



*Ernst Pfannenschmidt und Paul Watzlaw*

# Die Active Template Library

Microsofts Komponentenmodell hat eine recht lange Vorgeschichte, die nicht nur durch den Wandel der Begriffe – VBX, OCX, ActiveX – sondern auch durch die Einführung neuer Techniken und Ziele gekennzeichnet ist. Während es mit OLE, VBX- und später den OCX-Controls in erster Linie um die Integration von Desktop-Anwendungen ging, stehen im Zeitalter des Inter- und Intranets vor allem verteilte Objektsysteme und kleine, leichtgewichtige Komponenten im Vordergrund. Natürlich hat in der Zwischenzeit der Wust an APIs kräftig zugenommen, und so ist man als Entwickler für alle Werkzeuge dankbar, die einem die Erstellung moderner Anwendungen erleichtern. Eine dieser segensreichen »Erfindungen« ist die Active Template Library (ATL) – eine Ansammlung von C++-Template-Klassen.

Ursprünglich wurde die ATL unter dem leicht irreführenden Namen ActiveX Template Library geführt, was eine unmittelbare Beziehung zu den ActiveX-Controls nahelegen könnte. Obwohl ActiveX und auch OLE auf dem Fundament COM (Component Object Model) aufbauen, waren die ersten ATL-Versionen nicht so sehr für die Entwicklung grafischer Oberflächenkomponenten als vielmehr kleiner, schneller COM-Objekte gedacht. COM, das grundlegende Objektmodell von Microsoft, ist in seiner Funktionsweise noch recht primitiv. Wer sich aber mit der Entwicklung binärer, in DLLs gekapselter Klassenbibliotheken beschäftigt hat, wird COM sicherlich zu schätzen wissen. Dank dieses Standards können Binärobjekte ihre Funktionalität anderen Objekten oder Applikationen zur Verfügung stellen. Compiler-spezifische Probleme beim Export von Klassen und deren Methoden in DLLs gehören damit der Vergangenheit an. Inzwischen lassen sich

mit der ATL nicht nur COM-Objekte sondern auch ActiveX-Controls recht komfortabel entwickeln. Im Prinzip ist die ATL für alle Arten von COM-, DCOM- oder OLE-Objekten geeignet. Wer allerdings in den Genuß der neuesten Library kommen möchte, muß die Version 6.0 des Visual C++-Compilers erwerben. Ältere Releases der ATL ließen sich zwar über das Internet beziehen, die letzte Ausgabe kann man aber nur zusammen mit dem neuesten Microsoft-Compiler erhalten und einsetzen.

Kennern der Microsoft Foundation Classes wird sicherlich aufgefallen sein, daß sich vieles, was mit der ATL machbar ist, auch mit der MFC realisieren läßt. Deshalb sollte man noch kurz klären, was die wichtigsten Unterschiede zwischen ATL und der MFC sind. Oberflächlich betrachtet handelt es sich bei beiden Produkten um Klassenbibliotheken, und im Fall der MFC stimmt das durchaus. Die MFC hat im Laufe der Zeit eine Reihe von Erweiterungen erfahren und ist mittlerweile zum de-facto-Standard auf Microsoft-Plattformen geworden. Die Vorteile der MFC liegen hauptsächlich in der hohen Abstraktion der Win16- und Win32-API sowie den ausgereiften Wizards, mit denen sich schnell und einfach Programme erstellen lassen. Die Vorteile der vereinfachten Entwicklung muß man sich aber mit »fetten« Applikationen und dementsprechend hohem Speicherverbrauch erkaufen. Zudem macht die Abhängigkeit von der MFC-DLL die Wartung und Installation der darauf basierenden Anwendungen nicht einfacher. Aufgrund der vielen Neuerungen und Patches kann man davon ausgehen, daß sich auf den meisten Rechnern nicht nur eine Version der MFC-DLL tummelt. Das beste Beispiel dafür ist die MFC 4.2, von der es mindestens sechs Varianten gibt – VC++ 4.2 mit zwei Patches und VC++ 5.0 mit bisher zwei Patches. Eigene Anwendungen und Komponenten lassen sich immerhin an neue Versionen anpassen. Um die Lauffähigkeit bisheriger Software zu gewährleisten, sind erneute Tests jedoch unerlässlich. Bei Fremdsoftware kann man in der Regel nichts anderes machen, als auf ein frisches Release zu warten, das mit der neuen DLL zusammenarbeitet. Durch die unterschiedlichen Libraries steigt der ohnehin hohe Speicherverbrauch von MFC-Programmen. Eine Anwendung muß halt bei Bedarf »ihre eigene« DLL ins RAM laden.

