PROGRAMMER'S CHOICE

Frank Budszuhn

Visual C++

Windows-Programmierung
mit den MFC

Studentenausgabe

ADDISON-WESLEY

CD

3 COM, OLE und ActiveX

Das vorliegende Kapitel führt in die Programmierung des *Component Object Model* (COM) und einiger darauf aufbauender Technologien ein. Zunächst werden die Beweggründe für die Entwicklung von COM beleuchtet, erst dann wird auf die Implementierung eingegangen.

3.1 Einführung

Zunächst ist zu klären, worum es in diesem Kapitel überhaupt geht. Die Begriffe COM, OLE und ActiveX stehen alle für mehr oder weniger abstrakte Technologien und Konzepte. Selbst wenn man eine Vorstellung davon haben sollte, was hinter diesen Begriffen steht, so ist damit noch nicht unmittelbar klar, warum man diese Technologien und Konzepte überhaupt benötigt. Worum also geht es?

3.1.1 Motivation

Wiederverwendung

Alle in diesem Kapitel besprochenen Technologien haben mit dem zentralen Thema der Wiederverwendung zu tun. Wiederverwendung in Softwaresystemen ist ein kompliziertes Thema. Schon seit den sechziger Jahren gibt es das Schlagwort »Softwarekrise«. Software ist sehr aufwändig zu entwickeln, die üblichen Verfahren sind zeitaufwändig, teuer und fehleranfällig. Schon seit langer Zeit sucht man daher nach Techniken, um bestehende Software in anderen Umgebungen weiter- und wiederverwenden zu können. Die Informatik hat bereits eine Reihe solcher Verfahren vorgeschlagen und ist stetig dabei, neue oder verbesserte Verfahren der Softwareentwicklung zu erforschen. Hat man früher Funktionsbibliotheken erstellt, so ist es heute üblich, im Sinne der Objektorientierung Klassenbibliotheken und Frameworks zu entwerfen, um Programmcode wieder zu verwenden.

Standardbauteile

Die Zielsetzung ist es, bei der Softwareentwicklung ein ingenieurmäßiges Arbeiten zu ermöglichen, wie dies beispielsweise im Maschinenbau oder in der Architektur möglich ist. Heute ist man noch weit von diesem Ziel entfernt. Sowohl im Maschinenbau als auch in der Architektur gibt es Standardbauteile und Standardverfahren, um eine Konstruktionsaufgabe durchzuführen. Die Verfahren sind den betreffenden Ingenieuren beziehungsweise Architekten wohl bekannt; für Bauteile gibt es Kataloge, in denen diese verzeichnet sind. In der Softwaretechnik gibt es praktisch keine Standardbauteile. Viele Softwareentwickler haben ihre eigenen Bauteile; mit den gängigen Methoden der Wiederverwendung schaffen sich diese Programmierer im Laufe der Zeit ihre eigenen Bauteillager, sie verwenden also Programmcode wieder. Dies ist ganz im Sinne der Wiederverwendung. Allerdings ist der dabei betrachtete Kontext nicht zufrieden stellend: Jeder Programmierer kann nur seine eigenen Bauteile (oder die seines Teams, seiner Firma) wiederverwenden. Wechselt er zu einer anderen Firma, so muss er plötzlich mit völlig neuen Bauteilen umgehen, da er seine eigenen Bauteile aus rechtlichen Gründen nicht mitnehmen darf und die neue Firma wahrscheinlich ihre eigenen Bauteile verwendet. Übertragen auf den Maschinenbau würde dies bedeuten, dass jede Maschinenbaufirma ihren eigenen Typ Schrauben verwenden würde: Schrauben mit dreieckigen Köpfen, Schrauben mit Linksgewinde, Schrauben mit abenteuerlichen Maßen, Schrauben nur für Spezialschraubendreher. Natürlich müsste jede Firma diese Schrauben selbst herstellen, auf dem Markt sind sie schließlich nicht zu kaufen. Der Aufwand für Konstruktion und Herstellung der eigenen Schrauben wäre beträchtlich. Das kann sich keine Maschinenbaufirma leisten; stattdessen bestellt man Schrauben in Standardgrößen aus dem Katalog.

Softwarefirmen leisten sich den Luxus der Exklusivität. In vielen Firmen werden eigene Code- und Klassenbibliotheken sowie Frameworks entwickelt. Der Aufwand dafür ist erheblich. Die Entwicklungszeit praktisch einsetzbarer Klassenbibliotheken und Frameworks beträgt zumeist mehrere Mannjahre. Trotzdem sind diese »Schrauben« nur innerhalb des eigenen Entwicklungsteams einsetzbar, außerhalb der Firma möchte keiner diese fremden Bauteile haben; man verwendet lieber die eigenen (im Englischen wird dieses Phänomen als NIH, not invented here, bezeichnet). Nun kommt diese Verhaltensweise der Softwarefirmen und ihrer Entscheidungsträger nicht von ungefähr. Es gab bisher keinen echten Bauteilemarkt für Software. Es gab und gibt zwar Programmcode- und Klassenbibliotheken zu kaufen, doch sind dies nicht

wirklich Softwarebauteile. Ein Bauteil – wie eine Schraube – ist etwas fertiges. Es existiert, ist eindeutig spezifiziert, trägt eine Bezeichnung und hat oft ein klar umrissenes Einsatzgebiet. Man kann es bestellen, auspacken und einbauen. Kauft man sich jedoch eine Programmcode- oder Klassenbibliothek, so hat man kaum mehr als einen Konstruktionsplan für potenziell sehr viele Bauteile. Aber man muss diesen Konstruktionsplan zunächst verstehen, die richtigen Teile des Plans anwenden, dann das Bauteil selbst bauen (!), es auf Fehler prüfen und diese beseitigen, und erst dann kann man das so entstandene Bauteil verwenden.

Globaler Kontext

Oben ist bereits gesagt worden, dass in es in den Ingenieurdisziplinen nicht nur genormte Bauteile gibt, sondern dass dazu auch Standardverfahren bei der Konstruktion verwendet werden. Auch die Softwaretechnik hat bekannte Verfahren, um bei bestimmten Aufgaben zu Problemlösungen zu kommen. Allerdings schneidet die Softwaretechnik beim direkten Vergleich mit den Ingenieurdisziplinen wieder schlecht ab. In der Praxis sind diese Verfahren nämlich oft nicht bekannt, manchmal unausgereift und vage. Verfahren, die sich auf Softwarebauteile stützen, sind praktisch nicht vorhanden. Generell wird dem Aspekt einer globalen Entwicklergemeinde (Schrauben sind weltweit austauschbar!) bisher wenig Beachtung geschenkt. Softwareentwickler denken oft nur systemweit.

Damit man auch bei der Softwareentwicklung eine dem Ingenieurwesen gemäße Vorgehensweise einführen kann, müssen Standardbauteile und Vorgehensweisen für deren Einsatz definiert werden. Dabei sind folgende Punkte wichtig:

- Verwendbarkeit. Die Akzeptanz des Programmierers muss gewonnen werden. Das Bauteil muss »von der Stange« funktionieren, das heißt, es muss schnell und einfach zu ersehen sein, wie das Bauteil funktioniert. Es muss vollständig dokumentiert und darf nicht zu komplex sein. Je einfacher, desto besser. (Eine Schraube ist sofort verwendbar. Es dauert – normalerweise – nur kurz, bis man verstanden hat, wie der dazugehörende Schraubenzieher funktioniert.)
- Verfügbarkeit. Das Bauteil muss fertig sein. Es ist nicht akzeptabel, dass man es erst selbst zusammenbauen muss. Daraus folgt, dass ein Bauteil in binärer Form und nicht als Sourcecode vorliegen sollte. (Eine Schraube ist fertig; man muss sie nicht aus einem Stift selbst drehen.)

- Standards. Die Teile müssen genormt sein. Der Standard bestimmt das Verhalten und die Beschaffenheit des Bauteils. (Schrauben gibt es in Einheitsgrößen, normale Schrauben haben ein Rechtsgewinde.)
- Kontext. Der Kontext der Verwendung muss betrachtet werden. Bauteile müssen bestimmten Verwendungsmustern gerecht werden. Das Verwendungsmuster wird durch die Vorgehensweise bei der Erstellung des Gesamtsystems bestimmt. (Eine Schraube wird nicht einfach [ohne Dübel] in die Wand geschraubt; ein Fahrrad wird nicht zusammengenagelt.)

Dieses Kapitel beschreibt die ersten Lösungsvorschläge der Firma Microsoft für das Problem der globalen Wiederverwendung von Software.

3.1.2 Begriffe

Softwarekomponenten

Das, was im vorherigen Abschnitt als »Bauteil« beschrieben wurde, soll von nun an als *Softwarekomponente* bezeichnet werden. Eine Softwarekomponente ist ein Softwarebauteil, welches in binärer Form vorliegt und von dem es eine vollständige Beschreibung seiner Funktionsweise und seiner Verwendung gibt. Wie die beschriebene Funktion intern realisiert wird, ist dabei nicht von Interesse.

Komponentenobjektmodell

Um Softwarekomponenten zu erstellen und daraus Standardbauteile im beschriebenen Sinne machen zu können, hat Microsoft das *Komponentenobjektmodell* (COM) eingeführt. COM ist gleichsam die Werkbank zur Softwarekomponentenherstellung, die Basis, auf der genormte Bauteile entstehen können. Alle anderen in diesem Kapitel beschriebenen Technologien sind Anwendungen von COM, sie bauen auf COM auf. Diese Technologien definieren oder sind letztlich Verwendungsmuster von Softwarekomponenten, die auf COM basieren. Sie definieren Kontexte, in denen Softwarekomponenten wiederverwendet werden können. Folgende COM-basierte Technologien werden besprochen:

COM-basierte Technologien

- *Automation*. Bei der Automation (auch *Automation*) betrachtet man ganze Programme als Objekte. Das Programm stellt *Eigenschaften* (Variablen) und *Methoden* (Funktionen) zur Verfügung, mit denen es sich von anderen Programmen oder über eine Scriptsprache steuern lässt.

