4 Softwarearchitektonische Determinanten

Dieses Kapitel beschreibt die softwarearchitektonischen Grundlagen der später vorgestellten Entwürfe von Optimierungssystemschnittstellen. Hierzu wird zunächst in allgemeiner Form auf prozedurale und objektorientierte Programmierparadigmen eingegangen, und in einem zweiten Schritt werden Details entsprechender Schnittstellen- und Framework-Implementationen unter Windows dargestellt. Insbesondere sollen die Grundlagen der Komponentenframeworks COM und .NET skizziert werden.

4.1 Prozedurale und objektorientierte Programmierparadigmen

Im Lauf der letzten ungefähr 50 Jahre hat sich bekanntermaßen nicht nur die Leistungsfähigkeit der Hardware multipliziert, sondern damit einhergehend auch die Größe und Komplexität der Programme. Die Notwendigkeit, diese vervielfachte Komplexität besser handhabbar zu machen, führte zu einer graduellen Veränderung der Programmierparadigmen, d.h. der konzeptionellen Abbildung der Realität in einem Programm[1]. Die Veränderung der Programmierparadigmen war eng verknüpft mit der Entwicklung der Programmiersprachen und der von diesen vorgegebenen Abstraktionsprinzipien. So lassen sich mehrere Generationen von Programmiersprachen identifizieren, denen jeweils ein verändertes Programmierparadigma zu Grunde liegt (vgl. [Booch94, S. 26]): Sprachen der ersten Generation (z.B. Fortran I, Algol 58) dienten dazu, Programmierung auf einem etwas höheren Abstraktionsniveau als die damals übliche Assembler-Programmierung zu ermöglichen. In Sprachen der zweiten Generation (z.B. Fortran II, Cobol etc.) wurde verstärkt Gebrauch von Subroutinen gemacht, zuerst aus Gründen der Zeit- und Code-Ökonomie, später als bewusstes Mittel der Strukturierung und Abstraktion. Die Sprachen der dritten Generation lassen sich in zwei Gruppen teilen: Einerseits die Sprachen, bei denen strukturierte Programmierung und Modularisierung im Vordergrund steht (z.B. Pascal), andererseits objektorientierte Sprachen (z.B. Simula, Java), deren Hauptmerkmal die Kapselung von Algorithmen und zugehörigen Daten ist. Seit Anfang der 90er-Jahre existieren Programmiersprachen der vierten Generation (4GL), bei denen eine in der Regel objektorientierte Sprache den Kern eines integrierten Entwicklungssystems für Oberflächengestaltung, Datenbankanbindung etc. bildet. Komponentenkonzepte spielen in 4GL-Umgebungen eine wichtige Rolle. Ob 4GL-Sprachen aber auch ein höheres Abstraktionsniveau als 3GL-Sprachen implizieren, wird teilweise bezweifelt ([Whitehead02, S. 9]).

---

[1] [Ambler92, S.28] definiert den Begriff „Programmierparadigma” in diesem Sinne als „collection of conceptual patterns that together mold the design process and ultimately determine a program’s structure“.

Das prozedurale Programmierparadigma ist aktionsbezogen und stellt Prozesse und Algorithmen in den Vordergrund. Das Dekompositionsprinzip ist die Gliederung in Routinen, und folglich ist das typische Interfaceschema prozeduraler Programme der Aufruf von eigenen oder fremden Subroutinen, was Stroustrup mit dem Schlagwort *„Decide which procedures you want; use the best algorithms you can find.“* ([Stroustrup91, S. 2]) zusammenfasst.

Der Begriff „prozedural“ wird terminologisch teilweise unterschiedlich benutzt: Meyer verwendet in seinem Standardwerk ([Meyer97]) „prozedural“ im Sinne von „imperativ“, was im Kontext jenes Werkes heißt, dass auch eine objektorientierte Sprache prozedural sein kann. In dieser Arbeit hingegen wird „prozedural“ als Antonym zu „objektorientiert“ verwendet. Diese Bedeutung des Begriffs findet man auch an vielen anderen Stellen, wie etwa bei [Stroustrup91], [Booch94], [Wampler02] oder [Gamma95]. Eine andere terminologische Unterscheidung ist die zwischen „objektorientiert“ und „objektbasiert“. [Wegner87] bezeichnet als lediglich objektbasiert solche Sprachen, die zwar das Konzept des Objekts kennen, aber im Gegensatz zu objektorientierten Sprachen keine Vererbung ermöglichen. Beispiele sind ADA und vor allem Visual Basic 6.0 und VBA. Diese Unterscheidung wird von vielen Autoren, unter anderem auch von [Meyer97], übernommen.

Folgende Kriterien zeichnen eine objektorientierte Sprache aus (nach [Booch94, S. 25 ff]):

- Abstraction: Ein Objekt verfügt über eine Menge von Hauptmerkmalen, die es von anderen Objekten in charakteristischer Weise unterscheiden und die es zu einem Modell einer realen oder virtuellen Entität oder Aktion machen.
- Encapsulation: Trennung des Inneren eines Objekts (seine Implementation) vom Äußeren (seine in den Interfaces manifestierte Abstraktion)
- Modularity: Aufteilung eines Systems in zusammengehörige, aber lose gekoppelte Module als Strukturierungsprinzip oberhalb von Klassen bzw. Objekten
- Hierarchy: Hierarchische Anordnung von Abstraktionen

Darüber hinaus können weitere Kriterien ausgemacht werden, die häufig auf objektorientierte Sprachen zutreffen - jedoch nicht im Sinne notwendiger Bedingungen wie die zuvor genannten Merkmale:

- Typing: Festsetzung von Struktur- oder Verhaltenseigenschaften einer Gruppe von Entitäten (meist Klassen), die allen Elementen dieser Gruppe gemeinsam ist, verbunden mit der Festlegung von Austauschbarkeitsregeln für Entitäten unterschiedlichen Typs

- Concurrency: Möglichkeit der quasi-gleichzeitigen Aktivität mehrerer Objekte (via Multithreading oder Multiprocessing)
- Persistence: Fähigkeit, den Zustand eines Objektes über die Zeit (länger als den Ursprungsprozess) und über den Raum (über Rechnergrenzen hinweg) zu bewahren

Das Interfaceschema objektorientierter Sprachen besteht in der Erzeugung von Objekten aus Klassen und der Nutzung deren Methoden und Eigenschaften, wobei die Klassen ggf. in Vererbungszusammenhängen stehen. [Stroustrup91, S. 7] fasst das wie folgt zusammen: *"Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance."*

4.2 Komponentenmodelle

4.2.1 Softwarekomponenten

Die Frage, was eine (Software-)Komponente ist und durch welche Charakteristika sich eine Komponente von anderen Konstrukten der Softwarearchitektur (Objekte, Klassen, Module etc.) unterscheidet, ist nicht einfach zu beantworten. Da Komponenten in sehr unterschiedlichen Formen vorkommen, etwa als Source-Code-Module, als Teile einer Architektur, als Designbestandteile oder als eigenständig verteilbare und ausführbare Binärmodule (vgl. [Crnkovic02, S. 1]), gibt es in der Literatur eine Vielzahl unterschiedlicher Definitionen für den Begriff „Komponente". In [Broy98] etwa diskutieren neun Autoren ihre jeweils unterschiedlichen Definitionen des Komponentenbegriffs. Diese Arbeit legt die im Standardwerk von Clemens Szyperski ([Szyperski02, S. 41]) gegebene Definition zu Ggrunde:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Zur Erklärung, Ergänzung und Abgrenzung dieser Definition seien folgende Merkmale von Softwarekomponenten herausgestellt:

- **Kapselung:** Die wohl wichtigste Eigenschaft einer Komponente ist die strikte Kapselung der Implementation vom Umsystem, wobei Interfaces die einzigen Verbindungspunkte bilden. Man unterscheidet *Whitebox*- und *Blackbox*-Kapselung. Whitebox-Komponenten erlauben die Ansicht und Veränderung des Sourcecodes, während Blackbox-Komponenten meist in kompilierter Form die Details ihrer Implementation verbergen. Typischerweise liegen Komponenten als Blackbox-Kapseln vor (vgl. [Nierstrasz97]).

- **Interfaces:** Eine Komponente kommuniziert mit ihrer Umgebung ausschließlich über ein- und ausgehende Interfaces. Meist haben Komponenten mehrere Interfaces, die für unterschiedliche Aufgaben, bzw. Clients unterschiedliche Services anbieten. Interfaces sind dabei wie verbindliche Verträge, in denen Ein- und Ausgaben sowie Transaktionen spezifiziert sind.
- **Kontextunabhängigkeit:** Komponenten sind in sich abgeschlossene Einheiten, die, abgesehen von funktionalen, in Interfaces spezifizierten, expliziten Abhängigkeiten, keine weiteren impliziten Abhängigkeiten mit ihren Umsystemen besitzen. Komponenten sind also separabel vom spezifischen, applikationsbezogenen Kontext und können daher auch in anderen Applikationskontexten eingesetzt werden. Designprinzip einer Komponente sollte daher hohe interne Kohäsion, verbunden mit geringer externer Verkupplung sein (vgl. [Whitehead02, S. 20]).
- **Unabhängige Verteilbarkeit:** Verteilung und Einbettung („deployment") einer Komponente erfolgen unabgängig vom Entwicklungszyklus der internen Implementation, und das Update einer Komponente sollte nach Möglichkeit keine Re-Kompilation und kein Re-Linking der Clientapplikation nach sich ziehen ([Crnkovic02, S.5]).
- **Wiederverwertbarkeit:** Ähnlich wie Klassen, erlauben Komponenten prinzipiell die Wiederverwertung in einem anderen Kontext. Zwar werden keineswegs alle Komponenten zu diesem Zweck geschrieben, für die Kommerzialisierung auf Komponentenmärkten ist Wiederverwertbarkeit jedoch der entscheidende Faktor.
- **Zustandslosigkeit:** Eine Komponente hat keinen extern beobachtbaren Zustand und kann daher auch nicht von einer Kopie ihrer selbst unterschieden werden. Dies trennt eine Komponente von einem Objekt, das eine eindeutige Identität hat. Eine Komponente ist zustandslos, auch wenn ein aus ihr instanziertes Objekt zustandsbehaftet ist (vgl. [Szyperski02, S. 36-37]).
- **Verwendung durch Dritte:** Komponenten sind so beschaffen, dass ein Dritter sie zur Applikationskonstruktion verwenden kann, ohne Zugriff auf oder Verständnis von internen Implementationsdetails zu haben.
- **Ausführbare Form:** Komponenten sind immer in einer durch eine Engine ausführbaren Form, wobei die Engine ein Prozessor, ein Script Interpreter oder ein JIT-Compiler sein kann. Es ist nicht zwingend, dass die Komponente Binärform hat[1].

[1] In der ersten Ausgabe von [Szyperski02] heißt es noch „binary form", wurde aber in der zweiten Auflage des Buchs nach einiger Kritik auf „executable form" geändert. Sinngemäß ebenso [Szyperski98, S. 3].

Die zentralen Eigenschaften einer Komponente sind dabei statische Kapselung und Schnittstellen, was [Nierstrasz97] mit folgender knappen Definition umreißt: *Eine Softwarekomponente ist eine „statische Softwareabstraktion mit Plugs"*. Der Begriff „Plug" soll andeuten, dass sich Komponenteninterfaces wie Stecker und Steckdose verhalten. Eine andere, häufig gezogene Analogie ist die zwischen Softwarekomponenten und Komponenten, wie sie in der industriellen Fertigung, insbesondere bei Integrierten Schaltkreisen (IC), vorkommen, wo Standard-ICs in unterschiedlichen Geräten (=Applikationen) verbaut werden können.

4.2.2 Komponentenframeworks

Komponenten existieren nicht isoliert, sondern bedürfen zu ihrer Einbettung und Funktion eines softwareorganisatorischen Rahmens, der als Komponentenframework bezeichnet wird. Zur Abgrenzung und weiteren Erläuterung soll in diesem Zusammenhang auch auf die Begriffe *Komponentenmodell, Pattern, Interface* und *Contract* eingegangen werden.

Allgemein gesagt ist ein *Framework* das Modell einer typischen und wiederverwertbaren Situation (vgl. [DSouza98]), oder wie es [Johnson97, S. 39] mit Bezug auf objektorientierte Wiederverwertung umreißt: Ein Applikationsskelett, das vom Entwickler gefüllt und angepasst wird. [Nierstrasz97, S. 3] konkretisieren dies in Bezug auf Komponenten: „*Ein Komponentenframework besteht aus einer Bibliothek von Komponenten und einer zugehörigen Softwarearchitektur, welche die Basiseigenschaften der Plugs und die Art und Weise der Komposition festlegt.*". Komponentenframeworks standardisieren beispielsweise die Form der Fehlerbehandlung, des Datenaustauschs, der gegenseitigen Aufrufe oder der Persistenzmechanismen. Nach [Nierstrasz97] bildet das Framework auch die Grundlage, auf der die Verknüpfung der Komponenten zu einer Applikation stattfindet, ein Mechanismus, der als „Gluing" oder „Component Scripting" bezeichnet wird. Beispiele für Komponentenframeworks sind OLE, OpenDoc, oder JavaBeans. Die bekannteste Component Scripting Sprache ist Visual Basic.

Eng mit dem Begriff des Frameworks verbunden, jedoch keineswegs mit diesem identisch, ist das Konzept der *Patterns*. Patterns sind die kleinsten wiederkehrenden Architektureinheiten in objektorientierten Softwaresystemen. Sie beschreiben als Mikroarchitektur in abstraktrer Form die Interaktion zwischen Objekten ([Szyperski02, S. 156-157]). Was ihre Granularität anbelangt, sind sie somit oberhalb von Klassen, aber unterhalb von Frameworks einzuordnen. Patterns sind allerdings logischer und abstrakter Natur, während ein Framework eher physischer Natur ist. Ein Framework kann mehrere Patterns enthalten, jedoch nicht umgekehrt. Weitere Details und einen umfangreichen Katalog von Patterns findet man in [Gamma95]. Ein häufig genanntes Beispiel ist das Smalltalk *Model-View-Controller Framework (MVC)* (vgl.

