Analyse von Middleware-Technologien bzgl. Performanz und ihrer Eignung für lose und eng gekoppelte verteilte Anwendungen

Dissertation
zur Erlangung des Grades
»Doktor der Naturwissenschaften«

am Fachbereich Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz

vorgelegt von

Karsten Walpert

geboren am 30.06.1958 in Wanne-Eickel

Mainz, im April 2005

Tag der mündlichen Prüfung: 26 Juli 2005

D77 (Dissertation an der Johannes Gutenberg-Universität Mainz)

Abstrakt

Die vorliegende Dissertation analysiert die Middleware-Technologien CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture), COM/DCOM (**C**omponent **O**bject **M**odel/**D**istributed **C**omponent **O**bject **M**odel), J2EE (**J**ava-**2**-**E**nterprise **E**dition) und Web Services (inklusive .NET) auf ihre Eignung bzgl. eng und lose gekoppelten verteilten Anwendungen. Zusätzlich werden primär für CORBA die dynamischen CORBA-Komponenten DII (**D**ynamic **I**nvocation **I**nterface), IFR (**I**nterface **R**epository) und die generischen Datentypen Any und DynAny (dynamisches Any) im Detail untersucht. Ziel ist es,

a. konkrete Aussagen über diese Komponenten zu erzielen, und festzustellen, in welchem Umfeld diese generischen Ansätze ihre Berechtigung finden.
b. das zeitliche Verhalten der dynamischen Komponenten bzgl. der Informationsgewinnung über die unbekannten Objekte zu analysieren.
c. das zeitliche Verhalten der dynamischen Komponenten bzgl. ihrer Kommunikation zu messen.
d. das zeitliche Verhalten bzgl. der Erzeugung von generischen Datentypen und das Einstellen von Daten zu messen und zu analysieren.
e. das zeitliche Verhalten bzgl. des Erstellens von unbekannten, d. h. nicht in IDL beschriebenen Datentypen zur Laufzeit zu messen und zu analysieren.
f. die Vorzüge/Nachteile der dynamischen Komponenten aufzuzeigen, ihre Einsatzgebiete zu definieren und mit anderen Technologien wie COM/DCOM, J2EE und den Web Services bzgl. ihrer Möglichkeiten zu vergleichen.
g. Aussagen bzgl. enger und loser Koppelung zu tätigen.

CORBA wird als standardisierte und vollständige Verteilungsplattform ausgewählt, um die o. a. Problemstellungen zu untersuchen. Bzgl. seines dynamischen Verhaltens, das zum Zeitpunkt dieser Ausarbeitung noch nicht oder nur unzureichend untersucht wurde, sind CORBA und die Web Services richtungsweisend bzgl.

a. Arbeiten mit unbekannten Objekten. Dies kann durchaus Implikationen bzgl. der Entwicklung intelligenter Softwareagenten haben.
b. der Integration von Legacy-Applikationen.
c. der Möglichkeiten im Zusammenhang mit B2B (Business-to-Business).

Diese Problemstellungen beinhalten auch allgemeine Fragen zum Marshalling/Unmarshalling von Daten und welche Aufwände hierfür notwendig sind, ebenso wie allgemeine Aussagen bzgl. der Echtzeitfähigkeit von CORBA-basierten, verteilten Anwendungen.

Die Ergebnisse werden anschließend auf andere Technologien wie COM/DCOM, J2EE und den Web Services, soweit es zulässig ist, übertragen. Die Vergleiche CORBA mit DCOM, CORBA mit J2EE und CORBA mit Web Services zeigen im Detail die Eignung dieser Technologien bzgl. loser und enger Koppelung.

Desweiteren werden aus den erzielten Resultaten allgemeine Konzepte bzgl. der Architektur und der Optimierung der Kommunikation abgeleitet. Diese Empfehlungen gelten uneingeschränkt für alle untersuchten Technologien im Zusammenhang mit verteilter Verarbeitung.

Abstract

This dissertation analyses some middleware-technologies like CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture), COM/DCOM (**C**omponent **O**bject **M**odel/**D**istributed **C**omponent **O**bject **M**odel), J2EE (**J**ava-**2**-**E**nterprise-**E**dition) and Web Services (inclusive .NET) for their ability and usage within loosely and closely coupled distributed applications. Additionally it analyses in detail primary for CORBA the dynamic CORBA-components DII (**D**ynamic **I**nvocation **I**nterface), IFR (**I**nter**f**ace **R**epository) and the generic data types Any and DynAny (dynamic Any). The goals are:

a. To make precise Statements on how and when these generic approaches can be used.
b. To make precise statements on the performance when interpreting the build-up of unknown objects.
c. To make precise statements on the performance according to the communication between (remote) objects.
d. To make precise statements on the time behaviour when creating such objects and to insert arbitrary data to it.
e. To make precise statements on creating unknown data structures, i. e. data structures that are not described in IDL and will be created at runtime.
f. To describe the advantages/disadvantages of dynamic components, to define the application area where these components can be suitable and to compare it with other technologies like COM/DCOM. J2EE and Web Services.
g. To make precise statements on the usage of such components in loosely and closely coupled distributed applications.

CORBA is used as a standardized and complete middleware technology to study and analyse the given issues. At the time of the preparation of this dissertation the dynamic behaviour of CORBA has not been deeply analysed. Regarding to their possibility of dynamic behaviour CORBA and Web Services are trend-setter according to:

a. Working with unknown objects. This can have some implications for the design and implementation of intelligent software-agents.
b. The integration of legacy-applications.
c. The possibility for business-2-business.

These issues include also the more general question of marshalling/unmarshalling of data and their complexity and how to achieve real-time-ability for CORBA-based distributed applications.

The results will be, if allowable, assigned to different technologies like COM/DCOM, J2EE and Web Services. The comparisons CORBA vs. DCOM, CORBA vs. J2EE and CORBA vs. Web Services will show in detail whether these technologies are applicable for loosely and closely coupled distributed applications, or not.

Last, but not least, the results will be used for some general concepts and architectures in accordance to the optimization of communication. The recommendations are valid for all distributed applications/systems regardless of their used technology.

Vorwort

In der Tat, die Strukturen der einzelnen Unternehmungen haben sich im Laufe der beiden letzten Jahrzehnte grundlegend gewandelt und werden auch zukünftig durch Um- oder Neustrukturierungen Veränderungen unterliegen. Für all diese Unternehmungen ist es von größter Wichtigkeit, dass die gegenwärtigen IT-Landschaften an diese veränderten Strukturen angepasst werden. Die betriebswirtschaftlichen Methoden von Just-in-time-Production, Lean Management und Lean Production sind große Chancen für die einzelnen Unternehmen, müssen aber auch verwaltet und organisiert werden. Ohne EDV-Unterstützung ist dies nur schwer vorstellbar. Die traditionelle Datenverarbeitung wird kaum geeignet sein, in einem dezentralisierten Umfeld zentral Informationen zu verwalten und allen Berechtigten zur Verfügung zu stellen.

Seit Ende der achtziger Jahre werden in der Informatik Konzepte zum Down- oder Rightsizing und einer Verteilung von Anwendungen erörtert. Die Grundlage dieser Methoden ist ein geeignetes Client/Server-Modell, das sich im Verlauf weiterer Forschungsarbeiten zu den heute üblichen Mehr-Schichten-Architekturen inklusive der benötigten Softwareinfrastruktur weiterentwickelt hat.