- *Vereinheitlichter Datenaustausch*. Der vereinheitlichte Datenaustausch ermöglicht es, Daten zwischen und innerhalb von Applikationen weitgehend unabhängig von Transportmedium und Art der transportierten Daten auszutauschen. Der vereinheitlichte Datenaustausch wird beispielsweise über die Zwischenablage und bei Drag&Drop angewandt.
- *OLE*. *Object Linking and Embedding* ist eine Technologie, bei der Dokumente verschiedener Programme zusammen in einer Datei gespeichert werden können. Diese Dokumente werden auch als *Verbunddokumente* bezeichnet. Ein Dokument dient als Container (beispielsweise ein Word-Dokument), in den andere Dokumente eingebettet werden können (beispielsweise ein Excel-Dokument). Das Besondere an OLE ist, dass das eingebettete Dokument innerhalb der Applikation, die das Container-Dokument erstellt hat, bearbeitet werden kann. (Eine Excel-Tabelle kann in Word bearbeitet werden. Die Menü- und Symbolleisten aus Excel werden dazu in Word eingeblendet.)
- *ActiveX-Steuerelemente*. ActiveX-Steuerelemente sind auf der Basis von COM erstellte Steuerelemente. Diese Steuerelemente lassen sich in verschiedenen Programmierumgebungen wie Visual Basic, Delphi oder C++ einsetzen. Auch der Internet Explorer kann in HTML eingebettete ActiveX-Steuerelemente verwenden, damit können diese innerhalb von HTML-Seiten ausgeführt werden.

Abbildung 3.1 zeigt den Zusammenhang zwischen den in diesem Kapitel vorgestellten Technologien.

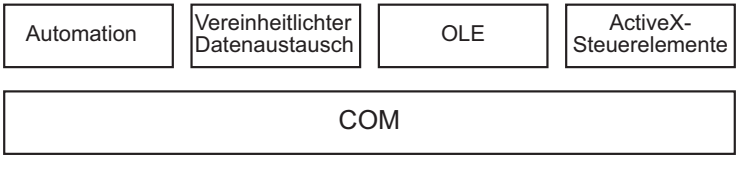

Abbildung 3.1: Auf COM basierende Technologien

COM als Basis

Wie man sehen kann, bildet COM das Fundament für Automation, vereinheitlichten Datenaustausch, OLE und ActiveX-Steuerelemente. Allerdings sind diese vier Anwendungen von COM nicht die einzigen. COM ist beispielsweise auch die Basis für die Multimediatechnologie von Microsoft, DirectX. Außerdem sind die

auf COM aufbauenden Technologien nicht völlig unabhängig voneinander. So wird beispielsweise die Automation innerhalb von ActiveX-Steuerelementen verwendet.

Leider ist das Gesamtsystem aus COM und den darauf aufbauenden Technologien nicht so harmonisch und zielgerichtet gewachsen, wie man aufgrund des bisher Gesagten vermuten könnte. Der große Plan einer auf COM basierenden Architektur zur Wiederverwendung von Softwarekomponenten ist erst langsam entstanden. So hat OLE bereits vor COM existiert. Da jedoch die erste Implementierung nicht besonders gelungen war, wurde sie im zweiten Anlauf auf der Basis von COM wiederholt. Damit wurde COM erstmals verwendet. COM hatte aber selbst bei der Einführung von OLE2 (der zweiten und jetzigen Version von OLE) noch keinen eigenen Namen, den bekam COM erst später.

Die Vorfahren der ActiveX-Steuerelemente sind ohne große Planung entstanden. Eher nebenbei wurde eine frühe Version von Visual Basic mit einer Architektur zur Verwendung von Steuerelementen von Drittherstellern ausgestattet. Diese VBX-Steuerelemente (*Visual Basic Extensions*) hatten einen solchen Markterfolg, dass Microsoft die etwas wackelige VBX-Architektur beim fälligen Umstieg von WIN16 auf WIN32 auf die Basis von COM gestellt und fortan OCX genannt hat. Durch den neuen, verbesserten Unterbau konnten OCX-Steuerelemente auch außerhalb von Visual Basic verwendet werden. Heute heißen OCX-Steuerelemente ActiveX-Steuerelemente. Die Namensänderung ist auf technische Verbesserungen zur ursprünglichen Spezifikation und auf Marketinggründe zurückzuführen.

Begriffsunklarheiten

Die inkonsistente Verwendung von Begriffen ist auf das nicht ganz gradlinige Wachstum der hier besprochenen Technologien zurückzuführen. Man sollte sich von unklaren Begriffen nicht verwirren lassen. Unter anderem treten folgende Begriffsunklarheiten auf:

- COM hatte zunächst keinen eigenen Namen. COM wurde anfangs unter dem Sammelbegriff OLE geführt.
- OLE bezeichnete zunächst die Technologie der Verbunddokumente und deren Einbettung und Verknüpfung. Mit Einführung von OLE2 bezeichnete OLE dann alle COM-basierten (damals noch nicht so genannten) Technologien. Heute bezeichnet OLE wieder die Technik der Verbunddokumente.

- Automation hieß früher OLE-Automatisierung oder Automatisierung.
- ActiveX-Steuerelemente hießen früher OLE-Steuerelemente oder OCX. Jetzt werden sie teilweise auch als COM-Steuerelemente bezeichnet.

Der Begriff ActiveX ist ohne weitere Zusätze leider nicht eindeutig. Meist sind mit ActiveX COM-basierte Technologien von Microsoft gemeint. Allerdings werden auch Internettechnologien aus dem Hause Microsoft zu ActiveX gezählt, obwohl diese mit COM teilweise nichts zu tun haben. Im Zweifelsfall sollte man genau sagen, was man meint (zum Beispiel ActiveX-Steuerelement) und die Schlagworte den Marketingleuten überlassen.

3.1.3 Übersicht

COM

Was soll in diesem Kapitel gezeigt werden? Zunächst ist es wichtig zu verstehen, was das Komponentenobjektmodell COM ist und wie es funktioniert. COM ist so wichtig, weil es die Grundlage aller weiteren in diesem Kapitel besprochenen Technologien ist. COM ist mittlerweile zu einem integralen Bestandteil von Windows geworden und ist die Basis vieler Windows-Technologien. Um zu verdeutlichen, wie COM funktioniert, wird zunächst ohne die Hilfe der MFC eine COM-Komponente erstellt. Bei der Implementierung ohne die Hilfe eines Frameworks oder die Laufzeitumgebung einer interpretierten Sprache wird das interne Wesen von COM deutlich werden. Erst danach wird auf eine Implementierung von COM-Komponenten mit den MFC hingearbeitet.

Automation

Mit der Automation wird die erste Anwendung von COM vorgestellt. Das bereits bekannte Programm StockChart wird als Automationsserver neu implementiert. Die Verwendung der Automation mit Visual Basic und mit dem Windows Scripting Host wird besprochen. Schließlich werden zwei Möglichkeiten der Implementierung von Automations-Clients innerhalb der MFC gezeigt.

Vereinheitlichter Datenaustausch

Danach wird der vereinheitlichte Datenaustausch besprochen. Drag&Drop sowie der Datenaustausch über die Zwischenablage werden behandelt.

OLE

Bei der Besprechung von OLE wird zunächst auf die der Technologie zugrunde liegenden Verbunddateien eingegangen. Danach wird das Programm StockChart zunächst als OLE-Server und anschließend als ActiveX-Dokumentserver implementiert.

ActiveX-Steuerelemente

Die Verwendung von ActiveX-Steuerelementen innerhalb der MFC wird ebenso beschrieben wie ihre Erstellung mit den MFC. Als Beispiel wird ein Drehknopf-Steuerelement verwendet.

3.2 Das Komponentenobjektmodell

Weshalb COM?

Das »C« in COM sagt es bereits: COM ist eine Technologie, die zur Erstellung von Softwarekomponenten entwickelt worden ist. Diese Komponenten werden vom Programmierer als Objekte angesprochen. Es ist zunächst nicht klar, weshalb man eine Technologie wie COM benötigt. Schließlich sind die meisten heute verwendeten Programmiersprachen mehr oder weniger objektorientiert. Techniken wie Abstraktion und Kapselung gehören zum Grundwissen eines Programmierers. Teile von Programmen lassen sich sehr leicht in DLLs auslagern. Folglich sollte die Erstellung von Softwarekomponenten eigentlich ganz einfach sein: Man nehme beispielsweise eine C++-Klasse, verpacke diese in eine DLL und vermarkte das Ganze als Softwarekomponente. Da es derartige Komponenten auf dem Markt praktisch nicht gibt, muss es wohl einige Probleme mit diesem Ansatz geben. Im Folgenden soll gezeigt werden, welche Probleme bei der Entwicklung von Softwarekomponenten mit C++ auftreten und wie diese von COM gelöst werden.

3.2.1 Anforderungen an eine Softwarekomponente

Implementierung und Schnittstelle

Eine Softwarekomponente besteht immer aus zwei Teilen. Zunächst soll eine Softwarekomponente eine Aufgabe ausführen, sie soll ein Problem lösen. Allgemein gesprochen nimmt dieser Teil der Komponente einen Satz von Eingangsdaten entgegen und produziert daraus einen Satz von Ausgangsdaten. Das ist der Implementierungsteil oder die *Implementierung* der Komponente. Um die Funktionalität der Komponente nutzen zu können, muss der Programmierer die Komponente in irgendeiner Form ansprechen können. Dazu stellt die Komponente eine oder mehrere Funktionen bereit, die der Programmierer aufrufen kann. Handelt es sich um eine objektorientierte Komponente, so kann ihre Funktionalität in Form von Objekten bereitgestellt werden, deren Methoden der Programmierer aufruft. Der Teil der Softwarekomponente, der dem Programmierer die Funktionalität der Komponente in Form von Funktions- oder Methodenaufrufen zur Verfügung stellt, wird als *Schnittstelle* der Komponente bezeichnet. Abbildung 3.2 zeigt eine Softwarekomponente mit Schnittstelle und Implementierung.