[Krasner88]), das im Wesentlichen auf den Patterns *Observer*, *Composite*, und *Strategy* (s. [Gamma95]) beruht.

Ein ebenfalls mit Komponentenframeworks nah verwandter, jedoch keineswegs identischer Begriff ist der des Komponentenmodells: Während das Komponentenmodell die Standards und Konventionen der Komponentenkonstruktion und -interaktion beschreibt, stellt das Komponentenframework die Infrastruktur für Einbettung und Steuerung von Komponenten dar. Ein Komponentenframework ist also eine Implementation von Diensten, die die Verwendung von Komponenten ermöglichen, die einem bestimmten Komponentenmodell entsprechen (vgl. [Bachman00]).

Interfaces sind kodifizierte Kontrakte, über die Clients und Dienstprovider aus teilweise unabhängigen Quellen miteinander interagieren. Bezogen auf ein Framework ist ein einzelnes Interface eine Einheit mit einer sehr feinen Granularität. Ein Interface kann von mehreren unterschiedlichen Clients benutzt werden, und es können mehrere Interfaces zu Kontraktsystemen zusammengeschlossen werden (vgl. [Helm90], der solche Zusammenschlüsse als *Behavioral Composition* bezeichnet). Für die komponentenbasierte Softwareentwicklung in offenen Systemen spielt die Fokussierung auf Interfaces und ihre Spezifikation eine immer bedeutsamere Rolle, weshalb einige Autoren auch von „Interface-zentrierter Architektur" sprechen (vgl. [Jonkers01], sowie zum ISpec-Ansatz [Jonkers00]).

Die [Crnkovic02, S. 16][1] entnommene Abbildung 12 zeigt die oben beschriebenen Zusammenhänge in einer anschaulichen grafischen Aufbereitung: Eine Komponente implementiert ein Interface, das die Spezifikationen eines wohldefinierten Kontrakts umsetzt. Eine Komponente kann auch mehrere Interfaces besitzen, und mehrere Komponenten können zusammen einem Pattern entsprechen. In der dargestellten Analogie kommen Komponentenframeworks elektronischen Platinen mit leeren Steckplätzen gleich, in die die Komponenten (die ICs) eingesteckt werden können. Das Framework, stellt dabei die „Verdrahtung" der Komponenten mittels diverser koordinierender Dienste (z.B. für Transaktionen, Persistenz etc.) sicher. Das Komponentenmodell spezifiziert in allgemeiner Weise den Typ der Komponenten und die Mechanismen ihrer Interaktion untereinander sowie mit dem Framework.

---

[1] Ursprünglich findet sich diese Veranschaulichung bei [Bachman00, S. 3], in [Crnkovic02, S. 16] ist sie jedoch grafisch besser dargestellt.

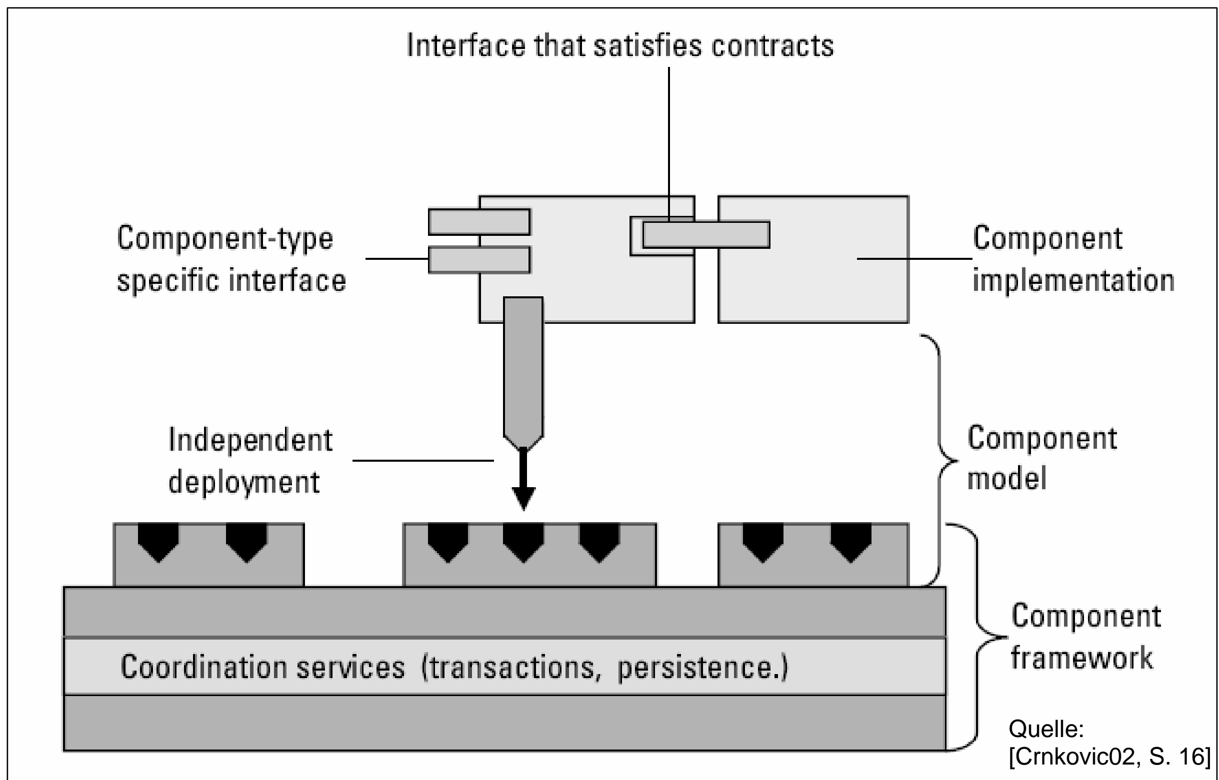


Abbildung 12: Komponenten, Komponentenframeworks und Komponentenmodell

4.2.3 Standards

Komponentenbasierte Softwareentwicklung (*Component Based Software Development, CBSD*) mit multiplen, zueinander nicht in direktem Kontakt stehenden Entwicklern ist nur möglich, wenn es stringente und verbindliche Standards gibt. Insbesondere das Komponentenmodell inklusive seiner Standardinterfaces sowie das Komponentenframework müssen allgemeinverbindlich spezifiziert sein. Auch für die Entstehung von Komponentenmärkten sind Standards eine notwendige Voraussetzung, wobei es letztlich unerheblich ist, ob diese von Standardisierungsgremien (z.B. OMG) oder großen Playern der Softwareindustrie (z.B. Microsoft, Sun) gesetzt werden – entscheidend sind Verbreitung und Verbindlichkeit (vgl. [Szyperski02]).

Einige wichtige Standards für Komponentenmodelle und -frameworks sollen im Folgenden kurz skizziert werden, wobei die für diese Arbeit relevanten Komponentenstandards unter Windows im Abschnitt 4.3 noch einmal detailliert aufgegriffen werden.

Die wesentlichen Komponentenstandards lassen sich in drei Gruppen aufteilen: Zum einen die von der *Object Management Group (OMG)* als Standardisierungsgremium entwickelten Standards, vor allem die *Common Object Request Broker Architecture (CORBA)*, zum anderen die

von Sun publizierten und protegierten Standards aus dem Java-Bereich (*JavaBeans* etc.) sowie schließlich die marktdominanten Standards *COM* und *.NET* von Microsoft. Detaillierte Beschreibungen und Vergleiche dieser Standards findet man z.B. in [Gruhn00], [Zwin05], [Szyperski02], [Crnkovic02] und [Wang05] sowie in knapperer Form in [Griffel98], [Stroux05] und [Schryen01].

4.2.3.1 CORBA

Die Object Management Group (OMG, [OMG07]) ist ein offener, nicht-kommerzieller Zusammenschluss von mehreren Hundert Mitgliedern, dessen Ziel die Standardisierung von Technologien aus dem Softwarebereich ist. Im Zentrum steht dabei die *Component Object Request Broker Architecture CORBA,* mit dem *CORBA Component Model* und weitere Verwandte Technologien, wie etwa CORBA-spezifische Interfacespezifikationen (z.B. für Telekommunikationsnetzwerke, WLAN etc.). Die erste Version 1.0 wurde bereits 1991 publiziert, und mittlerweile (2007) liegt CORBA in Version 3.0.3 mit einem neuen Komponentenmodell (*CCM, CORBA Component Model)* vor.

CORBA ist eine Spezifikation, kein Produkt. Es gibt jedoch eine größere Zahl von Produkten, sowohl aus dem Open Source Bereich (z.B. MICO, [Mico07]) als auch von kommerziellen Anbietern (z.B. Borlands VisiBroker ggf. in Verbindung mit C++-Builder oder JBuilder, [Borland07]), die CORBA, oder Teile davon, implementieren.

Herausragendes Merkmal von CORBA ist seine Plattform- und Sprachunabhängigkeit, d.h. Client- und Serverkomponenten können miteinander kommunizieren, auch wenn sie in unterschiedlichen Programmiersprachen und/oder auf unterschiedlichen Systemplattformen implementiert sind. Abbildung 13 schematisiert das Zusammenspiel von Client- und Serverkomponenten in CORBA. Aus rein logischer Sicht ruft die Client-Komponente eine Methode der Server-Komponente auf (Call) und erhält von dieser ggf. einen Rückgabewert (Return). Aus der logischen Sicht der beteiligen Komponenten stellt sich dies wie ein normaler Objekt-Methodenaufruf dar. Objektorientierung ist für CORBA übrigens nicht zwingend: Neben den verbreiteten objektorientierten Sprachen C++ und Java existieren auch CORBA-Mappings für prozedurale Sprachen, wie PL/1 und Cobol.

Tatsächlich existieren die Client- und Serverkomponenten in einem verteilten System jedoch auf unterschiedlichen Rechnern, die über ein TCP/IP-basiertes Netzwerk miteinander in Verbindung stehen. Dabei setzt das *Internet Inter ORB Protocol* (IIOP, vgl. [Minton07]) auf TCP/IP auf und ermöglicht die Kommunikation der *Object Request Broker (ORB).* ORBs dienen der Lokalisierung von Objekten im Netz, dem Referencing sowie dem Marshalling

von Methodenaufrufen, Parameter und Rückgabewerten. Object Request Broker sind sowohl client- wie auch serverseitig vorhanden. Die Komponenten kommunizieren jedoch nicht mit den ORBs direkt, sondern über so genannte Stubs und Skeletons, wobei Erstere für die Kodierung der Aufrufe (Marshalling) und Letztere für deren Dekodierung (Demarshalling) zuständig sind. CORBA Interfaces sind in einer *Interface Definition Language (IDL)* formuliert. Ein IDL-Compiler erzeugt daraus Stub- und Skeleton-Code. Liegt die Interfaceinformation bereits zur Kompilierungszeit vor, so erfolgt eine als *Static Invocation Interface (SII)* und *Static Skeleton Interface (SSI)* bezeichnete statische Bindung, andernfalls kommt eine als *Dynamic Invocation Interface (DII)* bzw. *Dynamic Skeleton Interface (DSI)* bezeichnete dynamische Bindung zum Einsatz, die die notwendigen Bindungsinformationen in den Registrierungsdatenbanken *Interface Repository* und *Implementation Repository* verwendet.

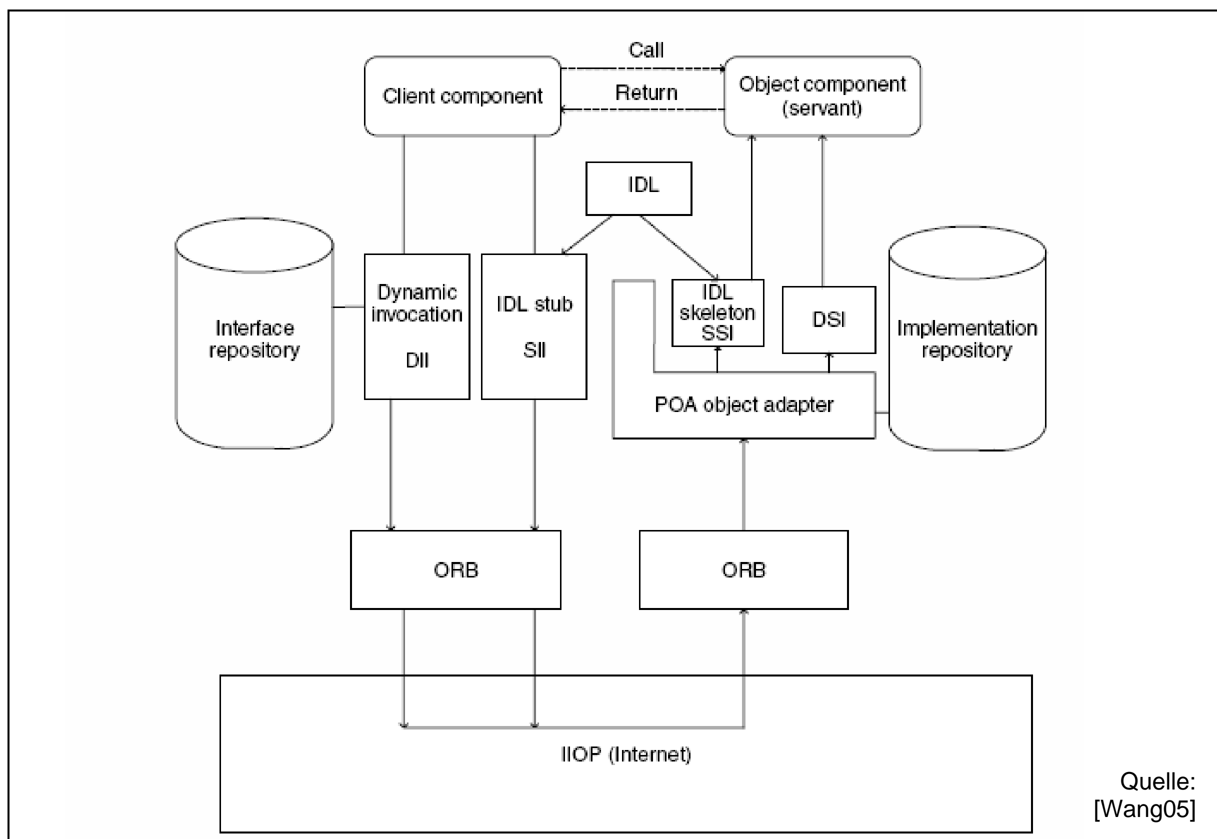


Abbildung 13: CORBA Übersicht

4.2.3.2 Java-Komponenten

Die Programmiersprache Java zeichnet sich durch Plattformunabhängigkeit aus, die dadurch erreicht wird, dass der Programmcode zunächst in einen plattformunabhängigen Java Bytecode übersetzt wird und dieser dann von der Java Virtual Machine auf dem jeweiligen System ausgeführt wird. Standen zunächst die Java Applets für die sichere Ausführung von Code auf

