Universität Hamburg
Fachbereich Informatik

Diplomarbeit

Dynamische Aktualisierungsmechanismen komponentenbasierter Software

eingereicht bei:
Prof. Dr. Winfried Lamersdorf
Arbeitsbereich Verteilte Systeme (VSYS)

Zweitbetreuer:
Prof. Dr. Heinz Züllighoven
Arbeitsbereich Softwaretechnik (SWT)

vorgelegt von:
Christian Martens
Am Backofen 22
22339 Hamburg

Hamburg, den 20. August 1999

Inhaltsverzeichnis

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Software-Entwicklung ist ein *evolutionärer* Prozeß. Betriebswirtschaftliche Rahmenbedingungen, gesetzliche Vorschriften, Kommunikationswege und Kommunikationsarten sowie Geschäftsprozesse und Arbeitsabläufe - das gesamte Umfeld, in dem Software eingesetzt wird, befindet sich in einem steten Fluß. Um den Anforderungen ihrer Benutzer gerecht zu werden, muß die Software dem Wandel ihrer Umgebung Rechnung tragen.

Dies bedingt, daß von Zeit zu Zeit installierte Software-Einheiten durch andere Software-Einheiten ersetzt werden, wodurch z.B. Fehler bereinigt werden oder neue Funktionalität zur Verfügung gestellt wird. Der damit verbundene Aktualisierungsvorgang kann heute in aller Regel nur vorgenommen werden, wenn während der Aktualisierung auf die zu ersetzende Software-Einheit nicht zugegriffen wird. Diese Einschränkung ist nicht wünschenswert:

Zum einen können Administratoren den Zeitpunkt, zu dem sie die Aktualisierung vornehmen, nicht beliebig wählen, sondern müssen auf bestimmte Wartungsfenster (z.B. in der Nacht oder am Wochenende) ausweichen. Zum anderen bringt dies zusätzlichen Aufwand für Systeme, die 24 Stunden am Tag und sieben Tage in der Woche verfügbar sein sollen. In diesem Fall muß dafür gesorgt werden, daß während der Aktualisierung auf ein Backup-System zugegriffen werden kann.

1.2 Ziel

Aus der geschilderten Situation ergibt sich die Herausforderung, einen Mechanismus zu finden, der eine *dynamische Aktualisierung* von Software zur Laufzeit zuläßt (Online-Update). Dabei wollen wir uns hier auf den Bereich der *komponentenbasierten Software* konzentrieren. Das Paradigma der Komponentenorientierung gewinnt im Bereich der Software-

Entwicklung zunehmend an Bedeutung. Auf die Idee von Software-Komponenten und insbesondere auf ihre Vorteile werden wir im Kapitel 1.4 noch einmal kurz eingehen.

Wir wollen in dieser Arbeit untersuchen, in welcher Weise dynamische Aktualisierungsmechanismen in einer komponentenbasierten Software-Umgebung realisiert werden können. Dazu soll eine konkrete Lösung für die Komponentenarchitektur von Microsoft - das Distributed Component Object Model (DCOM) - entworfen und beispielhaft implementiert werden.

1.3 Gliederung

Wir betrachten anfangs in Kapitel 2 einige neuere Aktualisierungs-mechanismen unter der Fragestellung, inwiefern diese durch die Möglichkeit eines Online-Updates bereichert werden könnten. In den folgenden Kapiteln der Arbeit soll die entwickelte Lösung erläutert werden.

Dies beginnt in Kapitel 3 mit einer Erläuterung von Microsoft’s DCOM, wobei besonders auf die in unserem Zusammenhang relevanten Aspekte eingegangen wird. Im Anschluß wird in Kapitel 4 die Architektur der entwickelten Lösung skizziert und anhand des Beispiels eines Verkaufs-Szenarios veranschaulicht. Schließlich gehen wir in Kapitel 5 auf die wichtigsten Implementierungsdetails ein, wobei insbesondere die aufgetretenen Probleme sowie Wege zu Ihrer Lösung geschildert werden.

Die Arbeit schließt in Kapitel 6 mit einer Zusammenfassung sowie einem Ausblick über potentielle Weiterentwicklungen. Insbesondere ist hier die Möglichkeit der Integration des dynamischen Aktualisierungsmechanismus in ein Versions-Management-Tool zu erwähnen.

In den Anhängen finden sich Literaturhinweise, ein Verzeichnis der Abbildungen sowie eine Liste der verwendeten Abkürzungen.

1.4 Komponentenbasierte Software

Das Paradigma der *Komponentenorientierung* besagt, daß Softwaresysteme nicht in Form von großen monolithischen Modulen entwickelt werden sollten, sondern als Systeme kooperierender *Komponenten*. Leider wird der Begriff „Komponente“ mit zum Teil extrem unterschiedlicher Semantik verwendet. Wir beziehen uns hier auf die Definition aus [Griffel 98]:

> *Eine Komponente ist ein Stück Software, das klein genug ist, um es in einem Stück erzeugen und pflegen zu können, groß genug ist, um eine sinnvoll einsetzbare Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten.*

Eine Komponentenarchitektur muß zum einen die Rahmenbedingungen spezifizieren, denen Komponenten genügen müssen, und zum anderen eine Infrastruktur (*Middleware*) zur Verfügung stellen, über die Komponenten miteinander kommunizieren können.

Komponentenbasierte Software bringt im Idealfall eine Reihe von Vorteilen mit sich:

- Leichtere Wiederverwendbarkeit und Anpaßbarkeit bereits entwickelter Software („plug & play“). Dabei ist es nicht nötig, den Sourcecode verstehen oder überhaupt besitzen zu müssen. Daraus resultiert eine gesteigerte Produktivität bei der Software-Entwicklung.
- Stabilere, weniger fehleranfällige Systeme. Denn zum einen sind einzelne Komponenten klein genug, um für den Entwickler überschaubar zu sein und zum anderen wird bei der Wiederverwendung von Komponenten kein Sourcecode verändert.
- Interoperabilität über Netzwerke und Plattformen hinweg. Diese wird von der Komponentenarchitektur bereitgestellt.

- Anbindung alter Systeme (legacy code) durch Proxy-Komponenten, die eine einheitliche Schnittstelle bereitstellen.
- Schließen der semantischen Lücke (siehe [Mertens 92]) zwischen softwarespezifischem und anwendungsspezifischem Wissen durch Aufteilung der Entwicklungsarbeit auf Komponentenentwickler einerseits und Anwendungsentwickler andererseits.
- Größere Chancen für kleinere Software-Produzenten, kleine aber innovative Teile für ein großes Gesamtsystem zu entwickeln.
- Leichtere Pflegbarkeit der Software durch Austausch einzelner Teile anstatt eines großen Gesamtsystems.
- Dynamische Aktualisierung von Systemen zur Laufzeit (Gegenstand dieser Diplomarbeit).

2 Aktualisierungsmechanismen

2.1 Überblick

In diesem Kapitel stellen wir verschiedene Methoden zur Aktualisierung von Software vor. Zwei jüngere Methoden sollen außerdem daraufhin untersucht werden, inwieweit sie von einem dynamischen Aktualisierungsmechanismus profitieren könnten.

Wir beginnen mit der *Demeter Methode* von K. J. Lieberherr. Dieses Verfahren dient zur automatischen Aktualisierung von Klassen einer objektorientierten Software. Es wird in Kapitel 2.2 vorgestellt.

In Kapitel 2.3 gehen wir dann auf die Möglichkeiten der *Active Channel Technology* von Microsoft ein. Insbesondere soll dabei eine Sonderform vorgestellt werden, die als *Software Update Channel* bezeichnet wird.

Wir schließen in Kapitel 2.4 mit einem Überblick über weitere verwandte Arbeiten auf dem Gebiet der dynamischen Aktualisierung und Rekonfiguration.

2.2 Demeter Methode

Die Demeter Methode von K. J. Lieberherr beschreibt einen Weg, Softwaresysteme um *Adaptivität* zu bereichern. Adaptivität wird dabei verstanden als die Fähigkeit von Software, ihr Verhalten an den jeweiligen *Kontext* anzupassen. In „Adaptive Object-Oriented Software - The Demeter Method“ [Lieberherr 96] skizziert der Autor die Vorgehensweise für objektorientierte Software.

Bei objekt-orientierter Software bilden die Klassenstrukturen den Kontext, d.h. sie sind bei Demeter nicht wie üblich integraler Bestandteil des Programmes, sondern der zu implementierende Algorithmus verwendet eine *generische Klassenstruktur*. Diese beschreibt bestimmte *Strukturbedingungen*, d.h. sie enthält bestimmte Elemente, welche im Programm

verwendet werden. Diese Elemente (Hooks) bezeichnen Knoten oder Pfade der Klassenstruktur.

Jede *spezielle Klassenstruktur*, die diese Hooks enthält und somit die Strukturbedingungen erfüllt, kann als Parameter (Customizer) des adaptiven Programms verwendet werden. Bei dem Prozeß der *Customization* wird aus dem adaptiven Programm und einer speziellen Klassenstruktur ein spezielles objekt-orientiertes Programm erstellt. Die beschriebene Vorgehensweise zur Entwicklung adaptiver Software ist in Abbildung 1 dargestellt.

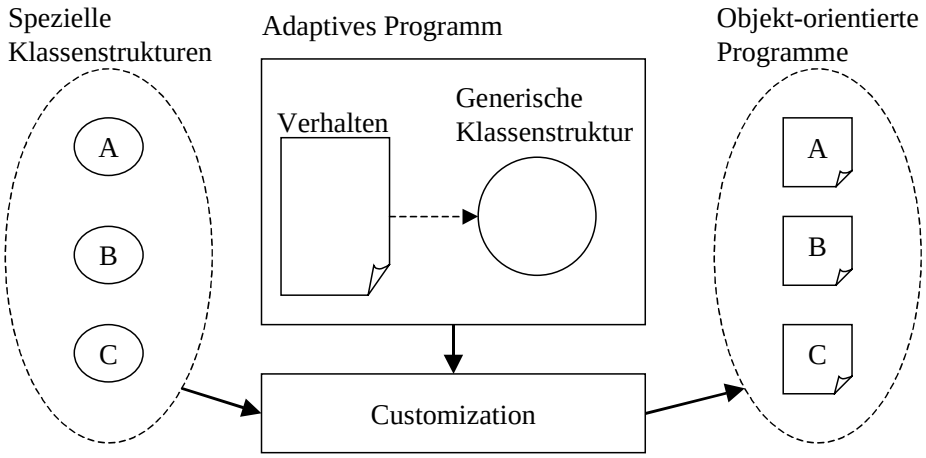

Abbildung 1: Adaptive Software

Ein *adaptives Programm* wie in Abbildung 1 dargestellt definiert Lieberherr wie folgt:

> *An **adaptive program** is a generic process model parameterized by graph constraints which define compatible structural models as parameters of the process model.*

Adaptive Software definiert also nicht ein spezielles Programm, sondern eine Familie von Programmen. Dadurch kann ein Programm weiter verwendet werden, auch wenn sich die zugrundeliegende Klassenstruktur geändert hat.

Ein praktisches Beispiel soll das geschilderte Verfahren veranschaulichen. Wir betrachten Abbildung 2:

a) Ursprüngliche Klassenstruktur:

b) Klassenstruktur nach der Umstrukturierung:

Abbildung 2: Änderung der Klassenstruktur

Ein Konzern ist intern strukturiert wie in Abbildung 2a dargestellt. Er verwendet ein adaptives Programm „Umsatz pro Land“, um seinen Umsatz in einem bestimmten Land zu berechnen. Die Strukturbedingungen dafür sind: Es gibt in der Klassenstruktur Knoten **Konzern**, **Umsatz** und **Land** sowie einen Pfad von **Konzern** zu **Umsatz** über **Land**. Nach einer Umstrukturierung des Konzerns kann aus dem adaptiven Programm und der neuen Klassenstruktur (Abbildung 2b) ohne manuelle Korrekturen eine neue Version des Programms „Umsatz pro Land“ erzeugt werden, welches der neuen Konzernstruktur Rechnung trägt.

Zur Spezifikation des adaptiven Programms wird die objekt-orientierte Sprache um bestimmte Sprachelemente („propagation patterns“) erweitert. Diese dienen dazu, Bezüge zu der generischen Klassenstruktur herzustellen. Die Klassenstruktur wiederum kann - je nach Entwicklungswerkzeug - entweder graphisch oder in Textform notiert werden.

Adaptive Software bringt eine Reihe von Vorteilen mit sich:

- Adaptive Programme können automatisch geänderten Strukturen angepaßt werden und sind somit leichter wiederverwendbar.
- Durch die notwendige Generalisierung kann die Aufgabe des Programmierers vereinfacht werden („paradox of the inventor“).

- Adaptive Programme sind leichter verstehbar, da sie unnötige Details der Klassenhierarchie ausblenden.
- Zu Beginn der Entwicklung muß noch nicht die gesamte Klassenstruktur bekannt sein, sondern es kann zunächst mit einer vereinfachten Struktur gearbeitet werden.

Die Möglichkeit eines Online-Updates wird in Bezug auf adaptive Software besonders interessant:

Man denke sich ein Szenario, in dem Software-Ingenieure die Demeter Methode zur Entwicklung von Komponenten verwenden. Zu beliebigen Zeitpunkten kann nun die Klassenstruktur durch ein graphisches Werkzeug geändert werden. Dies hätte eine erneute Übersetzung der Komponenten zur Folge. Dank der Möglichkeit einer dynamischen Aktualisierung könnten die alten Komponenten anschließend automatisch durch die neuen Versionen ersetzt werden.

2.3 Active Channel Technology

Unter der Bezeichnung *Active Channel Technology* hat Microsoft ein Pseudo-Push-Verfahren implementiert, um Daten aus dem Internet auf die Windows-Benutzeroberfläche zu bringen. Dabei kann es sich um Inhalte (HTML-Seiten) oder Programme wie z.B. Java Applets oder ActiveX Komponenten handeln. Realisiert wird dies durch einen automatisierten Pull-Mechanismus, der durch den Benutzer konfiguriert werden kann und in regelmäßigen Zeitabständen Inhalte aktualisiert.

Ein *Active Channel* (oder kurz *Channel*) ist eine Internet-Site, die durch eine CDF-Steuerungsdatei beschrieben ist. CDF (Channel Definition Format) ist eine XML-Anwendung, die von Microsoft für diesen Zweck entwickelt wurde.

