnvelope env = new EnvelopeClass();
IEnvelopeGEN envGEN = new EnvelopeClass();

// Won't compile
env.DefineFromPoints(3, ref pointArray);

// Doesn't work
env.DefineFromPoints(3, ref pointArray[0]);

// Works
envGEN.DefineFromPoints(ref pointArray);
```

.NET PROGRAMMING TECHNIQUES AND CONSIDERATIONS

This section contains several programming tips and techniques to help developers who are moving to .NET.

Casting between interfaces (QueryInterface)

.NET uses casting to jump from one interface to another interface on the same class. In COM this is called QueryInterface. VB .NET and C# cast differently.

VB .NET

There are two types of casts, implicit and explicit. Implicit casts require no additional syntax, whereas explicit casts require cast operators.

```
geometry = point           'Implicit cast
geometry = CType(point, IGeometry) 'Explicit cast
```

When casting between interfaces, it's perfectly acceptable to use implicit casts because there is no chance of data loss as there is when casting between numeric types. However, when casts fail, an exception (`System.InvalidCastException`) is thrown; to avoid handling unnecessary exceptions, it's best to test if the object implements both interfaces beforehand. The recommended technique is to use the *TypeOf* keyword, which is a comparison clause that tests whether an object is derived from or implements a particular type, such as an interface. The example below performs an implicit conversion from an *IPoint* to an *IGeometry* only if at run time it is determined that the *Point* class implements *IGeometry*.

```
Dim point As New PointClass
Dim geometry As IGeometry
If (TypeOf point Is IGeometry) Then
    geometry = point
End If
```

If you prefer using the `Option Strict On` statement to restrict implicit conversions, use the `CType` function to make the cast explicit. The example below adds an explicit cast to the code sample above.

```
Dim point As New PointClass
Dim geometry As IGeometry
If (TypeOf point Is IGeometry) Then
    geometry = CType(point, IGeometry)
End If
```

C#

In C#, the best method for casting between interfaces is to use the *as* operator. Using the *as* operator is a better coding strategy than a straight cast because it yields a null on a conversion failure rather than raising an exception.

The first line of code below is a straight cast. This is acceptable practice if you are absolutely certain the object in question implements both interfaces; if the object does not implement the interface you are attempting to get a handle to, .NET will throw an exception. A safer model is to use the *as* operator that returns a null if the object cannot return a reference to the desired interface.

```
IGeometry geometry = point;           // Straight cast
IGeometry geometry = point as IGeometry; // As operator
```

The example below shows how to handle the possibility of a returned null interface handle.

```
IPoint point = new PointClass();
IGeometry geometry = point;
IGeometry geometry = point as IGeometry;
if (geometry != null)
{
    Console.WriteLine(geometry.GeometryType.ToString());
}
```

Binary compatibility

Most existing ArcGIS Visual Basic 6 developers are familiar with the notion of binary compatibility. This compiler flag in Visual Basic ensures that components maintain the same GUID each time they are compiled. When this flag is not set, a new GUID is generated for each class every time the project is compiled. This has the adverse side effect of having to then re-register the components in their appropriate component categories.

To keep from having the same problem in .NET, you can use the *GUIDAttribute* class to manually specify a GUID for a class. Explicitly specifying a GUID guarantees that it will never change. If you do not specify a GUID, the type library exporter will automatically generate one when you first export your components to COM and, although the exporter is meant to keep using the same GUIDs on subsequent exports, it's not guaranteed to do so.

The example below shows a GUID attribute being applied to a class.

```
[VB.NET]
<GuidAttribute("9ED54F84-A89D-4fcd-A854-44251E925F09")> _
Public Class SampleClass
    '
End Class
```

```
[C#]
[GuidAttribute("9ED54F84-A89D-4fcd-A854-44251E925F09")]
Public class SampleClass
{
    //
}
```

Events

An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic. The object that raises (triggers) the event is called the event sender. The object that captures the event and responds to it is called the event receiver.

In event communication, the event sender class does not know which object or method will receive (handle) the events it raises. What is needed is an intermediary (or pointer-like mechanism) between the source and the receiver. The .NET Framework defines a special type (<Delegate>) that provides the functionality of a function pointer.

A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback.

To consume an event in an application, you must provide an event handler (an event-handling method) that executes program logic in response to the event and register the event handler with the event source. The event handler must have the same signature as the event delegate. This process is referred to as event wiring.

The ArcObjects code excerpt below shows a custom ArcMap command wiring up to the *Map* object's selection changed event. For simplicity, the event is wired up in the *OnClick* event.

[VB.NET]

```
'Can't use WithEvents because the outbound interface is not the
'default interface

'IActiveViewEvents is the sink event interface.
'SelectionChanged is the name of the event.
'IActiveViewEvents_SelectionChangedEventHandler is the delegate name.

'Declare the delegate.
Private SelectionChanged As IActiveViewEvents_SelectionChangedEventHandler

Private m_mxDoc As IMxDocument

Public Overloads Overrides Sub OnCreate(ByVal hook As Object)
    Dim app As IApplication
    app = hook
    m_mxDoc = app.Document
End Sub

Public Overrides Sub OnClick()
    Dim map As Map
    map = m_mxDoc.FocusMap

    'Create an instance of the delegate, add it to SelectionChanged event.
    SelectionChanged = New
    IActiveViewEvents_SelectionChangedEventHandler(AddressOf OnSelectionChanged)
```

```
        AddHandler map.SelectionChanged, SelectionChanged

    End Sub

    'Event handler
    Private Sub OnSelectionChanged()
        MessageBox.Show("Selection Changed")
    End Sub

[C#]
// IActiveViewEvents is the sink event interface.
// SelectionChanged is the name of the event.
// IActiveViewEvents_SelectionChangedEventHandler is the delegate name.
IActiveViewEvents_SelectionChangedEventHandler m_selectionChanged;
private ESRI.ArcGIS.ArcMapUI.IMxDocument m_mxDoc;

public override void OnCreate(object hook)
{
    IApplication app = hook as IApplication;
    m_mxDoc = app.Document as IMxDocument;

}

public override void OnClick()
{
    IMap map = m_mxDoc.FocusMap;

    // Create a delegate instance and add it to SelectionChanged event.
    m_selectionChanged = new
IActiveViewEvents_SelectionChangedEventHandler(SelectionChanged);
    ((IActiveViewEvents_Event)map).SelectionChanged += m_selectionChanged;
}
// Event handler
private void SelectionChanged()
{
    MessageBox.Show("Selection changed");
}
}
```

Error handling

The error handling construct in Visual Studio .NET is known as structured exception handling. The constructs used may be new to Visual Basic users but should be familiar to users of C++ or Java.

Structured exception handling is straightforward to implement, and the same concepts are applicable to either VB .NET or C#. VB .NET allows backward compatibility by also providing unstructured exception handling, via the familiar *On Error GoTo* statement and *Err* object, although this model is not discussed in this section.

Exceptions

Exceptions are used to handle error conditions in Visual Studio .NET. They provide information about the error condition.

An exception is an instance of a class that inherits from the `System.Exception` base class. Many different types of exception classes are provided by the .NET Framework, and it is also possible to create your own exception classes. Each type extends the basic functionality of the `System.Exception` class by allowing further access to information about the specific type of error that has occurred.

An instance of an Exception class is created and thrown when the .NET Framework encounters an error condition. You can deal with exceptions by using the `Try`, `Catch`, `Finally` construct.

Try, Catch, Finally

This construct allows you to catch errors that are thrown within your code. An example of this construct is shown below. An attempt is made to rotate an envelope, which throws an error.

[VB.NET]

```
Dim env As IEnvelope = New EnvelopeClass()
env.PutCoords(0D, 0D, 10D, 10D)
Dim trans As ITransform2D = env
trans.Rotate(env.LowerLeft, 1D)
Catch ex As System.Exception
    MessageBox.Show("Error: " + ex.Message)

    ' Perform any tidy up code.
End Try
```

[C#]

```
{
    IEnvelope env = new EnvelopeClass();
    env.PutCoords(0D, 0D, 10D, 10D);
    ITransform2D trans = (ITransform2D) env;
    trans.Rotate(env.LowerLeft, 1D);
}
catch (System.Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}

{
    // Perform any tidy up code.
}
}
```

You place a `try` block around code that may fail. If the application throws an error within the `Try` block, the point of execution will switch to the first `Catch` block.

The `Catch` block handles a thrown error. The application executes the `Catch` block when the `Type` of a thrown error matches the `Type` of error specified by the `Catch` block. You can have more than one `Catch` block to handle different kinds of errors. The code shown below checks first if the exception thrown is a *DivideByZeroException*.

```
[VB.NET]
...
Catch divEx As DivideByZeroException
    ' Perform divide by zero error handling.
Catch ex As System.Exception
    ' Perform general error handling.
...
```

```
[C#]
...
catch (DivideByZeroException divEx)
{
    // Perform divide by zero error handling.
}
catch (System.Exception ex)
{
    // Perform general error handling.
}
...
```

If you do have more than one Catch block, note that the more specific exception, Types, should precede the general System.Exception, which will always succeed the type check.

The application always executes the Finally block, either after the Try block completes, or after a Catch block, if an error was thrown. The Finally block should, therefore, contain code that must always be executed, for example, to clean up resources such as file handles or database connections.

If you do not have any cleanup code, you do not need to include a Finally block.

Code without exception handling

If a line of code not contained in a Try block throws an error, the .NET run time searches for a Catch block in the calling function, continuing up the call stack until a Catch block is found.

If no Catch block is specified in the call stack at all, the exact outcome may depend on the location of the executed code and the configuration of the .NET run time. Therefore, it is advisable to include at least a Try, Catch, Finally construct for all entry points to a program.

Errors from COM components

The structured exception handling model differs from the HRESULT model used by COM. C++ developers can easily ignore an error condition in an HRESULT if they want; in Visual Basic 6, however, an error condition in an HRESULT populates the *Err* object and raises an error.

The .NET run time's handling of errors from COM components is somewhat similar to the way COM errors were handled at VB6. If a .NET program calls a

function in a COM component (through the COM interop services) and returns an error condition as the HRESULT, the HRESULT is used to populate an instance of the *COMException* class. This is then thrown by the .NET run time, where you can handle it in the usual way, by using a Try, Catch, Finally block.

Therefore, it is advisable to enclose all code that may raise an error in a COM component within a Try block with a corresponding Catch block to catch a *COMException*. Below is the first example rewritten to check for an error from a COM component.

[VB.NET]

```
Dim env As IEnvelope = New EnvelopeClass()
env.PutCoords(0D, 0D, 10D, 10D)
Dim trans As ITransform2D = env
trans.Rotate(env.LowerLeft, 1D)
Catch COMex As COMException
    If (COMex.ErrorCode = -2147220984) Then
        MessageBox.Show("You cannot rotate an Envelope")

        MessageBox.Show _
            ("Error " + COMex.ErrorCode.ToString() + ": " + COMex.Message)
    End If
Catch ex As System.Exception
    MessageBox.Show("Error: " + ex.Message)
...

```

[C#]

```
{
    IEnvelope env = new EnvelopeClass();
    env.PutCoords(0D, 0D, 10D, 10D);
    ITransform2D trans = (ITransform2D) env;
    trans.Rotate(env.LowerLeft, 1D);
}
catch (COMException COMex)
{
    if (COMex.ErrorCode == -2147220984)
        MessageBox.Show("You cannot rotate an Envelope");

        MessageBox.Show ("Error " + COMex.ErrorCode.ToString() + ": " +
COMex.Message);
}
catch (System.Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
...

```

The *COMException* class belongs to the System.Runtime.InteropServices namespace. It provides access to the value of the original HRESULT via the *ErrorCode* property, which you can test to find out which error condition occurred.

Throwing errors and the exception hierarchy

If you are coding a user interface, you may want to attempt to correct the error condition in code and try the call again. Alternatively, you may want to report the error to the user to let them decide which course of action to take; here you can make use of the Message property of the Exception class to identify the problem.

However, if you are writing a function that is only called from other code, you may want to deal with an error by creating a specific error condition and propagating this error to the caller. You can do this using the Throw keyword.

To throw the existing error to the caller function, write your error handler using the Throw keyword, as shown below.

```
[VB.NET]
Catch ex As System.Exception
...

```

```
[C#]
catch (System.Exception ex)
{
    throw;
}
...

```

If you wish to propagate a different or more specific error back to the caller, you should create a new instance of an Exception class, populate it appropriately, and throw this exception back to the caller. The example shown below uses the *ApplicationException* constructor to set the Message property.

```
[VB.NET]
Catch ex As System.Exception
    Throw New ApplicationException _
        ("You had an error in your application")
...

```

```
[C#]
catch (System.Exception ex)
{
    throw new ApplicationException("You had an error in your application");
}
...

```

If you do this, however, the original exception is lost. To allow complete error information to be propagated, the Exception class includes the InnerException property. This property should be set to equal the caught exception, before the new exception is thrown. This creates an error hierarchy. Again, the example shown below uses the ApplicationException constructor to set the InnerException and Message properties.

```
[VB.NET]
Catch ex As System.Exception
    Dim appEx As System.ApplicationException = _
        New ApplicationException("You had an error in your application", ex)

```



```
        Throw appEx
    ...

    [C#]
    catch (System.Exception ex)
    {
        System.ApplicationException appEx =
            new ApplicationException("You had an error in your application", ex);
        throw appEx;
    }
    ...
```

In this way, the function that eventually deals with the error condition can access all the information about the cause of the condition and its context.

If you throw an error, the application will execute the current function's Finally clause before control is returned to the calling function.

Working with resources

Using strings and embedded images directly (no localization)

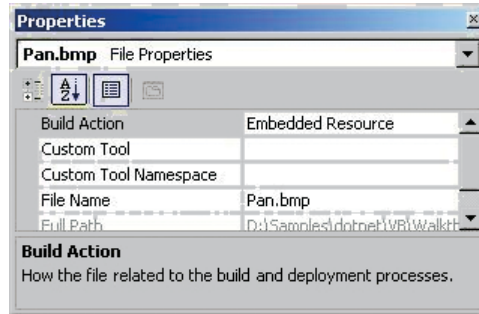
If your customization does not support localization now, and you do not intend for it to support localization later, you can use strings and images directly without the need for resource files. For example, strings can be specified and used directly in your code:

```
[VB.NET]
Me.TextBox1.Text = "My String"
```

```
[C#]
this.textBox1.Text = "My String";
```

Image files (BMPs, JPEGs, PNGs, and so forth) can be embedded in your assembly as follows:

1. Right-click the Project in the Solution Explorer, click Add, then click Add Existing Item.
2. In the Add Existing Item dialog box, browse to your image file and click Open.
3. In the Solution Explorer, select the image file you just added, then press F4 to display its properties.
4. Set the Build Action property to Embedded Resource.



Now you can reference the image in your code. For example, the following code creates a bitmap object from the first embedded resource in the assembly:

[VB.NET]

```
Dim res() As String = GetType(Form1).Assembly.GetManifestResourceNames()  
If (res.GetLength(0) > 0)  
    Dim bmp As System.Drawing.Bitmap = New System.Drawing.Bitmap(_  
        GetType(Form1).Assembly.GetManifestResourceStream(res(0)))  
    ...
```

[C#]

```
string[] res = GetType().Assembly.GetManifestResourceNames();  
if (res.GetLength(0) > 0)  
{  
    System.Drawing.Bitmap bmp = new System.Drawing.Bitmap(  
        GetType().Assembly.GetManifestResourceStream(res[0]));  
    ...
```

Creating resource files

Before attempting to provide localized resources, you should ensure you are familiar with the process of creating resource files for your .NET projects. Even if you do not intend to localize your resources, you can still use resource files instead of using images and strings directly, as described above.

Visual Studio .NET projects use an XML-based file format to contain managed resources. These XML files have the extension .resx and can contain any kind of data (images, cursors, and so forth) so long as the data is converted to ASCII format. RESX files are compiled to .resources files, which are binary representations of the resource data. Binary .resources files can be embedded by the compiler into either the main project assembly or a separate satellite assembly that contains only resources.

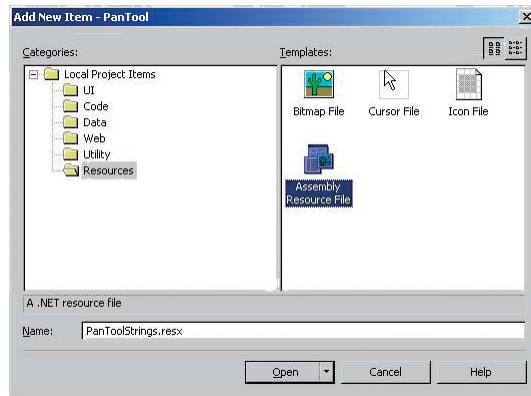
The following options are available to create your resource files. Each is discussed below.

- Creating a .resx file for string resources
- Creating resource files for image resources
- Compiling a .resx file into a .resources file

Creating a .resx file for string resources

If all you need to localize is strings—not images or cursors—you can use Visual Studio.NET to create a new .resx file that will be compiled automatically into a .resources module embedded in the main assembly.

1. Right-click the Project name in the Solution Explorer, click Add, then click Add New Item.
2. In the Add New Item dialog box, click Assembly Resource File.



3. Open the new .resx file in Visual Studio, and add name–value pairs for the culture-specific strings in your application.

Data for data					
	name	value	comment	type	mimetype
	Pan_Categor	Developer Samples	(null)	(null)	(null)
	Pan_Message	Move around the display by dragging	(null)	(null)	(null)
▶	Pan_Caption	Pan C#	(null)	(null)	(null)
*					

4. When you compile your project, the .resx file will be compiled into a .resources module inside your main assembly.

Creating resource files for image resources

The process of adding images, icons, or cursors to a resources file in .NET is more complex than creating a file containing only string values, because the tools currently available in the Visual Studio .NET IDE can only be used to add string resources.

However, a number of sample projects are available with the Visual Studio .NET Framework SDK that can help you work with resource files. One such sample is the Resource Editor (ResEditor).

A list of tools useful for working with resources can be found in the Microsoft .NET Framework documentation.

Additional information on the ResEditor sample can be found in the Microsoft .NET Framework documentation.

The ResEditor sample is provided by Microsoft as source code. You must build the sample first if you want to create resource files using this tool. You can find information on building the SDK samples under the SDK subdirectory of your Visual Studio .NET installation.

The ResEditor sample can be used to add images, icons, imagelists, and strings to a resource file. The tool cannot be used to add cursor resources. Files can be saved as either .resx or .resource files.



Creating resource files programmatically

You can create XML .resx files containing resources programmatically by using the *ResXResourceWriter* class (part of the .NET framework). You can create binary .resources files programmatically by using the *ResourceWriter* class (also part of the .NET framework). These classes will allow more flexibility to add the kind of resources you require.

These classes may be particularly useful if you want to add resources that cannot be handled by the .NET Framework SDK samples and tools, for example, cursors. The basic usage of the two classes is similar: first, create a new resource writer class specifying the filename, then add resources individually by using the *AddResource* method.

The code below demonstrates how you could create a new .resx file using the *ResXResourceWriter* class and add a bitmap and cursor to the file.

[VB.NET]

```
Dim img As System.Drawing.Image = CType(New  
System.Drawing.Bitmap("ABitmap.bmp"), System.Drawing.Image)  
Dim cur As New System.Windows.Forms.Cursor("Pencil.cur")
```

```
Dim rsxw As New System.Resources.ResXResourceWriter("en-AU.resx")  
rsxw.AddResource("MyBmp_jpg", img)  
rsxw.AddResource("Mycursor_cur", cur)  
rsxw.Close()
```

```
[C#]
System.Drawing.Image img = (System.Drawing.Bitmap) new
System.Drawing.Bitmap("ABitmap.bmp");
System.Windows.Forms.Cursor cur = new
System.Windows.Forms.Cursor("Pencil.cur");

System.Resources.ResXResourceWriter rsxw = new
System.Resources.ResXResourceWriter("en-GB.resx");
rsxw.AddResource("MyBmp_jpg", img);
rsxw.AddResource("Mycursor_cur", cur);
rsxw.Close();
```

The PanTool developer sample (Samples\Map Analysis\Tools) includes a script—*MakeResources*—that shows you how to use the *ResXResourceWriter* class to write bitmap, cursor files, and strings into a .resx file. It also shows you how to read from a .resx file using the *ResXResourceReader* class. The sample includes a .resx file that holds a bitmap, two cursors, and three strings.

Compiling a .resx file into a .resources file

XML-based .resx files can be compiled to binary .resources files either by using the Visual Studio IDE or the ResX Generator (ResXGen) sample in the tutorial.

- Any .resx file included in a Visual Studio project will be compiled to a .resources module when the project is built. See the ‘Using resources with localization’ section below for more information on how multiple resource files are used for localization.
- You can convert a .resx file into a .resources file independently of the build process using the .NET Framework SDK command `resgen`, for example:
`resgen PanToolCS.resx PanToolCS.resources`

More information on the ResXGen can be found in the Microsoft .NET Framework documentation.

Using resources with localization

This section explains how you can localize resources for your customizations.

How to use resources with localization

In .NET, a combination of a specific Language and Country/Region is called a *culture*. For example, the American dialect of English is indicated by the string “en-US”, and the Swiss dialect of French is indicated by “fr-CH”.

If you want your project to support various cultures (languages and dialects), you should construct a separate .resources file containing culture-specific strings and images for each culture.

When you build a .NET project that uses resources, .NET embeds the default .resources file in the main assembly. Culture-specific .resources files are compiled into satellite assemblies (using the naming convention `<Main Assembly Name>.resources.dll`) and placed in subdirectories of the main build directory. The subdirectories are named after the culture of the satellite assembly they contain. For example, Swiss–French resources would be contained in a fr-CH subdirectory.

The Visual Basic .NET and C# flavors of the Pan Tool developer sample illustrate how to localize resources for German language environments.

The sample can be found in the Developer Samples\ArcMap\Commands and Tools\Pan Tool folder. Strictly speaking, the sample only requires localized strings, but the images have been changed for the “de” culture as well, to serve as illustration.

A batch file named buildResources.bat has been provided in the Pan Tool sample to create the default .resources files and the culture-specific satellite assemblies.

When an application runs, it automatically uses the resources contained in the satellite assembly with the appropriate culture. The appropriate culture is determined from the Windows settings. If a satellite assembly for the appropriate culture cannot be found, the default resources (those embedded in the main assembly) will be used instead.

The following sections give more information on creating your own .resx and .resources files.

Embedding a default .Resources file in your project

1. Right-click the Project name in the Solution Explorer, click Add, then click Add Existing Item to navigate to your .resx or .resources file.
2. In the Solution Explorer, choose the file you just added and click F4 to display its properties.
3. Set the Build Action property to Embedded Resource.

This will ensure that your application always has a set of resources to fall back on if there isn't a resource DLL for the culture your application runs in.

Creating .Resources.dll files for cultures supported by your project

1. First, ensure you have a default .resx or .resources file in your project.
2. Take the default .resx or .resources file and create a separate localized file for each culture you want to support.
 - Each file should contain resources with the same Names; the Value of each resource in the file should contain the localized value.
 - Localized resource files should be named according to their culture, for example, <BaseName>.<Culture>.resx or <BaseName>.<Culture>.resources.
3. Add the new resource files to the project, ensuring each one has its Build Action set to Embedded Resource.
4. Build the project.

The compiler and linker will create a separate satellite assembly for each culture. The satellite assemblies will be placed in subdirectories under the directory holding your main assembly. The subdirectories will be named by culture, allowing the .NET run time to locate the resources appropriate to the culture in which the application runs.

The main (default) resources file will be embedded in the main assembly.

Assembly versioning and redirection

Applications that are built using a specific version of a strongly named assembly require the same assembly at run time. For example, if you create an application that uses ESRI.ArcGIS.System version 9.0.452, you will not be able to run this application on a system that has a newer version of ESRI.ArcGIS.System (for example, 9.0.0.692) installed. This may be the case if someone has installed a newer version of ArcGIS; however, using configuration files you can redirect an application to use a newer version of an assembly.

You have two choices for redirecting assemblies:

- Application configuration files
- Machine configuration files

Application configuration files

Application configuration files contain settings specific to an application. This file contains configuration settings that the common language run time reads, such as assembly binding policy or remoting objects, and settings that the application can read.

The name and location of the application configuration file depend on the application's host, which can be one of the following:

- Executable-hosted application—The configuration file for an application hosted by the executable host is in the same directory as the application. The name of the configuration file is the name of the application with a .config extension. For example, an application called myApp.exe can be associated with a configuration file called myApp.exe.config.
- ASP.NET-hosted application—ASP.NET configuration files are called Web.config. Configuration files in ASP.NET applications inherit the settings of configuration files in the URL path. For example, given the URL `www.esri.com/aaa/bbb`, where `www.esri.com/aaa` is the Web application, the configuration file associated with the application is located at `www.esri.com/aaa`. ASP.NET pages that are in the subdirectory `/bbb` use both the settings that are in the configuration file at the application level and the settings in the configuration file that are in `/bbb`.
- Internet Explorer-hosted application—If an application hosted in Internet Explorer has a configuration file, the location of this file is specified in a `<link>` tag with the following syntax:

```
<link rel="ConfigurationFileName" href="location">
```

In this tag, *location* is a URL to the configuration file. This sets the application base. The configuration file must be located on the same Web site as the application.

Machine configuration files

The machine configuration file, `Machine.config`, contains settings that apply to an entire computer. This file is located in the `%runtime install path%\Config` directory. `Machine.config` contains configuration settings for machine-wide assembly binding, built-in remoting channels, and ASP.NET.

The configuration system first looks in the machine configuration file for the `<appSettings>` element and other configuration sections that a developer might define. It then looks in the application configuration file. To keep the machine configuration file manageable, it is best to put these settings in the application configuration file. However, putting the settings in the machine configuration file can make your system more maintainable. For example, if you have a third-party component that both your client and server application use, it is easier to put the settings for that component in one place. In this case, the machine configuration file is the appropriate place for the settings, so you don't have the same settings in two different files.

Deploying an application using XCOPY will not copy the settings in the machine configuration file.

The configuration file below shows how to bind to an assembly and redirect it to a newer version.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="ESRI.ArcGIS.System"
          publicKeyToken="8fc3cc631e44ad86"
          culture="neutral" />
        <!-- Assembly versions can be redirected in application,
          publisher policy, or machine configuration files. -->
        <bindingRedirect oldVersion="9.0.0.452"
          newVersion="9.0.0.692"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

ARC GIS DEVELOPMENT USING .NET

Using .NET, you can customize the ArcGIS applications, create standalone applications that use ESRI's types, and extend ESRI's types. For example, you can create a custom tool for ArcMap, create a standalone application that uses the MapControl, or create a custom layer. This section discusses several key issues related to developing with ArcGIS and .NET.

Registering .NET components with COM

Extending ArcGIS applications with custom .NET components requires registering the components in the COM registry and exporting the .NET assemblies to a type library (TLB). When developing a component, there are two ways to perform this task: you can use the RegAsm utility that ships with the .NET Framework SDK or Visual Studio.NET, which has a Register for COM Interop compiler flag.

The example below shows an EditTools assembly being registered with COM. The /tlb parameter specifies that a type library should additionally be generated and the /codebase option indicates that the path to the assembly should be included in the registry settings. Both of these parameters are required when extending the ArcGIS applications with .NET components.

```
regasm EditTools.dll /tlb:EditTools.tlb /codebase
```

Visual Studio.NET performs this same operation automatically if you set the Register for COM Interop compiler flag; this is the simplest way to perform the registration on a development machine. To check a project's settings, click Project Properties from the Project menu, then look at the Build property under Configuration Properties. The last item, Register for COM Interop, should be set to True.

Registering .NET classes in COM component categories

Much of the extensibility of ArcGIS relies on COM component categories. In fact, most custom ArcGIS components must be registered in component categories appropriate to their intended context and function for the host application to

make use of their functionality. For example, all ArcMap commands and tools must be registered in the *ESRI Mx Commands* component category. There are a few different ways you can register a .NET component in a particular category but before doing so, the .NET components must be registered with COM. See the 'Registering .NET components with COM' section above for details.

Customize dialog box

Custom .NET ArcGIS commands and tools can quickly be added to toolbars via the Add From File button on the Customize dialog box. In this case, you simply have to browse for the TLB and open it. The ArcGIS framework will automatically add the classes you select in the type library to the appropriate component category.

Categories utility

Another option is to use the Component Categories Manager (Categories.exe). In this case, you select the desired component category in the utility, browse for your type library, and choose the appropriate class.

COM Register Function

The final and recommended solution is to add code to your .NET classes that will automatically register them in a particular component category whenever the component is registered with COM. The .NET Framework contains two attribute classes (*ComRegisterFunctionAttribute* and *ComUnregisterFunctionAttribute*) that allow you to specify methods that will be called whenever your component is being registered or unregistered. Both methods are passed the CLSID of the class currently being registered. With this information you can write code inside the methods to make the appropriate registry entries or deletions. Registering a component in a component category requires that you also know the component category's unique ID (CATID).

The code excerpt below shows a custom ArcMap command that automatically registers itself in the *MxCommands* component category whenever the .NET assembly in which it resides is registered with COM.

```
public sealed class AngleAngleTool: BaseTool
{

    [ComRegisterFunction()]
    static void Reg(String regKey)
    {
        Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(regKey.
Substring(18)+ "\\Implemented Categories\\" + "{B56A7C42-83D4-11D2-A2E9-
080009B6F22B}");
    }
    [ComUnregisterFunction()]
    static void Unreg(String regKey)
    {
        Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey(regKey.Substring(18)+
"\\Implemented Categories\\" + "{B56A7C42-83D4-11D2-A2E9-080009B6F22B}");
    }
}
```

To simplify this process, ESRI provides classes for each component category ArcGIS exposes with static functions to register and unregister components. Each class knows the GUID of the component category it represents, so registering custom components is greatly simplified. For more details on using these classes, see the ‘Working with the ESRI .NET component category classes’ section below.

Simplifying your code using the ESRI.ArcGIS.Utility assembly

Part of the ArcGIS Developer Kit includes a number of .NET utility classes that facilitate .NET development by taking advantage of a few .NET capabilities including object inheritance and static functions.

Working with the ESRI .NET Base Classes

ESRI provides two abstract base classes (*BaseCommand* and *BaseTool*) to help you create new custom commands and tools for ArcGIS. The classes are abstract classes (marked as *MustInherit* in Visual Basic .NET), which means that although the class may contain some implementation code, it cannot itself be instantiated directly and can only be used by being inherited by another class. Both base classes are defined in the *ESRI.ArcGIS.Utility* assembly and belong to the *ESRI.ArcGIS.Utility.BaseClasses* namespace.

These base classes simplify the creation of custom commands and tools by providing a default implementation for each of the members of *ICommand* and *ITool*. Instead of stubbing out each member and providing implementation code, you only have to override the members that your custom command or tool requires. The exception is *ICommand::OnCreate*; this member must be overridden in your derived class.

Using these base classes is the recommended way to create commands and tools for ArcGIS applications in .NET languages. You can create similar COM classes from first principles; however, you should find the base class technique to be a quicker, simpler, less error-prone method of creating commands and tools.

Syntax

Both base classes have an overloaded constructor, allowing you to quickly set many of the properties of a command or tool, such as *Name* and *Category*, via constructor parameters.

The overloaded *BaseCommand* constructor has the following signature:

```
[VB.NET]
Public Sub New( _
    ByVal bitmap As System.Drawing.Bitmap _
    ByVal caption As String _
    ByVal category As String _
    ByVal helpContextId As Integer _
    ByVal helpFile As String _
    ByVal message As String _
    ByVal name As String _
    ByVal tooltip As String)
```

```
[C#]
public BaseCommand(
    System.Drawing.Bitmap bitmap,
    string caption,
    string category,
    int helpContextId,
    string helpFile,
    string message,
    string name,
    string tooltip,
);
```

The overloaded BaseTool constructor has the following signature:

```
[VB.NET]
Public Sub New( _
    ByVal bitmap As System.Drawing.Bitmap _
    ByVal caption As String _
    ByVal category As String _
    ByVal cursor As System.Windows.Forms.Cursor _
    ByVal helpContextId As Integer _
    ByVal helpFile As String _
    ByVal message As String _
    ByVal name As String _
    ByVal tooltip As String _
)
```

```
[C#]
public BaseTool(
    System.Drawing.Bitmap bitmap,
    string caption,
    string category,
    System.Windows.Forms.Cursor cursor,
    int helpContextId,
    string helpFile,
    string message,
    string name,
    string tooltip,
);
```

Inheriting the base classes

You can use these parameterized constructors when you write your new classes, for example, as shown below for a new class called PanTool that inherits the BaseTool class.