Im Unterschied zur MFC ist die Active Template Library keine typische Klassenbibliothek mit einem riesigen Klassenbaum. Die ATL wird im Quellcode geliefert und ist nicht auf eine große Runtime-Library angewiesen. Wer noch nicht so viel Erfahrung mit C++-Programmierung hat, kann Templates am ehesten mit typischeren Makros vergleichen. Ein Template stellt somit eine Art Schablone dar, die erst durch das Hinzufügen eines oder mehrerer Parameter eine instanzierbare Klasse ergibt. Anders als bei der MFC ist im Fall der ATL ein tiefgreifendes Verständnis des Objektmodells von Microsoft notwendig. Auch um umfangreiche Kenntnisse der Win32 API kommt man nicht herum. Diese ist nur »dünn«, zum großen Teil sogar gar nicht gekapselt. Gerade bei der GUI-Programmierung gibt es im Vergleich zur MFC kaum Unterstützung. Den Komfort der MFC und deren Wizards hat die ATL also (noch) nicht erreicht. Doch genau in den Bereichen, in denen die Schwächen der MFC liegen, hat die ATL ihre Stärken. Mit ihrer Hilfe lassen sich Abhängigkeiten von bestimmten Laufzeitumgebungen – etwa der

MFC und sogar der CRT (C-Runtime) – vermeiden, was die Installation und Wartung von Komponenten erleichtert. Ferner kann man kleine, sehr kompakte Objekte (ab zirka 15 Kbyte) erzeugen. Insofern ist die ATL sicherlich die beste Wahl, wenn es darum geht, die System-Ressourcen zu schonen oder ein optimales Laufzeitverhalten eines Objekts zu erreichen. Speziell für Internet-Anwendungen spielen diese Faktoren eine dominante Rolle. Wer hat schon Lust, für ein wenig grafischen Schnickschnack Megabytes an Daten durch das Modem tröpfeln zu lassen. Aber auch lokale Systeme profitieren stark von dem schonenden Umgang mit Speicherplatz und den erheblich kürzeren Ladezeiten von ATL-basierten Objekten. Zum Schluß noch eine gute Nachricht für Entwickler von Client/Server-Anwendungen. Seit der Version 1.1 ist die ATL auch Bestandteil des DCOM-SDKs (Distributed COM) für Unix-Plattformen. Es bestehen demnach recht gute Chancen, mit Hilfe der ATL Objekte Plattform-übergreifend zu entwickeln.

Wenn man also Internetanwendungen mit schlanken ActiveX-Komponenten, verteilte Objektsysteme auf Basis von COM/DCOM, unabhängige OLE Automation Objekte, Active Server-Komponenten (ASP) oder das »Free-Threading Model« von NT ausnutzen will, kommt man an der ATL einfach nicht vorbei. Besonders bei DCOM-Applikationen ist die Verfügbarkeit auf anderen Plattformen ein Pluspunkt. Für ActiveX- beziehungsweise OCX-Komponenten, die nicht im Internet eingesetzt werden sollen, muß man von Fall zu Fall abwägen, ob sich der ATL-Mehraufwand lohnt. Besonders wenn viel GUI-Programmierung zu leisten ist, kann die MFC die bessere Wahl sein. Bei der Einbindung von OLE-Features in eine bestehende MFC-Anwendung ist MFC sicherlich die bessere Wahl. Grundsätzlich besitzt die ATL die folgenden Eigenschaften:

- Kleiner, schneller Code
- Einfache, generische COM-Objekte (*vtable*-Binding)
- »Dual interface« läßt sich einfach implementieren – dies ist ein Automation Server mit *vtable*-Binding sowie *IDispatch*-Interface
- Unterstützung für Aggregation
- Unterstützung für »connection point«
- »tear-off« interfaces – das sind Schnittstellen, die erst bei Anforderung erzeugt werden
- Unterstützung des COM-Fehlermechanismus
- Unterstützung für alle COM-Threading Modelle (single, apartment, free)
- Stark optimiert
- Sehr gute Kontrolle über COM-Features
- Dank dieser Eigenschaften lassen sich mit der ATL auch folgende Arten von Objekten/Komponenten erzeugen:
- Volle ActiveX-Controls (früher OCX-Controls)

- Internet Explorer-Controls (vereinfachte ActiveX-Controls)
- Property Pages
- DCOM-Objekte als NT-Services
- Automation Server mit »dual interface«

Doch bevor es ans Eingemachte der ATL geht, kommen zum besseren Verständnis des Microsoft-Objektmodells zuerst einige COM-Grundlagen. Als Anschauungsmaterial dient dabei ein kleines Hunde-Beispiel.