Eine CDF-Datei enthält im wesentlichen einen <CHANNEL>-Tag. Dieser enthält wiederum die HTTP-Adresse der Site sowie eine Reihe von Unterelementen:

- <TITLE>: Der Titel des Channel.
- <ABSTRACT>: Eine kurze Beschreibung der Inhalte.
- <LOGO>: Ein Symbol für die Arbeitsoberfläche.
- <SCHEDULE>: Die Definition des Zeitverhaltens, d.h. die Angabe, wie oft die Daten aktualisiert werden sollen.
- <ITEM>: Die Definition der eigentlichen Inhalte, z.B. die HTTP-Adresse einer HTML-Datei.

Beispiele für denkbare Anwendungen wären: Ticker mit Börsenkursen, Kurznachrichten oder eben Aktualisierung von Komponenten. Letzteres Beispiel wollen wir nun genauer betrachten.

Die Aktualisierung von ActiveX Komponenten wird durch einen speziellen Channel-Typ unterstützt, der als *Software Update Channel* bezeichnet wird. Zweck eines Software Update Channel ist es zum einen, den Benutzer über neue Versionen zu informieren, und zum anderen, ihn beim Download und bei der Installation von neuen Versionen zu unterstützen. Letzteres wird realisiert durch die Analyse von Eigenschaften der Systemumgebung wie Prozessor, Sprache oder Betriebssystem. Durch die Angaben in der CDF-Datei kann dann automatisch die passende Komponente ermittelt werden.

Software Update Channels bieten über das <SCHEDULE>-Tag die Möglichkeit, Komponenten in regelmäßigen Zeitabständen zu aktualisieren, falls eine neue Version auf dem angegebenen Server vorhanden ist. Diese Aktualisierung kann jedoch nicht zur Laufzeit erfolgen. Eine weitergehende Vision wäre nun folgende:

Der in dieser Arbeit gezeigte dynamische Aktualisierungsmechanismus wird durch das Betriebssystem zur Verfügung gestellt. Wenn eine neue Version einer Komponente auf den Server gebracht wird, so kann diese über einen Software Update Channel auf den lokalen Rechner gebracht werden und dort zur Laufzeit die alte Version ersetzen. Dieses Szenario ist auf der einen Seite für Rechner im Intranet einer Firma interessant, die Software-Komponenten in einer verteilten Umgebung einsetzt. Auf der anderen Seite

könnte es im Heimbereich dem Anwender Entlastung bei der Pflege seiner Software bringen.

Derartige Aktualisierungsszenarien setzen selbstverständlich voraus, daß ein berechtigtes Vertrauen in den Mechanismus sowie die daran beteiligten Akteure besteht. Weiterhin sind in Abhängigkeit von der Art der Änderung möglicherweise unterschiedliche Verfahren angebracht, z.B. automatische Aktualisierung bei Änderung der Unterversionsnummer vs. Rückfrage bei Änderung der Hauptversionsnummer.

Ausführliche Informationen über Active Channel Technology finden sich auf der Microsoft Homepage unter

http://msdn.microsoft.com/workshop/delivery/channel/overview/overview.asp

2.4 Weitere verwandte Arbeiten

Der Bereich der dynamischen Aktualisierung bzw. Rekonfiguration ist ein relativ junges Forschungsgebiet. Dennoch haben sich schon eine Reihe von Autoren mit dem Thema befaßt und unterschiedliche Ansätze vorgeschlagen:

Bereits zu einem frühen Zeitpunkt wurde in [Bloom 83] eine Unterstützung für dynamische Modifikation von Software durch die Programmier-umgebung vorgestellt. Der Ansatz basiert auf einer Erweiterung des am MIT entwickelten Argus Programming System, welches vergleichbar ist mit einer virtuellen Maschine, in der Softwaremodule in einer verteilten Umgebung ausgeführt werden. Durch eine zusätzliche Einrichtung wird Argus um einen speziellen Befehlssatz erweitert. Dieser enthält Befehle wie z.B. **all_instances**, **replace**, **rebind**, **get_state** und **put_state**, durch die der Benutzer einen Aktualisierungsvorgang „per Hand“ durchführen kann.

In [Endler 93] wird eine eigene Hochsprache namens GEREL (Generic Reconfiguration Language) vorgestellt, die zur Beschreibung von dynamischen Rekonfigurationsvorgängen dient. Ziel dabei ist, nebenläufige Rekonfigurationsvorgänge zu synchronisieren und sicherzustellen, daß die

zu aktualisierende Software zu jedem Zeitpunkt in einer konsistenten Konfiguration vorliegt. Dafür werden die spezifizierten Rekonfigurationen vor ihrer Ausführung durch den GEREL Interpreter auf ihre Gültigkeit überprüft.

[Veitch 96] befaßt sich besonders mit der dynamischen Rekonfiguration des Betriebssystem-Kernels. Dies ist ein besonders interessantes Problem, da viele Ansätze darauf basieren, daß die Funktionalität zur dynamischen Aktualisierung durch das Betriebssystem bereitgestellt wird, diese Funktionalität sich jedoch nicht auf das Betriebssystem selbst erstreckt.

Bei Arbeiten in diesem Forschungsbereich werden zum Teil unterschiedliche Terminologien verwendet. So unterscheidet [Kramer 90] zwischen *Programmed Changes* und *Ad-hoc Changes*. Dabei bezeichnen *Programmed Changes* statische Aktualisierungsvorgänge, die bereits zum Übersetzungszeitpunkt festgelegt werden, während mit *Ad-hoc Changes* dynamische Aktualisierungsvorgänge zur Laufzeit bezeichnet werden. [Purtilo 91] unterscheidet weiterhin zwischen *Module Implementation Changes* und *Structural Changes*. Ersteres meint die Aktualisierung von Softwaremodulen oder Komponenten, letzteres die Änderung der Beziehung zwischen den einzelnen Modulen.

Zusammenfassend läßt sich feststellen, daß die Bedeutung von dynamischer Aktualisierung bzw. Rekonfiguration allgemein anerkannt ist und bereits eine Reihe interessanter Vorschläge zu ihrer Realisierung existieren. Es bleibt abzuwarten, wie schnell sich diese Ideen durchsetzen und einen hohen Verbreitungsgrad erreichen.

3 Distributed Component Object Model (DCOM)

3.1 Überblick

In den nachfolgenden Kapiteln zeigen wir einen dynamischen Aktualisierungsmechanismus für die Komponentenarchitektur von Microsoft: Das Distributed Component Object Model (DCOM). Dieses Kapitel bietet eine kurze Einführung in DCOM.

DCOM hat bereits eine längere Entwicklungsgeschichte hinter sich: Begonnen 1987 mit DDE (Dynamic Data Exchange) als einfache Form der IPC (Interprocess Communication), setzte sich die Entwicklung über OLE (Object Linking and Embedding) zum 1995 veröffentlichten Component Object Model (COM) fort, welches seit der Ergänzung um RPC (Remote Procedure Call) im Jahr 1996 als DCOM bekannt ist. Oftmals wird auch nur COM als „die“ Komponentenarchitektur von Microsoft bezeichnet. Wir beziehen uns in dieser Arbeit auf DCOM als Obermenge von COM.

Zunächst werden in Kapitel 3.2 die *Grundlagen* von DCOM beschrieben. Auf einige Besonderheiten dieser Architektur wird in der Folge detaillierter eingegangen, da sie im Zusammenhang mit dynamischer Aktualisierung von besonderer Relevanz sind.

Wenn eine Komponente zur Laufzeit ausgetauscht werden soll, dürfen keine statischen Bindungen zu ihr bestehen, da diese nach einer Aktualisierung nicht mehr gültig wären. Dynamische Bindungen werden in DCOM durch eine besondere Schnittstelle realisiert: *IDispatch* (Kapitel 3.3).

Nach einer Aktualisierung müssen alle Referenzen auf die neue Instanz verweisen. Dies wird hier durch die Verwendung indirekter Referenzen realisiert, die von einem Namensdienst zur Verfügung gestellt werden. Namensdienste werden in DCOM durch einen speziellen Komponententyp realisiert, den *Moniker* (Kapitel 3.4).

Weiterhin muß bei der Aktualisierung einer instanziierten Komponente beachtet werden, daß sich die einzelnen Instanzen in einem bestimmten Zustand befinden. Damit diese Zustandsinformationen nicht verlorengehen, müssen sie auf die neuen Instanzen übertragen werden. Der Standard-Mechanismus zum Lesen und Schreiben von Zustandsinformationen in DCOM basiert auf einer Menge von Standard-Schnittstellen, die unter dem Oberbegriff *Persistent Storage* zusammengefaßt werden (Kapitel 3.5).
Um die Aktualisierung durchführen zu können, ist Kommunikation zwischen den beteiligten Komponenten nötig. Dafür wurde ein ereignisorientierter Mechanismus gewählt, der bei DCOM durch das Konzept der *Connection Points* unterstützt wird (Kapitel 3.6).
Besonderes Augenmerk muß ein Entwickler von Komponenten schließlich auf die Probleme der Nebenläufigkeit legen; dies gilt in verstärktem Maße in einer Windows-Umgebung. Die in DCOM verwendeten Threading-Modelle basieren auf dem Konzept der *Apartments* (Kapitel 3.7).
Ausführliche Informationen über das Distributed Component Object Model finden sich in [Eddon 98]. Aktuelle Informationen zu COM bzw. DCOM veröffentlicht Microsoft im Internet unter
http://www.microsoft.com/com/comPapers.asp

3.2 Grundlagen

Dieses Kapitel soll die wesentlichsten Grundlagen von Microsoft's DCOM vermitteln. Wir erläutern zunächst in den ersten beiden Abschnitten *Komponenten* und *Schnittstellen* und gehen dann auf das Problem der *eindeutigen Identifikation* ein, welches in DCOM eine wichtige Rolle spielt. Im Anschluß werden die verschiedenen Arten der *Kommunikation* zwischen Komponenten bei DCOM umrissen. Um die passenden Hilfsmittel zur Kommunikation zur Verfügung stellen zu können, verwenden Komponentenarchitekturen eine spezielle Beschreibungssprache, in der Komponenten und Schnittstellen spezifiziert werden. Wir erläutern hier die

bei DCOM verwendete *Microsoft IDL* (Interface Definition Language). Schließlich gehen wir auf den Vorgang der Erzeugung neuer Instanzen von Komponenten ein. Dies wird bei DCOM über sogenannte *Klassenobjekte* realisiert.

3.2.1 Komponenten

Abbildung 3 zeigt die von Microsoft verwendete symbolische Darstellung einer Komponente in DCOM:

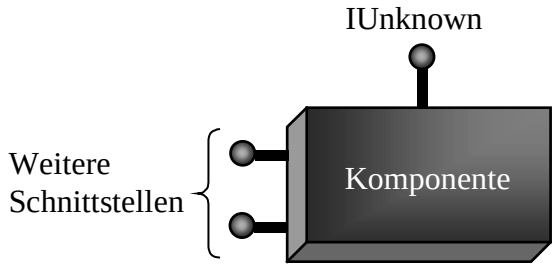

Abbildung 3: Symbolische Darstellung einer DCOM-Komponente

Jede Komponente kapselt eine bestimmte Funktionalität, die nach außen über Schnittstellen zur Verfügung gestellt wird. Schnittstellen werden in der Abbildung als kleine Stifte dargestellt, um ihre Bedeutung als Kontaktstellen zum Ausdruck zu bringen. Sie werden normalerweise am linken Rand der Komponente eingezeichnet.

Die Schnittstelle **IUnknown** spielt jedoch eine besondere Rolle und ist daher stets am oberen Rand der Komponente zu finden. Diese Schnittstelle wird von jeder DCOM-Komponente implementiert und stellt den ersten Zugang dar, den ein Benutzer einer Komponente zu einer neuen Instanz erhält. Über sie kann sich der Benutzer wiederum Zugang zu weiteren Schnittstellen verschaffen, die von der Komponente angeboten werden.

Eine Anmerkung zur Terminologie: Im Sprachgebrauch von Microsoft werden Komponenten als „Klassen“ bezeichnet. Als „Komponente“ wird wiederum eine binäre Datei bezeichnet, die eine oder mehrere Klassen enthalten kann. Um mit anderen Komponenten-Modellen konform zu

bleiben, wollen wir in dieser Arbeit voraussetzen, daß eine binäre Datei genau eine Hauptklasse (sowie möglicherweise einige Hilfsklassen) enthält und diese Hauptklasse als „Komponente“ bezeichnen. Als „Objekt“ wiederum bezeichnen wir die Instanz einer Komponente.

3.2.2 Schnittstellen

Jede Schnittstelle stellt eine Anzahl von *Methoden* und *Eigenschaften* zur Verfügung, wobei Eigenschaften durch Put- und Get-Methoden implementiert werden. Schnittstellen können entweder standardisiert (wie z.B. **IUnknown**) oder benutzerdefiniert sein.

Schnittstellen können voneinander erben, d.h. sie übernehmen alle Methoden ihrer Eltern. So muß beispielsweise jede DCOM-Schnittstelle von **IUnknown** erben. *Mehrfachvererbung* (siehe [Josuttis 94]) ist dabei zulässig. Dadurch kann es jedoch zu Mehrdeutigkeiten kommen, wenn mehrere Eltern die gleiche Methode deklarieren. Diese Mehrdeutigkeiten müssen durch eine explizite **reinterpret_cast**-Operation aufgelöst werden.

3.2.3 Eindeutige Identifikation

Um Komponenten und Schnittstellen eindeutig bezeichnen zu können, wird ihnen eine eindeutige 128 Bit lange Ganzzahl zugeordnet: Die *GUID* (globally unique identifier). Diese in DCOM verwendeten GUIDs von Microsoft sind äquivalent zu den in DCE (Distributed Computing Environment) definierten UUIDs und werden in manchen Fällen auch so bezeichnet.

Die GUID einer Komponente wird als *Class-ID* (CLSID) bezeichnet, die GUID einer Schnittstelle als *Interface-ID* (IID).

3.2.4 Kommunikation

Das Kommunikationsmodell zwischen Komponenten bei DCOM basiert auf dem Client/Server-Modell:

Ein Client-Programm bindet sich an eine Instanz einer Server-Komponente, ruft Methoden der Komponente auf und gibt sie wieder frei. Der zugrundeliegende Kommunikationsmechanismus ist zwar für den Client transparent, jedoch abhängig vom Typ der Server-Komponente. Man unterscheidet drei Typen: *In-Process Server*, *Local Server* und *Remote Server*. Abbildung 4 zeigt die verschiedenen Kommunikationsmechanismen für alle drei Typen von Servern.

