JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

COM+ in der Prozeßautomatisierung

Diplomarbeit zur Erlangung des akademischen Grades
Diplomingenieur

in der Studienrichtung
Informatik

Angefertigt am
Institut für Informationsverarbeitung und Mikroprozessortechnik

Betreuung:
o. Univ.-Prof. Dr. Jörg R. Mühlbacher

Von:
Christoph Emsenhuber

Mitbetreuung:
Oberrat Dipl.-Ing. Rudolf Hörmanseder

Linz, Oktober 2002

Danksagung

Mein Dank gilt zuallererst meinen Eltern, die mich die gesamte Studienzeit hindurch finanziell, ideell und familiär großzügig unterstützt haben. Durch sie war es mir möglich, mich frei von existentiellen Sorgen dem Studium zu widmen.

Meinem Lehrer, Herrn o. Univ.-Prof. Dr. Jörg R. Mühlbacher möchte ich besonders für die Themenstellung für die Diplomarbeit danken, aber auch dafür, daß er mir immer wieder mit ergänzenden Aufträgen zur Seite stand.

Herrn Oberrat Dipl.-Ing. Rudolf Hörmanseder darf ich meinen Dank aussprechen, daß er reges Interesse an meiner Arbeit nahm, mich mit Hinweisen und Ratschlägen unterstützte und damit zu einem vortrefflichen Begleiter für mich und meine Arbeit wurde.

Der VA Technologie AG, vor allem den Herren Dipl.-Ing. Johann Hörl und Dipl.-Ing. Anton Brandstötter verdanke ich hilfreiche Anregungen zur Themenstellung, vor allem aber war mir die Beistellung der betriebseigenen Fachliteratur eine große Hilfe.

Die Fachgespräche mit meinen Studienkollegen lösten in mir öfters wertvolle Impulse aus, weswegen ich mich auch diesen gegenüber zu aufrichtigem Dank verpflichtet weiß.

Nicht zuletzt bedanke ich mich bei allen Professoren, Assistenten und Lehrern für die Mühen und Unterweisungen, die sie mir während meines Studiums widmeten, da diese es mir erst ermöglichten, auf einem guten Fundament die Diplomarbeit zu erstellen und zu beenden.

Zusammenfassung

Programme und Programmteile miteinander interagieren zu lassen ist in der modernen Softwareentwicklung nicht mehr wegzudenken. Man denke an legendäre Begriffe wie RPC, Zwischenablage oder OLE. Mindestens genauso wichtig ist jedoch COM, das Component Object Model von Microsoft. Computerübergreifende Kommunikation sowie die Wiederverwendung und Versionierung von Software sind gute Gründe, den Einsatz von COM/DCOM zu überlegen.

Durch die Einführung von Windows 2000 brachte Microsoft eine neue Technologie namens COM+ auf den Markt.

Diese Arbeit befaßt sich damit, inwieweit das Konzept von COM+ für klassische Aufgaben im Bereich der Software-Entwicklung geeignet ist und welche Aspekte grundsätzlich bei der Verwendung überlegt werden sollen. Sie zeigt anhand von Beispielen Ansätze zur Realisierung von typischen Aufgabenstellungen auf sowie etwaige Probleme, die bei der Realisierung zu erwarten sind.

Abstract

Software development has become nearly impossible without the interaction between programs and pieces of programs. Examples are RPC, clipboard and OLE. But there is also COM, Microsoft's component object model. Distributed communication, code reuse and versioning are good reasons to use COM/DCOM.

With Windows 2000, Microsoft introduced a new technology, called COM+.

This thesis focuses on the question, to what extent COM+ can be used for software engineering and it shows that some aspects should be considered carefully.

The examples explain different ways to realize typical tasks and point out possible problems that can occur during the realization.

Inhaltsverzeichnis

1 COM-Grundlagen

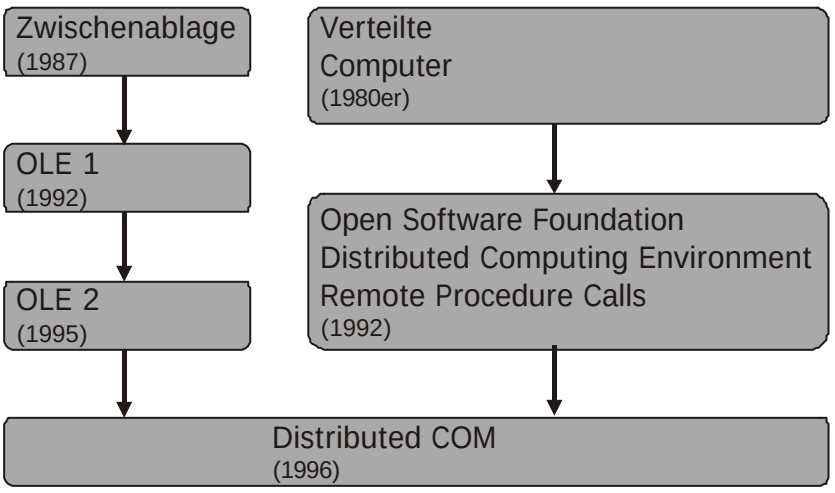
Dieser Abschnitt widmet sich den Grundlagen rund um COM. Dieses Wissen ist notwendig, um COM+ zu verstehen und damit arbeiten zu können.

Um große, komplexe Aufgaben zu realisieren, bedarf es ausgeklügelter organisatorischer Strukturen. Probleme ab einer gewissen Größe bedürfen einer strukturellen Aufteilung. Diese besteht darin, Probleme zu zerlegen und dafür Teillösungen zu suchen, welche dann zu einer Gesamtlösung zusammengefügt werden.

Die Idee, Probleme zu zerlegen ist gleichermaßen für die Softwareentwicklung anzuwenden. COM ist ein Hilfsmittel, um komplexe Aufgabenstellungen in der Softwareentwicklung in mehrere Ebenen zu zerlegen und anschließend eine Gesamtlösung zu erstellen.

1.1 Entwicklung von DCOM

Wie so oft hat Microsoft mit Distributed COM das Rad nicht innerhalb eines Tages erfunden. COM/DCOM ist vielmehr das Ergebnis zweier Entwicklungslinien, die früher verfolgt wurden.



Man muß vorausschicken, daß MS-DOS ursprünglich darauf ausgelegt war, daß nur eine Anwendung auf einem Computer laufen konnte. Erst mit Entwicklung leistungsfähigerer Prozessoren (i80286 und i80386) begann Microsoft an das gleichzeitige Laufen mehrerer Anwendungen auf einem Computer zu denken. Mit der Einführung von Windows gab es erstmals eine Multitasking-Umgebung, die bei den Anwendern den Wunsch nach Datenaus-

tausch zwischen den unterschiedlichsten Anwendungen aufkommen ließ. Diesem Wunsch wurde mit der Einführung der Zwischenablage und DDE (Dynamic Data Exchange) Rechnung getragen. DDE war jedoch für Software-Entwickler zu kompliziert, sodaß es nur sehr wenige Anwendungen gibt, die DDE erfolgreich implementieren.

Im Gegensatz zu DDE ist die Zwischenablage wesentlich einfacher für die Software-Entwickler aufgebaut. Aufgrund des Copy-Paste-Prinzip sind der Zwischenablage großer Erfolg und Akzeptanz bis heute beschert. Ein Problem konnte die Zwischenablage jedoch nicht lösen: wie verknüpft man beispielsweise ein Winword-Dokument mit einem Excel-Spreadsheet? Die Zwischenablage ist zu unflexibel, um Dokumente und deren Inhalte miteinander verknüpfen zu können. Sobald in einem eingebetteten Dokument eine Änderung notwendig wird, muß dieses bearbeitet, in die Zwischenablage kopiert und letztendlich in das Zieldokument eingefügt werden. Microsofts Antwort auf diese Problemstellung heißt OLE (Object linking and embedding).

OLE kam erstmals mit Windows 3.1 auf den Markt. OLE 1 sollte eine geeignete Schnittstelle bieten, um mit verknüpften Dokumenten besser umgehen zu können. In ein Winword-Dokument ein Excel-Spreadsheet einzubetten war schon seit DDE kein Kunststück mehr. OLE 1 ermöglichte es jedoch, in einem Excel-Spreadsheet Daten zu ändern und sorgte dafür, daß diese Änderungen auch in jene Dokumente übernommen wurden, in die das betreffende Excel-Spreadsheet eingebettet war. Das eingefügte Dokument konnte sogar durch Doppelklick geöffnet werden. OLE 1 basierte trotz neuem Namen in Wirklichkeit wieder auf DDE, welches das Protokoll für die Kommunikation zwischen den einzelnen Prozessen war. Die für damalige Zeit revolutionären Ansätze von OLE 1 wurden mit der Zeit weiter verfeinert, bis man erkannte, daß eingebettete Dokumente (Objekte) kleine Software-Komponenten sind, die in jede beliebige Anwendung eingebaut werden können und die Funktionalität der betreffenden Anwendung erweitern.

OLE 2 hat die Verknüpfungs- und Einbettungs-Funktionen von OLE 1 im Jahr 1993 durch Vorort-Aktivierung erweitert. Es war von nun an möglich durch Doppelklick auf ein Objekt dieses im gleichen Fenster zu bearbeiten, in dem sich das Objekt befindet. Was auf den ersten Blick hierbei nicht so auffiel war die Tatsache, daß sich OLE mehr und mehr zurückzog und statt dessen COM in den Vordergrund trat. 1996 war der Punkt erreicht, an dem COM mit der Einführung von Microsoft Windows NT 4.0 auch lernte, in Netzwerken sich mit mehreren Computern zu unterhalten; DCOM war geboren.

Wie wurde aber eine einfache Kommunikation im Netzwerk ermöglicht? In den frühen 80ern war es noch eine große Herausforderung, mehrere Computer über ein LAN zu vernetzen. Oftmals mußten Räder neu erfunden werden, sie ermöglichten jedoch kein ordentliches Zusammenspiel der Rechner im Netz. Das änderte sich einige Jahre später wie die Open Software Foundation (OSF) mit dem Ziel ins Leben gerufen wurde, Standards für die Vernetzung von Computern zu definieren. Remote Procedure Calls (RPCs) sind wohl das bekannteste Werkzeug in der Windows-Welt, um verteilte Anwendungen miteinander kommunizieren zu lassen.

Infolge dessen wurde der Grundstein für Distributed Computing Environment (DCE) gelegt. DCE beinhaltet eine Sammlung von Werkzeugen und Diensten, die die Erstellung von verteilten Anwendungen erleichtern. Nachdem sich RPCs bewährt haben, war es naheliegend, COM um den Kommunikationsmechanismus RPC zu erweitern und DCOM entstehen zu lassen.

1.2 Wozu Komponentenorientierung?

Programmierer haben die Eigenschaft, eine Sammlung von Routinen bei der Programmierung in vielen Programmen wiederzuverwenden. In einigen Routinen werden komplizierte Rechenoperationen ausgeführt, andere greifen auf Windows-Funktionen zurück. Da diese Bibliotheken im Allgemeinen in mehreren Programmen Verwendung finden, ist das Copy-Paste-Prinzip der verwendeten Bibliotheken in das jeweilige Programm der Weg des geringsten Widerstands. So schön und einfach diese Vorgehensweise nun klingen mag, so unschön wird sie mit steigender Anzahl der Verwendung.

Änderungen in Bibliotheken sind auf Dauer unvermeidbar, seien es Fehlerkorrekturen, Laufzeitoptimierung oder Erweiterung der Funktionalität. Die Folge einer derartigen Änderung ist das Suchen aller Anwendungen, die unter Verwendung der betroffenen Bibliothek erstellt wurden. Ein erneutes Kompilieren der Anwendungen ist unvermeidlich, um die Änderungen auch an die Applikationen weiterzugeben. Auch die Verwendung einer gemeinsamen Bibliothek erlöst den Programmierer nicht von unserem Problem.

Die Lösung dazu liegt im Erstellen sogenannter Komponenten, die den gesamten Funktionsumfang im Sinne obiger Bibliothek implementieren. Sie unterscheiden sich nur insofern von den besprochenen Bibliotheken, als daß sie eine Möglichkeit bieten, in kompilierter Form ver-

wendet zu werden und auch nach einer Modifikation mit bestimmten Auflagen ohne neuerlichem Erstellen der davon abhängigen Applikationen weiter verwendet werden können.

1.3 Universell Programmieren

Es ist keine einfache Aufgabe, Software zu entwickeln, die ein problemloses Zusammenspiel von Komponenten zuläßt, die in unterschiedlichen Entwicklungsumgebungen, aber auch in unterschiedlichen Programmiersprachen geschrieben wurde. Wer sich schon länger mit Programmiersprachen beschäftigt, wird wissen, daß jede Sprache seine eigene Syntax hat. Unterschiedliche Befehle für Funktionsaufrufe, Exception-Handling sowie die Adressierung von Variablen erschweren es Softwareentwicklern, mehrere Sprachen zusammenarbeiten zu lassen.

Sicherlich gibt es Mittel und Wege, diese Aufgabenstellung ohne COM oder mit COM vergleichbaren Programmierkonzepten (z.B. CORBA) zu lösen. Eigene Methoden zu entwickeln, die universelles Programmieren ermöglichen, sind jedoch sehr aufwendig und daher als ineffizient zu betrachten.

1.3.1 Unterschiedliche Programmiersprachen

Wie schaut jedoch ein Lösungsansatz für die Programmiersprachen-Problematik unter COM aus? Unter Berücksichtigung unterschiedlicher Methodenaufrufe in den einzelnen Sprachen ist erkennbar, daß die Syntaxen im wesentlichen einander sehr ähnlich sind. Gemeinsam ist ihnen die Übergabe von Parametern und gleichen, aber zumindest ähnlichen Typen zur Deklaration derselben. Unter COM werden die unterschiedlichen Syntaxen auf eine eigene Sprache zusammengefaßt. Ziel dieser gemeinsamen Sprache ist die Schnittstellendefinition einer COM-Komponente, die ein genormtes Aussehen nach außen hat und die Methoden einer Komponente gemeinsam mit ihren Parametern für alle COM-Sprachen in Binärform exakt festlegt. COM-Komponenten kommunizieren mittels Interfaces, die als *arrays of function pointers* definiert sind.

Dank ihrer Interfaces können COM-Komponenten mit jeder beliebigen COM-Entwicklungsumgebung erstellt werden. Vertreter von COM-Entwicklungsumgebungen sind beispielweise Microsoft Visual Basic, Microsoft Visual J++ und Microsoft Visual C++ (Active Template Library, kurz ATL). Es wären hier auch noch andere Sprachen als Vertreter für COM-Sprachen zu nennen, wenngleich man pauschal folgendes sagen kann: Jede Sprache, die die

Schnittstellendefinitionssprache von COM zur Erstellung eines binär kompatiblen Interfaces versteht, ist eine COM-konforme Sprache. Das trifft auf Visual FoxPro genauso zu wie auf Active Server Pages, die beim Aufruf einer Seite interpretiert werden. Wesentlich ist, daß die binäre Schnittstelle einer Komponente unveränderlich ist.

Letztendlich haben COM-kompatible Sprachen eine unterschiedliche Terminologie, können aber jede beliebige Komponente einer anderen Programmiersprache mit einer etwaigen Einschränkung der Mächtigkeit der Sprache in ihrer Funktionalität und Schnittstellengestaltung auf Binärebene in ihrer Funktionalität gleich nachbilden. Weiters können Komponenten auch unabhängig von der Programmiersprache wiederverwendet werden. Reduziert man Komponenten auf ihre Schnittstellendefinition und auf ihr Verhalten, unterscheiden sie sich nur durch ihre Bezeichnung in Abhängigkeit von der Programmiersprache, unter der sie erstellt wurden; Visual Basic bezeichnet seine Komponenten als ActiveX-DLLs, Visual J++ als COM-DLLs und Visual C++ ATL erzeugt sogenannte DLL-Server. Abgesehen von unterschiedlich lautendenden Namen ist aber stets vom gleichen Produkt – von COM-Komponenten – die Rede. [Pla99]

Wie gut das Zusammenspiel von Komponenten unterschiedlicher Programmiersprachen funktioniert, zeigt das Beispiel eines Softwareentwicklers: Dieser hatte ein auf COM basierendes Produkt namens Data Access Add-In (DAA) im Einsatz, das Klassen unter Visual Basic Zugriff auf eine relationale Datenbank gab. Während der Softwareentwicklung mit DAA war er noch sehr zufrieden. Nach Auslieferung der fertigen Anwendung an den Kunden mußte er jedoch feststellen, daß die Anwendung wesentlich langsamer arbeitete als vermutet und beabsichtigt gewesen war. Auf Nachfrage stellte sich heraus, daß DAA in Java geschrieben wurde. Er hätte DAA wohl nie verwendet, wenn er vor Beginn seiner Arbeit gewußt hätte, daß DAA in Java implementiert wurde. Dieses Beispiel zeigt nur zu gut, daß COM einem auch ausgezeichnete Dienste leistet, wenn es die Implementierung und deren Sprache einer Komponente verstecken soll. [Gor00]

1.3.2 Softwarebus

Softwareentwicklung und Softwareverwendung könnten wesentlich erleichtert werden, wenn Komponenten immer gleich wiederverwendet werden könnten – unabhängig davon, ob es sich dabei um eine Dynamic Link Library (DLL) oder ein Executable handelt. Es wäre für anspruchsvolle Software auch wünschenswert, wenn es bei der Entwicklung nicht mehr relevant ist, ob auf dem selben oder einem entfernten Rechner eine Komponente installiert

wird. Warum soll man für Softwareentwicklung nicht ein Äquivalent zu den Vorteilen eines Hardwarebus suchen, nur nicht für Hard- sondern für Software?

Der Hardwarebus eines Computers hat viele Vorzüge. Es ist dabei hervorzuheben, daß sich selbst Komponenten mit Computern und deren Busse verstehen, die erst nach den Computern entwickelt wurden. Sofern sich die zu installierenden Komponenten an einen gewissen Standard halten (PCI, ISA, SCSI...), lassen sich beliebige Komponenten in einen Hardwarebus einbauen.

Warum soll es Busse jedoch nur für Hard- und nicht auch für Software geben? Die Vorteile eines Hard- bzw. eines Softwarebus werden aus ihrer einzigen konzeptionellen Gemeinsamkeit erzielt, nämlich einem wohldefinierten Standard, an den sich alle Komponenten halten.

Abbildung 1 zeigt eine mögliche Architektur eines Softwarebus. Es ist hierbei hervorzuheben, daß dieser Softwarebus nicht auf einen einzigen Rechner limitiert ist. Sowohl die Architektur des Softwarebus COM als auch die von CORBA haben den Vorteil, daß Applikationen in einem Netzwerk verteilt laufen können. Hinter dieser Architektur steht die Überlegung, an einem beliebigen Rechner im Netzwerk eine neue Komponente zu installieren und alle übrigen Rechner können auf diese unter Einhaltung vorgegebener Sicherheitsrichtlinien zugreifen.

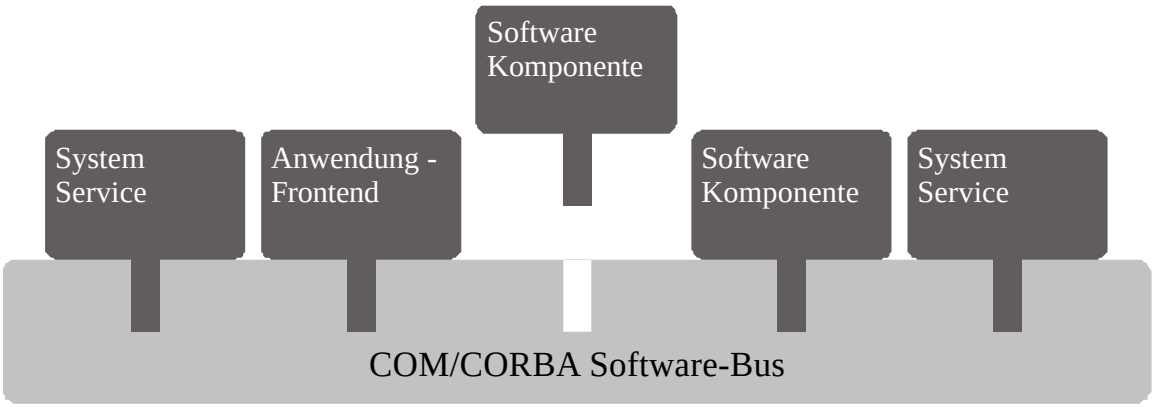


Abbildung 1: Architektur eines Softwarebus

Grundlage dafür ist, daß Frameworks eine besondere Architektur haben, um das Zusammenspiel unterschiedlicher Programmiersprachen zu ermöglichen. COM ist ähnlich wie CORBA als Softwarebus modelliert.

1.4 Aufgabenstellung

Um auch einen praktischen Nutzen aus dieser Arbeit ziehen zu können, werden die theoretischen Erläuterungen von COM durch praktische Bespiele ergänzt. Daraus soll ein Anwendungsbeispiel entwickelt werden, das Problemlösung für folgende Aufgabenstellung sein soll:

Durch die Einführung von COM+ mit seiner gegenüber MTS 2.0 und COM/DCOM erweiterten Funktionalität sind neue Möglichkeiten zu evaluieren, Kommunikation in verteilten System zu unterstützen. Im wesentlichen wird sich die Arbeit auf drei Säulen (COM/DCOM, IIS/ASP und MSMQ) stützen, die durch die Clients 1 bis 3 in Abbildung 2 dargestellt werden.

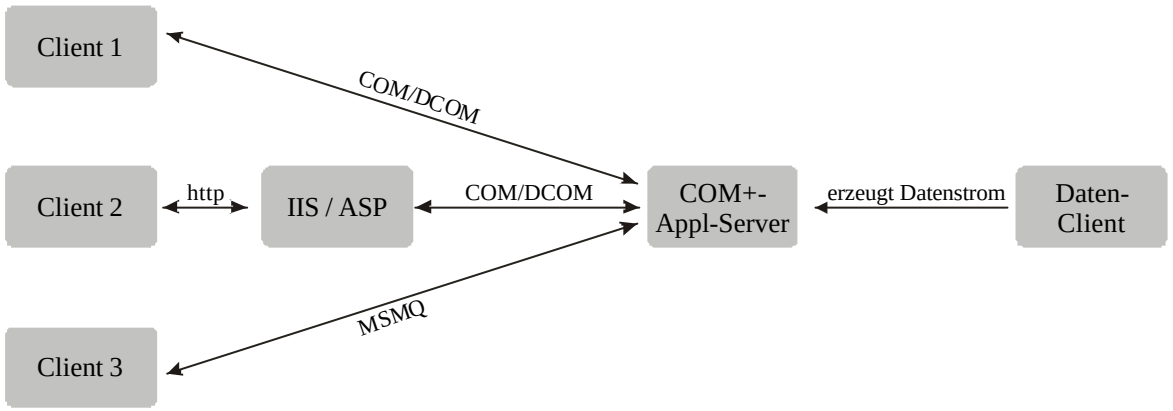


Abbildung 2: Skizzierung der Aufgabenstellung

Daten-Clients schicken einen beliebigen Datenstrom an einen COM+-Server, der die Daten wiederum an die Clients links verteilen soll. Für die Verteilung stehen unterschiedliche Wege zu Verfügung, die im folgenden an kurzen anschaulichen Beispielen demonstriert werden sollen. In weiterer Folge wird das Lösungsmodell sukzessiv verfeinert, um ein breites Spektrum von COM+ zu betrachten. Performance in großen Systemen – eventuell auch Server-Farmen – werden hier einen nicht unwesentlichen Einfluß auf die Qualität der zu entwickelnden Anwendungen haben.