Auch aus technischer Sicht sind Client/Server-Modelle, ihre Softwareinfrastruktur (Middleware) und darauf basierende verteilte Systeme und Anwendungen äußerst interessante Gebilde. Eine Beschäftigung mit diesem Thema ist eine Beschäftigung mit Teilgebieten der Technischen, der Praktischen und der Theoretischen Informatik.

Verteilte Systeme sind grundsätzlich autonome Systeme, die durch ein Netzwerk miteinander verbunden sind. Die einzelnen Rechner müssen mit geeigneter Software ausgestattet werden, die es erlaubt, dass die an einem verteilten System beteiligten Partner Daten untereinander austauschen und sich ggf. synchronisieren können. Die Heterogenität bzgl. der Prozessorarchitekturen und der verwendeten Betriebssysteme muss hierbei vollständig verborgen sein. Ebenso bedingt die Rechnervielfalt u. a. ein breites Spektrum von Übertragungstechniken und -medien.

Auf jedem der im Netz beteiligten Rechner laufen Programme, die die Verteilung des Systems oder einer Anwendung realisieren und die für den Datenaustausch verantwortlich sind. Die Verwaltung dieser Programme fällt in den Zuständigkeitsbereich von Betriebssystemen, die i. d. R. prozessorientiert arbeiten.

Da die Rechner und mit ihnen die jeweiligen Prozesse unabhängig voneinander und simultan arbeiten, entstehen Koordinierungsprobleme, d. h. die beteiligten Prozesse müssen sich in bestimmten Situationen, wie z. B. bei schreibendem Zugriff auf einen bestimmten Teil eines gemeinsamen Speichers, untereinander darüber verständigen, in welcher Reihenfolge die jeweiligen Prozesse eine bestimmte Aktion durchführen. In anderen Situationen sollen Prozesse zusammenarbeiten, um die vorhandene Parallelität zu ihren Gunsten auszunutzen.

Trotz einer langjährigen Diskussion und den Vorteilen, die verteilte Systeme und Anwendungen mit sich bringen, ist die Anzahl der implementierten und eingesetzten Systeme vergleichsweise gering. Dies ist u. a. auf enorme Investitionskosten bzgl. existierender Großrechenzentren zurückzuführen. Das beinhaltet sowohl die Anschaffung von Großrechnern und weiterer Peripherie, als auch die Entwicklung benötigter Software. Zum anderen ist die Vielzahl der unterschiedlichen Standardisierungen wie DCE (**D**istributed **C**omputing **E**nvironment), COM/DCOM (**D**istributed **C**o**m**ponent **M**odel), CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture), J2EE (**J**ava **2** **E**nterprise **E**dition), .NET oder Web Services verwirrend und es ist nicht immer ersichtlich, wann welche Technologie erfolgreich einzusetzen ist.

Trotz mehr als zwanzigjähriger Entwicklung und Forschung im Bereich der verteilten Verarbeitung herrscht bei potentiellen Anwendern immer noch Unklarheit darüber, was verteilte Verarbeitung eigentlich ist, wo die tatsächlichen Vorteile, aber auch die Nachteile zu sehen sind. Erschwerend kommt hinzu, dass im Bereich der Forschung wichtige Themen wie z. B. das Arbeiten mit unbekannten Objekten/Diensten bisher nicht im Detail untersucht wurde.

Verglichen mit der traditionellen Datenverarbeitung auf monolithischen Großrechnersystemen betragen die Kosten für die Beschaffung und Wartung von Hard- und Software oft nur einen geringen Anteil dessen, was man für Time Sharing Systeme mit vergleichbarer Leistung aufwenden müsste. Ein Großteil der benötigten Funktionalität wird auf PCs und Workstations verlagert, um so eine Entlastung von Großrechnern zu bewirken und gleichzeitig kostengünstige Standardsoftware und einfach zu bedienende vollgraphische Benutzeroberflächen für den Endbenutzer zur Verfügung zu stellen.

Verbunden mit dem Kostenargument ist bei der Auswahl von Hard- und Software der Vorteil der Herstellerunabhängigkeit. Die Idee des Client/Server-Computings ist die logische Konsequenz der Standardisierungsbemühungen von offenen Systemen, wie sie z. B. von der X/Open-Gruppe in den achtziger und neunziger Jahren forciert wurden. Klar definierte, offen gelegte Schnittstellen sind eine Voraussetzung für die Portabilität und Interoperabilität von Software. Dem Kunden ist damit die Möglichkeit gegeben, die für seine Problemstellung günstigste Lösung zu beschaffen, ohne sich in eine enge Abhängigkeit eines Computerherstellers oder in Abhängigkeit zu einem bestimmten Betriebssystems zu begeben.

Ein weiterer entscheidender Vorteil von Client/Server-Systemen ist die inhärente Flexibilität. Die Struktur der Informationsverarbeitung wird direkt an die organisatorischen Strukturen einer Unternehmung angepasst. Bei einer Um- oder Neustrukturierung kann auch das zugrunde liegende Client/Server-System relativ leicht rekonfiguriert werden.

Von besonderer Bedeutung ist die Skalierbarkeit verteilter Systeme und verteilter Anwendungen, also das kontinuierliche, modulare Mitwachsen der Systeme bei steigenden Benutzerzahlen und steigenden Anforderungen an die Kapazität.

Das Client/Server-Computing weist auch Nachteile auf; einige sind lediglich temporärer Natur und werden schnell gelöst sein, andere werden noch intensive Forschungs- und Entwicklungsarbeit bedingen.

Die Komplexität der Informationsverarbeitung wird durch eine Verteilung von Anwendungen oder Systemen nicht geringer, sondern nimmt aus den folgenden Gründen zu:

a. *Heterogenität*, d. h. die Anwendung ist auf verschiedene Rechner mit unterschiedlicher Hardware und ggf. unterschiedlichen Betriebssystemen verteilt.
b. *Zusätzliche Fehlermöglichkeiten*, z. B. dadurch, dass die einzelnen Komponenten miteinander kommunizieren müssen.
c. *Echte Parallelität der Ereignisse*, die u. U. eine Absprache über eine bestimmte Verarbeitungsreihenfolge bedingen.
d. *Konsistenzprobleme beim Datenzugriff*, d. h. alle mehrfach vorhandenen Datenbestände müssen zu jedem Zeitpunkt identisch, also konsistent, sein.
e. *Autonomie der beteiligten Teilsysteme*.

Es herrscht allgemeiner Konsens darüber, dass es die Aufgabe der System- und Anwendungssoftware ist und sein muss, die Komplexität für den Benutzer transparent und beherrschbar zu machen.

Die Object Management Group hat in einem Zusammenschluss verschiedener Hersteller ein verteiltes Objektmodell standardisiert, das die o. a. Forderungen erfüllt. Neben einem statischen, also einen zur Laufzeit nicht veränderbaren Zugang zu den entfernten Objekten, wurde in der Spezifikation ein dynamischer Zugang zu den entfernten Objekten berücksichtigt.

Genau an dieser Stelle setzt die vorliegende Arbeit ein, und untersucht u. a. das zeitliche Verhalten bei verteilten Ansätzen unter Berücksichtigung verschiedener Vorgehensweisen und Architekturen. Als Referenzplattform dient die von der Object Management Group standardisierte Softwareinfrastruktur CORBA, die zusätzlich eine Entwicklungsumgebung für verteilte, objektorientierte Anwendungen darstellt.