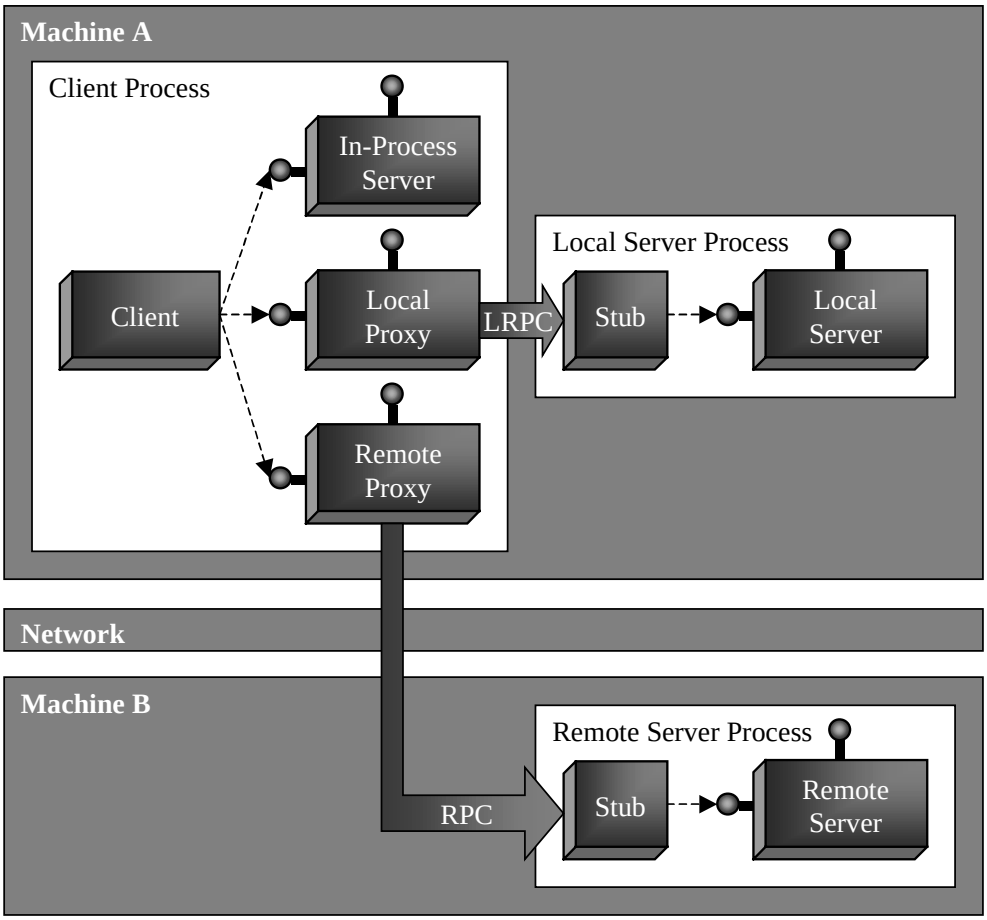

Abbildung 4: Kommunikationsmechanismen bei DCOM

Es folgt eine kurze Erläuterung der drei Server-Typen sowie des jeweiligen Kommunikationsmechanismus:

- *In-Process Server*: Sie laufen in demselben Prozeß wie der Client. In Windows-Betriebssystemen werden sie als DLL (Dynamic Link

Library) implementiert und direkt in den Adreßraum des Client geladen. Dadurch ist der Aufruf einer Methode genauso effizient wie ein normaler Funktionsaufruf.

- *Local Server*: Sie laufen in einem eigenen Prozeß auf derselben Maschine wie der Client. Methoden-Aufrufe werden über einen Mechanismus namens *Lightweight RPC* (LRPC) realisiert. Dies ist eine abgewandelte Form des RPC-Mechanismus für Aufrufe innerhalb einer Maschine, bei der Möglichkeiten zur Laufzeitoptimierung wie z.B. gemeinsame Speicherbereiche ausgenutzt werden.
- *Remote Server*: Aufrufe werden über RPC (Remote Procedure Call) an den DCOM-Server weitergeleitet. Dabei müssen - wie auch im Fall der Local Server - Parameter verflacht und in ein einheitliches Format konvertiert werden. Dieser Prozeß wird als *Marshalling* bezeichnet.

Eine gute Beschreibung des Kommunikationsmechanismus bei DCOM sowie ein Vergleich mit CORBA (Common Object Request Broker Architecture) und mit anderen Client/Server-Mechanismen findet sich in [Orfali 97]. Eine allgemeine Einführung in RPC als Dienst im Rahmen des OSI-Modells gibt [Kerner 93]; detaillierte Erläuterungen zu RPC und anderen Kommunikationsmechanismen bietet [Coulouris 94].

3.2.5 Microsoft IDL

Komponenten und Schnittstellen müssen in einer speziellen *Interface Definition Language* (IDL) spezifiziert werden, genannt *Microsoft IDL* (MIDL). Aus der IDL-Definition können dann mit Hilfe eines MIDL-Compilers die Client- und Server-Stubs generiert werden, welche zur Kommunikation zwischen den Komponenten benötigt werden.

MIDL basiert auf der IDL-Definition der OSF, die im Rahmen von RPC als Teil des DCE entwickelt wurde.

Als Beispiel betrachten wir die (leicht vereinfachte) IDL-Definition der Schnittstelle **IUnknown**:

```
[ object, uuid(00000000-0000-0000-C000-000000000046) ]
interface IUnknown
{
  HRESULT QueryInterface(
    [in] REFIID riid,
    [out, iid_is(riid)] void **ppvObject );
  ULONG AddRef();
  ULONG Release();
}
```

Das Schlüsselwort „`object`“ gibt an, daß es sich um eine DCOM-Definition handelt. Es folgt die GUID (bzw. UUID) der Schnittstelle. In dem Block „`interface IUnknown`“ werden schließlich die drei Methoden **QueryInterface()**, **AddRef()** und **Release()** deklariert. Die Methode **QueryInterface()** dient dazu, Zugang zu einer beliebigen Schnittstelle zu erhalten. Sie nimmt als Eingabeparameter („`[in]`“) eine Interface-ID entgegen und liefert als Rückgabeparameter („`[out]`“) einen Zeiger auf die gewünschte Schnittstelle zurück. (Schnittstellen sind vom Typ „`void *`“, daher die doppelte Indirektion.) Die Methoden **AddRef()** und **Release()** dienen der Referenzzählung: Wenn die Anzahl der Referenzen zu einer Instanz auf Null sinkt, wird die Instanz aus dem Speicher entladen.

3.2.6 Klassenobjekte

Um zur Laufzeit neue Instanzen einer Komponente erzeugen zu können, muß für jede Komponente (Hauptklasse) ein *Klassenobjekt* (Hilfsklasse) implementiert werden. Diese werden in [Eddon 98] wie folgt definiert:

> *A COM **class object** is an object that typically has no behavior except to create new instances of some other class. COM class objects normally implement the IClassFactory interface and thus are often referred to as **class factories**.*

Ein Klassenobjekt, das **IClassFactory** implementiert, wird also als *Klassenfabrik* bezeichnet. (Der semantisch korrekte Name wäre hingegen

„Objektfabrik“.) Die Standard-Schnittstelle **IClassFactory** enthält die Methode **CreateInstance()** zur Erzeugung neuer Instanzen. DCOM wiederum stellt die Funktion **CoCreateInstance()** zur Verfügung, welche Anforderungen an die zuständige Klassenfabrik weiterleitet.

3.3 IDispatch

Die Methoden einer verwendeten Schnittstelle werden stets statisch gebunden. Wenn eine Komponente *dynamische Bindung* ermöglichen soll, muß sie die Standard-Schnittstelle **IDispatch** implementieren. Diese enthält eine Methode **Invoke()**, über die eine beliebige andere Methode der Komponente aufgerufen werden kann, und zwar unabhängig davon, in welcher Schnittstelle sich diese andere Methode befindet. Die Angabe, welche Methode aufgerufen werden soll, erfolgt über eine eindeutige *Dispatch-ID* (DispID). Diese ist keine GUID, sondern eine laufende Nummer, die in der IDL-Definition explizit angegeben werden muß. Die Dispatch-IDs können zur Laufzeit über die Methode **GetIDsOfNames()** der **IDispatch** Schnittstelle erfragt werden.

Die in DCOM zur Verfügung gestellte Einrichtung der *Automation* (früher: *OLE Automation*) basiert auf dynamischer Bindung durch **IDispatch**. Automation ermöglicht das Fernsteuern von Anwendungen (z.B. Word, Excel, ...) durch den Aufruf von Methoden und Eigenschaften einer zur Applikation gehörenden Objekthierarchie.

3.4 Moniker

Ein Namensdienst wird in DCOM durch einen speziellen Komponententyp realisiert, den *Moniker*. Er dient dazu, einen Objektnamen in eine Objektreferenz aufzulösen. Beispiele für Moniker sind File Moniker (Auflösung von Dateinamen), URL Moniker (Auflösung von URLs) oder Class Moniker (Auflösung von Class-IDs).

Um ein Objekt per Moniker zu benennen, wird bei DCOM eine spezielle Zeichenfolge verwendet, genannt *Display Name*. Diese hat den Aufbau „ProgID:ObjectName“. Dabei gibt „ProgID“ den zu verwendenden Moniker, „ObjectName“ das gewünschte Objekt an.

Technisch zeichnet sich eine Moniker-Komponente dadurch aus, daß sie die Standard-Schnittstelle **IMoniker** implementiert. Außerdem implementiert das dazugehörige Klassenobjekt nicht **IClassFactory**, sondern die Standard-Schnittstelle **IParseDisplayName**.

Um zur Laufzeit eine neue Instanz eines Moniker erzeugen zu können, stellt DCOM die Funktion **MkParseDisplayName()** zur Verfügung. Diese gibt den übergebenen Display Name an die Methode **ParseDisplayName()** der Schnittstelle **IParseDisplayName** des zuständigen Klassenobjektes weiter. Diese erzeugt eine neue Instanz des Moniker.

Der instanziierte Moniker kann nun dazu verwendet werden, durch den Prozeß des *Binding* eine neue Instanz des eigentlich gewünschten Objektes zu erzeugen. Zu diesem Zweck stellt die **IMoniker** Schnittstelle die Methode **BindToObject()** zur Verfügung.

Sowohl beim Erzeugen der Moniker-Instanz als auch beim Erzeugen der Objekt-Instanz wird ein sogenannter *Bind Context* übergeben. Dieser enthält zusätzliche Informationen, die beim Binden des Moniker an das Objekt verwendet werden können. In Kapitel 5.4.15.4 werden wir auf die Verwendung des Bind Context zurückkommen.

In Windows 2000 spielen Moniker in Form des *Active Directory* eine zentrale Rolle. Detaillierte Informationen über das Active Directory finden sich auf der Microsoft Homepage unter

http://www.microsoft.com/Windows/server/Technical/directory/default.asp

3.5 Persistent Storage

Unter dem Begriff „Persistent Storage“ werden in DCOM eine Reihe von Standard-Schnittstellen zusammengefaßt, die in zwei Kategorien zerfallen: Persistenz auf Basis von Dateien und auf Basis von Komponenten.

3.5.1 Persistenz auf Basis von Dateien

Dieser Teil der Schnittstellen dient dem Zugriff auf Dateien, die in einem baumartig strukturierten Format abgelegt werden, genannt *Structured Storage*:

Der Zugriff auf die Datei erfolgt über einen *(Root) Storage*. Dieser kann eine Anzahl von *Streams* oder weitere *Storages* enthalten. Verglichen mit einem Dateisystem entspräche ein *Storage* einem Verzeichnis und ein *Stream* einer Datei. Aufgrund dieser Analogien spricht man auch vom „*file system within a file*“. Abbildung 5 zeigt ein Beispiel eines Structured Storage Files.

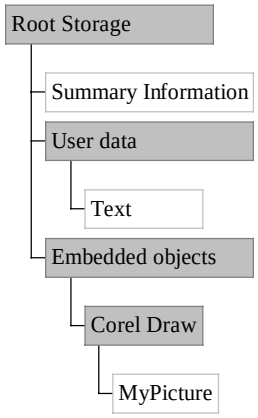

Abbildung 5: Structured Storage File

Als Standard-Schnittstellen zum Zugriff auf Structure Storage Files sind insbesondere **IStorage** und **IStream** zu nennen. **IStorage** beinhaltet Operationen, die vergleichbar sind mit solchen, welche in einem Dateisystem auf Verzeichnis-Ebene zu finden sind, z.B. zum Anlegen von Verzeichnissen, Verschieben von Verzeichnissen oder Wechseln des

aktuellen Verzeichnisses. **IStream** bietet Operationen analog dem Stream-orientierten Zugriff auf einzelne Dateien, z.B. zum Lesen oder Schreiben von Bytes.

3.5.2 Persistenz auf Basis von Komponenten

Der zweite Teil der Schnittstellen kann dazu verwendet werden, den aktuellen *Zustand* der Instanz einer Komponenten persistent abzulegen. Sie werden von dem Entwickler einer Komponente implementiert. Beispiele sind **IPersistStorage**, **IPersistStream** und **IPersistFile**. Alle diese Schnittstellen enthalten die Methoden **Load()** und **Save()**, wobei **IPersistStorage** den Zustand in einem Storage speichert, **IPersistStream** in einem Stream und **IPersistFile** auf herkömmliche Weise in einer binären Datei. Gemeinsam ist allen Schnittstellen dieser Kategorie, daß sie von der Standard-Schnittstelle **IPersist** erben. Eine Komponente unterstützt Persistenz also genau dann, wenn sie **IPersist** implementiert.

3.6 Connection Points

Die Standard-Kommunikation zwischen Komponenten bei DCOM basiert auf dem Client/Server-Modell: Eine Komponente (Client) erzeugt eine neue Instanz einer anderen Komponente (Server), ruft eine Anzahl von Methoden auf und gibt die Instanz wieder frei. Dieser Form der synchronen Kommunikation steht das Modell der asynchronen Kommunikation über Ereignisse gegenüber:

Eine Komponente A teilt einer anderen Komponente B mit, daß A an einem bestimmten Ereignis E interessiert ist. Wenn zu einem beliebigen späteren Zeitpunkt Ereignis E eintrifft, sendet B eine Nachricht „E ist eingetroffen“ an Komponente A. Ereignisorientierte Kommunikation wird in DCOM durch das Konzept der *Connection Points* realisiert.

