NEWS AND VIEWS ABOUT MICROSOFT FOUNDATION CLASS (MFC)

Cezar Botezatu, Ph.D., Cornelia Botezatu Paulina, Ph.D.
Romanian-American University, Bucharest, Romania

The Microsoft Foundation Class (MFC) Library is a collection of classes (generalized definitions used in object-oriented programming) that can be used in building application programs. The classes in the MFC Library are written in the C++ programming language. The MFC Library saves a programmer time by providing code that has already been written. It also provides an overall framework for developing the application program.

There are MFC Library classes for all graphical user interface elements (windows, frames, menus, tool bars, status bars, and so forth), for building interfaces to databases, for handling events such as messages from other applications, for handling keyboard and mouse input, and for creating ActiveX controls.

- The Microsoft Foundation Classes and Templates (MFC&T) provide tools to create small, lightweight controls, database applications, and full-featured, Windows desktop applications. All three libraries also provide support for COM.
- **Microsoft Foundation Class Library (MFC)**
- **Active Template Library (ATL)**
- **OLE DB Templates**
- Here are some key features regarding these libraries :
- ATL stands for *Active Template Library.* ATL is a set of template-based C++ classes designed to facilitate the process of creating (COM) objects.
- ATL has special support for key COM features, including: implementations of COM standard interfaces IUnknown, IClassFactory, IClassFactory2 and IDispatch; dual interfaces; standard COM enumerator interfaces; connection points; tear-off interfaces; and ActiveX controls.
- ATL code can be used to create single-threaded objects, apartment-model objects, free-threaded model objects, or both free-threaded and apartment-model objects.
- MFC is a C++ class library that can be used to create complex applications, ActiveX controls, and active documents.
- MFC gives us mature code that we can reuse directly; we don’t have to write, test or debug it. However no library is perfect and hardware is changing constantly, therefore MFC has to evolve too.
- MFC’s class libraries have been delivered under several versions, which we can see in our computers incarnated as .dll files (MFC40, MFC42, etc).
- If you have already created a COM object with MFC, it is recommended to continue development in MFC.
- When creating new objects, however, it might be better to use ATL if all of MFC's features are not needed.
- Template libraries constitute a type of code reuse which is different from function libraries or class libraries, because:
  1 Templates act like “C” macros on steroids: Source code is expanded directly, not only to implement actions but also to define and instantiate new classes (types)
  2 The optimizer can make the code generated more efficient.
  3 Bugs can be fixed by modifying the template source.
  4 You don't have to rebuild and distribute new copies of the libraries as long as you are willing to redistribute the source code.

Building COM Objects: Setting Up MFC Projects

With the recent noise about ATL, several developers have asked this : “Can we do that in MFC?” That is, can they create COM objects that implement custom and dual interfaces and then move to more powerful types of implementations such as aggregated objects, tear-off interfaces, and connection points?

When distributable size and instantiation speed are paramount, and you don’t need lots of built-in full-featured windowing, graphics, and database access, ATL is a far better choice than MFC. For example, an ATL-based ActiveX control can ship standalone in as little as 56k; its MFC counterpart–even statically linked via Visual C++–is 260k. Neither requires additional run-time support (and the MFC version can now be distributed free of the 900k MFC DLL), but a 56k download beats 260k anytime! Of course, ATL lacks the features, wizard support, and API wrappers of MFC. You’ll spend a lot of time writing direct calls to the Windows API in the implementation code of an ATL project. When release deadlines are tight, MFC might be the better option. It generally takes less time to fully implement complex COM objects in MFC than in ATL.

So MFC is a good tool for implementing COM objects in C++. After all, developers have hundreds of thousands of lines of legacy code that they don’t want to rewrite in ATL. Retrofitting COM onto an MFC project makes more sense, and it isn’t difficult because all CCmdTarget derivatives are capable of exposure as COM objects.

There’s only one real problem with MFC’s COM object support: The great support for implementing IDispatch via MFC and the Visual C++ ClassWizard means most developers have learned how to build automation components but know little about creating custom interface implementations or implementing dual interfaces.

COM Objects and MFC

In the strictest sense, a COM object is any software component that implements one or more COM interfaces (a COM interface is any interface de-rived from IUnknown). IUnknown provides a single, common type from which all interfaces are derived, so they can be treated uniformly. It also ensures all COM interface implementations include QueryInterface() for interface requests and navigation, and AddRef() and Release() to increment and decrement an interface reference count, letting the object delete the interface implementation from memory when it's no longer required.