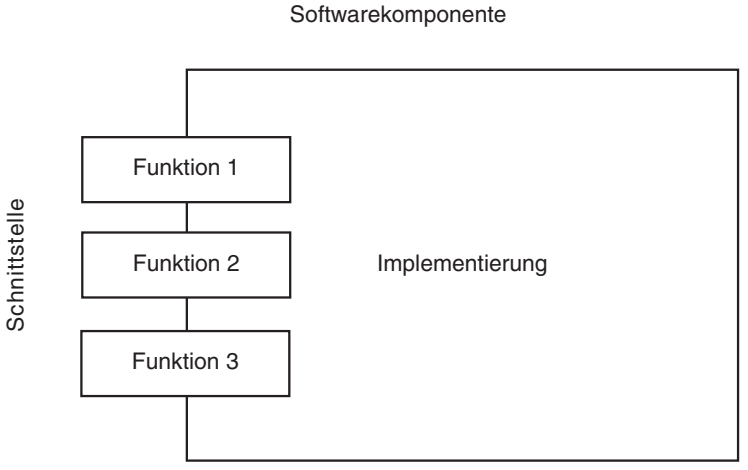

Abbildung 3.2: Softwarekomponente mit Schnittstelle und Implementierung

Kapselung

Normalerweise ist es wünschenswert, dass der Zugriff auf eine Softwarekomponente ausschließlich über die Schnittstelle erfolgen kann. Die Implementierung wird gekapselt, ein Zugriff auf sie ist nicht möglich. Auf diese Weise ist es dem Programmierer nicht möglich, Vorteile aus Implementierungsdetails der Komponente zu ziehen. Was auf den allerersten Blick wie eine Einschränkung erscheint, zahlt sich auf lange Sicht immer aus. Dadurch dass der Anwender der Komponente keinen Zugriff auf die Implementierung hat, ist es möglich, die Implementierung in neuen Versionen der Komponente zu ändern, beispielsweise um die Effizienz des verwendeten Algorithmus zu steigern. Idealerweise ist es dem Anwender der Komponente nicht nur unmöglich, auf die Implementierung der Komponente zuzugreifen, sondern er kennt sie überhaupt nicht. Die Komponente tritt als *Black Box* in Erscheinung, der Zugriff ist nur über die Schnittstelle möglich, das Innere ist unbekannt. Das Verhalten der Komponente wird allein über ihre Schnittstellenbeschreibung definiert. Die Verwendung von Black-Box-Komponenten bei der Softwareentwicklung trägt zur Robustheit von Programmen bei. Kann der Programmierer, der eine Softwarekomponente verwendet, schon durch die Kapselung nicht auf die Implementierung direkt zugreifen, so ist es ihm bei Kenntnis der Implementierung teilweise möglich, Seiteneffekte und damit nicht spezifiziertes Verhalten der Implementierung auch über die Schnittstelle auszunutzen (siehe Kasten). Da solches, nicht spezifiziertes Verhalten sich jedoch bei einer neuen

Implementierung ändern kann, trägt dessen Ausnutzung nicht zu einer Robustheit des Programms bei. Daher ist es von Vorteil, wenn der Anwender einer Komponente deren Implementierung nicht kennt und auch nicht in Erfahrung bringen kann.

Ausnutzung nicht spezifizierten Verhaltens

Nicht spezifiziertes Verhalten lässt sich üblicherweise auf zwei Weisen auch über Schnittstellen hinweg ausnutzen, wenn man die Implementierung kennt. Zum einen könnte ein Programmierer Parameter an Funktionen übergeben, von denen er weiß, dass die Werte dieser Parameter außerhalb des zulässigen Bereichs liegen. Da er die Implementierung kennt, weiß er, was für Ergebnisse zu erwarten sind. Eventuell findet er die auf diese Weise berechneten Werte nützlich und verwendet sie in seinem Programm.

Ein weiterer beliebter Fall von Ausnutzung nicht spezifizierten Verhaltens betrifft das Laufzeitverhalten eines Softwaremoduls oder einer Funktion. Wenn das Laufzeitverhalten einer bestimmten Funktion bekannt ist, kann dieses – beispielsweise als bekannte Verzögerungszeit – vom Programmierer ausgenutzt werden.

Es muss wohl kaum gesagt werden, dass beide Vorgehensweisen sehr schlechter Programmierstil sind. Es kann jedoch immer sein, dass unbeabsichtigt Implementierungsdetails ausgenutzt werden. Hat ein Programmierer die Komponente erstellt, die er später in einem Programm verwendet, dann ist es ihm vielleicht gar nicht bewusst, dass er implizit Implementierungsdetails der Komponente in seinem Programm verwendet. Den besten Schutz vor einer solchen unbeabsichtigten Verwendung von Implementierungswissen besteht darin, dass der Anwender einer Komponente die Implementierung derselben tatsächlich nicht kennt. Werden sowohl Softwarekomponenten als auch die Programme, die diese verwenden, in derselben Firma erstellt, so sollte man sie von verschiedenen Teams entwickeln lassen.

Binäre Komponenten

Eine Komponente sollte unabhängig von dem sie verwendenden Programm austauschbar sein. Das bedeutet, dass die Komponente in einem binären Format vorliegen muss, um eine Neukompilierung zu vermeiden. Die Bindung an die Komponente darf erst zur Laufzeit erfolgen, wenn man beim Austausch der Komponente eine Neukompilierung umgehen will. Binäre Softwarekomponen-

ten helfen – im Gegensatz zu Modulen auf Quellcodeebene – die monolithische Struktur von Anwendungen aufzubrechen. Die Mechanismen, die eine Softwarekomponente verwendet, um Verhalten zu implementieren und nach außen zugänglich zu machen, ähneln durchaus den Mechanismen, die dafür auf Quelltextebene verwendet werden. Die Auswirkungen auf das resultierende Softwaresystem als Ganzes sind aber deutlich andere. Das System ist nicht nur während seiner Konstruktion modular, sondern auch nach seiner Fertigstellung.

Unabhängigkeit von der Programmiersprache

Idealerweise lässt sich eine Softwarekomponente sowohl in einer beliebigen Programmiersprache erstellen, als auch von einer beliebigen Programmiersprache aus verwenden. Das binäre Format, in dem die Komponente ausgeliefert wird, sollte folglich sprachunabhängig sein. Komponenten, die sprachunabhängig sind, vergrößern vor allen Dingen die Marktchancen einer Komponente und helfen damit, einen Softwarekomponentenmarkt zu etablieren.

Globaler Kontext

Betrachtet man Softwarekomponenten in einem globalen Kontext und nimmt man einen großen Markt von unabhängigen Komponenten an, dann wird es wichtig, sowohl einzelne Komponenten als auch deren Schnittstellen eindeutig identifizieren zu können. Schließlich wird es für oft auftretende Probleme wahrscheinlich Problemlösungen mehrerer Komponentenhersteller geben. Diese Hersteller verwenden unter Umständen eine problemgerechte und damit teilweise gleiche Namensgebung. Es muss Mittel und Wege geben, solche Komponenten zu unterscheiden und gezielt ansprechen zu können.

Versionsproblem

Bei Softwarekomponenten muss man die Tatsache berücksichtigen, dass diese sich meist fortlaufend weiterentwickeln. Sofern diese Weiterentwicklung die Schnittstelle der Komponente betrifft, kann es zu Problemen kommen. Alte Programme verlangen die ursprüngliche Version der Schnittstelle, neuere Programme wollen die ganze Funktionalität der Komponente nutzen und verlangen die neue Version der Schnittstelle. Softwarekomponenten müssen dieses Versionsproblem lösen, damit sich Updates von Programm und Komponenten problemlos durchführen lassen.

Anforderungen an eine Softwarekomponente

Die folgende Liste fasst alle Anforderungen an Softwarekomponenten zusammen:

- Die Funktionalität einer Softwarekomponente darf für den Anwender der Komponente nur über eine Schnittstelle zugänglich sein. Die Implementierung der Komponente wird gekapselt

und darf nicht zugänglich sein. Das Verhalten einer Softwarekomponente wird ausschließlich über deren Schnittstelle spezifiziert.

- Bei der Verwendung von Softwarekomponenten ist es von Vorteil, sie als Black Box zu betrachten. Kennt der Anwender der Komponente deren Implementierung nicht, so ist er auch nicht versucht, implementierungsspezifische Details auszunutzen.
- Eine Softwarekomponente soll unabhängig vom sie verwendenden Programm austauschbar sein. Eine Neukompilierung des Programms darf nicht notwendig werden.
- Die Bindung des Programms an die Softwarekomponente soll erst zur Laufzeit erfolgen. Nur so lässt sich eine Komponente nachträglich, ohne Neukompilierung austauschen. Es ist vorteilhaft, wenn der Bindungsvorgang beim Programmstart erfolgt und alle Referenzen überprüft, damit es nicht zu unerwarteten Programmabstürzen aufgrund nicht auflösbarer Referenzen kommt.
- Komponenten sollen sprachunabhängig sein, sie sollen sich sowohl in verschiedenen Programmiersprachen implementieren, als auch von verschiedenen Programmiersprachen ausnutzen lassen.
- Softwarekomponenten und deren Schnittstellen sollen eindeutig identifizierbar sein.
- Verschiedene Versionen von Programm und Softwarekomponente müssen problemlos miteinander arbeiten können.

Es stellt sich die Frage, ob die genannten Anforderungen an Softwarekomponenten mit den bereits bekannten Hilfsmitteln von Windows und der Sprache C++ realisiert werden können.

3.2.2 Lassen sich die Anforderungen an Softwarekomponenten mit der Sprache C++ realisieren?

Einer der Hauptgründe für den Einsatz von Softwarekomponenten ist die Möglichkeit der Wiederverwendung in unterschiedlichen Softwareumgebungen. Eine Komponente wird einmal in möglichst allgemein gültiger Form erstellt und kann dann von verschiedenen Programmierern in unterschiedlichen Programmen eingesetzt werden.

Wiederverwendung in C++

Der Gedanke der Wiederverwendung ist in der Objektorientierung allgegenwärtig und war sicherlich einer der Beweggründe für die Entwicklung der Sprache C++. Die Sprache C++ versucht, die Wiederverwendung von Programmteilen durch verschiedene Techniken zu erreichen:

- *Polymorphie* bedeutet, dass eine Funktion auf mehr als einen Typ angewendet werden kann. Polymorphie kann in Form einfachen Überladens von Funktionsnamen mit unterschiedlichen Parameterlisten innerhalb einer Klasse oder in Form virtueller Funktionen durch mehrfache Verwendung eines Funktionsnamens innerhalb einer Vererbungshierarchie realisiert werden. Zudem tritt Polymorphie in Form von *Templates* auf.
- *Kapselung* schützt die Implementierung einer Klasse vor dem Zugriff von außen. C++ bietet drei Ebenen der Kapselung an: Durch Verwendung des Schlüsselworts *public* wird ein freier Zugriff erlaubt, das Schlüsselwort *protected* erlaubt einen Zugriff innerhalb der Vererbungshierarchie, und nur durch das Schlüsselwort *private* wird die Implementierung einer Klasse wirklich vor dem Zugriff von außen geschützt. Mit dem Schlüsselwort *friend* lassen sich zudem Löcher in die Schale der Kapselung bohren.
- Durch Vererbung, im Falle von C++ sogar durch Mehrfachvererbung, kann die Implementierung von Klassen wiederverwendet werden. Man spricht auch davon, dass die Implementierung vererbt wird, denn es wird tatsächlich Programmcode der Basisklassen in die abgeleiteten Klassen übernommen.