2 COM/DCOM Einführung

2.1 Die vier Säulen von COM

Nun ist es keineswegs einfach, in wenigen Sätzen eine komplette Definition von COM geben zu können. Am ehesten lassen sich die Funktionalität und Mächtigkeit von COM anhand seiner vier Entwicklungsideen erklären:

- **Wiederverwendbarkeit**

  Erstens sollten die einzelnen Komponenten wiederverwendbar sein, da Wartung sowie die Erweiterung von Komponenten in großen Anwendungen große Probleme mit sich bringen. Ein Aufteilen in wiederverwendbare Komponenten ist naheliegend, die die Erstellung neuer Applikationen, Erweiterung sowie Wartung erleichtern. Der Vorteil von COM liegt darin, daß es Klassen-Code in binären Komponenten verteilt. Das heißt, COM-Komponenten können ohne Abhängigkeiten vom Source-Code wiederverwendet werden. Im gleichen Zug müssen Software-Entwickler nicht mehr Algorithmen oder Teile ihres Quell-Codes preisgeben, wenn sie Komponenten an eine Entwicklergemeinde verteilen wollen. Daß durch die Wiederverwendung binärer COM-Komponenten Probleme vermieden werden können, die zur Compile-Zeit auftreten, ist ein positiver Seiteneffekt, der nicht zu vernachlässigen ist.

- **Interface-orientierte Programmierung**

  Um Mißverständnissen vorzubeugen, sei an dieser Stelle darauf verwiesen, daß Microsoft und ein Großteil der Fachliteratur die Unterstützung objekt-orientierter Programmierung als eine weitere Säule verstehen. Tatsächlich handelt es sich dabei jedoch um interface-orientierte Programmierung, da das Konzept der Objektvererbung nicht unterstützt wird. Unabhängig davon, mit welchen Anwendungen man sich unter Windows beschäftigt, Programmierer haben mitunter auch unwissentlich mit COM-Komponenten zu tun. Komponenten sind fertige Module oder Einzelteile, die von beliebigen Programmen wiederverwendet werden können. Man denke nur an Datenbankanbindung von Applikationen, Automatisierung von Office-Applikationen oder die Wiederverwendung von Teilen des Internet-Explorers. Diese Einzelteile als Objekte in Applikationen einzubauen, vereinfacht die Erstellung großer Programme

nicht unwesentlich. Letztendlich ist OOP, in diesem Fall jedoch zumindest interface-orientierte Programmierung doch eine Grundlage für einfachere Wartung der Quell-codes.

☞ **Unabhängigkeit von der verwendeten Programmiersprache**
Nicht minder wichtig ist bei COM die Unabhängigkeit von der Programmiersprache, in der eine fertige Komponente erstellt wurde. Da jede Programmiersprache in Abhängigkeit von der Aufgabestellung teiloptimale Lösungen anbietet, ist es naheliegend, die Vorteile der jeweiligen Sprachen auszuschöpfen und Teillösungen über gekapselte, wiederverwendbare, binäre Komponenten zu verknüpfen.

☞ **Client/Server-Architektur**
Schließlich soll COM aber auch Client/Server-Architekturen mit Ortsunabhängigkeit unterstützen. Ab einer gewissen Komplexität ist es nicht mehr möglich, alle geforderten Funktionen einer großen Applikation auf einem Rechner ablaufen zu lassen. Ein Aufteilen von einer Aufgabe in mehrere kleine Teilaufgaben auf mehreren Rechnern ist naheliegend und bei größeren Projekten auch schon zum Zeitpunkt der Programmentwicklung durchaus praktikabel.

Ergebnis dieser Forderungen ist Microsofts *Component Object Model*, kurz COM, um das man als zeitgemäßer Programmierer in der Windows-Welt nicht mehr herumkommt.

2.2 Interfaces

COM unterscheidet sich von anderen Komponententechnologien dadurch, daß es mit unbekannten Elementen umgehen kann. Wird eine COM-Komponente von einem Programm instanziiert, muß das Programm Information darüber bekommen können, welche Schnittstellen in der COM-Komponente implementiert sind. Die Methode *QueryInterface* von *IUnknown* ist die Antwort auf diese Problemstellung, nur was ist IUnknown und wie kommt eine Komponente dazu?

Es gilt, daß jede COM-Klasse IUnknown implementieren und jede COM-Schnittstelle davon erben muß. Die Schnittstelle von IUnknown definiert sich wie folgt[1]:

[1] Folgende Definition entspricht der Sprache IDL, mehr dazu in Abschnitt 2.3

```
interface IUnknown {
  HRESULT QueryInterface([in] REFIID riid,
          [out, iid_is(riid)] void **ppv);
  ULONG AddRef(void);
  ULONG Release(void);
};
```

QueryInterface wird automatisch aufgerufen, wenn eine COM-Klasse instanziiert wird. `riid` übergibt den GUID der angeforderten Schnittstelle. Existiert für den betreffenden GUID eine Implementierung, gibt diese über den Zeiger `ppv` einen vTable-Pointer zurück.

Ein vTable (engl. virtual method table) ist ein Array von Pointern. Jeder Pointer zeigt auf eine bestimmte Methode oder Property im betreffenden Objekt.

Die Methoden in einem vTable enthalten neben der Adresse noch andere Zusatzinformationen wie den Namen der Methode, den Return-Type sowie die Datentypen für die einzelnen Parameter.

COM vertraut darauf, daß vTable und die IDL-Schnittstelle übereinstimmen. Programmierer können davon ausgehen, daß bei Visual Basic, Visual C++ und Visual J++ das auch der Fall ist. Andernfalls würde es zu einem Kompilierungsfehler kommen.

`AddRef` und `Release` sind die zwei noch verbleibenden Methoden in IUnknown. Diese übernehmen die Verwaltung der Verweiszählung.

Erstellt ein Prozeß eine Instanz von einer COM-Klasse, wird ein Verweiszähler mit `AddRef` erhöht. Analog dazu wird der Verweiszähler einer COM-Klasse mit `Release` erniedrigt, wenn eine Instanz nicht mehr benötigt wird. Hat die Verweiszählung eines COM-Objekts den Wert Null erreicht, wird das COM-Objekt automatisch zerstört.

Wenn ein Interface von IUnknown erbt, wird dieses als benutzerdefiniertes Interface bezeichnet. Diese haben die Eigenschaft der *frühen Bindung* gemeinsam. Das heißt, daß der vTable eines Interface schon zur Compile-Zeit bekannt ist. Frühe Bindung basiert auf einem Vertrag, der durch die Typbibliothek vorgegeben ist.

Es ist zu beachten, daß Komponenten bei Verwendung von früher Bindung schneller sind, da schon zur Compile-Zeit bekannt ist, wo die Implementierung der verwendeten Methoden ist. Der Vorteil schnellerer Anwendungen bei früher Bindung bringt den Nachteil mit sich, daß die Implementierung den Vertrag nicht ändern darf ohne die davon abhängigen Anwendungen davon in Kenntnis zu setzen. Eine Änderung des Vertrages würde automatisch das erneute

Erstellen aller auf dem Vertrag aufbauenden Programme und/oder Komponenten erforderlich machen. Andernfalls führen derartige Änderungen im vTable zu Anwendungsfehlern.

Kommt anstatt früher Bindung *späte Bindung* (auch OLE-Automatisierung genannt) zum Einsatz, muß zur Compile-Zeit keine Typbibliothek angelegt werden; diese darf zur Laufzeit variieren. Späte Bindung kennt die IDispatch-Schnittstelle, der eine ganz besondere Bedeutung zukommt: Wird mit Hilfe von IDispatch eine Methode oder Property aufgerufen, ermittelt IDispatch, welche Methode gemeint ist. IDispatch ist ASP als Bindeglied zwischen COM-Komponenten und ASP beispielsweise für Aufrufe unerläßlich.

2.3 Interface Definition Language (IDL)

Wie schon erwähnt, kommunizieren COM-Applikationen miteinander und mit dem System durch eine als Interface spezifizierte Menge von Methoden. Ein Interface ist bei COM ein streng definierter Vertrag zwischen Software und Client, der eine Menge an Operationen zur Verfügung stellt.

Es wurde eingangs in Abschnitt 1.3 erwähnt, daß COM und CORBA konzeptionell gewisse Ähnlichkeiten haben. CORBA kennt ebenfalls die Interface Definition Language IDL. Das einzige, was COM IDL und CORBA IDL gemeinsam haben sind ihr Name und ihre Aufgabe. In der Syntax sind sie jedoch vollkommen unterschiedlich! [Gor00]

2.3.1 Typen

Um in IDL den Vertrag in Form einer Schnittstellendefinition zu schreiben sind zur Spezifizierung der Methoden Basistypen für Variablen unverzichtbar. Alle COM-Interfaces müssen in IDL definiert werden. IDL gestattet, verhältnismäßig komplexe Datentypen zur Beschreibung einer Schnittstelle zu verwenden.

Tabelle 1 listet die am häufigsten verwendeten IDL-Basistypen für die COM-Sprachen C++, Visual Basic und Java auf:

Sprache	IDL	Microsoft C++	Visual Basic	Microsoft Java
Basistypen	boolean	unsigned char	nicht vorhanden	char
	byte	unsigned char	nicht vorhanden	char
	small	char	nicht vorhanden	char

	short	short	Integer	short
	long	long	Long	int
	hyper	__int64	nicht vorhanden	long
	float	float	Single	float
	double	double	Double	double
	char	unsigned char	nicht vorhanden	char
	wchar_t	wchar_t	Integer	short
	enum	enum	Enum	int
	Interface Pointer	Interface Pointer	Interface Ref.	Interface Ref.
Erweiterte Typen	VARIANT	VARIANT	Variant	ms.com.Variant
	BSTR	BSTR	String	java.lang.String
	VARIANT_BOOL	short [-1/10]	Boolean [True/False]	boolean [true/false]

Tabelle 1: IDL-Basistypen in C++, Visual Basic und Java

Dem Typ BSTR (Bytestring) kommt eine ganz besondere Bedeutung zu, zumal es sich um einen durchaus häufig verwendeten Typ handelt. C/C++ verwendet ein null-terminiertes Array of Char und Visual Basic verwaltet seine Strings als Array of Char mit Längenangabe des Strings. Eine BSTR-Variable ist ein Pointer, der auf ein wide character array zeugt. Die Anzahl der Buchstaben im Array ist unmittelbar vor dem Array im Speicher abgelegt. Dieses Konzept hat zwei Vorteile: Erstens kann die Länge des Arrays schnell bestimmt werden, was mehrere NULL als Zeichen im Array zuläßt. Zweitens kann ein BSTR wesentlich einfacher zwischen unterschiedlichen Prozessen ausgetauscht / übertragen werden als ein null-terminierter String.

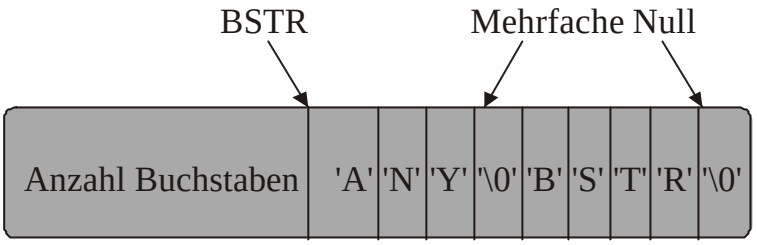


Abbildung 3: Aufbau eines Byte Strings

Zwecks besserem Verständnis der folgenden Abschnitte soll an dieser Stelle der Begriff des GUID erklärt werden. Ein GUID (engl. Globally Unique Identifier) ist ein 128-Bit langer Integer, der Datentypen wie Interfaces und Co-Klassen[2] eindeutig identifizieren soll. Ein GUID eignet sich zur Identifikation wesentlich besser als ein Klassenname, da Klassennamen durch Menschen und nicht durch Zufallszahlen vergeben werden und die Wahrscheinlichkeit geringer ist, daß für unterschiedliche Komponenten gleiche Namen existieren. Im übrigen ist die Anzahl von ca. $3.4 * 10^{83}$ Zahlenmöglichkeiten nicht zu vernachlässigen. Da gängige Compiler mit 128 Bit langen Zahlen jedoch nichts anfangen können, werden diese meist in einem anderen Datenformat gespeichert. Bei C++ und Visual Basic werden die 128 Bit in kleinere Integer zerlegt. Im hexadezimalen Zahlenformat reduziert sich der GUID auf 32 Zeichen. Dieses Zahlenformat ist auch als Registry-Format bekannt, da GUIDs in der Windows Registry auch genauso abgespeichert werden.

```
{F3CF9127-79BE-4506-8AA8-5C0213C5C489}
```

ist ein klassisches Beispiel für einen GUID.

Um etwaigen Problemen infolge der nächsten Abschnitte vorzubeugen, empfiehlt [Pat00] die aktuellste Version von MIDL zu verwenden. Im Allgemeinen hat man als Programmierer keine Probleme mit älteren MIDL-Releases zu arbeiten, jedoch unterstützen ältere Versionen eventuell nicht alle der aktuellen Features der Interface Definition Language.

2.3.2 Visual Basic

Ohne eingangs viel Erfahrung mit der Syntax der Interface Definition Language zu haben, soll in folgendem Beispiel veranschaulicht werden, wie trivial es ist ein IDL-File zu erzeugen bzw. dessen Syntax zu lesen. IDLs lassen sich bei der Programmierung unter Visual Basic sehr einfach erstellen (d.h. sie werden automatisch generiert). Abschnitt 2.7 wird noch eine

[2] COM-Klasse

genauere Beschreibung zur Erstellung von COM-Komponenten bringen, folgendes Beispiel soll jedoch das Konzept von IDL demonstrieren:

Nachdem ein Projekt vom Projekttyp *ActiveX-DLL* angelegt wurde, müssen die einzelnen Methoden spezifiziert werden. Eine Implementierung ist hierbei nicht zwingend erforderlich. Folgendes Beispiel der Klasse *HelloWorld* implementiert zwei Methoden:

```
Sub SayHello()
    MsgBox ("Hello world!")
End Sub

Sub SayAnything(message As String)
    MsgBox (message)
End Sub
```

`SayHello` ist ein einfacher Aufruf einer Methode ohne Parameter, `SayAnything` will bewußt jedoch einen String als Argument übergeben bekommen.

Nach erstmaligem Erstellen der DLL-Datei des betreffenden Projektes existiert für die Entwicklungsumgebung eine definierte Schnittstelle, die zwar nicht separat als IDL-Datei gespeichert wird, jedoch in der DLL vorhanden ist.

Um die Schnittstellendefinition sichtbar zu machen muß das zu Visual Studio 6.0 gehörende Dienstprogramm *OLE-Ansicht* (oleview.exe) gestartet werden. Der Befehl *View TypeLib* öffnet ein Dialogfenster, in dem die neu erstellte COM-Komponente *HelloWorld.dll* ausgewählt werden kann. Ergebnis dieses Beispiels ist folgende Interface-Definition, die einiger Erklärung bedarf.

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: HelloWorld.dll

[
  uuid(F3CF9127-79BE-4506-8AA8-5C0213C5C489),
  version(1.0)
]
library Projekt1
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-
000000000046}
    importlib("STDOLE2.TLB");

    // Forward declare all types defined in this typelib
```

```
    interface _Class1;

    [
      odl,
      uuid(5ADF5EEA-ABAC-48DC-BF1B-23405388FA76),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
    ]
    interface _Class1 : IDispatch {
        [id(0x60030000)] HRESULT SayHello();
        [id(0x60030001)] HRESULT SayAnything([in, out] BSTR* message);
    };

    [
      uuid(241EC540-0FBC-4FB4-8629-A2A8108C83B8),
      version(1.0)
    ]
    coclass Class1 {
        [default] interface _Class1;
    };
};
```

IDL zeichnet sich durch eine bestimmte Strukturierung aus. Eine COM-Komponente ist ein Paket, das über das `library` gekennzeichnet wird und in obigem Fall den Namen `Projekt1` hat. Dieses Paket kann wiederum mehrere COM-Klassen beinhalten. Obiges Beispiel beschränkt sich ausschließlich auf die Co-Klasse `Class1`.

Sowohl Pakete als auch Co-Klassen werden durch einen GUID identifiziert; bei Co-Klassen spricht man hier von einem Class ID (CLSID). Das geschieht durch das `uuid`(universally unique identifier)-Attribut.

Ein Library-Block kann im Gegensatz zu dem Beispiel auch andere Attribute enthalten. Der Vollständigkeit halber sollen hiermit alle Attribute aufgezählt werden:

- ☞ uuid: ID zur Identifikation der Bibliothek. Dieser Parameter ist zwingend erforderlich.
- ☞ version: Gibt die Versionsnummer der Bibliothek an.
- ☞ helpstring: Gibt in Textform zusätzliche Information zur Bibliothek und kann auch in Objektbrowsern abgelesen werden.

- ☞ lcid: Gibt die Sprache der Bibliothek an.
- ☞ hidden: Versteckt das Objekt vor einem COM-Objektbrowser.

`coclass Class1` definiert die Implementierung einer COM-Komponente. Die Definition einer Co-Klasse hat folgende Syntax:

```
[attributes]
coclass classname {
  [interface attributes] [interface | dispinterface] interfacename;
};
```

Folgende Attribute können für eine Co-Klasse definiert werden:

- ☞ uuid: Gibt die Class ID (CLSID) zur Identifizierung eines Objekts an.
- ☞ version: Gibt die Versionsnummer der Bibliothek an.
- ☞ helpstring: Gibt in Textform zusätzliche Information zur Klasse und kann auch in Objektbrowsern abgelesen werden.
- ☞ licensed: Veranlaßt die Klasse, das betreffende Objekt zu prüfen.
- ☞ hidden: Versteckt das Objekt vor einem COM-Objektbrowser.

Weiters gibt es noch folgende Attribute innerhalb der `coclass`-Definition:

- ☞ source: Gibt an, daß das Interface Events (Ereignisse) unterstützt.
- ☞ default: Legt fest (bei VBScript), welches Interface standardmäßig aufgerufen wird, wenn keines angegeben ist.
- ☞ restricted: Verhindert die Verwendung eines Interfaces durch Makroprogrammierer.

Interessant erscheint hier noch, daß in jeder IDL das Statement `importlib("STDOLE2.TLB")` vorkommt. STDOLE2.TLB muß immer importiert werden, da dieses die Schnittstellen von IUnknown und IDispatch definiert.

Der Block `interface _Class1 : IDispatch` besteht aus der Definition der Methoden `SayHello` und `SayAnything`. Methoden in COM-Komponenten haben grundsätzlich immer den return type `HRESULT`. `HRESULT` gibt an, ob ein Methodenaufruf unter COM erfolgreich aufgerufen wurde oder nicht. Wie läßt sich aber `HRESULT` mit einem beliebigen return type einer Funktion vereinbaren? Die Funktion

```
Function GetRandomNumber() As Double
```

wird in IDL folgendermaßen umschrieben:

```
HRESULT GetRandomNumber([out, retval] double*);
```

Die Richtungsattribute `[in], [out], [in, out]` geben die Richtung der Parameter an. Standardwert ist `[in]`, d.h. daß Daten vom Konsumenten nur zur Implementierung geschickt werden. `[out]` steht dementsprechend für einen Datenfluß in die entgegengesetzte Richtung. Die Kombination `[in, out]` gibt an, daß Daten von einem Konsumenten zur Komponente fließen, in der Implementierung gelöscht und neu geschrieben werden, um letztendlich wieder an den Konsumenten zurückgeschickt zu werden.

Man sieht, daß Visual Basic auf Umwegen IDL-Dateien erzeugen kann. Wenn es aber darum geht, fremde Komponenten zu verwenden, die nicht unter Visual Basic erzeugt wurden, kann das für den Software-Entwickler unangenehm werden. Um dennoch auf externe Komponenten zugreifen zu können, bietet Visual Basic einen Visual Basic-to-COM mapping layer. Wenn man bedenkt, daß in den vTables (vgl. frühe Bindung) die Methoden mit ihren Einsprungpunkten gespeichert sind, ist es kein Problem auf eine Komponente auch ohne IDL zugreifen zu können. [`Gor00` , `Gro00`]

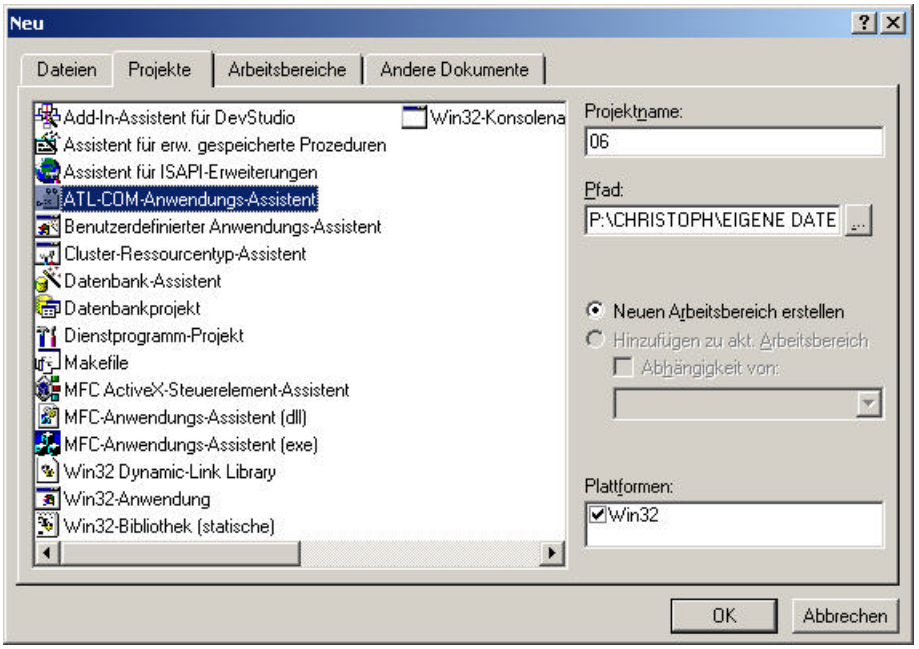
2.3.3 Visual C++ / ATL

Komponentenorientierte Programmierung unter Visual Basic bringt einige Einschränkungen mit sich. So hat der Benutzer keine Möglichkeit, die im Interface verwendeten GUIDs selbst zu bestimmen. Visual Basic hat automatisch GUIDs für die neue Komponente vergeben. COM stellt eine Funktion namens `CoCreateGUID` zur Verfügung, die neue GUIDs erzeugt. Diese Funktion baut auf dem Wissen auf, daß die MAC-Adresse einer Netzwerkkarte[3] in Verbindung mit der Systemuhrzeit eine Zufallszahl garantieren muß, die weltweit eindeutig ist. Wer einen GUID manuell außerhalb von Visual Basic erzeugen möchte, verwendet dazu das Programm *guidgen.exe*, das Bestandteil von Visual Studio ist.

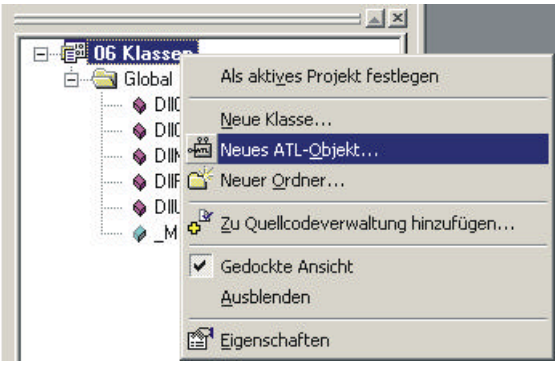
Zwar etwas gewöhnungsbedürftig, aber wesentlich eleganter gestaltet sich die Schnittstellen-definition unter Visual C++. Es gibt viele Wege, die nach Rom und zu einer IDL führen. Unter Zuhilfenahme der ATL (Active Template Library) wandelt sich ein auf den ersten Blick kompliziertes Unterfangen zu einer einfachen Sache.

Zunächst wird ein neues Projekt (ATL-COM-Anwendungs-Assistent) angelegt:

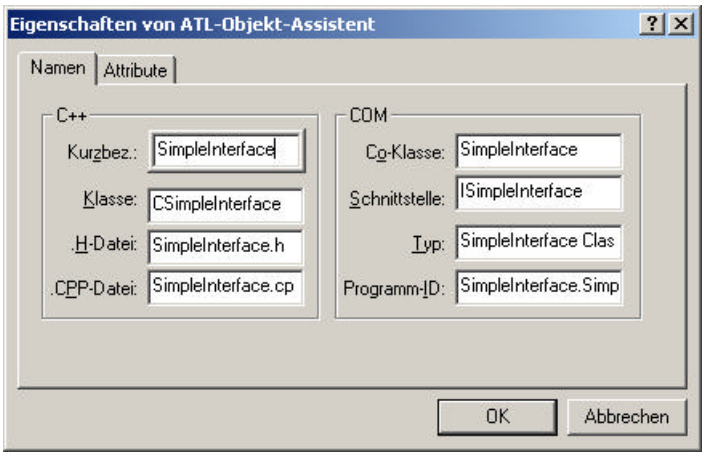
[3] Sollte keine Netzwerkkarte vorhanden sein, werden andere Informationen über den Computer und seine Hardware zur Berechnung des GUIDs verwendet.



Als *Server-Typ* wird *Dynamic Link Library (DLL)* gewählt und anschließend erzeugt der Assistent eine neue Projektumgebung, die schon die wichtigsten Elemente für COM-Komponenten enthält.

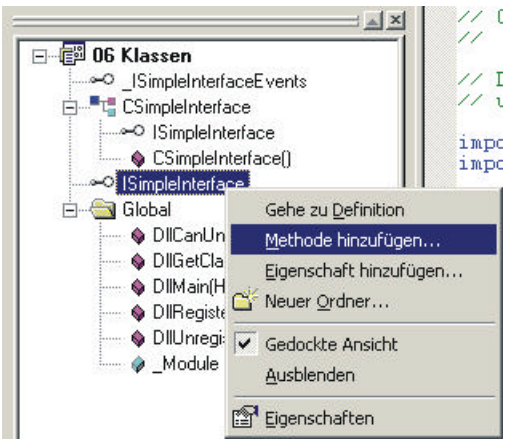


Anschließend wird dem Projekt ein neues ATL-Objekt hinzugefügt, das dann die Basis für die IDL werden soll. Im folgenden Dialog wird festgelegt, wie die neue Klasse (und daher auch das Interface) heißen sollen.



Es genügt hierbei, nur die Kurzbezeichnung (Komponentenname) auszufüllen, da der ATL-Objekt-Assistent die übrigen Namen (für Dateien, Klasse etc.) automatisch ergänzt. Der Assistent erstellt COM-Interface und Co-Klasse in einem Arbeitsschritt, weshalb eben noch mehr Namen notwendig sind. Der Schnittstelle wird ein "I" vorangestellt, welches eine COM-Schnittstelle kennzeichnen soll. Die Programm-ID setzt sich aus dem Komponentennamen und der Co-Klasse zusammen. Das *Threading-Modell* aus dem Dialog Einstellungen unter *Attribute* bleibt *Apartment* und auch die übrigen Einstellungen können vorerst akzeptiert werden.

Im Fenster Arbeitsbereich ist jetzt die neue Klasse in der Klassenansicht sichtbar.



Über das Kontextmenü lassen sich nun Methoden und Properties hinzufügen. Hierbei wird nicht nur automatisch das IDL-File von Visual C++ bearbeitet, sondern zugleich die Header-Datei und das betreffende Klassenmodul. Fazit: Wer auf IDL nicht verzichten will, der soll sich zwecks Einfachheit der ATL bedienen.

2.4 Versionsverwaltung

Komponenten-Versionierung zählt zu einer der Hauptstärken von COM. Um größtmögliche Flexibilität bei der Softwareentwicklung und -wartung zu erreichen, bedient man sich der Versionierung.

Komponentenorientierte Programmierung basiert auf der Überlegung, Komponenten mit gleicher Schnittstelle aber unterschiedlicher Versionsnummer und meist auch unterschiedlicher Implementierung zu unterstützen. Immerhin hat das Konzept der binären Komponenten den Vorteil, daß einzelne Komponenten einer großen Anwendung ausgetauscht oder erweitert werden können, ohne daß davon abhängige Teile beeinflußt werden. Meistens zumindest! Was aber, wenn bei der Installation eines Programms eine bestehende

Komponente von einer Komponente älteren Datums überschrieben wird? Das Problem dabei ist meist in Komponenten zu suchen, die von mehreren unterschiedlichen Programmen verwendet werden. Die MFC Runtime DLL mfc42.dll ist ein Klassiker für weit verbreitete DLLs; diese wird nämlich von jedem Programm verwendet, das mit Visual C++ geschrieben wurde.

COM wurde mit dem Hintergedanken entwickelt, daß einmal definierte Schnittstellen (Interfaces) von Komponenten nicht mehr verändert werden. Das heißt, daß Methoden in einem bestehenden Interface nicht mehr hinzugefügt, verändert oder entfernt werden dürfen. Will man dennoch eine bestehende Schnittstelle erweitern oder deren Implementierung verändern, muß ein neues Interface definiert werden.

Alte Schnittstellendefinitionen in neuen Versionen nicht zu verändern, erlaubt es alten wie neuen Clients, die neue Komponente zu verwenden. Weiters können neuere Clients alte Komponenten fragen, ob diese einen bestimmten Interface identifier (IID) und mit ihm bestimmte Operationen unterstützen. Zu diesem Zweck sei die Methode `QueryInterface` in `IUnknown` wärmstens empfohlen. Mittels `QueryInterface` kann ein neuer Client sein Verhalten an das Vorhandensein einer älteren Komponente anpassen, um gegebenenfalls nur einen eingeschränkten Funktionsumfang zu unterstützen. Letztendlich sollen dadurch ungültige Funktionsaufrufe als Auslöser für Laufzeitfehler rechtzeitig abgefangen werden können.

Wie auch immer man als Programmierer zu Interfaces von Komponenten stehen mag, zwei wesentliche Punkte soll man nie aus dem Auge verlieren: Sobald ein COM-Interface veröffentlicht wurde, soll es nicht mehr geändert werden und wenn wirklich einmal der Bedarf einer Änderung gegeben ist, so soll das Interface so definiert sein, daß die Kompatibilität zu älteren Clients gesichert bleibt; es dürfen also bestehende Schnittstellen nicht modifiziert oder enterbt werden.

Nehmen wir also an, es gibt von einer Komponente zwei Versionen, *v1* und *v2*, wobei *v2* mehr Methoden als *v1* unterstützt. Wird eine Anwendung mit der Komponente *v2* erstellt, ist dieser nicht bekannt, daß es auch eine *v1* mit einem eingeschränkten Interface gibt. Laufzeitfehler sind vorprogrammiert, sobald die Anwendung *v1* anstatt der aktuellen *v2* verwendet und eine Methode der neueren Komponente *v2* aufgerufen wird. Es gibt zwei Möglichkeiten dieses Problem zu vermeiden: Entweder werden stets die neuesten Versionen von Clients und COM-Komponenten verwendet, oder man verwendet User-Definierte Interfaces. Hierbei werden abstrakte Klassen definiert und separat von der Klassendefinition implementiert.

Versuchen wir doch die Klasse *HelloWorld* (*v1*) aus Abschnitt 2.3.2 um die Methode `SayNothing` zu erweitern (*v2*). Zuerst wird für *v1* ein Interface namens *IHelloWorld* definiert, analog dazu für *v2* ein Interface namens *IHelloWorld2*, wobei *IHelloWorld2* das gesamte Interface von *IHelloWorld* unterstützen muß. Folgendermaßen könnte die Implementierung des Clients aussehen:

```
Option Explicit
Dim HW As IHelloWorld
Set HW = New HelloWorld2

' Prüfen, ob HW IHelloWorld2 unterstützt
If TypeOf HW Is IHelloWorld2 Then
    Dim HW2 As IHelloWorld2
    Set HW2 = HW
    ' Zugriff auf HW über typsicheres Objekt HW2
Else
    ' Zugriff auf HW mit eingeschränktem Interface
End If
```

Zusammenfassend kann gesagt werden, daß diese Art der Versions- und Interfaceabfrage eine sehr einfache ist. Gerade in großen Projekten mit sich häufig ändernden Schnittstellen ist diese Methode sehr effizient und praktikabel.

2.4.1 Versionierung unter Visual Basic

Da Visual Basic IDL nicht kennt, muß ein anderer Mechanismus angeboten werden, um Versionskompatibilität zu ermöglichen. Abbildung 4 zeigt im Dialog *Projekteigenschaften* folgende Auswahlmöglichkeiten für Versionierung unter Visual Basic an:

- ☞ Keine Kompatibilität: Bei jedem Kompilierungs-Aufruf werden ein neuer CLSID und neue IIDs erstellt. Dadurch können Anwendungen eine früher referenzierte COM-Komponente beim Rekompilieren nicht mehr gefunden werden.
- ☞ Projekt-Kompatibilität: Komponenten können geändert werden, ohne daß neue CLSIDs und IIDs erstellt werden. Nur jene Klassen, die zur vorhergehenden Version nicht binär kompatibel sind, bekommen neue IIDs.
- ☞ Binär-Kompatibilität: Visual Basic überprüft die COM-Komponente auf Änderungen im binären Interface. Es gibt hier drei Unterteilungen
    - o Idente Version: es gibt keine Änderungen im Interface.

- Kompatible Version: Das bestehende Interface wurde nicht verändert, jedoch wurden neue Methoden, Objekte oder Eigenschaften hinzugefügt.
- Unkompatible Version: Mindestens eine Schnittstelle der alten COM-Komponente wurde verändert. Die Wiederverwendbarkeit als eine der 4 Säulen von COM ist nicht mehr gegeben.

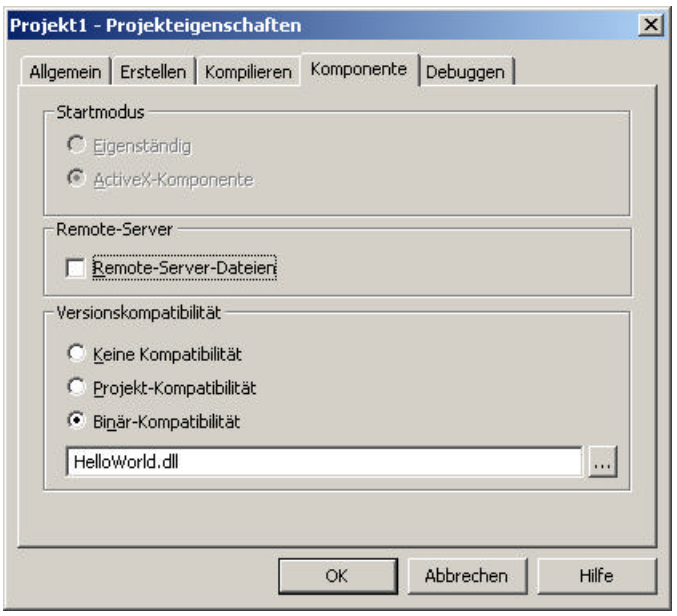


Abbildung 4: Projekteigenschaften - Versionskompatibilität

2.5 In-Process vs. Out-of-Process Server

COM-Komponenten sind Sammlungen von Methoden, die beliebig vielen Prozessen zur Verfügung gestellt werden; das trifft sowohl für weitere COM-Komponenten als auch auf Anwendungen zu. Funktionen und Methoden in DLLs werden auch In-Process-Routinen genannt, weil sie nur innerhalb des eigenen Speicherbereichs adressiert und daher aufgerufen werden können. Out-of-Process Server (Executables) hingegen haben den Vorteil auch außerhalb des eigenen Speicherbereichs Funktionen und Methoden für andere Prozesse zugänglich zu machen; davon leitet sich auch ihr Name ab.

Out-of-Process Server gehen mit Prozeßabstürzen fehlertoleranter um, sofern der Source-Code Error-Handling unterstützt. Dadurch wird die Stabilität der von einem Out-of-Process Server abhängigen Anwendungen sichergestellt und der Server-Prozeß kann neu gestartet werden. Ein anderer Grund Out-of-Process Server den In-Process Servern vorzuziehen ist die Tatsache, daß sich mehrere Anwendungen einen Out-of-Process Server miteinander teilen können.

Es ist naheliegend, daß Methodenaufrufe innerhalb des selben Prozesses (und Speicherbereich) schneller ablaufen werden als Methodenaufrufe eines Out-of-Process Server; das ist wohl auch der größte Nachteil eines Out-of-Process Server. Um prozeßübergreifende Methodenaufrufe zu ermöglichen, ist ein Vorgang namens Marshaling (mehr dazu in Abschnitt 2.9) notwendig, welcher verhältnismäßig viel Zeit in Anspruch nimmt. [Fre00]

2.5.1 Instanziierung

Ein In-Process Server läuft grundsätzlich immer im Prozeß seines Client. Out-of-Process Server haben stets ihren eigenen Prozeß. Es stellt sich nun die Frage wie viele Server-Prozesse erzeugt werden sollen. Immerhin kann im worst case jedes Objekt seinen separaten Server-Prozeß haben. Diese Frage läßt sich nicht allgemein beantworten, letztendlich muß sie vom Software-Entwickler entschieden werden unter dem Hinblick, was ihm zweckmäßig erscheint. Sowohl Visual C++ als Visual Basic bieten dem Programmierer die Möglichkeit für jede Klasse zu bestimmen, ob sie in einem separaten Server-Prozeß zu laufen hat oder nicht. Dies geschieht durch die *Instancing*-Eigenschaft einer COM-Klasse. Folgende Werte stehen dafür zur Auswahl::

- Private: Funktionen und Methoden der Klasse werden nicht veröffentlicht und können nur innerhalb des lokalen Projektes verwendet werden. Daraus ergibt sich, daß die Klasse weder einen CLSID noch einen IID hat.
- PublicNotCreatable: Die Klasse erzeugt eine nicht erzeugbare Co-Klasse sowie ein Interface. Diese Einstellung wird mitunter auch unter Visual Basic verwendet, um eine abstrakte Klasse zwecks Schnittstellendefinition ohne Implementierung zu erzeugen (vgl. Type Libraries unter Visual Basic). Die erzeugte Klasse ist wie bei *Private* keine richtige COM-Komponente, da keine Objekte instanziiert werden können und daher die Klasse nicht wiederverwendbar ist.
- MultiUse: Alle Instanzen einer Klasse laufen im gleichen Prozeß.
- GlobalMultiUse: Ist wie MultiUse und gibt zugleich Zugriff auf die Methoden als wären sie globale Funktionen.
- SingleUse: Dieser Wert kann nur bei Out-of-Process Servern der Instancing-Property zugewiesen werden. Es wird sichergestellt, daß jede Instanz der betroffenen Klasse in einem separaten Prozeß abläuft.

- ☞ GlobalSingleUse: Ist wie SingleUse und gibt zugleich Zugriff auf die Methoden als wären sie globale Funktionen.

Obwohl man für jede COM-Klasse obige Eigenschaft separat vergeben kann ist es empfehlenswert, für alle Klassen in einem Server die gleiche Einstellung zu verwenden. [MSD00, Pat00]

2.6 Threading

In den vergangenen Abschnitten wurde schon einige Male der Begriff des Threading geprägt. In den Programmbeispielen wurden bestimmte Threading-Einstellungen vorgegeben ohne deren Bedeutung näher zu erklären. Das Verständnis für Threadingmodelle und deren Bedeutung auf eigene Programme ist jedoch für COM-Programmierung unerläßlich.

Freundschaft mit Threads schließen zu können geht über die Leiche des Verständnisses für das Apartment-Konzept. Ein Apartment läßt sich am Einfachsten mit einem logischen Container innerhalb eines Prozesses beschreiben. Dieser Container ist für Objekte mit den gleichen Thread-Zugriffsanforderungen vorgesehen.

Schon der Umstieg von Windows 3.1 auf Windows 95/NT 3.5 oder höher brachte Softwareentwicklern immense Vorteile, was die Threading-Unterstützung anbelangt. Mit der Einführung der Multithreaded Apartments (MTAs) unter Windows NT 4.0 war es erstmals möglich, mehrere Objekte in einem Apartment zu haben.

Um ein COM-Objekt einem Thread zuzuordnen, muß über die Funktion `CoInitializeEx` COM initialisiert werden, wobei als Parameter die Konstanten `COINIT_APARTMENTTHREADED` oder `COINIT_MULTITHREADED` zur Festlegung des Threading-Modells übergeben werden. Erst das Terminieren des Threads oder der Aufruf der Funktion `CoUninitialize` entfernen den Thread wieder aus seinem zugeordneten Apartment.

Es sei hier darauf hingewiesen, daß sich in den vergangenen Jahren bei der Bezeichnung von Threading-Modellen zwei unterschiedliche Lager gebildet haben. Während die Win32 SDK-Dokumentation von Singlethreaded und Multithreaded Apartments spricht, verwenden andere Entwicklungsumgebungen die Bezeichnungen free threaded bzw. apartment threaded. Während free threaded Komponenten in einem MTA laufen, laufen apartment threaded Komponenten in einem STA.

Folgende Beschreibung der schon unter COM bekannten Theading-Modelle soll veranschaulichen, was bei der Wahl des idealen Threading-Modells beachtet werden muß.

Die Hauptfrage zum Finden des optimalen Threading-Modells sollte am Besten lauten: "Wie verhält sich nun aber ein singlethreaded Apartment bei gleichzeitigen Operationen?" Objekte haben es im Allgemeinen nicht sehr gern, wenn ein Thread dieses benutzt, während dieses Objekt schon für einen anderen Thread eine Operation ausführt. Datenkorruption kann die Folge sein. Hier sind entweder gute vorgefertigte Mechanismen gefragt, die Schaden durch gleichzeitige Operationen vermeiden helfen, oder Konzepte, die es gar nicht erst zu gleichzeitigen Aufrufen kommen lassen.

Objekte in einem STA werden in jenem Thread ausgeführt, in dem sie erzeugt wurden. STAs gestatten es nur einer Methode gleichzeitig ausgeführt zu werden. So einfach das Vermeiden von Parallelitäten nun sein mag, stellt sich jedoch die Frage, wie ein STA jedoch Methodenaufrufe aus anderen Apartments zulassen will; immerhin will man ja auch die Wiederverwendbarkeit einer Komponente unterstützen.

Beim Anlegen eines neuen Single Threaded Apartments, wird ein für den Programmierer nicht sichtbarer Bereich angelegt. In diesem wird eine Queue angelegt, an die andere Apartments Methodenaufrufe schicken. Diese Queue wird nach dem Prinzip first in first out abgearbeitet, wodurch sichergestellt ist, daß immer nur eine Methode gleichzeitig aufgerufen werden kann. Man kann also sagen, das STA wurde mit der Absicht entwickelt, eine Umgebung für Komponenten zu schaffen, in der sich der Programmierer nicht explizit um Code für Synchronisierung oder Sperren kümmern muß.

In MTAs wiederum können beliebig viele Methoden gleichzeitig in einem Apartment ausgeführt werden. Es ist jedoch Vorsicht geboten, da diese sich unter Umständen gegenseitig behindern oder ungewollte Seiteneffekte auftreten. Bei MTAs muß im Gegensatz zum STA eventuell explizit Quellcode zur Vermeidung dieser Seiteneffekte geschrieben werden.

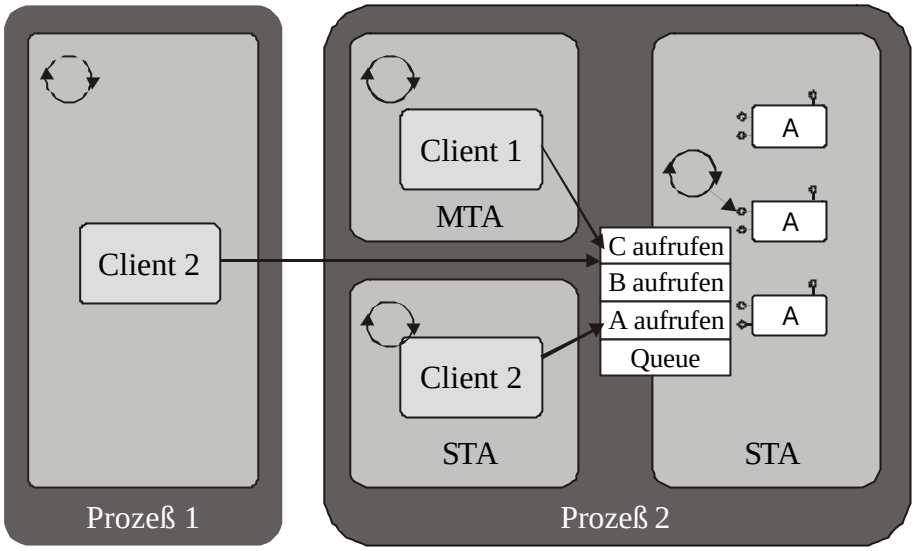


Abbildung 5: Gleichzeitige Aufrufe mehrerer Methoden aus unterschiedlichen Apartments

Neben den bis dato erwähnten Singlethreading und Multithreading kennt COM auch noch das Free Threading. Free Threading gestattet mehreren Threads ein beliebiges COM-Objekt innerhalb eines Prozesses gleichzeitig anzusprechen. Mit der aktuellen Version von Visual Basic ist es nicht möglich free threaded Komponenten zu erstellen. Die Verwendung von Visual C++ zur Erstellung von free threaded Komponenten ist daher unumgänglich. Das Free Threading-Modell kennt jedoch keinerlei Synchronisierungsmechanismen, so daß der Programmierer explizit wieder den erforderlichen Synchronisationscode selbst schreiben muß. Auch wenn das Schreiben von eigenen Synchronisierungsmechanismen durchaus mühsam sein mag, sollte der vernünftige Software-Entwickler zumindest einmal die Verwendung von C++ sorgfältig andenken. Da aber bei zeitkritischen Anwendungen die Geschwindigkeit eine immer größere Rolle spielt, sollte dieser Mehraufwand jedoch in Kauf genommen werden, um in den Genuß vom Free Threading zu kommen.

2.7 Anwendungsbeispiel

Um das Wissen der vergangenen Abschnitte zu festigen, scheint ein Anwendungsbeispiel anhand der behandelten COM-Grundlagen gerade das Richtige zu sein. Um aber auch die Wiederverwendbarkeit herausstreichen zu können, soll sich das Beispiel in die Teile *Erstellen einer COM-Komponente* und *Verwendung von COM* (Client – Frontend) gliedern. Nach Belieben darf auch die Programmiersprache der einzelnen Teile ausgetauscht werden – warum denn auch nicht? Folgende Beispiele werden anhand von Visual Basic erklärt, es ist

jedoch nicht wesentlich komplizierter eine gleiche Implementierung der COM-Komponente mit Visual C++ Active Template Library zu erzeugen.

2.7.1 Erstellen einer COM-Komponente

Die Komponente *HelloWorld* (vgl. Abschnitt 2.3.2) besteht aus den Methoden `SayHello` und `SayAnything`. Um den Umweg über die Erstellung von Type Libraries zu gehen, wird an dieser Stelle bewußt eine Klasse *IHelloWorld* mit Interface, jedoch ohne Implementierung erstellt, welche als Schnittstellendefinition für die eigentliche Komponente dienen soll. Um diese zu definieren, muß zuerst ein neues Projekt angelegt werden. Da es beabsichtigt ist einen In-Process Server zu erzeugen, wird ActiveX-DLL anstatt ActiveX-EXE (für Out-of-Process Server) ausgewählt.

Als Projektname kann ein beliebiger vergeben werden, der Klassenname sollte aber durchwegs aussagekräftig sein, da die Komponente ja fortan auch durch seinen Namen identifiziert wird. Um eine abstrakte Klasse zu erzeugen, wird die Eigenschaft *Instancing* auf *PublicNotCreatable* geändert. Im Dialog *Projekteigenschaften* werden alle Einstellungen übernommen, d.h. keine Modifikationen vorgenommen. Wird trotzdem der Schalter *Remote-Server-Dateien* aktiviert, so werden automatisch Type Libraries beim Kompilieren erstellt.

Folgender Code definiert die Schnittstelle der COM-Komponente:

```
Option Explicit

Sub SayHello()
End Sub

Sub SayAnything(message As String)
End Sub
```

Während `SayHello` keine Parameter hat, wird an `SayAnything` ein String übergeben.

Zusätzlich zu *IHelloWorld* wird noch eine weitere Klasse *HelloWorld* dem Projekt hinzugefügt. Drei Unterschiede gibt es aber schon im Vergleich zu *IHelloWorld*:

- ☞ Um auf die abstrakte Klasse *IHelloWorld* zu verweisen wird der Klasse das Statement `Implements IHelloWorld` am Anfang hinzugefügt.
- ☞ Den Methodennamen wird der Name der abstrakten Klasse durch "_" getrennt vorangestellt.
- ☞ Der Source-Code besteht diesmal aus Methodendefinition mit Implementierung.

☞ Die Eigenschaft *Instancing* hat den Wert *MultiUse*

```
Option Explicit
Implements IHelloWorld

Private Sub IHelloWorld_SayHello()
    MsgBox ("Hello world!")
End Sub

Private Sub IHelloWorld_SayAnything(message As String)
    MsgBox (message)
End Sub
```

Nach dem Registrieren der COM-Komponente mittels `RegSvr32` (siehe Abschnitt 2.8) wird sich ein entsprechender Eintrag mit *dcomcnfg.exe*, dem COM Explorer oder dem Objektbrowser von Visual Basic finden. Der uneingeschränkten Verwendung der soeben erstellten Komponente steht nichts mehr im Weg.

2.7.2 Verwendung von COM

Die COM-Komponente *HelloWorld* eignet sich fortan zur Wiederverwendung in anderen Komponenten oder Executables. Um zu einem sichtbaren Ergebnis zu kommen soll hier eine Anwendung erstellt werden. Zu diesem Zweck wird ein neues Projekt vom Typ *Standard-EXE* erstellt. Im Menüpunkt Projekt – Verweise wird auf die zu verwendende Komponente referenziert. In diesem Fall wird auf *Projekt1* verwiesen, da dies der Projektname der vorigen Komponente ist. Sollen COM-Komponenten verwendet werden, die noch nicht registriert sind, wird unter *Durchsuchen* ein Dialog geöffnet, in dem man dann die betreffende Komponente auswählen kann.

```
Option Explicit
Dim HW As IHelloWorld

Private Sub SayHelloBtn_Click()
    MsgBox HW.SayHello
End Sub

Private Sub SayAnythingBtn_Click()
    MsgBox HW.SayAnything(Text1.Text)
End Sub

Private Sub Form_Load()
```

```
    Set HW = New HelloWorld
End Sub

Private Sub Form_Terminate()
    Set HW = Nothing
End Sub
```

`HW` ist jenes Objekt, das den Objekttyp der vorher definierten Klasse `IHelloWorld` instanziiert. Beim Start der Anwendung instanziiert `HW` die COM-Komponente `HelloWorld`. Von nun an kann auf sämtliche Methoden (Funktionen und Properties) der betreffenden Klasse zugegriffen werden. Wird nun beispielsweise durch Klick auf die Schaltfläche `SayHelloBtn` `HW.SayHello` aufgerufen, gibt diese Methode einen String zurück, der wiederum als Text an eine Messagebox übergeben wird. Welche Operationen tatsächlich beim Methodenaufruf in der COM-Komponente durchgeführt werden, entzieht sich dank Objektkapselung der Kenntnis des Programmierers. Eine saubere Trennung zwischen Client- und Server-Anwendung ist die Folge.

2.8 Installation von COM

Um in den Genuß von COM/DCOM-Funktionalität kommen zu können, ist es vorab einmal wichtig sicherzustellen, daß auch wirklich auf dem eigenen Rechner (D)COM-Unterstützung installiert ist. Bis auf alte Rechner mit Windows 95 wird auf allen neuen Microsoft-Betriebssystemen die (D)COM-Unterstützung automatisch mitgeliefert. Wenn nicht, so ist ein Besuch im Microsoft Downloadcenter ratsam, das auf den Suchbegriff *DCOM95* das entsprechende Utility zum Aufrüsten bereitstellt. Wer Microsoft Office 2000 oder Internet Explorer 5.0 oder höher sein Eigen nennt, der hat (D)COM automatisch auf seinem Rechner installiert. Gelegentlich ist es auch schon unter Windows 98 Second Edition vorgekommen, daß zwar die (D)COM-Funktionalität verfügbar war, jedoch ein ganz wesentliches Programm zur Konfiguration eigener Komponenten gefehlt hat: *dcomcnfg.exe. dcomcnfg.exe* sollte im Pfad *%SystemRoot%\System32* zu finden sein; wenn nicht, dann findet sich diese Datei in der *win98_39.cab* der Installations-CD von Windows 98SE.

dcomcnfg.exe ist auch ein gutes Werkzeug, um festzustellen, welche Out-of-Process Server auf einem System registriert sind. Im Allgemeinen soll das den Programmierer nicht in Sicherheit wiegen, daß er keine Leichen in der Registry beherbergt, man kann bestenfalls

feststellen, welche registrierte Out-of-Process Server-Komponenten auf dem System installiert sind. Im übrigen trifft obiges für In-Process DLLs nicht zu.

ActiveX DLLs haben unter anderem auch Routinen eingebaut, die sich um das Registrieren (Routine `DllRegisterServer`) und das Unregistrieren kümmern. Die Syntax dazu lautet:

- ☞ `RegSvr32 NameDerKomponente` zum Registrieren bzw.
- ☞ `RegSvr32 /u DLLName`, wobei `/u` für Uninstall steht.

Nach Abschluß des (Un-)Registrierens öffnet sich eine Messagebox, die das Ergebnis des Vorgangs zusammenfaßt. Was versteckt sich aber hinter `RegSvr32`? `RegSvr32` ist eine Anwendung, die folgende Aufgaben beim Registrieren bzw. Unregistrieren einer In-Process Server-Komponente durchführt, wobei der Anwender von diesen Vorgängen nichts mitbekommt und diese auch nicht manuell anstoßen muß:

- ☞ Aufruf von `LoadLibrary` um Modul zu laden
- ☞ Aufruf von `GetProcAddress` um Pointer auf die Funktion `DllRegister` bzw. `DllUnregister` zu bekommen
- ☞ Aufruf von der Funktion `DllRegister` bzw. `DllUnregister`
- ☞ Aufruf von `FreeLibrary`

In-Process Server verwenden die Funktionen `DllRegisterServer` und `DllUnregisterServer` um sich selbst zu registrieren. Während beim Registrieren in der Windows Registry Einträge hinzugefügt werden, werden genau diese beim Unregistrieren wieder aus der Registry entfernt. Die Funktionen `DllRegisterServer` und `DllUnregisterServer` müssen von der betreffenden COM-Komponente exportiert werden. Ist dies nicht der Fall werden Programme wie `RegSvr32` erfolglos beim Aufruf dieser Funktionen abbrechen. Folgendes Listing unter Visual C++ zeigt die Implementierungs-struktur von `DllRegisterServer` und `DllUnregisterServer`.

```
// ...
// DllRegisterServer - Fügt der Systemregistrierung Einträge hinzu
// ...
STDAPI DllRegisterServer(void)
{
  // RegCreateKey() bzw. RegSetValue() zum Hinzufügen von Registry-Keys
  ...
  return NOERROR;
}
```

```
// ...
// DllUnregisterServer - Entfernt Einträge aus der Systemregistrierung
// ...
STDAPI DllUnregisterServer(void)
{
  // RegDeleteKey() zum Entfernen von Registry-Keys
  ...
  return NOERROR;
}
```

Es ist zu beachten, daß dieses Beispiel in der Implementierung der Funktionen `DllRegisterServer` bzw. `DllUnregisterServer` abweicht, wenn ATL zum Einsatz kommt, da diese einem die Arbeitsschritte für die Registrierung abnimmt. Wird anstatt einer In-Process Server COM-Komponente ein Out-of-Process Server erstellt, gibt es keine Funktionen `DllRegisterServer` und `DllUnregisterServer`, die unter Verwendung von `RegSvr32` für das korrekte (Un-)Registrieren sorgen. Statt dessen übernimmt die Aufgabe der Registrierung die Funktion `WinMain()`. Es müssen anschließend nur mehr über die Eingabeaufforderung die Parameter `/RegServer` bzw. `/UnRegServer` zwecks Registrierung übergeben werden.

Im übrigen darf sich jeder Softwareentwickler glücklich schätzen, daß einem die Implementierung obiger Funktionen unter Visual Basic und Visual C++ (ATL) durch das Softwarewerkzeug abgenommen wird.

2.8.1 Registry-Aufbau von COM

Nachdem unter Abschnitt 2.7.2 schon erwähnt wurde, daß COM-Komponenten in der Windows Registry eingetragen werden, soll der Registry hier nun besonderes Augenmerk geschenkt werden.

COM verwendet den Registry-Abschnitt `HKEY_CLASSES_ROOT` (HKCR), der sich in sechs Informations-Abschnitte aufteilt:

- ☞ Datei-Erweiterungen
- ☞ ProgIDs
- ☞ AppIDs
- ☞ CLSIDs

- ☞ Interfaces
- ☞ TypeLibs

Ergänzend zu diesen verwendet COM für Sicherheitseinstellungen noch den Schlüssel `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole`.

2.8.1.1 Datei-Erweiterungen

Datei-Erweiterungen sind Subschlüssel vom `HKEY_CLASSES_ROOT`-Schlüssel. Dieser Schlüssel legt fest wo COM nach der ProgID einer COM-Komponente nachschauen muß. Der Schlüssel einer Datei-Erweiterung hat folgende Struktur:

```
HKEY_CLASSES_ROOT
     { .Erweiterung } = { prog-id }
```

2.8.1.2 ProgIDs

ProgID-Schlüssel sind genauso wie Datei-Erweiterungen Sub-Schlüssel von `HKEY_CLASSES_ROOT`. ProgID-Schlüssel haben folgende Struktur in der Windows Registry:

```
HKEY_CLASSES_ROOT
     { ModulName.ObjektName } = Beschreibung
          CLSID = { CLSID als GUID }
          CurVer = { versionsunabhängige ProgID[4] }
     { ModulName.ObjektName.VersionsNummer } = Beschreibung
          CLSID = { CLSID als GUID }
```

Wozu benötigt man als Programmierer eigentlich einen ProgID? Aus obiger Struktur ist ersichtlich, daß jedem ProgID ein CLSID in Form eines GUID zugeordnet ist. GUIDs sind im Allgemeinen jedoch in der Praxis zu kompliziert, um über sie Komponenten zu identifizieren. ProgIDs sollen dieses Problem umgehen, indem der Programmierer selbst eine Bezeichnung für seine Komponente aussuchen kann. Der CLSID ist also eine Übersetzung eines einfachen ProgID in einen GUID. Da man also annehmen darf, daß ProgIDs und CLSIDs bijektiv sind, können folgende zwei APIs zwischen ProgID und CLSID übersetzen.

```
WINOLEAPI ProgIDFromCLSID (REFCLSID clsid, LPOLESTR FAR* lpszProgID);
WINOLEAPI CLSIDFromProgID (LPOLESTR lpszProgID, REFCLSID clsid);
```

[4] Es ist zweckmäßig einen ProgID zu haben, der unabhängig von der installierten Version ist, damit Programmierer nicht die installierte Version einer COM-Komponente abfragen müssen. Die Syntax ist `ModulName.ObjektName`

ProgIDs haben bewußt die Syntax `ModulName.Objektname.VersionsNummer`. `ModulName` und `ObjektName` sollten im Allgemeinen ausreichen, um ein Objekt eindeutig zu identifizieren. Sie reichen dann aus, wenn nur eine einzige Implementierung einer Komponente auf einem Rechner installiert ist. Der Integer `VersionsNummer` am Ende des ProgID ermöglicht hierbei aber eine Unterscheidung der einzelnen Versionen, indem er mit jeder neuen Implementierung automatisch erhöht wird und daher eine neue von einer alten Implementierung unterscheiden hilft.

2.8.1.3 AppIDs

Application IDs geben Auskunft darüber, wie ein COM-Server gestartet werden soll. Immerhin gilt es zu unterscheiden, ob auf einen COM-Server über das Netzwerk zugegriffen werden darf, welche Sicherheitseinstellungen er und seine Klienten benötigen und ob er als Service gestartet werden soll. AppIDs sehen wie folgt aus:

```
HKEY_CLASSES_ROOT
APPID
{ AppID als GUID }
    RemoteServerName = { DNS oder IP-Adresse }  // nur bei Client
    DLLSurrogate = { Pfad\Surrogat.exe oder NULL bei DllHost.exe }
    // nur bei Server
    LaunchPermission = REG_BINARY { binäre Sicherheitsbeschreibung }
    AccessPermission = REG_BINARY { binäre Sicherheitsbeschreibung }
    ActivateAtStorage = { Y oder N }
    LocalService =
        ServiceParameters =
        RunAs = { none | "Interactive User" | User Account }
            AuthenticationLevel =
```

Hat `RemoteServerName` einen Eintrag, so wird die betreffende Komponente auf dem jeweiligen Rechner ausgeführt.

Da die Schlüssel `LaunchPermission` und `AccessPermission` als binäre Beschreibung gespeichert werden, kann für deren Aufbau keine allgemeine Beschreibung gegeben werden. Um diese Einstellungen zu modifizieren bedarf es entweder der Anwendung *dcomcnfg.exe* bzw. eigens erstellter Programme, die Änderungen an den Sicherheitseinstellungen durchführen.

`ActivateAtStorage` läßt die Unterscheidung zu, ob ein Objekt lokal erstellt werden soll oder nicht. Fehlt `ActivateAtStorage` oder hat es den Wert `N`, so werden neue Objekte (z.B. Dateien) grundsätzlich auf jenem Rechner ausgeführt, auf dem die betreffende Komponente

läuft. Bei `Y` wird COM versuchen, das betreffende Objekt von jenem Rechner zu laden, auf welchem die Komponente beheimatet ist, d.h. in einer verteilten Architektur auf dem entfernten Computer.

Sollte beabsichtigt sein, eine COM-Komponente als Service von Windows NT auszuführen, so wird unter `ServiceParameters` der Name des betreffenden Services angegeben.

Zuletzt gibt `RunAs` an, ob eine COM-Komponente im Sicherheits-Kontext eines anderen Benutzers gestartet wird. Ergänzend zum User, in dessen Kontext die Komponente gestartet wird, kann auch noch festgelegt werden, in welchem Kontext die COM-Anwendung fortan laufen soll.

2.8.1.4 CLSIDs

CLSIDs werden im Schlüssel `HKEY_CLASSES_ROOT\CLSID` gespeichert. Der CLSID-Schlüssel hat folgende Syntax:

```
HKEY_CLASSES_ROOT
    CLSID
    { CLSID als GUID } = Beschreibung
        AppID = { AppID als GUID }
        ProgID = { ModulName.ObjektName.VersionsNummer }
        VersionIndependentProgID = { ModuleName.ObjektName }
        InProcServer32 = { Pfad\Modul.dll }
            ThreadingModel = { [kein Wert] | Apartment | Free | Both }
        InprocHandler32 = { Pfad\Handler.dll }
        LocalServer32 = { Pfad\Modul.exe }
        DefaultIcon = { Pfad\Modul, Resource-ID }
        TypeLib = { TypeLib als GUID }
```

Da schon einige Begriffe in den vergangenen Registry-Erklärungen gefallen sind, sollen hier nur mehr die noch unbekannten erwähnt werden. Wenn `InprocServer32` einen Wert (hier der jeweilige Pfad der DLL) besitzt, so kann die COM-Komponente als In-Process Server gestartet werden. `ThreadingModel` gibt logischerweise Information über das von der Komponente verwendete Threading-Modell. `InprocHandler` gibt den Pfad einer Datei als Objekt-Handler an, die u.a. für das Marshaling zuständig ist; als Standardwert wird *ole32.dll* angenommen. Handelt es sich bei der COM-Komponente nicht um einen In-Process Server, sondern um einen Out-of-Process Server, so wird der Pfad der EXE-Datei unter `LocalServer32` angegeben. Abschließend referenziert `DefaultIcon` noch ein Symbol an, das mit der Komponente assoziiert wird.

2.8.1.5 Interfaces

Informationen zu den COM-Schnittstellen finden sich unter
`HKEY_CLASSES_ROOT\Interface:`

```
HKEY_CLASSES_ROOT
    Interface
        { Interface als GUID } = Beschreibung
            ProxyStubClsid32 = { CLSID als GUID }
            TypeLib = { TypeLib als GUID }
```

Sobald eine COM-Komponente mittels DllRegisterServer registriert wird, wird im Interface-Subschlüssel ein Eintrag für die betreffende Komponente angelegt. `ProxyStubClsid32` referenziert den CLSID von Proxy und Stub der Komponente. Ein häufig anzutreffender Wert ist `{00020420-0000-0000-C000-000000000046}`, der für IDispatch steht.

2.8.1.6 TypeLibs

`TypeLib` ist der CLSID einer Type Library:

```
HKEY_CLASSES_ROOT
    TypeLib
        { TypeLib als GUID } = Beschreibung
            { major.minor } = Beschreibung
                { Lokale ID } = Beschreibung
                    { Plattform } = { Modul }
                    HELPDIR = { Hilfepfad }
```

Die `Lokale ID` ist ein Hex-String von 1 bis 4 Zeichen Länge und gibt Auskunft über die verwendete Sprache der Komponente; 0 steht für die im System eingestellte Standardsprache.

Die Registry-Einstellungen von COM kennen vier verschiedene Plattformen, unter denen eine COM-Komponente laufen kann: `win16`, `win32`, `mac` und `other`. Module können Type Libraries (*.tlb), DLLs und Executables sein.

`HELPDIR` spezifiziert zum Schluß noch den Pfad eines etwaigen Help-Files.

2.8.1.7 Sicherheitseinstellungen

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole` ist jener Pfad, in dem sämtliche Sicherheitseinstellungen für COM-Komponenten gespeichert sind.