## 8.1 Bello in COM – der verteilte Hund

Sieht man vorerst von speziellen Schnittstellen und höherwertigen Services wie Structured Storage ab, stellt sich COM als ein recht umgänglicher Mechanismus heraus. Ein COM-Objekt residiert in einem Server und stellt anderen Programmen beziehungsweise Clients seine Dienste zur Verfügung. Solche COM-Server lassen sich sowohl in DLLs als auch ausführbaren Dateien implementieren. Insofern können sie sich entweder in demselben Adreßraum wie der Client, in einem anderen Prozeß oder sogar auf einem anderen Rechner befinden. Dies bleibt jedoch vor dem Client verborgen.

Der transparente Zugriff auf ein COM-Objekt erfolgt über ein oder mehrere Interfaces, die das betreffende Objekt nach außen hin anbietet. Ein Interface könnte man am ehesten mit einer abstrakten C++-Basisklasse vergleichen. Solche Klassen definieren zwar Methoden, enthalten jedoch keine Funktionalität. Diese wird erst in abgeleiteten Klassen implementiert. Mit Hilfe des Interface-Konzepts trennt auch COM die Aufrufchnittstelle von der eigentlichen Implementierung. Auf diese Weise dürfen mehrere COM-Objekte dieselben Interfaces mit unterschiedlicher Funktionalität bereitstellen. Technisch gesehen ist ein Interface lediglich eine Tabelle mit Zeigern auf Funktionen und eine Interface-Instanz ein Zeiger auf solch eine Tabelle. Damit ist im Prinzip jede Programmiersprache, die Zeiger, Zeiger auf Funktionen und die Windows-API unterstützt, in der Lage, mit COM zu arbeiten.

Die einfachste Form eines COM-Objekts läßt sich in einem In-Process-Server mit *vtable*-Binding realisieren. »In-Process« bedeutet, daß das Objekt in einer DLL implementiert ist und zur Laufzeit im Speicherbereich des Clients ausgeführt wird. Da COM an sich sprachneutral ist, muß es einen Weg geben, Interfaces unabhängig von einer Programmiersprache zu definieren. Dies ist mit Hilfe der IDL (Interface Description Language) möglich. Aus der IDL-Beschreibung erzeugt anschließend der IDL-Compiler die sprachspezifischen Schnittstellen sowie eine TLB (Type Library). Diese Typbibliothek enthält genaue (Text)informationen über die Interfaces eines COM-Objekts. Skript- und Interpretersprachen, zum Bei-

spiel Visual Basic, können mittels der *IDispatch*-Schnittstelle die Informationen aus der TLB abfragen und mit deren Hilfe auf die Objektfunktionalität zugreifen. Für C++-Entwickler stellt sich die Verwendung von *IDispatch* jedoch als recht umständlich heraus, deshalb sind die meisten COM-Objekte mit einem »Dual Interface« ausgestattet. Dieses unterstützt sowohl die *IDispatch*-Schnittstelle als auch das *vtable*-Binding. Beim letzteren sind die Methoden eines Objekts über Zeiger erreichbar, für C/C++ einfach ideal. Es ist auch möglich, die IDL und ihre recht strenge Typisierung zu umgehen. Man definiert in diesem Fall die Interfaces, also die Zeigertabellen, direkt in einer Programmiersprache. In C++ läßt sich das *vtable*-Binding leicht durch abstrakte Basisklassen mit virtuellen Methoden realisieren. Allerdings verliert man mit dem *vtable*-Binding die Möglichkeit, Client und Server in unterschiedlichen Sprachen zu entwickeln. Da aber die IDL am Anfang recht verwirrend sein kann, beschränkt sich das folgende Beispiel zuerst auf die einfachste Form eines COM-Objekts. Zum besseren Verständnis der COM-Grundlagen wird das Programm zuerst nur in C++ ohne ATL präsentiert. Danach kommt »Bello« noch einmal, diesmal mit ATL-Unterstützung.

## 8.1.1 COM in Handarbeit

Das folgende COM-Objekt stellt nur ein Interface namens »IHund« zur Verfügung. Um das Programm einfach zu halten, besitzt *IHund* auch nur eine einzige Methode, nämlich »Bell«. Beim Aufruf dieser Methode wird eine Messagebox mit dem Text »Wau, wau!« erzeugt. Dies kann man bei einem In-Process-Server durchaus machen, da dieser im Adreßraum des Clients ausgeführt wird. Bei einem DCOM-Remote-Server hätte man den Text an den Client zurückgegeben und ihn erst dort angezeigt.