```
[VB.NET]
Public Sub New()
    MyBase.New( Nothing, "Pan", "My Custom Tools", _
        System.Windows.Forms.Cursors.Cross, 0, "", "Pans the map.",
        "PanTool", "Pan")
End Sub
```

```
[C#]
```

```
public PanTool() : base ( null, "Pan", "My Custom Tools",
    System.Windows.Forms.Cursors.Cross, 0, "", "Pans the map.", "PanTool",
    "Pan")
{
    ...
}
```

Setting base class members directly

As an alternative to using the parameterized constructors, you can set the members of the base class directly.

The base classes expose their internal member variables to the inheritor class, one per property, so you can directly access them in your derived class. For example, instead of using the constructor to set the Caption or overriding the Caption function, you can set the `m_caption` class member variable declared in the base class.

```
[VB.NET]
Public Sub New()
    MyBase.New()
    MyBase..m_bitmap = New
    System.Drawing.Bitmap(GetType().Assembly.GetManifestResourceStream("Namespace.Pan.bmp"))
    MyBase..m_cursor = System.Windows.Forms.Cursors.Cross
    MyBase..m_category = "My Custom Tools"
    MyBase..m_caption = "Pan"
    MyBase..m_message = "Pans the map."
    MyBase..m_name = "PanTool"
    MyBase..m_toolTip = "Pan"
End Sub
```

```
[C#]
public PanTool()
{
    base.m_bitmap = new
    System.Drawing.Bitmap(GetType().Assembly.GetManifestResourceStream("Namespace.Pan.bmp"));
    base.m_cursor = System.Windows.Forms.Cursors.Cross;
    base.m_category = "My Custom Tools";
    base.m_caption = "Pan";
    base.m_message = "Pans the map.";
    base.m_name = "PanTool";
    base.m_toolTip = "Pan";
}
```

Overriding members

When you create custom commands and tools that inherit a base class, you will more than likely need to override a few members. When you override a member in your class, the implementation code that you provide for that member will be executed instead of the default member implementation inherited from the base class. For example, the `OnClick` method in the `BaseCommand` has no implementation code at all, as `OnClick` will not do anything by default. This may be suitable for a tool, but is probably not for a command.

To override any member, you can right-click the member of the base class in the Solution Explorer Window, click Add, then click Override to stub out the mem-

ber as overridden. Note that if you right-click the member of the underlying interface (*ICommand* or *ITool*) instead of the base class member, the overridden member will not include the overrides keyword, and the method will instead be shadowed.

```
[VB.NET]
Public Overrides Sub OnClick()
    ' Your OnClick
End Sub
```

```
[C#]
public override void OnClick()
{
    // Your OnClick
}
```

Alternatively, to override a member of the base class, click Overrides from the dropdown list on the right on the Code Window Wizard bar, then choose the member you want to override from the left dropdown list. This will stub out the member as overridden.

What do the base classes do by default?

The table below shows the base class members that have a significant base class implementation, along with a description of that implementation. Override these members when the base class behavior is not consistent with your customization. For example, Enabled is set to True by default; if you want your custom command enabled only when a specific set of criteria has been met, you must override this property in your derived class.

Member	Description
ICommand::Bitmap	The given bitmap is made transparent based on the pixel value at position 1,1. The bitmap is null until set by the derived class.
ICommand::Category	If null, sets the category "Misc."
ICommand::Checked	Set to False.
ICommand::Enabled	Set to True.
ITool::OnContextMenu	Set to False.
ITool::Deactivate	Set to True.

Working with the ESRI .NET component category classes

To help register .NET components in COM component categories, ESRI provides the ESRI.ArcGIS.Utility.CATIDs namespace, which has classes that represent each of the ArcGIS component categories. Each class knows its CATID and exposes static methods (Register and Unregister) for adding and removing components. Registering your component becomes as easy as adding COM registration methods with the appropriate attributes and passing the received CLSID to the appropriate static method.

The example below shows a custom Pan tool that registers itself in the ESRI Mx

Commands component category. Notice in this example `MxCommands.Register` and `MxCommands.Unregister` are used instead of `Microsoft.Win32.Registry.ClassesRoot.CreateSubKey` and `Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey`.

[VB.NET]

```
Public NotInheritable Class PanTool
    Inherits BaseTool

    <ComRegisterFunction()> _
    Public Shared Sub Reg(ByVal regKey As [String])
        MxCommands.Register(regKey)
    End

    <ComUnregisterFunction()> _
    Public Shared Sub Unreg(ByVal regKey As [String])
        MxCommands.Unregister(regKey)
    End Sub
```

[C#]

```
public sealed class PanTool : BaseTool
{
    [ComRegisterFunction()]
    static void Reg(string regKey)
    {
        MxCommands.Register(regKey);
    }

    [ComUnregisterFunction()]
    static void Unreg(string regKey)
    {
        MxCommands.Unregister(regKey);
    }
}
```

Extending the server

When using .NET to create a COM object for use in the GIS server, there are some specific guidelines you need to follow to ensure that you can use your object in a server context and that it will perform well in that environment. The guidelines below apply specifically to COM objects you create to run within the server.

- You must explicitly create an interface that your COM class implements. Unlike Visual Basic 6, .NET will not create an implicit interface for your COM class that you can use when creating the object in a server context.
- Your COM class should be marshalled using the Automation marshaller. You specify this by adding *AutomationProxyAttribute* to your class with a value of `true`.
- Your COM class should generate a dual class interface. You specify this by adding *ClassInterfaceAttribute* to your class with a value of `ClassInterfaceType.AutoDual`.

- To ensure that your COM object performs well in the server, it must inherit from *ServicedComponent*, which is in the *System.EnterpriseServices* assembly. This is necessary due to the current COM interop implementation of the .NET Framework.

For more details and an example of a custom Server COM object written in .NET, see Chapter 4, ‘Developing ArcGIS Server applications’ in the *ArcGIS Server Administrator and Developer Guide*.

Releasing COM references

ArcGIS Engine and ArcGIS Desktop applications

An unexpected crash may occur when a standalone application attempts to shut down. For example, an application hosting a *MapControl* with a loaded map document will crash on exit. The crashes result from COM objects hanging around longer than expected. To avoid crashes, all COM references must be unloaded prior to shutdown. To help unload COM references, a special static *Shutdown* function has been added to the *ESRI.ArcGIS.Utility* assembly. The following code excerpt shows the function in use.

[VB.NET]

```
Private Sub Form1_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    ESRI.ArcGIS.Utility.COMSupport.AOUninitialize.Shutdown()
End Sub
```

[C#]

```
private void Form1_Closing(object sender, CancelEventArgs e)
{
    ESRI.ArcGIS.Utility.COMSupport.AOUninitialize.Shutdown();
}
```

The *AOUninitialize.Shutdown* function handles most of the shutdown problems in standalone applications, but you may still experience problems as there are COM objects that require explicit releasing; in these cases, call *System.Runtime.InteropServices.Marshal.ReleaseComObject()* to decrement the reference count, allowing the application to terminate cleanly. The *StyleGallery* is one such object, and the following example documents how to handle references to this class.

[VB.NET]

```
Dim styleGallery As IStyleGallery
styleGallery = New StyleGalleryClass
MessageBox.Show(styleGallery.ClassCount)
Marshal.ReleaseComObject(styleGallery)
```

[C#]

```
IStyleGallery sg = new StyleGalleryClass() as IStyleGallery;
MessageBox.Show(sg.ClassCount.ToString());
Marshal.ReleaseComObject(sg);
```

Working with cursors in ArcGIS Server

Some objects that you can create in a server context may lock or use resources

that the object frees only in its destructor. For example, a geodatabase cursor may acquire a shared schema lock on a file-based feature class or table on which it is based or may hold onto an SDE stream.

While the shared schema lock is in place, other applications can continue to query or update the rows in the table, but they cannot delete the feature class or modify its schema. In the case of file-based data sources, such as shapefiles, update cursors acquire an exclusive write lock on the file, which will prevent other applications from accessing the file for read or write. The effect of these locks is that the data may be unavailable to other applications until all of the references on the cursor object are released.

In the case of SDE data sources, the cursor holds onto an SDE stream, and if the application has multiple clients, each may get and hold onto an SDE stream, eventually exhausting the maximum allowable streams. The effect of the number of SDE streams exceeding the maximum is that other clients will fail to open their own cursors to query the database.

Because of the above reasons, it's important to ensure that your reference to any cursor your application opens is released in a timely manner. In .NET, your reference on the cursor (or any other COM object) will not be released until garbage collection kicks in. In a Web application or Web service servicing multiple concurrent sessions and requests, relying on garbage collection to release references on objects will result in cursors and their resources not being released in a timely manner.

To ensure a COM object is released when it goes out of scope, the *WebControls* assembly contains a helper object called *WebObject*. Use the *ManageLifetime* method to add your COM object to the set of objects that will be explicitly released when the *WebObject* is disposed. You must scope the use of *WebObject* within a *Using* block. When you scope the use of *WebObject* within a using block, any object (including your cursor) that you have added to the *WebObject* using the *ManageLifetime* method will be explicitly released at the end of the using block.

The following example demonstrates this coding pattern:

[VB.NET]

```
Private Sub doSomething_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles doSomething.Click
    Dim webobj As WebObject = New WebObject
    Dim ctx As IServerContext = Nothing
    Try
        Dim serverConn As ServerConnection = New ServerConnection("doug", True)
        Dim som As IServerObjectManager = serverConn.ServerObjectManager

        ctx = som.CreateServerContext("Yellowstone", "MapServer")
        Dim mapsrv As IMapServer = ctx.ServerObject
        Dim mapo As IMapServerObjects = mapsrv
        Dim map As IMap = mapo.Map(mapsrv.DefaultMapName)

        Dim flayer As IFeatureLayer = map.Layer(0)
        Dim fClass As IFeatureClass = flayer.FeatureClass

        Dim fcursor As IFeatureCursor = fClass.Search(Nothing, True)
```



```
webobj.ManageLifetime(fcursor)

Dim f As IFeature = fcursor.NextFeature()
Do Until f Is Nothing
    ' Do something with the feature.
    f = fcursor.NextFeature()
Loop

Finally
    ctx.ReleaseContext()
    webobj.Dispose()
End Try
End Sub

[C#]
private void doSomething_Click(object sender, System.EventArgs e)
{
    using (WebObject webobj = new WebObject())
    {
        ServerConnection serverConn = new ServerConnection("doug", true);
        IServerObjectManager som = serverConn.ServerObjectManager;

        IServerContext ctx =
som.CreateServerContext("Yellowstone", "MapServer");
        IMapServer mapsrv = ctx.ServerObject as IMapServer;
        IMapServerObjects mapo = mapsrv as IMapServerObjects;
        IMap map = mapo.get_Map(mapsrv.DefaultMapName);

        IFeatureLayer flayer = map.get_Layer(0) as IFeatureLayer;
        IFeatureClass fclass = flayer.FeatureClass;

        IFeatureCursor fcursor = fclass.Search(null, true);
        webobj.ManageLifetime(fcursor);

        IFeature f = null;
        while ((f = fcursor.NextFeature()) != null)
        {
            // Do something with the feature.
        }

        ctx.ReleaseContext();
    }
}
```

The *WebMap*, *WebGeocode*, and *WebPageLayout* objects also have a *ManageLifetime* method. If you are using, for example, a *WebMap* and scope your code in a using block, you can rely on these objects to explicitly release objects you add with *ManageLifetime* at the end of the using block.

Deploying .NET ArcGIS customizations

All ArcGIS Engine and Desktop customizations require an ArcGIS installation on all client machines. The ArcGIS installation must include the ESRI primary interop assemblies, which the setup program installs in the global assembly cache. For example, deploying a standalone GIS application that only requires an ArcGIS Engine license requires an ArcGIS Engine installation on all target machines.

Standalone applications

Deploying standalone applications to either ArcGIS Engine or Desktop clients involves copying over the executable to the client machine. Copying over the executable can be as simple as using xcopy or more involved, such as creating a custom install or setup program. Note that aside from the ArcGIS primary interop assemblies and the .NET Framework assemblies, all dependencies must additionally be packaged and deployed.

ArcGIS components

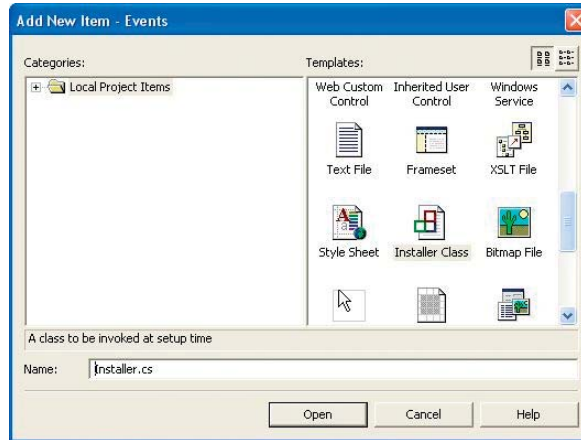
Components that extend the ArcGIS applications are trickier to deploy than standalone applications because they must be registered with COM and in specific component categories. As discussed earlier, implementing *COMRegisterFunction* and *COMUnregisterFunctions* facilitates deployment by providing self category registration, but this only occurs when the components are registered.

There are two techniques for registering components with COM. One option is to run the register assembly utility (RegAsm.exe) that ships with the .NET Framework SDK. This is typically not a viable solution as client machines may or may not have this utility and it's difficult to automate. The second and recommended approach is to add an automatic registration step to a custom setup or install program.

The key to creating a custom install program that both deploys and registers components is the *System.Runtime.InteropServices.RegistrationServices* class. This class has the members *Register.Assembly* and *Unregister.Assembly*, which register and unregister managed classes with COM. These are the same functions the RegAsm utility uses. Using these functions inside a custom installer class along with a setup program is the complete solution.

The basic steps below outline the creation of a deployable solution. NOTE: the steps assume you are starting with a solution that already contains a project with at least one COM-enabled class.

1. In Visual Studio.NET, add a new Installer Class and name it accordingly.



Override the Install and Uninstall functions that are implemented in the Installer base class and use the RegistrationServices class's *RegisterAssembly* and *UnregisterAssembly* methods to register the components. Make sure you use the SetCodeBase flag; this indicates that the code base key for the assembly should be set in the registry.

[VB.NET]

```
Public Overrides Sub Install(ByVal stateSaver As
System.Collections.IDictionary)
    MyBase.Install(stateSaver)
    Dim regsrv As New RegistrationServices
    regsrv.RegisterAssembly(MyBase.GetType().Assembly,
AssemblyRegistrationFlags.SetCodeBase)
End Sub
```

```
Public Overrides Sub Uninstall(ByVal savedState As
System.Collections.IDictionary)
    MyBase.Uninstall(savedState)
    Dim regsrv As New RegistrationServices
    regsrv.UnregisterAssembly(MyBase.GetType().Assembly)
End Sub
End Class
```

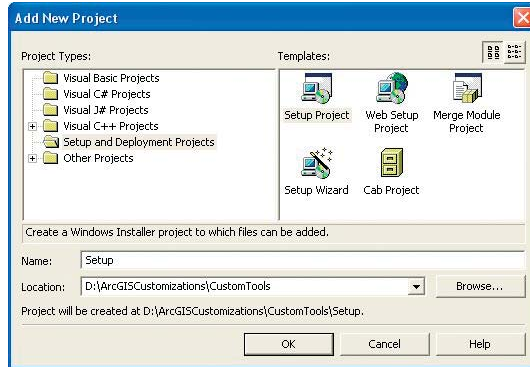
[C#]

```
public override void Install(IDictionary stateSaver)
{
    base.Install (stateSaver);
    RegistrationServices regSrv = new RegistrationServices();
    regSrv.RegisterAssembly(base.GetType().Assembly,
AssemblyRegistrationFlags.SetCodeBase);
}

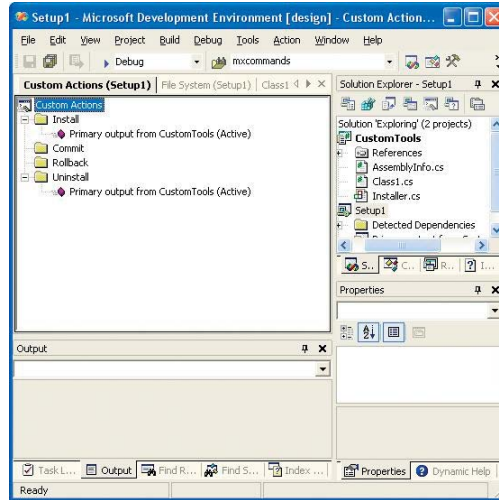
public override void Uninstall(IDictionary savedState)
{
    base.Uninstall (savedState);
}
```

```
RegistrationServices regSrv = new RegistrationServices();  
regSrv.UnregisterAssembly(base.GetType().Assembly);  
}
```

2. Add a setup program to your solution.



- a. In the Solution Explorer, right-click the new project and click Add > Project Output. Choose the project you want to deploy and choose Primary output.
- b. From the list of detected dependencies that is regenerated, remove all references to ESRI primary interop assemblies (for example, ESRI.ArcGIS.System) and stdole.dll. The only items typically left in the list are your TLB and Primary output from <AssemblyName><Version>, which represent the DLL or EXE you are compiling.
- c. The final steps involve associating the custom installation steps configured in the new installer class with the setup project. To do this, right-click the setup project in the Solution Explorer and click View Custom Actions.
- d. In the resulting view, right-click the Install folder and click Add Custom Action. Double-click the Application folder, then double-click the Primary output from the <AssemblyName><Version> item. This step associates the custom install function created earlier with the setup's custom install action.
- e. Repeat the last step for the setup's uninstall.



3. Finally, rebuild the entire solution to generate the setup executable file. Running the executable on a target machine installs the components and registers them with COM. The *COMRegisterFunction* routines then register the components in the appropriate component categories.

ArcGIS Server deployments

To deploy Web applications developed on a development server to product production servers, use the built-in Visual Studio.NET tools.

1. In the Solution Explorer, click your project.
2. Click the Project menu, then click Copy Project.
3. In the Copy Project dialog box, specify the deployment location.
4. Click OK.

In addition to copying the project, you must copy and register any related DLLs containing custom COM objects onto your Web server and all the GIS server's server object container machines.

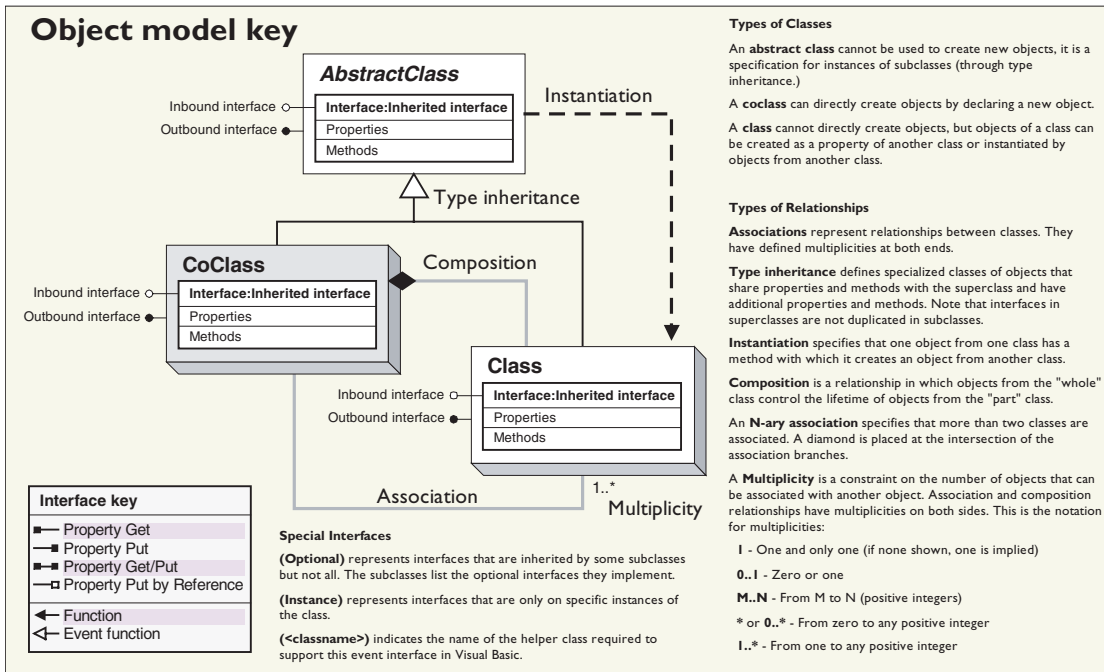
B

Reading the object model diagrams

The ArcObjects object model diagrams are an important supplement to the information you receive in object browsers. This chapter describes the diagram notation used throughout this book and in the object model diagrams that are accessed through ArcGIS Developer Help.

The diagram notation used in this book and the ArcObjects component model diagrams are based on the Unified Modeling Language (UML) notation, an industry diagramming standard for object-oriented analysis and design, with some modifications for documenting COM-specific constructs.

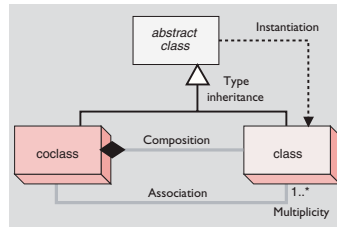
The object model diagrams are an important supplement to the information you receive in object browsers. The development environment, Visual Basic or other, lists all the classes and members but does not show the structure or relationships of those classes. These diagrams complement your understanding of the ArcObjects components.



Object model diagram key showing the types of ArcObjects and the relationships between them

CLASSES AND OBJECTS

There are three types of classes shown in the UML diagrams: abstract classes, coclasses, and classes.



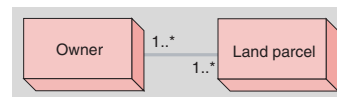
A *coclass* represents objects that you can directly create using the object declaration syntax in your development environment. In Visual Basic, this is written with the *Dim pFoo As New FooObject* syntax.

A *class* cannot directly create new objects, but objects of a class can be created as a property of another class or by functions from another class.

An *abstract class* cannot be used to create new objects; it is a specification for subclasses. An example is that a “line” could be an abstract class for “primary line” and “secondary line” classes. Abstract classes are important for developers who wish to create a subclass of their own since it shows which interfaces are required and which are optional for the type of class they are implementing. Required interfaces must be implemented on any subclass of the abstract class to ensure the new class behaves correctly in the ArcObjects system.

RELATIONSHIPS

Among abstract classes, coclasses, and classes, there are several types of class relationships possible.



In this diagram, an owner can own one or many land parcels, and a land parcel can be owned by one or many owners.

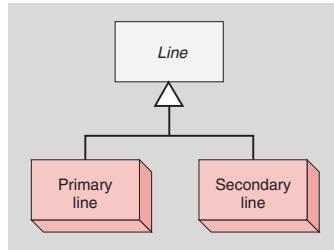
Associations represent relationships between classes. They have defined multiplicities at both ends.

A *multiplicity* is a constraint on the number of objects that can be associated with another object. This is the notation for multiplicities:

- 1—One and only one. Showing this multiplicity is optional; if none is shown, “1” is implied.
- 0..1—Zero or one
- M..N—From M to N (positive integers)
- * or 0..*—From zero to any positive integer
- 1..*—From one to any positive integer

TYPE INHERITANCE

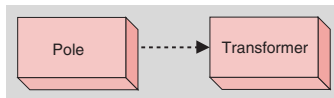
Type inheritance defines specialized classes that share properties and methods with the superclass and have additional properties and methods.



This diagram shows that a primary line (creatable class) and secondary line (creatable class) are types of a line (abstract class).

INSTANTIATION

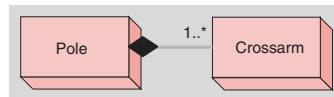
Instantiation specifies that one object from one class has a method with which it creates an object from another class.



A pole object might have a method to create a transformer object.

COMPOSITION

Composition is a stronger form of aggregation in which objects from the “whole” class control the lifetime of objects from the “part” class.



A pole contains one or many crossarms. In this design, a crossarm cannot be recycled when the pole is removed. The pole object controls the lifetime of the crossarm object.



C

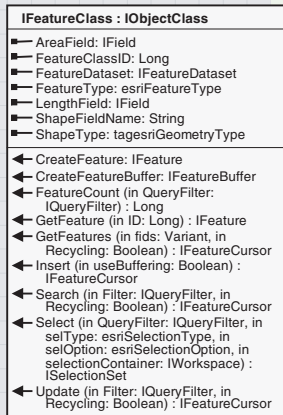
Illustrated code samples

The illustrated code samples in this appendix show the fundamentals of programming with ArcObjects. Each sample is accompanied by illustrations of the associated objects and their relationships. The code can be typed or copied into VBA, after which you can follow through with the VBA debugger.

Reading the illustrated code samples

The illustrated code samples in this section show you the fundamentals of programming with COM components in ArcObjects. Start by entering the VBA environment in ArcMap or ArcCatalog and type in the code. Step through the code in the VBA debugger. Look at these pages and study the relationships between coclasses and interfaces. A careful reading of the samples in this section gives you all the important concepts you need for developing with ArcObjects, as well as an introduction to the most important ArcObjects components.

The interface

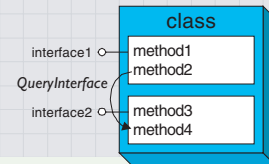


An interface is a specification of properties and methods. Many coclasses can implement the same interface. Interfaces allow a high degree of interoperability and shared behavior among a set of objects.

AreaField is a return property of type *IField*. *FeatureClassID* is of type *long*.

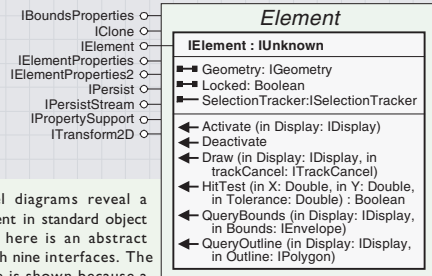
The *CreateFeature* method creates an object of type *IFeature*. *FeatureCount* takes in a query filter and returns a *long*.

QueryInterface



QueryInterface is a method in the *IUnknown* interface, which all COM objects inherit from. This method lets you query for and navigate to methods in other interfaces implemented by an object.

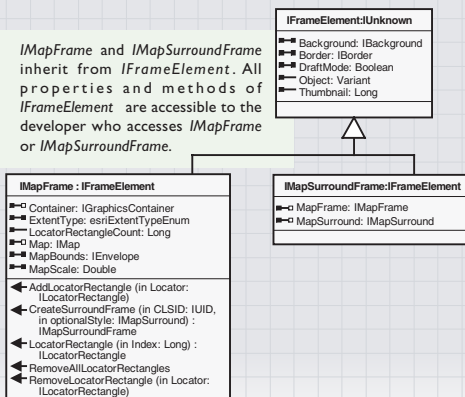
Type inheritance



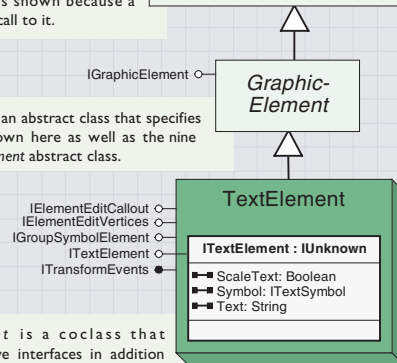
The object model diagrams reveal a structure not evident in standard object browsers. Shown here is an abstract class, *Element*, with nine interfaces. The *IElement* interface is shown because a code sample made a call to it.

Interface inheritance

IMapFrame and *IMapSurroundFrame* inherit from *IFrameElement*. All properties and methods of *IFrameElement* are accessible to the developer who accesses *IMapFrame* or *IMapSurroundFrame*.



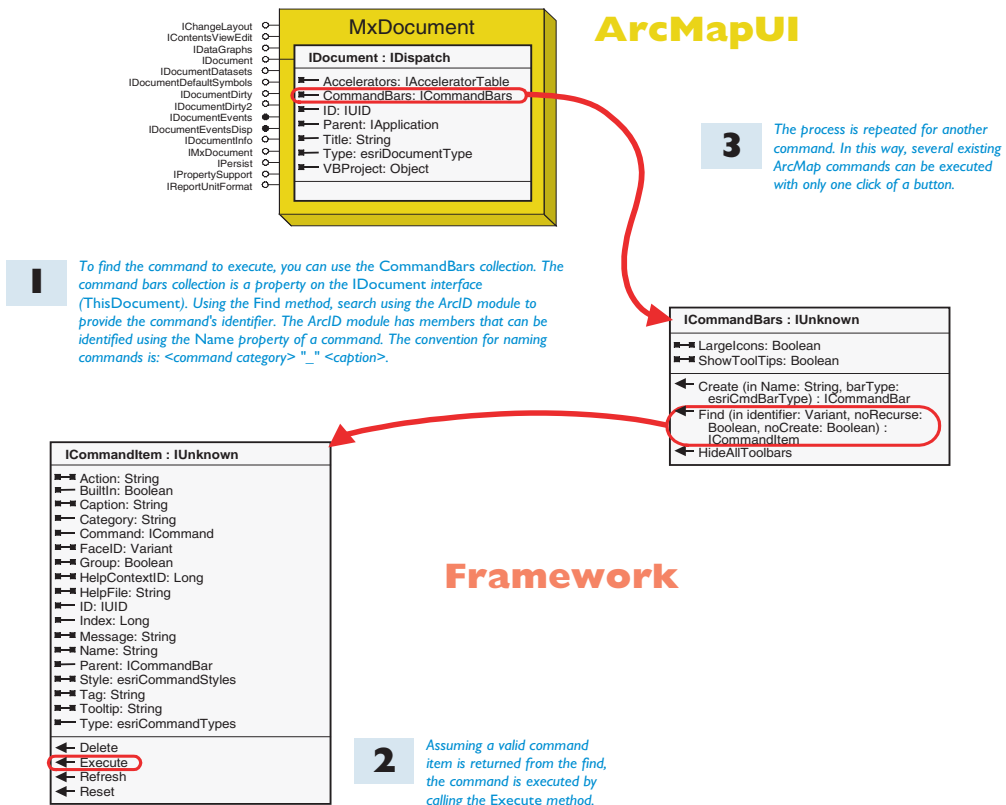
A *GraphicElement* is an abstract class that specifies the one interface shown here as well as the nine interfaces on the *Element* abstract class.



TextElement is a coclass that implements five interfaces in addition to the one from *GraphicElement* and nine from *Element*.

Locate and Execute Command on Toolbar

This sample illustrates how to programmatically execute existing commands on command bars within ArcMap.



Add this code to the Click event of a command in ArcMap.

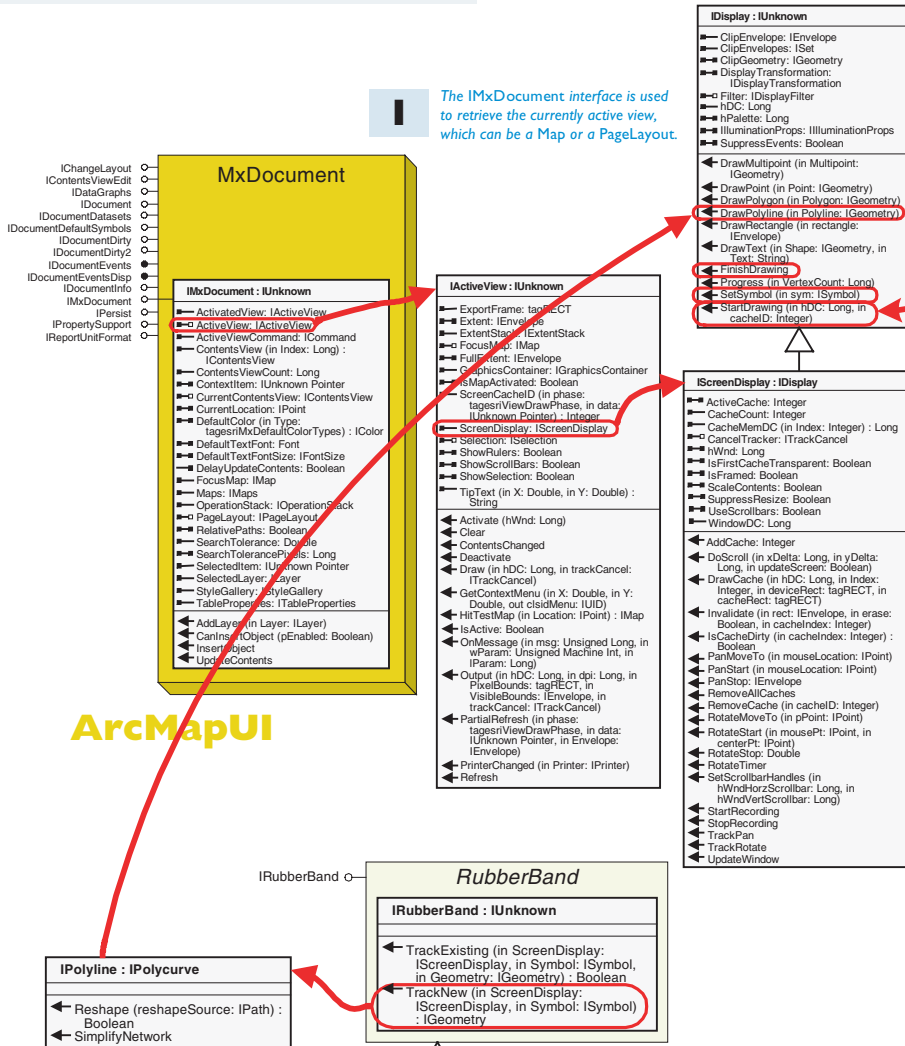
```
Dim pCommandItem As ICommandItem
```