```
HKEY_LOCAL_MACHINE
    SOFTWARE
        Microsoft
            Ole
                EnableDCOM = { Y oder N }
```

```
DefaultAccessPermission = REG_BINARY
    { binäre Sicherheitsbeschreibung }
DefaultLaunchPermission = REG_BINARY
    { binäre Sicherheitsbeschreibung }
LegacyAuthentication =
LegacyImpersonation =
LegacySecureReferences = { Y oder N }
```

Mit `EnableDCOM` wird die Aktivierung von entfernten Objekten via DCOM gesteuert. Bei `DefaultAccessPermission` und `DefaultLaunchPermission` hat nur eines der beiden Attribute einen Wert. Wie der Name schon impliziert, geben sie die DCOM-Sicherheitseinstellung beim Ausführen bzw. Starten einer Komponente an.

`LegacyAuthentication` repräsentiert die Sicherheitseinstellungen der Standardauthentifizierungsebene, die auch über das Tool *dcomcnfg.exe* modifiziert werden kann. Folgernde Werte stehen zur Auswahl:

- ☞ `RPC_C_AUTHN_LEVEL_DEFAULT` (Standard[5]): Es werden die Standardeinstellungen für Sicherheitschecks übernommen. Unter Windows NT 4.0/2000 ist diese Einstellung ident mit Verbinden.
- ☞ `RPC_C_AUTHN_LEVEL_NONE` (Kein): Es werden keine Sicherheitsprüfungen durchgeführt.
- ☞ `RPC_C_AUTHN_LEVEL_CONNECT` (Verbinden): Sicherheitsüberprüfungen werden erst durchgeführt, wenn eine Verbindung mit dem Server hergestellt wurde. Diese Einstellung setzt ein verbindungsorientiertes Netzwerkprotokoll voraus.
- ☞ `RPC_C_AUTHN_LEVEL_CALL` (Aufruf): Bei jedem Aufruf werden Sicherheitschecks durchgeführt.
- ☞ `RPC_C_AUTHN_LEVEL_PKT` (Paket): Authentifiziert jedes einzelne Paket.
- ☞ `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY` (Paketintegrität): In jedes Paket wird vom Sender signiert, um zu garantieren, daß es unverändert beim Server ankommt.
- ☞ `RPC_C_AUTHN_LEVEL_PKT_PRIVACY` (Paketvertraulichkeit): Ergänzend zur Paketintegrität werden die Pakete zusätzlich noch verschlüsselt.

[5] Die Bezeichnungen in der Klammer sind der jeweils dafür verwendete Ausdruck in *dcomcnfg.exe*

`LegacyImpersonation` ermöglicht die Identität anderer User und deren Sicherheitskontext anzunehmen. Folgende Einstellungen sind zulässig:

- ☞ `Anonymous` (Anonym): Die Server-Anwendung überprüft nicht die Identität des Aufrufers.
- ☞ `Identify` (Identifizieren): Der Aufrufer muß sich gegenüber der Server-Anwendung authentifizieren.
- ☞ `Impersonate` (Identität wechseln): Der Aufrufer darf seine Identität wechseln, sofern er nicht andere Objekte aufruft.
- ☞ `Delegate` (Delegieren): Der Aufrufer darf seine Identität wechseln auch wenn er andere Objekte aufruft.

Hat `LegacySecureReferences` den Wert `Y`, werden alle COM-Objekte der COM-Sicherheit unterworfen, auch wenn der Programmierer für sie nicht durch `CoInitializeSecurity()` einen eigene Sicherheitsrichtlinien definiert hat; der Anwender muß dafür jedoch in Kauf nehmen, daß seine Anwendung dadurch langsamer wird. [`Fre00`, `Lei00`]

Abschließend zum Thema COM und Registry möchte ich an dieser Stelle den *COM Explorer 2.0* (Download unter [`Dev01`]) weiterempfehlen. Mit diesem Werkzeug ist es ein Kinderspiel alle auf einem Computer registrierten Komponenten mitsamt ihren Registry-Einträgen in unterschiedlichen Ansichten zu betrachten, diese teilweise zu verändern, Komponenten zu löschen oder einen Bericht über eine oder mehrere COM-Komponenten zu erstellen.

2.9 DCOM/Marshaling

Die wohl größte Stärke von COM besteht in der Unterstützung von Ortstransparenz. Mit Ortstransparenz verbindet man die Unabhängigkeit davon, ob eine Komponente lokal oder auf einem entfernten Rechner ausgeführt wird und ob sie ein In-Process oder Out-of-Process Server ist. Das mag sich nun zwar trivial anhören, es bedarf jedoch viel Know-How und Implementierungsarbeit um Ortstransparenz zu erreichen. Dank COM muß kein zusätzlicher Code selbst geschrieben werden. Durch die Spezifikation eines Interfaces in IDL ist *midl.exe* in der Lage den dafür nötigen Code selbständig zu generieren.

Marshaling ist jener Prozeß, um Ortstransparenz zu erreichen. Da Methodenaufrufe von Interfaces verteilter Komponenten nicht möglich sind, wird dieses Problem mit Marshaling

umgangen. Wir wissen, daß Methodenaufrufe von Interfaces über Zeiger auf jene Speicheradressen durchgeführt werden, wo die Einsprungpunkte für die aufzurufenden Methoden im Speicher sind. Das ist jedoch nur innerhalb des gleichen Prozesses zulässig, da prozeßfremde Speicherbereiche geschützt werden. Aus Performance-Gründen ist dieser Weg zu bevorzugen mit der Einschränkung, daß das eben nicht immer möglich ist. Sobald eine COM-Komponente in einem separaten Prozeß (Out-of-Process Server) oder gar auf einem anderen Rechner läuft (man spricht hier von Distributed COM/DCOM), wird Marshaling eingesetzt, um prozeß-übergreifende Methodenaufrufe zu ermöglichen.
Marshaling erzeugt einen Proxy im Speicherbereich des Clients. Dieser Proxy sieht genauso wie das eigentliche COM-Interface aus, das ausgerufen werden soll. Er hat die gleichen Methoden, Parameter und return types wie das COM-Interface. Sobald eine lokale oder entfernte COM-Komponente instanziiert wird, wird auch der Proxy durch die COM-Runtime automatisch erzeugt. Wird nun eine Methode aufgerufen, so packt der Proxy den Methodenaufruf zu einer InterProcess Communication-Nachricht (IPC) zusammen. Diese Nachricht enthält jene Informationen, die die gewünschte Methode im Server-Prozeß aufrufen sowie die Parameter der Methode. Diese Nachricht wird mittels RPC zum Server-Prozeß geschickt. Man unterscheidet hier zwischen Local RPC auf einem lokalen Rechner und Distributed Computing Environment (DCE) RPC in verteilten Systemen. Ein sogenannter Stub im Server-Prozeß empfängt IPC-Message wieder und extrahiert aus den Informationen der Nachricht den gewünschten Methodenaufruf mit den erforderlichen Parametern. Nach Aufruf der Methode sendet der Stub den Rückgabewert der Methode an den Proxy zurück.

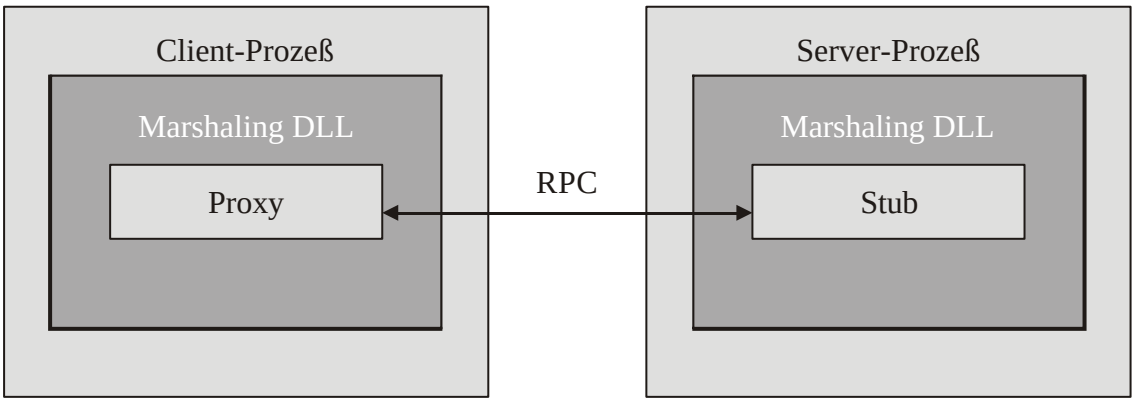


Abbildung 6: Marshaling eines Out-of-Process Servers

Wie erstellt man aber jetzt COM-Komponenten, die den Proxy-Stub-Mechanismus unterstützen? Während Visual Basic automatisch Distributed COM unterstützt, muß bei Visual C++ beachtet werden, im ATL-COM-Anwendungs-Assistent die Checkbox

Vereinigung von Proxy-/Stub-Code zulassen anzuklicken. Der Source-Code des Projekts wird so modifiziert, daß fortan der Proxy-/Stub-Layer automatisch generiert wird.

Werden In-Process Server auf im Netzwerk entfernten Rechnern ausgeführt, bedarf es eines sogenannten Surrogats. Standardmäßig wird dafür die Anwendung `dllhost.exe` verwendet. `dllhost.exe` wird von Windows zur Verfügung gestellt. Es steht dem Programmierer jedoch frei, ein selbstprogrammiertes Surrogat zu verwenden. Auf dieses wird bei `DLLSurrogate` referenziert. [Edd98]

3 Verbesserungen durch COM+

Im Vergleich zu COM kann COM+ mit Verbesserungsansätzen auf zwei Ebenen aufwarten. Einerseits war das die Erweiterung des Microsoft Transaction Server (MTS) von Version 2 auf 3, andererseits das Hinzufügen von bislang nicht vorhandenen Diensten.

Der Verbesserung des MTS sind folgende Funktionen zuzuschreiben:

- ☞ Transaktionsdienste
- ☞ Sicherheitsdienste
- ☞ Synchronisationsdienste

Als Erweiterung der Basisfunktionalität hat COM+ noch diese Dienste anzubieten:

- ☞ Ereignisdienst (Event Service)
- ☞ Warteschlangen (Message Queuing): Message Queuing ist ein Mechanismus, der COM-Clients auch dann miteinander kommunizieren läßt, wenn diese nicht gleichzeitig in Betrieb sind oder sie nicht zur Laufzeit miteinander direkt Daten austauschen können. Das Loslösen von HTTP und RPC als Kommunikationsstandard erleichtert die Entwicklung von umfangreichen Applikationen ungemein.
- ☞ Lastverteilung (Load Balancing): Gerade komplexe Geschäftsanwendungen sind dazu prädestiniert, schnell an ihre Leistungsfähigkeit zu stoßen. Lastverteilung ist Microsofts Antwort auf die Forderung zur Schaffung eines Mechanismus, der Software-Entwicklern ein vorgefertigtes System zur Lastverteilung zur Verfügung stellen soll.

3.1 Allgemeine Verbesserungen

3.1.1 Threading

COM+ beschert Software-Entwicklern ein neues Threading-Modell namens Neutral Threading. Neutral Threading ist der Schlüssel zur Lösung von zeitkritischen Anwendungen wie MTS oder COM+. Es ist zugleich jenes Threading-Modell, das bei COM+-Programmierung zu bevorzugen ist.

Wie von COM bekannt ist sollen Apartments als Sammlungen von Objekten diese vor gegenseitiger Einflußnahme schützen und abgrenzen. Nachteil von apartment-übergreifenden Aufrufen ist das zeitintensive Marshaling, wodurch COM-Anwendungen langsamer ablaufen. Singlethreaded Apartments haben genauso wie multithreaded Apartments nach wie vor ihre Berechtigung auch in der COM+-Programmierung. Neutralthreaded Komponenten werden stets in einem neutralen Apartment erzeugt. Dieses enthält keine Threads sondern nur Objekte. Obwohl das neutralthreaded Apartment ähnlich wie MTA ist, ermöglicht dieses mehreren Threads gleichzeitig Zugriff auf seine Objekte. COM+ kümmert sich nach wie vor um das Threadmanagement und um gleichzeitige Methodenaufrufe, garantiert aber, daß stets nur ein Methodenaufruf von einem Objekt zur gleichen Zeit ausgeführt werden kann.

3.1.2 Installation von COM+-Anwendungen

Abschnitt 2.8 hat gezeigt, wie COM-Anwendungen mittels *dcomcnfg.exe* registriert werden können. Windows 2000 bietet mit der Einführung von COM+ zugleich ein neues Hilfsmittel, das Component-Services-Snap-In, zur Registrierung von COM+-Komponenten an.

Zur Installation einer COM+-Komponente wird das Component-Services-Snap-In von Windows 2000 geöffnet.

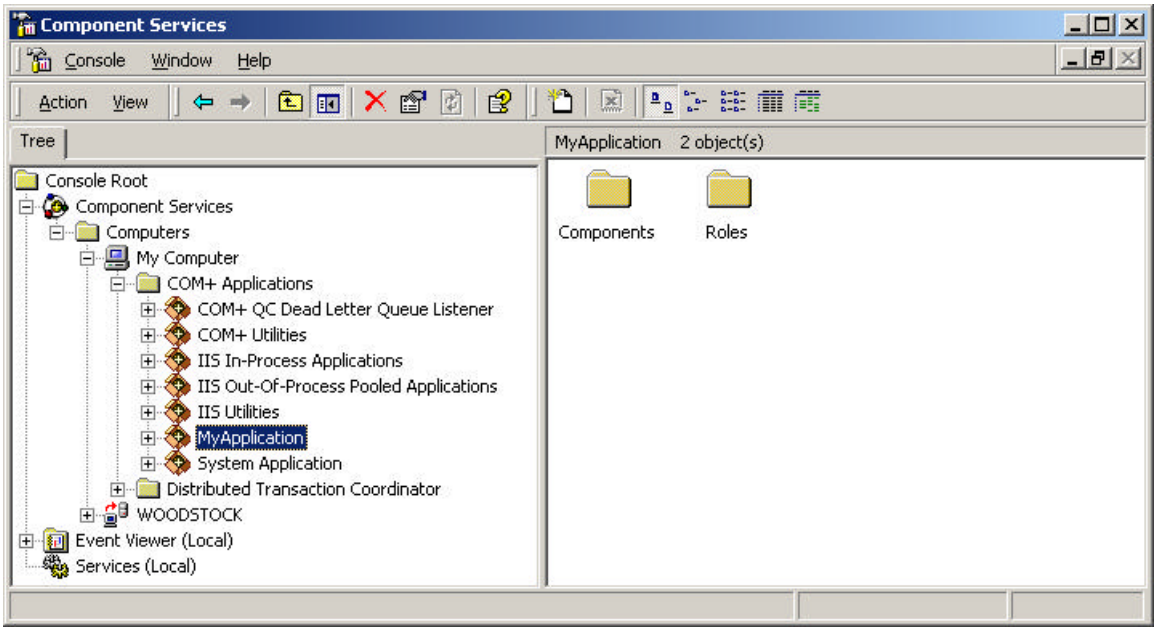


Abbildung 7: Component Services Snap-In

Abbildung 7 zeigt einen für Windows 2000 Server typischen Katalog mit COM+-Applikationen. Um für eine neue Klasse als COM+-Komponente einen separaten Raum zur Registrierung im System zu schaffen, soll eine neue COM+-Applikation angelegt werden. Zu

diesem Zweck muß mit der rechten Maustaste auf den Ordner *COM+ Applications* geklickt werden, um mit *New - Application* eine neue Applikation anzulegen.

Es öffnet sich ein Install Wizard, der dem Anwender nun zwei Möglichkeiten zur Auswahl anbietet: *Install pre-built application(s)* bedeutet, daß der Benutzer vorgefertigte Microsoft Installer-Files (*.msi) zur Installation fertiger Applikationen dem Wizard zur Installation übergeben würde. Da bei der erstmaligen Erstellung einer COM+-Applikation derartige Dateien nicht vorhanden sein können, wählt man *Create an empty application* als zweite Option aus. Als nächsten Schritt wird für die neue Anwendung ein passender Name vergeben.

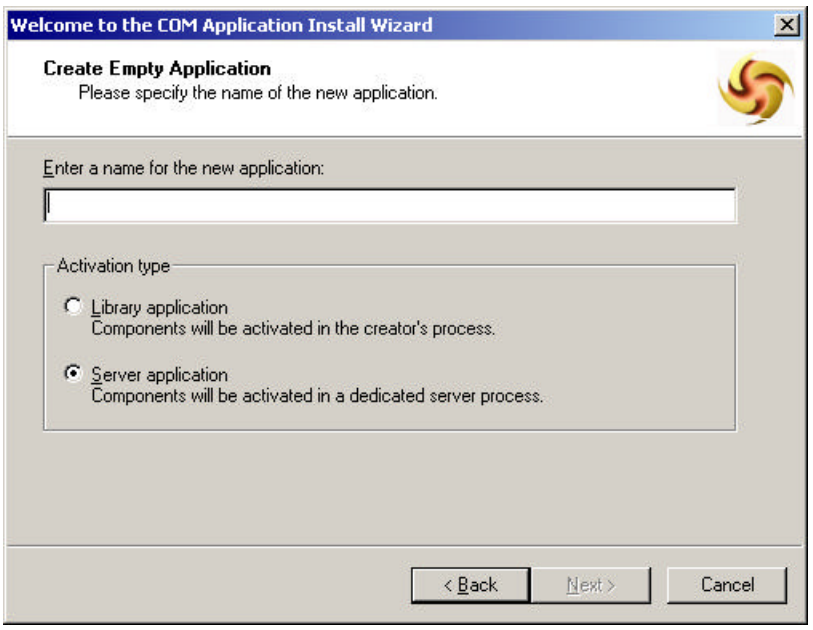


Abbildung 8: Installation einer neuen COM+-Applikation

Abbildung 8 bietet unter *Activation type* weiters noch zwei Möglichkeiten zur Einrichtung an. Während der Switch *Library application* nur die lokale Verwendung einer Komponente zuläßt, kann mit *Server application* auf eine Komponente auch von anderen Rechner zugegriffen werden. Bei letzterer Einstellung muß man jedoch in Kauf nehmen, daß die COM+-Anwendung an Geschwindigkeit einbüßt.

Die Auswahl über den User-Kontext, in dem eine Anwendung zu laufen hat, ist insofern von Interesse, als daß der aufrufende User genügend Rechte besitzen muß, um auf Komponenten zugreifen zu können. Mit *Administrator* als interaktiven Benutzer oder in dessen Kontext explizit für alle Aufrufer angegeben, ist der Zugriff auf eine Komponente für Versuchszwecke sichergestellt – wenngleich unberechtigtes Ausführen genauso zu Unannehmlichkeiten führen kann.

Anschließend können der COM+-Anwendung mittels des Menübefehls *Action – New – Component* beliebig viele Komponenten hinzugefügt werden.

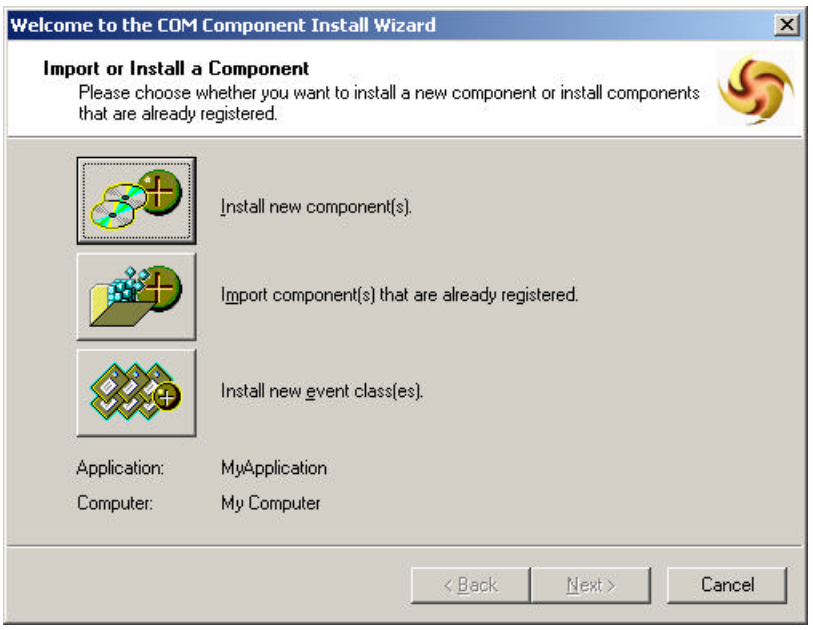


Abbildung 9: Installation von COM+-Komponenten

Abbildung 9 bietet folgende Auswahlmöglichkeiten an:

- ☞ Install new component(s): Es wird eine noch nicht im System registrierte Komponente der COM+-Anwendung hinzugefügt.
- ☞ Import component(s) that are already registered: Listet jene Komponenten zur Auswahl auf, die schon im System registriert sind.
- ☞ Install new event class(es): Erzeugt eine neue Event-Klasse; dazu aber mehr in Abschnitt 4.3.

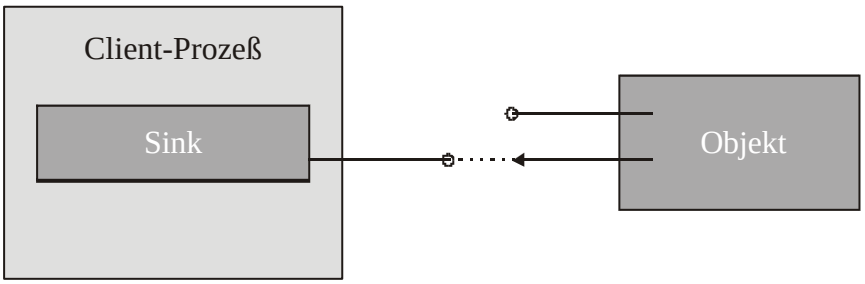
4 Event-Service

Nachrichtenübermittlung ist in verteilten Systemen ein klassisches Anwendungsgebiet für COM+. Angenommen, ein Prozeß, der Meßwerte erfaßt, soll an eine beliebige Zahl von Clients Meßdaten verteilen, sobald neue vorhanden sind. Nachfolgend wird der verteilende Prozeß Publisher bezeichnet, die empfangenden Clients als Subscriber (Abonnenten).

Es wäre eine einfache Methode, Subscriber in regelmäßigen Intervallen bei den jeweiligen Publishern den aktuellen Wert abzufragen zu lassen, um etwaige Änderungen zu erfassen. Problem hierbei ist jedoch, daß ein Subscriber nur in den wenigsten Anwendungsfällen wissen kann, wann neue Werte abzufragen sind. Um annähernd alle Änderungen erfassen zu können, müßten sehr kurze Abfrageintervalle angesetzt werden. Dies führt jedoch zu einer hohen Netzwerkbelastung einerseits, andererseits aber auch zu einer hohen Rechnerbelastung seitens des Publisher, die sich mit steigender Anzahl der Subscriber immens erhöhen würde.

4.1 Einfache Events

Im folgenden COM/DCOM-Beispiel wird die Veränderung der lokalen Variable `str` in einer Klasse *events* zu überwachen behandelt. Es sei gleich hier vorweg genommen, daß dieses Anwendungsbeispiel nur bei einem Subscriber pro Publisher funktionieren kann.



```
Private str As String

Public Property Let s(s As String)
    str = s
End Property

Public Property Get s() As String
    s = str
End Property
```

Wollte man nicht die Variable `str` über den Umweg der Property `Let/Get s` ansprechen wollen, könnte man im Prinzip die Klasse *events* auf folgenden Einzeiler reduzieren:

```
Public s As String
```

Da jedoch der Umweg über die Property `s` gewählt wurde, können als weiterer Schritt Events gefeuert werden. Zu diesem Zwecke können beliebige Event-Namen vergeben werden.