Unabhängig davon ob man rein in C++ arbeitet oder auf die Dienste der ATL zurückgreift, sollte man die Entwicklung eines COM-Objekts mit dem GUID-Generator beginnen. Dieses nützliche Tool befindet sich im »bin«-Verzeichnis des Visual C++-Compilers und generiert globale, systemweit eindeutige Identifizierer (Globally Unique Identifiers), sogenannte GUIDs. Wie erwähnt, erfolgt der Zugriff auf COM-Objekte transparent, das heißt, der Client weiß nicht, wo diese liegen und wie sie implementiert sind. Referenzen auf die Objekte erhält man nur mittels der GUIDs, die zusammen mit einem Objekt-Namen und dem absoluten Pfad des Objekts in der Registry abgelegt werden. Auf diese Weise stellt Windows die Verbindung zwischen Client und Server her.

Auch Interfaces besitzen eindeutige IDs, die aber nicht in der Registrierdatenbank landen müssen. Für deren Verwaltung sind die COM-Objekte selbst zuständig. Auch mehrere Objekte können dasselbe Interface anbieten. Es steht ihnen jedoch frei, wie sie dieses Interface intern implementieren. Man könnte beispielsweise ein anderes COM-Objekt mit dem Interface *IHund* erzeugen, doch beim Aufruf der Methode »Bell« würde der Text »Kläff, kläff!« erscheinen. Für das »Bello«-Beispiel hat »guidgen.exe« folgende IDs generiert:

```
// Datei Atl\Bello\Bello.h.
...
// GUID des COM-Objekts: {14F68780-E1ED-11d0-8CE9-004F4C029A9C}.
DEFINE_GUID( CLSID_Bello,
0x14f68780, 0xe1ed, 0x11d0, 0x8c, 0xe9, 0x0, 0x4f, 0x4c, 0x2, 0x9a, 0x9c);

// GUID des Interfaces: {14F68781-E1ED-11d0-8CE9-004F4C029A9C}.
DEFINE_GUID( IID_IHund,
0x14f68781, 0xe1ed, 0x11d0, 0x8c, 0xe9, 0x0, 0x4f, 0x4c, 0x2, 0x9a, 0x9c);
```

Im nächsten Schritt definiert man das Interface *IHund*. Dies ist im Prinzip nur eine abstrakte C++-Klasse, die von der *IUnknown*-Schnittstelle erbt:

```
// Datei Atl\Bello\Bello.h.
...
class IHund : public IUnknown
{
public:
    STDMETHOD(Bell)() = 0;
};
```

Um das Makro *STDMETHOD* sollte man sich keine großen Gedanken machen. Es legt die Aufrufkonvention sowie den Rückgabewert (*HRESULT*) fest und deklariert die Methode »Bell« als exportierbar. Da die Implementierung in einer abgeleiteten Klasse stattfindet, ist »Bell« zudem eine virtuelle Methode.

Genauso wie die anderen COM-Schnittstellen erbt auch *IHund* von dem *IUnknown*-Interface. Diese Schnittstelle ist sozusagen die Wurzel aller COM-Interfaces und besitzt drei fundamentale Methoden: *QueryInterface*, *AddRef* und *Release*. Somit muß auch jedes COM-Objekt diese drei Methoden implementieren, wobei in der Regel eine einzige Implementierung für alle Interfaces dieses Objekts genügt. Für jemanden, der viel mit C++ aber noch nie mit COM/OLE gearbeitet hat, ist die Methode *QueryInterface* vielleicht nicht sofort einleuchtend. Man neigt immer wieder dazu, COM-Objekte mit C++-Objekten/Klassen zu vergleichen. In C++ kann man Objekte dynamisch mittels *new* erzeugen und mit Hilfe der Pointer Methoden dieser Objekte aufrufen. Im Unterschied dazu erhält man nie einen direkten Zeiger auf ein COM-Objekt. Die Verbindung zu solch einem Objekt läuft ausschließlich über einen Interface-Pointer.

Angenommen, man hätte ein COM-Objekt namens »Tier« mit den Interfaces *IHund* und *IKatze*. Der Client bekäme – auf welche Weise auch immer – einen Zeiger auf die *IHund*-Schnittstelle von »Tier«. Da jedoch kein Pointer auf das COM-Objekt selbst besteht, könnte der Client nicht auf das *IKatze*-Interface zugreifen. Hier kommt nun die *QueryInterface*-Methode ins Spiel. Der Client kann diese Methode auf der *IHund*-Schnittstelle aufrufen und auf diese Weise erfahren, ob das Objekt auch das *IKatze*-Interface unterstützt. Wenn ja, liefert *QueryInterface* einen Pointer auf die gewünschte Schnittstelle, hier also auf *IKatze*. Dank der *QueryInterface*-Methode kann sich ein Client somit frei durch die Schnittstellen eines Objekts bewegen.