```
1 Set pCommandItem = ThisDocument.CommandBars.Find(ArcID.Query_ZoomToSelected)
If (pCommandItem Is Nothing) Then Exit Sub
```

```
2 pCommandItem.Execute
```

```
3 Set pCommandItem = ThisDocument.CommandBars.Find(ArcID.ReportObject_CreateReport)
If (pCommandItem Is Nothing) Then Exit Sub
pCommandItem.Execute
```

This sample uses a rubber banding line to obtain a digitized line geometry. With the geometry created, a symbol is created. The symbol is set as the current display symbol and the line is drawn. The color thickness and the style of the line symbol are set.



1 The IMxDocument interface is used to retrieve the currently active view, which can be a Map or a PageLayout.

Finally, the geometry is drawn on the screen. Notice the call to start drawing followed by the setting of the symbol, and then the actual drawing of the geometry. FinishDrawing ensures the synchronization of the drawing events.

5

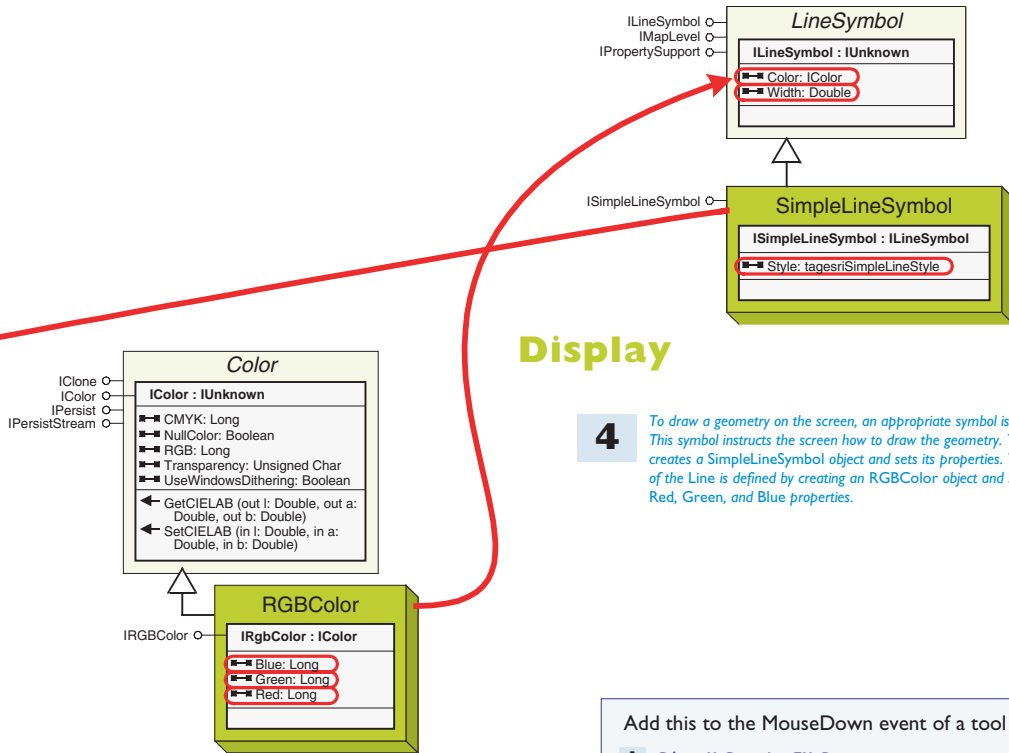
2

Since the IScreenDisplay interface of the active view is to be used frequently within the function, a local variable is used.

ArcMapUI

3

A RubberLine object is used to capture a digitized line geometry from the user. The TrackNew method takes the screen to draw to and the symbol to draw with and returns the created geometry.



Display

4

To draw a geometry on the screen, an appropriate symbol is required. This symbol instructs the screen how to draw the geometry. This step creates a SimpleLineSymbol object and sets its properties. The Color of the Line is defined by creating an RGBColor object and setting its Red, Green, and Blue properties.

Add this to the MouseDown event of a tool in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim pScreen As IScreenDisplay
  Set pScreen = pMxDoc.ActiveView.ScreenDisplay

3 Dim pPolyline As IPolyline
  Dim pRubber As IRubberBand
  Set pRubber = New RubberLine
  Set pPolyline = pRubber.TrackNew(pScreen, Nothing)

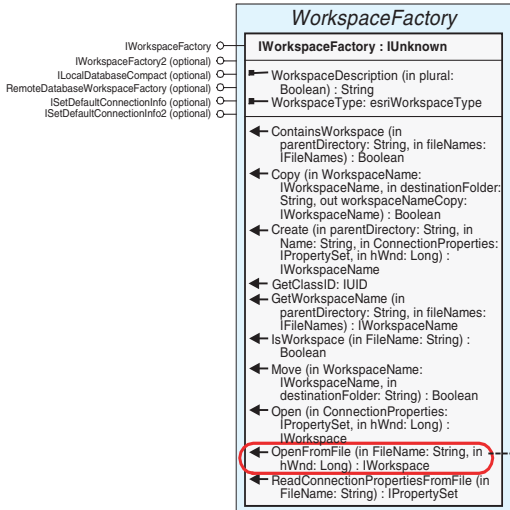
4 Dim pLineStyle As ISimpleLineStyle
  Set pLineStyle = New SimpleLineStyle

  Dim pRGBColor As IRgbColor
  Set pRGBColor = New RgbColor
  With pRGBColor
    .Red = 255
    .Green = 128
    .Blue = 128
  End With

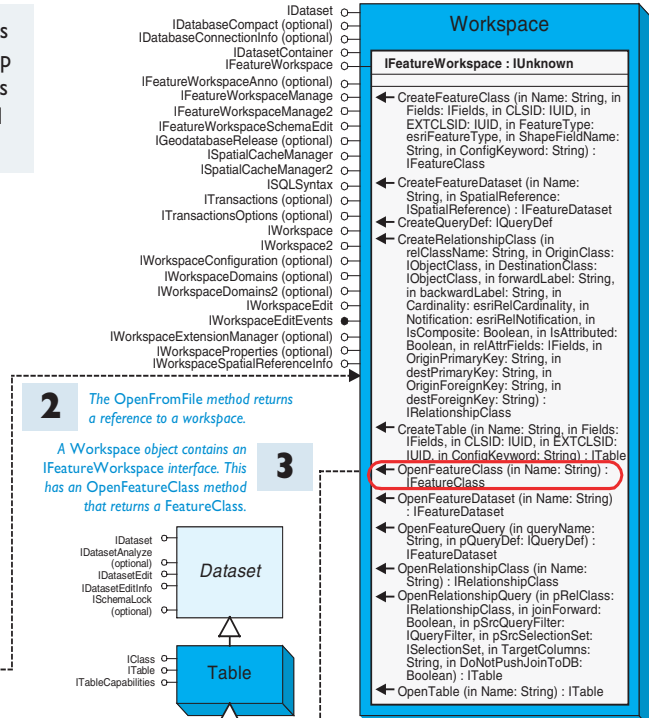
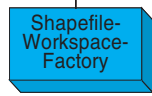
  With pLineStyle
    .Width = 2
    .Color = pRGBColor
    .Style = esriSLSSolid
  End With

5 With pScreen
  .StartDrawing pScreen.hDC, esriNoScreenCache
  .SetSymbol pLineStyle
  .DrawPolyline pPolyline
  .FinishDrawing
End With
  
```

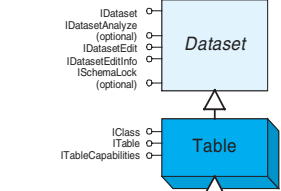
This sample opens a shapefile on the user's local disk and adds the contents to the map as a feature layer. The default symbology is used. This sample could easily be changed to support different data sources.



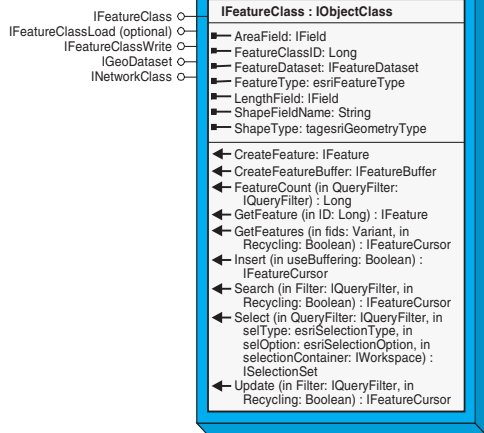
I
The ShapefileWorkspaceFactory coclass creates a shapefile workspace factory object.



2 The OpenFromFile method returns a reference to a workspace.
A Workspace object contains an IFeatureWorkspace interface. This has an OpenFeatureClass method that returns a FeatureClass.



Geodatabase



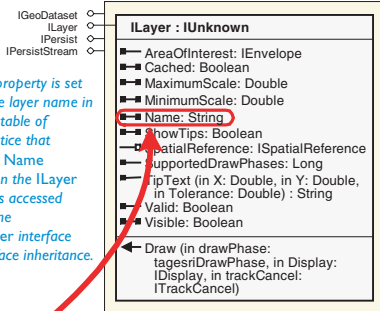
6

The IMxDocument interface is obtained from the ThisDocument global variable.

ArcMapUI

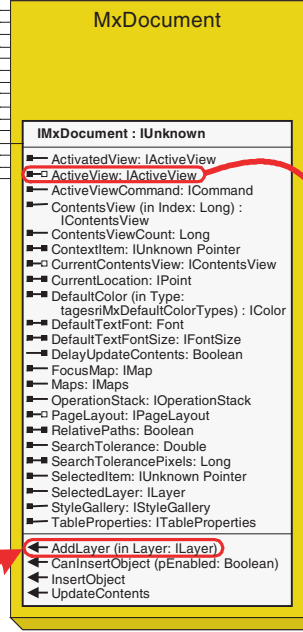
5

The Name property is set to display the layer name in the ArcMap table of contents. Notice that although the Name property is on the ILayer interface, it is accessed directly via the IFeatureLayer interface due to interface inheritance.



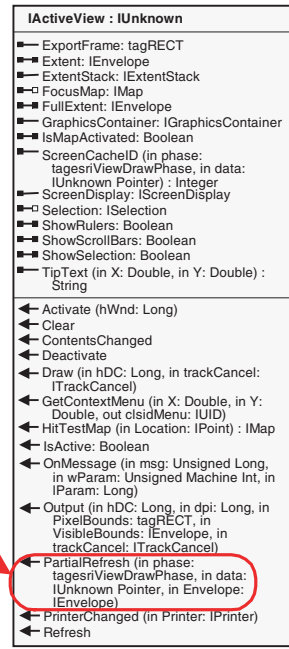
7

The AddLayer method on the IMxDocument interface adds the FeatureLayer object to ArcMap.



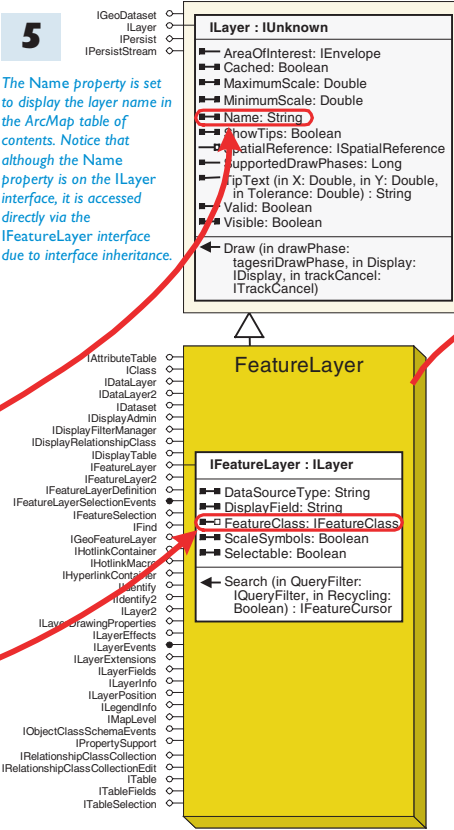
8

Finally, the newly added layer is drawn on the screen. Notice the use of the PartialRefresh method instead of the Refresh method; this ensures optimal drawing of all the map layers.



4

In order to add data to the map, create a FeatureLayer and associate the FeatureClass with it.



```

Add this code to the Click event of a UIButtonControl in ArcMap.

1 Dim pWorkspaceFactory As IWorkspaceFactory
  Set pWorkspaceFactory = New ShapefileWorkspaceFactory

2 Dim pWorkspace As IFeatureWorkspace
  Set pWorkspace = pWorkspaceFactory.OpenFromFile("C:\Source\", 0)

3 Dim pClass As IFeatureClass
  Set pClass = pWorkspace.OpenFeatureClass("USStates")

4 Dim pLayer As IFeatureLayer
  Set pLayer = New FeatureLayer
  Set pLayer.FeatureClass = pClass

5 pLayer.Name = pClass.AliasName

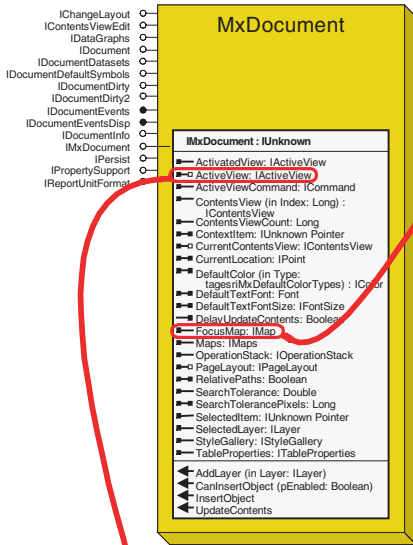
6 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

7 pMxDoc.AddLayer pLayer
8 pMxDoc.ActiveView.PartialRefresh esriViewGeography, pLayer, _
  Nothing
  
```

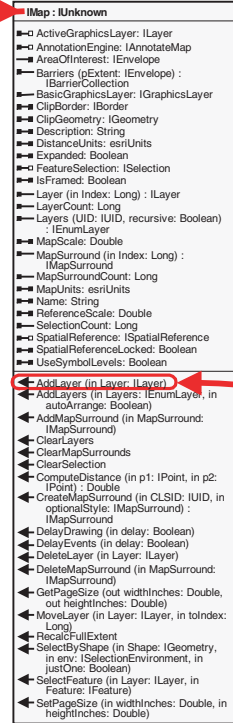
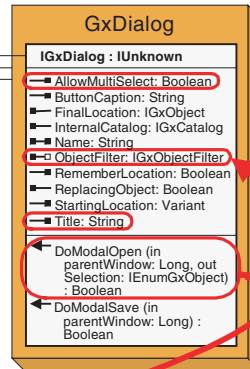
This example allows the user to select a feature dataset or feature class to be added to ArcMap using the *GxDialog*.

I To obtain the active map, use the *FocusMap* property of the *IMxDocument* interface.

ArcMapUI

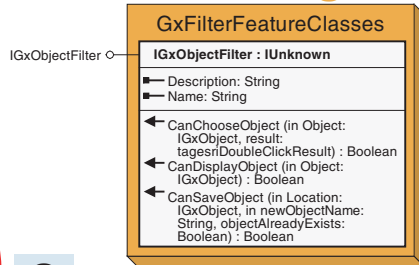


2 The *GxDialog* class provides the user interface used by all ArcGIS applications when selecting data sources.

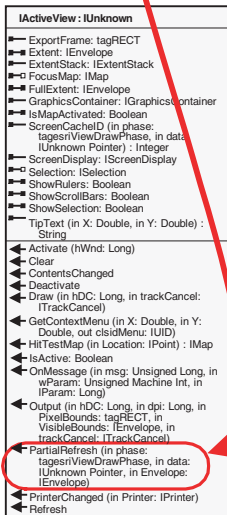


3 To limit the data sources available for selection within the dialog box, a *GxObjectFilter* is used. For this example, the filter only allows feature classes to be selected. Using filters simplifies the code after the selection is made.

CatalogUI

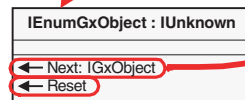


4 The *DoModalOpen* method on the *IGxDialog* interface is called to display the *GxDialog*. Once the user has finished, the selected feature classes can be accessed via the *GxObject* enumerator that is passed out of the method call.



8 Finally, the newly added layer is drawn on the screen. Notice the use of the *PartialRefresh* method instead of the *Refresh* method; this ensures optimal drawing of all the map layers.

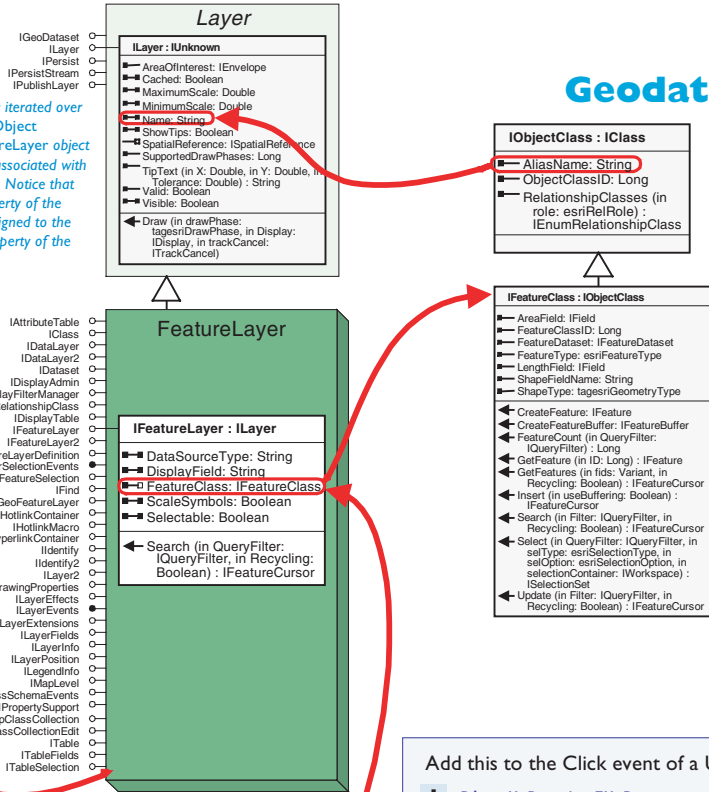
If the enumerator is nothing, no selections were made and the sub is exited. Otherwise, the enumerator is reset in preparation for its iteration.



7

The enumerator is iterated over and for each GxObject accessed, a FeatureLayer object is created that is associated with the FeatureClass. Notice that the Dataset property of the GxDataset is assigned to the FeatureClass property of the Layer.

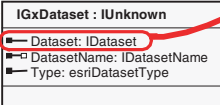
Geodatabase



Carto

6

Since an appropriate GxObjectFilter object was used, the GxObjects returned from the enumerator will support the IGxDataset interface.



Add this to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim pGxDialoG As IGxDialoG
  Set pGxDialoG = New GxDialoG
  pGxDialoG.AllowMultiSelect = True
  pGxDialoG.Title = "Select Feature Classes to Add to Map"

3 Dim pGxFilter As IGxObjectFilter
  Set pGxFilter = New GxFilterFeatureClasses
  Set pGxDialoG.ObjectFilter = pGxFilter

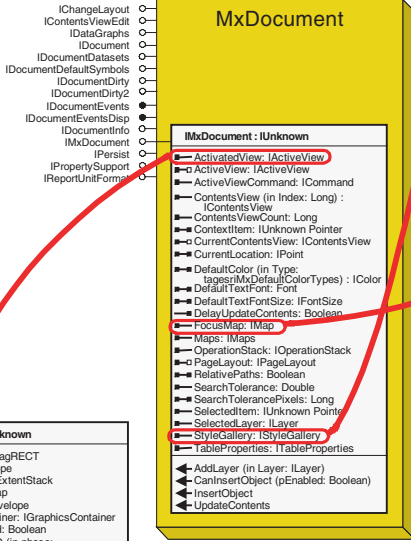
4 Dim pGxObjects As IEnumGxObject
  pGxDialoG.DoModalOpen ThisDocument.Parent.hWnd, pGxObjects

5 If (pGxObjects Is Nothing) Then Exit Sub
  pGxObjects.Reset

  Dim pLayer As IFeatureLayer
  Dim pGxDataset As IGxDataset
6 Set pGxDataset = pGxObjects.Next
  Do Until (pGxDataset Is Nothing)
7   Set pLayer = New FeatureLayer
   Set pLayer.FeatureClass = pGxDataset.Dataset
   pLayer.Name = pLayer.FeatureClass.AliasName
   pMxDoc.FocusMap.AddLayer pLayer
   Set pGxDataset = pGxObjects.Next
  Loop
8 pMxDoc.ActiveView.PartialRefresh esriViewGeography, _
  Nothing, Nothing
  
```

This sample goes through all polygon layers in the map and attempts to match the symbology from the standard style set to the layer name. ArcMap does this by default. Therefore, to see a real difference before testing the tool, layer names should be changed to reflect suitable styles. For example, try changing a layer name to "Glacier" and executing this command.

1 To begin, you must gain access to the current document.



IActiveView: IUnknown

- ExportFrame: tagRECT
- Extent: IEnvelope
- ExtentStack: IExtentStack
- FocusMap: IMap
- FullExtent: IEnvelope
- GraphicsContainer: IGraphicsContainer
- IsMapActivated: Boolean
- ScreenCacheID (in phase: tagesiViewDrawPhase, in data: IUnknown Pointer) : Integer
- ScreenDisplay: IScreenDisplay
- Selection: ISelection
- ShowRulers: Boolean
- ShowScrollBars: Boolean
- ShowSelection: Boolean
- TipText (in X: Double, in Y: Double) : String
- Activate (in Wnd: Long)
- Clear
- ContentsChanged
- Deactivate
- Draw (in HDC: Long, in trackCancel: ITrackCancel)
- GetContextMenu (in X: Double, in Y: Double, out clsidMenu: IUID)
- HitTestMap (in Location: IPoint) : IMap
- IsActive: Boolean
- OnMessage (in msg: Unsigned Long, in wParam: Unsigned Machine Int, in lParam: Long)
- Output (in HDC: Long, in dpi: Long, in PixelBounds: tagRECT, in VisibleBounds: IEnvelope, in trackCancel: ITrackCancel)
- PartialRefresh (in phase: tagesiViewDrawPhase, in data: IUnknown Pointer, in Envelope: IEnvelope)**
- PrinterChanged (in Printer: IPrinter)
- Refresh

2 An enumerator is obtained from the style gallery for the style gallery's FillSymbol entries that, when accessed, will loop over all the FillSymbols.

IStyleGallery: IUnknown

- Categories (in ClassName: String) : IEnumBSTR
- Class (in Index: Long) : IStyleGalleryClass
- ClassCount: Long
- Items (in ClassName: String, in styleSet: String, in Category: String) : IEnumStyleGalleryItem**
- AddItem (in Item: IStyleGalleryItem)
- Clear
- ImportStyle (in FileName: String)
- LoadStyle (in FileName: String, in ClassName: String)
- RemoveItem (in Item: IStyleGalleryItem)
- SaveStyle (in FileName: String, in styleSet: String, in ClassName: String)
- UpdateItem (in Item: IStyleGalleryItem)

ArcMapUI

3 Using the IMap layer properties, loop over all the layers in the map.

IMap: IUnknown

- ActiveGraphicsLayer: ILayer
- AnnotationEngine: IAnnotateMap
- AreaOfInterest: IEnvelope
- Barriers (pExtent: IEnvelope) : IBarriersCollection
- BasicGraphicsLayer: IGraphicsLayer
- ClipBorder: IBorder
- CityGeometry: IGeometry
- Description: String
- DistanceUnits: esriUnits
- Expanded: Boolean
- FeatureSelection: ISelection
- IsFramed: Boolean
- Layer (in Index: Long) : ILayer**
- LayerCount: Long**
- Layers (UID: IUID, recursive: Boolean) : IEnumLayer**
- MapScale: Double
- MapSurround (in Index: Long) : IMapSurround
- MapSurroundCount: Long
- MapUnits: esriUnits
- Name: String
- ReferenceScale: Double
- SelectionCount: Long
- SpatialReference: ISpatialReference
- SpatialReferenceLocked: Boolean
- UseSymbolLevels: Boolean
- AddLayer (in Layer: ILayer)
- AddLayers (in Layers: IEnumLayer, in autoArrange: Boolean)
- AddMapSurround (in MapSurround: IMapSurround)
- ClearLayers
- ClearMapSurrounds
- ClearSelection
- ComputeDistance (in p1: IPoint, in p2: IPoint) : Double
- CreateMapSurround (in CLSID: IUID, in optionalStyle: IMapSurround) : IMapSurround
- DelayDrawing (in delay: Boolean)
- DelayEvents (in delay: Boolean)
- DeleteLayer (in Layer: ILayer)
- DeleteMapSurround (in MapSurround: IMapSurround)
- GetPageSize (out widthInches: Double, out heightInches: Double)
- MoveLayer (in Layer: ILayer, in toIndex: Long)
- RecalcFullExtent
- SelectByShape (in Shape: IGeometry, in env: ISelectionEnvironment, in justOne: Boolean)
- SelectFeature (in Layer: ILayer, in Feature: IFeature)
- SetPageSize (in widthInches: Double, in heightInches: Double)

8 Finally, all the geographic layers are refreshed to update the map display.

Display

IStyleGalleryItem : IUnknown	
Category: String	
ID: Long	
Item: IUnknown Pointer	
Name: String	

Geodatabase

IFeatureClass : IObjectClass	
AreaField: IField	
FeatureClassID: Long	
FeatureDataset: IFeatureDataset	
FeatureType: esriFeatureType	
LengthField: IField	
ShapeFieldName: String	
ShapeType: IFeatureGeometryType	
CreateFeature: IFeature	
CreateFeatureBuffer: IFeatureBuffer	
FeatureCount (in QueryFilter: IQueryFilter): Long	
GetFeature (in ID: Long): IFeature	
GetFeatures (in fids: Variant, in Recycling: Boolean): IFeatureCursor	
Insert (in UseBuffering: Boolean): IFeatureCursor	
Search (in Filter: IQueryFilter, in Recycling: Boolean): IFeatureCursor	
Select (in QueryFilter: IQueryFilter, in selType: esriSelectionType, in selOption: esriSelectionOption, in selectionContainer: IWorkspace): ISelectionSet	
Update (in Filter: IQueryFilter, in Recycling: Boolean): IFeatureCursor	

IEnumStyleGalleryItem: IUnknown	
Next: IStyleGalleryItem	
Reset	

6

The style gallery enumerator is reset and then iterated over to look for a match between the style item name and the layer name.

Carto

ILayer : IUnknown	
AreaOfInterest: IEnvelope	
Cached: Boolean	
MaximumScale: Double	
MinimumScale: Double	
Name: String	
ShowTips: Boolean	
SpatialReference: ISpatialReference	
SupportedDrawPhases: Long	
TipText (in X: Double, in Y: Double, in Tolerance: Double): String	
Valid: Boolean	
Visible: Boolean	
Draw (in drawPhase: IFeatureDrawPhase, in Display: IDisplay, in trackCancel: ITrackCancel)	

IFeatureLayer : ILayer	
DataSourceType: String	
DisplayField: String	
FeatureClass: IFeatureClass	
ScaleSymbols: Boolean	
Selectable: Boolean	
Search (in QueryFilter: IQueryFilter, in Recycling: Boolean): IFeatureCursor	

IGeoFeatureLayer : IFeatureLayer	
AnnotationProperties: IAnnotateLayerPropertiesCollection	
AnnotationPropertiesID: IUID	
CurrentMapLevel: Long	
DisplayAnnotation: Boolean	
DisplayFeatureClass: IFeatureClass	
ExclusionSet: IFeatureIDSet	
Renderer: IFeatureRenderer	
RendererPropertyPageClassID: IUID	
SearchDisplayFeatures (in QueryFilter: IQueryFilter, in Recycling: Boolean): IFeatureCursor	

4

If the type of layer is not an IGeoFeatureLayer, continue to the next layer.

5

Using the FeatureClass property, check the shape type of the layer. If it is not Polygon, skip to the next layer.

Carto

ISimpleRenderer: IUnknown	
Description: String	
Label: String	
Symbol: ISymbol	

7

If a match in name is found, the symbol obtained from the style gallery is set into the renderer.

Add this to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim pStyleItems As IEnumStyleGalleryItem
  Set pStyleItems = pMxDoc.StyleGallery.Items("Fill Symbols", _
  "ESRI.style", "Default")
  Dim pGalleryItem As IStyleGalleryItem

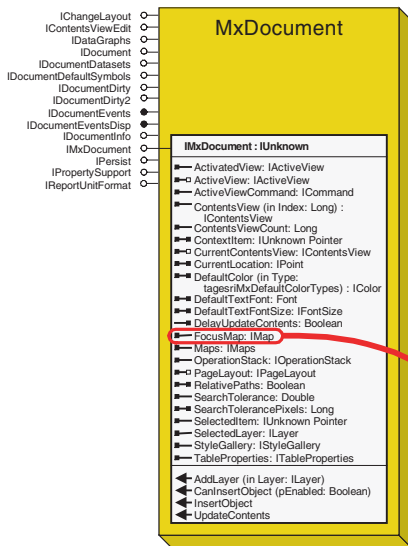
  Dim pRenderer As ISimpleRenderer
  Dim pGeoFeatureLayer As IGeoFeatureLayer
  Dim i As Long

3 For i = 0 To pMxDoc.FocusMap.LayerCount - 1
4   If (TypeOf pMxDoc.FocusMap.Layer(i) Is IGeoFeatureLayer) Then
5     pGeoFeatureLayer = pMxDoc.FocusMap.Layer(i)
6     If (pGeoFeatureLayer.FeatureClass.ShapeType = _
     esriGeometryPolygon) Then
7       pStyleItems.Reset
8       Set pGalleryItem = pStyleItems.Next
9       Do While (Not pGalleryItem Is Nothing)
10        If (pGeoFeatureLayer.Name = pGalleryItem.Name) Then
11          Set pRenderer = pGeoFeatureLayer.Renderer
12          Set pRenderer.Symbol = pGalleryItem.Item
13          Exit Do
14        End If
15      Loop
16    End If
17  End If
18  pMxDoc.ActivatedView.PartialRefresh esriViewGeography, _
  Nothing, Nothing
  
```

This sample loops through the selected features of the focus map. It loops using the *IEnumFeature* interface, which is reached through a *QueryInterface* from the *FeatureSelection* property of the map. For each feature it checks the geometry type and if *Polygon*, it performs a *QueryInterface* for the *IArea* interface. Using the *Area* property of the interface, it adds the area to a running total. At the end, it reports the total area via a message box.