Zu untersuchen sind hier Vorgehensweisen von statischen und dynamischen Operationenaufrufen, ihr Zeitverhalten und die benötigten Aufwände, um mit den benötigten statischen und/oder dynamischen Datentypen zu arbeiten. Ebenso von Interesse sind die Möglichkeiten der asynchronen Kommunikation und die Auswirkungen von dynamischen Ansätzen unter Berücksichtigung eines Gesamtsystems.

Die Ergebnisse der Ausarbeitung sind auf andere Technologien, die einen entfernten Methodenaufruf zulassen, oder die Möglichkeiten der asynchronen Kommunikation unterstützen, zu übertragen. Es ist somit unerlässlich, auch die Enterprise JavaBeans, DCOM, .NET und Web Services im Vergleich mit CORBA zu sehen und sie auf ihre Eignung bzgl. enger und loser Koppelung bei verteilten Anwendungen/Systemen zu untersuchen.

Ein Ziel muss sein, grundsätzliche Aussagen über den Aufbau verteilter Anwendungen zu tätigen, wobei die benötigten Aufwände und die Verarbeitungsgeschwindigkeit zu berücksichtigen ist. Von besonderem Interesse ist hierbei der dynamische Anteil der CORBA-Komponenten, die bisher nicht im Detail analysiert wurden, jedoch in vielerlei Hinsicht in Zukunft eine herausragende Rolle spielen werden. Im Zusammenhang mit EAI (Enterprise Application Integration), B2B (Business-to-Business) und auch im Zusammenhang mit intelligenten Softwareagenten werden die Ideen des dynamischen CORBA bahnbrechend sein.

Der Bereich des Grid-Computings ist eine Möglichkeit, in einem WAN dezentrale Ressourcen zu finden und zu nutzen. Da dies jedoch ausschließlich für Web Services von Interesse sein kann, speziell für B2B, wird dieser Bereich nicht untersucht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Definitionen

Verzeichnis der Beispiele

Verzeichnis der Programmbeispiele

Tabellenverzeichnis

Einleitung

Bereits zu Beginn der achtziger Jahre wurde der **R**emote **P**rocedure **C**all (RPC), also die Möglichkeit, eine entfernte Funktion aufzurufen, vorgestellt. Die damals vorherrschende Praxis, die Funktionen und alle zu übertragenden Daten durch eine geeignete Beschreibungssprache zu deklarieren und durch einen geeigneten Kompilierer in eine Zielsprache zu übersetzen, ist bis heute weitestgehend erhalten geblieben. Die Object Management Group standardisierte für verteilte, objektorientierte Anwendungen die Beschreibungssprache IDL, mit der es möglich ist, den Zugang zu entfernten Objekten deklarativ zu beschreiben. Zusätzlich bietet die Spezifikation die Möglichkeit dynamisch zu arbeiten. Dies beinhaltet asynchrone Methodenaufrufe ebenso wie die Möglichkeit, mit Objekten zu arbeiten, die zur Kompilierzeit noch nicht bekannt waren.

Die Idee zu dieser Ausarbeitung entstand in einem Projekt, bei dem eine CORBA-basierte verteilte Anwendung zu entwickeln war. Die Prämisse für dieses Projekt lautete, dass neben einem statischen Zugang in Form von fest vorgegebenen Schnittstellen auch ein dynamischer Zugang und das Arbeiten mit unbekannten Objekten, bzw. das Generieren von beliebigen Objekten zur Laufzeit möglich sein muss.

Zu dem Zeitpunkt waren die dynamischen CORBA-Komponenten bereits implementiert, aber eine Reihe von grundlegenden Fragen ist immer noch unbeantwortet:

1. Wie arbeitet man mit unbekannten Objekten und den von CORBA zur Verfügung gestellten Technologien?
2. Wie kann man zur Laufzeit neue Objekte erstellen, die nicht durch eine Beschreibungssprache im Vorfeld definiert wurden?
3. Welche Aufwände ergeben sich durch diesen generischen Ansatz?
4. Welches Laufzeitverhalten ist zu erwarten?
5. Unter welchen Bedingungen kann das dynamische CORBA sinnvoll eingesetzt werden?
6. Welchen Mehrwert kann das dynamische CORBA gegenüber einem statischen Zugang noch besitzen?
7. Sind die bahnbrechenden Ideen von CORBA, insbesondere dem dynamischen CORBA geeignet für lose Koppelungen, wie sie in den Bereichen B2B (Business-to-Business) oder bei intelligenten Softwareagenten benötigt werden?

Da diese Fragestellung nicht primär auf CORBA bezogen ist, wurde die Ausarbeitung dahingehend erweitert, dass

1. zusätzlich andere führende Technologien wie COM/DCOM, J2EE und Web Services im Detail untersucht wurden.
2. ihre Möglichkeiten bzgl. dynamischen Verhaltens mit dem dynamischen CORBA verglichen wurden.
3. die Fragestellungen auch im Hinblick mit diesen Technologien beantwortet werden.
4. allgemeine Empfehlungen bzgl. eng und lose gekoppelten Systemen/Anwendungen gegeben werden.

Eine wichtige Rolle spielt hierbei das verwendete Protokoll. Um die Frage nach optimaler Kommunikation zwischen einzelnen Komponenten beantworten zu können, wurde das Protokoll im Detail untersucht. Dadurch kann der Aufwand der Datenserialisierung, sowie der Aufwand, Daten in ein neutrales Format zu konvertieren, abgeschätzt werden.

Die vorliegende Dissertation unterteilt sich in die folgenden Kapitel:

Kapitel 1 beschreibt die Grundlagen, die für den weiteren Verlauf der vorliegenden Arbeit benötigt werden.

Kapitel 2 und 3 sind eine Einführung in die von der OMG standardisierten Softwareinfrastruktur CORBA. Von Interesse sind hierbei die Objektkommunikation und die wichtigsten standardisierten Dienste. Ebenfalls von hoher Relevanz ist die Beschreibungssprache IDL. Damit verbunden ist eine Reihe von Möglichkeiten:

1. Die benötigte Netzsoftware für das Arbeiten mit entfernten Objekten kann durch einen Kompilierer basierend auf der Beschreibung generiert werden.
2. Die zu übertragenen Daten und deren Serialisierung können ebenfalls durch einen geeigneten Kompilierer übernommen werden. Damit verbunden ist u. a.
   a. die Sprachunabhängigkeit. D. h. es ist irrelevant, in welcher Programmiersprache das entfernte Objekt oder der Klient implementiert sind.
   b. die Überdeckung der Heterogenität, die durch verschiedene Betriebssysteme auf unterschiedlicher Hardware bedingt ist.
   c. die Verwendbarkeit der Beschreibung durch Klienten, die erst zur Laufzeit und nicht zur Kompilierzeit Kenntnis über die entfernten Objekte erhalten.
3. Nach einigen Vorarbeiten dem Klienten eine Entwicklungsumgebung zu liefern, die weitestgehend mit dem lokalen Fall identisch ist.
4. Eine auf einer einfachen Sprache basierende, sehr gute Dokumentationsmöglichkeit zu besitzen, die mit geringem Aufwand in ein HTML-Dokument konvertiert werden kann.

Um den hohen Aufwand der Kommunikation zu unterstreichen, wird in Kapitel 4 zusätzlich eine Übersicht über die Serialisierung der zu übertragenden Datentypen beschrieben.

Kapitel 5 beschreibt die wichtigsten Objektdienste von CORBA. Dies beinhaltet den Namensdienst, den LifyCycle-Dienst, dessen Factory-Pattern später für die Enterprise JavaBeans benötigt wird, den Ereignis- und den Transaktionsdienst.