Zu klären bleibt noch, welche Aufgaben die Methoden *AddRef* und *Release* übernehmen. In einem System mit komponentenbasierter, dezentraler Architektur ist ein Verfahren notwendig, mit dem sich die »Lebensdauer« der Objekte steuern läßt. Mehrere Clients können gleichzeitig ein Objekt durch eines seiner Interfaces referenzieren. Dadurch kann kein Client mit Sicherheit entscheiden, ob ein Objekt noch benötigt wird oder gelöscht werden darf. Die Common Object Request Broker Architecture (CORBA) – eine Architektur für verteilte objektorientierte Systeme – spezifiziert zum Beispiel einen »Lifecycle«-Service, der automatisch für die Freigabe von nicht mehr benutzten Objekten sorgen soll. Microsoft hat für COM/DCOM keinen Mechanismus dieser Art vorgesehen. Insofern müssen Entwickler von COM-Servern als auch -Clients gewisse Regeln beachten, damit es nicht zu Speicherplatzverlusten kommt.

Für jedes seiner Interfaces sollte ein COM-Objekt einen 32-Bit unsigned Integer als Referenzzähler verwalten. Bei jedem Aufruf von *AddRef* wird der Zähler der betreffenden Schnittstelle erhöht, bei einem Aufruf von *Release* wieder dekrementiert. Wenn alle Zähler den Wert Null erreicht haben, wird das Objekt nicht mehr referenziert und kann gelöscht werden. Damit hängt es ganz von den Clients ab, ob der Speicher korrekt verwaltet wird. Von einigen Ausnahmen abgesehen gilt die Faustregel, daß auf jeder neuen Kopie eines Interface-Pointers *AddRef* aufgerufen werden muß. Wird diese Kopie nicht mehr benötigt, muß ein Aufruf von *Release* erfolgen. Ein gutes Beispiel hierfür ist die lokale Kopie eines globalen Interface-Pointers innerhalb einer Funktion. Da eventuell andere Funktionen die Kopie in der globalen Variablen löschen könnten, ist es zwingend notwendig, den Referenzzähler mittels *AddRef* zu erhöhen. In zwei Fällen kann man sich aber Aufrufe von *AddRef* beziehungsweise *Release* sparen.

Wenn von einem bestehenden Pointer eine neue Kopie erzeugt und diese mit Sicherheit vor dem ursprünglichen Pointer zerstört wird, können Aufrufe von *AddRef* und *Release* auf der zweiten Kopie entfallen. Dies trifft beispielsweise für die Übergabe eines Interface-Pointers an eine Funktion zu. Die »Lebensdauer« des Pointers verwaltet der aufrufende Code, so daß sich die aufgerufene Funktion nicht mehr um das Reference Counting kümmern muß. Der andere Fall tritt ein, wenn von einem Pointer eine weitere Kopie erzeugt und der erste Pointer vor der zweiten Kopie freigegeben wird. Dann darf man *AddRef* auf der zweiten und *Release* auf der ersten Pointer-Kopie weglassen. Als Beispiel mögen *Out*-Parameter sowie Return-Werte von Funktionen dienen. Innerhalb einer Funktion wird ein Interface-Pointer erzeugt und an den Aufrufer zurückgegeben, der für die Freigabe mittels *Release* verantwortlich ist. Ein zusätzliches *AddRef* im aufrufenden Code ist nicht nötig, da schon die aufgerufene Funktion für eine »stabile« Pointer-Kopie sorgt.

Mit dem Wissen um die Funktion der *IUnknown*-Methoden ist die Implementierung der *I Hund*-Schnittstelle nicht sonderlich schwierig. Man leitet einfach eine neue Klasse *CHund* von der Interface-Klasse ab und implementiert darin neben den Methoden von *IUnknown* auch die Methode *Bell*:

```
// Datei Atl\Bello\BelloImp.h.
...
class CHund : public IHund
{
public:
    CHund();

    STDMETHODCALLTYPE Bell();

    STDMETHODCALLTYPE QueryInterface( REFIID riid, void** ppObject);
    STDMETHODCALLTYPE AddRef();
    STDMETHODCALLTYPE Release();

protected:
    ULONG m_dwRefCount;
};
```