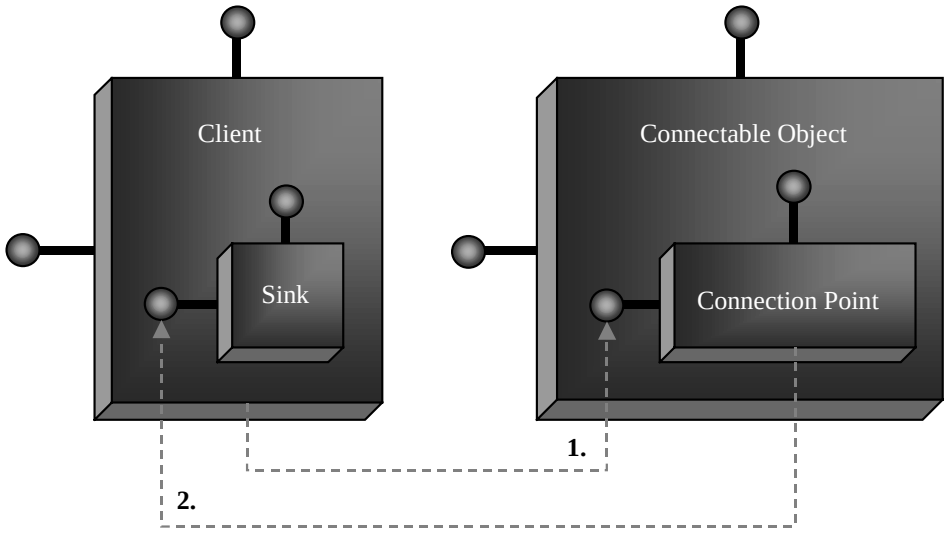

Abbildung 6: Connection Points

Zur Erläuterung betrachten wir Abbildung 6. Eine Komponente, die Ereignisse auslösen kann (Komponente B in obigem Beispiel), wird in DCOM als *Connectable Object* bezeichnet. Für jede Ereignisart besitzt diese Komponente ein Hilfsobjekt, den *Connection Point*. Dieser ist dafür zuständig, ein eingetroffenes Ereignis an alle Interessenten zu propagieren. Ein *Client*, der an einem bestimmten Ereignis E interessiert ist, kann sich bei dem zuständigen Connection Point für das Ereignis registrieren (**1.** in Abbildung 6). Er muß für das Ereignis E ebenfalls ein Hilfsobjekt besitzen, genannt *Sink*. Dieses nimmt später Nachrichten der Art „E ist eingetroffen“ entgegen (**2.** in Abbildung 6).

Der Sink muß also eine benutzerdefinierte Schnittstelle implementieren, über die das Connectable Object den Client zurückrufen kann, wenn E eingetroffen ist (Callback-Mechanismus). Diese Schnittstelle wird als *Outgoing Interface* (auch *Source Interface*) bezeichnet. Die Bezeichnung „Outgoing“ rührt daher, daß diese Schnittstelle nicht nur im Sink, sondern auch im Connection Point deklariert sein muß. Dort dient sie aber nur dem Senden, nicht dem Empfangen von Methoden-Aufrufen. In Abgrenzung zu

solchen Outgoing Interfaces werden alle anderen Schnittstellen auch als *Incoming Interfaces* bezeichnet.

Connection Points werden wieder über Standard-Schnittstellen realisiert: Ein Connectable Object muß die Schnittstelle **IConnectionPointContainer** implementieren. Über dessen Methode **FindConnectionPoint()** erhält ein Client Zugang zu einem bestimmten Connection Point. Der Connection Point muß wiederum die Schnittstelle **IConnectionPoint** implementieren. Diese enthält schließlich die Methode **Advise()**, mit der sich der Client für das spezifische Ereignis registrieren kann. Dabei übergibt er dem Connection Point einen Zeiger auf das Outgoing Interface seines Sink-Objektes.

Es bleibt festzuhalten, daß sich ereignisorientierte Kommunikation in DCOM relativ aufwendig gestaltet. Der Grund für diesen Aufwand ist darin zu suchen, daß dieses asynchrone Kommunikationsmodell letztlich wieder auf das Client/Server-Modell abgebildet wird: Der Connection Point tritt gegenüber dem Sink wieder als Client auf. Auf die Vorteile, die ein „echter“ asynchroner Kommunikationsmechanismus in DCOM hätte, werden wir im Ausblick (Kapitel 6.2) noch einmal zurückkommen. Eine Beschreibung eines asynchronen IPC-Mechanismus in Form eines Nachrichten-Systems findet sich in [Silberschatz 98].

3.7 Apartments

Bei der Entwicklung der Vorläufer von DCOM (DDE, OLE, COM) wurde in Microsoft’s Betriebssystem Windows noch kein Multithreading unterstützt. Erst mit dem Aufkommen von Windows 95 wurde das Betriebssystem um diese Fähigkeit erweitert. Dies ist die Ursache dafür, daß heute in DCOM sowohl Singlethreading als auch Multithreading unterstützt wird. Realisiert werden die verschiedenen Threading-Modelle durch das Konzept der *Apartments*.

Ein klassisches Threading-Modell bezeichnet einen *Prozeß* als eine Einheit, die Dinge besitzen kann („unit of ownership“). Ein Prozeß besitzt stets einen oder mehrere *Threads*. Ein Thread ist dabei ein Ausführungsstrang des Prozesses („unit of execution“). Ein *Apartment* ist nun eine Teileinheit eines Prozesses. Man unterscheidet zwei Arten von Apartments:

- *Singlethreaded Apartment* (STA): Dieses Apartment besitzt genau einen Thread. Er hat auf die Objekte im Apartment exklusiven Zugriff. Sämtliche Aufrufe in ein STA werden über eine Message Queue serialisiert.
- *Multithreaded Apartment* (MTA): Dieses Apartment besitzt einen oder mehrere Threads. Diese können auf die Objekte innerhalb des Apartments nebenläufig zugreifen.

In Abhängigkeit davon, welche Apartments eine Applikation verwendet, ordnet man ihr eines von vier Threading-Modellen zu (Abbildung 7):

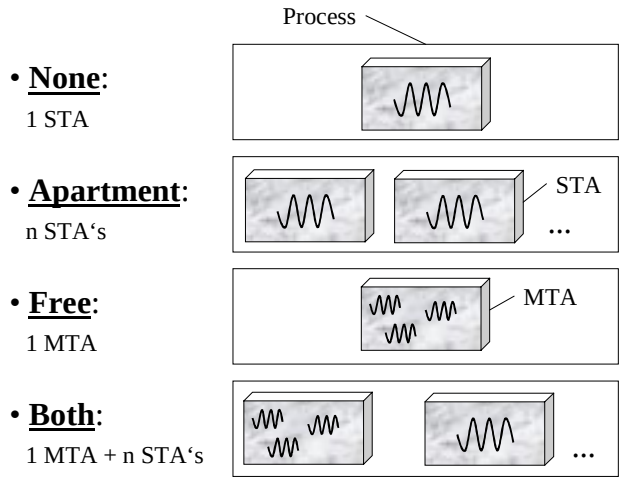

Abbildung 7: Threading-Modelle in DCOM

Die Threading-Modelle im einzelnen:

- *None*: Die Applikation verwendet genau ein STA. Dies trifft vor allem auf alte Applikationen zu, die keine Form von Multithreading verwenden.

- *Apartment*: Die Applikation verwendet ein oder mehrere STAs. Dieses Modell wird meist von GUI-basierten Applikationen verwendet.
- *Free*: Die Applikation verwendet genau ein MTA. Dies ist das Modell der Wahl für Server-basierte Anwendungen, die selbst die nötigen Synchronisationsmechanismen implementieren.
- *Both*: Die Applikation verwendet genau ein MTA und ein oder mehrere STAs.

Die Tatsache, daß Applikationen unterschiedliche Threading-Modelle verwenden, kann bei der Entwicklung zu zusätzlichen Komplikationen führen; siehe Kapitel 5.4.4.

4 Architektur

4.1 Einführung

In diesem Kapitel werden wir die Architektur einer konkreten Lösung zur dynamischen Aktualisierung in DCOM erläutern. Dazu skizzieren wir als anschauliches Beispiel in Kapitel 4.2 ein Verkaufs-Szenario mit kooperierenden Komponenten. Ziel ist es, einzelne Komponenten daraus zur Laufzeit zu aktualisieren. Dies wird in der entwickelten Lösung ermöglicht durch zwei besondere Komponenten: Den *ProxyMoniker* und den *UpdateManager*. Sie werden in Kapitel 4.3 näher erläutert. Schließlich wird in Kapitel 4.4 der Ablauf der dynamischen Aktualisierung in einzelnen Schritten gezeigt.

4.2 Verkaufs-Szenario

Zur Veranschaulichung der Erläuterungen in diesem und im nächsten Kapitel betrachten wir ein elektronisches Verkaufs-Szenario. Wir beginnen mit der groben Darstellung in Abbildung 8, die wir in der Folge schrittweise verfeinern wollen.

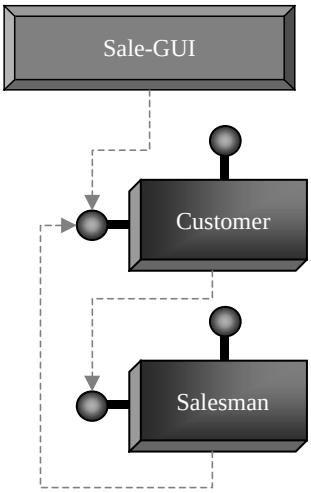

Abbildung 8: Verkaufs-Szenario (Schritt 1)

Ein Benutzer interagiert über ein GUI mit einer Customer-Komponente. Die Schnittstelle des Customer bietet dazu entsprechende Eigenschaften an wie z.B. **Product** oder **Price**, die vom Benutzer mit Werten belegt werden können. Wenn der Kauf getätigt werden soll, ruft der Benutzer die Methode **Buy()** des Customer auf. Dieser besitzt eine Referenz auf die Salesman-Komponente und ruft dort die Methode **Sell()** auf. Der Salesman besitzt umgekehrt ebenfalls eine Referenz auf den Customer. Während des Verkaufs kann er darüber die einzelnen Parameter (**Product**, **Price**, ...) erfragen.

Um eine instanziierte Komponente aktualisieren zu können, dürfen keine direkten Referenzen auf die Instanzen bestehen. Ansonsten wären diese Referenzen nach der Aktualisierung nicht mehr gültig. Indirekte Referenzierung kann in DCOM über Moniker realisiert werden (siehe Kapitel 3.4). Zur Unterstützung des Live-Update wurde ein spezieller *ProxyMoniker* implementiert. Es wird vorausgesetzt, daß alle Referenzen auf Instanzen von Komponenten, die aktualisierbar sein sollen, über den ProxyMoniker angefordert werden. Dieser tritt als Proxy für das eigentliche Objekt auf und stellt eine indirekte Referenz zur Verfügung. Eine verfeinerte Darstellung des Szenarios ist in Abbildung 9 gezeigt.

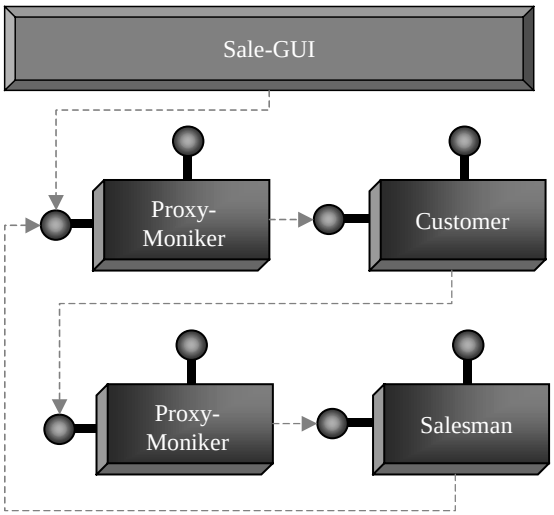

Abbildung 9: Verkaufs-Szenario (Schritt 2)

Der eigentliche Aktualisierungsvorgang wird durch eine zentrale Komponente gesteuert, den *UpdateManager*. Alle ProxyMoniker-Instanzen registrieren sich beim UpdateManager für das Update-Ereignis. Wird dieses Ereignis ausgelöst, informiert der UpdateManager alle ProxyMoniker-Instanzen, die ein Objekt referenzieren, das aktualisiert werden soll. Das Update-Ereignis kann beim UpdateManager z.B. per GUI durch einen Administrator ausgelöst werden. In Abbildung 10 ist nun das vollständige Szenario dargestellt.

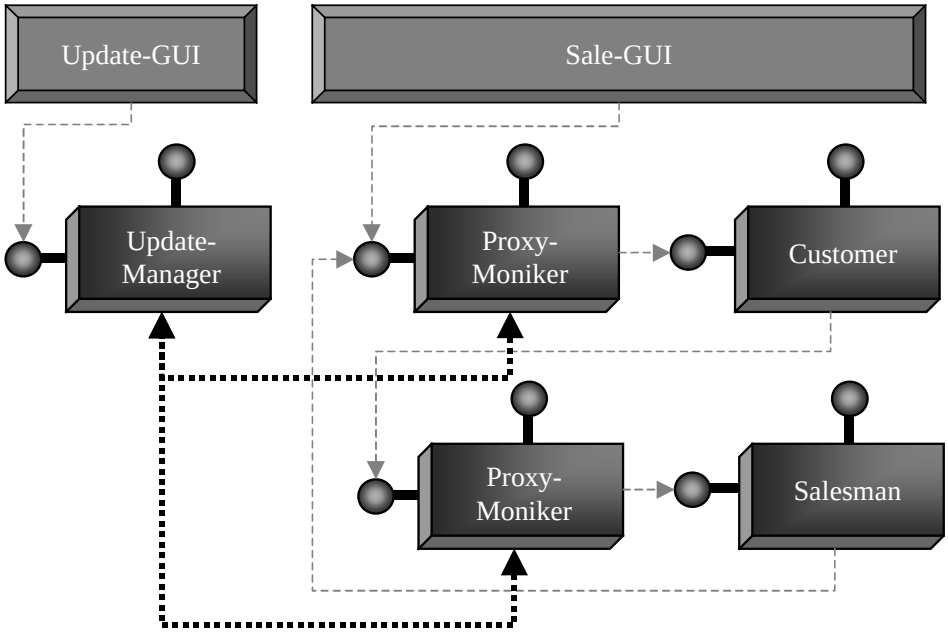

Abbildung 10: Verkaufs-Szenario (Schritt 3)

Die Verbindungen zwischen ProxyMoniker und UpdateManager sind nur schematisch gezeichnet. Diese beiden Komponenten sowie ihr Zusammenspiel sollen im nun folgenden Abschnitt genauer untersucht werden.