Kapitel 6 behandelt das bisher nicht im Detail untersuchte dynamische CORBA und somit insbesondere die Möglichkeiten einer losen Kopplung zwischen Klienten und Serverseite. Damit verbunden ist das Erfragen und das Interpretieren von Informationen über entfernte Objekte zur Laufzeit, mit diesen unbekannten Objekten zu arbeiten und anstelle des synchronen Paradigmas, das durch den entfernten Methodenaufruf häufig oktroyiert wird, die asynchronen Kommunikation zu verwenden. Von Interesse ist u. a. eine detaillierte Beschreibung von selbstbeschreibenden Datentypen und dynamischen Objekten, sowie eine detaillierte Untersuchung des Interface Repositories und der dynamischen Schnittstelle von CORBA.

Kapitel 7 ist eine erste Konsequenz aus den Kapitel 3 bis 6. Es beschreibt Grundlagen im Umgang mit CORBA, die zu beherzigen sind, wenn die verteilte Anwendung performant sein soll. Die Ergebnisse können unmittelbar auf andere Technologien wie Enterprise JavaBeans oder die (Java) Webdienste übertragen werden.

Kapitel 8, 9 und 10 beschäftigen sich mit Messungen bzgl. der Verarbeitungsgeschwindigkeit der dynamischen CORBA-Komponenten. Von Interesse ist hierbei

- ein Vergleich mit dem statischen Fall, um anhand der resultierenden Ergebnisse eine Zuordnung bzgl. enger oder loser Koppelung zu tätigen.
- den zeitlichen Aufwand, der durch das Interpretieren von Metadaten entsteht, um mit unbekannten Objekten zu arbeiten.
- der zeitliche Aufwand, der notwendig ist, ein generisches Objekt zu erzeugen und mit Daten zu füllen.
- die Vorzüge der dynamischen Komponenten gegenüber dem statischen Fall zu analysieren.

Kapitel 11 untersucht allgemein die Arbeitsweise von Komponenten und welche Charakteristiken sie aufweisen sollten.

Kapitel 12 ist eine kurze Einführung in die Beschreibungssprache XML, die im Zusammenhang mit den Deployment Deskriptoren bei Enterprise JavaBeans und den Web Services eine zentrale Rolle spielt.

In Kapitel 13 werden die Komponenten der Enterprise JavaBeans im Detail vorgestellt. Dies beinhaltet Session- und EntityBeans, den Java Message Service für die asynchrone Kommunikation inklusive Message Driven Beans und die Transaktionspolitiken von Enterprise JavaBeans.

Kapitel 14 vergleicht die beiden Technologien CORBA und J2EE miteinander.

Kapitel 15 beschäftigt sich mit dem Micorsoft COM/DCOM und den beiden wichtigen Komponenten Microsoft Message Queue und Microsoft Transaction Server, die auch in Microsoft .NET unverzichtbar sind.

Kapitel 16 vergleicht CORBA mit DCOM.

Kapitel 17 ist eine Einführung in die Web Services, inklusive des Protokolls SOAP, der Dienstbeschreibung WSDL und der benötigten Registrierung des Dienstes in eine Registrierungseinheit (UDDI).

Kapitel 18 vergleicht die Technologien CORBA und Web Services miteinander.

Generell werden in der vorliegenden Dissertation die Bereiche Kommunikation im statischen und dynamischen Fall, Aufbau und zeitliche Aspekte von dynamischen, zur Laufzeit unbekannten Datentypen und das Arbeiten mit der dynamischen Schnittstelle untersucht. Dies erfolgt auch im Vergleich von CORBA mit anderen führenden Technologien wie Web Services, DCOM und J2EE.

In Kapitel 19 werden die erzielten Ergebnisse zusammengefasst und ein Ausblick auf zu erwartende zukünftige Entwicklungen bzgl. eng und lose gekoppelten verteilten Anwendungen gegeben. Es wird eine Gegenüberstellung der Technologien in Form einer Matrix angegeben und eine Bewertung der erzielten Ergebnisse.

1 Grundlagen

Um Missverständnisse zu vermeiden, werden in den folgenden Kapiteln grundlegende und häufig benötigte Begriffe definiert und eine Einführung in die Themengebiete gegeben, die für den weiteren Verlauf unerlässlich sind.

1.1 Definitionen und Konventionen

Die folgenden Definitionen sind grundsätzlicher Art und werden lediglich angegeben, um eine einheitliche Fachterminologie herzustellen. Definitionen, die nicht unbedingt globaler Art sind, werden an den jeweils relevanten Stellen angegeben.

Definition 1: Betriebsmittel (Ressource)

Ein *Betriebsmittel (Ressource)* ist ein abstrakter Begriff für alle Hard- und Softwarekomponenten eines Rechners oder Rechnernetzes.

Dies beinhaltet z. B. Hardwarekomponenten wie Drucker, Diskettenlaufwerk oder eine Festplatte und Softwarekomponenten wie z. B. Dateien, Datenbanken, gemeinsamer Speicher u. ä.

Grundlegend für die Softwareentwicklung sind die Begriffe Programm, die Aktionen eines Programms und der Begriff des Prozesses, der nicht immer einheitlich gehandhabt wird. Bspw. heißt ein Prozess in der Prozessdatenverarbeitung *Task* und in der DEC-Terminologie wird häufig von einem *Image* gesprochen.

Definition 2: Programm

Ein *Programm* ist eine ausführbare Datei, bestehend aus Anweisungen und Daten.

Definition 3: Aktion

Eine *Aktion* A_k, $k \varepsilon \mathbf{N}_0$ ist eine atomare Einheit, d. h. sie ist eine nicht unterbrechbare Folge von Instruktionen.

Definition 4: Prozess

Eine logisch zusammenhängende Sequenz P von n Aktionen A_k, mit $k \varepsilon \mathbf{N}_0$, bezeichnet man als *Prozess*, in Zeichen $P := (A_0, ..., A_{n-1})$.

Demzufolge setzt sich ein Prozess aus mehreren atomaren Einheiten, den Aktionen, zusammen. Es ist wichtig festzuhalten, dass ein Prozess nicht unbedingt zeitlich zusammenhängend ablaufen muss, sondern lediglich die einzelnen Aktionen, aus denen er besteht. Die Aktionen eines Prozesses können gemäß der obigen Definition niemals parallel ablaufen.

Der Unterschied zwischen einem Programm und einem Prozess muss ebenfalls deutlich hervorgehoben werden: Ein Programm kann zu mehreren Prozessen führen, da es in einem Multitasking-Betriebssystem mehrfach abgearbeitet werden kann. Ein Prozess ist vielmehr dadurch bestimmt, dass er zu einem festen Zeitpunkt die Aktivierung genau eines Programms ist.