Außerdem besitzt *CHund* noch die Membervariable *m\_dwRefCount*, die die Zahl der Referenzen auf das *IHund*-Interface dieses COM-Objekts speichert. Da beim Erzeugen des Objekts noch kein Interface referenziert wird, muß man *m\_dwRefCount* im Konstruktor von *CHund* mit Null initialisieren:

```
// Datei Atl\Bello\Hund.cpp.
...
CHund::CHund()
{
    m_dwRefCount = 0;
}
```

Als nächstes steht die Implementierung von *QueryInterface* an. Diese Methode erwartet zwei Parameter, nämlich die GUID des gewünschten Interfaces sowie einen Zeiger auf einen Zeiger. Unterstützt ein COM-Objekt das betreffende Interface, wird dessen Zeiger in *ppObject* abgelegt und der Wert *NO\_ERROR* an den Aufrufer zurückgegeben. Andernfalls liefert der Aufruf von *QueryInterface* den Wert *E\_NOINTERFACE*. Das COM-Objekt »Bello« bietet nur die *IHund*-Schnittstelle, die sich bei Bedarf auch als *IUnknown*-Interface verwenden läßt:

```
STDMETHODIMP CHund::QueryInterface( REFIID riid, void** ppObject)
{
    if ( riid == IID_IUnknown || riid == IID_IHund)
        *ppObject = (IHund *) this;
    else
        return E_NOINTERFACE;
    AddRef();

    return NO_ERROR;
}
```

Jeder Aufruf von *QueryInterface* bedeutet eine weitere Referenz auf eine Schnittstelle und damit auf das COM-Objekt. Folglich muß *QueryInterface* automatisch einen Referenzzähler erhöhen. In diesem Fall kommen zwei Referenzzähler zum

Einsatz – ein globaler für alle Interfaces aller COM-Objekte sowie ein lokaler für jede Schnittstelle. Die Variable `g_dwRefCount` für den globalen Zähler wird in der Datei »BelloFactory.cpp« definiert und initialisiert:

```
STDMETHODIMP_(ULONG) CHund::AddRef()
{
    g_dwRefCount++;
    m_dwRefCount++;

    return m_dwRefCount;
}
```

Sowohl der lokale als auch der globale Referenzzähler müssen bei einem Aufruf von *Release* wieder dekrementiert werden. Da dieses COM-Objekt nur ein Interface unterstützt, speichert `m_dwRefCount` nicht nur die Zahl der Referenzen auf die *IHund*-Schnittstelle, sondern grundsätzlich auf das COM-Objekt. Sobald `m_dwRefCount` den Wert Null erreicht hat, bestehen keine Referenzen, und die Instanz kann getrost aus dem Speicher gelöscht werden:

```
STDMETHODIMP_(ULONG) CHund::Release()
{
    g_dwRefCount--;
    m_dwRefCount--;
    if ( m_dwRefCount == 0)
    {
        delete this;

        return 0;
    }
    return m_dwRefCount;
}
```

Übrigens sollte man sich auf die Rückgabewerte von *AddRef* sowie *Release* laut Microsoft nicht verlassen, was vielleicht auch nur ein begründetes Mißtrauen zu der Implementierung dieser Methoden in manchen COM-Objekten ist. Wie man aber im nächsten Abschnitt sehen wird, gehören dank der ATL Fehler bei *AddRef* und *Release* der Vergangenheit an.

Die Implementierung von *IHund* ist fast fertig, es fehlt jetzt nur noch die eigentliche Funktionalität, also die Methode *Bell*. Diese macht aber nichts weiter, als eine Messagebox auf dem Bildschirm mit dem Text »Wau, wau!« auszugeben:

```
STDMETHODIMP CHund::Bell()
{
    MessageBox( NULL, "Wau, wau!", "IHund->Bell", MB_OK);

    return S_OK;
}
```

Die Klasse *CHund* ist damit vollständig. Es stellt sich jetzt die Frage, auf welche Weise Clients Instanzen dieser Klasse, also mehrere COM-Objekte gleichen Typs, erzeugen können. Gerade am Anfang kann man leicht den Eindruck gewinnen,