MFC does some of the IUnknown work for you. It implements an aggregatable IUnknown in CCmdTarget that handles AddRef()/Release() implementation. And CCmdTarget's QueryInterface() implementation is table-driven via a set of macros similar to message-map macros: Plug in new interface IDs and addresses, and MFC's QueryInterface() looks them up at run time. CCmdTarget's IUnknown is prewired for aggregation, so you can build objects to be aggregated by other objects. Thus, the QueryInterface() can use the IUnknown of an aggregator, and the AddRef()/Release() defers to those of the outer object when yours is aggregated. Because MFC's IUnknown is wired into CCmdTarget, all derivative classes inherit the implementation via a few initialization functions and macros. (The same is true of Idispatch).

Set Up Your MFC Project for COM

You must address several issues when implementing COM objects in MFC:

- Defining the interface and the COM object in IDL
- Deriving a C++ class from the interface definition
- Implementing the interface on the C++ class
- Resolving QueryInterface() look-ups (via Interface Maps)
- The first issue concerns interface and object definitions. All COM interfaces should be defined via Interface Definition Language (IDL). Created by the Distributed Computing Environment (DCE) and adapted and extended by Microsoft for COM development, IDL is a simple, clean, platform-independent language by which you can define COM objects and interfaces in an abstract manner without respect to implementation language or tool. IDL's purpose is to create object and interface definitions while omitting implementation details. When IDL is compiled using Microsoft's IDL compiler, MIDL, it can generate an abundance of material useful to object implementers (i.e., server component developers) and object consumers (client component developers):
- A type library describing all the objects, interfaces, IDs, and types used in a particular COM object or server
- Header files that can be used by server authors to implement COM objects in C and C++
- Header files that can be used by client authors to use COM objects from clients implemented in C and C++
- Header and source files used to compile and complete a proxy-stub DLL to aid marshalling of custom data types and custom interfaces (I'll stick to type library marshalling in this example)

Traditionally, MFC-based COM projects use an IDL-derivative, Object Description Language (ODL). There was a time when the presence of ODL meant firing up two different compilers: one for IDL to generate header and source files and proxy-stub DLLs, and one to compile ODL to produce a type library. That's changed with the release of MIDL 3.0, which shipped with Windows NT 4: MIDL 3.0 treats ODL and IDL identically. Along the same lines, the ODL code would be the same if used in an IDL script. So whether you have an MFC project with ODL or an ATL project with IDL, you can still produce the header and source files, the proxy-stub components, and the type library as part of compiling the script.

The minimum ODL script for an MFC-based COM object consists of a library definition showing what goes into the type library, an interface definition defining the methods that can be used by the client, and a coclass or "COM object class" abstractly defining a COM object as a collection of specific interface implementations.

Automatizing COM Programming

- COM programming consists of defining and implementing interfaces based on *IUnknown.* Writing the code for QueryInterface, AddRef, and Release is very repetitive.
- COM objects are created by class factories, usually derived from IClasssFactory. Programming the code for CreateInstance and LockServer is also very repetitive.
- A COM component provides support for its COM Classes. It exposes the class factories and the objects' interfaces. This is a very repetitive task.
- The infrastructure necessary to support the interaction between a COM server and the COM library (the dll code) is also very repetitive.

Therefore the complexity of COM programming could be simplified if these repetitive tasks could be pre-created in class libraries.

Visual C++, MFC and BFC

The Base/1 Foundation Class Library (BFC) is an object-oriented framework for building Windows applications, especially Internet-enabled, database applications. BFC is based on the Microsoft Foundation Class Library (MFC), which is an extensive Visual C++ framework for doing general-purpose software development under Windows.

BFC is broken down into multiple components, which can be used in part or as an integrated whole. BFC has about 100 classes and 2000 functions that have been designed to be reused. Almost all BFC classes are derived from MFC classes and add significant extensions to MFC. BFC includes:

- an integrated collection of utility, database, screen and administrative classes

- state-of-the-art screen controls easily connected to the database classes
- a customizable distributed computing facility for executing long running tasks in parallel over multiple machines
- integrated support for leading reporting and graphics packages, Crystal Reports and Pinnacle Graphics

Applications built with BFC's core Database classes gain the benefit of portability between widely varying database types, currently including Oracle, IBM DB2, Sybase, SQL Anywhere, and Microsoft SQL Server and Access.

"We quickly realized that using MFC posed some challenges to a developer trying to create a complicated business system and began to search for a tool to simplify the process. We needed a tool that would allow us to quickly and simply create a system with dozens of screens and maintenance routines We found BFC and have been thrilled with the results."

John Lapenta, Director of Development, Entel Systems

BIBLIOGRAPHY

1. Richard Hale Shaw – "Building COM objects" article
2. MSDN Reference Library – www.microsoft.com
3. www.richardhaleshawgroup.com
4. www.cs.mdx.ac.uk

Надійшла до редакції 23 березня 2004р.