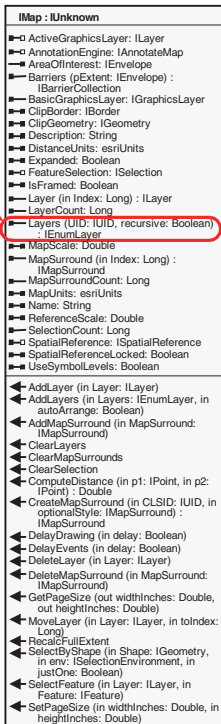
1 To obtain the layers of the map, you must first get access to the currently active map. Do this through the *FocusMap* property of the *IMxDocument* interface.

ArcMapUI



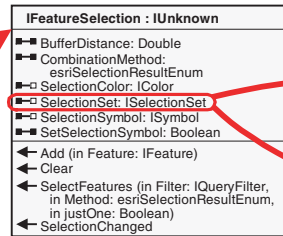
2 The UID helper object is used to represent the GUID for the *IGeoFeatureLayer* interface.

System



3 The UID object created previously is used to obtain an enumerator for all layers that support the *IGeoFeatureLayer* interface. Notice the resetting of the enumerator before its use.

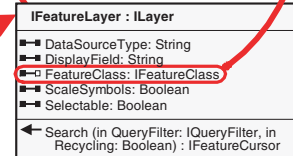
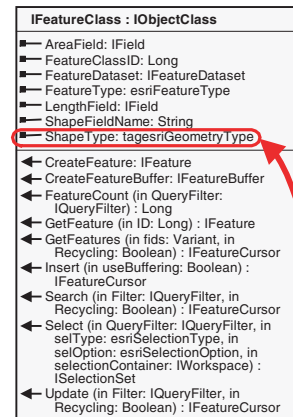
Carto



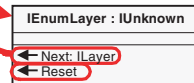
6 Obtain the *IFeatureSelection* interface by performing a *QueryInterface* to the *IFeatureLayer* interface.

5

If the shape type of the feature class is not a *Polygon*, the layer is skipped.



4 The layers enumerator is iterated over using the standard enumerator method, *Next*.



7

Before attempting to loop through selected features, a check is performed to ensure that there are selected features for the current layer. If there are no selected features, the layer is skipped.

ISelectionSet : IUnknown	
←	Count: Long
←	FullName: IName
←	IDs: IEnumIDs
←	Target: ITable
←	Add (in OID: Long)
←	AddList (in Count: Long, in OIDList: Long)
←	Combine (in otherSet: ISelectionSet, in selOp: esriSetOperation, out resultSet: ISelectionSet)
←	MakePermanent
←	Refresh
←	RemoveList (in Count: Long, in OIDList: Long)
←	Search (in pQueryFilter: IQueryFilter, in Recycling: Boolean, out ppCursor: ICursor)
←	Select (in QueryFilter: IQueryFilter, in selType: esriSelectionType, in selOption: esriSelectionOption, in selectionContainer: IWorkspace) : ISelectionSet

8

If there are selected features, a cursor onto these features is obtained from the layers selection set.

IFeatureCursor : IUnknown	
←	Fields: IFields
←	DeleteFeature
←	FindField (in Name: String) : Long
←	Flush
←	InsertFeature (in Buffer: IFeatureBuffer) : Variant
←	NextFeature: IFeature
←	UpdateFeature (in Object: IFeature)

9

For each feature returned by the cursor, the Area of the feature's shape is obtained and totalled. The area is obtained by performing a QueryInterface on the feature's shape for the IArea interface and getting the Area property from it.

10

Finally, the totalled area is displayed to the user in a standard Visual Basic Message Box.

Geodatabase

Geometry

IArea : IUnknown	
←	Area: Double
←	Centroid: IPoint
←	LabelPoint: IPoint
←	QueryCentroid (Center: IPoint)
←	QueryLabelPoint (LabelPoint: IPoint)

Add this code to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
    Set pMxDoc = ThisDocument

2 Dim pUID As New UID
    pUID = "{E156D7E5-22AF-11D3-9F99-00C04F6BC78E}" 'IGeoFeatureLayer IID

3 Dim pEnumLayer As IEnumLayer
    Set pEnumLayer = pMxDoc.FocusMap.Layers(pUID, True)
    pEnumLayer.Reset

    Dim pFeatureLayer As IFeatureLayer
    Dim pFeatureSelection As IFeatureSelection
    Dim pFeatureCursor As IFeatureCursor
    Dim pFeature As IFeature
    Dim pArea As IArea
    Dim dTotalArea As Double

4 Set pFeatureLayer = pEnumLayer.Next
    Do Until (pFeatureLayer Is Nothing)
5     If (pFeatureLayer.FeatureClass.ShapeType = esriGeometryPolygon) Then
6         Set pFeatureSelection = pFeatureLayer

7         If (pFeatureSelection.SelectionSet.Count > 0) Then
8             pFeatureSelection.SelectionSet.Search Nothing, True, pFeatureCursor
             Set pFeature = pFeatureCursor.NextFeature

             Do Until (pFeature Is Nothing)
9                 Set pArea = pFeature.Shape
                 dTotalArea = dTotalArea + pArea.Area
                 Set pFeature = pFeatureCursor.NextFeature
             Loop
         End If
     End If
     Set pFeatureLayer = pEnumLayer.Next
 Loop

10 MsgBox "Total Area for selected polygon features = " & CStr(dTotalArea)
  
```


This sample builds a spatial query filter, gets a feature cursor based on the filter and then loops over all the features, totalling the number of points, lines, and areas, and reports these to the user.

6 Each layer in the map is looped over; if the layer is not of type IGeoFeatureLayer, the layer is skipped.

Carto

1 The IMxDocument interface is obtained from the ThisDocument global variable.

```

IChangeLayout
IContentsViewEdit
IDataGraphs
IDocument
IDocumentDatasets
IDocumentDefaultSymbols
IDocumentDirty
IDocumentDirty2
IDocumentEvents
IDocumentEventsDisp
IDocumentInfo
IPropertySupport
IReportUnitFormat

MxDocument
IMxDocument : IUnknown
  - ActivatedView: IActiveView
  - ActiveView: IActiveView
  - ActiveViewCommand: ICommand
  - ContentsView (in Index: Long) : IContentsView
  - ContentsViewCount: Long
  - ContextItem: IUnknown Pointer
  - CurrentContentsView: IContentsView
  - DefaultColor (in Type: tagsesMxDefaultColorTypes) : IColor
  - DefaultTextFont: Font
  - DefaultTextFontSize: IFontSize
  - DelayUpdateContents: Boolean
  - FocusMap: IMap
  - Maps: IMaps
  - OperationStack: IOperationStack
  - PageLayout: IPageLayout
  - RelativePaths: Boolean
  - SearchTolerance: Double
  - SearchTolerancePixels: Long
  - SelectedItem: IUnknown Pointer
  - SelectedLayer: ILayer
  - StyleGallery: IStyleGallery
  - TableProperties: ITableProperties
  - AddLayer (in Layer: ILayer)
  - CanInsertObject (pEnabled: Boolean)
  - InsertObject
  - UpdateContents
    
```

3 The active view associated with the focus map is acquired in order for the rubber banding geometry to have the correct spatial reference.

```

IActiveView : IUnknown
  - ExportFrame: tagRECT
  - Extent: IEnvelope
  - ExtentStack: IExtentStack
  - FocusMap: IMap
  - FullExtent: IEnvelope
  - GraphicsContainer: IGraphicsContainer
  - IsMapActivated: Boolean
  - ScreenCacheID (in phase: tagsesrViewDrawPhase, in data: IUnknown Pointer) : Integer
  - ScreenDisplay: IScreenDisplay
  - Selection: ISelection
  - ShowRulers: Boolean
  - ShowScrollBars: Boolean
  - ShowSelection: Boolean
  - TagText (in X: Double, in Y: Double) : String
  - Activate (inWnd: Long)
  - Clear
  - ContentsChanged
  - Deactivate
  - Draw (in HDC: Long, in trackCancel: ITrackCancel)
  - GetContextMenu (in X: Double, in Y: Double, out clsidMenu: IUIID)
  - HitTestMap (in Location: IPoint) : IMap
  - IsActive: Boolean
  - OnMessage (in msg: Unsigned Long, in wParam: Unsigned Machine Int, in lParam: Long)
  - Output (in HDC: Long, in dpi: Long, in PixelBounds: tagRECT, in VisibleBounds: IEnvelope, in trackCancel: ITrackCancel)
  - PartialRefresh (in phase: tagsesrViewDrawPhase, in data: IUnknown Pointer, in Envelope: IEnvelope)
  - PrinterChanged (in Printer: IPrinter)
  - Refresh
    
```

2 A user-defined envelope defining the extent of the spatial query is required. The rubber envelope object is used.

```

IRubberBand
IRubberBand : IUnknown
  - TrackExisting (in ScreenDisplay: IScreenDisplay, in Symbol: ISymbol, in Geometry: IGeometry) : Boolean
  - TrackNew (in ScreenDisplay: IScreenDisplay, in Symbol: ISymbol) : IGeometry

RubberBand
RubberEnvelope
    
```

Display

4 The TrackNew method is called. This allows the user to drag the mouse to define the envelope.

```

IMap : IUnknown
  - ActiveGraphicsLayer: ILayer
  - AnnotationEngine: IAnnotateMap
  - AreaOfInterest: IEnvelope
  - Barriers (pExtent: IEnvelope) : IBarrierCollection
  - BasicGraphicsLayer: IGraphicsLayer
  - ClipBorder: IBorder
  - ClipGeometry: IGeometry
  - Description: String
  - DistanceUnits: esriUnits
  - Expanded: Boolean
  - FeatureSelection: ISelection
  - IsFramed: Boolean
  - Layer (in Index: Long) : ILayer
  - LayerCount: Long
  - Layers (UID: IUID, recursive: Boolean) : IEnumLayer
  - MapScale: Double
  - MapSurround (in Index: Long) : IMapSurround
  - MapSurroundCount: Long
  - MapUnits: esriUnits
  - Name: String
  - ReferenceScale: Double
  - SelectionCount: Long
  - SpatialReference: ISpatialReference
  - SpatialReferenceLocked: Boolean
  - UseSymbolLevels: Boolean
  - AddLayer (in Layer: ILayer)
  - AddLayers (in Layers: IEnumLayer, in autoArrange: Boolean)
  - AddMapSurround (in MapSurround: IMapSurround)
  - ClearLayers
  - ClearMapSurrounds
  - ClearSelection
  - ComputeDistance (in p1: IPoint, in p2: IPoint) : Double
  - CreateMapSurround (in CLSID: IUID, in optionalStyles: IMapSurround) : IMapSurround
  - DelayDrawing (in delay: Boolean)
  - DelayEvents (in delay: Boolean)
  - DeleteLayer (in Layer: ILayer)
  - DeleteMapSurround (in MapSurround: IMapSurround)
  - GetPageSize (out widthInches: Double, out heightInches: Double)
  - MoveLayer (in Layer: ILayer, in toIndex: Long)
  - RecalculateExtent
  - SelectByShape (in Shape: IGeometry, in env: ISelectionEnvironment, in justOne: Boolean)
  - SelectFeature (in Layer: ILayer, in Feature: IFeature)
  - SetPageSize (in widthInches: Double, in heightInches: Double)
    
```

```

IFeatureLayer : ILayer
  - DataSourceType: String
  - DisplayField: String
  - FeatureClass: IFeatureClass
  - ScaleSymbols: Boolean
  - Selectable: Boolean
  - Search (in QueryFilter: IQueryFilter, in Recycling: Boolean) : IFeatureCursor
    
```

```

IFeatureClass : IObjectClass
  - AreaField: IField
  - FeatureClassID: Long
  - FeatureDataset: IFeatureDataset
  - FeatureType: esriFeatureType
  - LengthFields: IField
  - ShapeFieldName: String
  - ShapeType: tagsesrGeometryType
  - CreateFeature: IFeature
  - CreateFeatureBuffer: IFeatureBuffer
  - FeatureCount (in QueryFilter: IQueryFilter) : Long
  - GetFeature (in ID: Long) : IFeature
  - GetFeatures (in fids: Variant, in Recycling: Boolean) : IFeatureCursor
  - Insert (in useBuffering: Boolean) : IFeatureCursor
  - Search (in Filter: IQueryFilter, in Recycling: Boolean) : IFeatureCursor
  - Select (in QueryFilter: IQueryFilter, in selType: esriSelectionType, in selOption: esriSelectionOption, in selectionContainer: IWorkspace) : ISelectionSet
  - Update (in Filter: IQueryFilter, in Recycling: Boolean) : IFeatureCursor
    
```

```

IEnvelope : IGeometry
  - Depth: Double
  - Height: Double
  - LowerLeft: IPoint
  - LowerRight: IPoint
  - MMax: Double
  - MMin: Double
  - UpperLeft: IPoint
  - UpperRight: IPoint
  - Width: Double
  - XMax: Double
  - XMin: Double
  - YMax: Double
  - YMin: Double
  - ZMax: Double
  - ZMin: Double
  - CenterAt (p: IPoint)
  - DefineFromPoints (Count: Long, in Points: IPoint)
  - Expand (dx: Double, dy: Double, asRatio: Boolean)
  - ExpandM (dm: Double, asRatio: Boolean)
  - ExpandZ (dz: Double, asRatio: Boolean)
  - Intersect (inEnvelope: IEnvelope)
  - Offset (X: Double, Y: Double)
  - OffsetM (M: Double)
  - OffsetZ (Z: Double)
  - PutCoords (XMin: Double, YMin: Double, XMax: Double, YMax: Double)
  - QueryCoords (out XMin: Double, out YMin: Double, out XMax: Double, out YMax: Double)
  - Union (inEnvelope: IEnvelope)
    
```

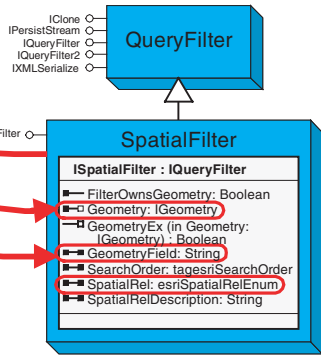

8

A feature cursor is obtained from the layer by calling the Search method passing in the SpatialFilter.

Geodatabase

5

A new spatial filter object is created. The shape and spatial reference is set.



7

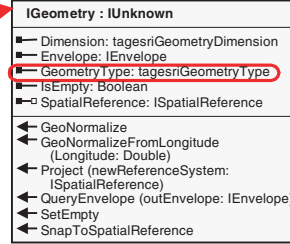
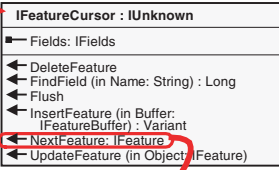
The spatial filter must be told what column in the database table holds the feature shape. This information is retrieved from the feature class.

Enumeration esriSpatialRelEnum

- 0 - esriSpatialRelUndefined
- 1 - esriSpatialRelIntersects
- 2 - esriSpatialRelEnvelopeIntersects
- 3 - esriSpatialRelIndexIntersects
- 4 - esriSpatialRelTouches
- 5 - esriSpatialRelOverlaps
- 6 - esriSpatialRelCrosses
- 7 - esriSpatialRelWithin
- 8 - esriSpatialRelContains
- 9 - esriSpatialRelRelation

10

Finally, the results of the selection are displayed in a Visual Basic message box.



Enumeration esriGeometryTypeConstants

- 0 - esriGeometryNull
- 1 - esriGeometryPoint
- 2 - esriGeometryMultipoint
- 3 - esriGeometryPolyline
- 4 - esriGeometryPolygon
- 5 - esriGeometryEnvelope
- 6 - esriGeometryPath
- 7 - esriGeometryAny
- 9 - esriGeometryMultiPatch
- 11 - esriGeometryRing
- 13 - esriGeometryLine
- 14 - esriGeometryCircularArc
- 15 - esriGeometryBezier3Curve
- 16 - esriGeometryEllipticArc
- 17 - esriGeometryBag
- 18 - esriGeometryTriangleStrip
- 19 - esriGeometryTriangleFan
- 20 - esriGeometryRay
- 21 - esriGeometrySphere

Geometry

9

This cursor is looped over and the features returned by the cursor are inspected. Based on their geometry type, the totals are updated accordingly.

Add this to theMouseDown event of aUIToolControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

  Dim pEnv As IEnvelope
  Dim pRubber As IRubberBand
2 Set pRubber = New RubberEnvelope

3 Dim pActiveView As IActiveView
  Set pActiveView = pMxDoc.FocusMap
4 Set pEnv = pRubber.TrackNew(pActiveView.ScreenDisplay, Nothing)

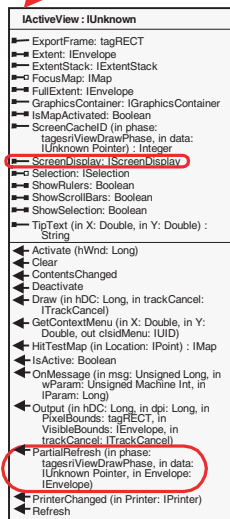
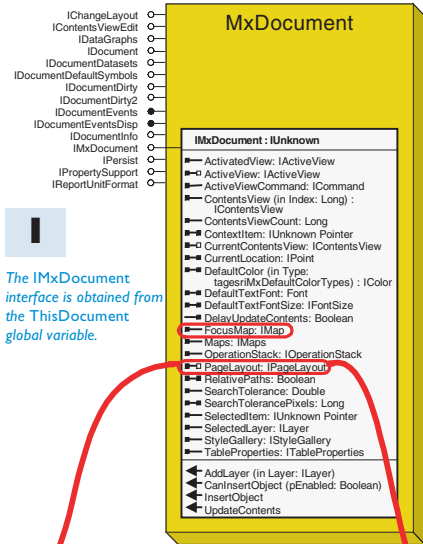
5 Dim pSpatialFilter As ISpatialFilter
  Set pSpatialFilter = New SpatialFilter
  Set pSpatialFilter.Geometry = pEnv
  pSpatialFilter.SpatialRel = esriSpatialRelIntersects

  Dim lPoints As Long, lPolygons As Long, lPolylines As Long
  Dim pLayer As IFeatureLayer
  Dim pFeatureCursor As IFeatureCursor
  Dim pFeature As IFeature
  Dim i As Long
  For i = 0 To pMxDoc.FocusMap.LayerCount - 1
6   If (TypeOf pMxDoc.FocusMap.Layer(i) Is IGeoFeatureLayer) Then
7     pSpatialFilter.GeometryField = pLayer.FeatureClass.ShapeFieldName
8
9     Set pFeatureCursor = pLayer.Search(pSpatialFilter, True)
     Set pFeature = pFeatureCursor.NextFeature
     Do Until (pFeature Is Nothing)
       Select Case pFeature.Shape.GeometryType
         Case esriGeometryPoint
           lPoints = lPoints + 1
         Case esriGeometryPolyline
           lPolylines = lPolylines + 1
         Case esriGeometryPolygon
           lPolygons = lPolygons + 1
       End Select
       Set pFeature = pFeatureCursor.NextFeature
     Loop
   End If
  Next i

10 MsgBox "Features Found:" & vbCrLf & lPoints & " Points " & vbCrLf & _
  lPolylines & " Polylines " & vbCrLf & lPolygons & " Polygons "
  
```

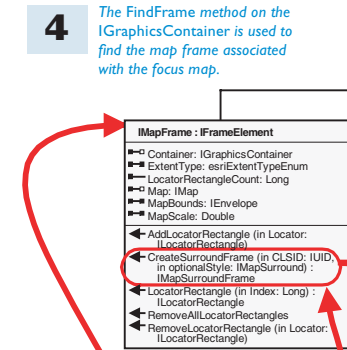
This example adds legend map surround to a page layout and fills the legend with the layers of the map. Map surrounds are dynamically linked to their associated map; therefore, any changes to the map are reflected in the map surround.

Carto

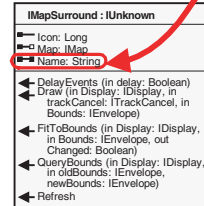
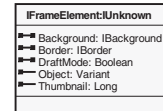
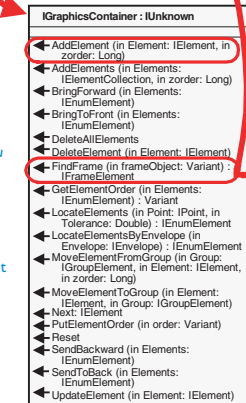


2 You must ensure that the active view associated with the `PageLayout` is used. Hence, you cannot use the `ActiveView` property of the `IMxDocument` interface since that may be associated with the `FocusMap`. You must perform a `QueryInterface` on `PageLayout` for its `IActiveView` interface.

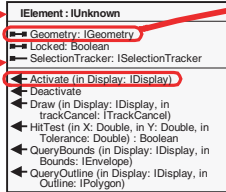
III Finally, the graphics layer of the screen is refreshed.



3 The graphic container associated with the `PageLayout` is obtained.



Carto

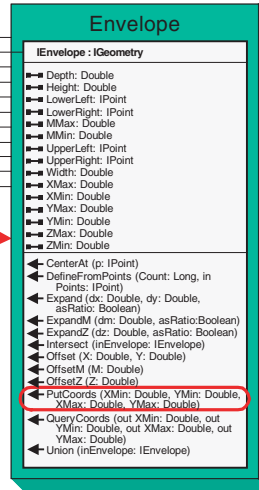


7

The IElement interface is accessed by a QueryInterface from IMapSurroundFrame. This interface is required to set the geometry of the frame. The geometry controls the location of the legend on the paper.

8

The geometry associated with the focus map's MapFrame is obtained.

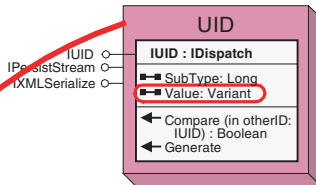


9

A new envelope geometry for the legend is created and positioned relative to the focus map's map frame.

Geometry

System



5

The CreateSurroundFrame method requires the GUID of the surround element type. A UID object is created and its value is set to the ID of the legend class.

Add this to the Click event of a UIButtonControl in ArcMap, and execute the command when in Page View.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim pActiveView As IActiveView
  Set pActiveView = pMxDoc.PageLayout

  Dim pGraphicsContainer As IGraphicsContainer
  Dim pMapFrame As IMapFrame
3 Set pGraphicsContainer = pMxDoc.PageLayout
4 Set pMapFrame = pGraphicsContainer.FindFrame(pMxDoc.FocusMap)

  Dim pMapSurroundFrame As IMapSurroundFrame
  Dim pUID As New UID
  Dim pElement As IElement
5 pUID.Value = "esriCarto.Legend"
6 Set pMapSurroundFrame = pMapFrame.CreateSurroundFrame(pUID, Nothing)
  pMapSurroundFrame.MapSurround.Name = "Legend"

7 Set pElement = pMapSurroundFrame

  Dim pMainMapElement As IElement
  Dim pMainEnv As IEnvelope
8 Set pMainMapElement = pMapFrame
  Set pMainEnv = pMainMapElement.Geometry.Envelope

  Dim pEnv As IEnvelope
  Set pEnv = New Envelope
  pEnv.PutCoords pMainEnv.XMax + 1.5, pMainEnv.YMin + 1.5, _
    pMainEnv.XMax - 1.5, pMainEnv.YMax - 1.5
  pElement.Geometry = pEnv
9 pElement.Activate pActiveView.ScreenDisplay
10 pGraphicsContainer.AddElement pElement, 0
11 pActiveView.PartialRefresh esriViewGraphics, Nothing, Nothing
  
```

This sample adds one of the more complicated types of graphic elements to a map or page layout, depending on the current view. The callout is added to the center of the view.

3 The `IElement` interface is used to set the geometry of the element. The `IElement` interface is obtained by performing a `QueryInterface` on the `ITextElement` interface.

2 A `TextElement` object is created and its `Text` property is set. This is the object that will be added to the graphics container.

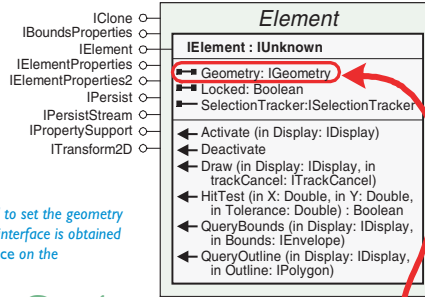
1 The `IMxDocument` interface is obtained from the `ThisDocument` global variable.

ArcMapUI

5 The geometry of the text element is a point. A new `Point` object is created and the coordinates are set, then the `Geometry` property of the `TextElement` is assigned this newly created point.

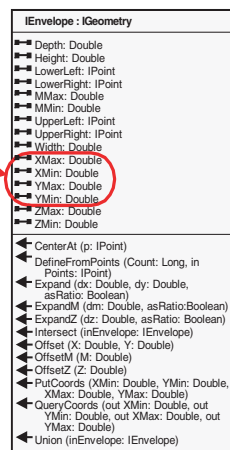
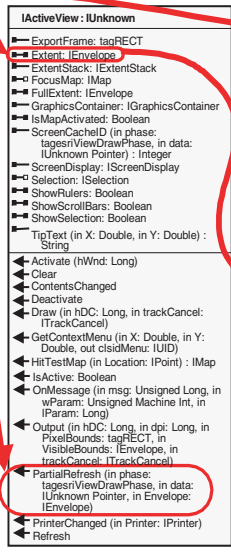
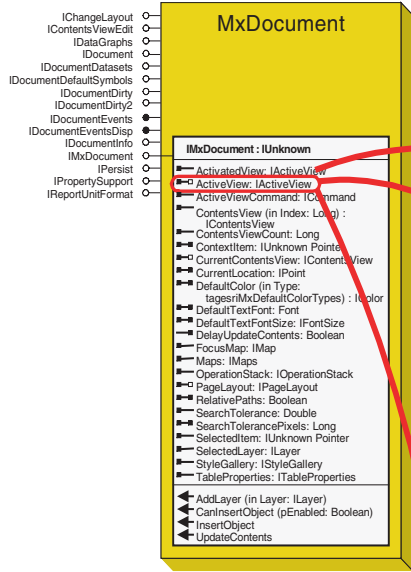
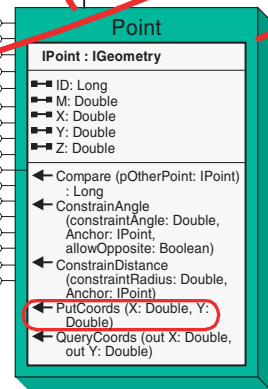
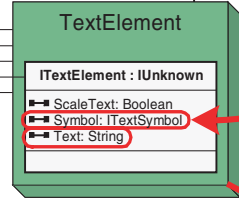
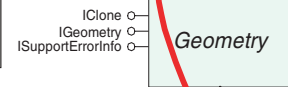
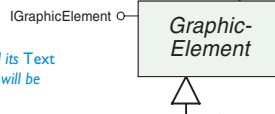
4 The center of the active view is calculated. This will be used to place the text element.

10 The graphics layer is redrawn to display the newly added text element. Once again, notice the use of the `PartialRefresh` method.



Carto

Geometry



8

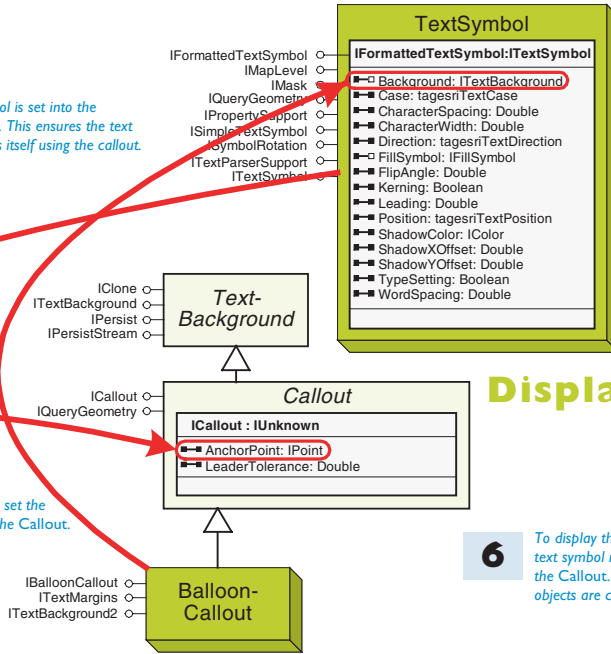
The text symbol is set into the TextElement. This ensures the text element draws itself using the callout.

7

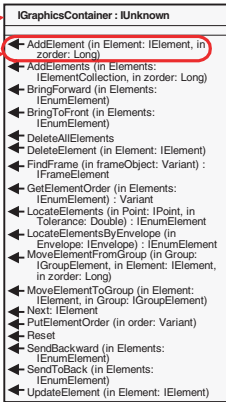
A Point is used to set the AnchorPoint of the Callout.

6

To display the text element as a callout, an appropriate text symbol must be used with the background set to be the Callout. The TextSymbol and BalloonCallout objects are created and associated with each other.



Display



9

The graphics container associated with the active view of the document is obtained by performing a QueryInterface on the IActiveView interface. The TextElement is then added to the container. This ensures that the element is saved within the map document.

Add this code to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim pTextElement As ITextElement
  Set pTextElement = New TextElement

3 Dim pElement As IElement
  Set pElement = pTextElement
  pTextElement.Text = "Text in a callout" & vbCrLf & "In middle of screen"

4 Dim dMidX As Double, dMidY As Double, pPoint As IPoint
  dMidX = (pMxDoc.ActiveView.Extent.XMax + pMxDoc.ActiveView.Extent.XMin) / 2
  dMidY = (pMxDoc.ActiveView.Extent.YMax + pMxDoc.ActiveView.Extent.YMin) / 2

5 Set pPoint = New Point
  pPoint.PutCoords dMidX, dMidY
  pElement.Geometry = pPoint

  Dim pTextSymbol As IFormattedTextSymbol
  Set pTextSymbol = New TextSymbol

6 Dim pCallout As ICallout
  Set pCallout = New BalloonCallout
  Set pTextSymbol.Background = pCallout

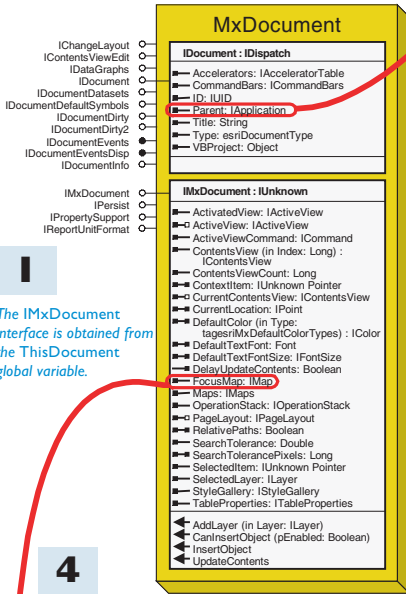
7 pPoint.PutCoords dMidX - pMxDoc.ActiveView.Extent.Width / 4, _
  dMidY + pMxDoc.ActiveView.Extent.Width / 20
  pCallout.AnchorPoint = pPoint

8 pTextElement.Symbol = pTextSymbol
9 Dim pGraphicsContainer As IGraphicsContainer
  Set pGraphicsContainer = pMxDoc.ActiveView
  pGraphicsContainer.AddElement pElement, 0
  pElement.Activate pMxDoc.ActiveView.ScreenDisplay
10 pMxDoc.ActiveView.PartialRefresh esriViewGraphics, Nothing, Nothing
  
```

This sample takes the current cursor coordinates and converts them from pixels to map units. It then projects these map coordinates to a projected and geographic spatial reference system, displaying the results in the Status Bar.