Wiederverwendung in Klassenbibliotheken und Frameworks

Diese von der Sprache C++ verwendeten Techniken eignen sich durchaus zur Wiederverwendung von Programmteilen. Üblicherweise wird gerade die Möglichkeit der Vererbung intensiv genutzt, was zum Entstehen von Klassenbibliotheken und Frameworks führt. Gerade ein solches Framework – die MFC sind ein sehr gutes Beispiel – bietet viele Stellen, an denen sich der Programmierer »einhängen« kann, um die vorhandene Funktionalität zu nutzen. »Einhängen« bedeutet, dass der Programmierer sich eine Basisklasse aussucht, die die von ihm gewünschte Funktionalität implementiert. Der Programmierer leitet von dieser Basisklasse eine eigene Klasse ab und verwendet so die bereits implementierte Funktionalität. Die MFC zeigen, dass auf diese Weise sehr viel Funktionalität wieder verwendet werden kann.

Vererbung hat keine Schnittstelle

Damit der Programmierer die Funktionalität des Frameworks wiederverwenden kann, muss er diesen allerdings sehr genau kennen. Da das Framework die Implementierung einer Klasse erbt, gibt es keine echte Schnittstelle. Vererbung bedeutet hier vielmehr, dass der Programmierer alles bekommt, was da ist. Zwar können Basisklassen ihre Interna durch das Schlüsselwort *private* schützen, doch zeigt die Praxis, dass es oft nicht sinnvoll ist, dies zu tun. Vielmehr kann die abgeleitete Klasse meist auf große Teile der Implementierung der Basisklasse zugreifen. Oft muss der Programmierer zumindest Teile der Implementierung der Basisklasse kennen, um deren Funktionalität wiederzuverwenden. Dies widerspricht klar den Anforderungen an eine Softwarekomponente, eine klare Schnittstelle und eine Black-Box-Implementierung zu bieten.

Enge Kopplung durch Vererbung

Doch selbst wenn es den Designern des verwendeten Frameworks gelungen sein sollte, die Implementierung aller Klassen nach außen und innerhalb der Klassenhierarchie vollständig abzuschirmen, ergeben sich weitere Probleme mit dem Ansatz der Wiederverwendung durch die Vererbung der Implementierung. Will man die vorhandene Funktionalität durch eine eigene ergänzen, so wird das oft durch das Überschreiben von virtuellen Funktionen erreicht. Mit der eigenen Version der überschriebenen Funktion ergänzt man bereits vorhandene Funktionalität. Um die vorhandene Funktionalität auszuführen, muss allerdings noch die entsprechende Funktion der Basisklasse aufgerufen werden. Doch muss diese vor oder nach der eigenen Verarbeitung aufgerufen werden? Oder vielleicht in einem bestimmten Fall auch gar nicht? Obwohl die Basisklasse vollständig abgeschirmt ist, besteht immer noch eine enge Kopplung zwischen Basisklasse und abgeleiteter Klasse. Der Programmierer muss immer noch Details der Implementierung der Basisklasse kennen, um zu entscheiden, wann er die Funktionen der Basisklasse aufrufen muss. Eine Black-Box-Implementierung ist selbst bei vollständiger Abschirmung der Basisklasse nicht gegeben.

Vererbungsgraf und Schnittstelle

Ein weiteres Problem betrifft die Schnittstelle einer Softwarekomponente. Die Schnittstelle einer C++-Klasse wird durch ihre Position innerhalb des Vererbungsgrafen mitbestimmt. Selbst wenn zwei C++-Klassen identische Funktionen bereitstellen, sind sie nicht austauschbar, sofern sich die Klassen an unterschiedlichen Positionen des Vererbungsgrafen befinden (vgl. Kasten). Die Kopplung zwischen Vererbungsgraf und Schnittstelle ist für eine Softwarekomponente nicht akzeptabel. Komponenten mit gleicher Schnittstelle müssen untereinander austauschbar sein.

Die Schnittstelle einer C++-Klasse wird durch ihre Position im Vererbungsgrafen mitbestimmt. Folgendes Beispiel soll dies verdeutlichen:

```
// Klasse A
class A
{
public:
   A(int nNum):nNumber(nNum),nFiller(17) {};

   int Square() { return nNumber*nNumber; };

protected:
   int nFiller; // Füllwert (Speicherlayout)
   int nNumber;
};

// Klasse B, abgeleitet von A
class B : public A
{
public:
   B(int nNum):A(nNum) {};

   void DoNothing() {};
};

// Klasse C, nicht Teil der Klassenhierarchie
class C
{
public:
   C(int nNum):nNumber(nNum) {};

   int  Square() { return nNumber*nNumber; }
   void DoNothing() {};

private:
   int nNumber;
};


void main (){
   B b(3);
   A *pA;
   B *pB;
   C *pC;

   pA = &b; // ok
   pB = &b; // ok
   pC = &b; //  *** Compilerfehler: ***
```

```
// "'class B *' kann nicht in 'class C *' konvertiert werden.
//  Die Typen, auf die verwiesen wird, sind nicht verwandt;
//  die Konvertierung erfordert einen reinterpret_cast-Opera-
tor
//  oder eine Typumwandlung im C- oder Funktionsformat."

   pA->Square ();
   pB->Square ();
   pC->Square ();
}
```

Die Klassen *B* und *C* haben beide die gleiche Schnittstelle, bestehend aus den Funktionen *Square* und *DoNothing*. Die Klasse B ist von der Klasse *A* abgeleitet, die die Funktion *DoNothing* nicht besitzt. Das heißt, dass sie eine andere, einfachere Schnittstelle besitzt. Die Klassen *B* und *C* haben einen unterschiedlichen internen Aufbau. Die Klasse *B* besitzt zusätzlich zu *C* eine geschützte Variable namens *nFiller.*

Eine Variable der Klasse *B* lässt sich problemlos anderen Variablen der Klassen *A* und *B* zuweisen. *B* und *C* sind jedoch nicht zuweisungskompatibel. Der Compiler liefert eine Fehlermeldung über einen Typkonflikt. Dies liegt daran, dass C++-Typen nicht über deren Schnittstelle, sondern über deren Position im Vererbungsgrafen bestimmt werden. Nimmt man die vom Compiler vorgeschlagene Typumwandlung vor, wird man überrascht sein: Der Aufruf von `pC->Square()` liefert keineswegs 9, sondern 289! Durch das andere Speicherlayout der Klasse *B* hat *C::Square* die Variable *nFiller* quadriert!

Wer jetzt denkt, dass man *Square* nur als virtuelle Funktion definieren müsste, damit das Beispiel (mit expliziter Typumwandlung) funktioniert, hat Recht. Allerdings ist zu beachten, dass alle Funktionen einer Schnittstelle virtuell und in der gleichen Reihenfolge(!) zu deklarieren sind. Die explizite Typumwandlung sorgt nur dafür, dass sich die Zeiger auf die *vtable* (Tabelle der virtuellen Funktionszeiger einer Klasse) der beiden Klassen zuweisen lassen. Stimmen die vtables nicht überein, kommt es zu Abstürzen oder unvorhergesehenen Resultaten. Diese Vorgehensweise ist möglich, aber nicht typsicher!

C++ und DLLs

C++-Programme werden normalerweise mit Hilfe des Compilers in eine einzige ausführbare Datei übersetzt. Das bedeutet, dass Komponenten, die sich auf Quellcodeebene eindeutig voneinander unterscheiden lassen, mit dem Übersetzungsvorgang zu einem

großen monolithischen Block verschmolzen werden. C++ kennt keine Komponenten in binärer Form. Unter Windows ist es allerdings möglich, C++-Klassen in DLLs einzubetten. Damit lässt sich die Forderung nach dem Austausch von Komponenten unabhängig vom Programm, das sie verwendet, realisieren. DLLs ermöglichen darüber hinaus die Bindung einer Komponente an ein Programm zur Laufzeit. Allerdings wird durch die Verwendung von DLLs als Container für C++-Komponenten das Problem des *Name Manglings* nicht gelöst.

Name Mangling

C++-Kompiler verwenden das Name Mangling, um Funktions- und Variablennamen eindeutig zu machen. Dies wurde bereits in Abschnitt 2.10, »MFC und DLLs«, gezeigt. Wie schon erwähnt, ist dieses Verfahren nicht standardisiert. Jeder Compilerhersteller kann hier sein eigenes Verfahren implementieren. Daher lassen sich keine allgemein verwendbaren Komponenten mit der Sprache C++ erstellen. Dies ist schon bei der Besprechung von MFC-Erweiterungs-DLLs deutlich geworden. C++-Klassen, die in DLLs eingebettet werden, können nur mit dem Compiler verwendet werden, der diese DLLs erstellt hat. Die für Softwarekomponenten geforderte Sprachunabhängigkeit wird nicht erreicht, es wird nicht einmal Compiler-Unabhängigkeit erreicht.

C++ und Schnittstellen

C++-Klassen bieten keine Mechanismen, um eindeutig identifizierbare Komponenten zu erzeugen. Das Gleiche gilt für die Schnittstelle einer Komponente. In C++ ist die Schnittstelle Teil der Implementierung und wird nicht explizit abgetrennt. Zwar ist es gängige Praxis, die Deklaration einer Klasse in einer Header-Datei und die Implementierung in einer Implementierungsdatei vorzunehmen, doch gibt es keinen Mechanismus in der Sprache, der diese Vorgehensweise erzwingt. Oft werden als *inline* deklarierte Funktionen gleich in der Header-Datei definiert. Dies ist eine klare Vermischung von Schnittstelle und Implementierung. Das Konzept einer expliziten Schnittstelle ist der Sprache C++ nicht bekannt. Die Schnittstelle einer C++-Klasse ist einfach das, was die Klasse als *public* deklariert: eine schlechte Voraussetzung, wenn man Schnittstellen eindeutig identifizieren möchte.