daß sich jeweils nur ein Objekt einer COM-Klasse instanzieren läßt. Dies wäre allerdings, gerade im Vergleich zu CORBA ein riesiger Schwachpunkt, und so gestattet auch COM beliebig viele Instanzen einer Klasse. Der Operator *new* läßt sich aber in einem System, dessen Komponenten in verschiedenen Adreßräumen oder sogar auf unterschiedlichen Rechnern liegen dürfen, nicht einfach so verwenden. COM greift deshalb auf ein bewährtes Verfahren namens »Objekt-Fabrik« (Factory) zurück. Eine Factory ist ein Objekt, das für die Erzeugung anderer Objekte zuständig ist. Hierfür stellt COM ein Standard-Interface, die Schnittstelle *IClassFactory* zur Verfügung, die ebenfalls von *IUnknown* abgeleitet ist und zusätzlich die Methoden *CreateInstance* und *LockServer* besitzt. Jeder COM-Server muß die *IClassFactory*-Schnittstelle implementieren, so auch »Bello«:

```
// Datei Atl\Bello\BelloImp.h
...
class CBelloFactory : public IClassFactory
{
public:
    CBelloFactory();

    STDMETHODCALLTYPE QueryInterface(REFIID riid, void** ppObject);
    STDMETHODCALLTYPE AddRef();
    STDMETHODCALLTYPE Release();

    STDMETHODCALLTYPE CreateInstance(IUnknown *pUnkOuter,
        REFIID riid, void** ppObject);
    STDMETHODCALLTYPE LockServer(BOOL fLock);

protected:
    ULONG m_dwRefCount;
};
```

Die Implementierung der *IUnknown*-Methoden verläuft genauso wie bei der *IHund*-Schnittstelle. Deshalb besitzt auch die *CBelloFactory*-Klasse einen internen Zähler, der im Konstruktor mit Null vorinitialisiert wird. Der einzige Unterschied liegt darin, daß *QueryInterface* anstelle von *IHund* die Schnittstelle *IClassFactory* unterstützt. Daß *CBelloFactory* keinen Zeiger auf das *IHund*-Interface zurückgeben kann, ist verständlich. Bis auf die Tatsache, daß ein *CBelloFactory*-Objekt Instanzen von *CHund* erzeugt, haben beide Klassen nichts gemeinsam – also auch kein Interface.

Die Erzeugung neuer Objekte erfolgt in der Methode *CreateInstance*. Wie erwähnt, erhält man nie einen direkten Zeiger auf ein COM-Objekt, sondern referenziert dieses ausschließlich über einen Interface-Pointer. Deshalb erwartet *CreateInstance* als zweiten und dritten Parameter die ID der gewünschten Schnittstelle sowie einen Zeiger, der auf den Interface-Pointer verweisen soll:

```
// Datei Atl\Bello\BelloFactory.cpp.
...
STDMETHODIMP CBelloFactory::CreateInstance(IUnknown *pUnkOuter,
      REFIID riid, void** ppObject)
{
  if (pUnkOuter!=NULL)
    return CLASS_E_NOAGGREGATION;

  CHund* pHund = new CHund;
  if ( FAILED( pHund->QueryInterface( riid, ppObject)))
  {
    delete pHund;
    *ppObject = NULL;

    return E_NOINTERFACE;
  }
  return NO_ERROR;
}
```

Der Sinn des ersten Parameters *pUnkOuter* erschließt sich im Zusammenhang mit dem Wiederverwendungskonzept von COM. Wenn in verteilten System die Rede von Vererbung ist, so ist in der Regel die Vererbung von Schnittstellen, nicht aber die der Implementierung gemeint. Mit Aggregation bietet COM dennoch einen Mechanismus, um bereits vorhandene Implementierungen nutzen zu können. Dabei aggregiert ein COM-Objekt ein anderes Objekt und stellt dessen Schnittstellen nach außen als eigene Interfaces dar. Deshalb müssen auch die *IUnknown*-Implementierungen des inneren und äußeren Objekts koordiniert werden. Falls ein Client auf irgendeiner Schnittstelle des inneren Objekts *QueryInterface*, *AddRef* oder *Release* aufruft, werden diese Aufrufe an das *IUnknown*-Interface des äußeren Objekts weitergeleitet. Die einzige Ausnahme bilden Aufrufe dieser Methoden auf der *IUnknown*-Schnittstelle des inneren Objekts. Nur in diesem Fall wird ein innerer Referenzzähler benutzt.

Somit wird in *pUnkOuter* der *IUnknown*-Pointer des äußeren Objekts übertragen, falls das zu erzeugende Objekt aggregiert werden soll. Andernfalls ist *pUnkOuter* NULL. Da *CHund*-Objekte nicht aggregierbar sind, prüft *CBelloFactory::CreateInstance*, ob ein NULL-Pointer übergeben wurde. Wenn ja, wird ein neues *CHund*-Objekt instanziiert und mittels *QueryInterface* befragt, ob es das gewünschte Interface unterstützt. Bei einer anderen Interface-ID als *IID\_IHund* scheitert der Aufruf von *QueryInterface* und das *CHund*-Objekt wird wieder gelöscht.