4.3 ProxyMoniker und UpdateManager

ProxyMoniker und UpdateManager spielen in der entwickelten Architektur die zentrale Rolle. Abbildung 11 zeigt die beiden Komponenten mit den

zugehörigen Hilfsobjekten *Sink* und *Connection Point* sowie den zur Verfügung gestellten Schnittstellen.

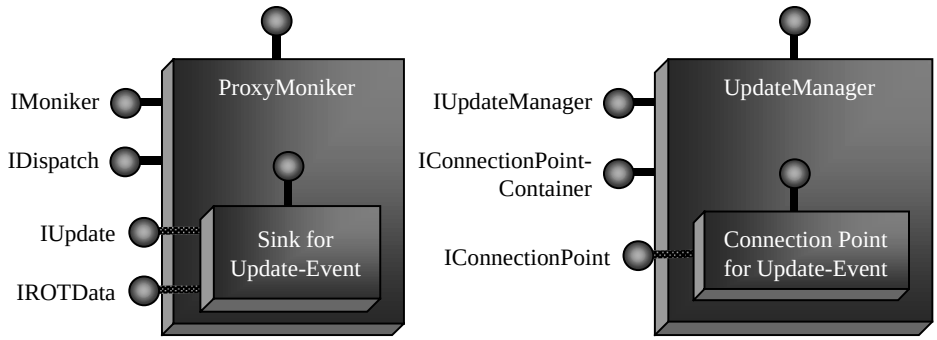

Abbildung 11: ProxyMoniker und UpdateManager

Über die Standard-Schnittstelle **IMoniker** stellt der ProxyMoniker Methoden zum Binden an eine neue Objekt-Instanz zur Verfügung (wie in Kapitel 3.4 beschrieben). Über **IDispatch** nimmt er Methoden-Aufrufe als Proxy entgegen und leitet diese an das gebundene Objekt weiter. Dafür muß das Objekt dynamische Bindung unterstützen; siehe dazu Kapitel 3.3. Die Schnittstellen **IUpdate** und **IROTData** werden für den Aktualisierungsvorgang benötigt. Ihre Verwendung wird im nächsten Abschnitt erläutert.

Der UpdateManager stellt seine Funktionalität über die Schnittstelle **IUpdateManager** zur Verfügung (siehe nächster Abschnitt). Über **IConnectionPointContainer** erhält der ProxyMoniker Zugang zum Connection Point. Bei diesem kann er sich über die Standard-Schnittstelle **IConnectionPoint** für das Update-Ereignis registrieren. Details zur Verwendung von Connection Points sind in Kapitel 3.6 erläutert.

4.4 Live-Update

Betrachten wir nun den Vorgang der Aktualisierung im Detail. Wir beziehen uns dabei auf das Beispiel des Verkaufs-Szenarios in Abbildung 10. Nehmen wir an, ein Benutzer (Administrator) möchte die Komponente „Customer“ durch eine Komponente „New Customer“ ersetzen. Die

Class-ID von „Customer“ sei clsid_v1, die Class-ID von „New Customer“ sei clsid_v2. Um den Vorgang zu initiieren, ruft der Benutzer (über ein GUI) die Methode **Update()** mit den Parametern clsid_v1 und clsid_v2 auf. Diese Methode gehört zur Schnittstelle **IUpdateManager** des UpdateManager. Dadurch wird das Update-Ereignis ausgelöst.

Dieses Ereignis muß der UpdateManager nun an alle ProxyMoniker propagieren, die eine Instanz der Customer-Komponente referenzieren. Die ProxyMoniker haben sich, wie oben geschildert, beim UpdateManager für das Update-Ereignis registriert. Um festzustellen, welche Komponente durch einen ProxyMoniker referenziert wird, ruft der UpdateManager die Methode **GetComparisonData()** der Schnittstelle **IROTData** des ProxyMoniker auf. Sie liefert die Class-ID des gebundenen Objektes zurück. In unserem Beispiel sind alle ProxyMoniker betroffen, die ein Objekt mit der Class-ID clsid_v1 referenzieren.

*Eine Randbemerkung zu **IROTData**: Diese Standard-Schnittstelle wird normalerweise von Monikern implementiert, die ihre Objekte in der „Running Object Table“ (ROT) registrieren. Die ROT ist eine globale System-Tabelle (innerhalb eines Rechners), die instanziierte Objekte enthält, welche durch einen Moniker referenziert werden. Wir verzichten hier auf die Verwendung der ROT, da sie auf Objekte beschränkt ist, die in einem eigenen Prozeß laufen. Da der UpdateManager einen ähnlichen Mechanismus implementiert, bietet sich jedoch die Verwendung von **IROTData** an.*

Jeder betroffene ProxyMoniker, erhält über die Schnittstelle **IUpdate** durch den UpdateManager nun die Nachricht **Update()** mit dem Parameter clsid_v2. Darauf führt der jeweilige ProxyMoniker den eigentlichen Aktualisierungsvorgang in folgenden Schritten durch:

- **Schritt 1**: Einfrieren von Methoden-Aufrufen, d.h. während der Aktualisierung werden keine Aufrufe an das gebundene Objekt weitergeleitet.

- **Schritt 2**: Erzeugen einer neuen Instanz der Komponente mit der Class-ID clsid_v2.
- **Schritt 3**: Speichern des Zustands der alten Instanz in einem Structured Storage (siehe Kapitel 3.5).
- **Schritt 4**: Initialisierung der neuen Instanz mit dem gespeicherten Zustand.
- **Schritt 5**: Freigeben der alten Instanz.
- **Schritt 6**: Freigeben von Methoden-Aufrufen. Alle weiteren Aufrufe werden an die neue Instanz weitergeleitet.

Im nächsten Kapitel soll nun die Implementierung der dargelegten Architektur aufgezeigt werden. Zur Veranschaulichung werden wir wieder auf das in diesem Kapitel vorgestellt Verkaufs-Szenario zurückgreifen.

5 Implementierung

5.1 Einführung

In diesem Kapitel soll die Implementierung der in Kapitel 4 dargestellten Architektur beschrieben werden. Dazu wollen wir uns zunächst die Rollen der wichtigsten Komponenten ins Gedächtnis rufen:

Der *ProxyMoniker* sorgt für die indirekte Referenzierung von Komponenten, welche uns einen Austausch einzelner Komponenten erlaubt, ohne daß die mit ihnen kooperierenden Komponenten davon betroffen sind.

Der *UpdateManager* verwaltet eine Liste aller per ProxyMoniker instanziierten Komponenten. Dies ermöglicht dem Administrator, zu einem beliebigen Zeitpunkt die Aktualisierung aller laufenden Instanzen zu veranlassen.

Beide Komponenten wurden mit *Visual C++ 5.0* implementiert. Diese Entwicklungsumgebung ist Teil des *Visual Studio* von Microsoft. Eine umfassende Einführung in die Sprache, die Umgebung sowie die zur Verfügung stehenden Klassenbibliotheken gibt [Fleischhauer 97]; allgemeine Konzepte der Win32-Programmierung erläutert [McGregor 96].

5.2 ProxyMoniker

Der ProxyMoniker hat mehrere Aufgaben zu erfüllen:

Zunächst einmal ist der ProxyMoniker eine *Komponente* und muß somit den in DCOM üblichen Bestimmungen genügen (Schnittstelle **IUnknown**). Weiterhin muß er in seiner Funktion als *Moniker* die Bindung zu einer Objekt-Instanz ermöglichen (Schnittstelle **IMoniker**). Im Anschluß daran muß er als *Proxy* des gebundenen Objektes agieren und Methoden-Aufrufe an das Objekt durchreichen (Schnittstelle **IDispatch**). Schließlich muß er die Funktionalität für den *Update*-Mechanismus bereitstellen (Schnittstellen **IROTData** und **IUpdate**).

Eine Komponente implementiert Schnittstellen, indem sie von der jeweiligen abstrakten Stub-Klasse erbt. Abbildung 12 zeigt ein UML-Diagramm des ProxyMoniker, der durch die Klasse **CProxyMoniker** (und die Hilfsklasse **CSink**) implementiert wird. Es sind dabei nur die hier interessierenden öffentlichen Methoden und Eigenschaften dargestellt. Zur Definition der UML (Unified Modeling Language) siehe [Booch 99].

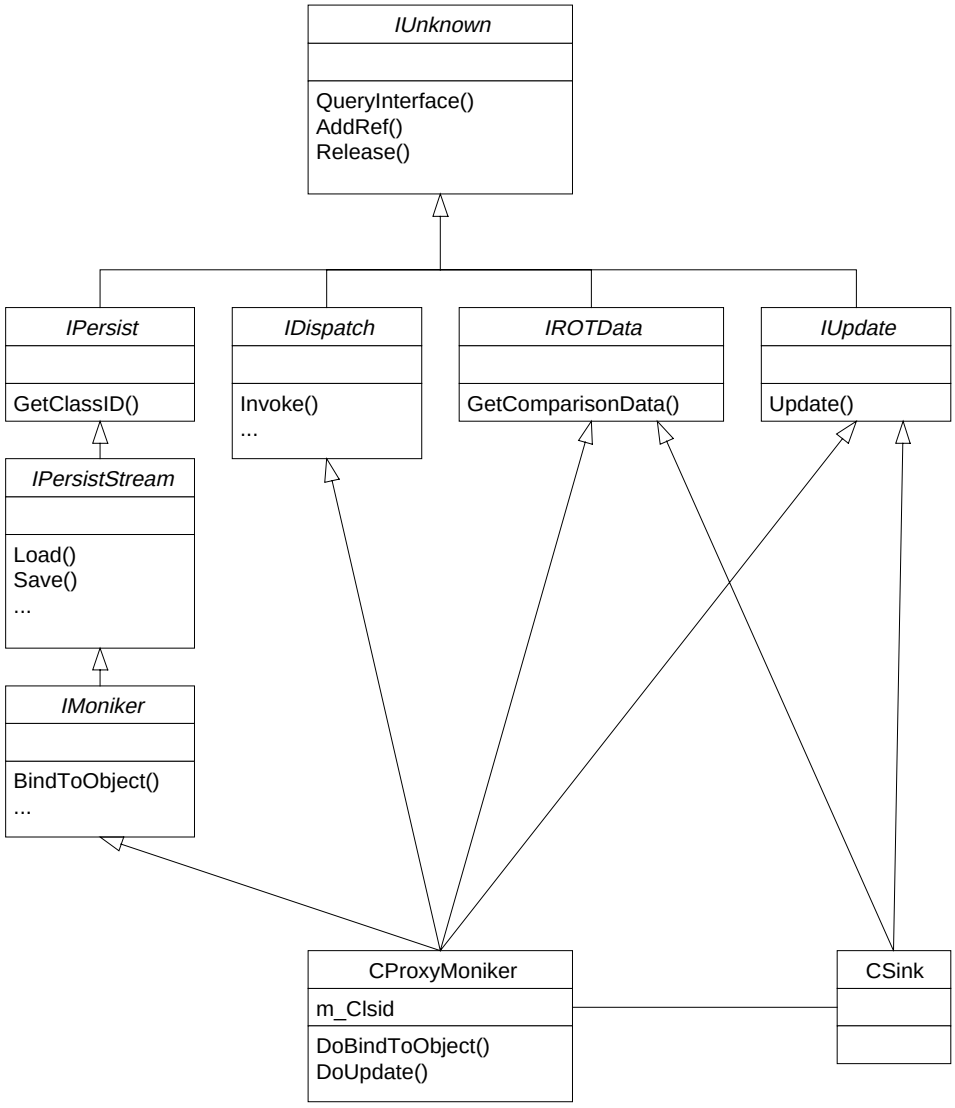

Abbildung 12: Der ProxyMoniker als UML-Diagramm

Hier die Deklaration der Klasse **CProxyMoniker** im Sourcecode:

```
class CProxyMoniker : public IMoniker, public IDispatch,
  public IROTData, public IUpdate
{
    // IUnknown
    HRESULT QueryInterface( REFIID riid, void** ppv );
    ULONG AddRef();
    ULONG Release();

    // IPersist
    HRESULT GetClassID( CLSID* pClassID );

    // IPersistStream
    HRESULT Load( IStream* pStm );
    HRESULT Save( IStream* pStm, BOOL fClearDirty );

    // IMoniker
    HRESULT BindToObject( IBindCtx *pbc, IMoniker *pmkToLeft,
      REFIID riidResult, void **ppvResult );

    // IDispatch
    HRESULT Invoke( DISPID dispId, REFIID riid,
      LCID lcid, WORD wFlags, DISPPARAMS FAR* pDispParams,
      VARIANT FAR* pVarResult, EXCEPINFO FAR* pExcepInfo,
      unsigned int FAR* pArgErr );

    // IROTData
    HRESULT GetComparisonData( unsigned char *pData,
      unsigned long cbMax, unsigned long* pcbData );

    // IUpdate
    HRESULT Update( BSTR NewClsid );

    // Helper functions
    bool DoBindToObject( IUnknown* pNewObject );
    void DoUpdate();

    CLSID m_Clsid;    // Class-ID of bound object
};
```

Wir beschreiben nun die Implementierung des ProxyMoniker gegliedert nach den vier genannten Funktionalitäten:

- Komponenten-Funktionalität (Schnittstelle **IUnknown**)
- Moniker-Funktionalität (Schnittstelle **IMoniker**)
- Proxy-Funktionalität (Schnittstelle **IDispatch**)
- Update-Funktionalität (Schnittstellen **IROTData** und **IUpdate**)

5.2.1 Komponenten-Funktionalität (IUnknown)

Jede DCOM-Komponente muß die Standard-Schnittstelle **IUnknown** implementieren. Diese realisiert zwei Grundfunktionalitäten:

- Über **QueryInterface()** erhält ein Client Zugang zu einer beliebigen Schnittstelle des ProxyMoniker.
- **AddRef()** und **Release()** dienen der Referenzzählung. In einer internen Variable wird gespeichert, wie viele Referenzen zu dieser Instanz bestehen. Diese Variable wird mit **AddRef()** inkrementiert und mit **Release()** dekrementiert. Wenn die Variable auf Null sinkt, wird die Instanz zerstört.