Versionsprobleme

Schließlich sollen verschiedene Versionen von Programm und Komponente problemlos miteinander arbeiten können. Auch hier kann es bei der Verwendung der Sprache C++ zu Problemen kommen, wie Listing 3.1 zeigt.

```
// Version 1.0
class CMyClass
{
public:
   CMyClass ();
   ~CMyClass ();
   long DoWhatIWant (long nInput);

private:
   long m_data;
};

// Version 2.0
class CMyClass
{
public:
   CMyClass ();
   ~CMyClass ();
   long DoWhatIWant (long nInput);

private:
   long m_data;
   long m_cache;
};
```

Listing 3.1: Unvollständige Kapselung von C++-Klassen

Version 1.0 der Komponente besitzt eine Schnittstelle, die aus Konstruktor, Destruktor und einer Funktion besteht, der ein Eingabewert übergeben wird und die daraus einen Ausgabewert erzeugt. Die Implementierung benötigt die Variable *m_data* für interne Berechnungen; *m_data* ist nicht Teil der Schnittstelle. In der Version 2.0 hat sich die Schnittstelle nicht verändert. Die Effizienz der Implementierung kann jedoch durch die Einführung der Variablen *m_cache* deutlich gesteigert werden. Beide Versionen der Komponente sollten austauschbar sein, da die Schnittstelle bei beiden gleich ist. Die Vermischung von Schnittstelle und Implementierung zeigt jedoch auch hier ihre Schwächen: Die beiden Komponenten sind nicht austauschbar, da die Objekte beider Klassen verschiedene Größen haben. Ein Objekt der Version 1.0 hat eine Größe von 4 Byte (`sizeof(long) == 4`), während ein Objekt der Version 2.0 aufgrund der dazugekommenen Variable die doppelte Größe hat. Obwohl die privaten Variablen die Schnittstelle der Komponente nicht verändern, wirken sie sich doch auf die äußeren Eigenschaften der Komponente aus. Die Komponente ist nicht in der Lage, die Größe ihrer internen Variablen zu verstecken. Die Größe der internen Variablen wirkt sich direkt auf die Speichergröße der

Komponente selbst aus. Damit ist die Kapselung der Implementierung unvollständig. Ein Programm, das eine Komponente der Version 1.0 erwartet, wird genau 4 Byte für diese bereitstellen. Wird in Wirklichkeit aber bereits die Komponente der Version 2.0 verwendet, dann kommt es mit hoher Wahrscheinlichkeit zum Programmabsturz, da die Implementierung der Komponente davon ausgeht, dass ihr 8 Byte Speicherplatz zur Verfügung stehen. Der Speicherplatz für *m_cache* gehört der Komponente aber gar nicht.

Aufgrund der Vermischung von Schnittstelle und Implementierung kann die Verwendung verschiedener Versionen von C++-Komponenten sehr gefährlich werden. Man muss sich darüber im Klaren sein, dass eine Veränderung der Implementierung einer Klasse sich auf die Größe der Objekte der Klasse auswirken kann. Für universell verwendbare Softwarekomponenten, die in der Lage sein müssen, in verschiedenen Versionen zusammenarbeiten zu können, ist die direkte Verwendung von C++-Klassen als Softwarekomponenten daher ungeeignet.

C++-Klassen sind keine Komponenten

Zusammenfassend lässt sich sagen, dass die Verwendung von C++-Klassen als Softwarekomponenten aus einer ganzen Reihe von Gründen nicht praktikabel ist. Diese Gründe sind insbesondere:

- Die Kapselung der Implementierung von C++-Klassen ist unvollständig. Dies gilt besonders wenn C++-Klassen Bestandteil einer Klassenhierarchie sind. Arbeitet man mit der Vererbung von Implementierungen, läuft man Gefahr, Details dieser Implementierung offen zu legen.
- C++-Klassen sind meist keine Black-Box-Implementierungen. Dies gilt wiederum besonderes, wenn Vererbung im Spiel ist.
- C++-Compiler produzieren normalerweise monolithische, ausführbare Dateien. Die Verwendung von DLLs löst das Problem nicht, da das Name Mangling des Compilers eine Sprach- und Compiler-Unabhängigkeit verhindert.
- Die Sprache C++ bietet keine saubere Trennung zwischen Schnittstelle und Implementierung. C++ kennt keine expliziten Schnittstellen, wie sie beispielsweise die Sprache Java kennt.
- Die Schnittstelle einer C++-Klasse wird durch ihre Position im Vererbungsgrafen mitbestimmt. Das ist für eine Softwarekomponente inakzeptabel, da sie so nicht frei austauschbar ist.

Es sei angemerkt, dass auch andere Programmiersprachen viele der Anforderungen an universell einsetzbare Softwarekomponenten nicht erfüllen. Daher bietet es sich an, ein Komponentenmodell einzuführen, das von der Implementierungssprache unabhängig ist. Genau das ist COM.

3.2.3 Eigenschaften des Komponentenobjektmodells COM

COM-Eigenschaften

COM ist ein Modell für die Erstellung programmiersprachenunabhängiger universell einsetzbarer Softwarekomponenten. COM wurde entwickelt, weil sich mit Hilfe normaler Programmiersprachen universell verwendbare Softwarekomponenten nicht oder nur sehr schwer erstellen lassen. Stattdessen wurde mit COM ein Objektmodell entwickelt, das sich an alle gängigen Programmiersprachen anpassen lässt. COM besitzt insbesondere folgende Eigenschaften:

- COM ist programmiersprachenunabhängig. COM-Anpassungen existieren für viele gängige Programmiersprachen wie C, C++, Java und Visual Basic.
- COM verzichtet auf die Vererbung von Verhalten. Das heißt, dass Vererbung zwar innerhalb einer Komponente durch die Implementierungssprache verwendet werden kann, außerhalb der Komponente kann diese ihre Implementierung jedoch nicht vererben. Die COM-Designer haben die Vererbung als ein Mittel zum Aufbrechen der Kapselung von Komponenten erkannt. Daher ist die Vererbung von Implementierungen in COM nicht erlaubt.
- Im Gegensatz zur Sprache C++ sind Schnittstellen in COM von der Implementierung völlig getrennt. In COM besitzen Schnittstellen eigene Bezeichner. Eine Komponente kann mehrere Schnittstellen haben. Ein Zugriff auf die Interna einer Komponente kann ausschließlich über deren Schnittstellen erfolgen. COM-Schnittstellen verfügen ausschließlich über Funktionen; Variablen sind nicht Bestandteil einer Schnittstelle.
- Schnittstellen können – oder besser gesagt müssen – in COM vererbt werden. Durch die Vererbung von Schnittstellen wird die Spezifikation von Verhalten vererbt, nicht die Implementierung. Eine Schnittstelle ist schließlich nichts anderes als eine Spezifikation von Verhalten.
- Komponenten (*COM-Klassen*) und Schnittstellen sind bei Benutzung von COM weltweit eindeutig identifizierbar. Zur

Identifizierung werden 128 Bit lange Integer-Zahlen, die *Globally Unique Identifier*, kurz GUID, verwendet.

- COM besitzt eine eingebaute Referenzzählung. Nicht mehr referenzierte Komponenten löschen sich selbstständig.
- COM implementiert ein Client-Server-Modell. COM-Komponenten werden als *COM-Server* bezeichnet. Die Programme, die sich der COM-Komponenten bedienen, heißen *COM-Clients*.

3.2.4 Schnittstellen

Nähert man sich COM zum ersten Mal, dann sollte man sich zunächst mit dem Konzept der Schnittstelle auseinander setzen. Auf COM-Komponenten kann in jedem Fall nur über Schnittstellen zugegriffen werden. Es ist wichtig zu verstehen, dass eine Schnittstelle selbst keine Klasse oder Komponente ist. Eine Schnittstelle implementiert keine Funktionalität. Sie ist nur der Zugang zur Implementierung.

Aufbau von COM-Schnittstellen

Jede Schnittstelle besteht aus einer Reihe von Funktionen. Variablen können nicht Teil einer Schnittstelle sein und es kann keine Instanzen von Schnittstellen geben.

Vererbung

Jede Schnittstelle muss von einer anderen Schnittstelle abgeleitet werden, von der sie Funktionen – im COM-Sprachgebrauch *Methoden* – erbt. Die Basisschnittstelle für alle anderen Schnittstellen heißt *IUnknown*, von ihr müssen alle weiteren Schnittstellen direkt oder indirekt abgeleitet werden. Per Konvention beginnen alle COM-Schnittstellen mit dem großen Buchstaben »I«, der für Interface steht.

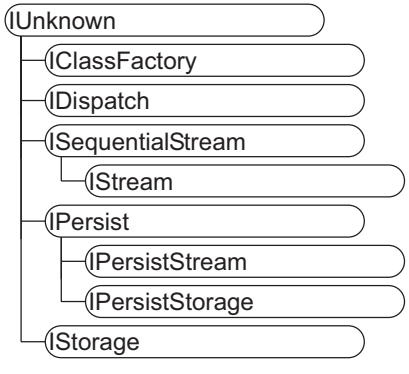

Abbildung 3.3: Hierarchie einiger COM-Schnittstellen

Unveränderlichkeit

COM-Schnittstellen sind per Definition unveränderlich. Eine Schnittstelle darf daher nicht erweitert werden. Sollten Änderungen an einer Schnittstelle notwendig werden, so ist eine neue Schnittstelle zu definieren. Dadurch, dass sich Schnittstellen nicht ändern dürfen, werden Versionsprobleme vermieden. Verwendet ein Programm eine Schnittstelle, so kann es sich darauf verlassen, dass die Schnittstelle immer ihrer ursprünglichen Definition entspricht.

In C++ werden COM-Schnittstellen auf C++-Klassen abgebildet, die ausschließlich rein virtuelle Funktionen haben. Listing 3.2 zeigt die Definition einer COM-Schnittstelle mit dem Namen *ISquare* in C++.

```
#include "unknwn.h"
interface ISquare : public IUnknown
{
   public:
      virtual HRESULT STDMETHODCALLTYPE
         GetSquare (long nValue, long *pnResult) = 0;
};
```

Listing 3.2: Definition einer COM-Schnittstelle

Für COM-Schnittstellen ist Mehrfachvererbung nicht zugelassen. Eine COM-Klasse kann jedoch mehrere Schnittstellen implementieren, so dass sich aus dem Verzicht auf Mehrfachvererbung bei Schnittstellen keine Nachteile ergeben.

3.2.5 COM-Klassen

Referenzzählung

COM-Klassen implementieren die Funktionalität von Komponenten. Im Gegensatz zu C++-Klassen können COM-Klassen keine Konstruktoren und Destruktoren über ihre Schnittstellen nach außen zugänglich machen. Instanzen von COM-Klassen werden über einen Referenzzählmechanismus verwaltet. Gibt es keine Referenzen mehr auf eine Instanz einer COM-Klasse, dann löscht sich diese Instanz selbst. Eine COM-Komponente kann mehrere Klassen implementieren.