Framework

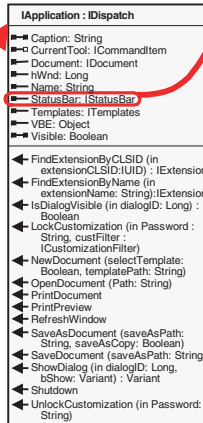
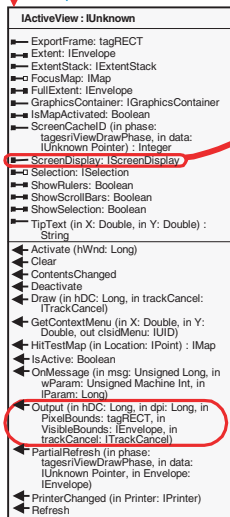
System



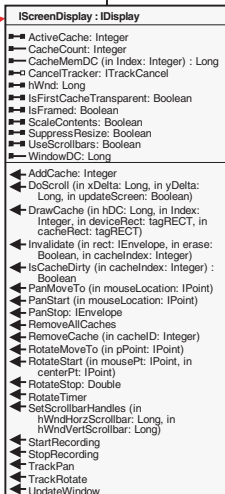
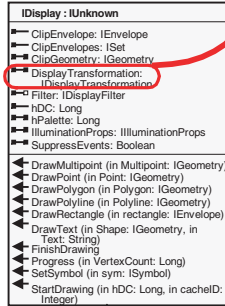
The **IMxDocument** interface is obtained from the **ThisDocument** global variable.

4

The active view of the focus map is obtained by performing a **QueryInterface** on the **FocusMap** property of the **IMxDocument** interface.

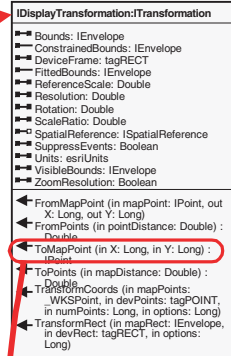
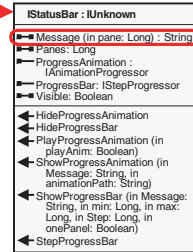


Display



9

The string is displayed in the status bar of the ArcMap application.

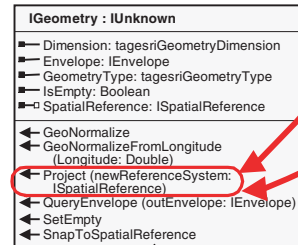


5

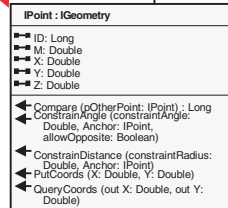
The cursor location in pixels (x,y) is converted to map units using a method on the **IDisplayTransformation** interface, then stored in a **Point** object. This point object will have the same spatial reference as the map.

8

The cursor point is projected from the Cassini coordinate system into the WGS 84 reference system and the coordinates are appended to a string.

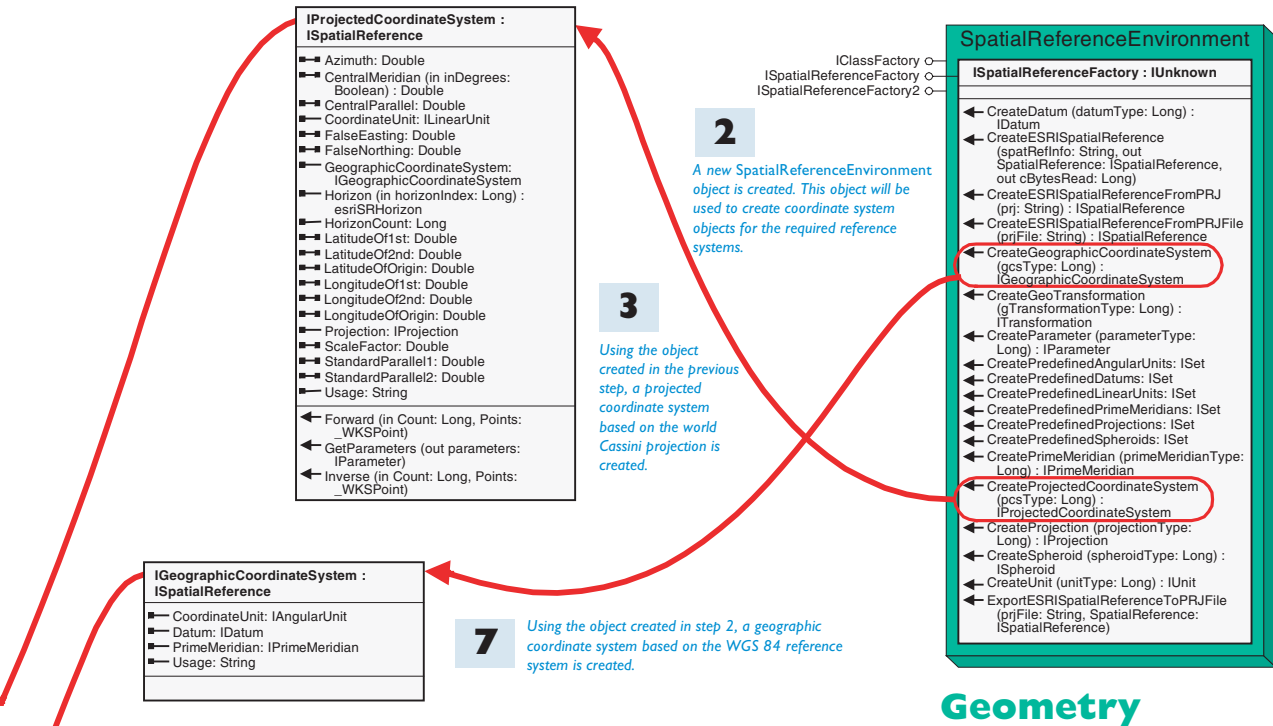


Geometry



6

The cursor point is projected from the map coordinates into the Cassini coordinate system and the projected coordinates are written to a string.



Geometry

Add this code to the MouseMove event of a UIToolControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
Set pMxDoc = ThisDocument

2 Dim pSpatialRefFactory As ISpatialReferenceFactory
Set pSpatialRefFactory = New SpatialReferenceEnvironment

Dim pProjectedCoordinateSystem As IProjectedCoordinateSystem
3 Set pProjectedCoordinateSystem =
pSpatialRefFactory.CreateProjectedCoordinateSystem(esriSRProjCS_World_Cassini)

4 Dim pActiveView As IActiveView
Set pActiveView = pMxDoc.FocusMap

5 Dim pPoint As IPoint
Set pPoint = pActiveView.ScreenDisplay.DisplayTransformation.ToMapPoint(x, y)

6 pPoint.Project pProjectedCoordinateSystem
Dim sMessage As String
sMessage = "Cassini : " & CStr(Round(pPoint.x, 2)) & ", " & _
CStr(Round(pPoint.y, 2))

7 Dim pGeographicCoordinateSystem As IGeographicCoordinateSystem
Set pGeographicCoordinateSystem = _
pSpatialRefFactory.CreateGeographicCoordinateSystem(esriSRGeoCS_WGS1984)

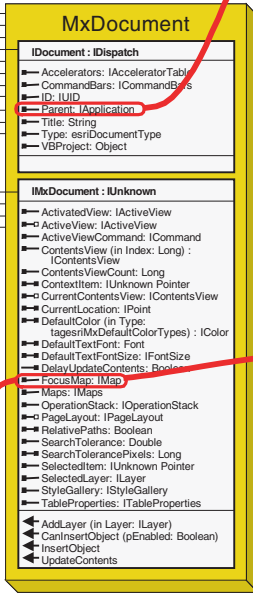
8 pPoint.Project pGeographicCoordinateSystem
sMessage = sMessage & " and WGS84 : " & CStr(Round(pPoint.x, 2)) & ", " & _
CStr(Round(pPoint.y, 2))

9 ThisDocument.Parent.StatusBar.Message(0) = sMessage
  
```


This sample displays the pixel value of the first raster layer in the map. This sample will display multiplane data in the form "(value 1, value 2, value 3)" for three planes.

1 The IIMxDocument interface is obtained from the ThisDocument global variable.

2 The active view for the focus map is obtained.



ArcMapUI

```

IActiveView : IUnknown
- ExportFrame : tagRECT
- Extent : IEnvelope
- ExtentStack : IExtentStack
- FocusMap : IMap
- FullExtent : IEnvelope
- GraphicsContainer : IGraphicsContainer
- IsMapActivated : Boolean
- ScreenCacheID (in phase: tagesiViewDrawPhase, in data: IUnknown Pointer) : Integer
- ScreenDisplay : IScreenDisplay
- Selection : ISelection
- ShowRulers : Boolean
- ShowScrollBars : Boolean
- ShowSelection : Boolean
- TipText (in X: Double, in Y: Double) : String
- Activate (hWnd: Long)
- Clear
- ContentsChanged
- Deactivate
- Draw (in hDC: Long, in trackCancel: ITrackCancel)
- GetContextMenu (in X: Double, in Y: Double, out clsidMenu: UIID)
- HitTestMap (in Location: IPoint) : IMap
- IsActive : Boolean
- OnMessage (in msg: Unsigned Long, in wParam: Unsigned Machine Int, in lParam: Long)
- Output (in hDC: Long, in dpi: Long, in PixelBounds : IEnvelope, in VisibleBounds : IEnvelope, in trackCancel: ITrackCancel)
- PartialRefresh (in phase: tagesiViewDrawPhase, in data: IUnknown Pointer, in Envelope: IEnvelope)
- PrinterChanged (in Printer: IPrinter)
- Refresh
    
```

```

IApplication : IDispatch
- Caption : String
- CurrentTool : ICommandItem
- Document : IDocument
- hWnd : Long
- Name : String
- Parent : IApplication
- Templates : ITemplates
- VBE : Object
- Visible : Boolean
- FindExtensionByCLSID (in extensionCLSID: UIID) : IExtension
- FindExtensionByName (in extensionName: String) : IExtension
- IsDialogVisible (in dialogID: Long) : Boolean
- LockCustomization (in Password : String, custFilter : ICustomizationFilter)
- NewDocument (selectTemplate: Boolean, templatePath: String)
- OpenDocument (Path: String)
- PrintDocument
- PrintPreview
- RefreshWindow
- SaveAsDocument (saveAsPath: String, saveAsCopy: Boolean)
- SaveDocument (saveAsPath: String)
- ShowDialog (in dialogID: Long, bShow_Variant) : Variant
- Shutdown
- UnlockCustomization (in Password: String)
    
```

System

```

IStatusBar : IUnknown
- Message (in pane: Long) : String
- Panes : Long
- ProgressAnimation : IAnimationProgressor
- ProgressBar : IStepProgressor
- Visible : Boolean
- HideProgressAnimation
- HideProgressBar
- PlayProgressAnimation (in playAnim: Boolean)
- ShowProgressAnimation (in Message: String, in animationPath: String)
- ShowProgressAnimation (in Message: String, in min: Long, in max: Long, in Step: Long, in onePanel: Boolean)
- StepProgressBar
    
```

```

IDisplayTransformation:Transformation
- Bounds : IEnvelope
- ConstrainedBounds : IEnvelope
- DeviceFrame : tagRECT
- FittedBounds : IEnvelope
- ReferenceScale : Double
- Resolution : Double
- Rotation : Double
- ScaleRatio : Double
- SpatialReference : ISpatialReference
- SuppressEvents : Boolean
- Units : esriUnits
- VisibleBounds : IEnvelope
- ZoomResolution : Boolean
- FromMapPoint (in mapPoint: IPoint, out X: Long, out Y: Long) : Double
- FromPoints (in pointDistance: Double) : Point
- ToMapPoint (in X: Long, in Y: Long) : Point
- ToPoints (in mapDistance: Double) : Double
- TransformCoords (in mapPoints: WKSPoint, in devPoints: tagPOINT, in numPoints: Long, in options: Long)
- TransformRect (in mapRect: IEnvelope, in devRect: tagRECT, in options: Long)
    
```

13 The raster values are displayed in the status bar.

Display

```

IDisplay : IUnknown
- ClipEnvelope : IEnvelope
- ClipEnvelopes : ISet
- ClipGeometry : IGeometry
- DisplayTransformation : IDisplayTransformation
- Filter : IDisplayFilter
- hDC : Long
- hPalette : Long
- IlluminationProps : IlluminationProps
- SuppressEvents : Boolean
- DrawMultipoint (in Multipoint: IGeometry)
- DrawPoint (in Point: IGeometry)
- DrawPolygon (in Polygon: IGeometry)
- DrawPolyline (in Polyline: IGeometry)
- DrawRectangle (in rectangle: IEnvelope)
- DrawText (in Shape: IGeometry, in Text: String)
- FinishDrawing
- Progress (in VertexCount: Long)
- SetSymbol (in sym: ISymbol)
- StartDrawing (in hDC: Long, in cacheID: Integer)
    
```

```

IMap : IUnknown
- ActiveGraphicsLayer : ILayer
- AnnotationEngine : IAnnotateMap
- AreaOfInterest : IEnvelope
- Barriers (eExtent: IEnvelope) : IBarrierCollection
- BasicGraphicsLayer : IGraphicsLayer
- ClipBorder : IBorder
- ClipGeometry : IGeometry
- Description : String
- DistanceUnits : esriUnits
- Expanded : Boolean
- FeatureSelection : ISelection
- IsFramed : Boolean
- Layer (in Index: Long) : ILayer
- LayerCount : Long
- Layers (UID: UIID, recursive: Boolean) : IEnumLayer
- MapScale : Double
- MapSurround (in Index: Long) : IMapSurround
- MapSurroundCount : Long
- MapUnits : esriUnits
- Name : String
- ReferenceScale : Double
- SelectionCount : Long
- SpatialReference : ISpatialReference
- SpatialReferenceLocked : Boolean
- UseSymbolLevels : Boolean
- AddLayer (in Layer: ILayer)
- AddLayers (in Layers: IEnumLayer, in autoArrange: Boolean)
- AddMapSurround (in MapSurround: IMapSurround)
- ClearLayers
- ClearMapSurrounds
- ClearSelection
- ComputeDistance (in p1: IPoint, in p2: Point) : Double
- CreateMapSurround (in CLSID: UIID, in optionalStyle: IMapSurround) : IMapSurround
- DelayDrawing (in delay: Boolean)
- DelayEvents (in delay: Boolean)
- DeleteLayer (in Layer: ILayer)
- DeleteMapSurround (in MapSurround: IMapSurround)
- GetPageSize (out widthInches: Double, out heightInches: Double)
- MoveLayer (in Layer: ILayer, in toIndex: Long)
- RecalcFullExtent
- SelectByShape (in Shape: IGeometry, in env: ISelectionEnvironment, in justOne: Boolean)
- SelectFeature (in Layer: ILayer, in Feature: IFeature)
- SetPageSize (in widthInches: Double, in heightInches: Double)
    
```

```

IScreenDisplay : IDisplay
- ActiveCache : Integer
- CacheCount : Integer
- CacheMemDC (in Index: Integer) : Long
- CancelTracker : ITrackCancel
- hWnd : Long
- IsFirstCacheTransparent : Boolean
- IsFramed : Boolean
- ScaleContents : Boolean
- SuppressResize : Boolean
- UseScrollBars : Boolean
- WindowDC : Long
- AddCache : Integer
- DoScroll (in xDelta: Long, in yDelta: Long, in updateScreen: Boolean)
- DrawCache (in hDC: Long, in Index: Integer, in deviceRect: tagRECT, in cacheRect: tagRECT)
- Invalidate (in rect: IEnvelope, in erase: Boolean, in cacheIndex: Integer)
- IsCacheDirty (in cacheIndex: Integer) : Boolean
- PanMoveTo (in mouseLocation: IPoint)
- PanStart (in mouseLocation: IPoint)
- PanStop : IEnvelope
- RemoveAllCaches
- RemoveCache (in cacheID: Integer)
- RotateMoveTo (in pPoint: IPoint)
- RotateStart (in mousePt: IPoint, in centerPt: Point)
- RotateStop : Double
- RotateTimer
- SetScrollbarHandles (in hWndHorzScrollbar: Long, in hWndVertScrollbar: Long)
- StartRecording
- StopRecording
- TrackPan
- TrackRotate
- UpdateWindow
    
```

5 The layers of the map are looped through. The first raster layer is processed and then the function is exited.

IPoint : IGeometry	
→ ID: Long	
→ M: Double	
→ X: Double	
→ Y: Double	
→ Z: Double	
← Compare (pOtherPoint: IPoint) : Long	
← ConstrainAngle (constraintAngle: Double, Anchor: IPoint, allowOpposite: Boolean)	
← ConstrainDistance (constraintRadius: Double, Anchor: IPoint)	
← ProjCoords (X: Double, Y: Double)	
← QueryCoords (out X: Double, out Y: Double)	

3 The cursor coordinates, in pixels, must be converted to map units. The ToMapPoint method on the IDisplayTransformation interface does this.

DbtPnt	
IPnt O	
→ IPnt : IUnknown	
→ X: Double	
→ Y: Double	
← Convert2Point (in env: IPoint)	
← Set2Point (in env: IPoint)	
← SetCoords (in X: Double, in Y: Double)	

4 A dbtPoint object is created and the coordinates are set to 1.0, 1.0. This will be used to define the size of the pixel block used to interrogate the raster.

IRasterProps : IUnknown	
→ Extent: IEnvelope	
→ Height: Long	
→ IsInteger: Boolean	
→ NoDataValue: Variant	
→ PixelType: rstPixelType	
→ SpatialReference: ISpatialReference	
→ Width: Long	
← MeanCellSize: IPnt	

9 The coordinates of the cursor are calculated in raster pixel units.

8 The IRasterProps interface is obtained. This provides information about the extent of the raster in both real-world units and pixels.

DataSources- Raster

6 The ILayer interface is accessed through a QueryInterface for the IRasterLayer interface. This interface gives access to raster-specific properties of the layer.

IRasterLayer : ILayer	
→ BandCount: Long	
→ ColumnCount: Long	
→ DataframeExtent: IEnvelope	
→ DisplayResolutionFactor: Long	
→ FilePath: String	
→ PrimaryField: Long	
→ PyramidPresent: Boolean	
→ Raster: IRaster	
→ Renderer: IRasterRenderer	
→ RowCount: Long	
→ ShowResolution: Boolean	
→ VisibleExtent: IEnvelope	
← CreateFromDataset (in RasterDataset: IRasterDataset)	
← CreateFromFilePath (in FilePath: String)	
← CreateFromRaster (in Raster: IRaster)	

IRaster : IUnknown	
→ ResampleMethod: rstResamplingTypes	
← CreateCursor: IRasterCursor	
← CreatePixelBlock (in Size: IPnt) : IPixelBlock	
← Read (in tlc: IPnt, in block: IPixelBlock)	

7 A pixel block the size of one pixel is created.

11 The planes or the raster are looped over, extracting the pixel values.

10 The pixel block for the raster location is populated.

IPixelBlock : IUnknown	
→ BytesPerPixel: Long	
→ Height: Long	
→ PixelType (in plane: Long) : rstPixelType	
→ Planes: Long	
→ SafeArray (in plane: Long) : Variant	
→ Width: Long	
← GetVal (in plane: Long, in X: Long, in Y: Long) : Variant	

12 Checks are made to ensure that there are raster values present at the location. If there are, they are appended to the value string.

Add this code to the MouseMove event of a UIToolControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
    Set pMxDoc = ThisDocument

    Dim pActiveView As IActiveView
2 Set pActiveView = pMxDoc.FocusMap
    Dim pPoint As IPnt
3 Set pPoint = pActiveView.ScreenDisplay.DisplayTransformation.ToMapPoint(x, y)

    Dim pBlockSize As IPnt
4 Set pBlockSize = New DbtPnt
    pBlockSize.SetCoords 1#, 1#

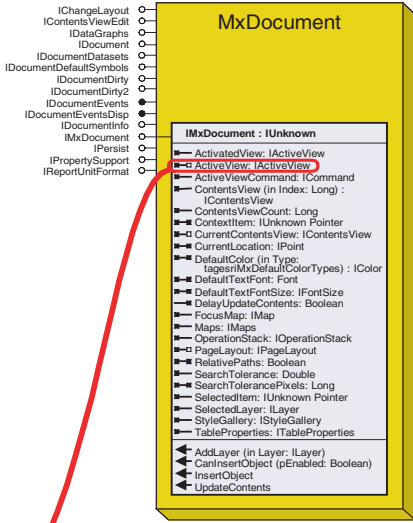
    Dim pLayer As IRasterLayer
    Dim pPixelBlock As IPixelBlock
    Dim vValue As Variant
    Dim i As Long, j As Long
    Dim sPixelVals As String
    sPixelVals = "No Raster"
    Dim pRasterProps As IRasterProps
    Dim dxSize As Double, dySize As Double
    Dim pPixel As IPnt
    Set pPixel = New DbtPnt
5 For i = 0 To pMxDoc.FocusMap.LayerCount - 1
    If (TypeOf pMxDoc.FocusMap.Layer(i) Is IRasterLayer) Then
6     Set pLayer = pMxDoc.FocusMap.Layer(i)
7     Set pPixelBlock = pLayer.Raster.CreatePixelBlock(pBlockSize)
8     Set pRasterProps = pLayer.Raster
        dxSize = pRasterProps.Extent.XMax - pRasterProps.Extent.XMin
        dySize = pRasterProps.Extent.YMax - pRasterProps.Extent.YMin
        dxSize = dxSize / pRasterProps.Width
        dySize = dySize / pRasterProps.Height
9     pPixel.x = (pPoint.x - pRasterProps.Extent.XMin) / dxSize
        pPixel.y = (pRasterProps.Extent.YMax - pPoint.y) / dySize
10    pLayer.Raster.Read pPixel, pPixelBlock
    For j = 0 To pPixelBlock.Planes - 1
11      If (sPixelVals = "No Raster") Then
        sPixelVals = "("
      Else
        sPixelVals = sPixelVals & ", "
      End If
      vValue = pPixelBlock.GetVal(j, 0, 0)
      sPixelVals = sPixelVals & CStr(vValue)
    Next j
    If (sPixelVals <> "No Raster") Then sPixelVals = sPixelVals & ")"
12    ThisDocument.Parent.StatusBar.Message(0) = "Raster value = " & sPixelVals
  Exit For
  End If
Next I

```

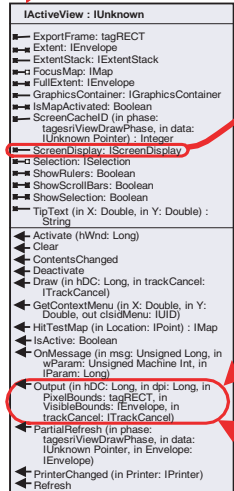
Export current view

This sample takes the current active view and exports it to a JPEG file. This code is similar to the next sample, which prints the active view to a PostScript printer.

1 The `IMxDocument` interface is obtained from the `ThisDocument` global variable.



ArcMapUI



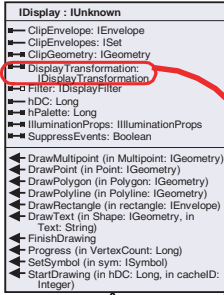
8 A call to the active view's `Output` method writes the current view to the exporter. Notice the `hDC` required by the `Output` method is obtained by calling `StartExporting` on the `Exporter`.

3 A new `ExportJPEG` object is created and the `IExporter` interface is obtained. The filename and resolution are set.



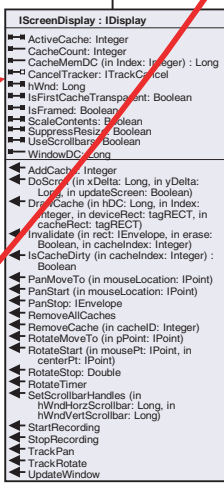
Output

The driver bounds envelope is populated with the coordinates from the device rectangle. This envelope is used to set the `IExporter PixelBounds` property.

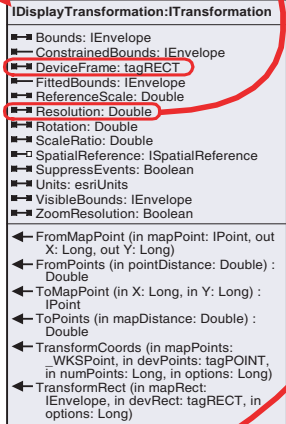


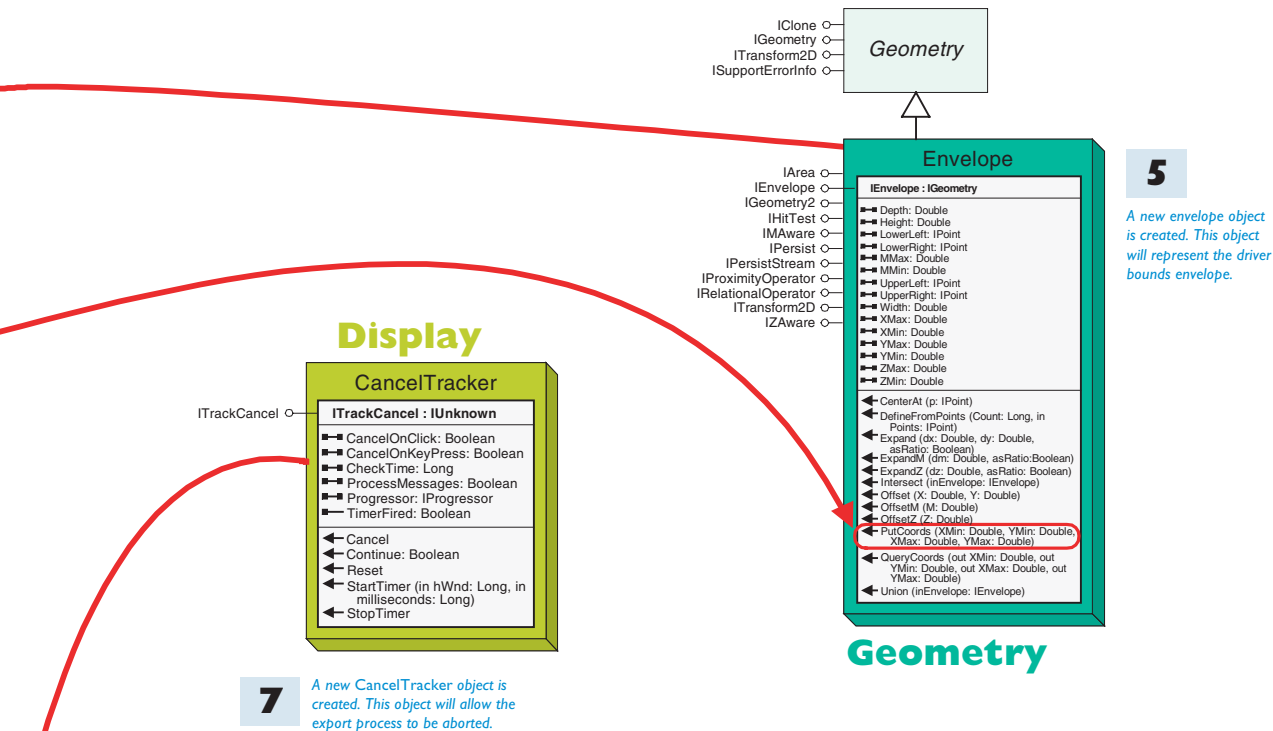
6

2 For convenience, the resolution of the screen is set to a local variable.



4 The device rectangle is stored as a local variable.





Add this code to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim lScrRes As Long
  lScrRes = 96

  Dim pExport As IExport
  Set pExport = New ExportJPEG
3 pExport.ExportFileName = "C:\Export.jpg"
  pExport.Resolution = lScrRes

4 Dim deviceRECT As tagRECT
  deviceRECT = pMxDoc.ActiveView.ScreenDisplay.DisplayTransformation.DeviceFrame

5 Dim pDriverBounds As IEnvelope
  Set pDriverBounds = New Envelope

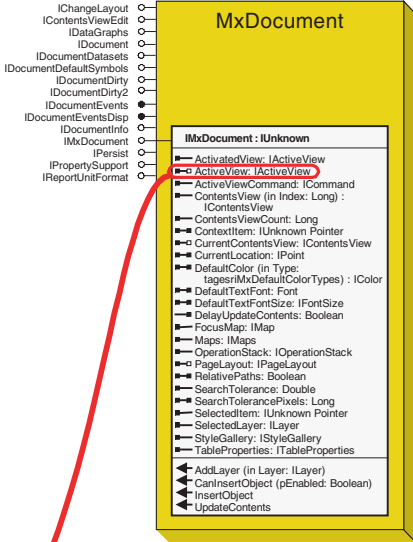
  pPixelBounds.PutCoords deviceRECT.Left, deviceRECT.bottom, deviceRECT.Right, _
  deviceRECT.Top
6 pExporter.PixelBounds = pDriverBounds

7 Dim pCancel As ITrackCancel
  Set pCancel = New CancelTracker

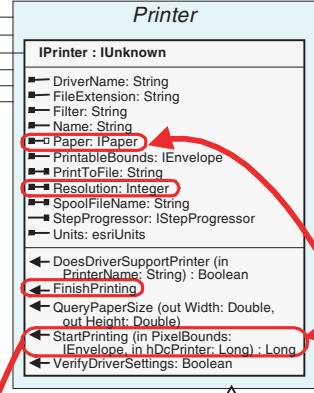
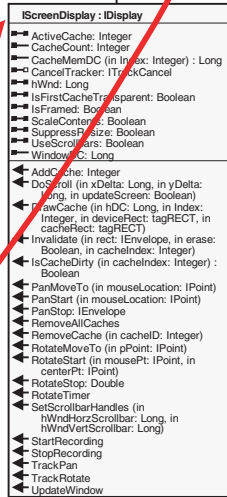
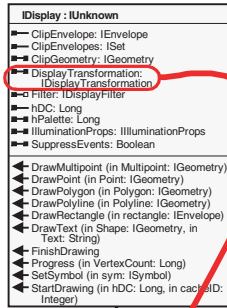
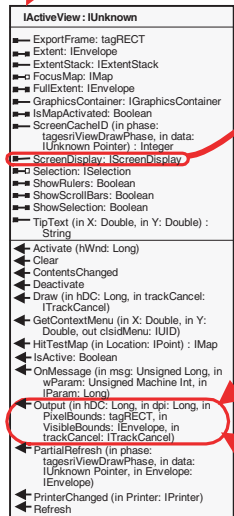
8 pMxDoc.ActiveView.Output pExport.StartExporting, lScrRes, deviceRECT, _
  nothing, pCancel

9 pExport.FinishExporting
  
```

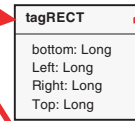
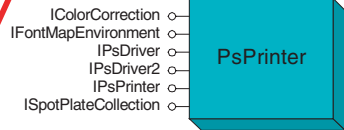
This sample takes the currently active view and prints the file to a PostScript printer. This code is similar to the previous sample, which exports the active view to a JPEG file.



ArcMapUI



Output



1 The IMxDocument interface is used to access the active view.

3 A new PsPrinter object is created and the IPrinter interface is obtained.

6 The resolution of the screen is then passed to the printer.

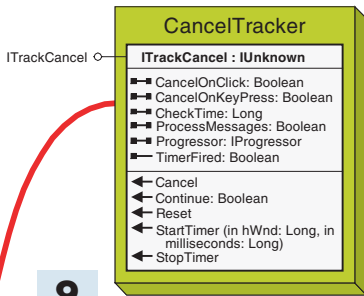
11 Finally, the FinishPrinting method is called. This call ensures that the drawing is completed and the printer receives the plot.

2 For convenience, the resolution of the screen is set to a local variable.

7 For convenience, the device rectangle is stored as a local variable.

10 A call to the active view's Output method exports the current view to the printer. Notice the hDC required by the Output method is obtained by calling the StartPrinting method of the Printer.

Display

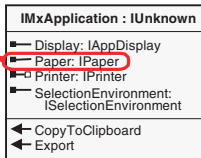


9

A new CancelTracker object is created. This object allows the printing process to be aborted.

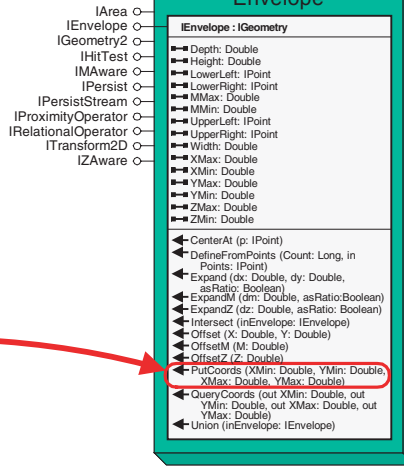
5

The paper object used by the application is set into the printer object.



4

The IMxApplication interface on the application object is required in order to get the page details. This interface is obtained by performing a QueryInterface on the Parent property of the ThisDocument variable.



8

A new envelope object is created. This object will represent the driver bounds. The driver bounds envelope is populated with the coordinates from the device rectangle. This envelope is used to set the IPrinter PixelBounds property.

Geometry

Add this code to the Click event of a UIButtonControl in ArcMap.

