UNIVERSITY OF CINCINNATI

DATE: 05-28-02

I, SRINIVAS RAYAPROLU ,
hereby submit this as part of the requirements for the degree of:

MASTERS

in:

COMPUTER SCIENCE

It is entitled:

USING COM OBJECT PROGRAMMING FOR ENHANCED LIBRARY SEARCH APPLICATIONS

Approved by:

FRED S ANNEXTSEIN

KENNETH BERMAN

CHAI-YUNG HAN

USING COM OBJECT PROGRAMMING FOR ENHANCED LIBRARY SEARCH APPLICATIONS

A thesis submitted to the

Division of Graduate Studies and Research of

University of Cincinnati

in partial fulfillment of the

requirements for the degree of

MASTERS OF SCIENCE

in the Department of

Electrical & Computer Engineering and Computer Science

of the College of Engineering

May, 2001

by

Srinivas Rayaprolu

B.E. in the Department of Computer Science
Osmania Univeristy, India, 1999

Thesis Advisors and Committee Chair Dr. Fred Annexstein and Dr. Kenneth Berman

ABSTRACT

The number and diversity of information sources on the Internet is increasing rapidly. With such an embarrassment of riches, a person who wishes to use the Internet as an information resource is going to need some assistance. Current search tools are inadequate in the sense that they cannot use multiple information sources in concert nor can they index the hundreds of billions of highly valuable documents “hidden” in proprietary databases.

It is the goal of this thesis to provide a set of integrated tools based on individual users requirements, and we focus on an application specific search of the resources at the University of Cincinnati library. We created a Software Tool for the Windows operating systems platform called “UC Library ToolBar” which provides a simple interface for searching not only the information on the web indexed by the search engines but also the information available in the databases to the Faculty/Student of the University of Cincinnati. For purpose of integration with existing browser software we choose the popular Microsoft IE platform. This led us to research and develop on Microsoft Component Object Model (COM) using ActiveX Template Library.

ACKNOWLEDGEMENTS

I would like to thank Dr. Fred Annexstein and Dr. Kenneth Berman, for their guidance and encouragement during the duration of this thesis and the entire Masters program.

I would like to express my love and gratitude to my dad, Late R. Surya Prakasa Rao who was a source of inspiration and the main motivation behind my doing my masters.

I would like to also thank my other family members for their constant support and understanding.

I would like to thank my friends who helped me in discussing my research and the ideas related to my thesis.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1: Goals of Thesis

The goal of this thesis is to design and implement an integrated search tool for the World Wide Web. The motivation for this comes from a number of problems searching the web using current existing tools. With the number and diversity of information sources on the Internet increasing rapidly, a person who wishes to use the Internet as an information resource is going to need some assistance. Currently there exist a number of standard search tools, such as Google, Lycos, Alta Vista, and Yahoo, which help people find information. However, the information covered by these Search Engines does not actually represent the whole gamut of information sources available on the web. Also, these Search Engines and the Directories are unable to interpret the results of their searches or use multiple information sources in concert [10]. The Library Tool we designed encapsulates all the existing search tools (search engines) and also allows access to the different databases available through the University of Cincinnati.

1.2: Background On the Web

As the web billows in size, search sites cover less of it, and what they cover is more likely to be popular commercial sites. According to a study, where once, as much as 40% of the web was indexed, by 1999 no one engine indexed more than 16%. If search technologies were to stand still, the phenomenal growth of the web would render them useless. There are already more than a billion pages and even the widest reaching search

engine covers barely half of these [11]. Within two years, the web may grow to 13 billion pages, and search engines face huge difficulties keeping pace.

A new study of the structure of the web provides little comfort. This contradicts earlier suggestions that any two pages on the web are connected by a relatively small number of hyperlinks [7]. The implication is that search engines must crawl from a greater diversity of starting points if they are to have any hope of giving a reasonable breadth of coverage. It's beginning to appear that centralized approaches to creating web indexes may not scale with the web's explosive growth. Catching up will likely require adopting some sort of distributed search approach.

One of the most worrisome developments on the Web is the inadequacy of the existing search tools to work in an era when Web sites increasingly depend on database queries and dynamically generated temporary URLs. Many sites have their own sophisticated searches, but one must have to visit the site and enter the search string manually. Data is generated dynamically for each query. There is no way for a search engine to find information during a web crawl, because no URL exists until the user queries the database. As a result the users never find many sites that have the information they want.

BrightPlanet has uncovered the "deep" Web – a vast reservoir of Internet content that is 500 times larger than the known "surface" World Wide Web [2]. What makes the discovery of the deep Web so significant is the quality of content found within. There are literally hundreds of billions of highly valuable documents hidden in searchable

databases that can't be retrieved by conventional search engines. Searching on the Internet today can be compared to dragging a net across the surface of the ocean. There is a wealth of information that is deep and therefore missed. The reason is simple: basic search methodology and technology have not evolved significantly since the inception of the Internet.

The deep Web is qualitatively different from the surface Web [1]. Deep Web sources store their content in searchable databases that only produce results dynamically in response to a direct request. Since the search engines create their indexes by crawling the web using static URLs, they cannot see or retrieve the content available in the deep web. Considering the amount of information available in the deep web (7,500 terabytes of information, compared to 19 terabytes of information in the surface Web) and the high relevancy of the information to every need, market and domain, any approach which does not try to include the information available in these databases does not represent an effective information and search tool.

1.3: Major Problems

With such a massive amount of information not being retrieved by the search engines, a searching strategy, which can access the information in the deep web, is necessary. There are problems in designing a search engine, which encapsulates the information available in the web sites as well as the web pages generated dynamically from the databases. First, every database uses its own index, its own searching strategy and its own criteria in ranking the results [3]. So to combine the results obtained from these databases is

practically impossible [9]. Second, if we could have the index available we could use our own search strategy to generate results. But due to the commercial nature of these databases they are not available.

A digital library provides one of the most important information environments in which to retrieve and refer to appropriate information directly online. Millions of people regularly access the Internet. However, this access is still more or less standardized in that almost every one uses the same means of information retrieval. It is not the case that one standardized way of information retrieval fits all needs. Different library users will have different personal requirements and interests in the use of library materials [8].

Companies like BrightPlanet, Intelliseek, Vivisimo, Inktomi, etc are front-runners in developing integrated search tools for searching the web and also customize the tools to different customers using them. These companies use innovative search technology in trying to design and develop tools suited to the different requirements of different users and companies.

1.4: Focus of Application

The focus of this application is the information content available to the Student/Faculty of the University of Cincinnati. Apart from the information available through the search engines, a Student/Faculty has access to a lot of information through different databases through the University Library. The experience of using one of these databases for research is a hard one. Firstly, The organization of these databases buries them under a myriad set of links, which makes it difficult to locate them. And after one has located the

database, one is confronted with a different interface to access the information available in the different databases. So the same query has to be entered in each of these forms to access the corresponding information associated with the databases. And this process has to be repeated every time one wants to access a different database. The problem only gets accentuated as the number of databases that the user wants to use regularly increases.

1.5: Toolbars

Toolbars provide one with an intuitive and easy to use interface. It provides one with a feature to group together user selected options and in the case of the library toolbar different databases and different search engines of the users choice. Once the toolbar has been installed it appears along with the Internet explorer toolbar. This allows one to search quickly from any website location, through different search engines and the different databases without having to visit any of the individual websites for the corresponding databases.

The toolbar provides a simple interface for searching the information on the web indexed by the search engines and the information available in the databases. It provides a simple interface to search through different search engines (Google, AltaVista, Yahoo, MetaCrawler, AllTheWeb, DirectHit), Databases (ACM/IEEE), Journals available through the Electronic Journal Center, the Library databases (UCLID and OHIOLINK). There would be no need to load and enter queries in the many separate web-forms for each of the different databases. A single query entered into a dialog toolbar (embedded as

a plug-in to the browser) could search these many databases according to the database selected by a single mouse click.

1.6: Organization

Chapter 2 provides a detailed description of the Internet Explorer's interface. Chapter 3 gives an introduction to COM and ATL. Chapter 4 explains the design issues of the toolbar. Chapter 5 details the conclusions and future directions.

CHAPTER 2

INTERNET EXPLORER PROGRAMMABLE INTERFACE

2.1: BandObjects

Internet explorer consists of three main categories of band objects: Explorer bars (Info and command bands), tool bands and desk bands. Info bands are vertical explorer bars and Command bands are horizontal explorer bars. There are two types of explorer bars. Vertical explorer bars are sometimes called Info bands and horizontal explorer bars, called command bands. Implementation of both is same. The Internet Explorer History, Favorites, and Search windows are all info bands [13].

Band objects are essentially COM objects that exist within a container. The container of a band object depends on the type to which the band object belongs. If the band object is a tool band, the Rebar Control that holds Internet Explorer's toolbars will contain it; if it is an Explorer Bar, it will be contained by Internet Explorer. Despite the difference between their functionalities, their basic implementations are similar. What really makes the difference is how the band object is registered, which in turn controls the type of the object and its container.

The Explorer Bar was introduced with Microsoft® Internet Explorer 4.0 to provide a display area adjacent to the browser pane. It is basically a child window within the Internet Explorer window, and it can be used to display information and interact with the

user in much the same way. Explorer Bars are most commonly displayed as a vertical pane on the left-hand side of the browser pane. However, an Explorer Bar can also be displayed horizontally, below the browser pane.

Tool Bands

IE5 introduced tool bands, which provide a convenient mechanism for adding bands to the browser's rebar. The bands can contain toolbars, or combo boxes, or radio controls, or whatever. This feature provides a way to put a window on a band contained by the Rebar Control that holds Internet Explorer's toolbars.

Fig 2.1: ToolBand

This toolbar essentially consists of a rebar control with four bands: three toolbars and a menu bar. Because it is implemented with the common controls application-programming interface (API), developers can create toolbars with any or all of its features.

The Rebar Control

The underlying structure of the Internet Explorer toolbar is provided by a rebar control. This control provides a way for users to customize the arrangement of a collection of tools. An application assigns child windows, which are often other controls, to a rebar control band. Rebar controls contain one or more bands, and each band can have any combination of a gripper bar, a bitmap, a text label, and a child window. However, bands cannot contain more than one child window. A rebar control displays the child window over a specified background bitmap. As one dynamically repositions a rebar control band, the rebar control manages the size and position of the child window assigned to that band.

The rebar control displays its bands in a rectangular area, typically at the top of the window. This rectangle is subdivided into one or more strips that are the height of a band. Each band can be on a separate strip, or multiple bands can be placed on the same strip. A rebar control provides users with two ways to arrange their tools. Each band usually has a *gripper* at its left-hand edge. Grippers are used when two or more bands on a single strip exceed the width of the window. By dragging the gripper to the left or right, users can control how much space is allocated to each band. Users can move the bands within the rebar's display rectangle by dragging and dropping. The rebar control then changes the display to accommodate the new arrangement of bands. If all the bands are removed from a strip, the height of the rebar will be reduced, enlarging the viewing area.

An application can add or remove bands as needed. Typically, applications enable users to select, which bands they want to have displayed through the View menu or a shortcut

menu. If the combined width of the bands on a strip exceeds the width of the window, the rebar control will adjust their widths as needed. Some of the tools might be covered by the adjacent band. Version 5.80 of the common controls provides a way to make tools that have been covered by another band accessible to the user. If one sets the RBBS_USECHEVRON flag in the **fStyle** member of the band's REBARBANDINFO structure, a *chevron* will be displayed for toolbars that have been covered. When a user clicks the chevron, a menu is displayed that allows him or her to use the hidden tools. Since each band contains a control, one can provide additional flexibility through the control's API.

Desk Bands

Band objects can also be used to create desk bands. While their basic implementation is similar to Explorer Bars, desk bands are unrelated to Internet Explorer. A desk band is basically a way to create a dockable window on the desktop. The user selects it by right-clicking the taskbar and selecting it from the Toolbars submenu.