Es soll an dieser Stelle kurz erwähnt werden, daß die Methode der Referenzzählung gewisse Probleme mit sich bringt. Auf diese Probleme werden wir in Kapitel 5.4.2 zu sprechen kommen.

5.2.2 Moniker-Funktionalität (IMoniker)

Ein Moniker zeichnet sich durch zwei Dinge aus (siehe Kapitel 3.4):

- Er implementiert die Standard-Schnittstelle **IMoniker**.
- Das zugehörige Klassenobjekt implementiert die Standard-Schnittstelle **IParseDisplayName**.

Wir schildern zunächst die Implementierung der Schnittstelle **IMoniker** und gehen anschließend kurz auf die Implementierung von **IParseDisplayName** durch das Klassenobjekt ein.

IMoniker erbt von **IPersistStream**, welche wiederum von **IPersist** erbt. Die einzige Methode von **IPersist** ist **GetClassID()**. Sämtliche Standard-Schnittstellen, die dazu dienen, ein gebundenes Objekt persistent zu machen, erben von **IPersist** (siehe Kapitel 3.5.2). Die Methode **GetClassID()** stellt nun sicher, daß jede Standard-Schnittstelle, über die ein Objektzustand abgefragt werden kann, auch den Typ des Objekts angeben kann (in Form der Class-ID).

Auf die Implementation der Methoden **Load()** und **Save()** der Schnittstelle **IPersistStream** durch den ProxyMoniker kommen wir in Kapitel 5.4.3 zurück.

Die für uns wichtigste Methode von **IMoniker** ist **BindToObject()**:

```
HRESULT CProxyMoniker::BindToObject( IBindCtx *pbc,
  IMoniker *pmkToLeft, REFIID riidResult, void **ppvResult )
{
    // Get class-object
    IClassFactory* pClassFactory;
    CoGetClassObject( m_Clsid, CLSCTX_SERVER, NULL,
      IID_IClassFactory, (void**)&pClassFactory );

    // Create the object
    IUnknown* pUnknown;
    pClassFactory->CreateInstance( NULL, IID_IUnknown,
      (void**)&pUnknown );
    pClassFactory->Release();

    // Do the ProxyMoniker-specific work
    bool ok = DoBindToObject( pUnknown );
    pUnknown->Release();
    if( !ok ) return S_FALSE;

    // Return a pointer to my (!) IUnknown
    *ppvResult = reinterpret_cast<IUnknown*>(this);
    AddRef();
    return S_OK;
}
```

In dem gezeigten Auszug wurden einige Befehle zur Fehlerbehandlung der besseren Übersichtlichkeit wegen weggelassen. Die Routine zeigt das typische Vorgehen eines Moniker, der ein neues Objekt erzeugt:
Zunächst wird mit **CoGetClassObject()** eine Instanz des Klassenobjektes erzeugt, welches für Objekte mit der Class-ID **m_Clsid** zuständig ist. Die Variable **m_Clsid** wird wiederum durch das Klassenobjekt des Moniker initialisiert (s.u.) und gibt die Class-ID des Objektes an, das durch diese Instanz des ProxyMoniker referenziert werden soll. Als Rückgabewert liefert **CoGetClassObject()** im letzten Parameter einen Zeiger auf die Standard-Schnittstelle **IClassFactory** des Klassenobjektes.
In der Methode **CreateInstance()** von **IClassFactory** wird nun ein neues Objekt erzeugt und ein Zeiger auf die Schnittstelle **IUnknown** des Objektes zurückgegeben. Damit hat das Klassenobjekt seine Aufgabe erfüllt und wird wieder freigegeben.
Damit der ProxyMoniker seine übrigen Funktionalitäten erfüllen kann, müssen beim Binden an ein neues Objekt noch einige Verwaltungsarbeiten durchgeführt werden. Diese sind in die Methode **DoBindToObject()** ausgelagert. Die wesentlichen Schritte darin sind:

- Speichern eines Zeigers auf die Schnittstelle **IDispatch** des gebundenen Objektes in einer internen Variable (m_pDispatch). Diese dient dazu, Aufrufe an das gebundene Objekt weiterzureichen (siehe Kapitel 5.2.3).
- Erzeugen einer Instanz der Klasse **CSink** und Registrierung dieses Objektes beim UpdateManager. Dadurch wird der ProxyMoniker vom Update-Ereignis benachrichtigt (siehe Kapitel 5.2.4).
- Eintragen der Objekt- und ProxyMoniker-Instanzen in eine globale Tabelle. Diese Tabelle dient dem Klassenobjekt des Moniker dazu, für eine angegebene Objekt-Instanz die zugehörige ProxyMoniker-Instanz zu finden. Dies ist nötig, damit Objekte Referenzen auf sich selbst erhalten können (siehe Kapitel 5.4.1).

Nach dem Aufruf von **DoBindToObject()** wird der lokale Zeiger auf das Objekt wieder freigegeben.

Am Ende von **BindToObject()** wird ein Zeiger auf die Schnittstelle **IUnknown** des ProxyMoniker zurückgeliefert. (Die **reinterpret_cast**-Operation ist nötig, weil der ProxyMoniker mehrere Schnittstellen implementiert; siehe dazu Kapitel 3.2.2.) Normalerweise geben Moniker einen direkten Zeiger auf das gebundene Objekt zurück. Durch die Rückgabe eines Zeigers auf sich selbst wird es jedoch möglich, daß der ProxyMoniker als Stellvertreter (*Proxy*) für das gebundene Objekt agiert. Dieser Funktionalität widmen wir uns im nächsten Abschnitt.

Wie oben angegeben, muß das für einen Moniker zuständige Klassenobjekt die Standard-Schnittstelle **IParseDisplayName** implementieren. Diese enthält eine einzige Methode namens **ParseDisplayName()**, welche folgende Aufgaben erfüllt:

- Erzeugen einer neuen Instanz des ProxyMoniker
- Initialisieren der Variable **m_Clsid** mit der Class-ID des Objektes, das durch diese Instanz des ProxyMoniker referenziert werden soll. Die Class-ID wird im Display Name als Parameter der Methode **ParseDisplayName()** übergeben (siehe Kapitel 3.4). Der Display Name hat den Aufbau: „proxy:<Class-ID>”.
- Rückgabe eines Zeigers auf die Schnittstelle **IMoniker** der erzeugten ProxyMoniker-Instanz.

Zusätzliche Aufgaben von **ParseDisplayName()** werden im Kapitel 5.4.1 erläutert.

5.2.3 Proxy-Funktionalität (IDispatch)

In seiner Funktion als Proxy leitet der ProxyMoniker Aufrufe von **Invoke()** an das gebundene Objekt weiter. Dabei muß verhindert werden, daß während einer laufenden Aktualisierung Aufrufe an das Objekt

weitergeleitet werden. Dies wird durch die Verwendung von Semaphoren realisiert.

Wir betrachten nun die Implementierung von **Invoke()**, welche in der Folge erläutert wird:

```
HRESULT CProxyMoniker::Invoke( DISPID dispIdMember,
  REFIID riid, LCID lcid, WORD wFlags,
  DISPPARAMS FAR* pDispParams, VARIANT FAR* pVarResult,
  EXCEPINFO FAR* pExcepInfo, unsigned int FAR* pArgErr )
{
    // Increase invocation counter
    WaitForSingleObject( m_hSemaphore1, INFINITE );
    m_iInvocations++;
    ReleaseSemaphore( m_hSemaphore1, 1, NULL );

    // Invoke Method
    HRESULT hr = m_pDispatch->Invoke( dispId, riid, lcid,
      wFlags, pDispParams, pVarResult, pExcepInfo, pArgErr );

    // Decrease invocation counter
    WaitForSingleObject( m_hSemaphore2, INFINITE );
    m_iInvocations--;

    // If there was an update-request
    //   and there are no more current invocations,
    //   then update now
    if( m_bUpdateRequested && (m_iInvocations == 0) )
    {
        DoUpdate();
        m_bUpdateRequested = false;
        ReleaseSemaphore( m_hSemaphore1, 1, NULL );
    }
    ReleaseSemaphore( m_hSemaphore2, 1, NULL );

    // Return result
    return hr;
}
```

Der Ablauf der Methode **Invoke()** in einzelnen Schritten:

- Der Zähler **m_iInvocations** wird inkrementiert.
- Die Methode **Invoke()** des gebundenen Objektes wird aufgerufen; dafür wurde beim Binden an das Objekt ein Zeiger auf die Schnittstelle **IDispatch** des Objektes in der Variablen **m_pDispatch** gespeichert (siehe vorherigen Abschnitt).
- Der Zähler **m_iInvocations** wird dekrementiert. Falls in der Zwischenzeit ein Aktualisierungswunsch durch das Flag **m_bUpdateRequested** signalisiert wurde und das gebundene Objekt keine weiteren Aufrufe abarbeitet, wird die Aktualisierung durch den Aufruf der Hilfsmethode **DoUpdate()** nun vorgenommen.

Der Grund für diese Art der Implementierung von **Invoke()** wird in Kapitel 5.4.4 geschildert. Die Hilfsmethode **DoUpdate()** wird im nächsten Abschnitt erläutert.

Die Methode **Invoke()** verwendet insgesamt zwei Semaphor-Variablen:

- **m_hSemaphore1** wird gesetzt, wenn eine Aktualisierung vorgenommen werden soll. Sie verhindert, daß der Zähler **m_iInvocations** inkrementiert wird. Vorausgesetzt, daß jeder Aufruf nur endliche Zeit benötigt, sinkt **m_iInvocations** nach einiger Zeit auf Null.
- **m_hSemaphore2** wird gesetzt, wenn der Zähler **m_iInvocations** dekrementiert wird. Sinkt der Zähler auf Null, wird die Aktualisierung durchgeführt. Die Semaphor-Variable verhindert dabei, daß der Zähler geändert wird, während er in der Funktion **Update()** gelesen wird (siehe nächster Abschnitt).

Die Verwendung von Semaphoren ist nötig, um inkonsistente Zustände während einer Aktualisierung zu verhindern. Eine detaillierte Beschreibung des Konzeptes von Semaphoren und anderer Mechanismen zur Synchronisation nebenläufiger Prozesse gibt [Jessen 87].

5.2.4 Update-Funktionalität (IROTData + IUpdate)

Jede ProxyMoniker-Instanz registriert sich beim UpdateManager für das Update-Ereignis, sobald sie sich an eine bestimmt Objekt-Instanz bindet. Wird das Update-Ereignis beim UpdateManager ausgelöst, fragt dieser zunächst über die Methode **GetComparisonData()** der Schnittstelle **IROTData** ab, ob das Ereignis für die jeweilige ProxyMoniker-Instanz relevant ist (siehe Kapitel 5.3). Trifft dies zu, wird die Aktualisierung durch den Aufruf der Methode **Update()** der Schnittstelle **IUpdate** ausgelöst:

```
HRESULT CProxyMoniker::Update( BSTR NewClsid )
{
    WaitForSingleObject( m_hSemaphore1, INFINITE );
    WaitForSingleObject( m_hSemaphore2, INFINITE );

    // Store new Class-ID in m_NewClsid
    wcs2clsid( NewClsid, &m_NewClsid );

    // If there a no current invocations, update now
    // else request update
    if( m_iInvocations == 0 )
    {
        DoUpdate();
        ReleaseSemaphore( m_hSemaphore2, 1, NULL );
        ReleaseSemaphore( m_hSemaphore1, 1, NULL );
    }
    else
    {
        m_bUpdateRequested = true;
        ReleaseSemaphore( m_hSemaphore2, 1, NULL );
    }

    return S_OK;
}
```

Die Methode **Update()** speichert zunächst die neue Class-ID in der internen Variablen **m_NewClsid**. Der eigentliche Aktualisierungsvorgang ist in die

Hilfsmethode **DoUpdate()** ausgegliedert, in der **m_NewClsid** verwendet wird. Falls das gebundene Objekt derzeit keine Aufrufe abarbeitet, kann es sofort aktualisiert werden. Ansonsten wird das Flag **m_bUpdateRequested** gesetzt und die Aktualisierung verzögert, bis der letzte Aufruf abgearbeitet wurde. Die Verwendung der Semaphor-Variablen wurde bereits im vorherigen Abschnitt erläutert.
Wir zeigen nun die Implementierung des Aktualisierungsvorgangs durch die Methode **DoUpdate()**:

```
void CProxyMoniker::DoUpdate()
{
    // Create new class-object
    IClassFactory* pClassFactory;
    CoGetClassObject( m_NewClsid, CLSCTX_SERVER, NULL,
      IID_IClassFactory, (void**)&pClassFactory );

    // Store IUnknown-Pointer to new object in pNewUnknown
    IUnknown* pNewUnknown;
    pClassFactory->CreateInstance( NULL, IID_IUnknown,
      (void**)&pNewUnknown );
    pClassFactory->Release();

    // Store IDispatch-Pointer to new object in pNewDispatch
    IDispatch* pNewDispatch;
    pNewUnknown->QueryInterface( IID_IDispatch,
      (void**)&pNewDispatch );

    // If objects are stateful, transfer old state to new state
    IPersistStorage* pOldPersistStorage;
    m_pDispatch->QueryInterface( IID_IPersistStorage,
       (void**)&pOldPersistStorage );
    if( pOldPersistStorage != NULL )
    {
        IPersistStorage* pNewPersistStorage;
        pNewDispatch->QueryInterface( IID_IPersistStorage,
          (void**)&pNewPersistStorage );
        if( pNewPersistStorage != NULL )
```

```
        {
            // Create storage
            IStorage* pStorage;
            StgCreateDocfile( NULL, STGM_DIRECT | STGM_READWRITE |
              STGM_SHARE_EXCLUSIVE, 0, &pStorage );

            // Save state from old object to storage
            pOldPersistStorage->Save( pStorage, false );

            // Load state from storage to new object
            pNewPersistStorage->Load( pStorage );

            // Release storage
            pStorage->Release();

            pNewPersistStorage->Release();
        }
        pOldPersistStorage->Release();
    }

    // Release old object
    m_pDispatch->Release();
    g_pObject[m_iProxyTableIndex]->Release();

    // Switch to new object
    m_Clsid = m_NewClsid;
    m_pDispatch = pNewDispatch;
    g_pObject[m_iProxyTableIndex] = pNewUnknown;
}
```

Der Aktualisierungsvorgang wurde bereits in Kapitel 4.4 (Live-Update) erläutert. Die Methode **DoUpdate()** implementiert die dort genannten Schritte 2 bis 5:

- Erzeugen einer Instanz mit der Class-ID **m_NewClsid**.
- Speichern des Zustands der alten Instanz in einem Structured Storage durch die Methode **Save()** der Standard-Schnittstelle **IPersistStorage** (siehe Kapitel 3.5).