```
Public Event PropertyGet()
Public Event PropertyLet()

Private str As String

Public Property Let s(s As String)
    str = s
    RaiseEvent PropertyLet
End Property

Public Property Get s() As String
    s = str
    RaiseEvent PropertyGet
End Property
```

So viel zum Aufbau eines Klassenmoduls. Modulvariablen werden mit folgender Syntax deklariert:

```
[Public|Private|Dim] variable {, variable}
```

variable wird in dieser Syntax wie folgt deklariert

```
[WithEvents] varname[([subscripts])] [As [New] typename]
```

Zur Instanziierung obiger Klasse sei folgendes Statement angenommen:

```
Dim x as New events
```

Die Property s kann nun durch

```
x.s = "any string"
```

und

```
MsgBox (x.s)
```

gesetzt sowie ausgelesen werden. Das genügt jedoch noch nicht, um einen der obigen Events zu feuern. Um das zu erreichen, ist es notwendig, bei der Instanziierung obiger Klasse das Schlüsselwort `WithEvents` anzugeben. Dieses Schlüsselwort hat nur dann einen Sinn, wenn die Variable einerseits kein Array ist, andererseits in der instanziierten Klasse auch Events definiert wurden.

Unter Anwendung des Schlüsselworts `WithEvents` wird der entsprechende Event `PropertyLet` bzw. `PropertyGet` gefeuert, sobald nun die dazugehörige Property gesetzt oder abgefragt wird. Sofern die zu den Event-Deklarationen gehörigen Event-Handler (`x_PropertyGet()`, `x_PropertyLet()`) definiert sind, können gefeuerte Events abgefangen werden.

```
Private Sub x_PropertyGet()
    MsgBox "PropertyGet"
End Sub
```

Obiges Beispiel würde beim Setzen der Property *s* eine Message-Box mit dem Text *PropertyGet* öffnen und das Schreiben eines neuen Wertes in die Property mitteilen.

4.2 Mehrere Subscriber

Aufgaben im Bereich der Prozeßautomatisierung erfordern durchwegs die Fähigkeit auch mehrere Subscriber gleichzeitig zuzulassen. Der COM+-Ereignisdienst (Event-Service), der von Windows 2000 bereitgestellt wird, bietet dafür eine triviale Lösung an. Im Event-System von COM+ können sogenannte Event-Klassen eingerichtet werden, die im Event-Katalog registriert werden. Dadurch können Publisher an die Event-Klassen Nachrichten versenden. Subscriber wiederum können sich als Abonnenten von Event-Klassen registrieren lassen. COM+ sorgt beim Feuern eines Events dafür, daß der gefeuerte Event an alle eingetragenen Subscriber verteilt wird.

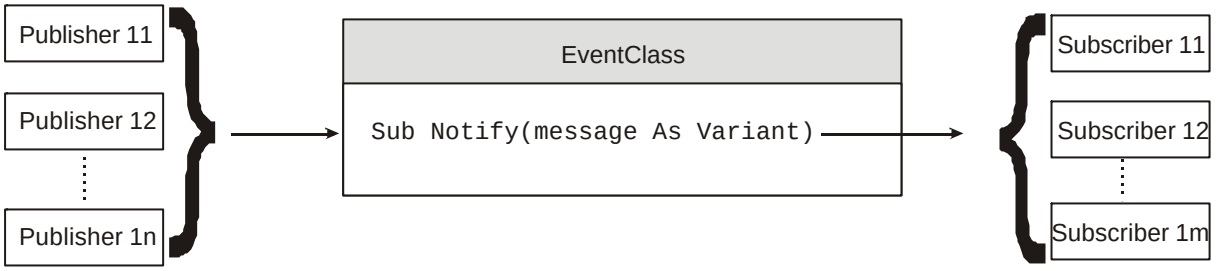


Abbildung 10: Event-Klasse mit Publisher und Subscriber

4.3 Einrichten der Event-Klasse

Event-Klassen lassen sich wahlweise durch IDL (ATL) oder Visual Basic erstellen. Die Vorgehensweise ist in beiden Fällen die gleiche. Es wird eine abstrakte Klasse mit beliebig vielen Methoden definiert. Unter Visual Basic wird eine neue Klasse erstellt, wobei das Klassenattribut `Instancing` auf `PublicNotCreatable` gesetzt wird. Die Methoden enthalten nur die Schnittstellendefinition, nicht jedoch die Implementierung. Die Implementierung bleibt den Subscribern vorenthalten. Eine Event-Klasse könnte also wie folgt aussehen:

```
Sub Notify(message As Variant)
' no implementation
End Sub
```

Nach dem Kompilieren des Codes erhält man eine DLL, die die Event-Klasse darstellt. Betrachtet man die IDL des DLL-Files, bekommt man die Schnittstellendefinition der Event-Klasse mit für den für Visual Basic-Programmierern verborgenen GUIDs zu sehen:

```
[
  uuid(870053C0-ACE8-4EBE-92EA-CA3B99E08748),
  version(1.0),
  helpstring("MyEvents")
]
library MyEvents
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-
000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _MyEventClass;

    [
      odl,
```

```
        uuid(F456C056-BF1D-4B95-8436-9D35A7C08140),
        version(1.0),
        hidden,
        dual,
        nonextensible,
        oleautomation
      ]
      interface _MyEventClass : IDispatch {
          [id(0x60030000)]
          HRESULT Notify([in, out] VARIANT* message);
      };

      [
        uuid(0A4B3975-F71B-4E0F-B0F7-7F7347E1D576),
        version(1.0)
      ]
      coclass MyEventClass {
          [default] interface _MyEventClass;
      };
  };
```

Abschließend muß die Event-Klasse im Component Services Snap-In registriert werden. Dazu sei Abbildung 9 aus Abschnitt 3.1.2 in Erinnerung gerufen. Im Gegensatz zu klassischen COM-Komponenten wird die Option *Install new event class(es)* ausgewählt und die entsprechende DLL angegeben. Im Component Services Snap-In lassen sich im übrigen auch Filter-Kriterien angeben, die festlegen, ob ein Event weitergeleitet werden soll oder nicht. Dieses Feature ist besonders bei großen Applikationen mit hohen Datenvolumina von Interesse.

4.4 Publisher

Publisher können also ab sofort Ereignisse feuern. Zu diesem Zweck wird ein Objekt der Event-Klasse instanziiert, welches entsprechenden Methodenaufruf ausführt.

```
o = New mydll.myclass
o.Notify ("Hello World")
Set o = Nothing
```

Der Methodenaufruf wird an die Event-Klasse geschickt, welcher anschließend an die Subscriber weitergeleitet wird.

4.5 Static Subscriber

Static Subscriber sind COM-Komponenten, die permanent im Speicher residieren. Sie werden auch genauso wie klassische COM-Komponenten im Component Services Snap-In registriert. Damit sie jedoch auch über gefeuerte Events informiert werden, bedarf es einer Änderung in im Properties-Dialog der Komponente. Unter dem Reiter *Advanced* muß im Feld Publisher ID der Name der Event-Klasse eingetragen werden, von der Events weitergeleitet werden sollen. Ab sofort ist die COM-Komponente ein static Subscriber.

So einfach auch das Erzeugen eines static Subscribers scheinen mag, müssen Programmierer beachten, welche Auswirkungen die wenigen Mausklicks auf das System haben! Sämtliche sichtbaren und unsichtbaren Informationen, die COM+ betreffen, werden von Windows 2000 im sogenannten COMAdminCatalog abgelegt. Dieser Katalog umfaßt sogenannte Collections, also Ansammlungen von Einstellungen, die abgefragt und manipuliert werden können. Jede Collection umfaßt hierbei jeweils genau einen Teilbereich zur Konfiguration von COM+-Komponenten. Wollte man nun eine registrierte Komponente als static Subscriber registrieren, müßte man die Collection SubscriptionsForComponent entsprechend manipulieren. Genau das macht nämlich auch das Component Services Snap-In.

Es wäre zu aufwendig, aus jeder im COMAdminCatalog enthaltenen Collection die Bedeutung jeder Property zu erklären. Hierbei sei auf entsprechende Dokumentation verwiesen. [MSD00]

4.6 Transient Subscriber

Wenn anstatt von permanent laufenden Prozessen temporär aufgerufene Applikationen als Subscriber von Event-Klassen auftreten, bedarf es einer Registrierung, die nicht mehr ganz so einfach ist wie bei den eben behandelten dauerhaften Subscribern. Im Klartext sollen Applikationen zur Laufzeit (meistens beim Aufruf) als Subscriber registriert und spätestens beim Beenden wieder unregistriert werden. Im weiteren werden derartige Applikationen als transient (temporäre) Subscriber bezeichnet.

Ein transient Subscriber kann entgegen den im Component Services Snap-In registrierten Prozessen als dauerhafte Subscriber in der Liste von COM+-Applikationen nicht hinzugefügt werden, da die Applikation als Subscriber nicht in die Collection SubscriptionsForComponent hinzugefügt werden kann. Transient Subscriber werden in der Collection

TransientSubscriptions abgelegt. Nachfolgendes Beispiel soll beispielhaft für das Arbeiten mit Collections zeigen, wie transient Subscriber im COMAdminCatalog registriert werden.

```
Private Sub Form_Load() ' register as transsubscriber
  Dim cat As COMAdmin.COMAdminCatalog
  Set cat = New COMAdmin.COMAdminCatalog

  Dim tsubs As COMAdmin.COMAdminCatalogCollection
  Set tsubs = cat.GetCollection("TransientSubscriptions")

  ' Add a new subscription
  Dim tsub As COMAdmin.COMAdminCatalogObject
  Set tsub = tsubs.Add
  tsub.Value("EventCLSID") = "{A7593AAF-93AD-4925-96AB-FB26AD359FD3}"

  tsub.Value("Name") = "My Sub"
  tsub.Value("SubscriberInterface") = Me
'  tsub.Value("MachineName") = "192.168.2.1"
  tsubs.SaveChanges

  ID = tsub.Value("ID")
End Sub
```

Um Zugriff auf die Klasse *COMAdmin* zu haben, muß die Bibliothek *COM + 1.0 Admin Type Library* in den References/Verweisen ausgewählt werden.

Solange Event-Klasse und transient Subscriber auf ein und demselben Rechner sind, funktioniert dieses System einwandfrei. Sobald jedoch der transient Subscriber auf einem anderen Rechner ausgeführt wird, muß dieser wissen, auf welchem Rechner die zu abonnierende Event-Klasse installiert ist.

Obwohl COM+ als durchaus ausgereiftes Framework betrachtet werden kann, birgt es gewisse Inkonsistenzen in seiner Architektur. In der für static Subscriber zuständigen Collection SubscriptionsForComponent gibt es nämlich interessanterweise die Property *MachineName*, die jenen Rechner identifiziert, von dem eine Event-Klasse abonniert werden soll. Trotz aller Ähnlichkeit verzichtet die Collection TransientSubscriptions jedoch auf genau diese Property.

Probiert man dennoch mit `tsub.Value("MachineName") = "192.168.2.1"` auf einen anderen Rechner zu verweisen, quittiert der Computer diesen Versuch beim Ausführen mit der Meldung *Invalid procedure call or argument*.

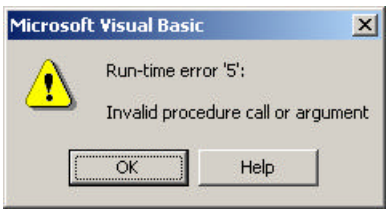


Abbildung 11: Fehlermeldung durch ungültiges Attribut

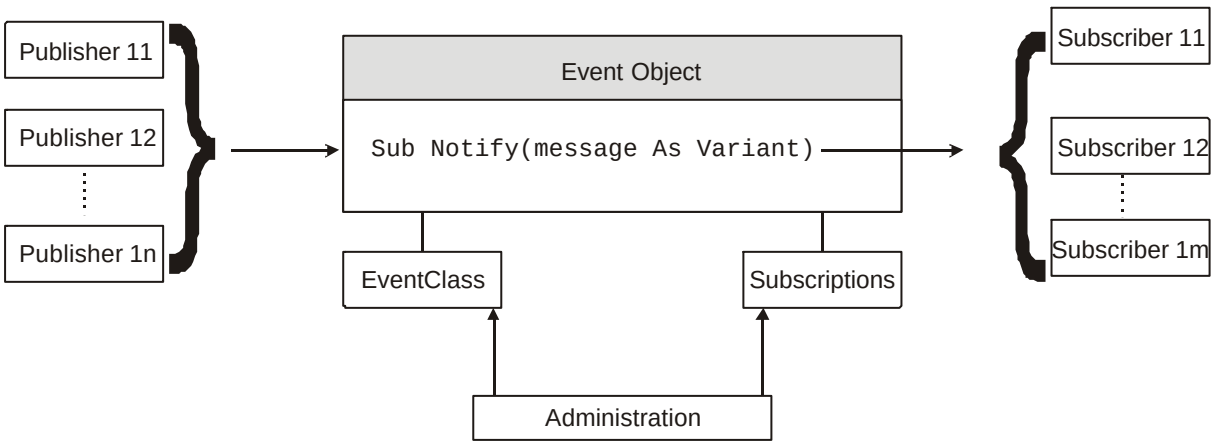
Das Entfernen von transienten Subscribern geht dafür wieder verhältnismäßig einfach.

```
Private Sub Form_Terminate() ' unregister as transsubscriber on close
form
  Dim cat As COMAdmin.COMAdminCatalog
  Set cat = New COMAdmin.COMAdminCatalog

  Dim tsubs As COMAdmin.COMAdminCatalogCollection
  Set tsubs = cat.GetCollection("TransientSubscriptions")
  tsubs.Populate

  Dim i As Long
  For i = 0 To tsubs.Count - 1
    If tsubs.Item(i).Value("ID") = ID Then
      tsubs.Remove i
      Exit For
    End If
  Next i

  tsubs.SaveChanges
End Sub
```



Transient Subscriber stoßen in ihrer Funktionalität jedoch schnell an ihre Grenzen. In verteilten Netzen ist es beispielsweise nicht möglich, sich als Subscriber für Events eines

anderen Rechners zu registrieren. Man kann zwar problemlos eine Event-Klasse als Proxy an andere Rechner weiter verteilen, jedoch können nur Events gefeuert werden. Als transient Subscriber wird man von einem auf einem anderen Rechner registrierten Event-System keine Events erhalten.

Folgendes Beispiel soll eine Lösung vorstellen, wie man das Problem umgehen kann, keine Events zu bekommen. Überlegung dabei ist, daß man eine Komponente erstellt, die als static Subscriber registriert wird. Diese wird ebenso wie andere Komponenten exportiert und als Proxy auf den entfernten Rechnern installiert, auf denen Subscriber laufen sollen.

```
Public Event Notify(message As Variant)

Implements MyEventClass

Private Sub MyEventClass_Notify(message As Variant)
  MsgBox "MyEventClass_Notify"
  RaiseEvent Notify(message)
End Sub
```

Wird nun ein Event gefeuert, wird der Event bis zu dem Prozeß weitergereicht. Es soll aber ein Ereignis kreiert werden, das über die Grenzen des Proxy/Stub-Layers durchgereicht werden kann. Dazu eignen sich klassische Events, die von interessierten Applikationen abgefragt werden können. Unschön an dieser Variante ist, daß für jeden static Subscriber eine neue Instanz dieser Komponente angelegt werden muß; eine konsistente Verwendung des Event-Systems ist jedoch nicht möglich. Es ist fraglich, ob das eine praktikable Lösung ist.

4.7 Erweiterte Funktionalität der Events

Grundsätzlich gibt es eine Menge an Attributen, die für Subscriptions vergeben werden können (`tsub.Value("MachineName") = "192.168.2.1"`), um ein maßgeschneidertes Event-System programmieren zu können.

Es ist jedoch an der Zeit, der Ursache der Fehlermeldung aus Abbildung 11 auf den Grund zu gehen. Es ist schon richtig, daß es das Attribut `MachineName` gibt. Was jedoch bis jetzt verschwiegen wurde ist die Tatsache, daß abhängig davon, ob static oder transient Subscriber nur unterschiedliche Attribute verfügbar sind! Dies ist selbst in der Fachliteratur nur schwer auf den ersten Blick ersichtlich. Für genauere Informationen über verfügbare Attribute und deren Mächtigkeit sei hiermit auf [`MSD00`] verwiesen.

5 Resource-Sharing

Sollen sich mehrere Prozesse oder Applikationen einen gemeinsamen Speicherbereich teilen, so ist das nicht trivial. Private Variablen in einer COM-Komponente haben das Problem, daß mit jeder Instanz einer Komponente ein eigener Speicherbereich angelegt wird. Was sich normalerweise als nützlich erweist, ist aber beim Teilen eines gemeinsamen Speicherbereichs durchaus unangenehm. Zu diesem Zweck gibt es sogenannte *Shared Properties*. Shared Properties sind nichts anderes als Eigenschaften, die über Prozeßgrenzen hinweg allen Prozessen zur Verfügung stehen. Dementsprechend können auch andere Rechner mittels DCOM auf Shared Properties zugreifen.

5.1 Shared Properties

Bevor nun gleich das Handling von Properties besprochen wird, sind einige grundlegende Überlegungen für das Arbeiten mit Shared Properties notwendig. Der *SharedPropertyGroupManager* (SPAM) ist eine Klasse, mit der *Shared Property Groups* angelegt und die spätere Zugriffsmethode darauf festgelegt werden. Anstatt eine Menge ungeordneter Properties organisiert der SPAM Properties in Shared Property Groups. Diese sind Organisationseinheiten, die jeweils eine beliebige Menge von Properties enthalten. Das macht Sinn, da jede Property Group für seine enthaltenen Properties einen internen Sperr-Manager beinhaltet. Dieser garantiert exklusive Sperren beim Zugriff auf die Properties.

Der SPAM speichert Informationen in Tupeln: zu jedem Namen gehört ein Wert, so daß Properties über vom Benutzer definierte Namen gelesen und geschrieben werden.

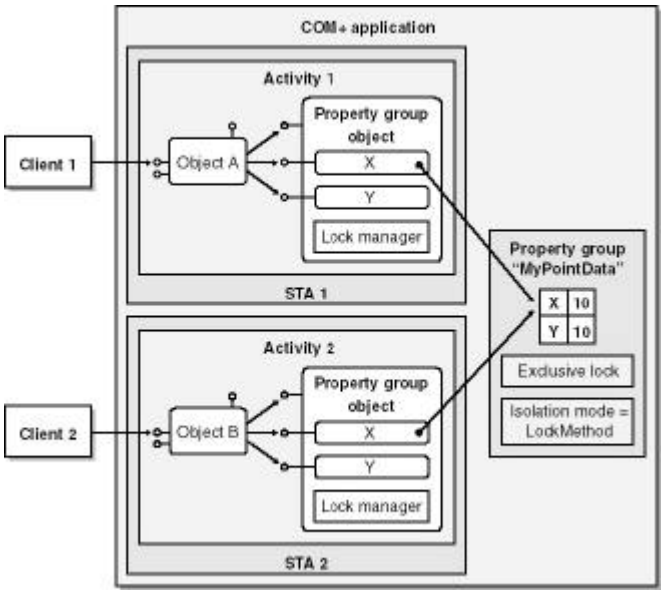


Abbildung 12: Architektur der Shared Properties

Voraussetzung dafür ist die Referenzierung der *COM+ Services Type Library (comsvcs.dll).* Es ist hier jedoch zu beachten, daß es die Idee der Shared Properties schon vor Windows 2000 gegeben hat. Hier muß jedoch auf die *Shared Property Manager Type Library (mtxspm.dll)* verwiesen werden.

```
Private PrivateSharedProperty As SharedProperty
Private SPG As SharedPropertyGroup

Private Sub Class_Initialize()
  ' Create SPG Manager
  Dim spgMgr As SharedPropertyGroupManager
  Set spgMgr = New SharedPropertyGroupManager
  ' Create and/or bind to shared property group
  Dim AlreadyExists As Boolean
  Set SPG = spgMgr.CreatePropertyGroup("MySPG", _
                                       LockMethod, _
                                       Process, _
                                       AlreadyExists)

' Create and/or bind to properties
  Set PrivateSharedProperty = SPG.CreateProperty("SharedProperty",
AlreadyExists)


  Set SP = New SharedPropertiesServer.SharedProperties
End Sub
```

Wenn obige Klasse instanziiert wird, so wird als erstes ein SharedPropertyGroupManager-Objekt angelegt. Dieses ist für die Verwaltung von Shared Property Groups zuständig. In obigem Beispiel erzeugt die Methode `CreatePropertyGroup` eine Property Group *MySPG*.

`LockMethod` ist einer von zwei Optionen, wie der Sperr-Mechanismus mit der betroffenen SPG umzugehen hat.

Konstante	Wert	Beschreibung
`LockSetGet`	0	Veranlaßt bei jedem Zugriff auf eine Property die Sperre derselben. Dadurch wird sichergestellt, daß Schreiben und Lesen gleichzeitig nicht möglich ist. Es ist jedoch möglich, auf andere Properties in der selben Property Group zeitgleich zuzugreifen. Diese Einstellung sollte bei Abhängigkeiten mehrerer Properties in einer Property Group untereinander nicht gewählt werden.
`LockMethod`	1	Sperrt exklusiv für den Aufrufer alle Properties in einer SPG beim Zugriff, solange die aufrufende Methode ausgeführt wird. Diese Einstellung wird bei Abhängigkeiten mehrerer Properties innerhalb einer Property Group gewählt.

Analog zur Konfiguration des Sperr-Mechanismus, läßt sich auch die Lebenszeit von SPGs beim Anlegen definierten.

Konstante	Wert	Beschreibung
`Standard`	0	Sobald sämtliche Verweise durch Applikationen und Prozesse auf die Property Group gelöscht sind, wird die Property Group automatisch zerstört.
`Process`	1	Die Property Group wird erst zerstört, wenn der erzeugende Prozeß terminiert. Sämtliche SharedPropertyGroup-Objekte müssen händisch auf `Nothing` gesetzt werden.

`AlreadyExists` ist ein Boolean Wert, der nach dem Aufruf der Methode `CreateProperty` wahr ist, wenn schon eine gleichnamige Property Group existiert. `False` bedeutet entsprechend, daß die betroffene Property Group durch den Methodenaufruf eben erst erstellt wurde.

Nach dem Anlegen der Property Groups können in diesen nun die eigentlichen Properties mit der Methode

```
Set PrivateSharedProperty = PropertyGroup.CreateProperty(name,
AlreadyExists)
```

hinzugefügt werden. `name` (String) identifiziert den Namen der betreffenden Property, `AlreadyExists` (Boolean) liefert analog zur Methode `CreatePropertyGroup` zurück, ob betreffende Property schon früher angelegt wurde.

Letztendlich kann, wie unten veranschaulicht, durch die Property `Value` der Wert von Properties gelesen und geschrieben werden.