Webseiten im Fokus der Entwicklung, so haben sich seit Ende der 1990er-Jahre javabasierte Komponententechnologien im Rahmen von Java 2 einen immer breiteren Raum verschafft.
Wie in Abbildung 14 dargestellt, unterscheidet die Java 2 Enterprise Edition (J2EE) drei Gruppen von Komponentenmodellen mit insgesamt neun Variationen (vgl. [Szyperski02, S. 268 ff.]): Auf dem Client Tier sind dies *Applets*, *Application Components* und *JavaBeans*, wobei Letztere zwar primär dem Client Tier zuzuordnen sind, aber auch auf anderen Tiers vorkommen können. Auf dem Web Server Tier befinden sich *Java Server Pages* und *Servlets* und auf dem Application Server Tier *Enterprise JavaBeans (EJBs)* in vier verschiedenen Variationen (entity, stateful session, stateless session und message driven). Der Backend Tier, ebenso wie die Tierübergreifenden Messaging-, Naming- und Directory-Services sind zwar Teil der Architektur bzw. des Frameworks, enthalten aber keine eigenständigen Komponentenmodelle.

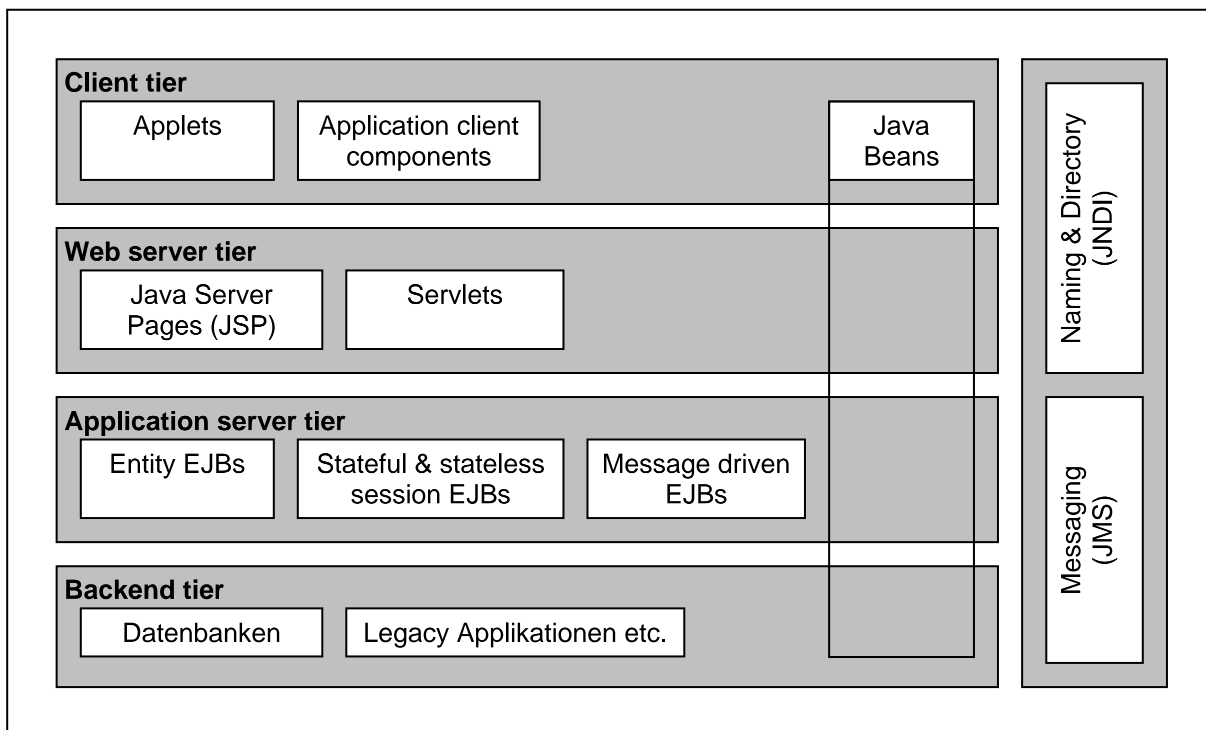


Abbildung 14: Komponentenmodelle in der J2EE-Architektur

Eines der wichtigsten Komponentenkonzepte in Java ist das der JavaBeans (vgl. [Brookshier99]), das Ähnlichkeiten mit OLE bzw. ActiveX in der Microsoft-Welt aufweist. Charakteristisch für JavaBeans ist zunächst die Unterscheidung in Run-Time- und Design-Time-Verhalten. JavaBeans enthalten Code, der den Programmierer während der grafischen, komponentenbasierten Programmentwicklung (Design-Time), z.B. beim Setzen von Eigenschaftswerten oder bei Persistenzfunktionalitäten, unterstützt. Eine JavaBean kann mehrere

Klassen und weitere Ressourcen enthalten. JavaBeans zeichnen sich vor allem durch folgende Merkmale aus (vgl. [Szyperski02, S. 285 ff.], [Wang05, S 38 ff.]):

- **Properties**: Attribute bzw. Eigenschaften der Komponente werden über Kombinationen von Get-/Set-Methoden ausgelesen und gesetzt. Properties können indiziert sein, sie können Validitätsüberprüfungen vornehmen und Exceptions auslösen.
- **Events**: JavaBeans können Eventobjekte erzeugen und diese an alle Objekte propagieren, die sich bei der jeweiligen JavaBean zuvor als Event-Empfänger (*Listener*) registriert haben. Dies ist sehr ähnlich dem in COM verwendeten Prinzip.
- **Introspection**: Über diesen Mechanismus stellen JavaBeans Informationen über die angebotenen Eigenschaften, Methoden und Events für die Abfrage durch geeignete Tools (z.B. grafische Entwicklungsumgebungen) zur Verfügung.
- **Customization**: Die Eigenschaften einer JavaBean können zur Design-Time z.B. über Editoren einer Entwicklungsumgebung (Property-Editoren) angepasst werden.
- **Persistence**: JavaBeans können ihren Zustand persistent machen, indem sie Informationen über ihre aktuellen Eigenschaften in Files abspeichern und wiederherstellen (serialisieren). Dazu muss die Komponente ein bestimmtes Interface (java.io.serializing) implementieren.

Enterprise JavaBeans unterscheiden sich – trotz der Namensverwandtschaft – ganz erheblich von den oben genannten JavaBeans. Während Letztere ein verbindungs- und eventorientiertes Komponentenmodell für den Einsatz vor allem in GUIs zu Grunde legen, zielt das Modell der EJBs auf den Einsatz im Server Tier. EJBs sind Softwarekomponenten, die in unterschiedlichen Applikationen und auf unterschiedlichen, EJB-kompatiblen Systemen wieder- verwendet werden können, ohne dass eine Neukompilation erforderlich wäre. Ein J2EE-Server hostet EJB-Container, in den die EJB-Instanzen ablaufen. Die Container bilden also die Laufzeitumgebung für die Enterprise JavaBeans und stellen Services für Sicherheit, Transaktionen, Lebenszyklusmanagement und Persistenz zur Verfügung. Clients kommunizieren dabei nicht direkt mit den EJBs, sondern nur vermittelt über den Container als Proxy. EJBs besitzen zwei Standardschnittstellen, das *Home Interface* und das *Remote Interface*. Das Home Interface enthält Methoden, die den Lebenszyklus der EJB-Instanzen betreffen (Erzeugung, Aktivierung, Suchen, Löschen), während das Remote Interface die eigentlichen, funktionalen Methoden enthält, die durch Clients von anderen Systemen aus aufgerufen werden können. Optional existiert für schnelle, overheadarme Zugriffe von lokalen Clients, die auf demselben Server wie die EJB laufen, auch ein so genanntes *Local Interface*. Auf die unterschiedlichen EJB-Typen (Entity, Session, Message-driven) soll hier nicht weiter eingegangen werden, stattdes-

sen sei auf die ausführliche Darstellung in der Literatur verwiesen, etwa auf [Schmieten02] für die ältren EJB-Standards 1.1 und 2.0, sowie auf [Eberling07] für den aktuellen Standard 3.0.

4.2.3.3 COM und .NET

Die Microsoft-Komponentenmodelle COM und .NET sind die mit Abstand marktbeherrschenden Technologien ([Szyperski02, S. 25-26]), wobei die Tendenz zur graduellen Ablösung von COM durch .NET klar erkennbar ist[1]. An dieser Stelle sollen die genannten Technologien nur kurz skizziert werden, die detaillierte Darstellung erfolgt in Kapitel 4.3, wo sich auch weitere Quellenhinweise finden. Aus der kaum überschaubaren Fülle an Literatur zum Thema seien hier als Übersichtswerke lediglich der Klassiker von D. Box ([Box97]) zu COM sowie [Beer06] zu .NET angeführt.

Den Ausgangspunkt der Entwicklung bildeten die *Visual Basic Extensions (VBX)* - nicht objektorientierte Komponenten der frühen Visual Basic (VB) Versionen 1.0 bis 3.0. Mit VB-Version 4.0 wurden die VBX durch COM-basierte Komponenten abgelöst. Diese werden oft als *OCX-Controls* bezeichnet, was aber unpräzise ist, da *.ocx* nur die Dateiendung der Bibliotheken (intern DLLs) ist. OCX-Files können mehrere *OLE*- bzw. *ActiveX-Controls* enthalten, wobei beide Begriffe Synonyme sind – aus Marketinggründen sollte mit dem Begriff „ActiveX" die Ausführbarkeit in geeigneten Webbrowsern unterstrichen werden[2]. Ursprünglich bezeichnete *OLE (Object Linking and Embedding)* lediglich eine Technologie für komponentenbasierte Verbunddokumente unter Windows. Version OLE 2 erweiterte den Anwendungsbereich jedoch zu einer einheitlichen, objekt-basierten, anpassbaren und erweiterbaren Dienstarchitektur zur Integration von Komponenten ([Brockschm97, Kap. 1]). Das ursprüngliche *Linking and Embedding* ist darin nur noch ein Subsystem der Gesamtarchitektur.

Abbildung 15 stellt die OLE-Dienstarchitektur in der Übersicht dar: Die unteren Technologien sind dabei generischer und für den Endbenutzer weniger sichtbar als die im oberen Bereich dargestellten. Die für alle Dienste gemeinsame Basis ist das *Component Object Model (COM)* mit seinen Interface- und Funktionalitätsspezifikationen. Zentral für die in dieser Arbeit implementierten Interfaces sind OLE-/ActiveX-Controls, die ihrerseits Gebrauch von tiefer liegenden Diensten machen, wie z.B. von Property Pages, Events, Persistenz oder Typ-