- Initialisierung der neuen Instanz mit dem gespeicherten Zustand durch die Methode **Load()** von **IPersistStorage**.
- Freigeben der alten und Wechsel zur neuen Instanz.

Die Schritte 1 und 6 (Einfrieren und Freigeben von Methodenaufrufen) werden durch die Verwendung von Semaphor-Variablen in den Methoden **Invoke()** und **Update()** implementiert, wie bereits weiter oben beschrieben.
Zu erwähnen bleibt, daß der UpdateManager die Schnittstellen **IROTData** und **IUpdate** des ProxyMoniker nicht direkt verwendet, sondern Aufrufe an das bei ihm registrierte Sink-Objekt sendet. Dieses leitet die Befehle an den ProxyMoniker weiter.

5.3 UpdateManager

Der UpdateManager realisiert die zentrale Verwaltung der einzelnen ProxyMoniker-Instanzen sowie eine Schnittstelle zum Benutzer oder zu einer anderen Software wie z.B. einem Versions-Management-Tool (siehe Kapitel 6.2). Seine Implementierung ist jedoch vergleichsweise einfach. Abbildung 13 zeigt die Darstellung als UML-Diagramm:

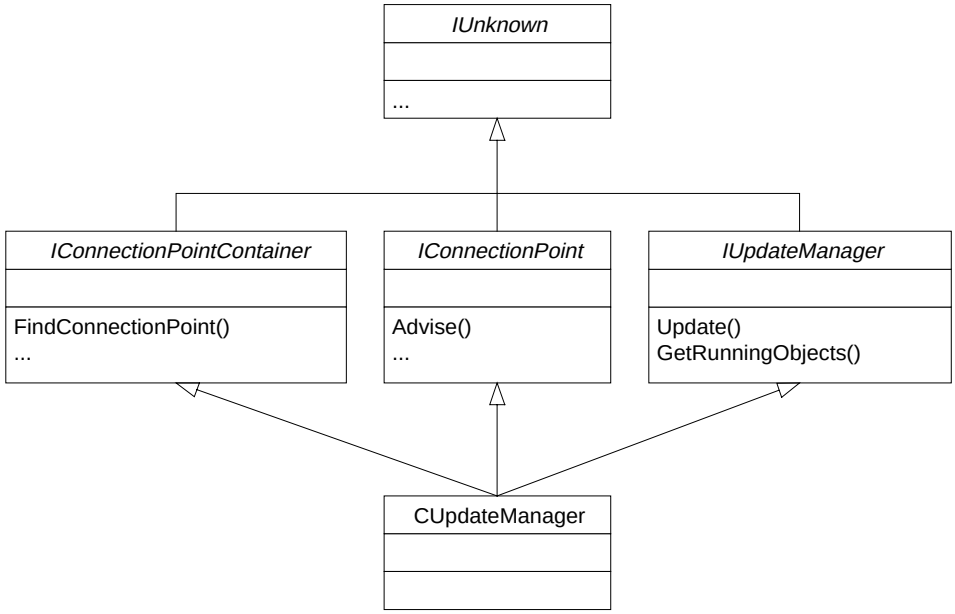

Abbildung 13: Der UpdateManager als UML-Diagramm

Wie in der Abbildung zu erkennen, implementiert der UpdateManager in seiner Eigenschaft als *Connectable Object* die Standard-Schnittstelle **IConnectionPointContainer** (siehe Kapitel 3.6). Normalerweise verwenden Connectable Objects für jeden Connection Point (CP) eine eigene Hilfsklasse, die den jeweiligen CP implementiert. Da der UpdateManager aber nur genau einen CP verwendet (namentlich den CP für das Update-Ereignis), kann er diesen auch selbst implementieren.
Wir erläutern nun die Implementierung der für uns relevanten Methoden des UpdateManager. Zunächst wieder die Deklaration im Sourcecode:

```
class CUpdateManager : public IConnectionPointContainer,
  public IConnectionPoint, public IUpdateManager
{
    // IUnknown
    ...

    // IConnectionPointContainer
    HRESULT FindConnectionPoint( REFIID riid,
      IConnectionPoint** ppCP);

    // IConnectionPoint
    HRESULT Advise( IUnknown* pUnknown, DWORD* pdwCookie );

    // IUpdateManager
    HRESULT Update( BSTR OldClsid, BSTR NewClsid, int* Instances );
    HRESULT GetRunningObjects( BSTR* ClsidList );
}
```

Die Methoden im einzelnen:

- **FindConnectionPoint()**: Wird ein CP für die Schnittstelle **IUpdate** angefragt, liefert diese Methode einen Zeiger auf die Schnittstelle **IConnectionPoint** des UpdateManager zurück.
- **Advise()**: Diese Methode trägt das übergebene Sink-Objekt in eine globale Tabelle des UpdateManager ein. Damit hat sich der Besitzer des Sink-Objektes für das Update-Ereignis registriert. Zurückgeliefert wird

der zugehörige Index in die globale Tabelle. Dieser kann später zur Deregistrierung verwendet werden.

- **Update()**: Mit dieser Methode können Instanzen von Komponenten zur Laufzeit aktualisiert werden. Der Parameter **OldClsid** gibt die Class-ID der zu aktualisierenden Komponente an, während **NewClsid** die Class-ID der neuen Komponente angibt. Die Methode durchläuft in einer Schleife alle registrierten ProxyMoniker-Instanzen: Über die Methode **GetComparisonData()** der Schnittstelle **IROTData** wird die Class-ID des gebundenen Objektes abgefragt. Entspricht dies der **OldClsid**, so wird die Methode **Update()** der Schnittstelle **IUpdate** des ProxyMoniker aufgerufen. Dabei wird die **NewClsid** als Parameter übergeben. Am Schluß wird die Anzahl der vorgenommenen Aktualisierungen im Parameter **Instances** zurückgeliefert.
- **GetRunningObjects()**: Die Methode fragt bei allen registrierten ProxyMoniker-Instanzen die Class-IDs ihrer gebundenen Objekte über die Methode **GetComparisonData()** der Schnittstelle **IROTData** ab. Diese werden dann verkettet und als Text mit Trennzeichen zurückgeliefert. Die Methode kann verwendet werden, um alle derzeit (über ProxyMoniker) instanziierten Komponenten anzuzeigen.

Zu dem UpdateManager wurde eine Typbibliothek namens „Updating Type Library“ erstellt. Durch das Einbinden dieser Typbibliothek kann er bequem von Visual Basic (VB) aus aufgerufen werden. Ein Beispiel:

```
Dim Manager As New UpdateManager
Dim Instances As Long, old_clsid As String, new_clsid As String
Rem ols_clsid = "..." : new_clsid = "..."
Instances = Manager.Update( old_clsid, new_clsid )
```

Zur Verwendung von Typbibliotheken siehe [Eddon 98]. Eine ausführliche Einführung in Visual Basic gibt [Kofler 96].

5.4 Probleme und Lösungen

Während der Entwicklung der dargestellten Lösung traten eine Reihe von Problemen auf. Diese DCOM-spezifischen Schwierigkeiten sowie die gewählten Lösungswege werden in der Folge beschrieben.

5.4.1 Problem der Referenzen auf sich selbst

Betrachten wir noch einmal das in Kapitel 4.2 vorgestellte Verkaufs-Szenario von Abbildung 10 (Seite 31). Über das „Sale-GUI“ wird (per ProxyMoniker) eine Instanz der Customer-Komponente erzeugt, die (wiederum per ProxyMoniker) eine Instanz der Salesman-Komponente erzeugt. Der Salesman soll nun wiederum Rückfragen an den Customer richten können; er benötigt also eine Referenz auf den Customer - und zwar wieder über einen ProxyMoniker, damit das Prinzip der losen Kopplung der Komponenten nicht verletzt wird.

Wie kommt nun der Salesman zu einer indirekten Referenz auf den Customer? Würde der Salesman beim ProxyMoniker eine Bindung zu einem Customer anfragen, würde stets eine neue Instanz des Customer erzeugt werden. Folglich muß der Customer dem Salesman eine indirekte Referenz auf sich selbst geben. Dazu muß der Customer eine Referenz auf den ProxyMoniker erhalten, der auf ihn zeigt. Dies ist jedoch in dem Moniker-Konzept von DCOM nicht vorgesehen.

Lösung: Verwendung des *Bind Context* (siehe Kapitel 3.4), welcher als Parameter von **ParseDisplayName()** an das Klassenobjekt des ProxyMoniker übergeben wird. Darin wird ein direkter Zeiger auf das aufrufende Objekt (in diesem Fall auf den Customer) übergeben. Das Klassenobjekt sucht dann in einer globalen Tabelle nach der dazugehörigen ProxyMoniker-Instanz. Wird diese gefunden, gibt das Klassenobjekt einen Zeiger auf die gefundene Instanz zurück; es wird in diesem Fall also *keine* neue ProxyMoniker-Instanz erzeugt.

5.4.2 Problem der zyklischen Referenzen

Das im vorherigen Abschnitt zitierte Beispiel von Abbildung 10 bringt noch ein weiteres Problem mit sich:
Wird das „Sale-GUI“ geschlossen, so gibt dieses die indirekte Referenz auf den Customer frei. Es besteht jedoch noch immer eine weitere indirekte Referenz, und zwar von Seiten des Salesman. Beide Objekte werden zwar nicht mehr benötigt, da das Programm, welches sie ursprünglich erzeugt hat, nicht mehr existiert; sie bleiben aber geladen, da sie gegenseitig auf sich verweisen und somit beide Referenzzähler größer als Null sind.
Lösung: Wenn der Customer die indirekte Referenz auf sich selbst an den Salesman übergibt, darf er ausnahmsweise seinen Referenzzähler nicht erhöhen! Im Gegenzug darf auch der Salesman beim Entladen kein **Release()** für die erhaltene Referenz aufrufen. Dadurch wird - zumindest aus Sicht der Referenzzähler - der Zyklus in den Verweisen unterbrochen. Da diese Lösung jedoch gegen ein fundamentales Prinzip von DCOM verstößt, ist sie mit Vorsicht zu genießen. Eine wesentlich bessere Lösung wäre ein durch die DCOM-Basisarchitektur bereitgestellter echter asynchroner Kommunikationsmechanismus. Diese etwas visionäre Möglichkeit werden wir im Ausblick kurz skizzieren (siehe Kapitel 6.2).
Probleme mit Threading-Modellen: VB unterstützt nur STA => Es kann nicht in **Update()** gewartet werden.

5.4.3 Problem der Persistenz von Referenzen

Während des Aktualisierungsvorgangs muß der Zustand der zu aktualisierenden Komponente persistent gemacht werden, damit er durch den Austausch nicht verloren geht. Teil des Zustands einer Komponente können auch Referenzen auf andere Komponenten sein. Handelt es sich dabei um indirekte Referenzen, so sieht das Moniker-Konzept bei DCOM vor, diese über die Schnittstelle **IPersistStream** des Monikers persistent zu machen. Der Moniker sorgt dann dafür, daß die referenzierte Komponente

persistent gemacht wird. Bei diesem Konzept tritt jedoch das bereits im vorherigen Abschnitt geschilderte Problem der zyklischen Referenzen auf: Komponente A soll persistent gemacht werden => Die Referenz auf Komponente B muß persistent gemacht werden => Komponente B muß persistent gemacht werden => Die Referenz auf Komponente A muß persistent gemacht werden => Komponente A muß persistent gemacht werden...

Lösung: Der ProxyMoniker gibt einen Aufruf von **Save()** nicht an das gebundene Objekt weiter, sondern sorgt durch Inkrementieren der Referenzzähler dafür, daß das gebundene Objekt (und die dazugehörige ProxyMoniker-Instanz) nicht entladen wird. Dabei schreibt **Save()** in den übergebenen Stream einen Index in die globale Tabelle des ProxyMoniker, in der Paare der Form [Objekt, ProxyMoniker] gespeichert sind.

Wird die neue Komponente geladen, erhält sie eine Referenz auf die ProxyMoniker-Instanz, indem sie den gespeicherten Index in einem *Display Name* der Form „proxy:[<Index>]“ an das Klassenobjekt des ProxyMoniker übergibt. Anschließend wird durch den Aufruf von **Load()** der Referenzzähler des gebundenen Objektes (und der ProxyMoniker-Instanz) wieder dekrementiert.

5.4.4 Problem der unterschiedlichen Threading-Modelle

Der ProxyMoniker ist ein In-Process Server, d.h. er läuft im selben Prozeß wie der Client, der ihn verwendet (siehe Kapitel 3.2.4). Da bei DCOM unterschiedliche Threading-Modelle existieren, kann es passieren, daß der Client nur das Apartment-Modell unterstützt (siehe Kapitel 3.7). Dies ist beispielsweise bei Verwendung des ProxyMoniker unter Visual Basic der Fall (*GetObject*-Befehl in VB). In diesem Szenario läuft der ProxyMoniker also innerhalb eines STA.

In der ursprünglichen Implementierung des ProxyMoniker war vorgesehen, daß er bei einem Aufruf von **Update()** innerhalb der Methode wartet, bis die

Anzahl der laufenden Invokationen auf Null gesunken ist. Dies funktioniert jedoch nicht innerhalb eines STA, da die Rückkehr von **Invoke()** erst erfolgen kann, wenn **Update()** wieder verlassen wurde.