*LockServer*, die zweite Methode von *IClassFactory*, muß man nicht unbedingt implementieren. Wenn es allerdings darauf ankommt, Objekte schnell zu erzeugen, ist dies durchaus empfehlenswert. COM-Server, die in Form von DLLs realisiert sind, sollten die Funktion *DLLCanUnloadNow* exportieren. Diese Funktion wird von Zeit zur Zeit aufgerufen, um zu prüfen, ob ein COM-Server (DLL) wieder aus dem Speicher entfernt werden darf. Dies passiert, wenn eine DLL keine Objekte mehr verwaltet, daß heißt, der Referenzzähler auf Null ist:

```
// Datei Atl\Bello\BelloFactory.cpp.
...
STDMETHODIMP DllCanUnloadNow()
{
    if ( g_dwRefCount)
        return S_FALSE;
    else
        return S_OK;
}
```

Um einen Server im Speicher zu behalten, reicht es somit aus, den globalen Referenzzähler *g\_dwRefCount* zu erhöhen. Selbst wenn alle Objekte dieses Servers freigegeben wurden, bleibt er aktiv. Man sollte jedoch nicht vergessen, für jeden Aufruf von *LockServer* mit dem Wert TRUE zum Schluß auch einen Aufruf mit FALSE durchzuführen:

```
STDMETHODIMP CBelloFactory::LockServer( BOOL fLock)
{
    if ( fLock)
        g_dwRefCount++;
    else
        g_dwRefCount--;

    return NO_ERROR;
}
```

Die letzte wichtige Funktion von »Bello« ist *DllGetClassObject*, die zum Erzeugen von Klassenobjekten dient. Da ein COM-Server mehrere COM-Klassen und damit Klassenobjekte unterstützen kann, prüft man zuerst die übergebene Klassen-ID:

```
STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid,
                        void** ppObject)
{
    if ( rclsid == CLSID_Bello)
    {
        CBelloFactory *pFactory= new CBelloFactory;

        if ( FAILED( pFactory->QueryInterface( riid, ppObject)))
        {
            delete pFactory;
            *ppObject=NULL;

            return E_INVALIDARG;
        }
    }
    else
        return CLASS_E_CLASSNOTAVAILABLE;
    return NO_ERROR;
}
```

Im obigen Fall können nur Klassenobjekte für *CLSID\_Bello* erzeugt werden. Bei anderen Klassen-IDs gibt *DllGetClassObject* den Fehlerwert *CLASS\_E\_CLASSNOT-*

*AVAILABLE* zurück. Der Rest des Codes ähnelt der Implementierung der *QueryInterface*-Methode. Es wird zuerst ein *CBelloFactory*-Objekt erzeugt und gefragt, ob es das übergebene Interface implementiert. Dies könnte *IClassFactory*, *IClassFactory2* oder ein davon abgeleitetes Interface sein, wobei *CBelloFactory* nur die erste Schnittstelle anbietet. Ist alles in Ordnung, bekommt man einen Interface-Pointer. Wie bei allen anderen COM-Objekten läßt sich auch ein Klassenobjekt nur auf diese Weise referenzieren.

Die Methoden *DLLRegisterServer* und *DLLUnregisterServer* kann man optional implementieren. Sie dienen dazu, einen COM-Server mittels des Programms »regsvr32.exe« zu registrieren oder wieder aus der Registrierungsdatenbank zu entfernen. Die Eintragung in die Registry beziehungsweise die Löschung erfolgt mit den folgenden Kommandos:

```
regsvr32 bello.dll // Server registrieren.
regsvr32 /u bello.dll // Server löschen.
```

Die Registrierung läßt sich auch direkt aus der Visual-C++-6.0-IDE durchführen.

Die Implementierung des COM-Servers ist damit abgeschlossen. Zum Testen der Server-Funktionalität gibt es auch ein kleines Client-Programm. Dieses arbeitet ausschließlich mit Instanzen der *IHund*-Schnittstelle, es braucht aber keine Ahnung von deren interner Implementierung zu haben. Deshalb benötigt der Client lediglich die Beschreibung der *IHund*-Schnittstelle und eine *Main*-Routine:

```
// Datei Atl\Bello\BelloTest\BelloUser.cpp.
...
#include "..\Bello.h"

int main(int argc, char **argv)
{
    IHund *pHund;

    CoInitialize(NULL);

    if (FAILED(CoCreateInstance(CLSID_Bello, NULL, CLSCTX_SERVER,
        IID_IHund, (LPVOID*) &pHund)))
    {
        puts("Kann COM-Objekt nicht erzeugen!");
        return -1;
    }

    pHund->Bell();
    pHund->Release();

    CoUninitialize();

    return 0;
}
```