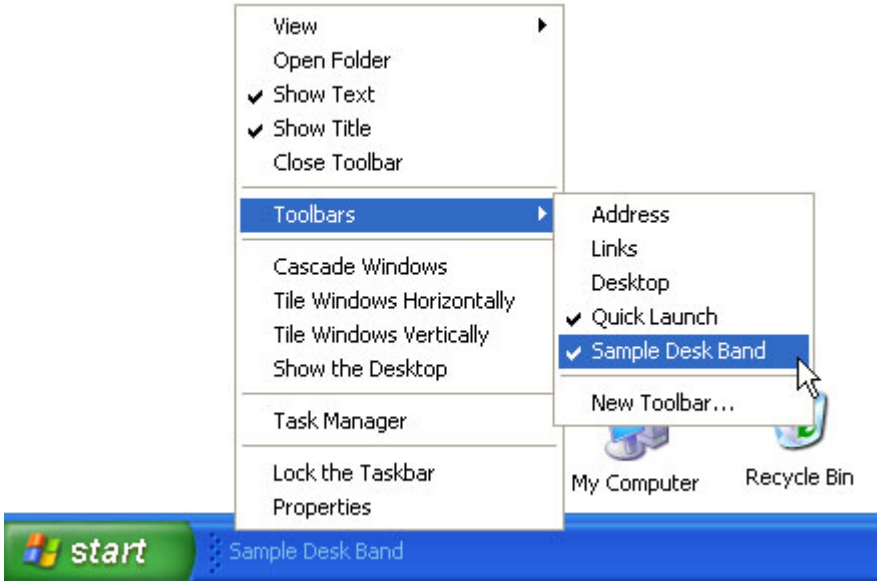

Fig 2.2: Sample DeskBand

Initially, desk bands are docked on the taskbar.

Fig 2.3: Sample Desktop Band

The user can then drag the desk band to the desktop, and it will appear as a normal window.

Fig 2.4: Sample DeskBand

2.2: Implementing Band Objects

Although they can be used much like normal windows, band objects are COM objects that exist within a container. Explorer Bars are contained by Internet Explorer, and desk bands are contained by the shell. The shell in terms of Windows is a graphical user interface provided by Windows that allows you to access the various components of the operating system. While they serve different functions, their basic implementation is very similar. The primary difference is in how the band object is registered, which in turn controls the type of object and its container. In addition to IUnknown and **IClassFactory**, all band objects must implement the following interfaces:

- IDeskBand
- IObjectWithSite
- IPersistStream
- IInputObject (optional)
- IContextMenu (optional)

- ICommandTarget (optional)

IDeskBand

IDeskBand is used to obtain information about a band object. This interface is used by the browser to obtain the display information for a band. This interface is derived from **IDockingWindow.** IDockingWindow interface provides docking feature to band objects so they can be docked inside a Windows Explorer window. This interface has three methods. These methods are

Methods	**Description**
CloseDW	Notifies the docking window object that it is about to be removed
ResizeBorderDW	Notifies the docking window object that the frame's border space has changed
ShowDW	Instructs the docking window object to show or hide itself.

IObjectWithSite

This interface provides communication between a band object and its container, a browser. An object implements this interface so its container can supply it with an interface pointer for its site object. Then the object can communicate directly with the site. It has two methods:

Methods	**Description**
SetSite	Provides the site's **IUnknown** pointer to the object being managed.
GetSite	Retrieves the last site set with **IObjectWithSite::SetSite**.

IPersistStream

Applications handle data in many ways, including displaying, moving, loading, and saving it. As a distributed object model system, COM has its own way of handling these standard-programming tasks. COM handles object persistence—the ability to move an object's state between the object in memory and a piece of persistent media such as a disk file through this interface.

IInputObject

The browser uses this interface. The IInputObject interface is used to notify the object of UI activation change and translate keyboard accelerators. If a band object accepts user input then this interface must be implemented. Internet Explorer implements **IInputObjectSite** and uses **IInputObject** to maintain proper user input focus when it has more than one contained window. There are three methods that need to be implemented by an Explorer Bar: **IInputObject::UIActivateIO**, **IInputObject::HasFocusIO**, and **IInputObject::TranslateAcceleratorIO**. Internet Explorer calls **IInputObject::UIActivateIO** to inform the Explorer Bar that it is being activated or deactivated. When activated, the Explorer Bar sample calls **IOleInPlaceSiteWindowless::SetFocus** to set the focus to its window. Internet Explorer calls **IInputObject::HasFocusIO** when it is attempting to determine which window has focus. If the Explorer Bar's window or one of its descendants has focus, **IInputObject::HasFocusIO** should return S_OK. If not, it should return S_FALSE. **TranslateAcceleratorIO** allows the object to process keyboard accelerators.

IContextMenu

This interface is used to create a context menu for a band object. This interface must be implemented if one wants to create context menus. Otherwise there is no need to implement this interface. This will create a menu on right mouse click of the explorer bar. IContextMenu has three functions.

Methods	**Description**
GetCommandString	Retrieves a command's text.
QueryContextMenu	Adds commands to a context menu.
InvokeCommand	Carries out the command associated with a context menu item.

Band objects provide a flexible and powerful way to extend the capabilities of Internet Explorer by creating custom Explorer Bars. Implementing a desk band allows you to extend the capabilities of normal windows. Although some COM programming is required, it ultimately serves to provide you with a child window for your user interface. From there, the bulk of the implementation can use familiar Windows programming techniques. All the above necessary features of a band object can be readily extended to create a unique and powerful user interface.

2.3: Windows registry and Internet Explorer

Windows registry plays a vital role in customizing IE. One can explore registry by running regedit from command line. The regedit is a registry editor, which displays a tree structure of registry database. The GUIDEN utility comes with Visual Studio. It generates a GUID. All information about IE settings is stored in the registry under two keys. These keys are:

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer and HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer.

If one wants an option to all users then add use HKEY_LOCAL_MACHINE or if the option is only for the current user then one uses HKEY_CURRENT_USER key.

CHAPTER 3

COMPONENT OBJECT MODEL AND ATL

3.1: Introduction

As a software developer one cannot avoid COM these days. It is everywhere. DirectDraw is based on COM; Windows scripting is based on COM; ASP is based on COM and when you insert a Visio drawing into a Word document you are using COM. If one wants to write an application that will be Internet ready, using distributed transactions and message queuing, then one will be using COM.

COM stands for the Component Object Model. The plus in COM+ means that the version of COM in Windows 2000 will be the newest version available. In fact, people are already calling it COM+ 1.0 expecting COM+ 2.0 to appear sometime in the future - it would be nice if Microsoft rationalized their versioning. Then there is DCOM or Distributed COM. People talk about DCOM objects as if they are something special, something exciting, as if they add something to COM objects. They do not. A DCOM object is a COM object.

One hears about OLE objects and ActiveX objects. They are simply COM objects. The term OLE is not as trendy as it was a few years ago, but basically COM first appeared as OLE 2.0 on 16-bit Windows 3.1 which was used to allow you to take objects like Excel tables and link and embed them in Word documents (and visa versa). OLE stood for Object Linking and Embedding. These days people generally use the term when talking about the desktop technologies of compound documents (the generic term for documents that can contain objects from other applications) or the OLE controls that one sees used

on VB forms (the scriptable 'widgets' that are used to show grids or calendars). Similarly ActiveX, they are just COM objects.

Wind the clock a bit further forward and one comes to MTS components and to the more up to date term of COM+ components. Again, these are just COM objects, albeit COM objects that 'live' in a special environment. As the name suggests, a component that runs under Microsoft Transaction Server can be run under a transaction. This is a great boon for the bankers, but it is also vitally important for any distributed application.

A transaction, groups together, COM objects that are involved to perform some task. These may live in many places on the network, and if one object throws an error you will want to ensure that the other objects in the transaction know about this. (Imagine a relay race where the second runner gives up and goes home; someone has to tell the third and fourth runners otherwise they will remain standing on the track waiting for the baton.) MTS transactions ensure that every work in the transaction is performed correctly, or if just one object reports an error then the entire work in the transaction is undone.

COM+ components take this a step further by allowing the component to have access to other facilities including Microsoft Message Queue Server. But again, the actual object itself is a COM object, you write it the same way as you would write any other COM object - they just have access to different facilities.

3.2: COM - What exactly is it?

The Component Object Model (COM) is a way for software components to communicate with each other. It's a binary and network standard that allows any two components to communicate regardless of what machine they're running on (as long as the machines are connected), what operating systems the machines are running (as long as it supports COM), and what language the components are written in. COM further provides location transparency: it doesn't matter when one writes one's components whether the other components are in-process DLLs, local EXEs, or components located on some other machine.

This is unlike the C++ approach, which promotes reuse of source code. ATL is a perfect example of this. While source-level reuse works fine, it only works for C++. It also introduces the possibility of name collisions, not to mention bloat from having multiple copies of the code in your projects.

In the most simplistic terms COM is merely a code maintenance utility. Windows uses *dynamic link libraries* (DLLs) to dynamically load code as and when it is needed. This way code can be shared between several applications (they just load the same DLL) and the application can be efficient with memory because when it no longer needs a DLL it can unload it, and Windows will remove it from memory.

Windows lets one share the code at the binary level using DLLs. After all, that's how Windows apps function - reusing kernel32.dll, user32.dll, etc. But since the DLLs are

written to a C interface, they can only be used by C or languages that understand the C calling convention. This puts the burden of sharing on the programming language implementer, instead of on the DLL itself.

Although this sounds easy it is fraught with problems, which it can be summarized to three:

- To locate and load the DLL
- To obtain the code in the DLL
- To unload the DLL

COM solves all these problems by defining a *binary standard*, meaning that COM specifies that the binary modules (the DLLs and EXEs) must be compiled to match a specific structure. The standard also specifies exactly how COM objects must be organized in memory. The binaries must also not depend on any feature of any programming language (such as name decoration in C++). Once that's done, the modules can be accessed easily from any programming language. A binary standard puts the burden of compatibility on the compiler that produces the binaries, which makes it much easier for the folks who come along later and need to use those binaries.

Windows provides a function, called **`LoadLibrary`**, to load DLLs. The DLL is loaded after finding it using either the hard coded path or a search performed by windows to locate the DLL. Generally, hard coded paths are not used because one cannot guarantee that the path will exist on other machines. So this means that most developers allowed

Windows to locate the DLL. The problem with this was to make sure that the DLL was picked up before any other with the same name. Most developers took one of two approaches. Either they put all of their DLLs in the Windows System folder (along side the system DLLs) or they put them in the application's current folder.

The immediate problem with the first was that the System folder got filled with lots of DLLs many of which were private to specific applications. Often if an application was uninstalled its DLLs would remain and your hard disk got gradually smaller and smaller as more and more DLLs ate it.

More concerning was that if an application needed a specific version of one of the system DLLs then installing it in the System folder meant that it would overwrite an existing DLL, possibly preventing other applications from working. The other solution was to put the DLLs in an application's folder. This meant that these DLLs were essentially private to the application and this led to the problem of multiple versions of the same DLL in lots of folders on a disk. Moreover, if these folders were in the search path then this meant that **`LoadLibrary`** called by another application could pick up the wrong version.

Once the correct version of the DLL was loaded, the next task would be to obtain the code in the DLL. This process was tedious, it only worked well if the functions were C functions and was called in a specific way, and as a consequence it was error prone. Once the DLL has been loaded and get access to its code then there is one more issue to face. DLLs take up memory and so once one has finished using it, must be unloaded. This

allows the DLL to release any resources it may have loaded and free up the memory the DLL took up. Failing to unload DLLs results in applications using more and more memory.

COM DLLs are called COM servers and before an application can use one the DLL must be registered with the system, identifying the code in the server using unique IDs. This is a once-only registration. After that, applications can use code in the DLL by referring to the unique IDs, it does not use the DLL name or its path. COM will locate the right DLL using a specified ID and load it for the application; it will go through all the tedious process of getting access to the code in the DLL and it will unload the DLL when one no longer needs it. Real dynamic linking without the pain.

Since one registers the DLL's location with COM, it means that the DLL can be anywhere on the local machine, and since the DLL name is unimportant you can use any name that you like. But this is a rather rosy view, so what happens if one DLL overwrites another one? Well, the code in a DLL knows about the unique IDs that one registers and when COM loads the DLL it asks the DLL for the required code. If the DLL does not recognize the ID then it returns an error, which COM will pass back to the application. This means that the application will not have the facilities that it wanted and possibly it will signal an error to the user. However, there is no way that incorrect code can be loaded and run by accident, so the possibility of a catastrophic failure is reduced.