---

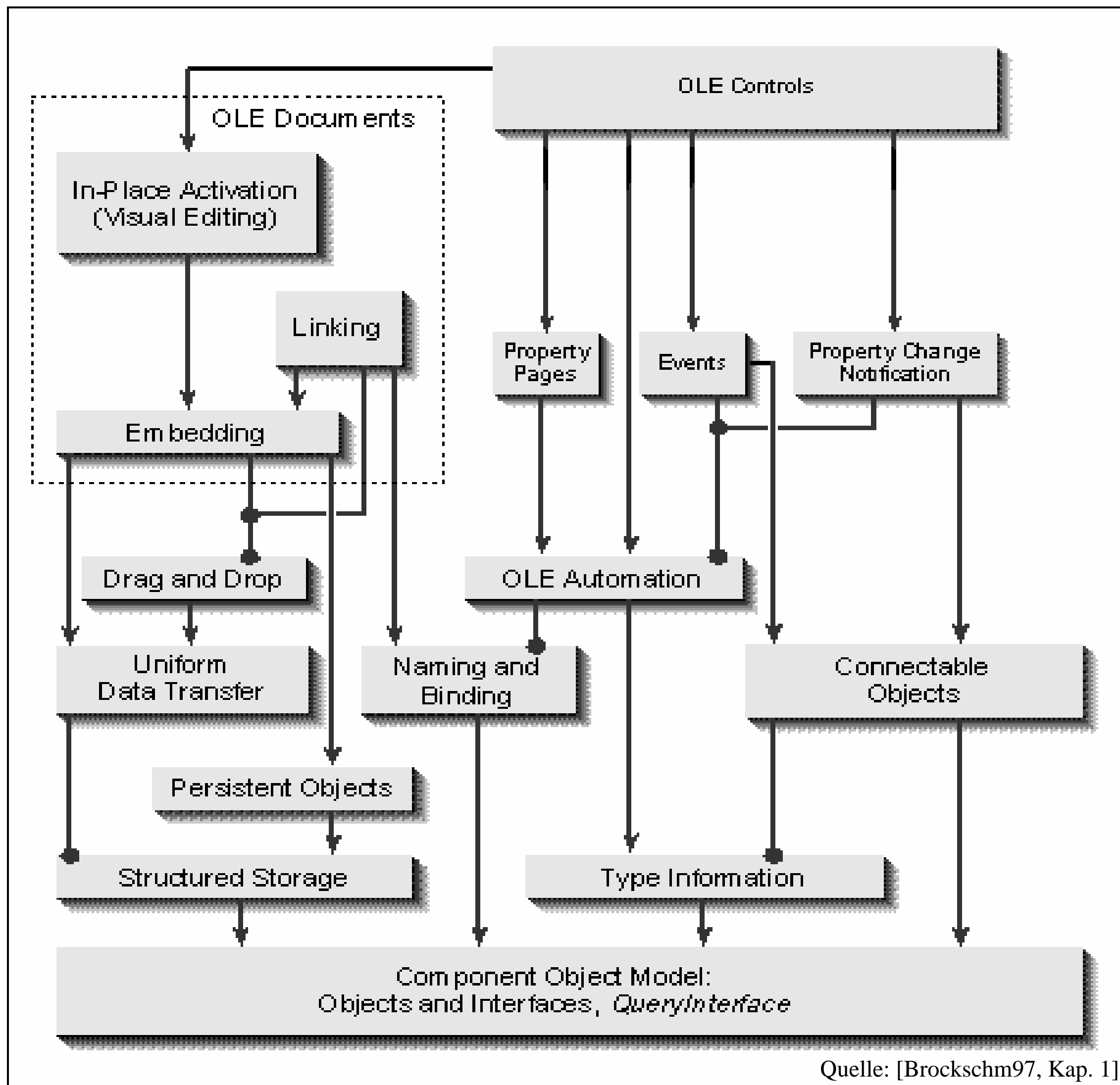
1 Beispielhaft sei die Entwicklung auf Komponentenmärkten, wie etwa ComponentSource (www.componentsource.com), genannt: Während [Szyperski02, S. 25] im Jahr 2001 dort noch 75,5% Marktanteil von COM-Komponenten und 6,9% für .NET ausmachte, liegen die Anteile im Mai 2007 bei nur noch 37,3% für COM, während 50,7% der angebotenen Komponenten bereits .NET-Komponenten sind.

2 Eine Klarstellung dieser oft falsch gebrauchten Termini gibt [Microsoft07]. Bei [Gruhn00, S.48] findet man eine Skizzierung der Begrifflichkeiten im inhaltlichen und zeitlichen Zusammenhang.

Informations-Diensten. Eine detaillierte Darstellung aller OLE-Dienste findet man z.B. in [Brockschm97], daher hier nur eine kurze Nennung der wichtigsten Technologien:

- **Controls**: Binärkomponenten zur Benutzung aus einer Vielzahl von Programmiersprachen und durch geeignete Automatisierungsclients
- **Property Pages**: GUI-Schnittstelle zum Design-Time-Editing der Eigenschaften eines Controls
- **Events**: Mechanismus für Benachrichtigungen der Clients einer Komponente bei Ereignissen (bei Änderung von Eigenschaften: **Property Change Notification**)
- **OLE Automation**: Interface mit Laufzeit-Bindungsmöglichkeit zur Steuerung von Komponenten und Anwendungen (Makro-Programmierung)
- **Type Information**: Mechanismus zur Selbstbeschreibung von Interfaces, Methoden, Eigenschaften und Argumenten durch die Komponente
- **Linking and Embedding**: Mechanismen für Verbunddokumente (**OLE Documents**)
- **Drag and Drop**: Benutzerinduzierter Datenaustausch zwischen unterschiedlichen Objekten über **Uniform Data Transfer**
- **Persistenz**: Möglichkeit der Speicherung von Zustandsinformationen eines Objekts
- **Structured Storage**: Speicherungsmechanismus und -format für OLE-Objekte

Das allen OLE-Technologien zu Grunde liegende Component Object Model ist eine Mischung aus Spezifikation und implementierter, systemweiter Infrastruktur (vgl. [Brockschm96]), also gemäß der in 4.2.2 genannten Definitionen sowohl Komponentenmodell als auch Komponentenframework. Allerdings stehen die spezifikatorischen Elemente im Vergleich zu den implementierten API-Funktionen des Frameworks im Vordergrund, weshalb COM korrekterweise auch den Begriff „Model“ im Namen führt. COM wurde in der ersten Hälfte der 1990er-Jahre entwickelt, um einer Reihe von Problemen Herr zu werden, die bei der DLL-orientierten Bibliotheksverwendung auftreten (z.B. Versionsverwaltung, Pfadabhängigkeiten, Zugriffe über Prozess- oder Rechnergrenzen hinweg etc. Details siehe [Brockschm96]).



Quelle: [Brockschm97, Kap. 1]

Abbildung 15: Übersicht OLE/COM-Technologien

Später, mit der Verfügbarkeit von Windows NT 4.0, wurde COM zu *Distributed COM (DCOM)* erweitert, indem Mechanismen zur rechnerübergreifenden Komponentenkommunikation dem Modell hinzugefügt wurden. DCOM macht die systemübergreifende Komponentennutzung für einen Client völlig transparent, d.h. der Client einer Komponente weiß in der Regel nicht, ob sich die Komponente im eigenen Prozessraum oder auf einem anderen Rechner befindet. Erreicht wird dies über *Proxies* (im Prozessraum des Clients) und *Stubs* (im Prozessraum des Servers), die über *Remote Procedure Calls (RPCs)* Aufrufe und Daten austauschen (vgl. [Gaedke97]).

Eine andere, ebenfalls ausführungslokalitätstransparente Erweiterung des Component Object Models ist COM+, das - ähnlich wie Enterprise JavaBeans - Probleme der Transaktionssicherheit, Verfügbarkeit und Skalierbarkeit zu lösen versucht. Hierzu führt COM+ eine als „Kontext" bezeichnete Zwischenschicht ein, die an die Komponente gerichtete Aufrufe abfängt und Vor- und Nachbereitungsaufgaben übernimmt (vgl. etwa [Zwin05]).

.NET ist Microsofts primäres Technologieparadigma der gegenwärtigen Dekade, in dem eine neuartige Komponentenarchitektur nur einen kleinen Teil ausmacht und dessen Hauptbestandteile die Folgenden sind:

- **Common Language Runtime (CLR)**: Ähnlich wie in Java wird der Quellcode zunächst in eine nicht direkt maschinenausführbare, plattformunabhängige Zwischensprache, die *Microsoft Intermediate Language (MSIL)* übersetzt, die zur Laufzeit von einem Just in Time Compiler in Maschinencode übersetzt wird. Nativer .NET-Code (managed code) wird dabei unter der Kontrolle (Buffer, Garbage Collection etc.) der CLR ausgeführt.
- **.NET Framework**: Einheitliches Schnittstellen- und Bibliothekssystem (GUI, I/O, Datenbanken, Web etc.) für alle .NET-Sprachen
- **Common Type System (CTS)**: Das für alle .NET-Sprachen einheitliche und verbindliche Typensystem ist Voraussetzung für die Sprachinteroperabilität in .NET.

Das .NET-Komponentenmodell unterscheidet sich erheblich von COM: .NET-Komponenten sind zwar ebenfalls sprachunabhängig, liegen aber nicht in Binärform, sondern in MSIL vor. Einzelne Klassen werden zusammen mit zugehörigen Metadaten zu Modulen (Komponenten) zusammengefasst, und ein oder mehrere Module können eine *Assembly* bilden. Assemblies enthalten neben den Modulen mit deren Code und Metadaten auch eigene Metadaten im so genannten *Manifest*. Eine .NET-Komponente ist also keine Assembly, sie wird vielmehr in einer Assembly ausgeliefert ([Wang05, S.199]). Während ein einzelnes Modul zum Kompilierungszeitpunkt vorhanden sein muss, können Assemblies dynamisch zur Laufzeit geladen werden. Ähnlich wie COM-Komponenten können auch .NET-Komponenten im gleichen (.dll) oder in einem unterschiedlichen Prozessraum (.exe) wie der Client ausgeführt werden. Ebenfalls ist eine Ausführung auf verteilten Systemen wie bei DCOM möglich, jedoch werden dazu andere Mechanismen als die RPCs bei DCOM benutzt. Insbesondere seien hier WebServices und die verwandten Technologien genannt (XML, SOAP etc., vgl. etwa [Newco02]). Zusätzlich gibt es noch die Ausführung in der *Application Domain*, einem virtuellen, eigenen Prozess im realen Prozessraum des Clients.
Wegen vieler Verbesserungen und Vereinfachungen, die .NET gegenüber COM bietet, wurde verschiedentlich die Frage gestellt: „Ist COM tot?“ ([Box00]). Dies ist klar zu verneinen, da zum einen bei näherer Analyse eine größere Zahl von Parallelen zwischen beiden Technologien existiert, so dass man .NET trotz seiner Verschiedenheit auch als eine Weiterentwicklung

von COM auffassen kann (vgl. [Box00]). Außerdem ist COM immer noch einer der zentralen Bestandteile des aktuellen Microsoft-Betriebssystems *Vista* ([Microsoft07b]). Vielmehr ist ein längerfristiges Nebeneinander beider Modelle zu erwarten, das vor allem durch die vorhandenen Brückentechnologien (RCW, CCW) ermöglicht wird.

4.2.4 Komponentenmärkte

Komponenten sind Softwarebausteine, die besonders für die Wiederverwertung durch Dritte geeignet sind. Vor allem Komponenten, die Standardprobleme lösen, können oft in Hunderten oder Tausenden von unterschiedlichen Applikationen Einsatz finden. Die Folge ist das Entstehen von Komponentenmärkten.

Die Entwicklung eines Komponentenmarktes für eine bestimmte Basiskomponententechnologie und das erfolgreiche Zusammenspiel der Hauptakteure (Anbieter, Nachfrager, Standardisierungsinstanzen und Frameworkentwickler) ist jedoch kein Automatismus, sondern ein sich selbst verstärkender Prozess, der vor allem im Anfangsstadium sehr fragil ist. Einigen Gedanken von [Szyperski02] folgend, kann man nachstehenden Prozess ausmachen: Ohne das Vorhandensein der kritischen Masse eines Marktes lohnt die Entwicklung vieler Komponenten, inklusive des beträchtlichen Overheads jenseits der eigentlichen Programmierung (Testing, Dokumentation etc.) nicht. Ein Komponentenmarkt bedarf allerdings einer (wenn auch simplen) technologischen Basis und verbindlicher Standards. Bevor also ein Markt entstehen kann, muss ein eine Art techno-ökonomischer „Bootstrap“ geschaffen werden. Diese Vorleistung muss von Standardisierungsinstanzen und Frameworkentwicklern kommen und sollte neben Spezifikationen und Basisdienstimplementationen auch Dokumentation und ein gewisses Maß an Public Relation für die neue Komponententechnologie umfassen. Aufbauend auf diesem Bootstrap beginnen Komponentenentwickler, ihre Produkte auf den Markt zu bringen, auf dem sich gleichzeitig, induziert durch die neuen technologischen Möglichkeiten des Bootstraps, eine Erwartung seitens der Nachfrager aufgebaut hat. Der so entstehende Komponentenmarkt bewegt nun wiederum die Standardisierungsinstanzen und Frameworkentwickler dazu, Spezifikationen und Frameworks zu verbessern und auszubauen, was wiederum den Markt fördert und wachsen lässt. Abbildung 16 soll diesen Prozess skizzieren.

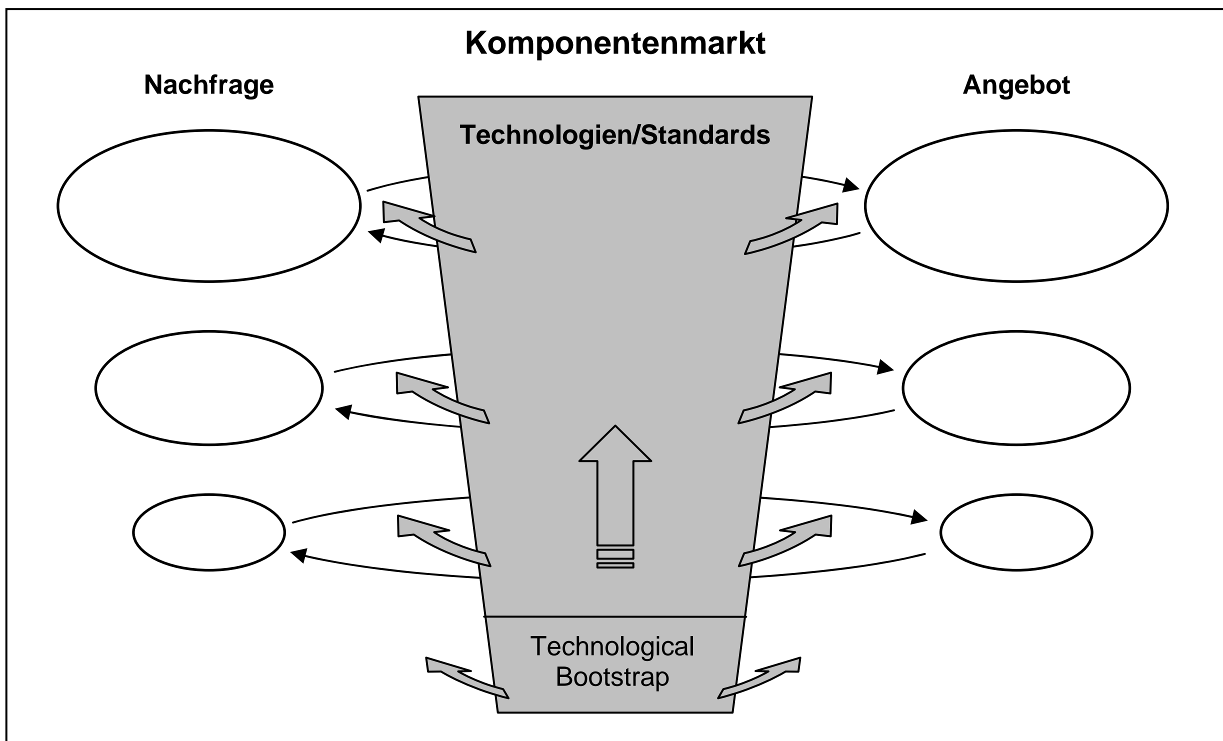


Abbildung 16: Entwicklung von Komponentenmärkten

Die rein technologische Qualität des Bootstraps ist dabei nicht alleinentscheidend, wichtig ist, dass darüber hinaus auch ökonomische Maßnahmen zur Ingangsetzung des Marktes erfolgen (Werbung, Kopplung an andere Produkte etc.) – was bei einer gewissen Marktmacht der Akteure leichter fällt. [Szyperski02, S. 18] formuliert das so: "*imperfect technology in a working market is sustainable; perfect technology without any market will vanish*."