Eine äquivalente Definition für einen Prozess kann somit wie folgt angegeben werden:

Definition 5: Prozess und Thread

1. Ein *Prozess* ist ein in Ausführung befindliches Programm.

2. Ein *Thread* ist ein leichtgewichtiger Prozess, der sich mit seinem Erzeuger den Adressraum teilt.

Der Unterschied zwischen einem Thread und einem Prozess besteht im Wesentlichen aus:
1. Ein Kindprozess wird erzeugt, in dem der Elternprozess eine vollständige Kopie von sich selbst erstellt und dieses Kind anschließend durch eine ausführbare Einheit überlädt und zur Ausführung bringt.
2. Zwischen dem Kind- und dem Elternprozess besteht ein Verwandtschaftsverhältnis. Terminiert der Erzeuger, so wird das Kind (unter UNIX) von dem init-Prozess adoptiert. Das Kind ist aber von der Terminierung des Elternprozesses ansonsten nicht betroffen.
3. Bei einem Thread wird lediglich eine Ausführungseinheit abgespalten. In Java ist dies z. B. ein Objekt, das das Schnittstelle *Runnable* implementiert, also die Methode *public void run()* geeignet implementiert hat.
4. Der Thread teilt sich mit seinem Erzeugerprozess den gemeinsamen Adressraum. Terminiert der Erzeuger, so werden alle von ihm erzeugten Threads ebenfalls zu einer ggf. vorzeitigen Terminierung gezwungen.
5. Der Thread bleibt fest verbunden mit seinem Erzeuger. Der Thread endet, sobald die Methode *run()* abgearbeitet ist.

Ein Thread hängt an seinem Erzeuger wie an einem Faden. Der Vorzug ist in der schnellen Erzeugung eines Threads zu sehen, als Nachteile sind zu benennen:
1. Die Abhängigkeit zu dem Erzeugerprozess. Terminiert der Erzeuger, so terminieren alle von ihm erzeugten Threads.
2. Bei der Verwendung von Threads ist ggf. aufgrund des gleichen Adressraumes der Zugriff auf bestimmte, gemeinsam genutzte Betriebsmittel oder gemeinsame Datenstrukturen zu regeln (s. u.).

I. A. versuchen Prozesse/Threads unabhängig voneinander auf bestimmte Betriebsmittel mehr oder weniger zeitgleich zuzugreifen.

Definition 6: Zueinander kritische Abschnitte

Eine Teilfolge von Aktionen eines Prozesses/Threads heißt ein *Abschnitt*. Zwei Abschnitte, die nicht parallel ablaufen dürfen, bezeichnet man als *zueinander kritisch*.

Kritische Abschnitte müssen durch bestimmte Mechanismen geschützt werden, d. h., dass bei dem Zugriff auf eine bestimmte Ressource der exklusive Zugriff eines Prozesses gewährleistet ist.

Definition 7: Synchronisation

Die Absprachen zwischen zwei oder mehreren Prozessen/Threads, die einen kritischen Abschnitt durchlaufen wollen, bezeichnet man als *Synchronisation*.

In der Absprache wird entschieden, welcher der Prozesse zu einem bestimmten Zeitpunkt den kritischen Abschnitt betreten darf. I. A. wird die Synchronisation von einem Master (z. B. dem Betriebssystem bei Verwendung von Semaphoren, der virtuellen Maschine von Java oder einem Transaktionsmonitor) geregelt.

In einem nachfolgenden Kapitel werden die benötigten Grundlagen für Client/Server-Systeme erarbeitet und alle für den weiteren Verlauf benötigten Begriffe im Hinblick auf Client/Server-Computing definiert. Dies beinhaltet parallele Prozesse ebenso wie die Synchronisation und die Kommunikation zwischen diesen Prozessen.

1.2 Der Einsatz monolithischer Prozesse

Die Entwicklung monolithischer Programmblöcke stammt aus der Großrechnerwelt und ist zeitlich vor der Einführung von Interprozesskommunikationsmechanismen und Netzwerken einzuordnen. Bzgl. der monolithischen Blöcke ist eine Reihe von Nachteile zu nennen:

1. Die Multitaskingeigenschaften des Betriebssystems werden schlecht genutzt.
2. Bei einer Mehrprozessoranlage ergibt sich durch den Einsatz eines monolithischen Blocks kein Vorteil, da dieser Block nur von einem Prozessor die CPU-Zeit bekommen kann.
3. Potentiell parallele Abläufe werden zwangsweise sequentialisiert.
4. Die Ausstattung des monolithischen Blocks mit Multitaskingfähigkeiten, die durch Threads herbeigeführt werden können, ist durch die enge Koppelung von Erzeugerprozess und Threads fatal.
5. Durch die Komplexität der Anwendung treten leicht Seiteneffekte auf.
6. Die Erweiterbarkeit der Anwendung ist in Frage gestellt. Aufgrund des zentralen Ansatzes besteht die Gefahr der „*Flickschusterei*“.
7. Eine Verteilung der Anwendung auf mehrere Rechner ist nicht möglich.
8. Die Testsituation eines großen Programms ist um ein Vielfaches ungünstiger als das Testen kleiner Programmeinheiten.
9. Neue Mitarbeiter benötigen einen langen Zeitraum der Einarbeitung, bevor sie derartige Programme pflegen oder gar weiterentwickeln können.

Bei Einsatz paralleler Prozesse können diese Nachteile weitestgehend vermieden werden, der Entwicklungs- und Testaufwand ist geringer, die Fehleranfälligkeit sinkt, die Wartbarkeit und Möglichkeit der Weiterentwicklung ist günstiger und die Prozesse können ggf. auf mehrere Rechner verteilt werden. Läuft die Anwendung nur auf einem Rechner, so werden die Multitaskingeigenschaften des Betriebssystems besser genutzt; bei Einsatz einer Mehrprozessoranlage ist echte parallele Verarbeitung möglich.

1.3 Parallele Prozesse

Die Bedeutung von parallelen Prozessen ist fundamental für die Verteilung von Anwendungen oder Systemen, die insbesondere durch die Begriffe Parallelität und Kommunikation geprägt sind. Die Begriffe verteilte Anwendung oder verteiltes System sind Softwareaspekte, d. h. verteilte Anwendungen und verteilte Systeme werden durch miteinander kommunizierende Prozesse realisiert.

Prozesse arbeiten unabhängig voneinander. Werden sie mit der Fähigkeit ausgestattet, untereinander zu kommunizieren, so können sie sich auch untereinander synchronisieren, d. h. zeitlich aufeinander abstimmen, und sie können kooperieren, d. h. eine gemeinsame Aufgabe bearbeiten. Auf Rechnern mit genau einer CPU erweckt das Multitaskingprinzip des Betriebssystems den Eindruck eines gleichzeitigen Arbeitens der Prozesse. In diesem Fall muss vielmehr von einer *Pseudoparallelität* gesprochen werden. Auf Rechner mit mehr als einer CPU, insbesondere in einem Rechnernetz, findet eine tatsächliche Parallelität statt, d. h., dass zwei oder mehr Handlungen zeitgleich stattfinden bzw. erfolgen können.

Eine weitere, nicht zu unterschätzende Bedeutung der parallelen Prozesse auf einem lokalen Rechner ist darin zu sehen, dass damit eine Abkehr von monolithischen Prozessen stattfindet und die Multitaskingfähigkeiten eines Betriebssystems besser genutzt werden.

In [Dannegger et al., 1991] werden parallele Prozesse wie folgt definiert:

Definition 8: Parallele Prozesse

Zwei Prozesse P und Q heißen *parallel*, wenn Q beginnt bevor P endet und umgekehrt. In Zeichen P || Q.

Mit dieser Definition werden beide Varianten der Parallelität erfasst. Zum einen die echte Parallelität, die auf Mehrprozessoranlagen gegeben ist und zum anderen die Quasi- oder Pseudoparallelität, die auf Einprozessoranlagen durch das Timesharing-Verfahren erreicht wird. Zu unterscheiden ist bei der echten Parallelität zwischen einer engen und einer losen Koppelung:

Definition 9: Enge und lose Koppelung

Prozessoren, die auf einen gemeinsamen Hauptspeicher zugreifen und eine gemeinsame Zeitablaufsteuerung benutzen, heißen *eng gekoppelt*. Anderenfalls nennt man die Prozessoren *lose gekoppelt*.