With a single DLL, the code can be accessed remotely or by other processes on the local machine, or loaded within the process that calls it. The DLL is written in the same way in all cases and the application that uses the code calls it in the same way irrespective of where the code is located. This is called location transparency and is a very powerful feature of COM.

3.3: Definitions of the Basic Elements

An *interface* is simply a group of functions. Those functions are called *methods*. Interface names start with *I*, for example IShellLink. In C++, an interface is written as an abstract base class that has only pure virtual functions. Interfaces may *inherit* from other interfaces. Inheritance works just like single inheritance in C++. Multiple inheritance is not allowed with interfaces.

A *coclass* (short for **c**omponent **o**bject **class**) is contained in a DLL or EXE, and contains the code behind one or more interfaces. The coclass is said to *implement* those interfaces.

A *COM object* is an instance of a coclass in memory. A COM "class" is not the same as a C++ "class", although it is often the case that the implementation of a COM class is a C++ class.

A *COM server* is a binary (DLL or EXE) that contains one or more coclasses.

Registration is the process of creating registry entries that tell Windows where a COM server is located. *Unregistration* is the opposite - removing those registry entries.

A *GUID* (rhymes with "fluid", stands for **g**lobally **u**nique **id**entifier) is a 128-bit number. GUIDs are COM's language-independent way of identifying things. Each interface and coclass has a GUID. Since GUIDs are unique throughout the world, name collisions are avoided (as long as one uses the COM API to create them). The term UUID (which stands for **u**niversally **u**nique **id**entifier) is also used at times. UUIDs and GUIDs are, for all practical purposes, the same. The objects and interfaces will need the same identifier on all machines so that any client can use the component. Further, no other object or interface may use that identifier, no matter where it came from. In other words, these identifiers must be globally unique. Fortunately, algorithms and data formats exist for creating such identifiers. By using the machine's unique network card ID, the current time, and other data, the identifiers, called GUIDs (globally unique identifiers) is created by a program called GUIDGEN.EXE. GUIDs are stored in 16-byte (128 bit) structures, giving 2^{128} possible GUIDs.

A *class ID*, or *CLSID*, is a GUID that names a coclass. An *interface ID*, or *IID*, is a GUID that names an interface.

There are two reasons GUIDs are used so extensively in COM:

- GUIDs are just numbers under the hood, and any programming language can handle them.

- Every GUID created, by anyone on any machine, is unique when created properly. Therefore, COM developers can create GUIDs on their own with no chance of two developers choosing the same GUID. This eliminates the need for a central authority to issue GUIDs.

An *HRESULT* is an integral type used by COM to return error and success codes. It is not a "handle" to anything, despite the *H* prefix.

Finally, the *COM library* is the part of the OS that one interacts with when doing COM-related stuff.

3.4: Working with COM Objects

COM is based on objects—but the objects aren't quite the objects one is used to in C++ or Visual Basic. First, COM objects are well encapsulated. One cannot gain access to the internal implementation of the object; there is no way of knowing what data structures the object might be using. In fact, the objects are so well encapsulated that COM objects are usually just drawn as boxes. Below is a drawing of an utterly encapsulated object. The implementation details are hidden completely.

Fig 3.1: Encapsulated Object

Interfaces: Communications with an object

The only way to access a COM object is through an interface. One can draw an interface called IFoo on an object like the one shown in Figure 2.

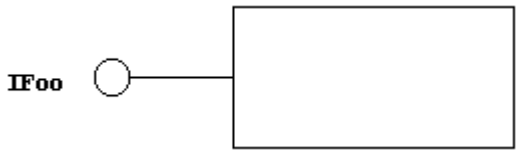

Fig 3.2: Interfaces: Communications with an object

For instance, the definition of **IFoo** might be:

```
class IFoo {
  virtual void Func1(void) = 0;
  virtual void Func2(int nCount) = 0;
};
```

There can be more than one function in the interface and that all of the functions are pure virtual functions: they do *not* have implementations in class IFoo. One is defining what functions are in the interface. Second—and more importantly—an interface is a *contract* between the component and its clients. In other words, an interface not only defines what functions are available, it also defines what the object *does* when the functions are called. This semantic definition is *not* in terms of the specific implementation of the object, so there's no way to represent it in C++ code (although one can provide a specific implementation in C++). Rather, the definition is in terms of the object's behavior, so that revisions to the object and/or new objects that also implement the interface (contract) are possible. In fact, the object is free to implement the contract in any way it chooses (as

long as it honors the contract). In other words, the contract has to be *documented outside of the source code*. This is especially important since clients won't get (and don't need) the source code.

This notion of a specific contract is crucial to COM and to component software in general. Without "ironclad" contracts, it would be impossible to interchange components. Interface contracts, like diamonds, are forever. In COM, once an interface is published contract by shipping a component, the contract is immutable—it cannot be changed in any way. One cannot add, delete or modify, because other components are depending on the contract. If a contract is changed, one will break that software. The internal implementation can be improved as long as the contract is honored.

If something is forgotten or if requirements change or do need to make modifications, one has to write a new contract. The standard OLE interface list has many of these: **IClassFactory** and **IClassFactory2**, **IViewObject** and **IViewObject2**, and so on. So if a new contract is written, how does software that only knows about the old contract still use the new components? Won't that mess old components up? COM objects can support multiple interfaces—they can implement multiple contracts. In fact, all useful COM objects support at least two interfaces. Visual ActiveX controls support about a dozen interfaces, most of them standard interfaces. In order for a component to support an interface, it has to implement each and every method in that interface, so this is a very substantial task. That's why tools like the Active Template Library (ATL) and so forth are popular: they provide implementation for all of the interfaces.

On the other hand, since these interfaces have something in common, it is not required to rewrite the whole implementation just to add a new interface that's almost exactly the same as the old one. COM supports inheritance of interfaces. As long as the functions already in IFoo, for example are not changed, another interface IFoo2 can be defined as follows:

```
class IFoo2 : public IFoo {
  // Inherited Func1, Func2
  virtual void Func2Ex(double nCount) = 0;
};
```

The COM binary standard applies to method calls, too—so COM defines what happens in order to call the function. Specifically, the same thing happens that happens for a virtual function call:

pFoo is dereferenced to find the *vtable* pointer in the object.

The *vtable* pointer is dereferenced and indexed to find the address of the function to be called. The function is called.

See Figure 3.3 for the numbered steps:

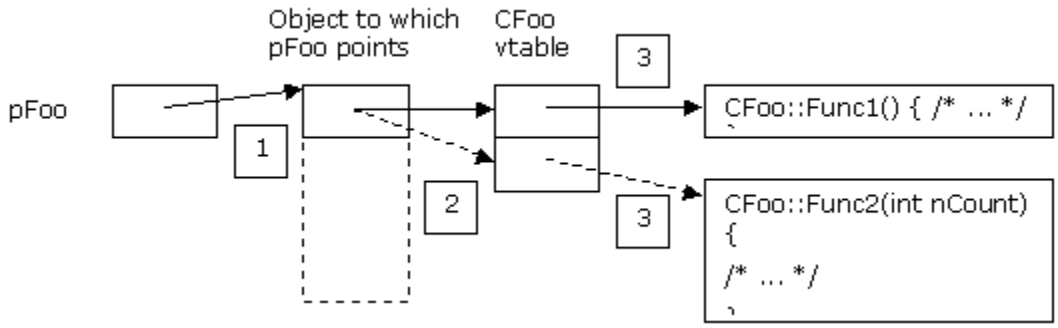

Figure 3.3: C++ virtual function calls through interface pointer

Every language has its own way of dealing with objects. For example, in C++ you create them on the stack, or use new to dynamically allocate them. Since COM must be language-neutral, the COM library provides its own object-management routines. A comparison of COM and C++ object management is listed below:

Creating a new object

- In C++, use operator new or create an object on the stack.
- In COM, call an API in the COM library.

Deleting objects

- In C++, use operator delete or let a stack object go out of scope.
- In COM, all objects keep their own reference counts. The caller must tell the object when the caller is done using the object. COM objects free themselves from memory when the reference count reaches 0.

Now, in between those two stages of creating and destroying the object, you actually have to *use* it. When you create a COM object, you tell the COM library what interface you need. If the object is created successfully, the COM library returns a pointer to the requested interface. You can then call methods through that pointer, just as if it were a pointer to a regular C++ object.

Creating a COM object

To create a COM object and get an interface from the object, you call the COM library API CoCreateInstance(). The prototype for CoCreateInstance() is:

```
HRESULT CoCreateInstance (
   REFCLSID  rclsid,
   LPUNKNOWN pUnkOuter,
   DWORD     dwClsContext,
   REFIID    riid,
   LPVOID*   ppv );
```

The parameters are:

rclsid

The CLSID of the coclass. For example, you can pass CLSID_ShellLink to create a COM object used to create shortcuts.

pUnkOuter

This is only used when aggregating COM objects, which is a way of taking an existing coclass and adding new methods to it. For our purposes, we can just pass NULL to indicate we're not using aggregation.

dwClsContext

Indicates what kind of COM servers we want to use. For this article, we will always be using the simplest kind of server, an in-process DLL, so we'll pass CLSCTX_INPROC_SERVER. One caveat: you should not use CLSCTX_ALL (which is the default in ATL) because it will fail on Windows 95 systems that do not have DCOM installed.

riid

The IID of the interface you want returned. For example, you can pass IID_IShellLink to get a pointer to an IShellLink interface.

ppv

Address of an interface pointer. The COM library returns the requested interface through this parameter.

When one calls CoCreateInstance(), it handles looking up the CLSID in the registry, reading the location of the server, loading the server into memory, and creating an instance of the coclass you requested.

Deleting a COM object

As stated before, you don't free COM objects, you just tell them that you're done using them. The IUnknown interface, which every COM object implements, has a method

Release(). You call this method to tell the COM object that you no longer need it. Once you call Release(), you must not use the interface pointer any more, since the COM object may disappear from memory at any time.

If your app uses a lot of different COM objects, it's vitally important to call Release() whenever you're done using an interface. If you don't release interfaces, the COM objects (and the DLLs that contain the code) will remain in memory, and will needlessly add to your app's working set. If your app will be running for a long time, you should call the CoFreeUnusedLibraries() API during your idle processing. This API unloads any COM servers that have no outstanding references, so this also reduces your app's memory usage.

Continuing the above example, here's how you would use Release():

```
// Create COM object as above.  Then...

if ( SUCCEEDED ( hr ) )
    {
    // Call methods using pISL here.

    // Tell the COM object that we're done with it.
    pISL->Release();
    }
```

The Base Interface - IUnknown

Every COM interface is derived from IUnknown. The name is a bit misleading, in that it's not an unknown interface. The name signifies that if you have an IUnknown pointer to a COM object, you don't know what the underlying object is, since *every* COM object implements IUnknown.

IUnknown has three methods:

AddRef() - Tells the COM object to increment its reference count. You would use this method if you made a copy of an interface pointer, and both the original and the copy would still be used.

Release() - Tells the COM object to decrement its reference count.

QueryInterface() - Requests an interface pointer from a COM object. You use this when a coclass implements more than one interface.

When a COM object is created with CoCreateInstance(), an interface pointer is returned. If the COM object implements more than one interface (not counting IUnknown), one uses QueryInterface() to get any additional interface pointers that you need. The prototype of QueryInterface() is:

```
HRESULT IUnknown::QueryInterface (
   REFIID iid,
   void** ppv );
```

The parameters are:

Iid - The IID of the interface you're requesting.

ppv - Address of an interface pointer.

QueryInterface() returns the interface through this parameter if it is successful. One must also call the Release() function to tell the COM object once they are done using the interface.

C++ can't express everything that needs to be expressed in an interface. COM objects can be DLLs to be used in-process, meaning in the same address space. So if one passes a pointer to some data to an in-process server, the server can dereference the pointer directly. But COM object can also be a local (out-of-process) server in separate EXE address spaces, or even accessed remotely. If one needs to pass a pointer to a COM method in such an object, the pointer is meaningless in any other address space. What's meaningful is the data to which the pointer points. This data has to be copied into the other address space—and perhaps back. This process of copying the right data is called marshalling. Thankfully, COM does the marshalling in most cases.