Am Beispiel der o.g. Komponententechnologien CORBA und COM lässt sich dies deutlich nachvollziehen. CORBA hatte zwar einen anspruchsvollen Spezifikationsprozess durchgemacht und wird von Vielen als technisch überlegene Architektur gesehen (etwa [Lewan98], [Schryen01]), konnte aber bei Weitem nicht die Marktdurchdringung erreichen wie COM, da es an ökonomischem Rückhalt und an Schlagkraft auf Seiten der OMG und ihrer Mitglieder fehlte - jedenfalls im Vergleich zu Microsoft. COM hingegen konnte bereits auf dem techno-ökonomischen Bootstrap der alten VBX-Controls aufsetzen und dank der nicht nur technologisch, sondern auch marktstrategisch cleveren Verbindung mit erfolgreichen Produkten (Visual Basic, MS-Office) schnell einen Markt schaffen und vergrößern. COM zeigt auch in Form von DCOM und COM+, wie ein wachsender Markt die weitere Verbesserung und Erweiterung der Komponententechnologien und -standards fördert.

4.3 Bibliotheks- und Schnittstellendesigns unter Windows

4.3.1 Statische und dynamische Bibliotheken

Statische Bibliotheken enthalten Objektcode, der während der Erstellung eines Programms hinzugelinkt wird, während dynamische Bibliotheken ausführbaren Code enthalten, der zur Laufzeit in den Prozessraum einer Anwendung hineingeladen wird. Entsprechend ihrer Dateiendungen unter Windows wird eine statische als *LIB* und eine dynamische Bibliothek als *DLL* bezeichnet. Beide Bibliothekstypen können sowohl prozedurale als auch objektorientierte Konstrukte enthalten. Die objektorientierte MFC existiert z.B. als LIB und als DLL (plus Typelibrary). Der Grundtyp aller DLL- und LIB-Interfaces ist jedoch immer die „Flat API" ([Brockschm96]) in Form einer einfachen Liste exportierter Funktionsaufrufe.

Statische Bibliotheken

Nach der Kompilation eines Programms sucht der Linker nicht aufgelöste Funktionsreferenzen (Symbols) in den angegebenen LIBs und kopiert diese in die zu erstellende ausführbare Datei, wobei eine Anpassung nicht relativer Adressreferenzen auf die Zielposition erfolgen muss. Die Verwendung statischer Bibliotheken ist nur in Compiler-Sprachen, wie C++ oder Fortran, möglich, nicht aber in Interpreter-Sprachen wie VB6. Wichtig für die korrekte Ansprechbarkeit von LIBs (wie auch von DLLs) ist die Verwendung übereinstimmender *Calling Conventions* durch das einbettende Programm und die Bibliothek. Die Calling Convention entscheidet darüber, wie symbolische Funktions- und Variablennamen aussehen (*Naming Conventions*) und wie Parameter und Rückgabewerte übergeben werden (Prozessorregister oder Stack, Reihenfolge, Verantwortlichkeit für Stack Cleanup etc.). Übliche Calling Conventions sind beispielsweise *cdecl*, *stdcall*, oder *fastcall*. Die korrekte Verwendung passender Calling Conventions ist insbesondere bei der Mixed-Language-Programmierung zu beachten, wenn z.B. eine Fortran-LIB von einem C-Programm eingebunden werden soll (Details siehe [Intel07]).

Dynamische Bibliotheken

Hauptvorteil einer DLL ist die leichte Austauschbarkeit, die Modularisierung und die Möglichkeit der Speicherplatz sparenden Mehrfachverwendung durch unterschiedliche Applikationen. Das Format einer DLL unter Windows ist das *Portable Executable Format (PE),* das auch für ausführbare Dateien verwendet wird. Eine DLL kann neben dem eigentlichen Code auch Daten und Ressourcen (z.B. Icons) enthalten, die in unterschiedlichen Sektionen der

Bibliothek liegen. Während Code-Sektionen von mehreren Prozessen geteilt werden können, sind Daten-Sektionen – mit Ausnahme der *Shared Sections* – einzelprozessbezogen. Es gibt zwei unterschiedliche Arten, wie DLLs eingebunden werden können:

- *Implicit Run-Time Linking*: Wenn zu einer DLL eine Import-Library existiert, so wird gegen diese Library verlinkt und die DLL zur Laufzeit automatisch geladen. Ein explizites Abfragen von Funktionseinsprungspunkten ist nicht erforderlich.
- *Explicit Run-Time Linking*: Hierbei werden die Windows-API-Funktionen `LoadLibrary`, `GetProcAddress` und `FreeLibrary` benutzt, um zur Laufzeit den DLL-Handle und die einzelnen Funktionspointer zu bekommen, über die dann die Routinen der DLL aufgerufen werden können. Dies ist das flexiblere Konzept, da der Client auf nicht vorhandene DLLs reagieren kann.

4.3.2 Klassenbibliotheken

Eine Klassenbibliothek ist eine Gruppierung von entweder direkt oder indirekt per Vererbung nutzbaren Klassen, wobei die Klassen flexibel wiederverwendbare Bausteine darstellen, die jedoch den Kontrollfluss bei der einbindenden Applikation belassen ([Züllighoven05, S. 90]). Klassenbibliotheken gibt es meist für die Sprachen C++ und Java. Eine C++-Klassenbibliothek besteht aus Headern zur Typen-Deklaration und Objektcode in einer oder mehreren LIBs oder DLLs (plus Type Libraries). Die Verfügbarkeit des Sourcecodes einer Klassenbibliothek wird zwar von Einigen als äußerst nützlich empfohlen ([Sing97]), ist aber unter Aspekten von Copyright und geistigem Eigentum als problematisch anzusehen. Vorhandener Sourcecode kann allerdings durch die Möglichkeit der Neukompilation die Portabilität einer Klassenbibliothek fördern. Auf einem höheren Abstraktionsniveau, oberhalb von Klassenbibliotheken, sind (Klassen-)Frameworks angesiedelt, die Referenzapplikationsstrukturen und Entwurfsmuster auf einer Metaebene definieren (vgl. [Buschmann96]). So könnte man die Windows-Klassenbibliotheken MFC und ATL durchaus auch als Framework bezeichnen, wobei allerdings die Trennung zwischen beiden Begriffen fließend ist.

Die Abgrenzung einer Klassenbibliothek zu einer Komponentenbibliothek anhand der in 4.2.1 aufgeführten Kriterien ist insbesondere dann schwierig, wenn die Klassenbibliothek nicht als statische Linklibrary (LIB), sondern in bereits ausführbarer Form vorliegt, da die ausführbare Form ein typisches Komponentenmerkmal ist. Unter Windows ist dies vor allem bei .NET-„Klassenbibliotheken" der Fall, die in der Regel als DLLs vorliegen und als solche zwar nicht unmittelbar binär, aber doch mittels der CLR ausführbar sind. Auf solche „.NET-Klassenbibliotheken" treffen alle Kriterien für Komponenten (vgl. 4.2.1 , [Szyperski02, S.

41]) zu, so dass man eigentlich korrekterweise von „Komponentenbibliotheken“ sprechen müsste.
Klassenbibliotheken existieren für sehr generische Aufgaben, wie die *Standard Template Library (STL)* in C++, aber auch für sehr spezifische Problemstellungen. Für den Optimierungsbereich enthält beispielsweise [Voß02] eine Sammlung von „Optimization Software Class Libraries”.

4.3.3 Component Object Model

Das Component Object Model ist die softwarearchitektonische Grundlage der bereits in 4.2.3.3 erwähnten OLE-Technologien. In seinem Kern spezifiziert COM, wie auf die Funktionalitäten einer Komponente zugegriffen wird und ermöglicht so die systemweite Interoperabilität von binären Softwarebausteinen. Neben diesen Spezifikationen, die das eigentliche Komponentenmodell bilden, existiert unter Windows noch die COM-API, eine Systembibliothek von Hilfsfunktionen für das Management von COM-Komponenten.

4.3.3.1 Das Basisinterface IUnknown

COM bezieht sich ausschließlich auf Schnittstellen, nicht auf die Implementation, d.h. COM-Komponenten können in beliebigen Sprachen und auf beliebige Weise konstruiert werden, solange sie bezüglich ihrer Schnittstellen spezifikationskonform sind. Eine COM-Komponente kann eine oder mehrere Schnittstellen implementieren, sie muss jedoch immer die für alle Komponenten verbindliche Schnittstelle `IUnknown` enthalten. `IUnknown` besitzt drei Methoden:

```
QueryInterface(REFIID riid, void** ppvObject);
AddRef();
Release();
```

Anders als in C++, wo es Pointer auf Objektinstanzen gibt, wird unter COM mit Schnittstellenpointern gearbeitet. Bei Instanzierung eines neuen COM-Objekts erhält der Client zunächst einen Zeiger auf dessen stets vorhandenes `IUnknown`-Interface. Um zu erfahren, ob die Komponente eine bestimmte Schnittstelle unterstützt, ruft der Client dann die Methode `QueryInterface` auf. Dabei gibt er die Interface-ID (IID) der gewünschten Schnittstelle an. Die IID ist ein zufällig erzeugter, aber global eindeutiger 128-Bit-Wert, den der Client beispielsweise aus einem Header hat, in dem das gewünschte Interface beschrieben ist. Besitzt die Komponente die zur IID passende Schnittstelle, so gibt sie dem Client über das zweite

Argument von `QueryInterface` einen Pointer auf die besagte Schnittstelle zurück, ansonsten wird ein Fehler signalisiert. Mit diesem Pointer können anschließend die Funktionen der Schnittstelle aufgerufen werden (s. Abbildung 17).

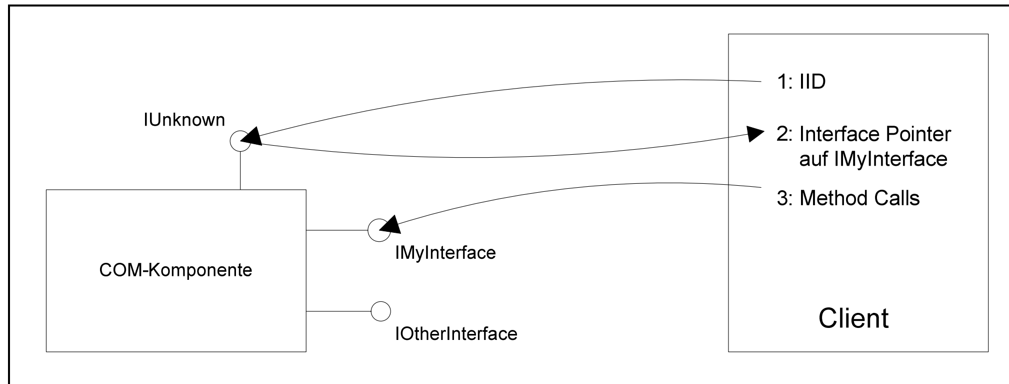


Abbildung 17: COM Interface-Referenzierung

Die beiden anderen Methoden von `IUnknown` dienen als Referenzzähler (zum allgemeinen Prinzip der Referenzzählung vgl. etwa [Coplien04]). Wann immer ein Client eine Referenz auf eine Schnittstelle erhält, erhöht die Komponente intern einen Zähler. Dies kann implizit beim Aufruf von `QueryInterface` oder explizit durch Aufruf von `AddRef` geschehen. Benötigt der Client die Komponente bzw. deren Interface nicht mehr, lässt er sie mit `Release` los, wodurch der Referenzzähler um eins herabgesetzt wird. Hat der Referenzzähler Null erreicht, leitet die Komponente ihre eigene Destruktion ein.

Der Aufruf einer Schnittstellenmethode erfolgt über ein COM-spezifisches Konstrukt, die *VTable*, die ihren Ursprung im Layout virtueller Klassen in C++ hat. Diese Tabelle enthält Funktionspointer auf die Startadressen der implementierten Methoden. Mittels eines Interface-Pointers und der Kenntnis über die relative Position der gewünschten Funktion (erhältlich etwa aus einem Header), lassen sich somit die Implementationsroutinen aufrufen. Da ein Interface immer vom Basisinterface `IUnknown` abgeleitet werden muss, enthalten die VTables auf ihren ersten Positionen immer dessen drei Standardfunktionspointer.