Rechnernetze sind somit typische, lose gekoppelte Multiprozessorsysteme, die mit entsprechender Software ausgestattet sind und die Prozessen auf dem einen Rechner eine Zusammenarbeit mit Prozessen auf dem anderen Rechner gestatten.

Etwas schwächer als der Begriff der Parallelität ist der Begriff der *Nebenläufigkeit* (*Concurrency*), der zum Ausdruck bringt, dass zwei oder mehr Handlungen in beliebiger Reihenfolge durchgeführt werden können.

Die beiden Formen der Parallelität und die Definition der Nebenläufigkeit bedingen, dass Prozesse in die Lage versetzt werden, sich in kritischen Abschnitten gegenseitig auszuschließen, d. h. die Parallelität wird in derartigen Fällen zwangsweise sequentialisiert. Diese Notwendigkeit ist bspw. gegeben, wenn mehrere Prozesse gleichzeitig schreibend auf einen gemeinsamen Speicherbereich zugreifen wollen. Der exklusive Zugriff von Prozessen kann auf einem lokalen Rechner durch die Verwendung von Semaphoren oder Monitoren geregelt werden. In einer verteilten Anwendung oder in einem verteilten System ist die Synchronisation von Prozessen nicht mehr mit lokalen Methoden handhabbar. Lamport stellte 1978 ein Verfahren vor, mit dem Ereignisse in einem verteilten System vollständig geordnet werden können.

Im Folgenden sei E die Menge von Ereignissen in einem verteilten System bzw. einer verteilten Anwendung.

Definition 10: "Happened Before" Relation

Zwei Ereignisse A, B ε E erfüllen die Relation *Happened-Before*, in Zeichen A $\rightarrow$ B, falls eine der folgenden Bedingungen erfüllt ist:

1. A und B sind Ereignisse innerhalb desselben Prozesses und A wurde vor B ausgeführt.
2. A ist das Ereignis Senden einer Nachricht N und B das Ereignis Empfangen der Nachricht N.
3. Die Transitivität der Relation $\rightarrow$ ist erfüllt, d. h.:

   $\forall$ A, B, C ε E mit A $\rightarrow$ B $\wedge$ B $\rightarrow$ C folgt A $\rightarrow$ C.

Lamport setzt voraus, dass die Happened-Before-Relation irreflexiv ist, d. h.

$\forall$ A ε E: $\neg$(A $\rightarrow$ A).

Dieser Sachverhalt entspricht unserem natürlichen Zeitverständnis. In Bedingung (1) spiegelt sich die Erkenntnis wider, dass innerhalb eines sequentiellen Prozesses die Aktionen angeordnet sind. Bedingung (2) ordnet die Ereignisse auch auf verschiedenen Rechnern an, wobei die grundlegende Erkenntnis verwendet wird, dass eine Nachricht erst abgeschickt werden muss, bevor sie empfangen wird. Bedingung (3) ist die transitive Hülle der in (1) und (2) definierten Relationenpaare. Des Weiteren folgt unmittelbar die Antisymmetrie der Relation, d. h. gilt A $\rightarrow$ B, so gilt auch $\neg$(B $\rightarrow$ A). Insgesamt folgt, dass die Happened-Before-Relation eine irreflexive Halbordnung ist.

Mit den Eigenschaften dieser Relation kann der Begriff der o. a. Nebenläufigkeit zweier Prozesse präzisiert werden:

Definition 11: Nebenläufigkeit (Concurrency)

Ereignisse A, B ε E, die nicht miteinander in "Happened-Before"-Relation stehen, heißen *nebenläufig* (*concurrent*).

Anschaulich formuliert besagt Definition 1.3.4, dass nebenläufige Ereignisse sich nicht beeinflussen können.

Ein klassisches Synchronisationsproblem ist der wechselseitige Zugriff auf gemeinsame Betriebsmittel, deren Erlangung in diesem Zusammenhang das Durchlaufen eines kritischen Abschnitts bedeutet. Trotz Synchronisation können kritische Abschnitte problematisch werden. Dies ist z. B. dann der Fall, wenn ein Prozess andere Prozesse davon abhält, bestimmte Betriebsmittel zu nutzen, d. h., wenn der Prozess Betriebsmittel *sperrt* und nicht wieder freigibt.

Ein Prozess P_0 benutzt das Betriebsmittel A, z. B. einen Drucker. Dazu hat P_0 rechtzeitig dieses Betriebsmittel gesperrt, also exklusiv für sich reserviert. Ein Prozess P_1 hat für sich ein Betriebsmittel B, z. B. ein Plattenlaufwerk gesperrt, das von P_0 für die weitere Arbeit, z. B. das Lesen bestimmter Daten, ebenfalls benötigt wird. P_0 muss also warten, bis P_1 das Betriebsmittel B freigibt. Will P_1 seine Arbeit mit dem Betriebsmittel B kurzfristig unterbrechen, um Daten zu drucken, so muss P_1 warten, bis P_0 den Drucker wieder freigibt. Da beide Prozesse nichts voneinander wissen, warten beide unendlich lange auf die Erlangung des erforderlichen Betriebsmittels.

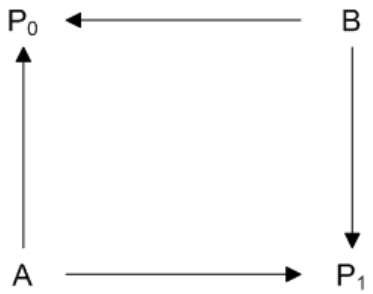


Abbildung 1: Deadlock-Situation

Definition 12: Verklemmung (Deadlock)

Wenn zwei oder mehr Prozesse sich gegenseitig blockieren und dadurch in einen inaktiven Zustand gelangen, so nennt man diesen Zustand eine *Verklemmung* (*Deadlock*).

Eine Verklemmung (s. Abbildung 1) ist also ein zeitabhängiger Zustand wechselseitiger Blockierung von einer Menge von Prozessen. Für das Eintreten eines Deadlocks sind die folgenden vier *Coffman*-Bedingungen notwendig:

(1) *Der wechselseitige Ausschluss*: Die Prozesse benötigen einen exklusiven Zugriff auf Betriebsmittel.
(2) *Warten auf*: Die Prozesse belegen bereits Ressourcen, während sie auf die Zuweisung von weiteren warten.
(3) *Keine Verdrängung*: Zugeteilte Betriebsmittel können nicht wieder entzogen werden.
(4) *Zirkulares Warten*: Es ergibt sich eine zyklische Wartesituation.

Man spricht vom *Verhindern eines Deadlocks*, wenn eine dieser Bedingungen ausgeschlossen wird.

In der Praxis findet man jedoch häufig die Vorgehensweise, dass die Synchronisation von Prozessen/Objekten durch einen Transaktionsmonitor vorgenommen wird, der eine Ver-

klemmung nicht erkennt, sondern über einen Zeitwert gesteuert die Auflösung eines Deadlocks vornimmt.

1.4 Client/Server-Systeme

Ein Client/Server-System ist ein Informationssystem, das die Rollen und den Ablauf der Zusammenarbeit verteilter Softwarekomponenten beschreibt. Die beteiligten Prozesse/Objekte können durchaus auf verschiedenen Rechnern platziert sein; in diesem Fall erfolgt die Kommunikation über ein Netzwerk. Befinden sich die Prozesse auf dem gleichen Rechner, so ist auch eine Kommunikation über lokale Interprozesskommunikationsmechanismen wie z. B. das Shared Memory oder Message Queues vorstellbar.