```

1 Dim pMxDoc As IMxDocument
  Set pMxDoc = ThisDocument

2 Dim lScrRes As Long
  lScrRes = pMxDoc.ActiveView.ScreenDisplay.DisplayTransformation.Resolution

3 Dim pPrinter As IPrinter
  Set pPrinter = New PsPrinter

4 Dim pMxApp As IMxApplication
  Set pMxApp = ThisDocument.Parent

5 Set pPrinter.Paper = pMxApp.Paper
6 pPrinter.Resolution = lScrRes

7 Dim deviceRECT As tagRECT
  deviceRECT = pMxDoc.ActiveView.ScreenDisplay.DisplayTransformation.DeviceFrame

8 Dim pDriverBounds As IEnvelope
  Set pDriverBounds = New Envelope
  pDriverBounds.PutCoords deviceRECT.Left, deviceRECT.bottom, deviceRECT.Right, _
  deviceRECT.Top

9 Dim pCancel As ITrackCancel
  Set pCancel = New CancelTracker

10 pMxDoc.ActiveView.Output pPrinter.StartPrinting(pDriverBounds, 0), lScrRes, _
  deviceRECT, pMxDoc.ActiveView.Extent, pCancel
11 pPrinter.FinishPrinting
  
```

This command takes the current displayed data layer and draws the data extent in a thick red line in the preview.

2 If the current view is not a Preview, the procedure is exited.

3 To access the preview-specific properties, the IGxPreview interface is accessed through a QueryInterface call on the IGxView interface.

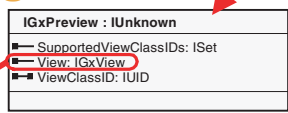
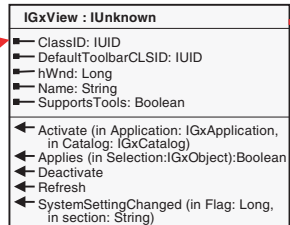
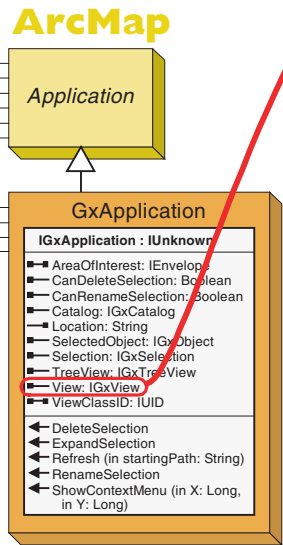
4 There are potentially many types of previews. If it is not a geographic preview, the procedure is exited.

Finally, with the symbol and geometry of the extent obtained, the extent is drawn on the screen.

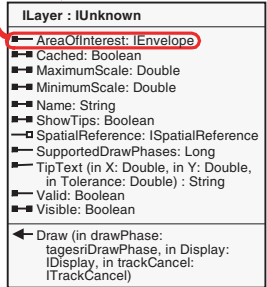
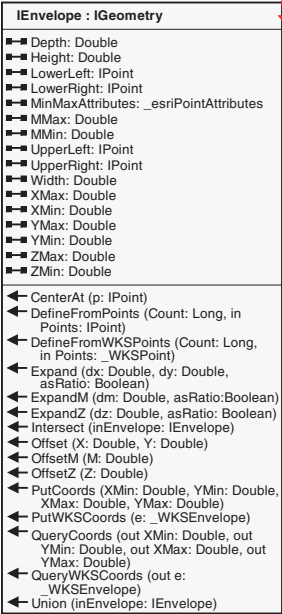
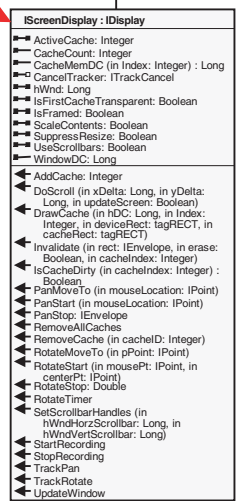
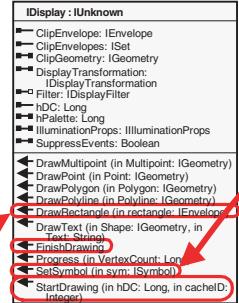
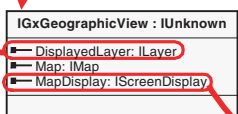
5 The extent of the currently displayed layer is assigned to an envelope variable.



1 The IGxApplication interface is obtained by accessing the ApplicationGlobal variable.

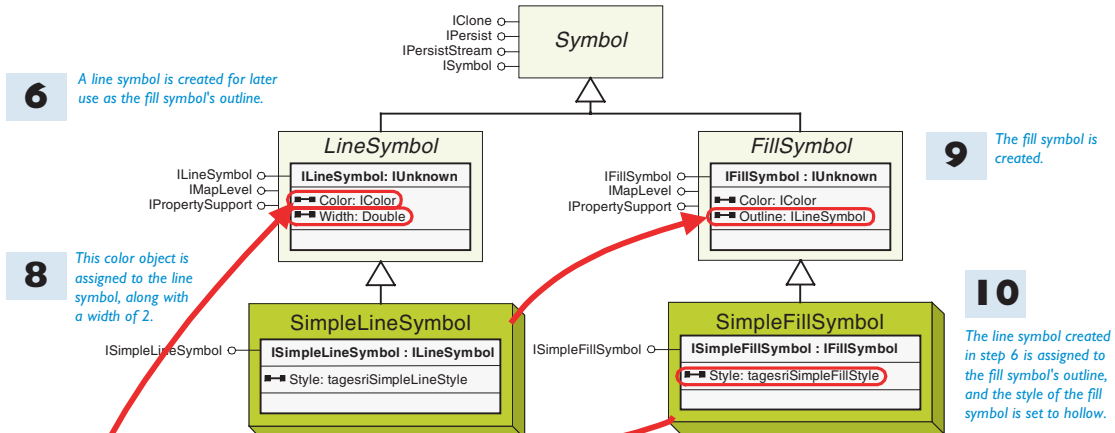


CatalogUI



Carto

Geometry



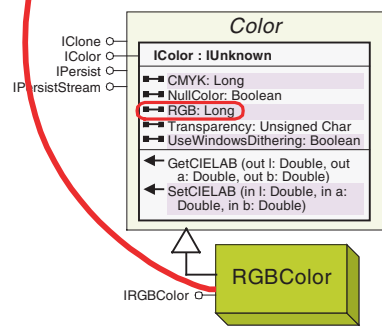
6 A line symbol is created for later use as the fill symbol's outline.

9 The fill symbol is created.

8 This color object is assigned to the line symbol, along with a width of 2.

10 The line symbol created in step 6 is assigned to the fill symbol's outline, and the style of the fill symbol is set to hollow.

Display



7 An RGB color object is created and its color set to red.

Add this to the Click event of a UIButtonControl in ArcCatalog.

```

1 Dim pGxApp As IGxApplication
    Set pGxApp = Application

2 Dim pGxView As IGxView
    Set pGxView = pGxApp.View
    If (TypeOf pGxView Is IGxPreview) Then
3     Dim pGxPreview As IGxPreview
        Set pGxPreview = pGxView

        If (TypeOf pGxPreview.View Is IGxGeographicView) Then
4             Dim pGxGeoView As IGxGeographicView
                Set pGxGeoView = pGxPreview.View

5             Dim pEnv As IEnvelope
                Set pEnv = pGxGeoView.DisplayedLayer.AreaOfInterest

6             Dim pLineStyle As ISimpleLineStyle
                Set pLineStyle = New SimpleLineStyle

7             Dim pColor As IColor
                Set pColor = New RgbColor

8             pColor.RGB = vbRed
                With pLineStyle
                    .Color = pColor
                    .Width = 2
                End With

9             Dim pFillSymbol As ISimpleFillSymbol
                Set pFillSymbol = New SimpleFillSymbol

10            With pFillSymbol
                .Style = esriSFShollow
                .Outline = pLineStyle
            End With

11            With pGxGeoView.MapDisplay
                .StartDrawing 0, esriNoScreenCache
                .SetSymbol pFillSymbol
                .DrawRectangle pEnv
                .FinishDrawing
            End With
        End If
    End If
  
```

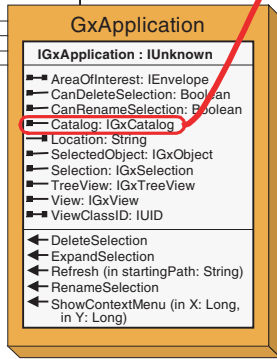
This code sample inspects the selected objects in the ArcCatalog browser and if they are feature classes in a geodatabase, makes an edit to their alias name.

ArcMap

- IApplication
- IDockableWindowManager
- IExtensionManager
- IMultiThreadedApplication
- IVBAAApplication
- IWindowPosition



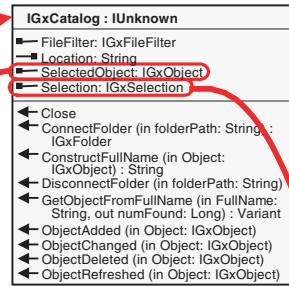
- IGxApplication
- IGxCatalogEvents
- IGxCatalogEventsDisp
- IGxViewContainer



ArcCatalog

2

For convenience, the IGxCatalog interface is stored as a local variable.

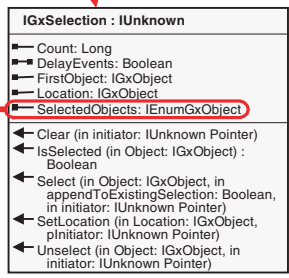


3

The selection of GxObjects is obtained from the Catalog.

4

The SelectedObjects property is accessed through a QueryInterface for an enumerator. This will allow you to iterate over all the selected objects within the Catalog.

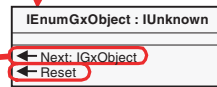


1

The IGxApplication interface is obtained by accessing the Application global variable.

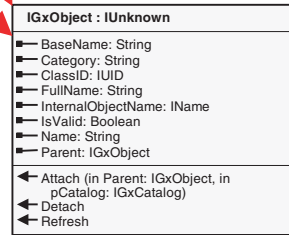
5

Start iterating by asking the enumerator for its next object. This is repeated until the enumerator returns nothing.



6

Check for Null. If it is Null, use the selected object from the Catalog and not the enumerator.

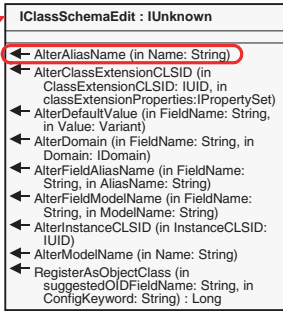


7

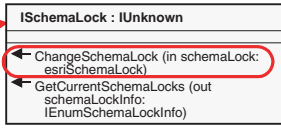
The type of the GxObject is checked. If it supports the IGxDataset interface, its type is a feature class, and the workspace type is not filesystem, it is processed. Otherwise it is skipped.

13

Notice the error handling code that checks for a specific error return value.

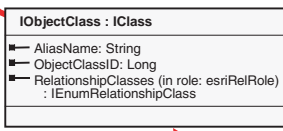


9
The method to edit the schema is on the IClassSchemaEdit interface. This is accessed through a QueryInterface from the IObjectClass interface.

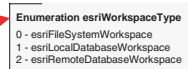
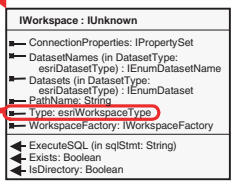
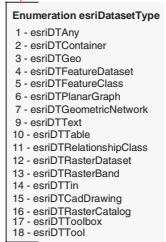
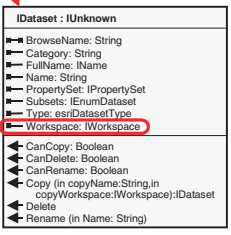
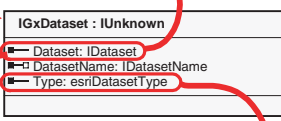


11 The schema edit is made.
12 The exclusive lock is released.

10 It is possible that when you ask the database for an exclusive lock it will fail because another user is editing, hence you must prepare for this with a specialized error handler.



8
To make the schema change, you must have a schema lock. The schema lock interface is accessed through a QueryInterface from the IObjectClass interface.



Geodatabase

Add this to the Click event of a command in ArcCatalog.

```

1 Dim pGxApp As IGxApplication
  Set pGxApp = Application

2 Dim pGxCatalog As IGxCatalog
  Set pGxCatalog = pGxApp.Catalog

3 Dim pGxSelection As IGxSelection
  Set pGxSelection = pGxCatalog.Selection

4 Dim pGxObjects As IEnumGxObject
  Set pGxObjects = pGxSelection.SelectedObjects
  pGxObjects.Reset

5 Dim pGxObject As IGxObject
  Set pGxObject = pGxObjects.Next

6 If (pGxObject Is Nothing) Then Set pGxObject =
  pGxCatalog.SelectedObject

7 Dim pGxDataset As IGxDataset
  Dim pObjectClass As IObjectClass
  Dim pClassSchemaEdit As IClassSchemaEdit
  Dim pSchemaLock As ISchemaLock
  Do Until (pGxObject Is Nothing)
    If (TypeOf pGxObject Is IGxDataset) Then
      Set pGxDataset = pGxObject
      If ((pGxDataset.Type = esriDTFeatureClass) And _
        (pGxDataset.Dataset.Workspace.Type <> _
          esriFileSystemWorkspace)) Then
        Set pObjectClass = pGxDataset.Dataset
        Set pSchemaLock = pObjectClass
        Set pSchemaLock = pObjectClass

8
9
10 On Error GoTo lockDB

11 pSchemaLock.ChangeSchemaLock esriExclusiveSchemaLock
    On Error GoTo 0
    pClassSchemaEdit.AlterAliasName "ArcObjects Updated Alias"
12 pSchemaLock.ChangeSchemaLock esriSharedSchemaLock
      End If
    End If
    Set pGxObject = pGxObjects.Next
  Loop

Exit Sub

lockDB:

13 If (Err.Number = FDO_E_SCHEMA_LOCK_CONFLICT) Then
  MsgBox "Unable to obtain exclusive database lock",
  vbExclamation + vbOKOnly, "Database Lock Error"
Else
  MsgBox "Unknown error getting schema lock", vbExclamation +
  vbOKOnly, "Database Error"
End If
Err.Clear

```




D

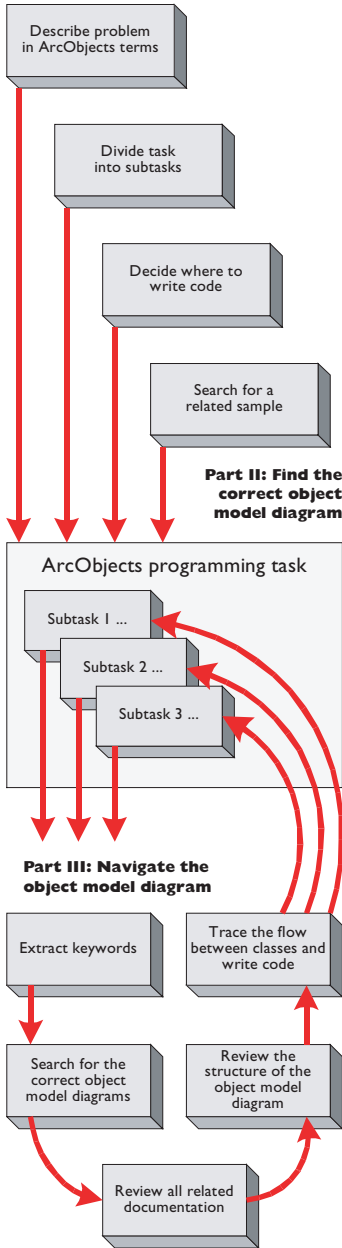
ArcObjects problem-solving guide

The ArcObjects problem-solving guide presents a methodology to help you solve real-world ArcObjects programming tasks that customize or extend the ArcGIS Desktop applications.

The guide helps you describe and categorize your task and documents how to use the help resources and tools to solve the problem programmatically. In the end, the guide will not only help solve individual problems but will also help you understand and navigate the structure of ArcObjects.

Steps of the ArcObjects problem solving guide

Part I: Define the ArcObjects programming task



The ArcObjects library is a comprehensive set of COM components designed to provide developers with the ability to extend and customize ArcGIS applications, such as ArcMap and ArcCatalog. The ArcObjects library consists of more than 1,000 classes and 2,000 interfaces that are visually documented in several dozen object model diagrams.

With this extensive set of classes, you can create a wide variety of customizations and custom applications to extend existing ArcGIS applications. However, as you begin developing with ArcObjects, you may find the extent of the ArcObjects library overwhelming, and it may be difficult to know where to begin. The goal of this problem-solving guide is to present a methodology to help you solve real-world ArcObjects programming tasks that customize or extend the ArcGIS Desktop applications.

The guide helps you describe and categorize your task and documents how to use the help resources and tools to solve the problem programmatically. In the end, the guide will not only help solve individual problems but will also help you understand and navigate the structure of ArcObjects.

The guide is broken into three parts. Part One is designed to help you define the ArcObjects programming task as clearly as possible. Part Two illustrates how to use the help resources to locate the correct object model diagram you should start with. Part Three provides an example of how to navigate the object model diagrams to assemble the code required to solve the task.

The following steps outline each part of the problem-solving guide:

PART ONE: DEFINE THE ARCOBJECTS PROGRAMMING TASK

1. Describe the problem in ArcObjects terms.
2. Identify subtasks.
3. Decide where to write the code.
4. Search for a related sample or recommended methodology.

PART TWO: LOCATE THE CORRECT OBJECT MODEL

1. Identify a subtask.
2. Extract keywords.
3. Search for the correct object model diagrams.
4. Review all related documentation.

PART THREE: NAVIGATE THE OBJECT MODEL DIAGRAM

1. Review the structure of the object model diagram.
2. Trace the flow between classes and assemble code.

Although there are three parts, this type of problem solving is really one continuous process. You may find it necessary to revisit some steps as you gain knowledge about a particular topic by reading the pages in this book and exploring the wide variety of code samples available.

ARE YOU READY?

Before getting started with this problem-solving guide, you should be familiar with the basic terminology behind COM and ArcObjects, and you should know how to use the available help resources and tools. Here is a checklist of some topics discussed earlier in this developer guide that should be familiar to you:

- How to program with COM interfaces and classes in Visual Basic (Appendix A, ‘Developer environments’)
- How to use the ArcGIS Developer Help system
- How to read and interpret the ArcObjects object model diagrams with Acrobat Reader
- How to use ESRI’s object browser, *EOBrowser*, to inspect the structure of ArcObjects not visible with other object browsers
- How to access continually updated information at ESRI’s technical resource Web site, <http://arcgisdeveloperonline.esri.com>

It is particularly important to understand the previous appendixes in this guide along with the illustrated code samples before starting with this problem-solving guide.

The best way to learn ArcObjects is to first become familiar with the fundamental ArcGIS and COM terminology and concepts, then learn how to effectively use all the help resources, tools, documentation, and samples that are at your disposal. Chapter 2, ‘ArcGIS software architecture,’ and Appendix A, ‘Developer environments,’ of this book provide a good foundation for the basic terms and concepts, while this section focuses specifically on how to use the help resources to solve programming tasks related to ArcObjects.

For detailed information on Visual Basic and COM programming techniques, reference Appendix A, ‘Developer environments’, in this book.

This guide does not attempt to provide an all-encompassing method for every ArcObjects programming task. It simply provides a methodology that can help you clearly define your initial objective and make effective use of the many resources and tools available.

USING THE ARCOBJECTS PROBLEM-SOLVING GUIDE

This problem-solving guide uses a real-world ArcObjects programming problem to explain the details of each step. To learn the methodology behind this guide, first follow the instructions and complete the real-world programming task defined below, then define your own problem and use these steps to solve your own development task.

This problem-solving guide will solve this example task: Add a dataset called States to ArcMap.

PART ONE: DEFINE THE ARCOBJECTS PROGRAMMING TASK

The most important aspect of successfully using the problem-solving guide is being able to define the task itself. A task may originate from a real-world GIS problem at your workplace or may be the result of an enhancement you would like to make to the existing ArcGIS system. A task may be as simple as adding a *UIToolControl* to the user interface of ArcMap to zoom in on the map or as detailed as creating a custom feature for the geodatabase. In either case, to define the task as completely as possible, you should consider the following steps:

1. Describe the problem using ArcGIS terminology.
2. Divide the task into smaller subtasks.
3. Decide where to compile the source code.
4. Find an existing sample or recommended methodology.

To become familiar with basic ArcGIS terminology, refer to these ESRI books: Getting Started with ArcGIS, Building a Geodatabase, and Modeling Our World, as well as the other resources mentioned earlier.

Describe the problem in ArcGIS terms

When defining the problem, it is useful to frame the task with ArcGIS terminology and describe the actions as completely as possible. This will help you find topics in the help system and the relevant components in ArcObjects.

In many cases, this step will also force you to go back and review important background topics and reading materials related to the task at hand. From this research, you will gain further insight about how a particular task can be solved.

For this example, the original task description is Add a dataset called States to ArcMap.

Using ArcGIS terminology, this statement could be expanded like this: Access the States feature class from a personal geodatabase and add it to ArcMap.

The most noticeable change to the description is that it has been expanded by identifying the datasets involved and using the proper ArcObjects terminology. For example, the dataset named States has been more accurately defined as a feature class that resides in an existing personal geodatabase (stored in a Microsoft Access database).

Another important change is that the actions in the description have also been more completely defined. It now reveals the fact that it will be necessary to open the database first, then add a feature class in it to ArcMap. As you will see in the next step, it is important to identify these actions, since they can be treated as two separate programming tasks when building the final code.

Define subtasks

This step forces you to revisit the original task description and determine if it can be broken down into smaller, more manageable subtasks. This process allows you to focus on smaller parts of the original problem at one time and, therefore, smaller sections of the ArcObjects object model diagrams when it comes time to write code. The easiest way to identify subtasks is to look for verbs or action words that are hidden in the description. From the original task description, two subtasks can be easily identified.

From your expanded statement—Access the States feature class from a personal geodatabase and add it to ArcMap—you can identify two subtasks:

- Access the States feature class.
- Add the new layer to the map.

Each subtask will be solved individually as you traverse through Parts Two and Three of this guide. This is important because it enables you to focus on small parts of the problem and smaller sections of the object model diagrams.

Decide where to write the code

With the problem description and subtasks defined, you need to decide where to write the code and how to provide the functionality to end users.

Remember that where you test code and where you write the final code are two different issues. During the testing and initial design phase, it is always recommended to start writing code as a VBA macro in either ArcMap or ArcCatalog.

You should always begin by trying to write ArcObjects code in the VBA environment in ArcMap or ArcCatalog. If necessary, this code can be moved to a different development environment before final compilation and distribution.

There, you can easily assemble, test, and debug the source and experiment with any number of classes or interfaces. After completing the testing phase, you can decide to leave the code as a VBA macro or move it to another format.

Deciding where to write the final application code can be a complicated matter, and as you gain experience developing with ArcObjects, your decision making will improve. In general, the answer is governed by the type of application you are developing and how you want to deliver the functionality to end users.

In general, there are three ways to write ArcObjects code for ArcGIS Desktop applications:

- As a VBA macro in an ArcGIS Desktop application
- As an ActiveX COM component, such as a DLL or OCX
- As a standalone EXE

You should also note that browsing the samples and associated documentation might help you determine where to locate your code. This is covered in detail in the next step.

Writing VBA macros in ArcGIS applications

For information about how to get started with the VBA environment, see the VBA topic in Appendix A, 'Developer environments', as well as related topics in the ArcGIS Developer Help system.

As mentioned, you should start development by using the VBA environment in one of the existing ArcGIS applications. VBA is a simple programming language with many utilities, such as design-time code completion and the object browser, that will help you assemble code quickly.

Here are some more reasons to choose the VBA environment:

- It's fast and easy to create, test, and debug macros inside ArcMap and ArcCatalog.
- The standard ESRI type libraries are already referenced for you.
- Important global variables, such as the *Application* and *Document*, are available.
- It's simple to assemble UI forms using VBA and ActiveX components.
- It's straightforward to integrate VBA code with new ArcObjects UIControls.
- It's relatively easy to migrate VBA code to VB ActiveX DLL projects.
- Many code samples available in the help system are macros that can be cut, pasted, and run within the VBA environment.

After the testing phase, you can easily save the VBA code into a Normal.mxt, Project.mxd, or custom Project.mxt file. Projects, documents, and templates can then be delivered to end users so they can take advantage of the new functionality your application provides. (See the topic on storing customizations in Chapter 3, 'Developing for ArcGIS Desktop applications'.)

Writing ActiveX COM components

This approach of writing ActiveX COM components must be taken if you want to extend the existing ArcObjects architecture. Custom components can reside at the application or geodatabase level.

If you want to use a programming language other than VBA or if you want to package ArcObjects functionality into a COM DLL, EXE, or OCX, you will have to work outside the VBA development environment. This approach generally requires creating a project, referencing ArcObject type libraries, adding code, then compiling the source into a binary file.

Writing ActiveX COM components should be done when you want to extend the existing ArcObjects architecture by adding new custom components. The process requires implementing one or more ArcObjects interfaces in the new object.

Unlike working in the VBA environment, all new components require Component Category registration to work correctly.

Both these topics are discussed in Chapter 3, ‘Developing for ArcGIS Desktop applications’.

These are some advantages of building custom components:

- They can be easily delivered to end users via custom setup programs.
- You can hide ArcObjects code in a binary file and deliver the functionality to end users with a setup program.
- You can extend and customize virtually every aspect of the ArcGIS technology.

Components can be broadly categorized into two areas of customization: those that reside at the application level, such as custom buttons, toolbars, windows, and extensions, and those that reside at the geodatabase level, such as custom feature class extensions and custom features. Some of these more advanced customizations cannot be accomplished through the VBA environment.

The main disadvantage of working outside the VBA environment is that you will have to acquire and use another COM-compliant development tool. Another consideration is the fact that you do not have direct access to the *Application* and *ThisDocument* global variables.

The development tool you choose must support the creation of new components as well as the implementation of COM interfaces to acquire a hook back into the ArcGIS applications. For more details, see Chapter 3, ‘Developing for ArcGIS Desktop applications’. Interfaces that provide this functionality will allow you to acquire references to the *Application* and *ThisDocument* global variables, just as if you were working in the VBA environment. Another disadvantage is that it is often more difficult to debug the code. See the topic on Visual Basic in Appendix A, ‘Developer environments’.

Standalone applications

ArcObjects can be used to write standalone applications. This generally requires creating a project, referencing ArcObject type libraries, then assembling the required code to support the functionality of the application.

There are several ESRI controls that can help you embed ArcObjects functionality in your application. However, as an ArcGIS Desktop developer you can only work with the ESRI *Map* and *PageLayout* controls. The ArcGIS Engine Developer Kit provides more controls and functionality and is the recommended solution for creating complex standalone applications.

More information on ArcGIS Engine can be found in the *What is ArcGIS?* book.

For information on building standalone applications with ArcGIS Engine, refer to the ArcGIS Engine Developer Guide.

Following are some advantages of building standalone applications with either ArcGIS Desktop or ArcGIS Engine:

- You can use the ESRI Map control to simplify the embedding of ArcObjects functionality in your application.
- You can design a highly customized user interface specific to your application.
- You can quickly create small, lightweight applications.

These are the disadvantages of building standalone applications:

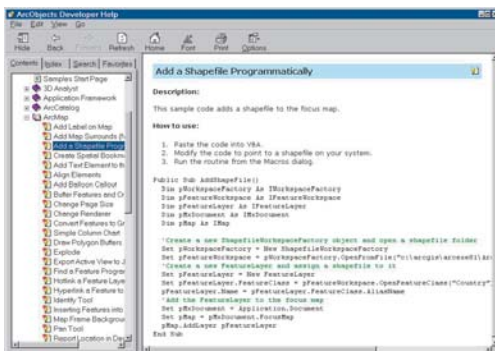
- You cannot take advantage of the extensive functionality that ESRI has built into the existing ArcGIS applications, such as ArcMap or ArcCatalog.
- If you are not using the Map control, you will have to provide your own map display for visual applications.
- You will have to design your own data loading and layer management tools.
- You cannot use ArcMap documents or templates to their fullest capacity.
- You cannot take advantage of the components that give you the ability to extend the existing ArcMap and ArcCatalog framework.
- None of the extensions, including the Editor, can be used.

Although it is possible, it is not recommended to create standalone applications if the functionality you desire can be realized by extending existing ArcGIS applications, such as ArcMap and ArcCatalog. All ArcGIS applications share the same application framework, designed to be extended by third-party developers.

If you create a standalone application, you have a significantly higher development effort. The Map control mitigates but does not eliminate this additional effort. Standalone applications are appropriate only for highly specialized implementations.

Of the three options for writing code—as VBA macros in ArcMap or ArcCatalog, as ActiveX COM components, or as standalone applications—the example used in this problem-solving guide, adding a dataset called States to ArcMap, will simply be run as a VBA macro stored in a map document (.mxd file).

Find a related sample or recommended methodology



The last step is to search all the available resources for a code sample and look for any documentation that may be related to the task at hand. To accomplish this, you will need to make use of the help resources and tools. As you may already know, there is often more than one way to accomplish a programming task. The recommendation here is to search the available resources for similar implementations to help you decide how to go about solving the problem.

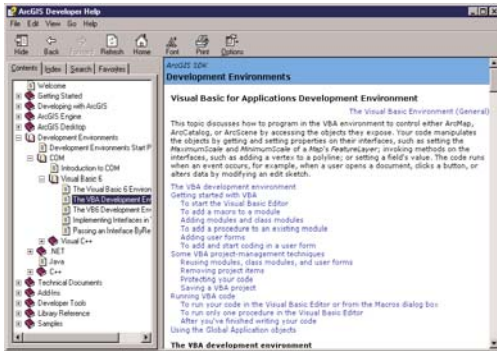
The easiest way to locate a sample is to search using the ArcGIS Developer Help system.

1. Start the ArcGIS Developer Help system.
2. Click the Search tab and type “Add”.

The samples in the ArcGIS Developer Help system fall into two categories: tips and tools. Tips are smaller examples of ArcObjects code that you can generally cut and paste, then run as a VBA script in ArcMap or ArcCatalog. Tools are more complete examples of applications that often require compilation and component category registration. Many of the tools are COM components themselves. If you find a tip or tool that may be useful, be sure to store it in the Favorites tab for future reference.

3. Sort by clicking the Title field. You can sort by location as well.
4. Browse down until you find “samples” and locate the “Add a shapefile programmatically” sample. Open the page and study the sample.
5. Click the Contents tab. This reveals the location of the sample. Browse the other samples in this folder structure. Make note of the location of the sample.
6. Click the Favorites tab, give the current topic a title, and add the sample to your Favorites list.

Unfortunately, in this case it was not possible to find a sample that solves the exact problem, but a sample was found that relates to the problem. The sample found illustrates how to open and load a shapefile into ArcMap. Since you are not ready to write code at this point, the sample was simply stored in the Favorites list so it can be referenced later. This will still prove to be a valuable step later when writing code.



Whether a sample was located or not, it is a good idea to look for background information related to the current task. The ArcGIS Developer Help system contains some topics that you might find valuable in the development environments section. These pages provide some useful information, such as the basic principles related to working with ArcObjects in VB and VBA. Although the documentation doesn't relate to the problem description, it still relates to the overall task since this example will be written as a VBA macro. Therefore, it is a good idea to review this documentation.

1. Open the Development Environments section in the ArcGIS Developer Help system.
2. Open the COM and Visual Basic 6 subsections, then review the documentation related to the VBA development environment.

If nothing is found that directly relates to the task at hand, it is a good idea to visit the other documentation available. You can check some other resources, such as the ArcGIS Desktop Help, and ESRI books, such as *What is ArcGIS?*, *Building a Geodatabase*, and *Modeling Our World*.

Summary of Part One

Now that you have more clearly defined the various components of the task and have done some research on the topic, it is possible to move on to the next step, which will help identify which object model diagram to start with.

Here is all the task-related information found in Part One of the problem-solving guide for the current example:

Task defined in ArcGIS terminology: Access the States feature class from an existing Access personal geodatabase and add it to ArcMap.

Subtask 1: Access the States feature class.

Subtask 2: Add the new layer to the map.

Where to write the code: As a VBA macro in ArcMap.

Located sample: Add a shapefile to ArcMap programmatically.

PART TWO: FIND THE CORRECT OBJECT MODEL DIAGRAM

This section explains how to use the help resources and tools to locate the correct object model diagram required to solve a task. As a reminder, the remaining steps in Parts Two and Three are designed to work through one subtask at a time.

Therefore, you will need to proceed through all the remaining steps with Subtask 1, then come back here to solve Subtask 2.

Identify a subtask

Start with the first subtask defined in Part One.

Original task: Access the States feature class from an existing Access personal geodatabase and add it to ArcMap.

Subtask 1: Access the States feature class.

Subtask 2: Add the new layer to ArcMap.

Extract keywords

It is important to use the correct ArcObjects terminology when describing the original task to extract meaningful keywords from each subtask. These keywords are important because they can be used later to search for topics in the help system and for classes in the object model diagrams.

This step requires that you extract keywords from the subtask description. This is not an exact science, but the more ArcObjects terms used in the original description, the more success you will have here. Therefore, it should be evident that it is critical to define the initial task correctly in the first step of Part One.

Two terms can be extracted from the previously defined subtask: “access” and “feature class”.

Search for the correct object model diagram

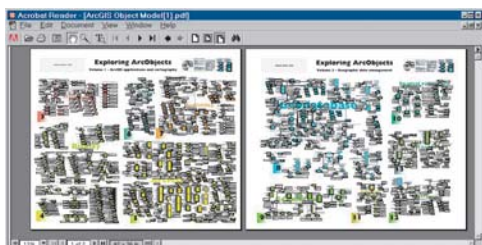
The objective of this step is to use the keywords defined above to identify the correct object model diagram. One way to find the object model diagram is to use Adobe Acrobat Reader to search the ArcGIS object model PDF file. Searching the entire ArcGIS object model should lead you to one or more words or classes that are directly associated with an object model diagram.

The ArcGIS object model is a simplified version of the entire ArcObjects library. This object model contains subsystems that are composed of one or more object model diagrams.

The methodology here is to search the object model with the keywords defined in the last step, identify the appropriate subsystem or object model diagram, then go directly to the associated chapter in

the book to learn more about the related classes. The chapters of the book provide both a detailed description of the classes and a number of helpful code samples.

Another method to find the object model diagram is to search the AllOMDs.pdf file. This diagram contains all the object model diagrams with expanded inter-

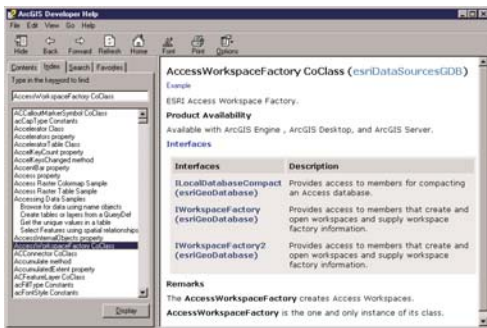


The ArcGIS Object Model.pdf file contains subsystems that contain one or more object model diagrams. This diagram only shows those classes that are documented in the ArcObjects book. To search against the entire ArcObjects library, you can also use the AllOMDs.pdf file.

faces, members, and enumerations. It can be searched using Acrobat Reader just like the ArcGIS object model diagram, but since it contains considerably more detail, expect the search to point to many more hits. The advantage of using this object model diagram is that it will cover virtually every class and interface in the entire ArcObjects library at one time.

The remaining method involves searching the ArcGIS Developer Help system. Here you can enter the keywords into the Index or Search tabs and look for results that return an ArcObjects class, leading you to the object model diagram documenting that class.

Use the Index tab to search for keywords in the ArcGIS Developer Help.



1. Open the ArcGIS Developer Help.
2. Click the Index tab and type in the keyword “access”. You will find the only ArcObject class listed is the *AccessWorkspaceFactory* coclass.
3. Click *AccessWorkspaceFactory* coClass in the list and read the description in the right pane. The library that contains this class is in parentheses next to the class name at the top. In this case it is the *esriDataSourcesGDB* library. The corresponding object model diagram is *DataSourcesGDBObjectModel*. Each library is usually associated with an object model with a similar name.
4. Next type the “featureclass” keyword into the index.

The list will display the *FeatureClass* class.

5. Click the *FeatureClass* class and read the description in the right pane. This class is located in the *esriGeodatabase* library. The corresponding object model diagram is *GeoDatabaseObjectModel*.

In this example, the search results point to two geodatabase libraries and object model diagrams. Therefore, this clearly indicates that you should start with these diagrams to solve the subtask.

Review the documentation

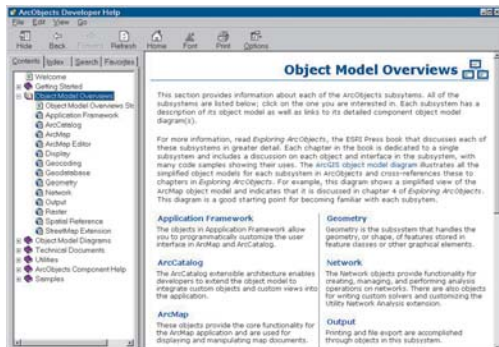
With the object model diagrams identified, the last step in Part Two is to review the available ArcObjects documentation. The best place to start is with the

Object Model Overviews section of the ArcGIS Developer Help system. The Object Model Overviews Start page provides a brief description of each subsystem that composes the ArcObjects library. At a minimum, you will find an overview of each subsystem that provides a description of the main classes associated with each subsystem.

Review the appropriate Object Model Overview page in the ArcGIS Developer Help system.

1. Go to the ArcGIS Developer Help system and click Object Model Overviews.

2. From the Object Model Overviews Start page, click the desired object model. For this example, click *Geodatabase*.



3. Read the overview information available to learn about the classes that belong to the selected object model diagram.

The object model diagram overviews provide some background information for the most important classes in each object model diagram. From this, you should be able to identify new keywords that you may have missed or even class names that are directly related to the current subtask. Add these keywords to the existing keyword list to improve your ability to navigate through the object model diagram.

From the Geodatabase Overview page, you should have been able to identify the following keywords: Access, Feature class, Workspace, and Factory.

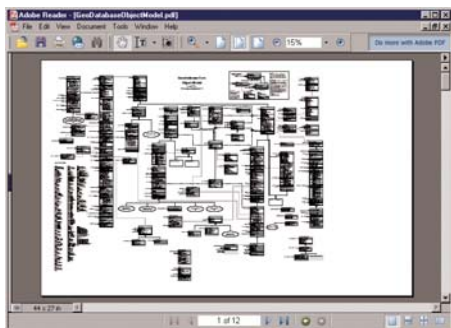
Reviewing this chapter should provide you with a solid understanding of what the main classes and interfaces are for as well as some good code samples. This last step is one of the most important parts of the entire problem-solving guide.

PART THREE: NAVIGATE THE OBJECT MODEL DIAGRAM

The last part of the guide involves navigating the object model diagrams and assembling the required code to solve each subtask. This is generally the most difficult step because it involves the use of many of the help resources and tools and is generally not a linear process. As you become more familiar with the help tools and the object model diagrams, this process will become easier.

Review the structure of the object model diagrams

It is a good idea to familiarize yourself with the general structure of the object model diagrams before proceeding. The easiest way to accomplish this is to use Acrobat Reader to zoom in and pan around each model.



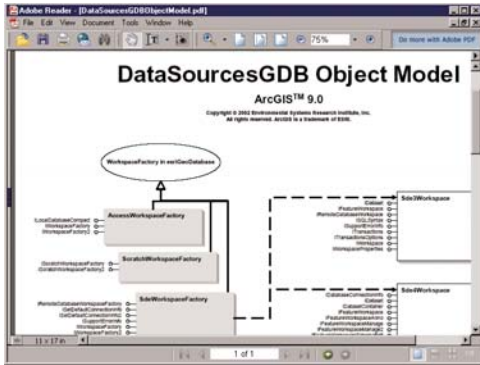
When searching the object model diagrams, it is important to pay attention to the UML symbols that identify relationships between classes. If there is no obvious relationship joining two classes, or if they are located in completely different parts of the model, you should keep in mind that they are still likely associated with each other in some way. It's also important to inspect all the interfaces associated with the classes since they may contain members that are references to other classes.

1. Open the DataSourceGDBObjectModel object model diagram with Acrobat Reader.
2. Zoom in and pan around the diagram to view the overall structure. You will notice that the workspace factories all inherit from the Workspace Factory class in the geodatabase object model.
3. Repeat the steps with the GeoDatabaseObjectModel.

Another way to become familiar with the object model diagram is to examine the relationship between classes and interfaces of an existing sample. It is recommended that you physically trace the flow between the classes and interfaces to understand how the classes relate to one another. This knowledge will be useful since it will help you assemble your own code in the next step.

In Part One, step 4, the “Add a shapefile to ArcMap programmatically” sample was located. Use this to start exploring the geodatabase object model diagram.

1. Click the Favorites tab where you saved the link to this sample in the ArcGIS Developer Help system. Make note of the classes used in this sample.



2. Open the geodatabase object model diagram and search for the main classes used in the sample.
3. Follow the inheritance symbols all the way to the feature class.
4. Pay special attention to any inheritance relationships that may exist.

Trace the flow between the classes and assemble code

In this step you will search for classes in the object model diagrams based on the keywords identified for the current subtask. After locating some potential classes, you will go to the ArcObjects Developer Help system and look for any help topics that may be available. The last step is to start writing the code based on the knowledge you have gained from these steps.

Start with the first subtask by searching for the keywords in the DataSourcesGDBObjectModel object model diagram.

Subtask 1: Access the States feature class.

Keyword List: Access, Feature Class, Workspace, Factory

1. Using Acrobat Reader, zoom in to about 75 percent, and search the DataSourcesGDBObjectModel object model diagram for the first keyword in the list: Access.

You should easily find the *AccessWorkspaceFactory* class.

2. Once you find the class, go back to the ArcObjects Developer Help system and use the Index tab to search for all instances of *AccessWorkspaceFactory*. Once a help topic is located, browse the available information along with any examples. To determine what interfaces the class supports, expand the Interfaces hyperlink on the page. Identify these below.

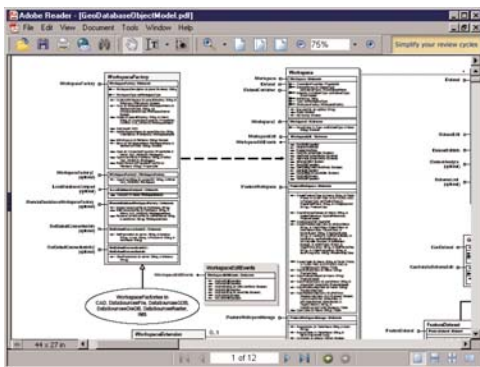
AccessWorkspaceFactory supports the following interfaces: *IWorkspaceFactory*, *IWorkspaceFactory2*, and *ILocalDatabaseCompact*.

If no help topic is available, use the Search tab to find all related help documents in the system, such as samples, you might have missed in the initial steps.

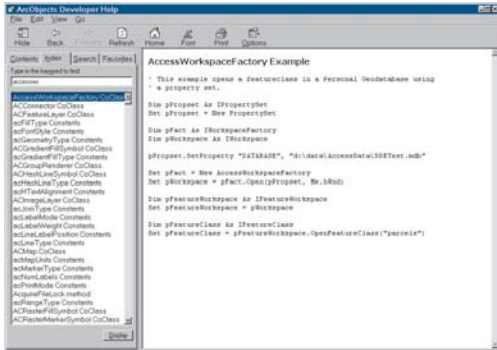
3. Now, return to the object model diagram and follow the inheritance symbols that connect the *AccessWorkspaceFactory* class to the wormhole that says *WorkspaceFactory* in *esriGeodatabase*. A wormhole is a link to another part of the ArcGIS Object model.

4. Open the Geodatabase object model diagram and locate the *WorkspaceFactory* abstract class. Note that the abstract class supports the *IWorkspaceFactory* interface. This information is valuable because it indicates that *AccessWorkspaceFactory* also must implement

WorkspaceFactory. It is important to note that this inheritance information can only be derived from the object model diagram itself or from the discussions in the associated chapters in this book.



At this point, you might also be interested in discovering what other coclasses implement *IWorkspaceFactory*. The easiest way is to look at the coclasses that inherit from *Workspace* on the object model diagram, but this can also be discovered two other ways. The first is to use the ArcObjects Developer Help system and click the Index tab to search for *IWorkspaceFactory*. Expand the CoClasses that implement the *IWorkspaceFactory* hyperlink to list the classes that support the interface.



This will list all the coclasses for you. The second way is to use the ESRI EOBrowser application to search for all the coclasses that implement the same interface.

5. With the information you have gathered from the object model diagram, the sample, the help system, and the EOBrowser, you should be able to write some basic code to cocreate an instance of the *AccessWorkspaceFactory* class. At this point, you could also go back to the ArcGIS Help system and look for an

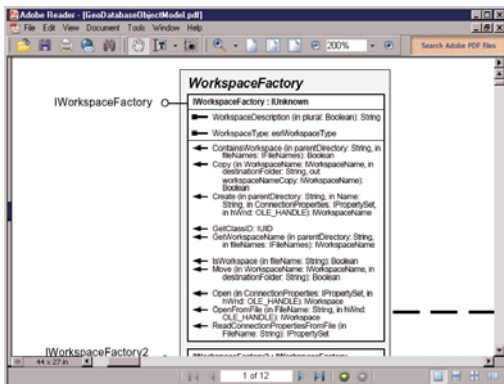
example on the same page that was located for the *AccessWorkspaceFactory* search. With this information and from browsing the object model diagram, the code could be assembled like this:

```
' Subtask 1. Access the States feature class.
Dim pwSF as IWorkspaceFactory
Set pwSF = New AccessWorkspaceFactory
```

6. Now, inspect the members of the *IWorkspaceFactory* interface and try to identify which one can be used to open the geodatabase. Again, this information can be acquired using multiple tools. You can:

- Read and interpret the members on the object model diagram.
- Search for the interface in the ArcGIS Developer Help system by expanding the Members hyperlink.
- Display the members in VB/VBA using IntelliSense or by pressing F2 to view the object browser.
- Search for the interface using the ESRI EOBrowser and expand it to inspect all its members.

Although there are many avenues to take, it is generally recommended to use the ArcGIS Developer Help system since it provides a description of each member, the required parameters, and often a code sample.



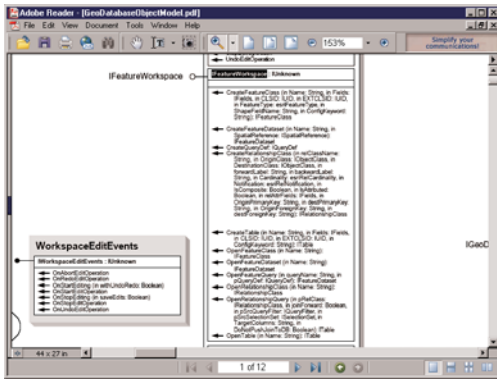
7. After inspecting the members of *IWorkspaceFactory*, it should be obvious that there are multiple members that can be used to open a geodatabase. In this case, since the filepath of the database is known to be C:\data\US.mdb, the *IWorkspaceFactory::OpenFromFile* member can be used. Since the *IWorkspaceFactory::OpenFromFile* member returns a reference to an *IWorkspace* interface, it will be necessary to store this return value.

The code so far might look like this:

```
' Subtask 1. Access the States feature class.
Dim pWSF as IWorkspaceFactory
Dim pWS as IWorkspace
```

```
Set pWSF = New AccessWorkspaceFactory
Set pWS = pWSF.OpenFromFile("c:\data\US.mdb", 0)
```

8. If you inspect the *IWorkspace* interface, you will see that it will take several calls to search and open the *States* feature class if *IWorkspace::Datasets* or *IWorkspace::DatasetNames* are used. In this case it will be necessary to loop through all the feature classes available just to identify the *States* feature class in the enumeration. Since you already know the name of the feature class to open, you should look for a way to optimize this process. The best resource at this point would be the esriGeodatabase library overview in ArcGIS Developer Help, but if you inspect the class carefully, you might find it immediately.



9. If you look at the *Workspace* class on the object model diagram or if you review the esriGeodatabase library overview, you will notice that this class also supports the *IFeatureWorkspace* interface. This interface is designed to provide feature class-level access to a workspace. It supports an *IFeatureWorkspace::OpenFeatureClass* member, which takes a string name directly and returns an *IFeatureClass* reference. Since you can provide the name as a string and directly return a reference, you should use this interface to return a reference to the *States* feature class. To access the interface, it will be necessary to use *QueryInterface* against the *IWorkspace* reference. It should also be noted that the return value must be stored as an *IFeatureClass* reference.

You should recognize that there is often more than one way to solve a problem using the numerous classes and interfaces available in the ArcObjects library. When this is the case, you should research the documentation and test to find out which set of classes and interfaces work best to solve your particular programming task.

After assembling the code it might look like this:

```
' Subtask 1. Access the States feature class.
Dim pWSF as IWorkspaceFactory
Dim pWS as IWorkspace
Dim pFWS as IFeatureWorkspace
Dim pFC as IFeatureClass
```

```
Set pWSF = New AccessWorkspaceFactory
Set pWS = pWSF.OpenFromFile("c:\data\US.mdb", 0)
Set pFWS = pWS ' QI
Set pFC = pFWS.OpenFeatureClass("States")
```


To optimize the code even further, rewrite it as follows:

' Subtask 1. Access the States feature class.

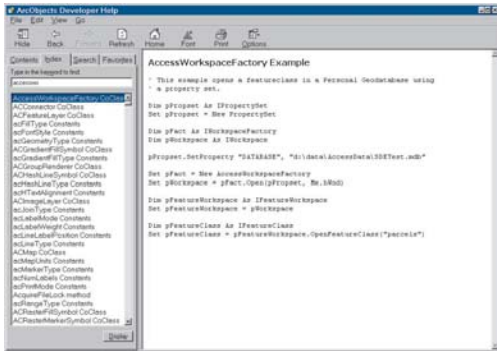
```
Dim pWSF as IWorkspaceFactory
Dim pFWS as IFeatureWorkspace
Dim pFC as IFeatureClass
```

```
Set pWSF = New AccessWorkspaceFactory
Set pFWS = pWSF.OpenFromFile("c:\data\US.mdb", 0)
Set pFC = pFWS.OpenFeatureClass("States")
```

Now that the code for the first subtask has been completed, you must return to Part Two of the problem-solving guide to assemble the code for the last subtask.

If you return to the sample that was identified in Part One, step 4, you will notice that there are classes and interfaces that have not yet been located on an object model diagram. Take this time to look for these classes in the ArcGIS object model diagrams, step 1 of Part Two.

New keyword list: Application, MxDocument, Map, FeatureLayer, Add



1. Using the ArcGIS Developer Help system, click the Favorites tab where you saved the link to this sample. View the sample.
2. Click the Index tab and search for the keywords. Remember, you are looking for items that lead to object classes and, ultimately, library and object model names.

This step reveals matches in the following libraries and object models:

Application coClass in esriArcMap; MxDocument coClass in esriArcMapUI; Map and FeatureLayer coClasses in esriCarto.

Go to step 2, trace the flow between the classes, and assemble code. Start this process by searching for the keywords for the current subtask.

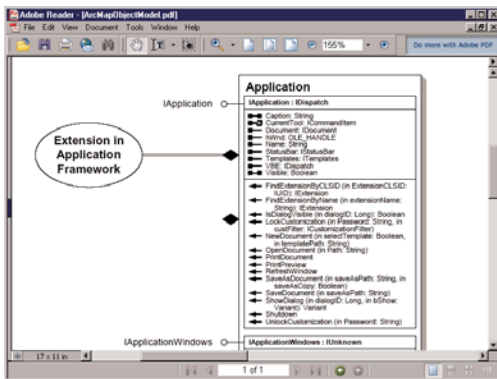
Subtask 2: Add the layer to ArcMap.

Keyword List: Application, MxDocument, Map, FeatureLayer, Add

1. Using Acrobat Reader, search the relevant object model diagram for each keyword. For Application, you should find the Application class in the ArcMap object model diagram.

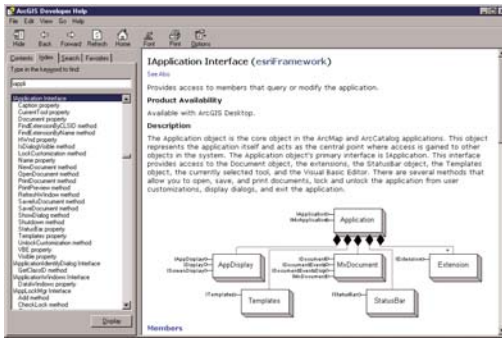
Inspect the interfaces that the class supports.

2. Once you find the class, go back to the ArcGIS Developer Help system and use the index to search for that class. For this



first keyword, click *Application* *coClass*, then the *esriArcMap* reference in the popup. Read the information available. The help documentation reveals that *Application* is the primary object for ArcMap and the other desktop applications that appeared in the popup. Click the Interfaces hyperlink and view the interfaces associated with the *Application* class. Click *IApplication* (*esriFramework*) to view the information available. Look for an example and write code to access the application.

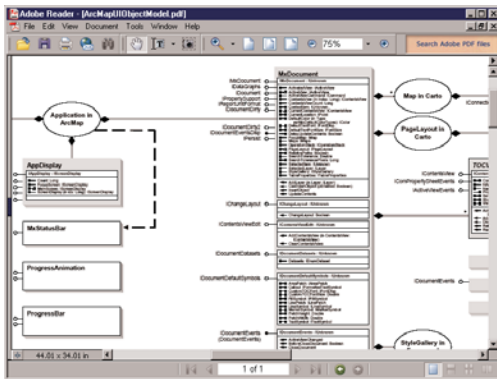
' Subtask 2. Add the new layer to the map.
 Dim pApp as IApplication
 Set pApp = Application



Now expand the members of *IApplication* with the Members hyperlink. This information reveals that it is possible to access the current document with the *IApplication::Document* member. The code could be updated as follows:

' Subtask 2. Add the new layer to the map.
 Dim pApp as IApplication
 Dim pDoc as IDocument
 Set pApp = Application
 Set pDoc = pApp.Document

3. Now open the ArcMapUI object model diagram and find the *MxDocument* class. Notice the wormhole link from this class to *Application* and *Map*. Inspect the interfaces associated with this class. Notice that *IDocument* does not provide a member to access the *Map* class, but the *IMxDocument* interface does. Navigate the diagram to find the *Map* class.

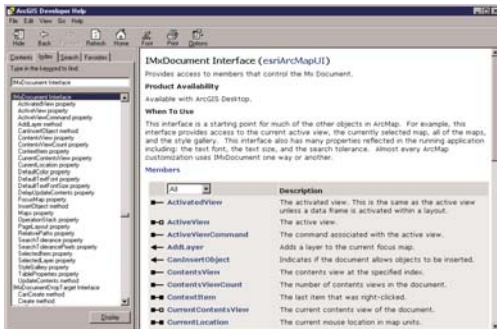


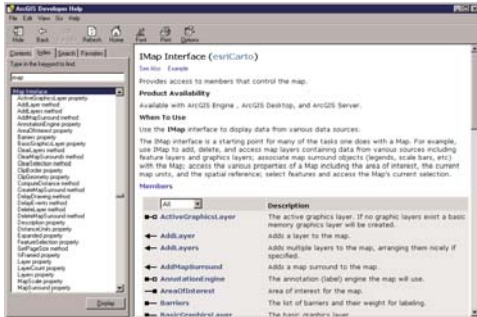
4. Go back to the ArcGIS Developer Help system and use the index to search for *MxDocument*. Read the information available. Click the Interfaces hyperlink. Click *IMxDocument* and expand the members. Notice that the *IMxDocument* interface supports the *FocusMap* member and returns a reference to *IMap*. Use this member to access the *Map* class.

Update the code to get a reference to the document's map.
 ' Subtask 2. Add the new layer to the map.

Dim pApp as IApplication
 Dim pDoc as IDocument
 Dim pMxDoc as IMxDocument
 Dim pMap as IMap

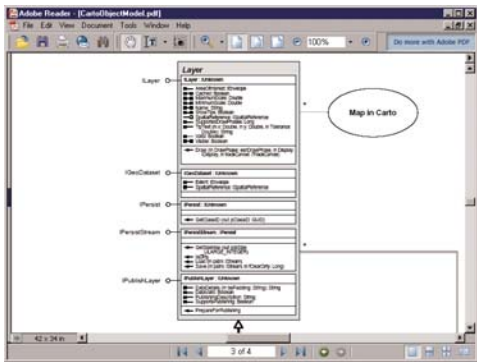
Set pApp = Application
 Set pDoc = pApp.Document





Set pMxDoc = pDoc ' QI
Set pMap = pMxDoc.FocusMap

5. Go back to the ArcGIS Developer Help system and use the index to search for the *Map* coclass. Click the Interfaces hyperlink and the *IMap* interface. Expand the members and locate the *AddLayer* member. This member will be used later to add a layer to the map, but first you need to create the new layer and associate it with the “States” data.



6. Locate the wormhole in the Carto object model diagram that connects the *Map* class (page 1 of the Carto OMD) to the map layer object model diagram (page 3 of the Carto OMD). Open the map layer object model diagram and browse the contents. Search for the *FeatureLayer* keyword until you find the class. Inspect the inheritance relationship between *FeatureLayer* and *Layer*. Also, identify the interface inheritance between *IFeatureLayer* and *ILayer*.

The interface inheritance information can also be acquired if you go back to the ArcGIS Developer Help system and use the index to search for the *FeatureLayer* coclass. Expand the Interfaces hyperlink and notice that it supports the *ILayer* interface.



7. Now, inspect the members of *IFeatureLayer* more closely by using the ArcObjects Developer Help system or the object model diagram. Notice it supports an *IFeatureLayer::FeatureClass* member property. From the documentation and the information in the *esriCarto* library overview, it should be obvious that you need to use this property to connect the *FeatureClass* class to the *FeatureLayer* class. The feature class contains a reference to the “States” dataset that was acquired in Subtask 1. Also, set the name of the layer to *IFeatureClass::AliasName*.

The last step is to add the new layer to the *Map*.

' Subtask 2. Add the new layer to the map.

```
Dim pApp as IApplication
Dim pDoc as IDocument
Dim pMxDoc as IMXDocument
Dim pMap as IMap
Dim pFL as IFeatureLayer
```

```
Set pApp = Application
Set pDoc = pApp.Document
Set pMxDoc = pDoc ' QI
```

```

Set pMap = pMxDoc.FocusMap
Set pFL = New FeatureLayer
Set pFL.FeatureClass = pFC ' pFC From Subtask 1.
pFL.Name = pFC.AliasName
pMap.AddLayer pFL

```

8. Now that you understand the relationship between the classes and interfaces, the code can be optimized. Rewrite the code as follows:

' Subtask 2. Add the new layer to the map.

```

Dim pApp as IApplication
Dim pDoc as IMxDocument
Dim pFL as IFeatureLayer
Set pApp = Application
Set pMxDoc = pApp.Document
Set pFL = New FeatureLayer
Set pFL.FeatureClass = pFC ' pFC From Subtask 1.
pFL.Name = pFC.AliasName
pMxDoc.FocusMap.AddLayer pFL

```

9. Now, assemble all the code from Subtasks 1 and 2. It will look like this:

' Subtask 1. Access the states feature class.

```

Dim pWSF as IWorkspaceFactory
Dim pFWS as IFeatureWorkspace
Dim pFC as IFeatureClass

Set pWSF = New AccessWorkspaceFactory
Set pFWS = pWSF.OpenFromFile("c:\data\US.mdb", 0)
Set pFC = pFWS.OpenFeatureClass("States")

```

' Subtask 2. Add the new layer to the map.