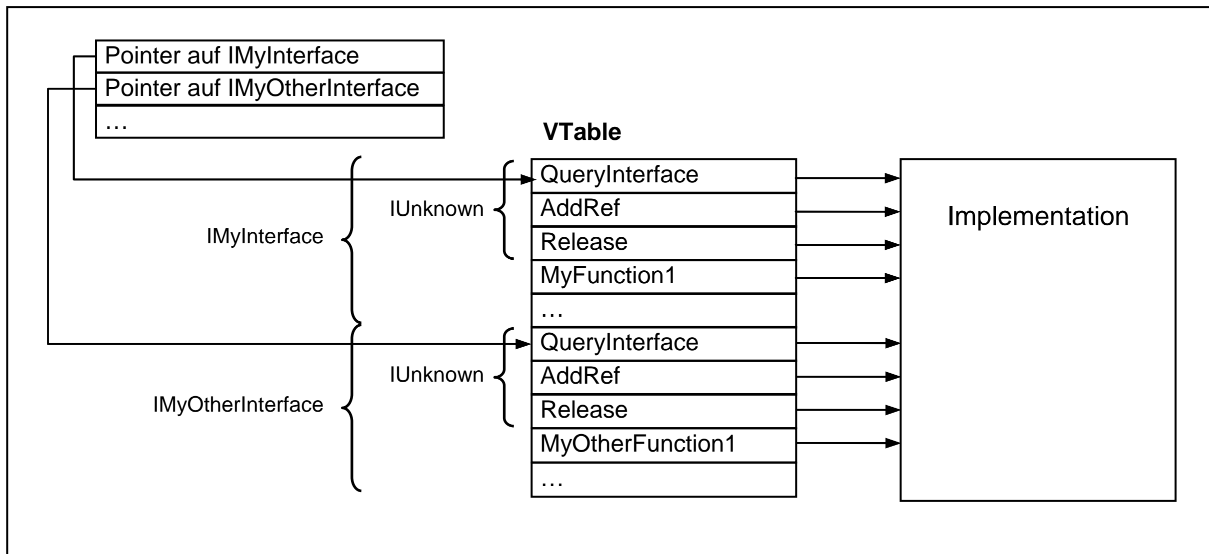


Abbildung 18: Interfaces und Vtables

4.3.3.2 Class Factories

Zur Erzeugung einer neuen Komponenteninstanz bedarf es in COM einer so genannten *Class Factory* (zum allgemeinen Factory Pattern vgl. [Gamma95]) . Eine Class Factory ist ihrerseits eine normale COM-Komponente, die das Interface `IClassFactory` besitzt. Dieses Interface enthält unter anderem die Methode `CreateInstance`, die unter Angabe einer Interface-ID einen Pointer auf das so bezeichnete Interface einer neuen Komponenteninstanz als Rückgabeargument liefert. Um die Class Factory einer Komponente zu erhalten, gibt es in jeder COM-DLL eine Funktion `DllGetClassObject` mit folgender Signatur:

```
STDAPI DllGetClassObject(
    REFCLSID rclsid,
    REFIID riid,
    LPVOID * ppv);
```

Die COM-API lädt eine COM-DLL mit `LoadLibray` und ruft `DllGetClassObject` auf, wobei die eindeutige Class-ID der Komponente und die Interface-ID der Class Factory (`IID_IClassFactory`) angegeben werden müssen, als Rückgabewert liefert die DLL einen Pointer auf ein Class Factory Interface. Dieser Pointer kann dann benutzt werden, um eine Instanz der eigentlichen Komponente zu erzeugen. Dieser umständliche Prozess (DLL laden, Class Factory Pointer holen, instanzieren) kommt bei Benutzung von COM-Komponenten ständig vor und wird deshalb in der zentralen COM-Instanzierungsfunktion `CoCreateInstance` der API zusammengefasst:

```
STDAPI CoCreateInstance(
```

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

Angegeben werden Klassen-ID, Interface-ID und der Instanzierungskontext (im Prozess, außerhalb des Prozesses oder Remote), worauf bei erfolgreicher Instanzierung der Komponenten ein Zeiger auf das gewünschte Interface (`ppv`) zurückgegeben wird. Eine detailliertere Beschreibung und sämtliche Informationen über weitere COM-API-Funktionen gibt [Microsoft07c].

Zur eindeutigen Identifizierung von COM-Bibliotheken, Klassen und Interfaces werden stets 128 Bit-Werte benutzt, die als TypeLibID, ClassID, InterfaceID (IID) oder generell als UUID (Universally Unique Identifier) oder GUID (Globally Unique Identifier) bezeichnet werden. IIDs für eigene Schnittstellen erzeugt der Programmierer selbst mit dem Tool guidgen.exe, während die IIDs von Standardschnittstellen vorgegeben sind, so ist z.B. die IID von `IUnknown` in Hex-Darstellung `00000000-0000-0000-C000-000000000046`. Diese IDs werden zusammen mit anderen Informationen, wie dem Speicherort der Komponenten, ihrem textualen Namen, ihrer Versionsnummer etc., in der Registry abgelegt. Eine korrekte Registrierung ist daher Voraussetzung für das einwandfreie Funktionieren der COM-Mechanismen. Die (Selbst-)Registrierung erfolgt über die in jeder Komponenten-DLL enthaltenen Standardfunktionen `DllRegisterServer` und `DllUnRegisterServer`.

4.3.3.3 Die Schnittstelle IDispatch

Neben `IUnknown` existiert noch eine weitere zentrale Schnittstelle: `IDispatch`. Dieses Interface erlaubt die späte Bindung eines Client an eine Komponente zur Laufzeit, was insbesondere für Skript- und Koordinationssprachen wie Visual Basic von Bedeutung ist (vgl. [Griffel98, S. 307 ff.]). Eine COM-Komponente, die `IDispatch` implementiert, wird als *Automatisierungs-Komponente* oder *Automatisierungs-Server* bezeichnet. `IDispach` ist wie alle Interfaces von `IUnknown` abgeleitet und besitzt darüber hinaus folgende Methoden:

`GetTypeInfoCount`	Anzahl der TypeInformation-Interfaces, die die Komponente anbietet
`GetTypeInfo`	Holt einen Zeiger auf ein TypInfo-Interface (ITypeInfo)
`GetIDsOfNames`	Wird mit einem Array von Methodennamen aufgerufen und gibt die dazu passenden Methoden-IDs (DispIDs) zurück, falls die Mehtode vorhanden ist

Invoke	Dient dem Aufruf einer Methode über deren DispID. Übergeben werden auch deren Argumente in einer speziellen Struktur. Zurückgegeben werden Rückgabewerte soewie evtl. Fehler- und Exception-Informationen.

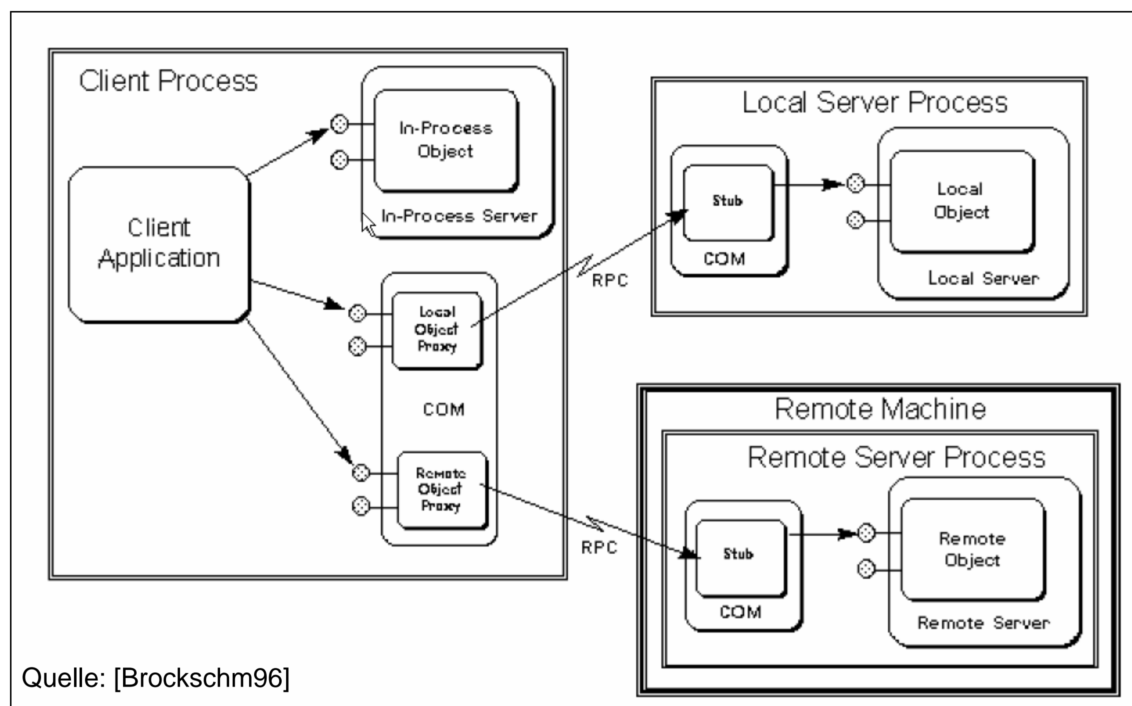
Tabelle 13: Idispatch-Methoden

Die Argumente einer Komponentenmethode, die über IDispatch.Invoke aufgerufen wird, müssen in Form einer bestimmten Struktur (DISPPARAMS) übergeben werden, die im Wesentlichen ein Array von VARIANTs darstellt. Dies hat eine wichtige Implikation, die auch für das spätere Design der Modellierungskomponenten von Bedeutung ist: Über IDispatch können nämlich nur solche Daten übergeben werden, die über den Typ VARIANT abbildbar sind. Hierzu zählen neben SHORT und LONG, FLOAT und DOUBLE noch eine Reihe weiterer Typen. Jedoch sind keineswegs alle in C++ verwendbaren Datentypen erlaubt. So müssen Strings etwa vom Typ BSTR sein und Arrays können nur als SAFEARRAY übergeben werden. Weiterhin können Pointer auf IUnknown- und IDispatch-Interfaces als Parameter verwendet werden, nicht jedoch beliebige Pointertypen wie in C++ oder in nicht automatisierungskonformen Interfaces. Weiterhin legen die Spezifikationen von IDispatch-Interfaces fest, dass nur ein einziger Rückgabeparameter ([out, retval]) erlaubt ist, was ebenfalls einschränkend beim Design einer automatisierungskonformen Komponentenbibliothek ist. Wird sowohl eine VTable-, als auch eine IDispatch- Schnittstelle zur Verfügung gestellt, so redet man von einem *Dualen Interface* (vgl. etwa [Rogerson97] oder [Denning97]).

4.3.3.4 Lokale und Remote-Ausführung

Ein wichtiges Feature von COM ist Transparenz in Bezug auf den Ausführungsort des Codes. So kann eine COM-Komponente als logischer Server im selben Prozessraum wie der Client, in einem anderen Prozessraum oder sogar auf einem andern Rechner liegen, ohne dass der Client unterschiedliche Ansteuerungsmechanismen verwenden müsste. Der Zugriff auf einen In-Process-Server ist problemlos, da die Schnittstellenzeiger unmittelbare Adressbedeutung haben und keiner weiteren Umsetzung bedürfen. Müssen aber Prozess- oder Rechnergrenzen überschritten werden, so ist dies nur mit *Marshalling* möglich (Abbildung 19). Dabei überträgt das COM-Framework Aufrufe, Parameter und Rückgabewerte zwischen den Prozessen und Rechnern über Remote Procedure Calls (RPCs). Im Prozessraum des Clients befindet sich ein *Proxy*, der die Interfaces der Komponenten nachbildet, aber nicht die eigentliche Implementation enthält, sondern lediglich Aufrufe und Parameterdaten an das Framework weitergibt, das seinerseits diese an einen im Prozessraum des Servers befindlichen *Stub* übermittelt. Dieser Stub agiert wie ein lokaler Pseudo-Client, indem er die echten Interfaces der

Komponente aufruft. Rückgabewerte schlagen den umgekehrten Weg ein. Bei Remote Servern tritt zusätzlich noch das Problem auf, dass aufgrund der Netzwerkverbindung die Kommunikation zwischen Proxy und Stub unterbrochen werden kann. Ein Proxy liefert in diesem Fall einen `Disconnected` Errorcode, ohne dass es zu einem Absturz kommt, und umgekehrt muss sich der Client in gewissen Zeitabständen beim Server melden. Tut er das nicht, so leitet der Server seine eigene Destruktion ein, um seine Ressourcen wieder freizugeben.



Quelle: [Brockschm96]

Abbildung 19: Ortstransparenz bei COM

4.3.3.5 Objektorientierte Charakteristika in COM

Im Folgenden soll kurz darauf eingegangen werden, wie COM die in Abschnitt 4.1 aufgeführten Charakteristika objektorientierter Strukturen (nach [Booch94, S. 25 ff]) umsetzt:

- Abstraction: Ein COM-Objekt bildet eine reale Entität ab, indem es diese mit Methoden, Eigenschaften und Ereignissen nachstellt. Dabei sind Eigenschaften und Ereignisse allerdings Konstrukte, die die Client-Programmiersprache (etwa VB oder Delphi) erzeugt, denn auf COM-Ebene existieren nur Methodenaufrufe, wobei Eigenschaften durch Get- und Set-Methoden und Events über *Connection Points* rein methodenbasiert realisiert werden.
- Encapsulation: COM spezifiziert eine radikale Kapselung der Implementation einer Komponente mit ausschließlichem Zugriff über Interfaces. Public Member-Variablen oder *Friend*-Zugriffe wie in C++ gibt es nicht.