3.5: ATL or MFC

Most current ActiveX development uses MFC because MFC has been around the longest and many C++ developers know it. Also, unlike the other techniques, MFC enables developers to concentrate on the behavior of the object rather than the interface. The

downside (especially for Internet distribution) is the size of the controls and the need for run time DLL to exist with the container.

ATL is able to generate code each time you need it using templates. Thus, you don't need libraries or DLLs that have to ship along with the control. ATL requires that a class be derived from the several base classes existing as templates. Typically, developers will use the ATL wizard to create the classes automatically. ATL also has drawbacks. It's much more difficult to deal with interfaces using ATL since one must create each interface one needs for the application. Also, ATL does not support the Class Wizard that's able to automatically keep the Object Description Language and interface definition language files in synch with your code. The wizards leave a lot to be desired.

MFC, Visual Basic, Visual J++, Visual FoxPro, and Delphi are so popular for writing ActiveX controls: their run times implement all the methods of the dozen or so interfaces, allowing one to concentrate on the problem at hand. But using these techniques to implement your ActiveX controls is a Faustian bargain: the run times for all of these products are, to one degree or another, big and slow. If you need your controls to be small and fast, you'll either implement them in C++ directly or you'll use ATL.

ATL's forte is in the development of components but not necessarily in the use of component's. Dean McCrory, one of the key developers of Microsoft Foundation Classes, has often said that one of the big advantages of using MFC is that you get to reuse code that you didn't write or debug—someone else's code. So MFC gives you print preview

virtually for free—that's several thousand lines of code one doesn’t have to write. And it makes it easy to write OLE in-place editing servers.

That code is especially valuable when it's been tested and debugged. No library is perfect. But the code in a mature library has been used and debugged by many, many programmers—and, the libraries are likely to be faster and more robust than one has time to write by himself. They basically encapsulate all of the developer's expert knowledge and allow one to stand on their shoulders by reusing it.

Some of the reasons for using ATL instead of MFC:

- ATL provides dual interface support as part of its basic implementation. MFC requires a lot of additional work to add dual support.
- ATL provides support for all of COM's threading models, in particular the free threading model. MFC does not and probably will never support the free threading model because MFC is thread safe only at the class level.
- ATL does not require MFC's 1 Meg runtime (MFC40.DLL). This isn't necessarily an issue because it is present on most systems, however it definitely increases load times for your component.

3.6: ActiveX Template Library

ATL was originally designed as a way to write fast, small COM components. It was especially intended for Automation components that could, for instance, implement business rules and database access in a multitier architecture. In its first version, ATL did

not have any facilities for any sort of user interface—ATL 1.0 controls could not be visual ActiveX controls, for instance (not without almost as much work as implementing it in C++ without ATL, that is). Version 2.0 added the templates necessary to build visual ActiveX controls.

Features

ATL gives you several important features, including:

- All of the power of C++.
- No run-time library, unless one wants to use it.
- A relatively high-level way of abstracting objects and interfaces.
- Automatic handling of class factory, object creation, reference counting, and QueryInterface.
- Stock implementations of standard interfaces.
- The need for speed—with ease

ATL gives you the leanest, meanest code around. First off, it relies on no run-time libraries at all—so the only time your control needs to use a run-time library is if *your* code makes use of it. Second, through the magic of templates, ATL controls contain only the code they actually need. So ATL controls are very comparable in size and speed to controls hand-coded by COM experts in C++. But they're far easier to write.

Since a module (a DLL or EXE) can implement more than one component, the ATL Wizards break the component creation process in the integrated development

environment (IDE) up into two steps. First, you create the module, and then you add the components into the module.

Handling Messages

The most basic ATL window class is **CWindow**, an object-oriented wrapper for the Windows API. **CWindow** makes window manipulation easier, and remarkably, adds no overhead. Unfortunately, **CWindow** does not let you define how a window responds to messages. One can use **CWindow** functions to center a window or hide a window, even send a message to a window, but what happens when the message reaches the window depends on its window class, and that was determined when the window was created. If it was created as an instance of the "button" class, it acts as a button; if it's a "list box" then that's how it behaves. There is no way, using **CWindow**, to alter that.

Fortunately, ATL has another class, **CWindowImpl**, which does allow you to specify new behavior for a window. **CWindowImpl** inherits from **CWindow**, one can still use all those handy **CWindow** member functions, but what makes **CWindowImpl** special is that it also lets you define message-handling behavior. In traditional Windows programming, when one wants to specify a window's response to messages, one writes a window procedure; in ATL, one defines a "message map" in the ATL window class.

First, one derives the class from **CWindowImpl**,

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
```

The new class's name must be passed as an argument to the **CWindowImpl** template. Within the class definition, the message map is defined as:

```
BEGIN_MSG_MAP(CMyWindow)
  MESSAGE_HANDLER(WM_PAINT,OnPaint)
END_MSG_MAP ()
```

The following line:

MESSAGE_HANDLER (WM_PAINT, OnPaint)

means, "When the window receives a **WM_PAINT** message, invoke member function **CMyWindow::OnPaint**". The member functions that handle the messages are defined as:

```
LRESULT OnPaint(
  UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{ ... }
```

The four arguments to the handler functions are the message identifier, two parameters whose contents depends on the message, and a flag that the handler can use to indicate that the message has been dealt with or needs further processing. When the window receives a message, the message map entries are examined starting at the top, so it's a good idea to put the most frequent messages first. If no matching entry is found in the message map, the message is passed to the default window procedure.

Message Maps

There are three groups of message handling macros:

- **Message handlers**, for all messages (such as **WM_CREATE**, **WM_PAINT**,..)
- **Command handlers**, specifically for **WM_COMMAND** messages (typically sent by predefined child window controls, such as buttons or menu items).
- **Notification handlers**, specifically for **WM_NOTIFY** messages (typically sent by common controls, such as status bars or list view controls).

Message Handler Macros

There are two message handler macros:

MESSAGE_HANDLER

MESSAGE_RANGE_HANDLER

The first macro maps a specific message to a handler function; the second maps a range of messages to a handler. Message handler functions have the following prototype:

LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

where **uMsg** identifies the message, and **wParam** and **lParam** are the message parameters (their contents depend on the message). A handler function uses the bHandled flag to indicate whether it handled the message. If bhandled is set to **FALSE** in the handler function, the rest of the message map will be searched for another handler for the message. This can be useful when one wants to have multiple handlers for the same message, perhaps in different classes using chaining, or if one just wants to do something

in response to a message but not actually handle it. The bHandled flag is set to **TRUE** before the function is called, so unless the function explicitly sets bHandled to **FALSE**, no further message map processing will be performed.

Command Handler Macros

Command handler macros only handle commands (**WM_COMMAND** messages), but they allow one to specify the mapping in terms of the command code or the identifier of the control sending the command.

COMMAND_HANDLER maps commands with a specified command code from a specified control to a handler function.

COMMAND_ID_HANDLER maps commands with any command code from a specified control to a handler function.

COMMAND_CODE_HANDLER maps commands with a specified command code from any control to a handler function.

COMMAND_RANGE_HANDLER maps commands with any command code from a range of controls to a handler function.

COMMAND_RANGE_CODE_HANDLER maps commands with a specified command code from a range controls to a handler function.

Command handler functions have the following prototype:

LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

where **wNotifyCode** is the notification code, **wID** is the identifier of the control sending the command, **hWndCtl** is the handle of the control sending the command, and **bHandled** is as described previously.

Notification Handler Macros

Notification handler macros map notifications (**WM_NOTIFY** messages) to functions, depending on the notification code and the identifier of the control sending the notification. These macros are equivalent to the command handler macros, except that these handle notifications, not commands. The different Notification handler macros are

NOTIFY_HANDLER

NOTIFY_ID_HANDLER

NOTIFY_CODE_HANDLER

NOTIFY_RANGE_HANDLER

NOTIFY_RANGE_CODE_HANDLER

Notification handler functions have the following prototype:

LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

where **idCtrl** is the identifier of the control sending the notification, **pnmh** is a pointer to an **NMHDR** structure, and **bHandled** is as described previously.

Notifications include a pointer to a notification-specific structure; for example, when a list view sends a notification, it includes a pointer to a **NMLVDISPINFO** structure. All such structures have an **NMHDR** header, which is what pnmh points to.

Adding Functionality to Existing Window Classes

There are several ways to add functionality to an existing window class. If the class is an ATL window class, one can derive a new class from it, as described in Base Class Chaining. This is basically C++ inheritance with just a small twist because of message maps. To extend the capabilities of a predefined window class, such as the button or list box controls, one can super class it. This defines a new class that is based on the predefined class, but with a message map that adds to the functionality of the underlying window class.

Sometimes one will want to modify the behavior of a window *instance*, rather than a *class*—perhaps one needs an edit control on a dialog box to do something special. In that case one can write an ATL window class that subclasses the existing edit control. Messages directed to the edit control are routed first through the message map of the sub classing object.

Also one could make the edit control a contained window, a window that passes the messages it receives to the message map of its containing class for processing; the containing class can implement special message handling behavior for the contained window.

Finally, there is message reflection, in which the window receiving a message does not handle it, but instead reflects it back to the sending window, which must then handle the message itself. This technique can help make controls more self-contained.

Base Class Chaining

Having defined an ATL windowing class with some functionality, one can derive a new class from it to take advantage of inheritance. For example:

```
class CBase: public CWindowImpl< CBase >
// simple base window class: shuts down app when closed
{   BEGIN_MSG_MAP( CBase )
     MESSAGE_HANDLER( WM_DESTROY, OnDestroy )
  END_MSG_MAP()
  LRESULT OnDestroy( UINT, WPARAM, LPARAM, BOOL& )
  {     PostQuitMessage( 0 );
    return 0;
  }      };

class CDerived: public CBase
// derived from CBase; handles mouse button events
{   BEGIN_MSG_MAP( CDerived )
     MESSAGE_HANDLER( WM_LBUTTONDOWN, OnButtonDown )
```

```
  END_MSG_MAP()
  LRESULT OnButtonDown( UINT, WPARAM, LPARAM, BOOL& )
  {     ATLTRACE( "button down\n" );
    return 0;
  }};
// in WinMain():
  ...
  CDerived win;
  win.Create( NULL, CWindow::rcDefault, "derived window" );
```

The problem with this code is that when one run's this program in the debugger, a window will appear. If one clicks inside the window, "button down" will appear in the Debug output window. This is behavior of **CDerived**. However, when closes the **CDerived** window, the application will not stop executing, even though **CBase** handles **WM_DESTROY** messages and **CDerived** inherits from **CBase**.

The reason is that one has to explicitly "chain" one message map to another:

```
BEGIN_MSG_MAP( CDerived )
   MESSAGE_HANDLER( WM_LBUTTONDOWN, OnButtonDown )
   CHAIN_MSG_MAP( CBase ) // chain to base class
END_MSG_MAP()
```

Any messages not handled in the message map of **CDerived** will be passed to the message map of **CBase** for processing. The chaining is not done automatic because

of multiple inheritance, which is fundamental to ATL's architecture. In general, there is no way to know which one of multiple base classes to chain to, so the decision is left to the programmer.

Alternate Message Maps

Message map chaining allows one message map to handle messages from multiple window classes, which creates a problem: The same **WM_PAINT** handler (for instance) is called for all window classes in the chain, even though you presumably want different painting behavior depending on the class. To solve this problem, ATL uses *alternate message maps*: it divides a message map into sections, each identified by a number. Each such section is an alternate message map:

```
// in class CBase:
  BEGIN_MSG_MAP( CBase )
    MESSAGE_HANDLER( WM_CREATE, OnCreate1 )
    MESSAGE_HANDLER( WM_PAINT, OnPaint1 )
    ALT_MSG_MAP( 100 )
    MESSAGE_HANDLER( WM_CREATE, OnCreate2 )
    MESSAGE_HANDLER( WM_PAINT, OnPaint2 )
    ALT_MSG_MAP( 101)
    MESSAGE_HANDLER( WM_CREATE, OnCreate3 )
    MESSAGE_HANDLER( WM_PAINT, OnPaint3 )
  END_MSG_MAP()
```

The message map of **CBase** consists of three sections: the default message map (implicitly numbered 0) and two alternate message maps (numbered 100 and 101).

When the message map is chained, the identifier of the desired alternate message map is specified. For example:

```
class CDerived: public CBase {
  BEGIN_MSG_MAP( CDerived )
    CHAIN_MSG_MAP_ALT( CBase, 100 )
  END_MSG_MAP()
  ...
```

The message map of **CDerived** chains to alternate message map 100 in **CBase**, so when a **CDerived** window receives a **WM_PAINT** message, handler function **CBase::OnPaint2** will be invoked.

Message Reflection

Message reflection involves getting a window to respond to messages that it *sends out*.

When the user interacts with a control, the control typically informs its parent window by sending it a **WM_COMMAND** or **WM_NOTIFY** message; the parent window then takes some action in response. For example:

```
class CParentWindow: CWindowImpl<CParentWindow>
{
  // assume that this window will have a child
  // button control with an ID of ID_BUTTON
```

```
  BEGIN_MSG_MAP( CParentWindow )
    COMMAND_ID_HANDLER( ID_BUTTON, OnButton )
    MESSAGE_HANDLER( WM_CTLCOLORBUTTON, OnColorButton )
    ...
```

When the button is clicked, it sends a command to the parent window, and **CParentWindow::OnButton** is invoked. Similarly, when the button is about to be drawn, it sends a **WM_CTLCOLORBUTTON** message to the parent; **CParentWindow::OnColorButton** responds with the handle of a brush. In some cases, it is useful to have the control itself respond to the messages it sends, rather than the parent window. ATL provides a facility for *message reflection*: when a control sends its parent a message, the parent can reflect the message back to the control.