3.2.6 GUIDs

IIDs und CLSIDs

GUIDs sind 128 Bit lange Integer-Zahlen. Wie bereits erwähnt, werden COM-Schnittstellen durch die Zuweisung von GUIDs weltweit eindeutig identifizierbar gemacht (siehe Kasten). Eine zur Identifizierung einer Schnittstelle verwendete GUID wird als *Interface Identifier*, IID, bezeichnet. Neben Schnittstellen werden auch COM-Klassen durch GUIDs identifiziert. Im Fall einer zur Identifizierung einer

COM-Klasse verwendeten GUID spricht man von einem *Class Identifier*, CLSID. Listing 3.3 zeigt die IID von *IUnknown*.

```
{00000000-0000-0000-C000-000000000046}
```

Listing 3.3: IID von IUnknown

GUIDs werden durch die Funktion *CoCreateGuid* der COM-Laufzeitbibliothek erzeugt. Diese Funktion ruft ihrerseits die API-Funktion *UuidCreate* auf, die die eigentliche Arbeit ausführt. Das im Text beschriebene Programm GUIDGEN verwendet intern diese Funktionen, um GUIDs zu erzeugen. Daneben gibt es das Programm UUIDGEN, das in einer DOS-Box läuft und ebenfalls GUIDs erzeugt.

Der Algorithmus von *UuidCreate* erzeugt laut Microsoft »mit einem sehr hohen Grad der Sicherheit« keine doppelten GUIDs. Diese Aussage gilt für alle weltweit erzeugten GUIDs. Wenn man jedoch weiß, welche Grundlage der Algorithmus für seine Berechnungen verwendet, kann man diesen sehr hohen Grad an Sicherheit noch etwas erhöhen.

Zum einen stützt sich *UuidCreate* auf die aktuelle Zeit, um einen eindeutigen Wert zu erhalten. Da jedoch viele Uhren innerhalb von Computern falsch gehen und *UuidCreate* weltweit sicherlich zur gleichen Zeit mehrfach aufgerufen wird, ist dies keinesfalls ausreichend. Natürlich hilft der sehr große Wertebereich von 128 Bit. Praktisch sollten in diesem Wertebereich keine doppelten Einträge auftreten, aber auch das ist keinesfalls sicher. Falls der Rechner, auf dem die Funktion *UuidCreate* ausgeführt wird, eine Netzwerkkarte (Ethernet oder Token Ring) besitzt, so wird auch deren Netzwerkadresse als Grundlage zur Berechnung der GUID verwendet. Die Netzwerkadressen solcher Karten werden weltweit eindeutig vergeben, das heißt, jede Netzwerkkarte hat ihre eigene Adresse (diese Hardware-Adresse sollte nicht mit der IP-Adresse eines Computers verwechselt werden). Entwickelt man Software, die später an eine größere Zahl von Kunden ausgeliefert werden soll, so empfiehlt es sich, die GUIDs auf einem Computer mit Netzwerkkarte erzeugen zu lassen, da sie dann mit höherer Wahrscheinlichkeit eindeutig sind (die letzte Fehlerquelle liegt jetzt beim Netzwerkkartenhersteller). Das Programm UUIDGEN gibt eine Warnung aus, wenn es keine Netzwerkkarte finden konnte, das Programm GUIDGEN leider nicht.

Das Programm GUIDGEN

Um eindeutige GUIDs zu erzeugen, gibt es im Windows-API die Funktion *CoCreateGuid*. Man kann dazu jedoch auch das Hilfsprogramm GUIDGEN aufrufen, das sich im Verzeichnis MICROSOFT VISUAL STUDIO .NET\VC7\BIN befindet. GUIDGEN kann die erzeugte GUID bereits in alle gängigen Darstellungsformate konvertieren (Abbildung 3.4).

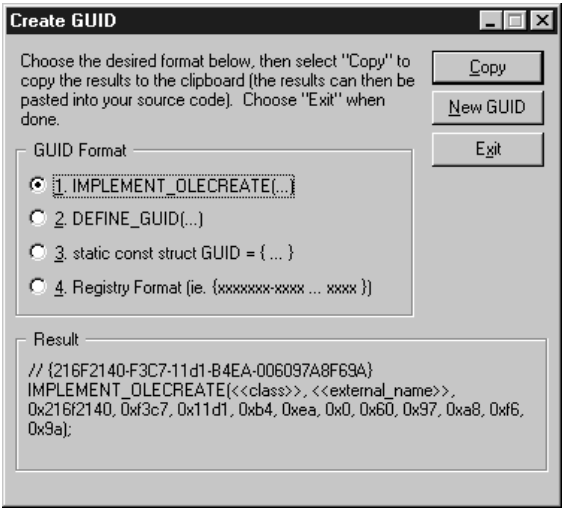

Abbildung 3.4: GUID-Erzeugung mit dem Programm GUIDGEN

Da gängige C++-Compiler keinen Datentyp für 128 Bit lange Integer-Zahlen bereitstellen, wird im COM-API dafür der Datentyp *GUID* definiert. *IID* und *CLSID* sind ebenfalls als Typen definiert.

3.2.7 Die COM-Notation

Für COM wurde eine eigene Notation entwickelt, mit der sich Komponenten grafisch darstellen lassen. Abbildung 3.5 zeigt eine Komponente, die die Schnittstellen *IUnknown*, *ISquare* und *ISquareRoot* implementiert.

Jede Schnittstelle wird in dieser Notation durch einen eigenen »Anschluss« bezeichnet. Der Anschluss für *IUnknown* wird üblicherweise nach oben hin ausgeführt, da *IUnknown* als Basisschnittstelle von allen COM-Komponenten implementiert werden muss. Die Komponente selbst wird als Rechteck mit abgerundeten Ecken dargestellt.

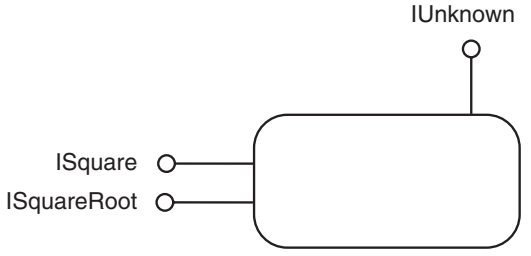

Abbildung 3.5: Komponente in COM-Notation

3.2.8 Die Schnittstelle *IUnknown*

IUnknown ist die Basisschnittstelle, von der alle anderen COM-Schnittstellen abgeleitet werden. Listing 3.4 zeigt die Definition dieser Schnittstelle.

```
interface IUnknown
{
public:
   virtual HRESULT STDMETHODCALLTYPE QueryInterface(
      REFIID riid,
      void **ppvObject) = 0;
   virtual ULONG STDMETHODCALLTYPE AddRef(void) = 0;
   virtual ULONG STDMETHODCALLTYPE Release(void) = 0;
};
```

Listing 3.4: Definition von IUnknown

AddRef und Release

Die Methoden *AddRef* und *Release* dienen zur Instanzzählung von COM-Objekten. Der Aufruf von *AddRef* bewirkt, dass im COM-Objekt der Referenzzähler um 1 erhöht wird. *Release* dekrementiert den Referenzzähler. Ist der Referenzzähler bei null angekommen, löscht sich das COM-Objekt selbsttätig.

QueryInterface

Über die Methode *QueryInterface* können Zeiger auf andere von der Klasse implementierte Schnittstellen angefordert werden. Damit lassen sich zwar Schnittstellen eines COM-Objekts durch Ausprobieren erfragen, normalerweise sollte der Programmierer aber bereits wissen, welche Schnittstelle er anfordern möchte. Zur Identifizierung der gewünschten Schnittstelle wird *QueryInterface* eine Referenz auf eine IID übergeben, die die angeforderte Schnittstelle eindeutig beschreibt. Unterstützt die Komponente die angeforderte Schnittstelle, so gibt *QueryInterface* einen Zeiger auf die Schnittstelle zurück und erhöht selbsttätig den Referenzzähler. Man kann dann sofort Methoden der angeforderten Schnittstelle

aufrufen. Benötigt man die Dienste einer Schnittstelle nicht mehr, dann gibt man sie durch den Aufruf von *Release* wieder frei.

Bei COM wird oft mit Zeigern auf Schnittstellenzeiger gearbeitet. Wie bei der Methode QueryInterface *werden dabei Zeiger auf Zeiger auf* void *verwendet. Die doppelte Verwendung von Zeigern ist notwendig, damit die Methode* QueryInterface *einen Zeiger auf eine COM-Schnittstelle an seinen Aufrufer zurückliefern kann. Zeiger auf* void *werden verwendet, damit der Schnittstellenzeiger selbst beliebige Typen annehmen kann. Übergibt man einen Zeiger an eine C++-Funktion, die einen Zeiger auf einen Zeiger auf* void *erwartet, dann kann der Compiler keine automatische Typkonvertierung vornehmen. Dies ist eine Eigenheit der Sprache C++. Man wird daher bei der COM-Programmierung oft auf explizite Typumwandlungen auf* void** *stoßen!*

3.2.9 Klassenfabriken

IClassFactory

Eine COM-Komponente muss Instanzen ihrer COM-Klassen selbst anlegen können. Obwohl theoretisch jede COM-Klasse ihren eigenen Mechanismus zur Erzeugung von Instanzen implementieren könnte, ist es ganz im Sinne des COM-Gedankens, hierfür eine COM-Schnittstelle zu verwenden, die von allen Komponenten implementiert wird. Diese Schnittstelle heißt *IClassFactory* (es gibt außerdem eine neuere Version dieser Schnittstelle mit dem Namen *IClassFactory2*, die lizenzierte Komponenten unterstützt). Die Klasse, die *IClassFactory* implementiert, wird als *Klassenfabrik* bezeichnet. Der Name Klassenfabrik ist irreführend und von Microsoft schlecht gewählt, da die Klassenfabrik Objekte fabriziert und keine Klassen. Der Ausdruck Objektfabrik wäre treffender. Die Implementierung der Klassenfabrik muss statisch erfolgen, schließlich ist die Klassenfabrik selbst ein COM-Objekt, und es ist noch niemand da, der eine Instanz der Klassenfabrik erstellen könnte. Da die Klassenfabrik statisch ist, kann es immer nur eine Instanz von ihr geben. Im Sinne der Entwurfsmuster ist die Klassenfabrik ein *Singleton* (siehe das Buch von Gamma et al. im Literaturverzeichnis). Abbildung 3.6 zeigt eine Komponente mit Klassenfabrik.

Die Schnittstelle *IClassFactory* definiert zwei Methoden, *CreateInstance* und *LockServer*. Listing 3.5 zeigt die Definition von *IClassFactory*.