- Modularity: Komponentenbibliotheken, Komponenten und Interfaces bilden in COM die Modularisierungseinheiten, aus denen größere Softwarearchitekturen konstruiert werden können. Diese Module sind die eigentlichen Funktionsträger, die lediglich vom „Glue“ ([Nierstrasz97], [Crnkovic02]) einer (Scripting-)Sprache zusammengefügt werden müssen.
- Hierarchy: Zur Hierarchisierung benutzt COM einfache Interfacevererbung, aber keine Implementationsvererbung. So sind etwa alle COM-Interfaces vom Basisinterface `IUnknown` abgeleitet und weitere Ableitungsketten sind möglich: Beispiel hierfür ist `IDispatch`, ebenfalls von `IUnknown` abgeleitet, dient aber seinerseits als Basisinterface für alle Automatisierungskomponenten. Implementationshirarchisierung ist lediglich in Form von *Containment* und *Aggregation* vorhanden. Containment und Aggregation sind Mechanismen zur funktionalen Einverleibung einer inneren Komponente durch eine äußere (Details siehe [Microsoft07d]).
- Typing: Nimmt man den Grad der Austauschbarkeit unterschiedlicher Daten- bzw. Entitäts-Typen als Maß starker oder schwacher Typen-Stringenz (vgl. [Booch94, S. 63]), so kann COM als schwach typisiert eingestuft werden – vor allem, wenn eine COM-Komponente automatisierungskonform ist. Denn dann müssen die Typen aller Übergabeparameter durch den Typ `VARIANT` abbildbar sein, was zu Mehrdeutigkeiten insbesondere bei der Übergabe von Interfacepointern führen kann, denn `VARIANT` sieht lediglich `IDispatch` und `IUnknown` als Interfacetypen vor. Solange ein Interfacepointer *p* also von einem dieser Interfaces abgeleitet ist, kann eine genaue Typüberprüfung erst zur Laufzeit vorgenommen werden.
- Concurrency: COM basiert auf einem komplizierten Threadingmodell, das mit so genannten *Apartments* arbeitet, die eine logische Zuordnung von Objektinstanzen zu Threads darstellen. Ähnlich, wie ein Haus aus mehreren Apartments bestehen kann, so kann in diesem Modell ein Prozess ebenfalls mehrere Apartments beinhalten, wobei ein Apartment eine rein logische Entität ist, und weder mit einem Thread noch mit einem Objekt gleichzusetzen ist. Man kann im Wesentlichen zwei Arten von Apartments unterscheiden (vgl. [Rogerson97, S 311 ff.]):
    - Single Threaded Apartments (STA): In einem STA können eine oder mehrere Instanzen eines COM-Objekts existieren, die aber nur von einem einzigen Thread direkt angesprochen werden können. Nur der Erzeuger-Thread, der das Apartment mit der API-Funktion `CoInitialize` initialisiert hat, kann mittels Interfacepointern direkt auf dessen COM-Objekte zugreifen, während alle

andern Threads nur über vom COM-Framework vermitteltes Windows-Messaging, indirekt via Marshalling mit den Objektinstanzen kommunizieren können.

- o Multi Treaded Apartments (MTA): Einem MTA können ein oder mehrere Threads zugeordnet sein, die alle auf die COM-Objektinstanzen des MTA direkt, d. h. ohne Marshalling, zugreifen können. COM-Objekte in einem MTA müssen also selbst sicherstellen, dass keine Probleme durch gleichzeitige Zugriffe mehrerer Threads entstehen (z.B. durch Verwendung von Synchronisationsmechanismen wie Events, Mutexes, Semaphoren etc.). Die Programmierung einer MTA COM-Komponente ist daher erheblich anspruchsvoller als die einer STA-Komponente.

Innerhalb eines Prozesses können beide Threading-Formen parallel verwendet werden, ein Prozess kann maximal ein MTA sowie zusätzlich ein oder mehrere STA enthalten. COM-Komponenten publizieren durch einen entsprechenden Registry-Wert, welches Threading-Modell sie unterstützen. Weitere Informationen, insbesondere zum Threading-Verhalten von Proxies und Stubs finden sich in [Microsoft03].

- Persistence: Das Prinzip der *Strukturierten Speicherung* (*Structured Storage*) bildet einen zentralen Bestandteil der COM-/OLE-Architektur und dient der hierarchischen Speicherung von Informationen in ein einzelnes File (vgl. [Brockschm97, Kap. 7]). Dabei verhält sich die einzelne Datei wie eine Art Filesystem, das aus Ordnern (*Storage* genannt) und Files (*Streams* genannt) besteht. So können die Zustandsdaten von unterschiedlichen Komponenten innerhalb eines OLE-Containers zusammenhängend gespeichert und auch wieder ausgelesen werden. Eine Komponente, die Structured Storage zur eigenen Persistenzsteuerung verwendet, muss einige OLE-Standardinterfaces (u.a. `IPersistStorage`, `IPersistStream`) implementieren, deren Methoden vom Container aufgerufen werden können. In einer Entwicklungsumgebung wie Visual Studio werden beispielsweise die Eigenschaften von COM-Controls mithilfe der OLE-Persistenzmechanismen gespeichert und wieder hergestellt.

4.3.3.6 Property Pages

Schließlich sei noch ein OLE-Feature erwähnt, das für die Benutzbarkeit von programmierbaren Komponenten in grafischen Entwicklungsumgebungen wichtig ist, und von dem auch die MOPS-Komponenten Gebrauch machen: Mithilfe von *Property Pages* können die Eigenschaften einer Komponente zur Entwurfszeit für den Entwickler bequem und übersichtlich

editierbar gemacht werden. Eine Property Page ist dabei nichts anderes als eine eigenständige COM-Komponente, die eine grafische Oberfläche besitzt und zumindest das OLE-Standard-interface `IPropertyPage` implementieren muss. Eine COM-Komponente kann eine oder mehrere Property Pages besitzen und muss diese über sein Standardinterface `ISpecifyPropertyPages` abfragbar machen. Property Pages werden in einem modalen Dialog, den so genannten *Property Frame* in einzelnen Tabs dargestellt. Weitere Details gibt z. B. [Brockschm97, Kap. 16].

4.3.3.7 Bibliotheken zur COM-Programmierung

Obwohl es COM-/DCOM-Implementationen für andere Betriebssysteme gibt (für UNIX beispielsweise mit *EntireX* der *Software AG*, [SwAG07]), beschränkt sich die Verbreitung von COM faktisch jedoch auf die Windows-Welt. Dort ist die Durchdringung des Systems mit COM-basierten Technologien jedoch ganz erheblich, und es ist in den vielen Programmiersprachen möglich, COM-Komponenten zu entwickeln. Sehr einfach kann dies mit VB6 bewerkstelligt werden, aber auch Delphi und sogar Fortran bieten COM-Unterstützung. Die flexibelsten und mächtigsten Möglichkeiten der COM-Entwicklung bietet jedoch eindeutig C++. Dort stehen neben der kaum praktikablen, „manuellen" COM-Entwicklung im Wesentlichen zwei Bibliotheken zur Verfügung: Die *Microsoft Foundation Classes (MFC)* und die *Active Template Library (ATL)*[1]. Die MFC dienten ursprünglich der objektorientierten Kapselung der Windows-API, was auch immer noch ihre primäre Aufgabe ist. COM-Support wurde nachträglich hinzugefügt und besteht aus einer Sammlung von Klassen, die als Basisklassen für eigene COM/OLE-Implementationen dienen können: Die Klasse `CCmdTarget` implementiert beispielsweise `Iunknown`, und `COleDispatchImpl` implementiert das Standardinterface `IDispatch`. Weiterhin bietet MFC eine Anzahl von Makros zur Unterstützung der COM-Programmierung, wie z.B. für Interface-Tabellen und Message-Maps. Die ATL hingegen ist eine sehr komplexe C++-Template-Bibliothek, die explizit für COM entwickelt wurde. Wie MFC bietet auch ATL Basisklassen zur Wiederverwendung für eigene COM-Komponenten: So implementieren etwa die Klassen `CComObjectRootEx` und `CComObjectRootBase` die Standardschnittstelle `Iunknown`, und `IDispatchImpl` implementiert `IDispatch`. Weiterhin gibt es eine Vielzahl weiterer Hilfsklassen, die teils COM-bezogen sind, teils aber einfach nur häufig benötige Funktionalitäten bereitstellen, wie

[1] Für beide Frameworks existiert neben der Microsoft Referenz-Dokumentation ([Microsoft07e], [Microsoft07i]) eine große Fülle von Literatur, wobei hier nur als Beispiele [Shepherd96, Kap 11 ff.] für die MFC und das relativ neue [Tavares06] für ATL aufgeführt seien.

z.B. Zeichenkettenbehandlung (`CComBSTR`, `CString`). Ein wichtiger praktischer Unterschied zwischen den Bibliotheken ist, dass die MFC aufgrund ihres umfangreichen Klassengerüsts einen gewissen Overhead hat, der dazu führt, dass die mit ihr erstellten Komponenten größer sind als solche, die mit der schlankeren ATL erzeugt wurden. Einen guten und detaillierten Vergleich beider Bibliotheken findet man bei [Shepherd00], und [Onion98] leitet her, warum die Architektur der ATL mit ihren teils rekursiven Templates für effizientes COM erforderlich ist.

Zur Beschreibung der Interfaces existiert für COM die *Interface Definition Language* (*IDL* bzw. *MIDL* für *Microsoft IDL*, vgl. [Microsoft07g]), ähnlich wie es sie auch mit *OMG IDL* für CORBA gibt. Der MIDL-Compiler erzeugt aus MIDL-Sourcecode automatisch C-Code für Proxies und Stubs und kann ebenso Type Libraries und Registrierungsinformationsdateien erzeugen. Type Libraries enthalten Informationen über exponierte Komponenten. Sie können eigenständige Dateien mit der Endung *.tlb* oder in eine Komponentenbibliothek eingebettet sein. Die COM-/OLE-Spezifikation sieht u.a. die Interfaces `ITypeLib` und `ITypeInfo` vor, mit denen ein Client die in Type Libraries enthaltenen Informationen abfragen kann. Wird beispielsweise eine Komponentenbibliothek in Form einer DLL von einer Entwicklungsumgebung wie Visual Basic referenziert, so werden die genannten Interfaces benutzt, um dem Entwickler zur Entwurfszeit Typ-Informationen über die Bibliothek zu geben. Type Libraries können aber auch zur Laufzeit benutzt werden, um Informationen zu COM-Komponentenfunktionen zu erhalten, die dann via `IDispatch` aufgerufen werden (*Late Binding*).

4.3.4 .NET als neues Komponenten- und Schnittstellenparadigma

.NET ist Microsofts neues Plattformparadigma und umfasst als solches Laufzeitplattform, Sicherheitsarchitektur, Middleware für verteilte Anwendungen inklusive zugehöriger Protokolle, neue Programmiersprachen (z.B. VB.NET, C#), Typensystem sowie eine neue Komponenten- und Schnittstellenarchitektur[1].

Das .NET-Framework setzt auf bestehenden Windowsplattformen auf und enthält in seinem Kern die *Common Language Runtime (CLR)*, die ähnlich wie eine Java Virtual Machine, maschinenunabhängigen Zwischencode (*Common Intermediate Language, CLI*) ausführt, der

[1] Die Literatur zum Thema .NET ist kaum zu überschauen, so dass für diesen Abschnitt nur einige Titel exemplarisch genannt werden können: Einen knappen, allerdings auch sehr oberflächlichen Überblick über .NET gibt [Zwin05, S. 66 ff.]. Etwas umfangreicherer ist die Einführung von [Wang05, S. 194 ff.]. Sehr detailliert widmet sich [Box02] der CLR. [Löwy05] fokussiert auf die komponentenorientierte .NET-Entwicklung und behandelt in der 2. Auflage auch die Neuerungen des Frameworks 2.0. Eine anspruchsvolle Übertragung von Pattern-orientierter Programmierung auf .NET ist [Gross05]. Die komplette .NET-Referenz und weitere Informationen finden sich bei Microsoft selbst ([Microsoft07f] und verlinkte Seiten).

von .NET-kompatiblen Compilern (C#, Delphi.NET etc.) erzeugt wurde. Die CLR übersetzt während der Laufzeit mit ihrem *Just In Time Compiler (JIT)* CIL-Code in Maschinencode. Daneben übernimmt die CLR auch das Code-Management, das neben der Durchsetzung von Sicherheitsfeatures vor allem in der Speicherverwaltung (Garbage Collection etc.) besteht.

Weiterhin besitzt .NET eine umfassende Framework-Klassenbibliothek, die in unterschiedliche Namespaces organisiert ist. Unterhalb des Root-Namensraums `System`, befinden sich Sub-Namespaces, die Klassen für bestimmte Funktionalitäten gruppieren. Beispiele sind:

- `System.IO`: Ein- und Ausgabe
- `System.XML`: XML Unterstützung
- `System.Data`: ADO.NET Datenbankunterstützung
- `System.Windows.Forms`: GUI-Building-Klassen

.NET-Sprachen zeichnen sich durch eine außergewöhnliche Interoperabilität untereinander aus, so dass die Benutzung einer in Sprache A erstellen Komponente durch Sprache B unmittelbar möglich ist. Darüber hinaus ist in .NET sogar sprachübergreifende Vererbung möglich: Eine C#-Klasse kann z.B. von einer VB.NET-Klasse erben. Dies ermöglicht ein für alle .NET-Sprachen verbindliches, gemeinsames Typensystem (*Common Type System, CTS*). Zwar war vor .NET teilweise bereits Mixed-Language-Programmierung möglich, erforderte jedoch tiefgreifende Kenntnisse der Aufrufkonventionen sowie der internen Abbildung von Datentypen und verlangte entsprechende Konvertierungsmechanismen (Beispiel: Aufruf einer Fortran-Routine aus C/C++).