Ein Client/Server-Modell ist somit die Grundlage, Anwendungen - bestehend aus mehreren parallelen Prozessen/Objekten - auf mehrere Rechner zu verteilen und Systemressourcen sinnvoll einzusetzen.

Die Begriffe Klient und Server werden nicht immer einheitlich gehandhabt. Die einen verstehen unter einem Klienten grundsätzlich ein Endgerät und unter einem Server immer einen Host. Diese Vorstellung ist nicht vollständig korrekt, da sowohl der Dienstleistungsnehmer (Klient), als auch der Dienstleistungserbringer (Server) Prozesse/Objekte sind, die eine Aufgabe fordern oder erbringen. Demzufolge ist Client/Server-Computing das Zusammenspiel einzelner Prozesse/Objekte, die sich auf einem oder mehreren Rechnern befinden können.

Definition 13: Serverprozess, Klient

1. Ein *Server* ist ein Prozess, der eine bestimmte Dienstleistung im Auftrag anderer erbringt.
2. Ein *Klient* ist ein Prozess, der eine von einem Server angebotene Dienstleistung in Anspruch nimmt.

Entsprechend ist der Begriff eines Serverobjektes oder eines entfernten Objektes definiert.

Angesichts der Rollen, die einem Klienten bzw. einem Server zukommen, ergibt sich als äquivalente Definition:

Definition 14: Rollen von Klient und Server

1. Ein *Server* ist ein Prozess, der auf eingehende Anfragen wartet.
2. Ein *Klient* ist ein Prozess, der eine Verbindung zu einem Server initiiert.

Zwischen einem Klienten und Server besteht eine 1:n Beziehung in beide Richtungen, d. h. ein Klient kann im Laufe der Verarbeitung auf mehrere verschiedene Server zugreifen, und ein Server kann viele verschiedene Kunden bedienen.

Bei Serverprozessen/-objekten sind bestimmte Charakteristiken von Interesse, wie z. B. der Zustand der ggw. Verbindung.

Definition 15: Charakteristiken eines Server

1. Die Informationen, die ein Server über den Zustand einer Interaktion hat, heißen *Zustandsinformationen*.
2. Ein *zustandsbehafteter Server* verwaltet Zustandsinformationen über seine Klienten, so dass für die Dauer der Dienstleistung eine logische Verbindung zwischen Klient und Server besteht.
3. Ein *zustandsloser Server* speichert keine Zustandsinformationen über seine Klienten, d. h., dass ein Klient bei einem Datenaustausch mit einem zustandslosen Server bei jedem Zugriffswunsch sämtliche Parameter übersenden muss.

4. Ein *iterativer Server* ist ein Serverprozess, der alle Anfragen sequentiell abarbeitet.
5. Ein *konkurrierender Server* ist ein Serverprozess, der bei Eintreffen einer Anfrage einen neuen Prozess/Thread erzeugt, der die gestellte Aufgabe abarbeitet.

Mit den obigen Definitionen kann das Zusammenspiel paralleler Prozesse im Zusammenhang mit Client/Server-Systemen wie folgt definiert werden.

Definition 16: Client/Server-System

Ein *Client/Server-System* ist ein zwei- oder mehrstufiges Modell, das im Wesentlichen aus den folgenden Komponenten besteht:

1. dem/den Klient(en), um Anforderungen an die (verteilte) Anwendung zu richten.
2. dem/den Server, um im Auftrag der Klienten die Anforderungen durchzuführen, und die Antworten an den Auftraggeber zurückzuschicken.
3. einer Netzwerkinfrastruktur, i. A. ein **L**ocal **A**rea **N**etwork (LAN).

Ein Client/Server-System umfasst notwendigerweise Hard- und Software. Zur Hardware gehören die Rechner, die benötigte Peripherie und die Kommunikationsausrüstung. Auf Seiten der Software sind die eigentlichen Anwendungskomponenten, das lokale Betriebssystem, die Verteilungsplattform, Datenbanken und weitere Komponenten zu benennen. Es muss an dieser Stelle deutlich hervorgehoben werden,

1. dass Client/Server-Systeme nicht an spezifische Konfigurationen von Hard- und Software gebunden sind.
2. dass der Begriff Client/Server nicht unbedingt bedeutet, dass ein Klient stets auf einem PC und ein Serverprozess stets auf einem Serverrechner residieren.
3. dass der Begriff Client/Server nicht das Vorhandensein bestimmter Softwarefunktionen bei Klienten und Server impliziert. Bspw. bedeutet Klient nicht unbedingt, dass dieser Prozess über eine graphische Benutzeroberfläche verfügt.

Es wurde bereits hervorgehoben, dass die an einer Client/Server-Anwendung beteiligten Softwarekomponenten i. A. auf autonomen und ggf. auch auf heterogenen Rechnern mit verschiedenen Betriebssystemen laufen, die über ein Netzwerk miteinander verbunden sind.

Zur Realisierung derartiger Anwendungen bedarf es demzufolge einer unterstützenden Softwareinfrastruktur, die die Verteilungsfunktionalität handhabbar macht.

Definition 17: Middleware

Die Softwarekomponenten, die für die Verteilung eines Systems oder einer Anwendung notwendig sind, nennt man *Middleware*. Synonyme Bezeichnungen sind *Verteilungsplattform, Verteilungsinfrastruktur* oder auch *Netzbetriebssystem*.

Auf jedem an der Anwendung beteiligten Rechner wird die Verteilungsplattform als Betriebssystemzusatz installiert. Für ein allgemeingültiges Client/Server-Modell muss eine Verteilungsplattform die folgenden Punkte erfüllen:

1. Unabhängigkeit vom Transportsystem. Es muss für den Entwickler irrelevant sein, ob über ein Netzprotokoll oder lokal kommuniziert wird. Auch welches Netzprotokoll verwendet wird muss vor dem Entwickler verborgen bleiben.
2. Die Kooperation von Prozessen muss unterstützt werden. Diese Forderung ist eine unmittelbare Folgerung, die sich aus der Fähigkeit zur Kommunikation ergibt.
3. Die Verbergung der Komplexität. Der Klient besitzt keinerlei Information darüber wo, wann und wie seine Anfrage bearbeitet wird und wie viele Prozesse/Objekte an der Beantwortung der Anfrage beteiligt sind. Ggf. sind auch Altanwendungen eines Großrechners für den Benutzer nicht sichtbar zu integrieren.
4. Adaption, also die Anpassungsfähigkeit des Systems an veränderte Rahmenbedingungen. Dies beinhaltet die Forderung nach Erweiterbarkeit, der beliebigen Verteilbarkeit auf weitere Rechner und die Portabilität der Software über heterogene Hardware und Betriebssysteme hinweg.

In den nachfolgenden Kapiteln werden an den jeweils relevanten Stellen die o. a. Punkte detailliert betrachtet.