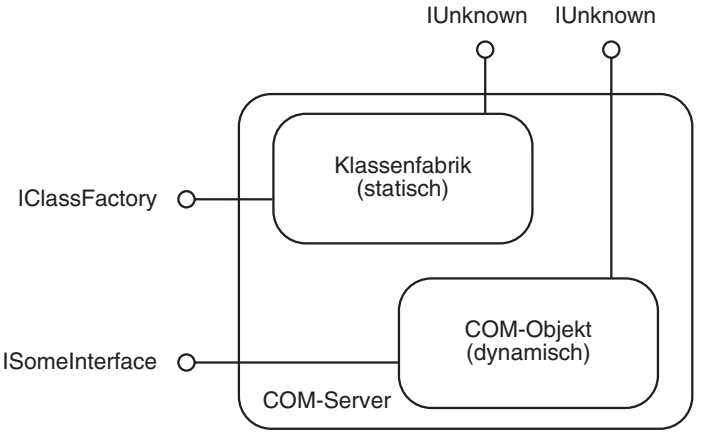

Abbildung 3.6: Klassenfabrik und Objektinstanz

```
interface IClassFactory : public IUnknown
{
public:
   virtual HRESULT STDMETHODCALLTYPE CreateInstance(
      IUnknown *pUnkOuter,
      REFIID riid,
      void **ppvObject) = 0;
   virtual HRESULT STDMETHODCALLTYPE LockServer(
      BOOL fLock) = 0;
};
```

Listing 3.5: Definition von IClassFactory

Mit der Methode *CreateInstance* kann eine Instanz eines COM-Objekts erstellt werden. Über den Parameter *riid* wird dabei zugleich eine Schnittstelle angegeben, mit der auf diese Instanz zugegriffen werden kann. Die Methode *LockServer* kann dazu verwendet werden, den COM-Server zwangsweise im Speicher zu halten. Normalerweise muss diese Methode jedoch nicht aufgerufen werden.

3.2.10 Die COM-Laufzeitbibliothek

Um die Arbeit mit COM-Servern unter Windows zu unterstützen, gibt es als Teil des Windows-API die *COM-Laufzeitbibliothek*. Mit Hilfe der COM-Laufzeitbibliothek können beispielsweise COM-Server im System gefunden und gestartet werden. Funktionen der Bibliothek beginnen mit dem Präfix »Co«. Die bereits erwähnte Funktion *CoCreateGuid* ist eine Funktion der COM-Laufzeitbibliothek. Damit die COM-Laufzeitbibliothek verwendet werden kann,

muss sie zunächst initialisiert werden. Die COM-Bibliothek selbst stellt dafür die Funktionen *CoInitialize* und *CoUninitialize* zur Verfügung. In MFC-Programmen ruft man jedoch stattdessen zu Beginn des Programms, also normalerweise in der Funktion *InitInstance* des Applikationsobjekts, die Funktion *AfxOleInit* auf. Diese initialisiert nicht nur die COM-Laufzeitbibliothek, sondern auch die OLE-DLLs.

3.2.11 Verwendung eines COM-Servers

Um die bisher vorgetragene COM-Theorie mit etwas Leben zu erfüllen, soll zunächst die Verwendung eines COM-Servers gezeigt werden. Dieser Server implementiert sowohl eine Klassenfabrik als auch eine über *IUnknown* hinausgehende Schnittstelle. Diese Schnittstelle heißt *ISquare* und implementiert genau eine Methode, die das Quadrat einer Integer-Zahl berechnet. Die Definition dieser Schnittstelle wurde bereits in Listing 3.2 aus Abschnitt 3.2.4, »Schnittstellen«, gezeigt. Abbildung 3.7 zeigt den Server in COM-Notation.

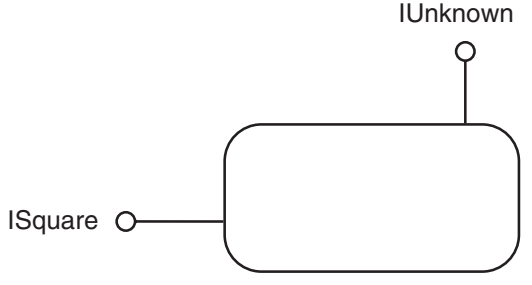

Abbildung 3.7: ISquare-Server in COM-Notation

Damit die Verwendung eines solchen COM-Servers gezeigt werden kann, wird zunächst eine Implementierung eines solchen Servers benötigt. Ganz im Sinne von COM soll der Server als Black Box betrachtet werden, die Implementierung ist nicht interessant. Das Visual C++-Projekt eines solchen Servers befindet sich auf der Begleit-CD im Verzeichnis KAPITEL3\ATLSERVER. Damit der Server verwendet werden kann, muss das Projekt auf die Festplatte kopiert und übersetzt werden. Der Server wird dabei automatisch registriert. (Alle COM-Server müssen vor der Verwendung registriert werden. Wie man dies bewerkstelligt, wird in Abschnitt 3.2.14, »Registrierung von COM-Servern«, beschrieben.)

Beispielprogramm SquareClient

Im Verzeichnis KAPITEL3\SQUARECLIENT der Begleit-CD befindet sich das Programm SquareClient. Dieses Programm implementiert einen COM-Client, der auf vier verschiedene Implementierungen eines COM-Servers zugreifen kann. Die vier Server implementieren alle die *ISquare*-Schnittstelle, sind aber unterschiedlich realisiert. Der erste Server ist die bereits erwähnte Black Box-Implementierung. Die drei anderen Server werden im Folgenden besprochen.

Das Programm SquareClient ist als dialogfeldbasierende Anwendung mit dem Anwendungs-Assistenten erstellt worden. Mit vier Optionsfeldern wird der gewünschte COM-Server ausgewählt. In einem Eingabefeld kann eine Integer-Zahl eingegeben werden. Besteht eine Verbindung mit einem COM-Server, so wird nach einem Klick auf OK das Quadrat der Zahl durch den COM-Server berechnet, an den Client zurückgegeben und dann von diesem ausgegeben. Kann die Verbindung zum gewählten COM-Server nicht aufgebaut werden, so wird eine Fehlermeldung angezeigt. Abbildung 3.8 zeigt das Programm SquareClient.

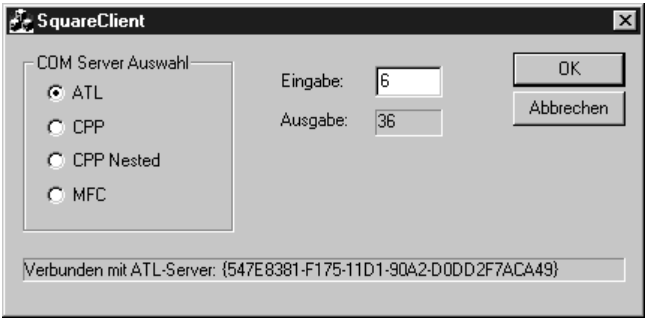

Abbildung 3.8: Das Programm SquareClient

Initialisierung der COM-Bibliothek

Bei der Erstellung des Programms SquareClient mit dem Anwendungs-Assistenten sind die Optionen für ActiveX und für Automation nicht ausgewählt worden. Damit man trotzdem Funktionen der Windows-COM-Bibliothek aufrufen kann, ist es notwendig, die Header-Datei AFXOLE.H von Hand einzubinden und in *CSquareClientApp::InitInstance* die Funktion *AfxOleInit* aufzurufen. Damit wird die COM-Bibliothek initialisiert.