```
class CParentWindow: CWindowImpl<CParentWindow>
{
  BEGIN_MSG_MAP( CParentWindow )
    MESSAGE_HANDLER( WM_CREATE, OnCreate )
    MESSAGE_HANDLER( WM_DESTROY, OnDestroy )
    ...other messages that CParentWindow will handle...
    REFLECT_NOTIFICATIONS()
  END_MSG_MAP()
  ...
```

When the parent window receives a message, it examines its message map; if none of the message map entries match the message, the **REFLECT_NOTIFICATIONS** macro

causes the message to be sent back to the control that sent it. The control provides handlers for the reflected messages, like this:

```
class CHandlesItsOwnMessages: CWindowImpl<CHandlesItsOwnMessage>
{
public:
  DECLARE_WND_SUPERCLASS( _T("Superbutton"), _T("button") )
  BEGIN_MSG_MAP( CHandlesItsOwnMessage )
    MESSAGE_HANDLER( OCM_COMMAND, OnCommand )
    MESSAGE_HANDLER( OCM_CTLCOLORBUTTON, OnColorButton )
    DEFAULT_REFLECTION_HANDLER()
  END_MSG_MAP()
  ...
```

The message identifiers for reflected messages start with OCM_, not WM_. This allows one to distinguish between messages originally intended for the control and messages that have been reflected back to the control. The **DEFAULT_REFLECTION_HANDLER** macro takes care of unhandled reflected messages.

ATL Dialog Box Classes

ATL provides **CSimpleDialog** and **CDialogImpl** to simplify using a dialog box resource from within your application. **CSimpleDialog** is a class that creates modal dialog boxes from templates. It provides command handlers for standard buttons such as **OK** and

Cancel. One can think of **CSimpleDialog** as a kind of message box, except that you design the "message box" in the dialog box editor so you have control over its layout.

To display such a dialog box in response to, say, a user clicking **About** in your application's **Help** menu, you would add the following command handler to your main window's class:

```
BEGIN_MSG_MAP( CMyMainWindow )
  COMMAND_ID_HANDLER( ID_HELP_ABOUT, OnHelpAbout )
  ...

LRESULT OnHelpAbout( WORD, WORD, HWND, BOOL& )
{
  CSimpleDialog<IDD_DIALOG1> dlg;
  int ret = dlg.DoModal();
  return 0;
}
```

The ID of the dialog box resource (**IDD_DIALOG1**) is passed to the **CSimpleDialog** template. **DoModal** displays the dialog box. When the user clicks **OK**, **CSimpleDialog** closes the dialog box and returns the ID of the button. (**CSimpleDialog** has command handlers for **IDOK**, **IDCANCEL**, **IDABORT**, **IDRETRY**, **IDIGNORE**, **IDYES**, and **IDNO**.)

CSimpleDialog only handles simple modal dialog boxes. For more complicated dialog boxes, or for modeless dialog boxes, there is **CDialogImpl**. (**CSimpleDialog** is actually a limited kind of **CDialogImpl**). To implement a modeless dialog box, one has to derive a class from **CDialogImpl**. Pass the name of the new class as a template argument to **CDialogImpl**, as you would for **CWindowImpl**:

```
class CMyModelessDialog: public CDialogImpl<CMyModelessDialog>
{
```

Unlike **CSimpleDialog**, one doesn't pass the dialog box resource identifier as a template argument; however, one still have to "connect" this class with a dialog box resource, and that is achieved by defining an enum in the class:

```
public:
  enum { IDD = IDD_DIALOG1 };
```

To declare a message map:

```
BEGIN_MSG_MAP( CMyDialog )
  MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog )
  MESSAGE_HANDLER(WM_CLOSE, OnClose )
  ...
END_MSG_MAP()
```

The handler function definitions are custom, but one thing that has to be done is to call **DestroyWindow** in response to a **WM_CLOSE** message since a modeless dialog box is implemented:

```
LRESULT OnClose( UINT, WPARAM, LPARAM, BOOL& )
{
  DestroyWindow();
  return 0;
}
...
}; // CmyModelessDialog
```

To get this dialog box on the screen, an instance of the class has to be constructed and call its **Create** function:

```
CMyModelessDialog dlg;
dlg.Create( wndParent );
```

ATL provides a simplified, elegant, and powerful Windows programming model. Beyond the convenience of wrapper functions, message maps, and macros, techniques such as chaining, window subclassing and superclassing, contained windows, and reflected messages afford great flexibility in the design and implementation of window and dialog box classes. Perhaps most impressive is that, even while providing all this power and flexibility, ATL imposes only minimal memory and execution time overhead.

Threading Models

Every COM component has a Threading Model registry attribute that one can specify when one develops the component. This attribute determines how the component's objects are assigned to threads for method execution. If one thinks of the building as an application's process, each apartment is a distinct area in which a COM object can be created. The object wizard provides options to choose form while creating different objects.

Apartment Threading Model (single threaded apartments)

This model was introduced in the first version of COM with Windows NT3.51 and later Windows 95. The apartment model consists of a multithreaded process that contains only one COM object per thread. **Single Threaded Apartments** (STA)- This also means that each thread can be called an apartment and each apartment is single threaded. All calls are passed through the Win32 message processing system. COM ensures that these calls are synchronized. Each thread has its own apartment or execution context and at any point in time, only one thread can access this apartment. Each thread in an apartment can receive direct calls only from a thread that belongs to that apartment. The call parameters have to be marshaled between apartments. COM handles marshalling between apartments through the Windows messaging system.

Free threading Model(multi threaded apartments)

This model was introduced with Windows NT 4.0 and Windows95 with DCOM. The free threaded model allows multiple threads to access a single COM object. Free threaded COM objects have to ensure thread synchronization and they must implement message handlers, which are thread aware and thread safe. Calls may not be passed through the Win32 messaging system nor does COM synchronize the calls, since the same method may be called from different processes simultaneously. Free threaded objects should be able to handle calls to their methods from other threads at any time and to handle calls from multiple threads simultaneously. Parameters are passed directly to any thread since all free threads reside in the same apartment. These are also called **Multi-Threaded Apartments** (MTA)

Both Apartment and Free threaded Model

It is possible for a process to have both the apartment and free threaded model. The only restriction is that you can have only one free threaded apartment but you can have multiple single threaded apartments. Interface pointers and data have to be marshaled between apartments. Calls to objects within the STAs will be synchronized by Win32 whereas calls to the MTAs will not be synchronized at all.

Threaded Neutral Apartment Model

Components that use the Thread Neutral Apartment model (TNA), mark themselves as Free Threaded or Both. Here the component instances run on the same thread type as the caller's thread. Each instance of a COM class can run on a different thread each time a method is called. When a thread is executing a method in a COM object, and that method

creates a new object, MTS will suspend the current thread and create a new thread to handle the new object. Like the MTA, TNAs allow more than one thread to enter an apartment. However, once a thread has entered an apartment, it obtains an apartment-wide lock and no other thread can enter that apartment until it exits. This model was introduced into MTS and COM+ to ensure that context switches are faster.

The **interface** chosen can be dual or custom. That means that the object can be called via both the custom interface (like the ones developed by developers) and via the Automation, or **IDispatch**, interface. Scripting languages use Automation interfaces only, so selecting "Dual" allows the object to be used from a scripting language. The performance is much better when using a custom interface, so if the client can use it (most can), it should.

Aggregation is a special way of containing (or nesting) objects inside other objects. Aggregation is normally quite tricky, but ATL handles all the details. The object is more flexible if it's aggregated.

CHAPTER 4

TOOLBAR DESIGN

4.1: Overview

The IE toolbar consists of a COM component supporting IDeskband and a few other necessary interfaces for which IE looks for when loading registered toolbars, explorer bars and deskbands. The RBDeskband ATL Object Wizard provides most of the framework for the toolbar. The design consists of creating a project, a new COM object to house the toolbar, and a few CWindowImpl classes using the CWindowImpl ATL Object Wizard. Then connecting these parts together, will produce the IE toolbar. Visually the toolbar consists of an editbox and a toolbar with the five buttons on it. In actuality the toolbar consists of the fore mentioned and a non-visible window that is used to reflect messages to the Toolbar window, which will process or forward messages to itself and the edit box.

4.2: Creating the Project

Creating the module is almost trivial: we select **New…** from the **File** menu, and then select the **Projects** tab and fill in the directory and project name. The project dialog box will look something like the following.

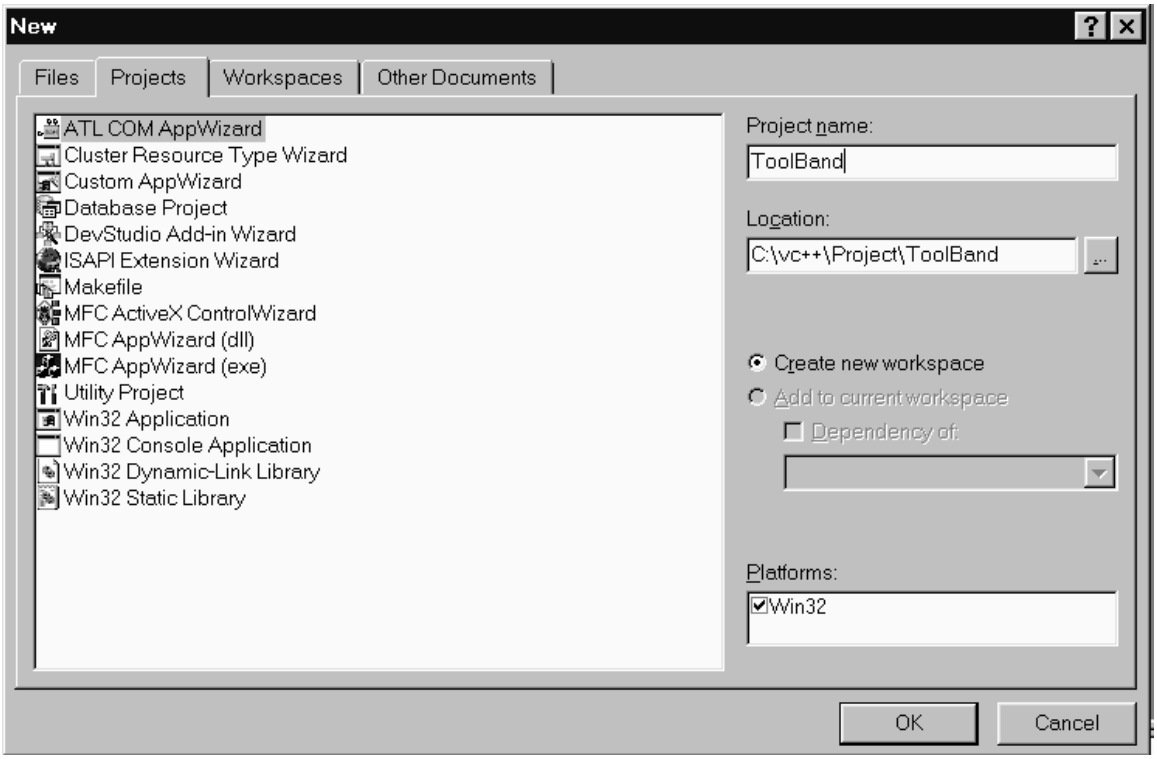

Fig 4.1: Project Dialog Box

When one clicks **OK**, the following wizard is shown

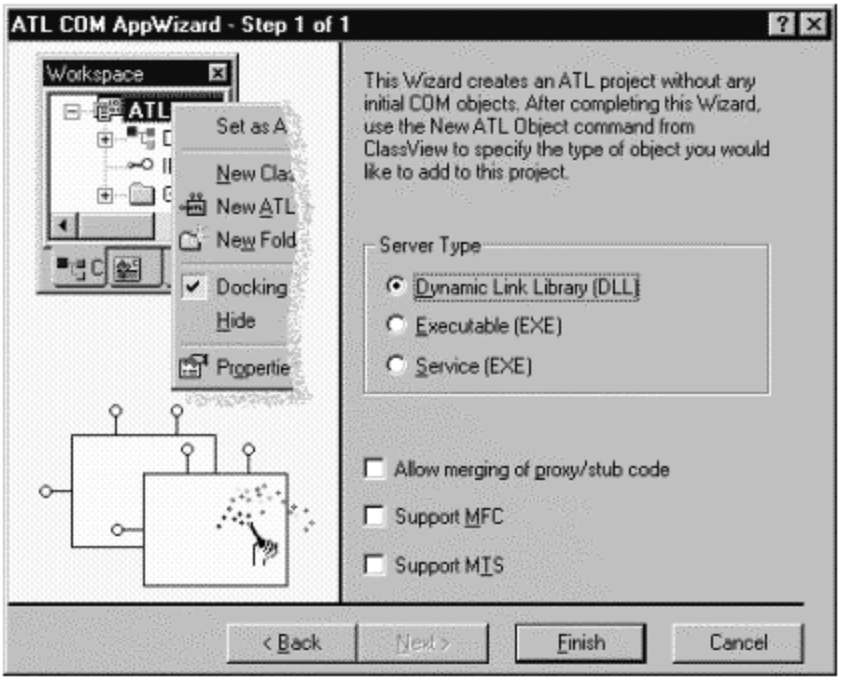

Fig 4.2: ATL COM AppWizard

The main choice here is the type of module—DLL (for in-process server), EXE (for out-of-process server), or NT Service EXE. For the ToolBar an in-process (DLL) is chosen. The last three check boxes are not required for this project since we're not using MFC, MTS, or even proxy/stub code. On clicking **Finish**, a project that containing the following group of files is created.

An **.idl** file for the project that contains only the initial declarations for the type library.

A linker **.def** file that contains exports for the four functions COM DLLs must export.

An **.rc** resource file that contains a version resource and a string for the project name.

A **resource header file** that contains resource ID definitions.

stdafx.h and **stdafx.cpp** files to do system includes properly for fast builds using precompiled headers.

And finally, the source code **.cpp** file that contains implementations for all of the global functions necessary for a COM DLL.

The ToolBand.cpp files includes the declaration of a global object and a map:

```
#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "ToolBand.h"
#include "ToolBand_i.c"
CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

The file **initguid.h** is a standard OLE system file included in exactly one file in the project so that the globally unique identifier (GUID) structures are defined.

ToolBand.h and **ToolBand_i.c** are generated by Microsoft Interface Definition Language (MIDL) from the IDL file when the project is built. Because MIDL runs before the C++ compiler does, the files will be created in plenty of time for the build.

ToolBand.h contains the declarations of the interfaces and components.

ToolBand_i.c contains the definitions of the GUIDs being used.

Next is a global declaration of a variable called **_Module**. This object contains the class object (class factory) plus all of the other module-oriented code, such as registration of

classes, and so on. The object wizard will fill this map in. Each element of the map array will contain a CLSID and a C++ class name. The **_Module** object reads this map so it can create objects based on their CLSID.

The **DllMain** function simply calls **Init** and **Term** functions in the **_Module** object. While Init initializes the data members Term releases the data members

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
                    LPVOID /*lpReserved*/)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      _Module.Init(ObjectMap, hInstance, &LIBID_BEEPCNTMODLib);
      DisableThreadLibraryCalls(hInstance);
   }
   else if (dwReason == DLL_PROCESS_DETACH)
      _Module.Term();
   return TRUE;    // ok
}
```

DllMain also calls **DisableThreadLibraryCalls** so the DLL won't get a call each time a new thread attaches it.

The four functions COM DLLs must have are also in this file:

- **STDAPI DllCanUnloadNow()** determines whether the DLL that implements this function is in use. If not, the caller can safely unload the DLL from memory.

- **STDAPI DllGetClassObject()** retrieves the class object from a DLL object handler or object application. **DllGetClassObject** is called from within the CoGetClassObject function when the class context is a DLL.
- **STDAPI DllRegisterServer()** instructs an in-process server to create its registry entries for all classes supported in this server module. If this function fails, the state of the registry for all its classes is indeterminate.
- **STDAPI DllUnregisterServer()** instructs an in-process server to remove only those entries created through DllRegisterServer.

4.3: Creating The DeskBand Object

For the DLL to actually expose something, one needs to add an IDeskBand derived component to the project container. The easiest way to add a COM component is to use the ATL Object Wizard. Just select **New ATL Object…** from the **Insert** menu, to get a dialog box that looks like the following.

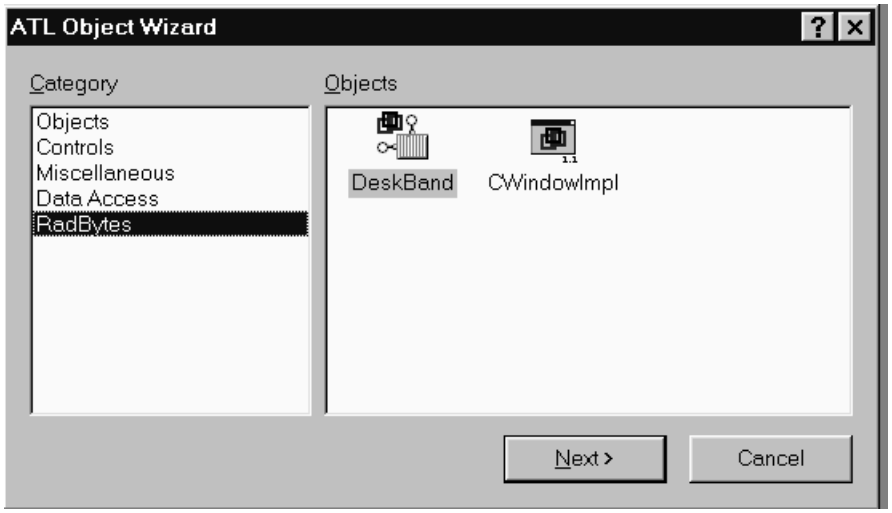

Fig 4.3: ATL Object Wizard

Selecting DeskBand from the RadBytes category, the Wizard creates the files necessary for the DeskBand's base implementation. Selecting Next opens a property sheet that allows one to set the names of the object and to set its attributes.

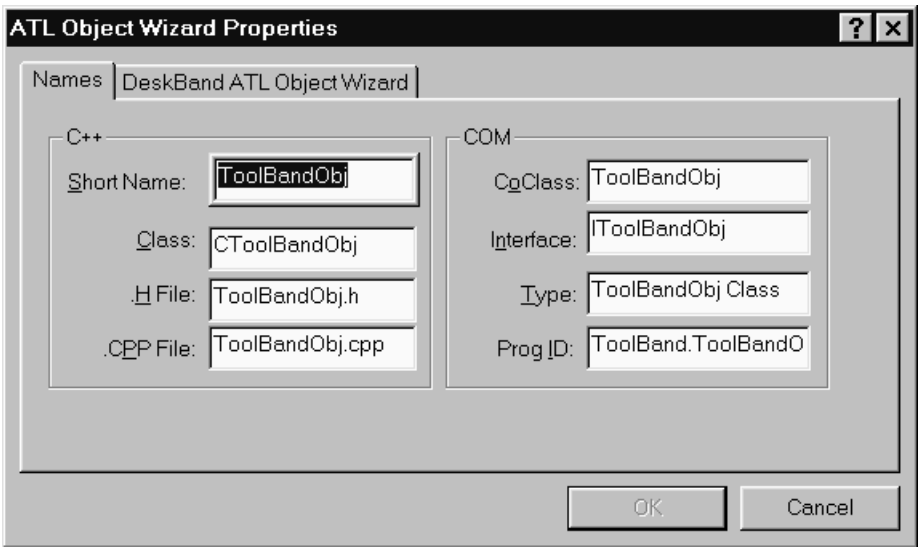

Fig 4.4: ATL Object Properties Wizard

Clicking on the DeskBand ATL Object Wizard will provide a choice of creating the an Internet explorer toolbar or a DeskBand or a Internet Explorer left side band or Internet explorer Bottom side band. Since the project involves creating an Internet Explorer ToolBar , the corresponding option is chosen.

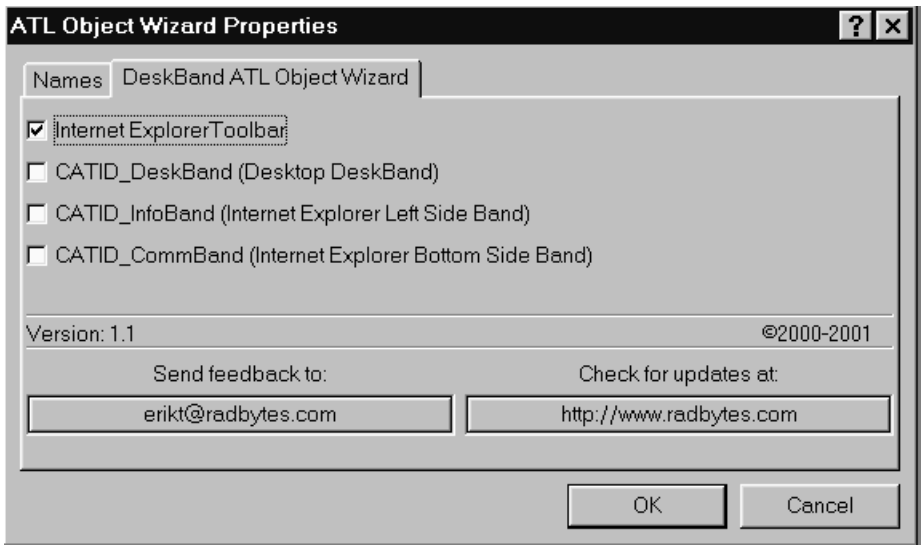

Fig 4.5: DeskBand ATL Object Wizard

The ATL Object Wizard adds several files to the project and made changes to several others. The wizard creates the following files

ToolBandObj.rgs, it consists of the register information for the toolbar project.

ToolBandObj.h, it defines all the COM interfaces needed for the toolbar and also outlines the functions in each of the interface.

ToolBandObj.cpp, it provides implementation for all functions defined in the header file.

The wizard also updates the idl file accordingly to associate a UUID for the interface created. There are various attributes in the IDL, enclosed in square brackets. The attributes always apply to the thing immediately after, so the UUID attribute applies to the interface—it's the IID for the interface. (UUID, or universally unique identifier, is a synonym for GUID). There is IDL code to define the object as well. For instance, the code fragment to define the object might look like this:

```
[uuid(0E1230F8-EA50-42A9-983C-D22ABC2EED3B)]
coclass ToolBandObj
        {
                [default] interface IToolBandObj;
        };
```

This is how the CLSID is associated with the class—and how the set of interfaces the class implements is defined. The project has the DeskBand implementation that can be modified to produce the toolbar needed for the project. Once the window classes have been created needed and then make appropriate changes to the Desbkand object to use the window classes. The ATL Object Wizard added implementation and header files for the object. The ToolBandObj.h file contains the declaration of the component's implementation class:

```
class ATL_NO_VTABLE CToolBandObj :
        public CComObjectRootEx<CComSingleThreadModel>,
        public CComCoClass<CToolBandObj, &CLSID_ToolBandObj>,
        public IDeskBand,
        public IObjectWithSite,
        public IPersistStream,
        public IInputObject,
```

```
        public IDispatchImpl<IToolBandObj, &IID_IToolBandObj,
&LIBID_TOOLBANDLib>
{
public:
        CToolBandObj();
DECLARE_REGISTRY_RESOURCEID(IDR_TOOLBANDOBJ)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_CATEGORY_MAP(CToolBandObj)
//      IMPLEMENTED_CATEGORY(CATID_InfoBand)
//      IMPLEMENTED_CATEGORY(CATID_CommBand)
//      IMPLEMENTED_CATEGORY(CATID_DeskBand)
END_CATEGORY_MAP()
BEGIN_COM_MAP(CToolBandObj)
        COM_INTERFACE_ENTRY(IToolBandObj)
        COM_INTERFACE_ENTRY(IOleWindow)
        COM_INTERFACE_ENTRY_IID(IID_IDockingWindow,
IDockingWindow)
        COM_INTERFACE_ENTRY(IObjectWithSite)
        COM_INTERFACE_ENTRY_IID(IID_IDeskBand, IDeskBand)
        COM_INTERFACE_ENTRY(IPersist)
        COM_INTERFACE_ENTRY(IPersistStream)
        COM_INTERFACE_ENTRY(IInputObject)
        COM_INTERFACE_ENTRY(IDispatch)
```

END_COM_MAP()

It also has the COM map, used by ATL's implementation of **QueryInterface**, must contain an entry for each and every interface (except **IUnknown**) used by the control.

The class also contains a number of macros. **ATL_NO_VTABLE** tells the compiler to not construct a vtable for this class. This can only be used on classes that are not themselves instantiated. ATL derives a class from **CToolBandObj** and instantiates that. **CToolBandObj** doesn't need a vtable because no objects of exact type **CToolBandObj** will ever be created, so we can save memory by using ATL_NO_VTABLE.

The object created will actually be of type **CComObject< CToolBandObj >**—in other words, the **CComObject** template class will be used with the object type as a parameter. A derivation diagram describing this follows.

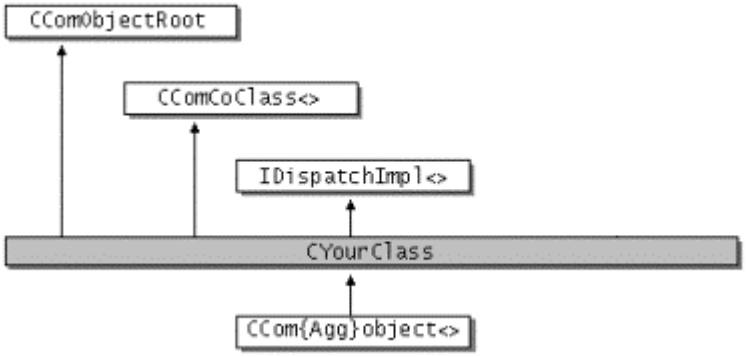

Fig 4.6: COM object Derivation Diagram

The primary job of **CComObject** is to provide implementations of the **IUnknown** methods. These *must* be provided in the most-derived class so that the implementations

can be shared by all of the interfaces that derive from **IUnknown**. **CComObject**'s methods just call the implementations in **CComObjectRootEx**.

The next thing to note is the COM map, which contains macros for the two interfaces we implement—our custom interface and **IDispatch**, the Automation interface. An entry for **IUnknown** is not present because it's assumed.

4.4: Creating the window classes

The project consists of three window classes. One for the Edit Box, one for the toolbar, and one for message reflection back to the toolbar.

The Edit Window

This window is created by deriving the class from the standard EDIT Button window class. Methods will be added to the class to help support functionality of the toolbar. The files associated with this window are BandEditCtrl.h and BandEditCtrl.cpp. For the EditQuote implementation, the keystrokes from the user must be processed and let the host that created the deskband object know our edit box has focus. To accomplish the first part, the DeskBand object will implement the IInputObject interface. So the host will query for that interface and so that one can recieve messages and be given the chance to recieve focus. When the host sends the band messages to process they come through the IInputObject::TranslateAccelerator method. The DeskBand will implement this method and it is best if the edit box, which will process the message, copy the

TranslateAcceleratorIO method definition so the Deskband can forward the message easily through a logical method call.

The Toolbar Window

The toolbar can be created via the resource editor, Once you have your toolbar you need to create the toolbar dynamically. The CBandToolBarCtrl is inherited from CToolBarCtrl and is used to create the toolbar dynamically.

DWORD dStyle = WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | CCS_NODIVIDER | CCS_NORESIZE | CCS_NOPARENTALIGN | TBSTYLE_TOOLTIPS | TBSTYLE_FLAT;

HWND hWnd = m_wndToolBar.CreateSimpleToolBarCtrl(hWndChild, IDR_TOOLBAR_TEST, FALSE, dStyle);

This is done in the RegisterAndCreateWindow function that is called from SetSite method. The toolbar consists of one edit box and 5 buttons. Each button is associated with a menu, which brings a different functionality to the toolbar. Each menu is created in the resource editor and every menu item is associated with a different functionality. The homepage button is associated with customizing and uninstallation. The web button is used to search through the top search engines as well as the dictionary and thesaurus. The library button is associated with searching the Uclid and OhioLink. The Journal

button is associated with the Electronic Journal Center and finally the database button is associated with ACM and IEEE.

Invisible window

The best way to handle the messages of the toolbar is to let the control handle its own messages via "Message Reflection". The best way to reflect the messages is to create an invisible control that acts as the parent for the toolbar. The invisible control will reflect the toolbar messages back to itself. See CBandToolBarReflectorCtrl class for the invisible control that will be used. To differentiate the messages reflected from the messages received directly macros are added to the **stdafx.h.** So for example e.g WM_COMMAND reflected comes back as OCM_COMMAND.

4.5: Additional features

Dialog box

The dialog box is created to customize the menu under the journal button to the needs of the user. The menu under the journal button changes according to the choice of subject from the customize dialog box. The choice can be amongst Arts and Humanities, Business, Computers, Chemistry, Mathematics, Earth Science, Engineering, Health Sciences and Medicines, Life Science, Physics and Anatomy and Social Sciences. When one of the subjects is chosen, it assigns a value to the variable, which determines the menu that is loaded when the drop down arrow is chosen from the journal button.

Browser Navigation

In order to Navigate on the browser you need to instantiate the IWebBrowser2 COM Object. This is usually done on the SetSite Method.

Registry

The file, ToolBandObj.rgs, contains the source for the script for ATL's registry manipulation code. Most of it corresponds exactly to the registry entries that must be made so the COM run time can find the control. For the toolband control, it looks like this:

```
HKCR
{       ToolBand.ToolBandObj.1 = s 'ToolBand Sample'
        {       CLSID = s '{0E1230F8-EA50-42A9-983C-D22ABC2EED3B}'
        }
        ToolBand.ToolBandObj = s 'ToolBand Sample'
        {       CLSID = s '{0E1230F8-EA50-42A9-983C-D22ABC2EED3B}'
                CurVer = s 'ToolBand.ToolBandObj.1'
        }
        NoRemove CLSID
        {       ForceRemove {0E1230F8-EA50-42A9-983C-D22ABC2EED3B} = s
'ToolBand Sample'
                {
                        ProgID = s 'ToolBand.ToolBandObj.1'
                        VersionIndependentProgID = s 'ToolBand.ToolBandObj'
```