```
Public Property Get SP() As Variant
  SP = PrivateSharedProperty.Value
End Property

Public Property Let SP(SP As Variant)
  PrivateSharedProperty.Value = SP
End Property
```

5.2 In Memory Data Base

Trotz großer Ankündigung hat Microsoft bei Windows 2000 gegenüber seinen release candidates auf die Auslieferung von der In Memory Data Base (IMDB) verzichtet. [MSD00] Sie hätte eine Schicht zwischen Client und Datenbankservern sein sollen, die als Cache Daten auf den Datenbankservern zwischenspeichert. Ziel sollte sein, Operationen die vorwiegend Daten lesen und nur selten schreiben beträchtliche Geschwindigkeitssteigerungen gegenüber klassischen Zugriffen auf Datenbankservern zu erreichen.

Zu diesem Zweck sollten häufig verwendete Datenbankelemente transparent im Hauptspeicher des Clientprozesses in der Mittelschicht zwischengelagert werden. Für den Client selbst hätte es keinen Unterschied gemacht, ob in der Mittelschicht eine IMDB steht oder sämtliche Operationen direkt an die Datenbank weitergeleitet werden. Prozesse in der Middleware greifen einfach nach Bedarf auf die IMDB zu. Die Kommunikationsarten DCOM, MSMQ sowie http via IIS wären nach wie vor unverändert möglich. [Pla99]

In [MSD00] erwähnt Microsoft ein einziges Mal den *Transactional Shared Property Manager* (TSPAM). Dabei scheint es sich um den in Abschnitt 5.1 Shared Property Manager zu handeln, der zusätzlich noch Transaktionen unterstützt. Es ist denkbar, daß der TSPAM als Erweiterung des SPAM für Datenbankanwendungen mit Windows 2000 nachfolgenden Betriebssystemen nachgeliefert wird. Konkrete Informationen dazu gibt es jedoch keine.

6 COM+ und IIS

Der Internet Information Server (IIS 5) bietet dem Webseiten-Gestalter die Möglichkeit, COM+-Komponenten in Active Server Pages einzubinden. Unabhängig von der Version von Windows 2000 müssen wahlweise bei der Installation von Windows 2000 oder nachträglich unter *Control Panel – Add/Remove Programs – Add/Remove Windows Components* die *Internet Information Services (IIS)* aktiviert werden. Ab der Installation findet man nun in der *Microsoft Management Console (mmc)* ein Snap-In für den Internet Information Server (siehe Abbildung 13).

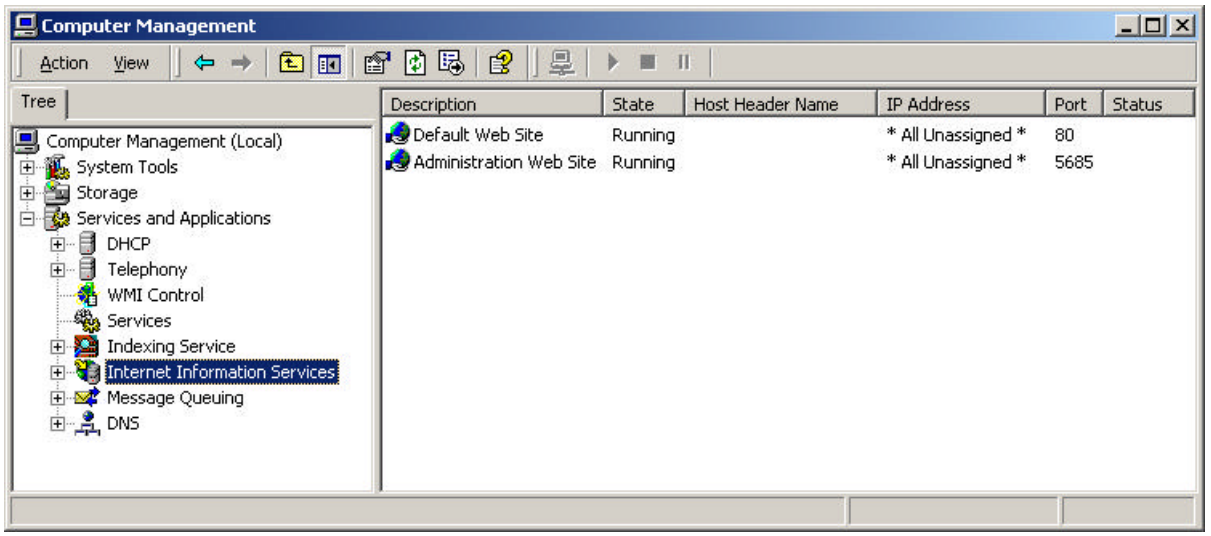


Abbildung 13: IIS Snap-In in der Microsoft Management Console

Sobald die gewünschte Web-Site eingerichtet wurde, kann man sich an die Programmierung seiner ersten Active Server Pages (ASPs) machen. Active Server Pages sind genau genommen statische HTML-Seiten, die um einen dynamischen Skript-Teil erweitert werden. Der dynamische Teil wird vom IIS bei Anforderung einer Seite interpretiert und dieser gibt inmitten von HTML-Tags etwaige Ergebnisse zurück. Folgendes Beispiel soll die Funktionsweise von ASP-Programmierung verdeutlichen:

```
<%@ LANGUAGE="VBSCRIPT"%>
<html><body>
<h2>customer list</h2>
<p>
<%
  dim sProgID, Wuff
  sProgID ="MyDog.Class1"
```

```
    set Wuff = Server.CreateObject(sprogid)
    Response.Write Wuff.who & Wuff.rollover
  %>
  </p>
  </body></html>
```

Ähnlich der Syntax aus Visual Basic werden im dynamischen Teil erstmals Variablen angelegt. Um einer Variablen ein Objekt zuweisen zu können, muß anstatt des *New*-Operators *Server.CreateObject(sProgID*[6]*)* verwendet werden. Anschließend steht über die Objektvariable der Zugriff auf alle öffentlichen Methoden und Properties offen. Bei obigem Beispiel liefern die Properties *who* und *rollover* von *Wuff* jeweils einen String zurück, der mittels *Response.Write* in das HTML-Dokument geschrieben wird.

Bei der Erstellung der DLL ist jedoch zu beachten, daß nur Apartment Threaded als Threading-Modell zulässig ist.

Problem bei dieser Vorgehensweise ist, daß das http ein zustandsloses Protokoll ist und somit bei jedem Aufruf einer ASP ein neues Klassenobjekt instanziiert wird. Es muß also ein Weg gesucht werden, wie man einen Zustand so lange halten kann, wie eine Web-Applikation in Verwendung ist. Im folgenden wird die Dauer einer Sitzung als Session bezeichnet. Idealerweise nimmt man sich Cookies zu Hilfe, die für jede Session den aktuellen Zustand zwischenspeichern. Zusätzlich dazu müssen noch Mechanismen in der dazugehörigen ActiveX DLL eingeführt werden, die ein Auslesen sowie Setzen eines Zustandes ermöglichen. Bei obigem Beispiel genügt es, eine Zählervariable vom Typ Integer zwischenzuspeichern.

```
  <%
    dim sProgID, Wuff, count
    sProgID ="MyDogServer.Wuff"

    count = session("count")                ' liest cookie aus, sofern
  vorhanden
    set Wuff = Server.CreateObject(sprogid)
    wuff.count = count                      ' reicht Zustand an DLL
  weiter
    Response.Write count & "<br>"
    Response.Write Wuff.who & ".<br>" & wuff.rollover
    Response.Write "</p><p>" & Wuff.who & ". <br>" & wuff.rollover
```

[6] Server Program Identifier

```
    session("count") = wuff.count            ' speichert aktuellen Zustand
in Cookie
%>
```

Was sich hier auf den ersten Blick trivial liest, ist in Wirklichkeit nicht ganz so einfach. Entgegen der Fachliteratur ist davon abzuraten, mittels `dim count as session` direkt auf ein Session-Objekt zuzugreifen. Die Folge sind nur unzählige Fehlermeldungen.

Wird nun eine Anfrage an den Webserver geschickt, sucht er nach Cookies, die eventuell schon im Speicher sind. Ist kein Cookie vorhanden, so verbleibt `count` im Nullzustand. Anschließend reicht der Webserver wie schon im vorigen Beispiel die Anfrage weiter und setzt den internen Zähler `cnt` in der Klasse Wuff auf den Wert von count. Vor dem Beenden des Scripts wird der aktuelle Zustand wieder zurück in ein Cookie geschrieben.

Voraussetzung dafür ist jedoch, daß der Web-Client Cookies akzeptiert. Andernfalls wird sich die Klasse Wuff bei jeder Seitenanfrage aufs neue im Nullzustand wiederfinden. Im allgemeinen wird für jede Session, d.h. für jedes Browserfenster, der jeweilige Zustand separat gespeichert. Es ist jedoch zu beachten, daß beim Erstellen eines Browserfensters durch Duplizieren (Strg-N beim Microsoft Internet Explorer) beide Fenster exakt das gleiche Cookie und somit immer den gleichen Zustand haben.

6.1 PHP

PHP gehört auch zu den Vertretern jener Sprachen zur Erstellung von HTML-Seiten, die mit COM arbeiten können. Ähnlich wie bei IIS in Verbindung mit Active Server Pages kann sich auch PHP nur eines eingeschränkten Bereichs der COM+-Funktionalität bedienen. Message Queuing oder die Unterstützung des Event-Systems von COM+ sind nicht möglich. Wenn von PHP im Zusammenhang mit COM/DCOM die Rede ist, muß auch klargestellt werden, daß die COM/DCOM-Einbindung nur auf Windows-Computern funktionieren kann. Das heißt daß PHPs Stärke als plattformunabhängige Sprache nicht ausgespielt werden kann.

Zum besseren Verständnis des folgenden Code-Beispiels sei vorausgeschickt, daß COM ein reservierter Klassenname in PHP ist. Instanzen werden davon erstellt, indem man den Namen der gewünschten COM-Komponente übergibt.

Folgende Funktionen sind grundsätzlich (außer in PHP 4) verfügbar:

- ☞ `int com_load(string component_name [, string server_name [, int codepage]])` erstellt eine neue COM-Komponente, wobei `component_name` der

Name der Komponente ist. Der return value vom Typ Integer wird bei einigen der nachfolgenden Funktionen als `com_identifier` dazu verwendet, die instanziierte COM-Komponente zu referenzieren. Bei einem Fehler ist dieser Wert `false`. `server_name` gibt den Rechner an, auf dem die Komponente installiert ist. `codepage` legt die Zeichencodierung fest und hat einen der möglichen Werte `CP_ACP`, `CP_MACCP`, `CP_OEMCP`, `CP_SYMBOL`, `CP_THREAD_ACP`, `CP_UTF7` und `CP_UTF8`.

- ☞ `new COM(string component_name)` bewirkt das gleiche wie `com_load`, nur wird ein COM-Objekt als return value zurückgeben.
- ☞ `mixed com_invoke(int com_identifier, string function_name [, mixed function parameters, ...])` ruft die Methode `function_name` einer COM-Komponente auf und gibt den Rückgabewert der Methode zurück.
- ☞ `mixed com_propget(resource com_identifier, string property)` liefert den Wert einer Property. Hat auch `com_get` als Alias.
- ☞ `void com_propput(resource com_identifier, string property, mixed value)` weist einen Wert der übergebenen Property zu. Hat auch `com_set` und `com_propset` als Alias.

Auch wenn man über die Sinnhaftigkeit des folgende Codebeispiels streiten mag, ist gut ersichtlich, wie mächtig PHP in Verbindung mit COM werden kann. PHP legt ein neues Winword-Objekt an,

```
<?
$word = new COM("Word.Application") or die("Kann Word.Application nicht
anlegen");
```

macht das Winword-Fenster jedoch unsichtbar,

```
$word->Visible = False;
```

erstellt ein neues Dokument von der Vorlage myTemplate.dot,

```
$word->Documents->Add("myTemplate.dot");
```

schließt wieder das neue Dokument ohne zu speichern

```
$doc = $word->ActiveDocument;
$doc->close(0);
```

und beendet letztendlich wieder Instanz von Winword.

```
$word->Quit();
?>
```

Unter Verwendung der oben beschriebenen Funktionen läßt sich obiges Beispiel auch so schreiben:

```
<?
$word = com_load("Word.Application") or die("Kann Word.Application nicht
anlegen");
com_set($word, "Visible", false);
com_invoke($word, "Documents.Add", "myTemplate.dot");
$doc = com_get($word, "ActiveDocument");
com_invoke($doc, "close", 0);
com_invoke($doc, "Quit");
?>
```

Auch in Zukunft wird Webservern unter Linux ein Umstieg auf Windows nicht erspart bleiben, um von der COM/DCOM-Unterstützung profitieren zu können. So ist es letztendlich unter Windows 2000 nur eine Frage der eigenen Religion, ob ein separater PHP-Prozessor installiert werden soll oder man nicht gleich auf die vorhandene Unterstützung der Active Server Pages zurückgreift. Sollte es jedoch in absehbarer Zeit gelingen COM/DCOM auf andere Plattformen zu portieren, würde der Einsatz von PHP im Vergleich zu ASP den Verzicht auf Windows 2000 mit IIS5 zweifelsohne erleichtern. [PHP01, Rat01, Sch99]

7 Message Queuing

Mit Message Queuing stellt Windows 2000 einen Dienst zur Verfügung, über den Komponenten asynchron miteinander kommunizieren können. Betrachtet man die Architektur von COM/DCOM, haben wir es dort nur mit synchronen Aufrufen zu tun. Das hat den Nachteil, daß das Client-Objekt so lange durch einen Aufruf blockiert ist bis dieser abgearbeitet ist.

Früher war es durchwegs üblich, daß Entwickler von COM/DCOM-Anwendungen eigene Mechanismen implementieren mußten, um asynchrone Kommunikation zwischen Komponenten zu ermöglichen.

7.1 Microsoft Transaction Server

Durch die Einführung des Microsoft Transaction Server (MTS) 1998 stellte Microsoft erstmals einen standardisierten Mechanismus für asynchrone Kommunikation (Microsoft Message Queuing – MSMQ) zur Verfügung. Einerseits erfordert die Verwendung von MSMQ eine Neuentwicklung bereits bestehender Programme, andererseits ist die Installation von MSMQ ein höchst kompliziertes Unterfangen, das sich weit komplizierter gestaltet als In Handbüchern nachzulesen.

Um in den Genuß der Funktionalität von Message Queuing zu kommen, müssen folgende Voraussetzungen gegeben sein: Entweder hat der Benutzer Windows NT 4.0 in einer Domain mit dazugehörigem SQL-Server oder Windows 2000 mit Active Directory. Zu beachten ist, daß der betroffene Computer nicht in einer Workgroup angemeldet sein kann.

7.2 Queued Components

Mit COM+ stellt Microsoft einen neuen Weg zur Verfügung, Message Queuing zu verwenden, ohne aufwendig Programme umschreiben zu müssen. COM+ trennt Kommunikation zwischen den Komponenten und MSMQ und stellt eine einheitliche Schnittstelle zur Verfügung, über die Methoden asynchron aufgerufen werden können. Von der Programmierung unterscheidet sich diese Methode nicht von klassischer COM/DCOM-Programmierung.

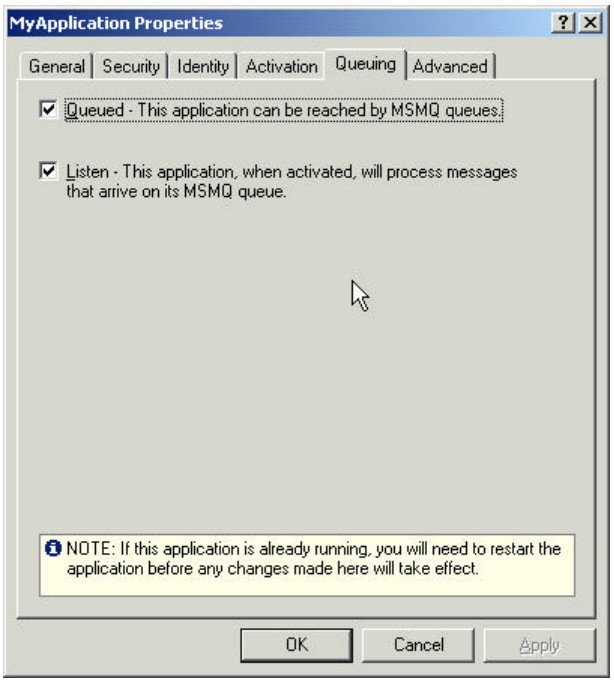


Abbildung 14: Component Services Snap-In – COM+ Application Properties

Um Message Queuing nutzen zu können, genügt die Aktivierung zweier Checkboxes in den Properties der betreffenden COM+ Application im Component Services Snap-In (siehe Abbildung 14). Dadurch wird zugleich eine Public Queue mit dem Namen der Applikation (hier *MyApplication*) angelegt. Einer Verwendung von Message Queuing steht nichts mehr im Weg.

8 .NET

Zweifelsohne ist es Microsoft mit COM+ bis zu einem gewissen Grad gelungen, eine Softwarearchitektur hervorzubringen, die mittels binärer Komponenten das Zusammenspiel von Quellcode verschiedenster Programmiersprachen ermöglicht. Auch war schon davon zu hören, daß Microsoft COM als offenen IETF[7]-Standard einführen wollte. DCOM kann für Softwareentwickler als guter Ansatz angenommen werden, verteilte Applikationen im Netzwerk zu erstellen ohne sich ausführlich mit der Protokollebene auseinandersetzen zu müssen. Der Einsatz von COM/DCOM in heterogenen Netzwerken wird jedoch an der fehlenden Unterstützung windows-fremder Betriebssysteme noch länger scheitern.

[c't01, Rat01]

[7] Internet Engineering Task Force

9 Index

10 Abbildungsverzeichnis

11 Glossar

☞ GUID	Globally Unique Identifier
☞ IID	interface ID
☞ CLSID	class ID
☞ IDL	Interface Definition Language
☞ UDT	user defined type
☞ SCM (scum)	Service Control Manager
☞ TLS	thread-local storage (-> [Pat00], p. 194)
☞ MTA	multithreaded apartment (->[Pat00], p. 195)
☞ STA	singlethreaded apartment
☞ SID	security ID
☞ WAM	Web Application Manager

12 Quellenverzeichnis

[c't01] c't – magazin für computertechnik, Verlag Heinz Heise, Hannover, 06/2001

[Dev01] 4Developers / COM Explorer, http://www.4dev.com

[Gor00] The COM and COM+ Programming Primer, Gordon A., Prentice Hall PTR, Upper Saddle River, 2000, ISBN 0-13-085032-2

[Lei00] COM+ Unleashed, Leinecker R., Sams Publishing, Indiana, 2000, ISBN 0-672-31887-3

[Pla99] COM+ verstehen, Platt D., Microsoft Press, Redmond, 1999, ISBN 3-86063-610-3

[Box98] Essential COM, Box D., Addison Wesley, ???, 1998, ISBN 0-201-63446-5

[Edd98] Inside Distributed COM, Eddon G., Eddon H., Microsoft Press, Redmond, 1998, ISBN 1-57231-849-X

[Gro00] Windows DNA, Gross Ch., Galileo Press, Bonn, 2000, ISBN 3-934358-75-6

[MSD00] MSDN Library, July 2000

[Sch99] php: dynamische webauftritte professionell realisieren, Schmid E. et.al., Markt und Technik, München, 1999, ISBN 3-8272-5524-4

[PHP01] PHP: Hypertext Preprocessor, http://www.php.net

[Pat00] Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0, Pattison T., Microsoft Press, Redmond, 2000, ISBN 0-7356-1010-X

[Rat01] Webanwendungen mit PHP 4.0 entwickeln, Ratschiller T. et.atl., Addison-Wesley, München, 2001, ISBN 3-8273-1730-4

[Obe00] Understanding & Programming COM+, Oberg R., Prentice Hall PTR, Upper Saddle River, 2000, ISBN 0-13-023114-2

[Fre00] Visual Basic Developer's guide to COM and COM+, Freeze W., Sybex, San Francisco, 2000, ISBN 0-7821-2558-1

Eidesstattliche Erklärung

Ich erkläre an Eides Statt, daß ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, im Oktober 2002

Curriculum Vitae

Name: Christoph Emsenhuber

Geburtsdatum und Geburtsort:	4. Dezember 1977 in Vöcklabruck, Österreich
Staatsbürgerschaft:	Österreich
Familienstand:	ledig

1984 – 1988
Volksschule 42, Linz

1988 – 1990
Bundesrealgymnasium Linz Auhof

1990 – 1996
Bundesrealgymnasium Linz Auhof, Schulversuch Informatik, abschließend Reifeprüfung

01.07.1993 – 30.09.1993
Privatbüro Dipl.-Ing. Hubert Lohr, Lenzing, Österreich

11.07.1994 – 05.08.1994
Volkskreditbank AG, Linz, Österreich

10.08.1994 – 15.10.1994
Privatbüro Dipl.-Ing. Hubert Lohr, Lenzing, Österreich

10.07.1995 – 04.08.1995
Volkskreditbank AG, Linz, Österreich

08.07.1996 – 02.08.1996
Volkskreditbank AG, Linz, Österreich

aufrechtes Dienstverhältnis seit 01.08.1996
Dr. med. univ. Hubert-Hermann Lohr, praktischer Arzt, Lenzing, Österreich

1996 – 2000
Studium der Datentechnik an der Johannes Kepler Universität Linz
Abschluß mit der Berufsbezeichnung *Akademisch geprüfter Datentechniker*

1996 – 2002
Studium der Informatik an der Johannes Kepler Universität Linz

07.07.1997 – 01.08.1997
Oberösterreichische Kraftwerke AG - Fernwärme Vöcklabruck, Österreich

13.07.1998 – 07.08.1998
Oberösterreichische Kraftwerke AG, Linz, Österreich

12.07.1999 – 06.08.1999
Energie AG Oberösterreich, Linz, Österreich

Appendix

☞ \helloworld\IHelloWorld.cls

```
Option Explicit

Implements IHelloWorld

Private Function IHelloWorld_SayHello() As String
    IHelloWorld_SayHello = "Hello world!"
End Function

Private Function IHelloWorld_SayAnything(message As String) As String
    IHelloWorld_SayAnything = "Hello world!" & vbCrLf & message
End Function
```

☞ helloworld\HelloWorld.cls

```
Option Explicit

Public Function SayHello() As String
End Function

Public Function SayAnything(message As String) As String
End Function
```

☞ helloworld\HelloWorldClient.frm

```
Option Explicit

Dim HW As IHelloWorld

Private Sub SayHelloBtn_Click()
    MsgBox HW.SayHello
End Sub

Private Sub SayAnythingBtn_Click()
    MsgBox HW.SayAnything(Text1.Text)
End Sub

Private Sub Form_Load()
    Set HW = New HelloWorld
End Sub
```

```
Private Sub Form_Terminate()
    Set HW = Nothing
End Sub
```

```
// 02.idl : IDL-Quellcode für 02.dll
//

// Diese Datei wird mit dem MIDL-Tool bearbeitet,
// um den Quellcode für die Typbibliothek (02.tlb)und die
Abruffunktionen zu erzeugen.
```

☞ helloworld\atl\02.idl

```
import "oaidl.idl";
import "ocidl.idl";
   [
         object,
         uuid(EF84B118-BF8E-4810-85C2-27C707CC0A1C),
         dual,
         helpstring("IHelloWorld-Schnittstelle"),
         pointer_default(unique)
   ]
   interface IHelloWorld : IDispatch
   {
         [id(1), helpstring("Methode SayHello")] HRESULT SayHello();
   };

[
   uuid(6775B906-3EAE-49C1-AD74-00CE91180F8B),
   version(1.0),
   helpstring("02 1.0 Typbibliothek")
]
library MY02Lib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   [
         uuid(B06A7CC8-7553-48C5-B2F1-C867E823DB0D),
         helpstring("HelloWorld Class")
   ]
   coclass HelloWorld
```

```
    {
          [default] interface IHelloWorld;
    };
};
```

☞ \atl\06\06.cpp

```
// 06.cpp : Implementierung der DLL-Exporte.