Listing 3.6 zeigt die Implementierungsdatei der Klasse *CSquareClientDlg*, in der die Verwendung der COM-Server implementiert ist.

```
// SquareClientDlg.cpp : Implementierungsdatei
//

#include "stdafx.h"
#include "SquareClient.h"
#include "SquareClientDlg.h"

// Von Hand eingefügt:
#include "afxole.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////
/
// CAboutDlg-Dialogfeld für Anwendungsbefehl "Info"

class CAboutDlg : public CDialog
{
public:
   CAboutDlg();

// Dialogfelddaten
   enum { IDD = IDD_ABOUTBOX };

   protected:
   virtual void DoDataExchange(CDataExchange* pDX);

// Implementierung
protected:
   DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
// CSquareClientDlg Dialogfeld
```

```
CSquareClientDlg::CSquareClientDlg(CWnd* pParent /*=NULL*/)
   : CDialog(CSquareClientDlg::IDD, pParent),
     m_pISquare (NULL)
{
   m_strOut = _T("");
   m_nIn = 0;
   m_strMessage = _T("");
   // Beachten Sie, dass LoadIcon unter Win32 keinen
   // nachfolgenden DestroyIcon-Aufruf benötigt
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

CSquareClientDlg::~CSquareClientDlg ()
{
   // Wenn noch eine ISquare-Schnittstelle gehalten wird,
   // diese freigeben
   if (m_pISquare)
   {
      m_pISquare->Release ();
   }
}

void CSquareClientDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Text(pDX, IDC_STATIC_OUT, m_strOut);
   DDX_Text(pDX, IDC_EDIT1, m_nIn);
   DDX_Text(pDX, IDC_STATIC_MSG, m_strMessage);
}

BEGIN_MESSAGE_MAP(CSquareClientDlg, CDialog)
   ON_WM_SYSCOMMAND()
   ON_WM_PAINT()
   ON_WM_QUERYDRAGICON()
   ON_BN_CLICKED(IDC_RADIO1, OnRadioATL)
   ON_BN_CLICKED(IDC_RADIO2, OnRadioWIN32)
   ON_BN_CLICKED(IDC_RADIO3, OnRadioWIN32Nested)
   ON_BN_CLICKED(IDC_RADIO4, OnRadioMFC)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
/
// CSquareClientDlg Nachrichten-Handler

BOOL CSquareClientDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Hinzufügen des Menübefehls "Info... " zum Systemmenü.
   // IDM_ABOUTBOX muss sich im Bereich der Systembefehle
   // befinden.
```

```
   ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
   ASSERT(IDM_ABOUTBOX < 0xF000);

   CMenu* pSysMenu = GetSystemMenu(FALSE);
   if (pSysMenu != NULL)
   {
      CString strAboutMenu;
      strAboutMenu.LoadString(IDS_ABOUTBOX);
      if (!strAboutMenu.IsEmpty())
      {
         pSysMenu->AppendMenu(MF_SEPARATOR);
         pSysMenu->AppendMenu(MF_STRING,
                              IDM_ABOUTBOX,
                              strAboutMenu);
      }
}

   // Symbol für dieses Dialogfeld festlegen.
   // Wird automatisch erledigt, wenn das Hauptfenster der
   // Anwendung kein Dialogfeld ist
   SetIcon(m_hIcon, TRUE);   // Großes Symbol verwenden
   SetIcon(m_hIcon, FALSE);  // Kleines Symbol verwenden

   // Zunächst Eingabefeld abschalten
   EnableInput (false);

   return TRUE;
}

void CSquareClientDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
   if ((nID & 0xFFF0) == IDM_ABOUTBOX)
   {
      CAboutDlg dlgAbout;
      dlgAbout.DoModal();
   }
   else
   {
      CDialog::OnSysCommand(nID, lParam);
   }
}

// Wollen Sie Ihrem Dialogfeld eine Schaltfläche "Minimieren"
// hinzufügen, benötigen Sie den nachstehenden Code, um das
// Symbol zu zeichnen. Für MFC-Anwendungen, die das
// Dokument/Ansicht-Modell verwenden, wird dies automatisch
für
// Sie erledigt.

void CSquareClientDlg::OnPaint()
{
```

```
   if (IsIconic())
   {
      CPaintDC dc(this); // Gerätekontext für Zeichnen

      SendMessage(WM_ICONERASEBKGND,
                  (WPARAM) dc.GetSafeHdc(), 0);

      // Symbol in Client-Rechteck zentrieren
      int cxIcon = GetSystemMetrics(SM_CXICON);
      int cyIcon = GetSystemMetrics(SM_CYICON);
      CRect rect;
      GetClientRect(&rect);
      int x = (rect.Width() - cxIcon + 1) / 2;
      int y = (rect.Height() - cyIcon + 1) / 2;

      // Symbol zeichnen
      dc.DrawIcon(x, y, m_hIcon);
   }
   else
   {
      CDialog::OnPaint();
   }
}

// Die Systemaufrufe fragen die Cursorform ab, die angezeigt
// werden soll, während der Benutzer das zum Symbol
// verkleinerte Fenster mit der Maus zieht.
HCURSOR CSquareClientDlg::OnQueryDragIcon()
{
   return (HCURSOR) m_hIcon;
}

void CSquareClientDlg::OnRadioATL()
{
   // CLSID des ATL-Servers
   static const CString strCLSID
      (_T("{547E8381-F175-11D1-90A2-D0DD2F7ACA49}"));

   if (AttachToServer (strCLSID))
   {
      m_strMessage = _T("Verbunden mit ATL-Server: ") +
                     strCLSID;
      EnableInput (true);
   }
   else
   {
      m_strMessage = _T("Verbindung mit ATL-Server konnte "
                        "nicht hergestellt werden.");
      EnableInput (false);
   }
```

```
   UpdateData (false);
}

void CSquareClientDlg::OnRadioWIN32()
{
   // CLSID des WIN32-Servers
   static const CString strCLSID
      (_T("{B8EF6063-F1C2-11d1-90A2-D0DD2F7ACA49}"));

   if (AttachToServer (strCLSID))
   {
      m_strMessage = _T("Verbunden mit WIN32-Server: ") +
                     strCLSID;
      EnableInput (true);
   }
   else
   {
      m_strMessage = _T("Verbindung mit WIN32-Server "
                        "konnte nicht hergestellt werden.");
      EnableInput (false);
   }

   UpdateData (false);
}

void CSquareClientDlg::OnRadioWIN32Nested()
{
   // CLSID des CPP-Servers (nested)
   static const CString strCLSID
      (_T("{E3C91780-F24F-11d1-90A2-D0DD2F7ACA49}"));

   if (AttachToServer (strCLSID))
   {
      m_strMessage = _T("Verbunden mit WIN32-Server (nested):
")
                     + strCLSID;
      EnableInput (true);
   }
   else
   {
      m_strMessage = _T("Verbindung mit WIN32-Server (nested)
"
                        "konnte nicht hergestellt werden.");
      EnableInput (false);
   }

   UpdateData (false);
}

void CSquareClientDlg::OnRadioMFC()
{
```

```
   // CLSID des MFC-Servers
   static const CString strCLSID
      (_T("{4831F131-F290-11D1-90A2-D0DD2F7ACA49}"));

   if (AttachToServer (strCLSID))
   {
      m_strMessage = _T("Verbunden mit MFC-Server: ") +
                     strCLSID;
      EnableInput (true);
   }
   else
   {
      m_strMessage = _T("Verbindung mit MFC-Server konnte
      nicht"
                        "hergestellt werden.");
      EnableInput (false);
   }

   UpdateData (false);
}

////////////////////////////////////////////////////////////
// AttachToServer
//
// Diese Funktion baut die Verbindung mit dem COM-Server auf.
//
BOOL CSquareClientDlg::AttachToServer(const CString &
strCLSID)
{
   // IID für ISquare als String:
   static const CString strIIDSquare
      (_T("{547E8380-F175-11D1-90A2-D0DD2F7ACA49}"));
   CLSID clsid;
   IID iid;
   HRESULT hRes;
   IUnknown *pIUnknown;
   IClassFactory *pIClassFactory;

   // Wenn noch eine ISquare-Schnittstelle gehalten wird,
   // diese erst freigeben:
   if (m_pISquare)
   {
      m_pISquare->Release ();
      m_pISquare = NULL;
   }

   // String in eine CLSID umwandeln:
   if (FAILED(AfxGetClassIDFromString (strCLSID, &clsid)))
   {
      MessageBox (_T("CLSID hat ungültiges Format"));
      return false;
```

```
   }

   // String in eine IID umwandeln:
   if (FAILED(AfxGetClassIDFromString (strIIDSquare, &iid)))
   {
      MessageBox (_T("IID hat ungültiges Format"));
      return false;
   }

   // Zeiger auf IUnknown-Schnittstelle besorgen:
   hRes = CoGetClassObject (clsid,
                            CLSCTX_ALL,
                            NULL,
                            IID_IUnknown,
                            (void**)&pIUnknown);
   if (FAILED (hRes))
   {
      ShowCOMError (hRes);
      return false;
   }

   // Fragen, ob das Klassenobjekt
   // eine Klassenfabrik implementiert:
   pIUnknown->QueryInterface (IID_IClassFactory,
                              (void**)&pIClassFactory);
   // IUnknown-Schnittstelle freigeben:
   pIUnknown->Release ();
   if (!pIClassFactory)
   {
      MessageBox (_T("COM-Server hat keine Klassenfabrik!"));
      return false;
   }

   // Object instanziieren, ISquare-Schnittstelle anfordern:
   pIClassFactory->CreateInstance (NULL,
                                   iid,
                                   (void**)&m_pISquare);
   // IClassFactory-Schnittstelle freigeben:
   pIClassFactory->Release ();
   if (!m_pISquare)
   {
      MessageBox (_T("Konnte kein Objekt erzeugen"));
      m_pISquare = NULL;
      return false;
   }

   return true;
}

void CSquareClientDlg::ShowCOMError(HRESULT hRes)
{
```

```
   CString strErr;

   switch (hRes) {
      case REGDB_E_CLASSNOTREG:
         strErr = _T("CLSID nicht registriert.");
         break;

      case CO_E_DLLNOTFOUND:
         strErr = _T("COM-Server-DLL nicht gefunden");
         break;

      case CO_E_APPNOTFOUND:
         strErr = _T("COM-Server-EXE nicht gefunden");
         break;

      case E_ACCESSDENIED:
         strErr = _T("Zugriffsfehler");
         break;

      case CO_E_ERRORINDLL:
         strErr = _T("Fehler in COM-Server-Datei");
         break;

      case CO_E_APPDIDNTREG:
         strErr = _T("Server gestartet, aber hat sich "
                     "nicht registriert!");
         break;

      default:
         strErr = _T("Unbekannter Fehler");
         break;
   }

   MessageBox (strErr);
}

void CSquareClientDlg::OnOK()
{
   // Nur ausführen, wenn eine Verbindung mit
   // einem COM-Server besteht:
   if (m_pISquare)
   {
      long nResult;

      UpdateData (true);
      // Methode des COM-Servers aufrufen:
      m_pISquare->GetSquare (m_nIn, &nResult);
      m_strOut.Format ("%li", nResult);

      UpdateData (false);
   }
```

```
}

void CSquareClientDlg::EnableInput(BOOL bEnable)
{
   if (bEnable)
      GetDlgItem (IDC_EDIT1)->EnableWindow (true);
   else
      GetDlgItem (IDC_EDIT1)->EnableWindow (false);

   m_nIn = 0;
   m_strOut = _T("");

   UpdateData (false);
}
```

Listing 3.6: Implementierungsdatei des Programms SquareClient

Verbindung mit dem COM-Server

Die Verbindung zu einem COM-Server wird aufgebaut, indem man auf eins der Optionsfelder klickt. Für jedes Optionsfeld ist eine Nachrichtenbehandlungsfunktion angelegt worden. Von der Struktur her sind diese vier Funktionen – *OnRadioATL, OnRadioWIN32, OnRadioWIN32Nested* und *OnRadioMFC* – gleich. Die CLSID des anzusprechenden COM-Servers wird als konstanter String definiert. Durch Aufruf der Funktion *CSquareClientDlg::AttachToServer* wird versucht, eine Verbindung zum gewählten COM-Server aufzubauen. Je nachdem, ob die Verbindung hergestellt werden konnte oder nicht, wird eine Fehlermeldung oder eine Erfolgsmeldung in die DDX-Variable *m_strMessage* geschrieben. Diese Meldung wird durch den abschließenden Aufruf von *UpdateData* in ein statisches Textfeld des Dialogfelds übernommen und damit angezeigt. Die Funktion *CSquareClientDlg::EnableInput* sperrt das Eingabefeld gegen Eingaben, falls keine Verbindung mit dem COM-Server hergestellt werden konnte.

Das Beispielprogramm SquareClient hält immer nur eine Verbindung zu maximal einem COM-Server. Wird eine Verbindung zu einem anderen COM-Server gewünscht, so muss eine eventuell noch bestehende Verbindung vorher abgebaut werden. Als Indikator, ob eine Verbindung zu einem COM-Server besteht, wird die Variable *m_pISquare* verwendet. Hat *m_pISquare* den Wert *NULL*, so bedeutet dies, dass keine COM-Server-Verbindung besteht. Werte ungleich *NULL* bedeuten, dass *m_pISquare* ein Zeiger auf eine gültige *ISquare*-Schnittstelle ist und somit eine Verbindung zu einem COM-Server besteht.