```
                ForceRemove 'Programmable'
                InprocServer32 = s '%MODULE%'
                {
                        val ThreadingModel = s 'Apartment'
                }
                'TypeLib' = s '{5297E905-1DFB-4A9C-9871-A4F95FD58945}'
            }       }       }
HKCU
{   NoRemove Software
  {     NoRemove Microsoft
    {       NoRemove 'Internet Explorer'
      {         NoRemove MenuExt
        {           ForceRemove '&Google' = s
'res://%MODULE%/MENUSEARCH.HTM'
          {             val Contexts = b '10'
          }   }     }       }       }       }
```

This script is used for both registering and unregistering the component. By default when registering, all of the keys are added to whatever keys might already be in the registry. The **ForceRemove** keyword modifies this behavior so that the key to which **ForceRemove** is applied is removed (including subkeys) before it's added back in.

When unregistering, the default is to remove every key (and subkey of those) that's listed in the script. It's crucial to override this for the CLSID key by using the **NoRemove**

keyword, as above. If this is not done, unregistering this component would remove the entire CLSID tree, thereby unregistering every COM object on the user's system. So one has to be *very* careful if you edit the registry script file. The wizard usually edits this files as objects and interfaces are added. The resource compiler builds this script into the components resource section, where it's available by ID (in this case, IDR_TOOLBANDOBJ).

Adding entries to the context menu

Also in the rgs file is the way to add fields to the context menu. The line

ForceRemove '&Google' = s 'res://%MODULE%/MENUSEARCH.HTM'

Creates an entry for google in the context menu. The action to be performed when google is selected from the context menu is defined in **MENUSEARCH.HTM**

One can add any number of items by adding more entries in to this file following the above procedure.

Drag and Drop

The CBandEditCtrl class is inherited from a WTL CEdit control that has drag and drop facility. i.e. one can drag text from the browser straight to the CEdit Control. The Edit control in this module allows one to drag a URL or text from Explorer or any other Dragable enabled container in to the edit box. Once the URL or text is dropped the toolband will retrieve the results from the site specified(google here) with the text chosen as the query.

The three functions:

• OnDragEnter. Called during a drag and drop when the cursor initially passes over the window associated with the COleDropTarget object. In almost all cases, this function does nothing except call OnDragOver.

• OnDragOver. Called during a drag and drop as the cursor moves over the window associated with the COleDropTarget object. This function sets the current drop effect for the cursor.

• OnDrop. Called when the user actually “drops” an object on the window associated with the COleDragTarget object. This function must fetch the data from the OLE Clipboard and paste it into the target window.

The class derived from COleDropTarget must be added to the CWnd-derived class as a member variable. The window class must be registered as a drop target via the COleDropTarget::Register function.

ToolTips

Tool tips are automatically displayed for buttons and other controls contained in a parent window derived from CFrameWnd. This is because CFrameWnd has a default handler for the TTN_GETDISPINFO notification, which handles TTN_NEEDTEXT notifications from tool tip controls associated with controls. However, this default handler is not called when the TTN_NEEDTEXT notification is sent from a tool tip control associated with a control in a window that is not a CFrameWnd, such as a control on a dialog box or a form view. Therefore, it is necessary to provide a handler function for the TTN_NEEDTEXT notification message in order to display tool tips for child

controls. This can be easily done in WTL by using the following Message Handler in the overriden ToolBar Control

NOTIFY_CODE_HANDLER(TTN_NEEDTEXT, OnToolbarNeedText);

Automatic Installation

Internet" and "download" are widely recognized terms, but there has been some confusion over what exactly is meant by the word "component" in this context.

Component

A Web-bound software distribution package usually describes the smallest installable component: a Microsoft ActiveX® Control (.ocx), a .dll, an .exe, a Java class file, or an applet.

Packaging

Code authors need to package their software components for automatic download. The packaging mechanism is usually a cabinet file addressed by the CODEBASE attribute of an OBJECT element on a Web page.

Cabinet File

The cabinet format provides an efficient way to package multiple files. The key features of the cabinet format are that multiple files may be stored in a single cabinet file (.cab) and that data compression is performed across file boundaries, significantly improving the compression ratio. Cabinet files are used to reduce file size and therefore the

associated download time for Web content. This is essentially the same technology that has been used for years to compress software distributed on disks. Microsoft supplies a cabinet resource kit containing tools a developer can use to build .cab files.Cabinet files should be digitally signed.

Signing

Code received through the Internet lacks shrink-wrapped packaging and store clerks to vouch for its reliability. Users are understandably wary when they're asked to download software from the Internet. Digital signatures associate a software vendor's name with a given file, providing accountability for software developers. A signature reassures your users by creating a path from them to you if your software harms their systems. When a user's security level is set at the default level, any object identified by the **OBJECT** tag on a Web page must be digitally signed. Internet Explorer's default security setting won't install a component that doesn't have a digital signature. One can purchase digital signatures from a certificate authority, a company that validates your identity and issues a certificate for you.

Certificate

A certificate is a set of data that identifies a software distributor. Certificates are issued (and renewed) by a third party certification authority (CA) that verifies the software distributor's identity. The CA may also provide the tools you need to digitally sign your components. Your digital certificate is included with all components you digitally sign.

INF File

An information file (.inf) provides installation instructions that Internet Explorer uses to install and register your software, as well as any files required by your software.

An INF file consists of any number of named sections, each containing one or more items. Each section has a particular purpose—for example, to copy files or add entries to the registry. Each of the items in a section contributes to the purpose of the section.

Signing and Marking ActiveX Controls

Internet Explorer has special security mechanisms for ActiveX controls. The ActiveX controls you can automatically download over the Internet can do anything—including things that would damage your system. Java attempts to solve this problem by severely limiting what a Java™ applet can do. It can't, for instance, access the client computer's file system. ActiveX controls take a different approach: they demand positive identification of the author of the control, verify that the control hasn't been modified since it was signed, and identify safe controls—kind of like shrink-wrapping a control for download over the Internet. Because of this approach, ActiveX controls can use the full power of the client's system safely.

To sign the control, one will need to obtain a certificate from a Certificate Authority such as VeriSign, Inc. at http://digitalid.verisign.com/developer/ms_pick.htm.

There are two classes of digital IDs for Microsoft Authenticode™ technology. Class 2 certificates, for individuals who publish software, cost US$20 per year and require that

you provide your name, address, e-mail address, date of birth, and Social Security Number. After VeriSign verifies this information, the certificate will be issued.

Class 3 certificates, for commercial software publishers, cost US$400 per year and require a Dun and Bradstreet rating in addition to company name, location, and contacts.

Once the certificate is obtained, the SIGNCODE program provided with the ActiveX software development kit (SDK) to sign the code. The code has to be re-signed if the code is modified. The signatures are only checked when the control is first installed—the signature is not checked every time Internet Explorer uses the control. Once the code is signed, even users whose security setting is high will be able to download, install, and register your controls.

Embedding the code in to a web page

Once the signed cabinet file has been generated it is embedded in to the webpage as below.

<html> <head> <Title> The Library Toolbar</title>

<OBJECT ID="ToolBand"

CLASSID="CLSID:0E1230F8-EA50-42A9-983C-D22ABC2EED3B"

CODEBASE="http://www.ececs.uc.edu/~srayapro/ToolBandWorking.cab">

</OBJECT>

</head></html>

The CLASSID is the GUID of the control and the CODEBASE is the path to the server where the cabinet file is saved. When a user visits the web page the Active X Control is

automatically installed on his machine. These two are embedded between the OBJECT tag.

Installation using InstallShield

The toolbar can also be installed by creating a setup program using Installshield. Installshield allows one to create a customized setup program that will allow one to transfer the DLL on to the hard drive of any user. While using this procedure one will have to run the **regsvr32** utility to actually register the DLL.

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

5.1: Conclusions

This toolbar is an effort to provide a tool which provides the user with a simple interface to search the WWW, Uclid/OhioLink, Electronic Journal Center and the ACM and IEEE databases. Below you see the ToolBar as it would appear in the explorer window.

Fig 5.1: The Library ToolBar

The toolbar consists of 5 buttons(HomePage, Web, Library, Journal, Database). The toolbar also consists of edit box, which is common to all the buttons. You only have to enter the text in the toolbar once and choose one of the options from the Web or Library or Journal or Database Button and get the corresponding results based on your choice. When you leave the edit box empty and choose one of the option it will take you to the corresponding web site where you can continue with a more specific search. Each button is associated with a particular functionality. There is a default value associated with each button. The homepage button consists of a customize option , a help option and an uninstall option.

Fig 5.2:The HomePage Button

The **Customize** option is associated with the journal button. When you click on the option, you will get the following wizard.

Fig 5.3: The Customize Option

through which you can choose which journal subject you want to search. Each choice is associated with a explanation of subjects you would be searching through when you select one of the option. To change the journal subject you want to search just select one of the subjects and click **Apply** and then **OK**. One will notice the menu under the journal button has changed accordingly. The **Help** option brings you to the current page, which details the features of the toolbar. The **Uninstall** option is used to uninstall the toolbar. Upon Uninstallation the toolbar will continue to appear in any windows already open but will not appear in any new window.

The **Web** button consists of the most popular search engines: Google, AltaVista, AllTheWeb, Yahoo, Direct Hit and Meta Crawler. It also consists of a Dictionary and a Thesaurus.

Fig 5.4: The Web Button

The default value associated with the button is google i.e., when you enter the query and click the button the results from google are returned. The **Library** button is used to

search **UCLID** and **OHIOLINK** available to the University of Cincinnati. You can search Uclid or OhioLink with respect to Author, Title, Subject and Keyword.

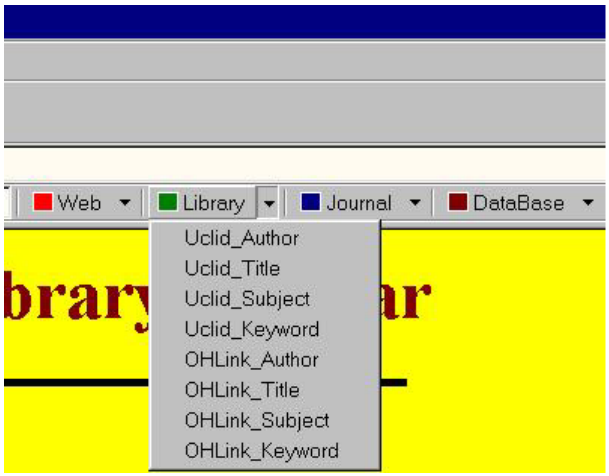

Fig 5.5: The Library Button

The journal button is based on the Electronic Journal Center available through the University of Cincinnati. The electronic journal center has different databases based on the subject. By default when you click on the journal button you search through the Computer Science journals and you will have options to search using the Author, Article Title, ISSN, Keyword, Journal Title and Abstract. The subjects available through this ToolBar are Arts and Humanities, Business, Computers, Chemistry, Mathematics, Earth Sciences, Engineering, Health Sciences and Medicine, Life Sciences, Physics and Astronomy and Social Sciences.

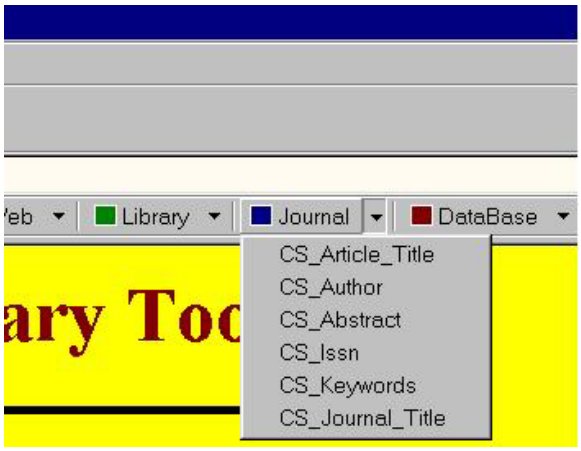

Fig 5.6: The Journal Button

The default value associated with the button is the Electronic Journal web site of University of Cincinnati. Finally the **Database** button searches through ACM and IEEE magazines.

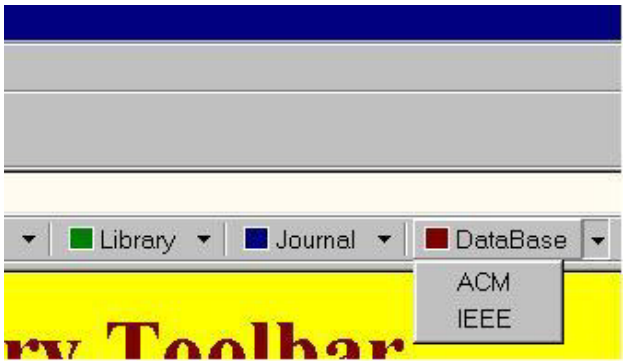

Fig 5.7: The DataBase Button

The default value of the button takes you to a list of all the databases available online through the University of Cincinnati.

5.2: Future Directions

Ideally the future extensions to the toolbar would be to design a tool, which combines the results of all the different libraries and provides one common result. The division could be on the basis of Web, Books, Journals, Databases, etc. When a user clicks on the web button, if the results provided are a combination of all the search engines rather then a specific search engine would be ideal.

The problem again in designing such a tool is that all the different information sources have different interfaces, different criteria and different search algorithms. Until a standard is decided in designing the way information is stored, the way the information is retrieved and the criteria in which they are ranked, more and more customized tools suiting the users personal choices will have to be designed. Personalization provides the control on the kind of information sources chosen, the kind of information retrieved to the user and thus providing high quality content and better interfaces to access them. The toolbar can be extended to include more functionality, adding more digital libraries available online. Also, features like query history can be added.

BIBLIOGRAPHY:

1. BrightPlanet.com LLC. " ***The Deep web: Surfacing Hidden Value.***" White Paper, July 2000. http://completeplanet.com/tutorials/DeepWeb/index.asp.
2. Valerie S. Allen, Abe Lederman, "***Searching the Deep Web- Distributed Explorit directed Query Applications***", Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, September 9-12, 2001, New Orleans, Louisiana, United States, Page-456.
3. ***Distributed Indexing/Searching Workshop***, May 28-29,1996 in Cambridge, Massachusetts, Sponsored by the World Wide Web Consortium, http://www.w3.org/Search/9605-Indexing-Workshop/.
4. Dushay, N., J. C. French, et al., "***A Characterization Study of NCSTRL Distributed Searching***," Cornell University Computer Science, Technical Report TR99-1725, January 1999, http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR99-1725.
5. Erik Selberg, Ron Daniel, Ken Weiss, Leslie Daigle, Luis Gravano, ***DISW'96 Query routing and Searching Breakout,*** http://www.w3.org/Search/9605IndexingWorkshop/ReportOutcomes/S6Group1.html**.**

5. Marti A Hearst, Luis Gravano (Ed.) "***Next Generation Web Search: Setting Our Sites",*** IEEE Data Engineering Bulletin, Special issue on Next Generation Web Search, September 2000, Page-38.
6. Monika Henzinger, Luis Gravano (Ed.), "***Link Analysis in Web Information Retrieval",*** IEEE Data Engineering Bulletin, Special issue on Next Generation Web Search, September 2000, Page 3.
8. Steve Lawrence, Luis Gravano (Ed.), "***Context in Web Search",*** IEEE Data Engineering Bulletin, Special issue on Next Generation Web Search, September 2000, Page 25.
9. Ross Tyner, M.L.S., "***Sink or Swim: Internet Search Tools & Techniques",*** February 18, 2002, http://www.ouc.bc.ca/libr/connect96/search.htm**.**
10. William Cohen, Andrew McCallum, Dallan Quass, Luis Gravano (Ed.), "***Learning to Understand the Web",*** IEEE Data Engineering Bulletin, Special issue on Next Generation Web Search, September 2000, Page 17.
11. Internet Exceeds 2 Billion Pages. July 10, 2000, http://www.cyveillance.com/us/newsroom/pressr/000710.asp.
12. Chris Sherman, "***The Future Revisited: What's New with Web Search",*** May 2000, Information Today, Inc., Online Journal, http://www.infotoday.com/online/OL2000/sherman5.html.
13. MSDN library, October 2001.
14. The Code Project Developers Website, http://www.codeproject.com/
15. The Mind Cracker Developers Website, http://www.mindcracker.com/
16. The Code Guru Developers Website, http://www.codeguru.com.

17. The Windows Developers Website, http://www.idevresource.com/

18. The Development Forum, http://www.devx.com/

19. The Microsoft Home Page, http://www.microsoft.com/

20. The COM Developers Resources, http://www.widgetware.com/

21. The MSDN Online Edition, http://msdn.microsoft.com/

22. The COM Threading Model,

    http://www.execpc.com/~gopalan/com/com_threading.html

23. The ATL Object Wizard for designing BandObjects, http://www.radbytes.com

24. Ivor Horton's "***Beginning Visual C++ 6***"

25. Chris H. Pappas & William H. Murray, III "***The Complete Reference Visual C++ 6***"