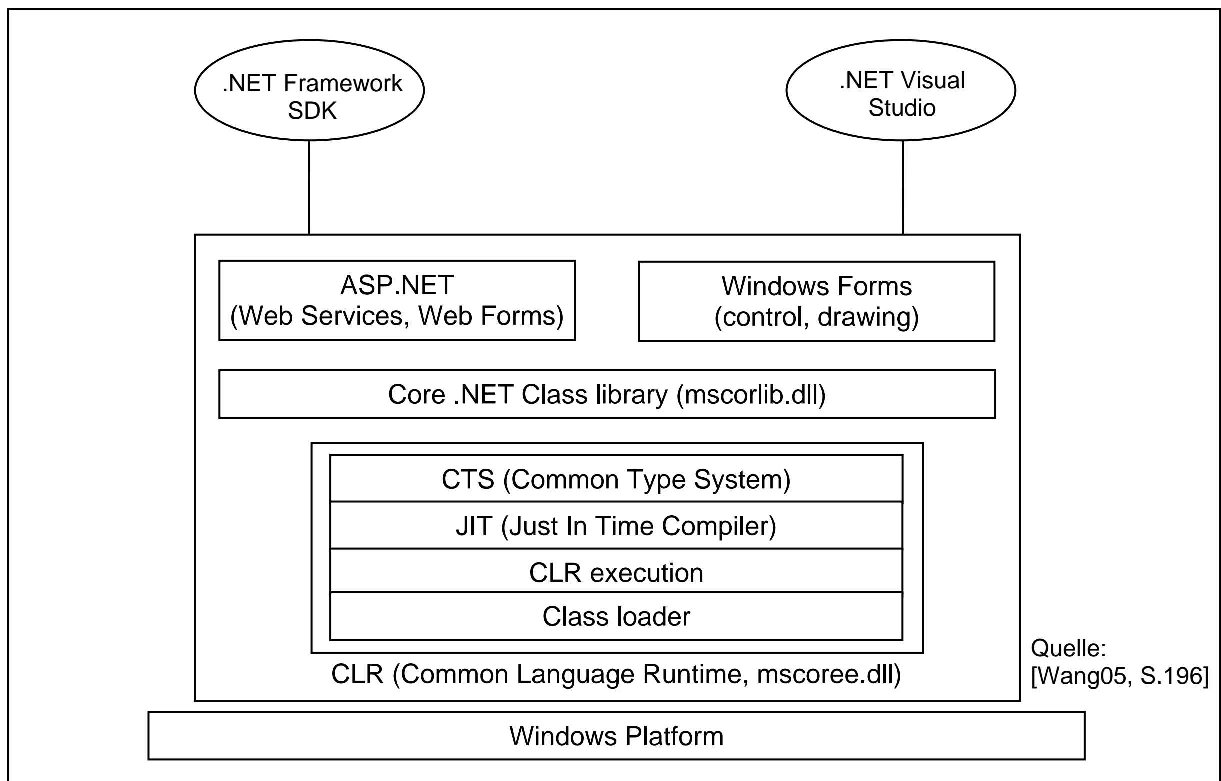


Abbildung 20: .NET Framework

Herkömmlicher Assemblercode enthält keine Metainformationen über Klassen, Operationen, Attribute etc. – CIL-Code hingegen tut dies und erlaubt während der Laufzeit über die so genannte *Reflection API* des .NET Frameworks Zugriff auf diese Informationen.
Das .NET-Komponentenmodell sieht sprachübergreifende, binäre CIL-Kompatibilität vor und benutzt als Distributionseinheiten *Assemblies*. Eine Assembly besteht aus folgenden Teilen:

- Manifest (Name der Assembly, Version, zur Assembly gehörige Files etc.)
- Metadaten der Module
- CIL-Code der Module
- Ressourcen (Icons, Bitmaps etc.)

Eine Assembly kann sich in einer oder mehreren Dateien befinden; die Dateien können DLLs, EXEs, Netmodules oder Ressourcefiles sein. Eine Assembly kann einen oder mehrere Namespaces enthalten, ein Namespace kann sich aber auch über mehrere Assemblies erstrecken. Ein Modul besteht aus CIL-Code und Metadaten, enthält aber im Gegensatz zu Assemblies kein Manifest und dient als Input für den Linking-Vorgang zur Erzeugung einer Assembly. Ein Modul kann eine oder mehrere Klassen enthalten, kann aber nicht dynamisch geladen werden. Module besitzen standardmäßig die Endung *.netmodule*.

Man unterscheidet visuelle und non-visuelle .NET-Komponenten, wobei die visuellen Komponenten in der Entwicklungsumgebung aus der Toolbox per Drag & Drop auf einem Windows-Form platziert werden können. .NET-Komponenten können lokal ausführbar sein (DLL), auf dem selben Rechner (EXE) oder remote (EXE). Lokale Ausführung heißt in .NET – anders als in COM – Ausführung innerhalb der selben *Application Domain* wie der Client. Eine Application Domain ist kein Thread, sondern eine Art Sub-Prozess innerhalb eines Prozesses, der unabhängig vom Hauptprozess gestartet und gestoppt werden kann und der Ausführungsisolierung dient.

Was die Distribution von .NET-Komponenten anbelangt, so unterscheidet man private und gemeinsame (*shared*) Komponenten. Private Komponenten arbeiten nur für einen Client, haben von diesem Kenntnis, unterstützen keine Versionierung und befinden sich meist im gleichen Verzeichnis wie der Client. Gemeinsame Komponenten können von unterschiedlichen Clients benutzt werden, sie unterstützen Versionsmanagement und müssen im Global Assembly Cache (GAC) liegen. Solche Komponenten müssen außerdem über eine digitale Signatur verfügen, über die zur Laufzeit das Zusammenpassen von Client und Komponente überprüft wird.

4.3.5 Interoperabilität zwischen COM und .NET

.NET enthält diverse Mechanismen, um sich in bestehende Systemlandschaften zu integrieren und vorhandene Bibliotheken zu nutzen. Hier müssen vor allem die Interoperabilitätsmechanismen zu COM erläutert werden, wie sie etwa bei [Troelsen02] ausführlich beschrieben sind. Das .NET-Framework erlaubt den Zugriff aus gemanagtem .NET-Code auf nicht gemanagten Win32-Code. Solche Zugriffe laufen über die *Platform Invocation Services (PInvoke)* ab, die im Namespace `System.Runtime.InteropServices` lokalisiert sind und vor allem das Marshalling von gemanagten Datentypen in ihre nicht gemanagten Gegenstücke übernehmen. Ein einfaches Beispiel hierfür wäre der Aufruf einer Win32-API-Routine aus einem VB.NET Programm.

	COM	**.NET**
Code	Unmanaged, native Code.	Managed Intermediate Language. JIT-Compilation durch CLR, automatische Speicherverwaltung, Allokation mit Garbage Collection, Bounds Check etc.
Objektmodell	Interfaces	Objekte
Lokalisierung	DLLs oder EXEs mit Registrierung	Assemblies ohne Registrierung
Type Info	Type Libraries / IDL	Metadata in Assemblies

Datentypen	OLE Automation Compatible Types (bei Zugriff von Automatisierungs-clients wie VB6)	Common Type System (CTS)
Error handling	HRESULTS	Exception Objekte
Vererbung	Einfache Interface-Inheritance Aggregation und Containment	Interface- und Implementation-Inheritance
Framework	-	.NET Framework
Threadding	Komplizierte, unterschiedliche Threading-Modelle (Appartment etc.)	Einfaches, uniformes Threadding

Tabelle 14: Unterschiede zwischen COM- und .NET-Komponenten

Komplizierter ist der Zugriff aus .NET auf COM-Komponenten und umgekehrt. Da zwischen beiden Welten eine Reihe von gravierenden Unterschieden (Tabelle 14) bestehen, müssen die Aufrufe über spezielle Wrapper-Objekte erfolgen, welche Transformationsaufgaben übernehmen. Greift ein .NET-Client auf einen COM-Server zu, so geschieht dies über einen *Runtime Callable Wrapper (RCW)*. In umgekehrter Richtung spricht man von einem *COM Callable Wrapper (CCW)*.

Zur Konstruktion eines RCW wird zunächst einmal die Typinformation der COM-Komponente benötigt, die z.B. einer Type Library entnommen werden kann und aus der die Metadaten für .NET erzeugt werden. Dies macht die .NET- Entwicklungsumgebung VS2005 beim Hinzufügen einer COM-Referenz automatisch. Wie in Abbildung 21 schematisiert, bietet der RCW dem .NET-Client nur die eigentlichen User-Interfaces der COM-Komponente an, die Basisschnittstellen `IUnknown` und `IDispatch` werden nicht durchgereicht, da diese in .NET keine Bedeutung hätten. Ihre Bedienung übernimmt der RCW, der dafür sogen muss, dass das COM-Objekt solange aktiv bleibt, d.h. einen Referenzzähler größer Garbage Collector Null hat, solange der RCW in Benutzung ist. Wird schließlich der RCW „entsorgt", so muss er alle Referenzen auf das COM-Objekt mit der `IUnknown`-Funktion `Release` freigeben. Darüber hinaus obliegen dem RCW aber noch weitere Transformationsaufgaben:

- Exception Management: COM-Returncodes (HRESULTS) müssen in .NET-Exceptions umgesetzt werden, und – falls vorhanden – müssen Fehlerbeschreibungen über das COM-Interface `IErrorInfo` ausgelesen und an ein Exception-Objekt übertragen werden.
- Marshalling: Typen, die in COM und .NET die gleiche interne Darstellung haben (isomorphe Typen) benötigen keine Transformation, andere Typen müssen hingegen umgeformt werden, so dass eine transparente Nutzung des gekapselten COM-Objekts möglich ist. Dies betrifft Typen wie z.B. BOOLEAN, CHAR, STRING, ARRAY etc.
- Umsetzung von `[out,retval]` Werten in .NET-Rückgabewerte

- Thread Management: .NET kennt keine Thread-Apartments und muss für COM passende STA- oder MTA-Apartments erzeugen.
- Unterstützung der späten Bindung (*Late Binding*) über COM `IDispatch`-Interfaces und .NET `Type`-Objekte

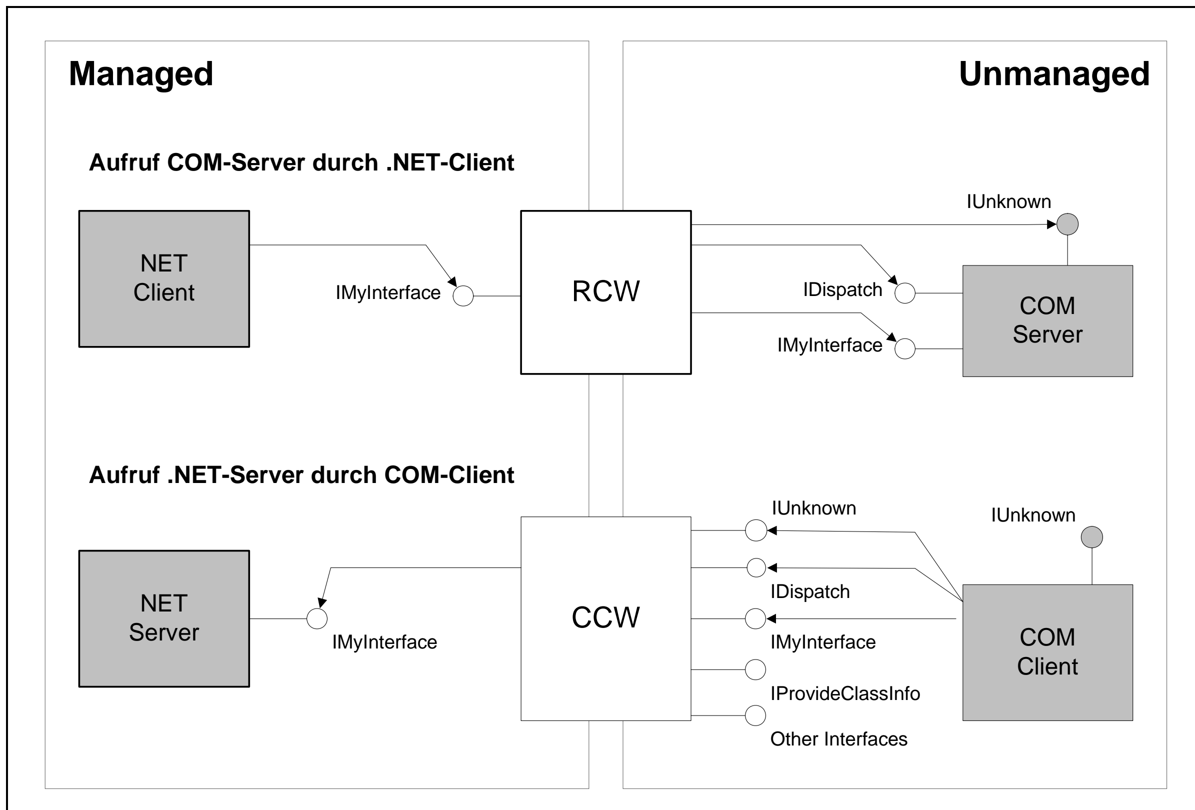


Abbildung 21: Brücken zwischen COM und .NET

Zur Herstellung der umgekehrten Interoperabilität, also zum Aufruf einer .NET-Komponente durch einen COM-Client, muss der COM Callable Wrapper die üblichen COM-Interfaces emulieren. Darüber hinaus sind weitere Transformationsaktionen erforderlich, wie etwa das Marshalling nicht isomorpher Typen, die Registrierung des CCW und die Erzeugung von Type Libraries aus .NET-Metadaten. Da der Zugriff aus COM auf .NET für die in dieser Arbeit implementierten Software keine Rolle spielt, sei hier nur auf weiterführende Literatur verwiesen, wie insbesondere [Troelsen02] für Komponenteninteroperabilität oder etwa auch [Archer03] für das Zusammenspiel von .NET und MFC-Anwendungen.