```

Dim pApp as IApplication
Dim pMxDoc as IMxDocument
Dim pFL as IFeatureLayer

Set pApp = Application
Set pMxDoc = pApp.Document
Set pFL = New FeatureLayer
Set pFL.FeatureClass = pFC ' pFC from Subtask 1.
pFL.Name = pFC.AliasName
pMxDoc.FocusMap.AddLayer pFL

```

SUMMARY

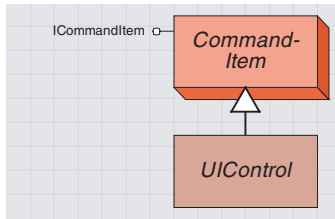
It should be clear now that there are several ways to solve ArcObjects programming problems. The similarities between all of them, however, are being able to use the help documents and resources effectively and being able to read the object model diagrams. Hopefully, this guide has provided you with an opportunity to visit the main resources that are available and exercise their use to solve this real-world problem.



E

UIControls

UIControls are VBA-based commands whose interfaces are only available in VBA. This appendix shows class and interface diagrams for these controls.



UIControls represent buttons, combo boxes, edit boxes, or tools in a custom dialog box.

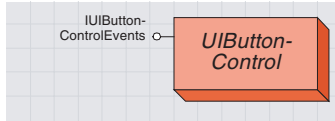


UIControls are VBA-based commands. This means that VBA code stored in a document or template defines and determines the behavior of this type of command. If a *UIControl* was created in a document, it can only be accessed in that document. If a *UIControl* was created in a template, it can be accessed in the template and any document that uses the template. If a *UIControl* was created in the Normal template, it can be accessed at all levels. There are four different types of *UIControl*: *UIButtonControl*, *UIComboBoxControl*, *UIEditBoxControl*, and *UIToolControl*.

To create a new *UIControl*, use the New *UIControl* button on the Customize dialog box in the ArcGIS applications; this creates a *UIControl* stub. While the Customize dialog box is still open, you can drag the new *UIControl* to any toolbar. You can then write the code that defines and determines the behavior of the *UIControl*. This code is written in the Visual Basic Editor in the *ThisDocument* Code window for the document or template in which you created the *UIControl*.

The new *UIControl* is listed in the Object box on the Code window; select the *UIControl* in this list. Then, click one of the functions listed in the Procedures/Events box on the Code window. This will stub out the function in the Code window. You can now write your code. When the Visual Basic Editor is open, your *UIButtonControl* is in design mode. To fully test your button in ArcMap or ArcCatalog, you need to close the Visual Basic Editor.

The interfaces for *UIControls* are usable only in Visual Basic for Applications.



A *UIButtonControl* acts as a button or menu item that performs a simple task when clicked. You can set properties such as status bar message, *ToolTip*, enabled state, and checked state.

A *UIButtonControl* acts as a button or menu item that performs a simple task when clicked.

UIButtonControlEvents :	UIButtonControl Events interface
← Checked: Boolean	Requests whether the specified item is checked.
← Click	The specified item was clicked.
← Enabled: Boolean	Requests whether the specified item is enabled.
← Message: String	Requests the current message text for the specified item.
← ToolTip: String	Requests the current <i>ToolTip</i> text for the specified item.

The *UIButtonControlEvents* interface defines the properties of a *UIButtonControl*, such as the enabled state, checked state, *ToolTip*, and status bar message. This interface also has a *Click* method that defines what action occurs when the button is clicked.

The following VBA code is a full implementation of a *UIButtonControl* that reports the number of selected features in all the layers. This control is enabled only when there are layers in the map.

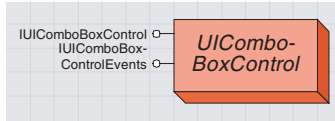
```
Private Function UIButtonControl1_Checked() As Boolean
    UIButtonControl1_Checked = False
End Function

Private Sub UIButtonControl1_Click()
    Dim pMxDoc As IMxDocument
    Dim SelCount As Long
    Set pMxDoc = Application.Document
    SelCount = pMxDoc.FocusMap.SelectionCount
    MsgBox SelCount
End Sub

Private Function UIButtonControl1_Enabled() As Boolean
    Dim pMxDoc As IMxDocument
    Dim LayerCount As Long
    Set pMxDoc = Application.Document
    LayerCount = pMxDoc.FocusMap.LayerCount
    If LayerCount > 0 Then
        UIButtonControl1_Enabled = True
    Else
        UIButtonControl1_Enabled = False
    End If
End Function

Private Function UIButtonControl1_Message() As String
    UIButtonControl1_Message = "Return selection count for all layers"
End Function

Private Function UIButtonControl1_ToolTip() As String
    UIButtonControl1_ToolTip = "Selection Count"
End Function
```



A *UIComboBoxControl* is a dropdown list box control that can be added to a toolbar.

A *UIComboBoxControl* has properties and methods that allow you to change, add, and remove items in the combo box list. The *EditChange*, *SelectionChange*, and *KeyDown* events allow you to control what happens when a user changes the text or selection in the combo box.

UIComboBoxControlEvents :	UIComboBoxControl Events interface
← EditChange	Occurs when the user types within the edit portion of the combobox.
← Enabled: Boolean	Requests whether the specified item is enabled.
← GotFocus	Occurs when UIComboBoxControl gets focus.
← KeyDown (in keyCode: Long, in shift: Long)	Occurs when the user presses a key.
← LostFocus	Occurs when UIComboBoxControl loses focus.
← Message: String	Requests the current message text for the specified item.
← SelectionChange (in newIndex: Long)	Occurs when the user selects an item in the combobox.
← ToolTip: String	Requests the current ToolTip text for the specified item.

The *UIComboBoxControlEvents* interface defines the properties of a *UIComboBoxControl*, such as the enabled state, ToolTip, and status bar message. This interface also has *EditChange*, *KeyDown*, and *SelectionChange* methods that allow you to control what happens when a user changes the text or selection in the combo box.

The following VBA code displays a message box that reports the currently selected item when the selection changes in the combo box.

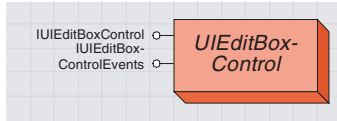
```
Private Sub UIComboBoxControl1_SelectionChange(ByVal newIndex As Long)
    MsgBox UIComboBoxControl1.Item(newIndex)
End Sub
```

UIComboBoxControl : IDispatch	UIComboBox Control interface
■ EditText: String	Returns or sets the edit text within the combobox.
■ Item (in index: Long) : String	Returns the text at the specified index.
■ ItemCount: Long	Returns the number of items currently inside of the combobox.
■ ListIndex: Long	Returns or sets the selected index within the combobox.
← AddItem (in itemText: String, index: Variant)	Adds an item to the combobox, optionally at the specified index.
← DeleteItem (in index: Long)	Deletes an item from the combobox at the specified index.
← RemoveAll	Removes all items from the combobox.

The *UIComboBoxControl* interface has properties and methods that allow you to change, edit, and remove items in the combo box list.

The following VBA macro adds items to *UIComboBoxControl1* and selects the first item in the list.

```
Public Sub PopulateComboBox()
    UIComboBoxControl1.AddItem "Red"
    UIComboBoxControl1.AddItem "Green"
    UIComboBoxControl1.AddItem "Blue"
    UIComboBoxControl1.AddItem "Yellow"
    UIComboBoxControl1.ListIndex = 0
End Sub
```

A *UIEditBoxControl* is an editable text box control that can be added to a toolbar.

A *UIEditBox* has a property to set the text that appears in the Edit box. The *Change* and *KeyDown* events allow you to control what happens when a user changes the text in the Edit box.

UIEditBoxControlEvents :	UIEditBoxControl Events interface
← Change	Occurs when the user types within the editbox.
← Enabled: Boolean	Requests whether the specified item is enabled.
← GotFocus	Occurs when UIEditBoxControl gets focus.
← KeyDown (in keyCode: Long, in shift: Long)	Occurs when the user presses a key.
← LostFocus	Occurs when UIEditBoxControl loses focus.
← Message: String	Requests the current message text for the specified item.
← ToolTip: String	Requests the current Tooltip text for the specified item.

The *UIEditBoxControlEvents* interface defines the properties of a *UIEditBoxControl*, such as the enabled state, ToolTip, and status bar message.

This interface also has *Change* and *KeyDown* methods that allow you to control what happens when a user changes the text in the Edit box.

The following VBA code uses the *KeyDown* method to report the current text in the Edit box if Return is pressed.

```

Private Sub UIEditBoxControl1_KeyDown(ByVal keyCode As Long, ByVal shift As Long)
    If keyCode = vbKeyReturn Then
        MsgBox UIEditBoxControl1.Text
    End If
End Sub
  
```

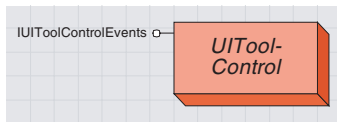
UIEditBoxControl : IDispatch	UIEditBoxControl interface
Text: String	Returns or sets the editbox text.
Clear	Clears the contents of the editbox.

The *UIEditBoxControl* interface has a *Text* property for getting and setting the text in the *UIEditBox* control and a *Clear* method for deleting the text.

The following VBA macro sets the text in a *UIEditBoxControl* called *UIEditBoxControl1*.

```

Public Sub SetText()
    UIEditBoxControl1.Text = "Hello"
End Sub
  
```



A UIToolControl interacts with the application's display.

A *UIToolControl* is similar to a COM command that implements the *ITool* interface. This type of control can interact with the application's display. You can set all the properties that *UIButtonControls* have and define what occurs on events, including mouse move, mouse button press and release, keyboard key press and release, double-click, and right-click.

UIToolControlEvents :	UIToolControl Events interface
← ContextMenu (in x: Long, in y: Long) : Boolean	Occurs when the user clicks the right mouse button.
← CursorID: Variant	Requests the cursor ID of the specified item.
← DblClick	Occurs when the user double clicks the mouse.
← Deactivate: Boolean	Occurs when the tool is deactivated.
← Enabled: Boolean	Requests whether the specified item is enabled.
← KeyDown (in keyCode: Long, in shift: Long)	Occurs when the user presses a key.
← KeyUp (in keyCode: Long, in shift: Long)	Occurs when the user releases a key.
← Message: String	Requests the current message text for the specified item.
← MouseDown (in button: Long, in shift: Long, in x: Long, in y: Long)	Occurs when the user presses a mouse button.
← MouseMove (in button: Long, in shift: Long, in x: Long, in y: Long)	Occurs when the user moves the mouse.
← MouseUp (in button: Long, in shift: Long, in x: Long, in y: Long)	Occurs when the user releases a mouse button.
← Refresh (in hDC: Long)	Occurs when the map is refreshed.
← Select	Occurs when the tool is selected.
← ToolTip: String	Requests the current Tooltip text for the specified item.

The *IUIToolControlEvents* interface defines the properties of a *UIToolControl*, such as the enabled state, cursor, ToolTip, and status bar message. This interface also has methods that allow you to control what happens on events, including mouse move, mouse button press and release, keyboard key press and release, double-click, and right-click.

The following VBA code displays the x,y coordinates of the left mouse button click in the ArcMap status bar message.

```
Private Sub UIToolControl1_MouseDown(ByVal button As Long, _
ByVal shift As Long, ByVal x As Long, ByVal y As Long)
' Check for left button press
If button = 1 Then
' Convert x and y to map units.
Dim pPoint As IPPoint
Dim pMxApp As IMxApplication
Set pMxApp = Application
Set pPoint = pMxApp.Display.DisplayTransformation.ToMapPoint(x, y)
' Set the statusbar message
Application.StatusBar.Message(0) = Str(pPoint.x) & ", " & Str(pPoint.y)
End If
End Sub
```



F

Bibliography

*This bibliography represents some of the books developers at ESRI
reference when developing ArcGIS applications.*

This bibliography is not intended as a complete resource, but it does contain many of the everyday references that ESRI developers use when developing Visual Basic, Visual C++, Visual Studio .NET code, and ArcObjects.

It is not necessary to buy all these books before programming in COM; rather, look at these books and others that are available, and perhaps buy the one most suitable to your development track. The books listed are from various companies; however, there are many other companies producing books for developers of COM components. You are encouraged to look at these other books, too.

ATL

Grimes, Richard. *ATL COM Programmer's Reference*. Chicago: Wrox Press Inc., 1998.

Grimes, Richard. *Professional ATL COM Programming*. Chicago: Wrox Press Inc., 1998.

Grimes, Richard, et al. *Beginning ATL 3 COM Programming*. Chicago: Wrox Press Inc., 1999.

King, Brad, and George Shepherd. *Inside ATL*. Redmond, WA: Microsoft Press, 1999.

Rector, Brent, Chris Sells, and Jim Springfield. *ATL Internals*. Reading, MA: Addison–Wesley, 1999.

C++

Lippman, Stanley. *C++ Primer: Second Edition*. Reading, MA: Addison–Wesley, 1991.

Lippman, Stanley. *Inside the C++ Object Model*. Reading, MA: Addison–Wesley, 1996.

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Reading, MA: Addison–Wesley, 1992.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison–Wesley, 1996.

Shepard, George, and David Kruglinski. *Inside Visual C++: Fifth Edition*. Redmond, WA: Microsoft Press, 1998.

Stroustrup, Bjarne. *The C++ Programming Language: Third Edition*. Reading, MA: Addison–Wesley, 1997.

COM

Box, Don. *Essential COM*. Reading, MA: Addison–Wesley, 1998.

Chappell, David. *Understanding ActiveX and OLE: A Guide for Developers and Managers*. Redmond, WA: Microsoft Press, 1996.

Effective COM: 50 Ways to Improve Your COM and MTS-Based Applications. Edited by Don Box, Keith Brown, Tim Ewald, and Chris Sells. Reading, MA: Addison–Wesley, 1998.

Major, Al. *COM IDL and Interface Design*. Chicago: Wrox Press Inc., 1999.

Platt, David S. *Understanding COM+*. Redmond, WA: Microsoft Press, 1999.

Rogerson, Dale. *Inside COM: Microsoft's Component Object Model*. Redmond, WA: Microsoft Press, 1997.

Software Engineering

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison–Wesley, 1995.

The New Hacker's Dictionary: Second Edition. Edited by Eric Raymond. Cambridge, MA: MIT Press, 1993.

VBA

Cummings, Steve. *VBA For Dummies*. New York: IDG Books Worldwide, 1999.

Getz, Ken, and Mike Gilbert. *VBA Developer's Handbook*. San Francisco: Sybex, 1997.

Lomax, Paul. *VB and VBA in a Nutsbell: The Language*. Sebastopol, CA: O'Reilly & Associates, 1998.

Visual Basic 6

Lewis, Thomas. *VB COM*. Chicago: Wrox Press Inc., 1999.

Microsoft Visual Basic 6.0 Programmer's Guide. Redmond, WA: Microsoft Press, 1998.

Pattison, Ted. *Programming Distributed Applications With COM and Microsoft Visual Basic 6.0*. Redmond, WA: Microsoft Press, 1998.

Wright, Peter. *Beginning Visual Basic 6 Objects*. Chicago: Wrox Press Inc., 1998.

Visual Studio .NET

Blair, Richard, et al. *Beginning VB .NET*. Birmingham, UK: Wrox Press Inc., 2002.

Box, Don, and Chris Sells. *Essential .NET, The Common Language Runtime. Volume 1*. Boston: Addison–Wesley, 2002.

Chappell, David. *Understanding .NET, A Tutorial and Analysis*. Boston: Addison–Wesley, 2002.

Nathan, A. *.NET and COM: The Complete Interoperability Guide*. Indianapolis, IN: Sams Publishing, 2002.

Reynolds–Haertle, Robin A. *OOP With Microsoft Visual Basic .NET and Microsoft Visual C# Step by Step*. Redmond, WA: Microsoft Press, 2002.

Templeman, Julian, and John Paul Mueller. *COM Programming With Microsoft .NET*. Redmond, WA: Microsoft Press, 2003.

Index

A

- Abstract class 135
- Active Template Library. See ATL
- ActiveX DLL 8, 172–173
- AddrOf method. See IUnknown interface
- Aggregation. See COM: aggregation
- AOInitialize coclass 75
- Apartment 141–142
- API 12
- Application object 171, 174
- ArcCatalog
 - customizing 52
 - described 7
- ArcGIS
 - development
 - possibilities 3
- ArcGIS 9 Developer overview 2
- ArcGIS architecture
 - ArcObjects libraries 20
 - developed 19
 - extendable 19
 - key concepts 19
 - modular
 - benefits 19
 - requirements 19
 - compatibility 19
 - modularity 19
 - multiple platform support 19
 - scalability 19
- ArcGIS Desktop
 - common application framework 46
 - customization options 47
 - custom objects 48
 - extensions 48
 - framework components 48
 - user interface customization 47
 - VBA macros 47
 - VBA UI controls 47
 - deployments
 - ArcEditor 74
 - ArcInfo 74
 - ArcView 74
 - object models 74
 - using one code base 74
 - graphical user interface 46
 - supported APIs
 - COM 23
 - NET 23
- ArcGIS Desktop application libraries 33
 - 3DAnalystUI 40
 - ArcCatalog 35
 - ArcCatalogUI 35
 - ArcGlobe 40
 - ArcMap 36
 - ArcMapUI 36
 - ArcReaderControl 41
 - ArcScan 41
 - ArcScene 40
 - CartoUI 35
 - Catalog 34
 - CatalogUI 35
 - DataSourcesRasterUI 35
 - DisplayUI 34
 - Editor 36
 - EditorExt 38
 - Framework 34
 - GeoDatabaseDistributedUI 38
 - GeoDatabaseUI 34
 - Geoprocessing 38
 - GeoprocessingUI 38
 - GeoStatisticalAnalyst 41
 - GeoStatisticalAnalystUI 41
 - GlobeCoreUI 40
 - LocationUI 36
 - Maplex 42
 - OutputExtensions 38
 - OutputExtensionsUI 40
 - OutputUI 34
 - Publisher 41
 - PublisherUI 42
 - SpatialAnalystUI 40
 - SurveyDataEx 43
 - SurveyExt 42
 - SurveyPkgs 42
 - TrackingAnalyst 42
 - TrackingAnalystUI 42
- ArcGIS Desktop applications 4
 - ArcCatalog 6
 - ArcMap 6
 - ArcToolbox 6
- ArcGIS Desktop components
 - building 68
 - basic steps 68
- ArcGIS Desktop Developer Kit
 - described 4
- ArcGIS Desktop extensions 6
- ArcGIS Desktop license checking 74
- ArcGIS Desktop software products 6
 - ArcEditor 6
 - ArcInfo 6
 - ArcView 6
- ArcGIS Developer resources 12
 - add-ins 14

- ArcGIS Developer Documentation Series 14
- ArcGIS Developer Online Web site 15
- Developer Tools 13
- ESRI Support Center 16
- Samples 12
- Training 16
- Visual Basic 6 14
 - ESRI Align Controls with Tab Index 14
 - ESRI Automatic References 14
 - ESRI Code Converter 14
 - ESRI Command Creation Wizard 14
 - ESRI Compile and Register 14
 - ESRI ErrorHandler Generator 14
 - ESRI ErrorHandler Remover 14
 - ESRI Interface Implementer 14
- Visual Studio .NET 14
 - ESRI .NET Code Converter 14
 - ESRI Component Category Registrar 14
 - ESRI GUID Generator 14
- ArcGIS developer tools
 - Component Categories Manager 13
 - ESRI Object Browser 13
 - install directory 13
 - exceptions 13
 - Library Locator 13
- ArcGIS Engine
 - developer components
 - shown 18
- ArcGIS Engine Developer Guide
 - described 14
- ArcGIS Engine Developer Kit
 - described 4
- ArcGIS Engine libraries 25, 26–27
 - 3DAnalyst 31
 - Carto 28
 - Controls 31
 - DataSourcesFile 28
 - DataSourcesGDB 28
 - DataSourcesOleDB 28
 - DataSourcesRaster 28
 - Display 27
 - GeoAnalyst 31
 - GeoDatabase 27
 - GeoDatabaseDistributed 28
 - Geometry 26
 - GISClient 27
 - GlobeCore 31
 - Location 30
 - NetworkAnalysis 30
 - Output 27
 - Server 27
 - SpatialAnalyst 32
 - System 26
 - SystemUI 26
- ArcGIS objects
 - library
 - subdividing 20
- ArcGIS Server Developer Guide
 - described 15
- ArcGIS Software Developer Kit. See SDK
- ArcGlobe
 - customizing 51
 - document extension (.3dd) 52
 - Normal.3dt 52
- ArcMap
 - customizing 50
 - global customization with project templates 51
 - map documents 50
 - map templates 50
 - selective customization with project templates 50
 - described 6
 - document extension (.mxd) 50
 - starting programmatically 175
 - template 166, 170
- ArcObjects 3
 - accessing APIs 23
 - apartment threading model 20
 - categories 18
 - coarse-grained objects 3
 - consuming the API 23
 - described 18
 - developing with 4
 - end use of objects 18
 - ArcGIS Desktop 18
 - ArcGIS Engine 18
 - ArcGIS Server 18
 - extending the API 23
 - COM 23
 - NET 24
 - fine-grained objects 3
 - illustrated code samples 263
 - add feature class to ArcMap 268
 - add layer to ArcMap using GxDialog 270
 - add map surround to page layout 278
 - add text callout to active view 280
 - display map extent in GxView as envelope 290
 - display raster cell value in status bar 284
 - draw digitized line onscreen 266
 - edit feature class schema 292
 - export current view 286
 - geometry projection 282
 - locate and execute command on toolbar 265
 - loop through selected area features 274
 - print current view 288
 - spatial query 276
 - style gallery auto symbol selection 272
 - license checking
 - application 75
 - extension 75

- functional 77, 78
 - types of 74
 - problem-solving guide 295
 - Part 1: Define the ArcObjects Programming Task 297
 - Part 2: Find the correct object model diagram 303
 - Part 3: Navigate the object model diagram 305
 - requirements 18
 - Threads in Isolation model 21
 - ArcObjects library
 - C++
 - header file 20
 - COM
 - type library 20
 - components
 - requirements 20
 - Java
 - Java package 20
 - NET
 - .NET Interop Assembly 20
 - ArcScene
 - customizing 51
 - document extension (.sxd) 52
 - Normal.sxt 52
 - ArcToolbox 6
 - ATL 178
- B**
- Bibliography 319
 - Binding 139
 - BSTR 151
- C**
- C++. See Visual C++
 - Callback mechanism 138
 - Class factory. See COM: class factory
 - Classes 134
 - Coclass 135
 - Coding standards 144. See *also* Visual Basic: coding guidelines; Visual C++: coding guidelines
 - COM
 - Active Template Library. See ATL
 - aggregation 140–141
 - background 132–133
 - class factory 134
 - client 133
 - client storage 146
 - containment 140–141
 - described 132–143
 - Direct-To-COM (DTC). See *also* Direct-To-COM
 - DLL 133
 - EXE 133
 - instantiating objects 142
 - instantiation of features 150
 - interface. See *also* Interface
 - described 135
 - marshalling 141
 - review 64
 - server 133
 - COM API 23
 - limitations
 - Visual Basic language 23
 - supported platform 24
 - Com DLLs
 - packing and deploying
 - methods 89
 - Commands with VBA
 - custom
 - creating 8
 - Compatibility 19, 21
 - maintaining between releases 21
 - Component category 65, 142, 151–152, 173
 - Component Category Manager 152
 - Component development 64
 - languages
 - C++ Builder 67
 - Delphi 67
 - Visual Basic 6 67
 - Visual C++ 67
 - Visual Studio 6 67
 - Visual Studio NET 67
 - performance differences 67
 - Component development environment
 - choosing 67
 - factors 67
 - Component Object Model. See COM
 - Components
 - application level 9
 - custom 9
 - advantages of building 9
 - writing 8
 - domain-specific 6
 - functionality
 - described 66
 - geodatabase level 9
 - plugging into ArcGIS Desktop 65
 - Containment. See COM: containment
 - CORBA 132
 - Cross-thread communication
 - reducing 21
 - Cursor
 - insert 149–150
 - recycling 149
 - update 149–150
 - Custom feature 140
 - Custom objects
 - creating 8
 - Customizations
 - storing

- documents and templates 49
- Customizing ArcGIS Desktop applications
 - introduced 7

D

- Data types 150–151
- DCE 134, 137
- Debugging. See Visual Basic: debugging; Visual C++:
 - debugging
- Desktop GIS 2
- Developer help
 - Start menu options
 - shown 12
- Direct-To-COM 180
- Dispatch event
 - interface 138
- DLL 136, 143, 172–173
 - customizations
 - packing and deploying 86
 - developments
 - packing and deploying 87
- DTC. See Direct-To-COM
- Dynamic Link Library (DLL). See DLL

E

- Edit operations 149
- Editing
 - rules for geodatabase integrity 148–150
- Editor coclass 148
- EditSelection coclass 161
- Embedded GI 2
- Enumerator interfaces 146, 165
- Envelope coclass 146
- Error handling 146, 155, 160
- ESRI Object Browser
 - displayed 13
- Event handling 147–148, 162
- Exception handling. See Error handling

F

- Feature
 - COM instantiation of 150
 - editing shape of 150
- Feature CoClass 150
- FeatureLayer CoClass 164, 166

G

- Geodatabase
 - editing rules 148–150
- Globally unique identifier. See GUID
- GUID 65, 134, 142

- GxDocument CoClass 167

H

- HRESULT 80, 81, 146, 160
 - license-related 83

I

- IActiveViewEvents interface 173
- IAOInitialize interface 75
- IApplication interface 138, 174, 175
- ICommand interface 65, 177
- Identifies interface 146
- IDispatch interface 139–140, 147
- IDL 137, 150–152
- IDocument interface 167, 174, 175
- IDocumentEvents interface 138
- IDocumentEventsDisp interface 138
- IEnumFeature interface 165
- IEnumGxObject interface 161
- IExtension interface 151, 173
- IMap interface 164
- IMxDocument interface 164
- Inheritance
 - interface 140
 - type. See Type inheritance
- Interface
 - and Visual Basic 156–159
 - default 138, 157
 - deprecated 136
 - described 134–136
 - notification interface 146
 - optional 136
 - outbound 138, 147–148, 162, 173
- Interface Definition Language (IDL). See IDL
- IPoint interface 159, 160, 163, 164
- IPolygon interface 146
- IRootLevelMenu interface 146
- Is keyword 148, 164
- IShortcutMenu interface 146
- IUnknown interface 136–137, 138, 156–159
- IWorkspaceEdit interface 148, 149

J

- Java 132
- JavaScript 139

M

- Main STA 21
- Map template
 - described 50
- Marshalling. See COM: marshalling

Method calls
 license-related HRESULTs
 returned 84
Microsoft Component Object Model (COM)
 8, 18, 132. *See also* COM
Microsoft Interface Definition Language. *See* IDL
Mobile GIS 2
Modularity 19
Multithreaded applications
 considerations 20
 scalability 20
 thread safety 20
MxDocument CoClass 167

N

NET 46
NET API
 exceptions 24
 supported platform 24
NET Framework DLLs
 packing and deploying
 methods 87
Normal template 51
 Normal.3dt 52
 Normal.gxt 52
 Normal.mxt 51
 Normal.sxt 52
 Windows 2000 and XP 51
Notification interface 146

O

Object browser utility 145
Object Definition Language. *See* IDL
Object library. *See* Type library
Object model diagrams
 classes and objects
 abstract class 261
 class 261
 coclass 261
 described 261
 composition
 described 262
 instantiation
 described 262
 interpreting 260
 relationships
 abstract class 261
 multiplicity 261
 types of 261
 type inheritance
 described 262
OLE automation 24
Open Group's Distributed Computing Environment. *See*

DCE

Outbound interface. *See* Interface: outbound

P

Platform support
 multiple 19
 described 21
Point CoClass 159, 160, 163
Polymorphism 135
ProgID 172, 183
Programmable identifier. *See* Prog ID
Property by reference 147, 159, 161
Property by value 147, 161

Q

QI. *See* Query interface
Query
 performance 149
Query interface 136–137

R

Regedit 152
Registry 142–143, 152
 script 152
Release method. *See* IUnknown interface

S

Scalability 19, 20
 achieving 20
 ArcEngine
 ArcObjects components 20
 ArcServer
 ArcObjects components 20
 memory within the objects 20
 multithreaded applications 20
SCM 141, 142
SDK 225, 226, 239, 240, 241, 244, 254
Server GIS 2
Set 161
Singleton objects 134, 175
SRI Library Locator
 displayed 13

T

TabIndex property 153
Table 139, 173
ThisDocument object 167, 171, 174
Thread 141
Type inheritance 134
Type library 137, 150, 159, 174

VB reference 174
TypeOf keyword 164

U

UML diagrams
 types of classes shown 261
Unicode 197, 199
Unified Modeling Language (UML) 260
Universally Unique Identifier (UUID). See GUID
Using this guide 10
 chapter guide 10

V

VB. See Visual Basic
VBA. See Visual Basic; Visual Basic for Applications
VBA customizations 53
 adding a menu to a toolbar
 tutorial 55
 adding buttons to a toolbar
 tutorial 54
 creating a new toolbar
 tutorial 53
 removing buttons from a toolbar
 tutorial 54
 renaming a toolbar
 tutorial 54
 saving changes to a template
 tutorial 55
 showing and hiding toolbars
 tutorial 53
 tutorials 53
 user interface
 tutorial 53
VBA development environment 46
VBA developments
 packing and deploying
 methods 86
VBA environment
 reasons to choose 8
VBA macros 55
 adding a macro to a toolbar
 tutorial 58
 adding code for the UIButtonControl
 tutorial 60
 adding code for the UIToolControl
 tutorial 61
 ArcID module 56
 calling built-in commands
 tutorial 59
 changing button properties 63
 creating a command in VBA
 tutorial 60
 creating a macro
 tutorial 57
 creating a tool in VBA
 tutorial 61
 getting help in the Code window
 tutorial 59
 invoking the Visual Basic Editor directly
 tutorial 58
 preset ArcObjects variables 55
 Application variable 56
 ThisDocument 56
 VBA project organization 56
VBA macros in ArcGIS applications
 writing 8
VBScript 139
VBVM. See Visual Basic:Virtual Machine (VBVM)
Visual Basic
 arrays 154
 coding guidelines 153–165
 coding standards
 ambiguous type matching 155
 arrays 154
 bitwise operators 155
 default properties 154
 indentation 154
 intermodule referencing 154
 multiple property operations 154
 order of conditional determination 154
 parentheses 153
 type suffixes 155
 variable declaration 153
 While Wend constructs 156
 collection object 165
 collections 165
 creating COM components 172
 data types 151
 debugging 176–178
 with ATL helper object 178
 with Visual C++ 177
 error handling 155
 event handling 162
 getting handle to application 174–176
 implementing interfaces 173
 interfaces 156–159
 Is keyword 164
 Magic example 158
 memory management 155
 methods 161
 parameters 162
 passing data between modules 163
 PictureBox 155
 starting ArcMap 175
 TypeOf keyword 164
 variables
 Option Explicit 153
 Private 153

- Public 153
- Virtual Machine (VBVM) 156, 159, 160
- Visual Basic 6 46
 - add-ins
 - displayed 14
 - limitations 67
- Visual Basic for Applications 138
 - and ArcGIS 166–171
 - customizations
 - packing and deploying 86
 - developing
 - disadvantages 64
 - getting started 167
 - locking code 170
 - UIControls
 - described 313
 - UIButtonControl class 315
 - UIComboBoxControl class 316
 - UIControl classes 314
 - UIEditBoxControl class 317
 - UIToolControl class 318
- Visual C++
 - Active Template Library. See ATL
 - coding guidelines 195–220
 - coding standards
 - argument names 196
 - function names 195
 - type names 195
 - data types 151
 - debugging 196–220
 - naming conventions 195

W

- Windows NT
 - profiles location 51
- Workspace CoClass 148