// Hinweis: Proxy/Stub-Information
//      Um eine eigene Proxy/Stub-DLL zu erstellen,
//      führen Sie nmake -f 06ps.mk im Projektverzeichnis aus.

#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "06.h"

#include "06_i.c"
#include "SimpleInterface.h"


CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_SimpleInterface, CSimpleInterface)
END_OBJECT_MAP()

////////////////////////////////////////////////////////////////////////
/////
// DLL-Einsprungpunkt

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID
/*lpReserved*/)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        _Module.Init(ObjectMap, hInstance, &LIBID_MY06Lib);
        DisableThreadLibraryCalls(hInstance);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
```

```
        _Module.Term();
    return TRUE;    // OK
}

/////////////////////////////////////////////////////////////////////////
/////
// Verwendet, um zu entscheiden, ob die DLL von OLE aus dem Speicher
entfernt werden kann

STDAPI DllCanUnloadNow(void)
{
    return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}

/////////////////////////////////////////////////////////////////////////
/////
// Liefert eine Klassenfabrik zurück, um ein Objekt des angeforderten
Typs anzulegen

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    return _Module.GetClassObject(rclsid, riid, ppv);
}

/////////////////////////////////////////////////////////////////////////
/////
// DllRegisterServer - Fügt der Systemregistrierung Einträge hinzu

STDAPI DllRegisterServer(void)
{
    // Registriert Objekt, Typelib und alle Schnittstellen in Typelib
    return _Module.RegisterServer(TRUE);
}

/////////////////////////////////////////////////////////////////////////
/////
// DllUnregisterServer - Entfernt Einträge aus der Systemregistrierung

STDAPI DllUnregisterServer(void)
{
    return _Module.UnregisterServer(TRUE);
}
```

☞ \atl\06\06.idl

```
// 06.idl : IDL-Quellcode für 06.dll
//

// Diese Datei wird mit dem MIDL-Tool bearbeitet,
// um den Quellcode für die Typbibliothek (06.tlb)und die
Abruffunktionen zu erzeugen.

import "oaidl.idl";
import "ocidl.idl";
   [
         object,
         uuid(CBD14D27-B8FA-41EA-97B3-E75674041701),
         dual,
         helpstring("ISimpleInterface-Schnittstelle"),
         pointer_default(unique)
   ]
   interface ISimpleInterface : IDispatch
   {
         [propget, id(1), helpstring("Eigenschaft myLong")] HRESULT
myLong([out, retval] long *pVal);
         [propput, id(1), helpstring("Eigenschaft myLong")] HRESULT
myLong([in] long newVal);
   };

[
   uuid(9473AB20-3E5E-4FEF-90FB-39DE7DC17F34),
   version(1.0),
   helpstring("06 1.0 Typbibliothek")
]
library MY06Lib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   [
         uuid(00C22CC1-04EC-4AF1-A678-D5BE468D4957),
         helpstring("_ISimpleInterfaceEreignisschnittstelle")
   ]
   dispinterface _ISimpleInterfaceEvents
```

```
    {
        properties:
        methods:
    };

    [
        uuid(09F9F24D-09CA-49AD-B256-0EFFFF7278D7),
        helpstring("SimpleInterface Class")
    ]
    coclass SimpleInterface
    {
        [default] interface ISimpleInterface;
        [default, source] dispinterface _ISimpleInterfaceEvents;
    };
};
```

☞ \atl\06\SimpleInterface.h

```
// SimpleInterface.h : Deklaration von CSimpleInterface

#ifndef __SIMPLEINTERFACE_H_
#define __SIMPLEINTERFACE_H_

#include "resource.h"       // Hauptsymbole

/////////////////////////////////////////////////////////////////////////////
/////
// CSimpleInterface
class ATL_NO_VTABLE CSimpleInterface :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CSimpleInterface, &CLSID_SimpleInterface>,
   public IConnectionPointContainerImpl<CSimpleInterface>,
   public IDispatchImpl<ISimpleInterface, &IID_ISimpleInterface,
&LIBID_MY06Lib>
{
public:
   CSimpleInterface()
   {
        pLong = 0;
   }

DECLARE_REGISTRY_RESOURCEID(IDR_SIMPLEINTERFACE)
```

```
DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CSimpleInterface)
   COM_INTERFACE_ENTRY(ISimpleInterface)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()
BEGIN_CONNECTION_POINT_MAP(CSimpleInterface)
END_CONNECTION_POINT_MAP()


// ISimpleInterface
public:
   STDMETHOD(get_myLong)(/*[out, retval]*/ long *pVal);
   STDMETHOD(put_myLong)(/*[in]*/ long newVal);
private:
   long pLong;
};

#endif //__SIMPLEINTERFACE_H_
```

☞ \events\events.cls

```
Public Event PropertyGet()
Public Event PropertyLet()

Private str As String

Public Property Let s(s As String)
    str = s
    RaiseEvent PropertyLet
End Property

Public Property Get s() As String
    s = str
    RaiseEvent PropertyGet
End Property
```

☞ \events\EventClient.frm

```
Dim WithEvents muh As Events

Private Sub Form_Load()
    Set muh = New Events
End Sub
```

```
Private Sub get_Click()
    Text2.Text = muh.s
End Sub

Private Sub let_Click()
    muh.s = Text1.Text
End Sub

Private Sub muh_PropertyChanged()
End Sub

Private Sub muh_PropertyGet()
    MsgBox "PropertyGet"
End Sub

Private Sub muh_PropertyLet()
    MsgBox "PropertyLet"
End Sub
```

☞ \framework\events\Events1.cls

```
Option Explicit

Sub Notify(message As Variant)
' no implementation
End Sub
```

☞ \framework\events\Events2.cls

```
Option Explicit

Implements MyEventClass

Private ID As String

Public Event Notify(message As Variant)


Private Sub Class_Initialize()
  Dim cat As COMAdmin.COMAdminCatalog
  Set cat = New COMAdmin.COMAdminCatalog

  Dim tsubs As COMAdmin.COMAdminCatalogCollection
```

```
    Set tsubs = cat.GetCollection("TransientSubscriptions")

    ' Add a new subscription
    Dim tsub As COMAdmin.COMAdminCatalogObject
    Set tsub = tsubs.Add
    tsub.Value("EventCLSID") = "{0A4B3975-F71B-4E0F-B0F7-7F7347E1D576}"


    tsub.Value("Name") = "My Sub2"
    tsub.Value("SubscriberInterface") = Me

    tsubs.SaveChanges

    ID = tsub.Value("ID")
  End Sub

  Private Sub Class_Terminate()
    Dim cat As COMAdmin.COMAdminCatalog
    Set cat = New COMAdmin.COMAdminCatalog

    Dim tsubs As COMAdmin.COMAdminCatalogCollection
    Set tsubs = cat.GetCollection("TransientSubscriptions")
    tsubs.Populate

    Dim i As Long
    For i = 0 To tsubs.Count - 1
      If tsubs.Item(i).Value("ID") = ID Then
        tsubs.Remove i
        Exit For
      End If
    Next i

    tsubs.SaveChanges
  End Sub

  Private Sub MyEventClass_Notify(message As Variant)
  '    MsgBox "MyEventClass_Notify"
      RaiseEvent Notify(message)
  End Sub
```

☞ \framework\publisher\publisher.frm

```
  Dim WithEvents muh As Events
```

```
Private Sub Form_Load()
    Set muh = New Events
End Sub

Private Sub get_Click()
    Text2.Text = muh.s
End Sub

Private Sub let_Click()
    muh.s = Text1.Text
End Sub

Private Sub muh_PropertyChanged()
End Sub

Private Sub muh_PropertyGet()
    MsgBox "PropertyGet"
End Sub

Private Sub muh_PropertyLet()
    MsgBox "PropertyLet"
End Sub
```

☞ \framework\events\events1.idl

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Events1.dll

[
  uuid(870053C0-ACE8-4EBE-92EA-CA3B99E08748),
  version(1.0),
  helpstring("MyEvents")
]
library MyEvents
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-
000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _MyEventClass;
```

```
    [
      odl,
      uuid(F456C056-BF1D-4B95-8436-9D35A7C08140),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
    ]
    interface _MyEventClass : IDispatch {
        [id(0x60030000)]
        HRESULT Notify([in, out] VARIANT* message);
    };

    [
      uuid(0A4B3975-F71B-4E0F-B0F7-7F7347E1D576),
      version(1.0)
    ]
    coclass MyEventClass {
        [default] interface _MyEventClass;
    };
};
```

☞ \framework\events\events2.idl

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Events2.dll

[
  uuid(129AB118-A233-49B8-B439-4B978FEB1C51),
  version(1.0),
  helpstring("MyEventProcess")
]
library MyEvents2
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-
000000000046}
    importlib("stdole2.tlb");
    // TLib : MyEvents : {870053C0-ACE8-4EBE-92EA-CA3B99E08748}
    importlib("Events1.dll");
```

```
    // Forward declare all types defined in this typelib
    interface _MyEventProcess;
    dispinterface __MyEventProcess;

    [
      odl,
      uuid(59CE7BB8-9105-4CA7-A1A3-824D83388C00),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
    ]
    interface _MyEventProcess : IDispatch {
    };

    [
      uuid(A05E0AE4-EF34-4699-B784-40CA9AD69F89),
      version(1.0)
    ]
    coclass MyEventProcess {
        [default] interface _MyEventProcess;
        interface _MyEventClass;
        [default, source] dispinterface __MyEventProcess;
    };

    [
      uuid(446B53E7-0B13-42AC-8B39-A0B63A23C3BA),
      version(1.0),
      hidden,
      nonextensible
    ]
    dispinterface __MyEventProcess {
        properties:
        methods:
            [id(0x00000001)]
            void Notify([in, out] VARIANT* message);
    };
};
```

☞ \framework\transsubscriber\transsubscriber.frm

```
Option Explicit
```

```
Private ID As String
Implements MyEventClass

Private Sub Form_Terminate() ' unregister as transsubscriber on close
form
  Dim cat As COMAdmin.COMAdminCatalog
  Set cat = New COMAdmin.COMAdminCatalog

  Dim tsubs As COMAdmin.COMAdminCatalogCollection
  Set tsubs = cat.GetCollection("TransientSubscriptions")
  tsubs.Populate

  Dim i As Long
  For i = 0 To tsubs.Count - 1
    If tsubs.Item(i).Value("ID") = ID Then
      tsubs.Remove i
      Exit For
    End If
  Next i

  tsubs.SaveChanges
End Sub

Private Sub Form_Load() ' register as transsubscriber
    Dim cat As COMAdmin.COMAdminCatalog
    Set cat = New COMAdmin.COMAdminCatalog

    Dim tsubs As COMAdmin.COMAdminCatalogCollection
    Set tsubs = cat.GetCollection("TransientSubscriptions")

    ' Add a new subscription
    Dim tsub As COMAdmin.COMAdminCatalogObject
    Set tsub = tsubs.Add
    tsub.Value("EventCLSID") = "{0A4B3975-F71B-4E0F-B0F7-7F7347E1D576}"

    tsub.Value("Name") = "My Sub"
    tsub.Value("SubscriberInterface") = Me
    tsubs.SaveChanges

    ID = tsub.Value("ID")
```

```
'    ' get last value from database
'    Dim SP As New SharedPropertiesServer.SharedProperties
'    Dim data As PropertyBag
'
'    MyTxtBox.Text = SP.ReadProperty("text", "nix test")
'    frmMain.Caption = "Transient Subscriber " & Len(MyTxtBox)
'
'    Set SP = Nothing
End Sub

Private Sub MyEventClass_Notify(message As Variant)
    Dim data As PropertyBag
    Set data = New PropertyBag

    data.Contents = message
    MyTxtBox.Text = data.ReadProperty("text", "nix text") ' & vbCrLf &
Now - data.ReadProperty("timestamp", "nix zeit")

    frmMain.Caption = "Subscriber " & Len(MyTxtBox)

    Set data = Nothing
End Sub
```

☞ \SharedProperties\SharedProperties.cls

```
Option Explicit

Private PrivateSharedProperty As SharedProperty
Private SPG As SharedPropertyGroup

Public Event PropertyGet()
Public Event PropertyLet()

Private Sub Class_Initialize()
  ' Create SPG Manager
  Dim spgMgr As SharedPropertyGroupManager
  Set spgMgr = New SharedPropertyGroupManager
  ' Create and/or bind to shared property group
  Dim AlreadyExists As Boolean
  Set SPG = spgMgr.CreatePropertyGroup("SharedPropertyGroup", _
                                       LockMethod, _
                                       Process, _
                                       AlreadyExists)
```

```
  ' Create and/or bind to properties
  Set PrivateSharedProperty = SPG.CreateProperty("SharedProperty",
AlreadyExists)
End Sub

Public Property Get SP() As Variant
    SP = PrivateSharedProperty.Value
    RaiseEvent PropertyGet
End Property

Public Property Let SP(SP As Variant)
    PrivateSharedProperty.Value = SP
    RaiseEvent PropertyLet
End Property
```

☞ \SharedProperties\SharedPropertiesClient.frm

```
Option Explicit

Private WithEvents SP As SharedPropertiesServer.SharedProperties

Private Sub Form_Load()
  Set SP = New SharedPropertiesServer.SharedProperties
End Sub

Private Sub get_Click()
    SharedText.Text = SP.SP
End Sub

Private Sub SharedText_Change()
'    MsgBox "SharedText_Change()"
    SP.SP = SharedText.Text
End Sub

Private Sub SP_PropertyLet()
    Debug.Print "SP_PropertyLet"
    If SharedText.Text <> SP.SP Then
        SharedText.Text = SP.SP
        Debug.Print "PropertyChange"
    End If
End Sub
```

☞ \SharedDBConnect\SharedDBConnectServer.cls

```
Option Explicit

Const undefined = -1

Private SPG As SharedPropertyGroup

Private cnn As SharedProperty
Private rst As SharedProperty
Private rst2 As SharedProperty


Implements MyEventClass

Private Declare Function timeGetTime Lib "winmm.dll" () As Long

Private Sub Class_Initialize()
    Dim AlreadyExists As Boolean
    ' Create SPG Manager
    Dim spgMgr As SharedPropertyGroupManager
    Set spgMgr = New SharedPropertyGroupManager
    ' Create and/or bind to shared property group
    Set SPG = spgMgr.CreatePropertyGroup("SharedDB", _
                                         LockSetGet, _
                                         Process, _
                                         AlreadyExists)

    ' Create and/or bind to properties
    Set cnn = SPG.CreateProperty("SharedDB.cnn", AlreadyExists)
    Set rst = SPG.CreateProperty("SharedDB.rst", AlreadyExists)
    Set rst2 = SPG.CreateProperty("SharedDB.rst2", AlreadyExists)

    If Not AlreadyExists Then ' create new database connection
        cnn.Value = New ADODB.Connection
        cnn.Value.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=P:\christoph\Eigene
Dateien\diplomarbeit\prgs\framework\SharedDBConnect\db1.mdb;"
        cnn.Value.Open

        rst.Value = New ADODB.RecordSet
        rst.Value.Open "MyTable", cnn.Value, adOpenKeyset,
adLockOptimistic
```

```
            ' log creation of new DB connection
            rst2.Value = New ADODB.RecordSet
            rst2.Value.Open "MyLog", cnn.Value, adOpenKeyset,
    adLockOptimistic

            WriteLog ("Neue Instanz erstellt")
            WriteLog ("Neuer DB-Connect erstellt")
        Else
            WriteLog ("Neue Instanz erstellt")
        End If
    End Sub

    Private Sub WriteLog(str As String)
        Dim field As field

        rst2.Value.AddNew

        For Each field In rst2.Value.Fields
            If field.Name = "event" Then field.Value = str
    ' ereignis aufzeichnen
            If field.Name = "timeGetTime" Then field.Value = timeGetTime
    ' timestamp
        Next
        rst2.Value.Update
    End Sub

    Public Sub WriteValues(RecordSet As Variant)
        Dim data As New PropertyBag
        Dim field As field
        Dim tmp As Variant

        rst.Value.AddNew

        data.Contents = RecordSet


        For Each field In rst.Value.Fields ' nur "text"
            tmp = data.ReadProperty(field.Name, undefined)
            If tmp <> undefined Then
                field.Value = tmp
            End If
        Next
```

```
    rst.Value.Update
End Sub

Private Sub Class_Terminate()
    WriteLog ("Instanz wird zerstört")
End Sub

Private Sub MyEventClass_Notify(message As Variant)
    WriteValues (message)
End Sub
```

☞ \SharedDBConnect\Form1.frm

```
Private dummy As SharedDBConnect.MySharedProperties

Private Sub Form_Load()
    Set dummy = New MySharedProperties
End Sub

Private Sub Form_Terminate()
    Set dummy = Nothing
End Sub

Private Sub Text1_Change()
    Dim data As New PropertyBag

    data.WriteProperty "text", Text1.Text
    dummy.WriteValues data.Contents
End Sub
```

☞ \mydog\MyDogServer.cls

```
Option Explicit

Private ctr As Integer

Public Property Get who() As String
  ctr = ctr + 1
  who = "Ich bin die Methode who aus der COM+-Applikation
MyDogServer.Wuff - immerhin habe ich schon " & ctr & " mal gebellt"
End Property

Public Property Get RollOver() As String
  RollOver = "Jaja, auch Wuff-Objekte beherrschen perfekt nen
Überschlag"
```

```
End Property

Public Property Let count(counter As Integer)
    ctr = counter
End Property

Public Property Get count() As Integer
    count = ctr
End Property
```

☞ \mydog\MyDogClient.frm

```
Option Explicit

Private Hund As Wuff

Private Sub Form_Load()
    Set Hund = New Wuff
    MsgBox "form wurde gerade geöffnet"
End Sub

Private Sub who_Click(Index As Integer)
    MsgBox Hund.who
End Sub

Private Sub rollover_Click(Index As Integer)
    MsgBox Hund.rollover
End Sub
```

☞ \mydog\asp01.asp

```
<%@ LANGUAGE="VBSCRIPT"%>
<html><TITLE>auf den hund gekommen</TITLE>
<body>
<h1>auf den hund gekommen</h1>
<p>
<%
   dim sProgID, Wuff, count

   sProgID ="MyDogServer.Wuff"

' erzeugt ein temporäres cookie, das beim verlassen der website
(eigentlich schließen des browsers) gelöscht wird
```

```
' cookie speichert den zwischenstand der session, da ansonsten bei jedem
request wegen der zustandslosigkeit des http
' der counter i in class1 wieder auf 0 initialisiert wird :(((
' möglicherweise ein problem auf server-farmen???
   count = session("count")

   set Wuff = Server.CreateObject(sprogid)

   wuff.count = count

   Response.Write count & "<br>"

   Response.Write Wuff.who & ".<br>" & wuff.rollover
   Response.Write "</p><p>" & Wuff.who & ". <br>" & wuff.rollover
   session("count") = wuff.count

%>
</p>
</body></html>
```

☞ \framework\http\asp01.asp

```
<%@ LANGUAGE="VBSCRIPT"%>
<html><TITLE>SharedProperties lesen</TITLE>
<body>
<h1>SharedProperties lesen</h1>

<p><a href="javascript:location.reload()" target="_self"><img border=0
src="refresh.bmp" width="13" height="16"
alt="refresh.bmp (1.334 Bytes)" align="middle"> Aktualisieren</a></p>

<p><%
   dim SP
   set SP =
Server.CreateObject("SharedPropertiesServer.SharedProperties")
   Response.Write (SP.ReadProperty("text")) & "<br>"
   ' jetzt einen wert lesen, der nicht gesetzt wird
   Response.Write (SP.ReadProperty("123")) & "<br>"
   Response.Write (SP.ReadProperty("123", "any default-string")) &
"<br>"
   Response.Write (SP.ReadProperty("timestamp"))
   set SP = Nothing
%></p>
```

```
</body></html>
```

☞ \framework\http\asp02.asp

```
<%@ LANGUAGE="VBSCRIPT"%>
<html><TITLE>SharedProperties lesen</TITLE>
<body>
<h1>SharedProperties lesen</h1>

<FORM action="asp01.asp" method=POST id=form1 name=form1>
<p><INPUT type="text" name="test" value="<%
   dim SP, o
   set SP =
Server.CreateObject("SharedPropertiesServer.SharedProperties")
   Response.Write SP.SP
%>"> <INPUT type="submit" name=submit>
</p>
</FORM>

<p>
test == <%
   if Request.Form("test")="" then
         Response.Write "Nothing"
   else
         response.write Request.Form("test") &  Request.Form("test") &
"<br> neuen Wert in SPG schreiben"

       Set o = Server.CreateObject("MyEvents.MyEventClass")
       o.Notify Request.Form("test")
'      Set o = Nothing
   end if

'  set SP = Nothing
%>
</p>

<p><a href="http://archimedes/framework/asp01.asp" target="_self"><img
border=0 src="refresh.bmp" width="13" height="16"
alt="refresh.bmp (1.334 Bytes)" align="middle"> Aktualisieren</a></p>
</body></html>
```

☞ \IsDSEnabled\IsDSEnabled.cls

```
Option Explicit
```

```
Dim myapp As New MSMQApplication

Sub main()
    If myapp.isdsenabled Then
      MsgBox "i am in directory mode"
    Else
      MsgBox "i am NOT in directory mode"
    End If
End Sub
```

☞ \overloading\01.frm

```
Private Sub Form_Load()

End Sub

Public Sub Notify(message As String)

End Sub

Public Sub Notify(message As Integer)

End Sub
```