Lösung: Gelöst wurde dieses Problem durch die bereits in Kapitel 5.2.4 geschilderte Verwendung des Flag **m_bUpdateRequested**.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das Thema „Dynamische Aktualisierung“ hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Mit zunehmender Größe der eingesetzten Softwaresysteme steigt der Bedarf, Teile davon mit der Zeit zu modifizieren. Ein jüngerer Ansatz zur Entwicklung von Softwaresystemen ist das Konzept komponentenbasierter Software. Komponentenbasierte Software ist für dynamische Aktualisierung prädestiniert, da Systeme bei diesem Konzept nicht aus einem monolithischen Modul bestehen, sondern aus einer Menge autarker Komponenten, die miteinander interagieren.

Es existieren eine Reihe unterschiedlicher Ansätze zur Implementierung von Aktualisierungsmechanismen. Zwei dieser Ansätze - die Demeter Methode von K. J. Lieberherr und die Active Channel Technology von Microsoft - wurden zu Beginn kurz vorgestellt. Im Anschluß wurde eine beispielhafte Implementierung dynamischer Aktualisierung für das Komponentenmodell von Microsoft, das Distributed Component Object Model, gezeigt. Dafür wurden zunächst die Grundlagen sowie einige Besonderheiten von DCOM erläutert. Im Hauptteil wurden Architektur und Implementierung der gewählten Lösung geschildert. Die Lösung beruht auf dem Konzept der dynamischen Bindung von Komponenten über *Moniker*. Dieses Form der losen Kopplung von Komponenten ermöglicht in Verbindung mit der Implementierung von Persistenz durch *Persistent Storage* einen Austausch einzelner Komponenten zur Laufzeit.

Um in der gewählten Implementierung dynamisch aktualisierbar zu sein, muß eine Komponente folgende Voraussetzungen erfüllen:

- Implementierung von dynamischer Bindung durch **IDispatch**
- Implementierung von Persistenz durch **IPersistStorage**
- Rückkehr von jedem Methodenaufruf nach endlicher Zeit

Die letzte Bedingung ist notwendig, da erst aktualisiert wird, wenn die Komponente keinen Methodenaufruf mehr bearbeitet. Durch diese Einschränkung wird sichergestellt, daß sich die Komponente vor der Aktualisierung in einem konsistenten Zustand befindet. Mit diesem Zustand wird dann die neue Komponente nach der Aktualisierung initialisiert.
Abschließend wurden einige der bei der Entwicklung aufgetretenen Probleme sowie die gewählten Lösungswege erläutert.

6.2 Ausblick

Der in dieser Arbeit entwickelte dynamische Aktualisierungsmechanismus kann in verschiedener Weise erweitert werden. Von besonderem Interesse ist der Einsatz in einer verteilten Umgebung. Durch die Aufteilung der Funktionalität auf die Komponenten ProxyMoniker und UpdateManager ist eine einfache Einsatzmöglichkeit in einer verteilten DCOM-Umgebung bereits geschaffen: Auf jeder Maschine muß dafür ein ProxyMoniker-Prozeß existieren, zuständig für alle referenzierten Objekte, die auf dem lokalen Rechner instanziiert worden sind. Außerdem muß der UpdateManager auf jeder Maschine als Remote Server registriert sein. Dabei wird die Adresse eines Rechners angegeben, der in diesem Szenario die Rolle eines zentralen Koordinators innehat. Auf diesem Rechner muß dann der UpdateManager-Prozeß laufen, über den Aktualisierungsvorgänge gestartet werden können.
Ein Problem bei diesem Szenario ist, daß der Koordinator in einem großen Netz schnell zum Flaschenhals werden kann, da er von jeder Instanziierung auf jeder Maschine informiert wird. Abhilfe könnte ein hierarchisches Domänenkonzept schaffen. Dabei wäre jeder UpdateManager-Prozeß nur für die Maschinen innerhalb seiner Domäne zuständig. Aktualisierungsvorgänge würden sich immer auf eine bestimmte Domäne beziehen. Der für diese Domäne zuständige UpdateManager müßte einen Auftrag dann an alle

UpdateManager darunterliegender Domänen propagieren. Erst auf der untersten Ebene würde der Auftrag an die ProxyMoniker propagiert werden.

Bei dem entwickelten Konzept kann pro Auftrag nur eine Komponente aktualisiert werden. Eine wünschenswerte Erweiterung wäre die Möglichkeit, eine Gruppe zusammenhängender Komponenten zu spezifizieren, die aktualisiert werden sollen. Auf diesem Weg könnte z.B. eine gesamte Anwendung durch eine neue Version ersetzt werden.

Eine weitere Ergänzungsmöglichkeit wäre die Implementierung transaktional gesicherter Aktualisierungen. Sollte während des Aktualisierungsvorgangs ein Fehler auftreten, könnte ein Rollback durchgeführt werden, bei dem der Zustand vor der Aktualisierung wiederhergestellt wird, d.h. die alte Komponente wird wieder instanziiert und mit dem gesicherten Zustand initialisiert.

Wünschenswert wäre es weiterhin, daß der Aktualisierungsmechanismus durch die Basisinfrastruktur von DCOM bereitgestellt wird. Da sämtliche Instanziierungen über die Basisinfrastruktur vorgenommen werden, ergibt sich hier das Potential für eine deutliche Verbesserung der Konsistenz und Performanz.

Eine etwas visionäre Modifikation von DCOM wäre die Ablösung des Client/Server-Modells durch einen asynchronen Kommunikations-mechanismus. Den Hauptvorteil bei diesem Ansatz sieht der Autor darin, daß der umständliche und fehleranfällige Mechanismus des Referenzzählens, welcher bei DCOM dem Programmierer aufgeladen wird, durch eine automatische Garbage-Collection ersetzt werden könnte.

Abschließend bleibt festzuhalten, daß der gezeigte dynamische Aktualisierungsmechanismus in ein Konfigurations-Tool integriert werden sollte, welches die semantische Korrektheit sowie Dauerhaftigkeit der Aktualisierung sicherstellt. Ein Beispiel einer solchen Integration in ein Versions-Management-Tool zeigt die verwandte Arbeit [Benthien 99].

A Literaturverzeichnis

[Benthien 99] T. Benthien: *Versionsbasiertes Komponentenmanagement auf Basis des Microsoft Component Object Model*. Diplomarbeit an der Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Verteilte Systeme, 1999.

[Bloom 83] T. Bloom: *Dynamic Module Replacement in a Distributed Programming System*. Ph.D. thesis at MIT, Laboratory for Computer Science, 1983.

[Booch 99] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Coulouris 94] G. Coulouris, J. Dollimore, T. Kindberg: *Distributed Systems - Concepts and Design*, Second Edition. Addison-Wesley, 1994.

[Eddon 98] G. Eddon, H. Eddon: *Inside Distributed COM*. Microsoft Press, 1998.

[Endler 93] M. Endler: *A Language for High-Level Programming of Dynamic Reconfiguration*. Oldenbourg Verlag, 1993.

[Fleischhauer 97] C. Fleischhauer: *Das große Buch zu Visual C++ 5*. Data Becker, 1997.

[Griffel 98] F. Griffel: *Componentware – Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag, 1998.

[Jessen 87] E. Jessen, R. Valk: *Rechensysteme - Grundlagen der Modellbildung*. Springer-Verlag, 1987.

[Josuttis 94] N. Josuttis: *Objektorientiertes Programmieren in C++*. Addison-Wesley, 1994.

[Kerner 93] H. Kerner: *Rechnernetze nach OSI*, 2. Auflage. Addison-Wesley, 1993.

[Kofler 96] M. Kofler: *Visual Basic 4.0 - Effiziente Programmentwicklung unter Windows 95*. Addison-Wesley, 1996.

[Kramer 90] J. Kramer: *Configuration Programming - A Framework for the Development of Distributable Systems*. Proceedings of the IEEE International Conference on Computing Systems and Software Engineering, 1990.

[Lieberherr 96] K. J. Lieberherr: *Adaptive Object-Oriented Software - The Demeter Method*. PWS Publishing Company, 1996.

[McGregor 96] R. W. McGregor: *Peter Norton's Guide to Windows95/NT4 Programming with MFC*. Sams Publishing, 1996.

[Mertens 92] P. Mertens, F. Bodendorf, W. König, A. Picot, M. Schumann: *Grundzüge der Wirtschaftsinformatik*, 2. Auflage. Springer-Verlag, 1992.

[Orfali 97] R. Orfali, D. Harkey: *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, 1997.

[Purtilo 91] J. M. Purtilo, C. R. Hofmeister: *Dynamic Reconfiguration of Distributed Programs*. Proceedings of the 11th International Conference on Distributed Computing Systems, 1991.

[Silberschatz 98] A. Silberschatz, P. B. Galvin: *Operating System Concepts*, Fifth Edition. Addison-Wesley, 1998.

[Veitch 96] A. C. Veitch, N. C. Hutchinson: *KEA - A Dynamically Extensible and Configurable Operating System Kernel*. Proceedings of the 3rd International Conference on Configurable Distributed Systems, 1996.

B Verzeichnis der Abbildungen

C Abkürzungsverzeichnis

API: Application Programming Interface; Sammlung von Routinen, die ein OS dem Programmierer zur Verfügung stellt.

C/S: Client-Server; bezeichnet ein Prinzip, nachdem bei der Kommunikation zwischen zwei Instanzen eine Instanz als Dienstnutzer (Client) und eine als Dienstanbieter (Server) auftritt; siehe Kapitel 3.2.4.

CDF: Channel Definition Format; XML-Anwendung von Microsoft zur Definition eines Active Channel; siehe Kapitel 2.3.

CLSID: Class-ID; die GUID, die einer DCOM-Klasse zugeordnet ist; siehe Kapitel 3.2.3.

COM: Component Object Model; Modell interagierender Komponenten; Vorläufer von DCOM.

CORBA: Common Object Request Broker Architecture; Spezifikation einer Architektur, die eine Kommunikation von Komponenten über eine ORB ermöglicht; Teil der OMA.

CP: Connection Point; Verbindungspunkt eines Connectable Object, bei dem sich andere Komponenten für ein bestimmtes Ereignis registrieren können; siehe Kapitel 3.6.

DCE: Distributed Computing Environment; Standard der OSF für verteilte Systeme; spezifiziert u.a. RPC.

DCOM: Distributed Component Object Model; siehe Kapitel 2.4.

DDE: Dynamic Data Exchange; erster IPC-Mechanismus zum Datenaustausch zwischen Anwendungen unter Windows; Vorläufer von OLE.

DispID: Dispatch-ID; ID in DCOM, die einer Methode zugeordnet ist und einen dynamischen Aufruf der Methode über das Interface IDispatch ermöglicht; siehe Kapitel 3.3.

DLL: Dynamic Link Library; binäre Datei unter Windows, die eine Sammlung von Funktionen enthält; wird bei Bedarf zur Laufzeit in den Hauptspeicher geladen.

GEREL: Generic Reconfiguration Language; Sprache zur Beschreibung von Rekonfigurationsvorgängen; siehe [Endler 93].

GUID: Globally Unique ID; eine 128 Bit lange Ganzzahl, die zur eindeutigen Kennzeichnung von Elementen bei DCOM verwendet wird; siehe Kapitel 3.2.3.

GUI: Graphical User Interface; graphisch-orientierte Benutzer-schnittstelle einer Applikation

HTML: Hypertext Markup Language; Beschreibungssprache für Internet-Seiten.

HTTP: Hypertext Transfer Protocol; Protokoll für den Transport von Daten im Internet.

ID: Identifier = Eindeutige Identifikationsnummer.

IDL: Interface Definition Language; Beschreibungssprache zur Definition von Schnittstellen.

IID: Interface-ID; die GUID, die einer Schnittstelle in DCOM zugeordnet ist; siehe Kapitel 3.2.3.

IPC: Interprocess Communication; eine Einrichtung des Betriebs-systems, welche Kommunikation zwischen Prozessen ermöglicht.

ISO: International Standards Organization; setzt sich zusammen aus Standardisierungsorganisationen aus ca. 90 Ländern.

LRPC: Lightweight RPC; abgewandelter RPC-Mechanismus auf einer einzigen Maschine; siehe Kapitel 3.2.4.

MFC: Microsoft Foundation Classes; Klassenbibliothek von Microsoft für die Entwicklung von Win32-Anwendungen; siehe [McGregor 96].

MIDL: Microsoft IDL; IDL für DCOM; siehe Kapitel 3.2.5.

MIT: Massachusetts Institute of Technology; Universität in Boston.

MTA: Multi-threaded Apartment; Apartment in COM, das mehrere Threads enthalten kann; siehe Kapitel 3.7.

OLE: Object Linking and Embedding; Verknüpfung von Objekten mit Verweis- oder Kopiersemantik; Vorläufer von COM.

OMA: Object Management Architecture; eine Architektur für den Einsatz objekt-orientierter Techniken in verteilten heterogenen Systemen; spezifiziert von der OMG.

OMG: Object Management Group; Herstellerkonsortium ohne Microsoft.

ORB: Object Request Broker; realisiert eine Kommunikations-plattform für verteilte Komponenten auf Basis des C/S-Modells; zentrale Instanz in CORBA.

OS: Operating System = Betriebssystem.

OSF: Open Software Foundation; Herstellerkonsortium.

OSI: Open Systems Interconnection; Satz von Standards für offene Netze, der von der ISO entwickelt wurde.

ROT: Running Object Table; globale Tabelle (innerhalb eines Rechners), die instanziierte Objekte enthält, welche durch einen Moniker referenziert werden.

ProgID: Programmatic ID; logische ID einer Komponente; nur eindeutig auf einem einzelnen Rechner; verweist auf eine Class-ID. Beispiel: Die ProgID „Word.Document.8“ verweist auf „{00020906-0000-0000-C000-000000000046}“.

RPC: Remote Procedure Call; Prozeduraufruf auf einem entfernten Server; siehe Kapitel 3.2.4.

STA: Single-threaded Apartment; Apartment in DCOM, das nur einen Thread enthalten kann; siehe Kapitel 3.7.

UML: Unified Modeling Language; graphische Sprache zur Spezifikation von Softwaresystemen; siehe [Booch 99].

URL: Unified Resource Locator; logische Adresse einer Ressource im Netzwerk.

UUID: Universally Unique ID; eindeutiger Bezeichner in DCE.

VB: Visual Basic; Programmiersprache von Microsoft, welche insbesondere für Adhoc-Lösungen geeignet ist; erlaubt die bequeme Verwendung von DCOM-Komponenten; siehe [Kofler 96].

Win32: 32-Bit API der Windows-Betriebssysteme.