1.4.1 Verteilte Anwendungen ohne Verteilungsplattform

Client/Server-Systeme basieren auf parallelen Prozessen, die miteinander kommunizieren und ggf. sich untereinander synchronisieren müssen, um im Verbund gemeinsame Aufgaben kooperierend zu erledigen. Die häufig vorgetragene Ansicht, allen Prozessen einen so genannten *Well-Known-Port* zuzuweisen, damit ein Klientenprozess direkt Kontakt mit einem Dienst aufnehmen kann, der die von ihm gewünschte Dienstleistung erbringt, ist aus den folgenden Gründen nicht tragbar:

1. Jeder Klient benötigt das gesamte Verteilungswissen, d. h. er muss bei jeder Anforderung einer Dienstleistung wissen, auf welchem Rechner und an welchem Port sich ein geeigneter Service befindet.
2. Durch eine derartige feste Beziehung zwischen Klient- und Serverprozessen sind wichtige inhärente Eigenschaften verteilter Anwendungen wie z. B. Rekonfiguration der Anwendung oder Objektmigration zur Laufzeit nicht mehr möglich.
3. Die für eine verteilte Anwendung notwendigen Transparenzeigenschaften, wie z. B. Lokationstransparenz, entfallen weitestgehend.
4. Die Weiterentwicklung der Anwendung ist in Frage gestellt.
5. Die zur Verfügungstellung von Replikaten ist technisch möglich, aber die lastgesteuerte Verteilung von Anfragen ist gar nicht oder nur sehr schwer zu implementieren.
6. Die Ausfallsicherheit ist gar nicht oder nur sehr schwer zu implementieren.
7. Die Verwaltung der Ports ist in einem hohen Maße fehleranfällig.
8. Die Integration bestehender Anwendungen ist gar nicht oder nur sehr schwer möglich.

Aus den o. a. Gründen wird deutlich, dass eine Softwareinfrastruktur notwendig ist, um die Menge von Prozessen/Objekten sinnvoll zu verteilen und zu verwalten, wobei der Klient entsprechend entlastet und mit einem höheren Maß an Allgemeingültigkeit entwickelt werden kann.

Definition 18: Schicht (Tier) einer Softwarearchitektur, Model-View-Controller

1. Die einzelnen Softwareschichten einer Softwarearchitektur werden i. A. mit *Schicht* (*Tier*) bezeichnet (vgl. 2-Schichten-Architektur in Abbildung 2).
2. Das theoretische Modellierungskonzept zum Aufbau verteilter Anwendungen wird als *Model-View-Controller-Konzept* (MVC) bezeichnet.

Hierbei ist das Modell die eigentliche Geschäftslogik und u. U. auch die Datenzugriffsschicht, View bezeichnet genau den Teilausschnitt des gesamten Systems, der von dem Klienten gesehen wird und Controller bezeichnet einen Zugang zum serverseitigen System oder eine Geschäftskomponente, die mehrere Arbeitsschritte unsichtbar für den Klienten abwickelt.

In der Praxis findet man in Abhängigkeit der Anwendung u. U. eine Zusammenfassung von der View- und der Controllerschicht. Ist eine Softwareinfrastruktur gegeben, so gelangt man zumindest zu einer 3-Tier-Architektur. Ohne eine Verteilungsplattform ist zumindest eine 2-Schichten-Architektur gegeben. Ein typisches Beispiel für diese so genannte 2-Schichten-Architektur ist z. B. eine Java-Applikation, die direkt eine Datenbankanfrage stellt.

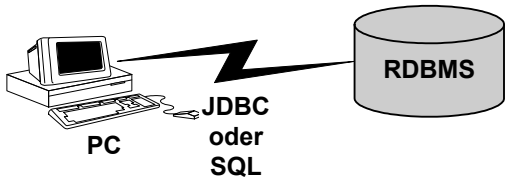


Abbildung 2: 2-Tier Architektur

In dem o. a. Beispiel (vgl. Abbildung 2) muss der Klient über eine Reihe von Informationen bzgl. der Datenbank verfügen. Bspw. ist der richtige Datenbanktreiber auszuwählen oder die Lokation des Datenbankservers muss bekannt sein.

1.4.2 Gründe für den Einsatz einer Verteilungsplattform

Um eine Anwendung oder ein System sinnvoll auf einen oder mehrere Rechner zu verteilen, bedarf es einer Softwareinfrastruktur, die es erlaubt Dienste weitestgehend transparent in Anspruch zu nehmen (vgl. [Coulouris et al., 1994], [Walpert, 1997]. In [Geihs, 1995] findet sich eine hervorragende Kurzeinführung zu diesem Thema, die hier primär als Grundlage dient). Eine detaillierte Beschreibung der Umsetzung dieser Anforderungen bzgl. CORBA befindet sich in Kapitel 5.

Die Notwendigkeit dieser Softwareinfrastruktur begründet sich wie folgt:

1. Klientenprozesse sollten weitestgehend unabhängig vom Verteilungswissen sein. Dies beinhaltet Informationen über an der Anwendung beteiligte Rechner, die Lokation der Objekte/Dienste etc. Grundsätzlich sollte der Benutzer lediglich spezifizieren, **was** er will und nicht **wer** die Anfrage ausführt, **wo** sie ausgeführt wird und **wie** seine Anfrage bearbeitet wird. D. h., dass auch das Serverbetriebssystem vor dem Benutzer verborgen bleibt.
2. Eine Anwendung lebt in dem Sinne, dass sie weiterentwickelt wird. D. h. es kommen neue funktionale Anforderungen hinzu, die durch zusätzliche Prozesse/Objekte beschrieben werden; andere Prozesse/Objekte werden durch Rekonfiguration der Anwendung auf andere Rechner verteilt etc.
3. Integration: Bestehende und zukünftig zu entwickelnde Anwendungen müssen derart integriert werden, dass eine einheitliche Systemsicht erhalten bleibt.
4. Verteilungstransparenz: Die einzelnen Arten der Transparenz werden in den folgenden Kapiteln detailliert betrachtet.

Weitere Gründe für den Einsatz einer Verteilungsplattform werden in den folgenden Kapiteln angegeben. Bisher muss festgehalten werden, dass eine Verteilungsplattform für eine verteilte Anwendung sinnvoll ist und dass durch den Einsatz von Middleware ein höheres Maß an Flexibilität erreicht wird als durch rudimentäres Client/Server-Computing.

Es existieren verschiedene Middlewareprodukte (s. Abbildung 3). Von Interesse muss hierbei die Interoperabilität verschiedener Produkte zueinander sein. Die Rolle des Klienten kann eine Java-Applikation, ein Servlet in einem Webserver oder ein PDA sein. Der Zugang zu einer CORBA-basierten verteilten Anwendung muss i. A. durch die Nutzung standardisierter Protokolle stattfinden. Verfügt ein Klient nicht über die notwendige Software, um direkten Kontakt zu einem CORBA-Objekt oder zu einem Enterprise JavaBean aufzunehmen, so ist ein zusätzliche Softwareinstanz, z. B. ein Webserver, notwendig. Auf diesem Webserver müssen geeignete Servlets die Klientenanfrage auf Basis des Protokolls RMI over IIOP (**R**emote **M**ethod **I**nvocation over **I**nternet **I**nter**o**rb **P**rotocol) weiterleiten.

Bei Verwendung von Middleware gelangt man automatisch zu einer 3-Schichten oder n-Schichten Architektur (vgl. Abbildung 3).

1.4.3 Anforderungen an eine Verteilungsplattform

Im Folgenden werden die Anforderungen beschrieben, die eine Verteilungsplattform für ein allgemeines Client/Server-System aufweisen muss, damit die Verteilbarkeit der Anwendung für die Anwendungsentwickler handhabbar und transparent ist. Der Fokus wird hierbei auf so genannte *general purpose*-Systeme gerichtet, die nicht speziell auf einen Anwendungsbereich zugeschnitten sind (vgl. Geihs, 1995).

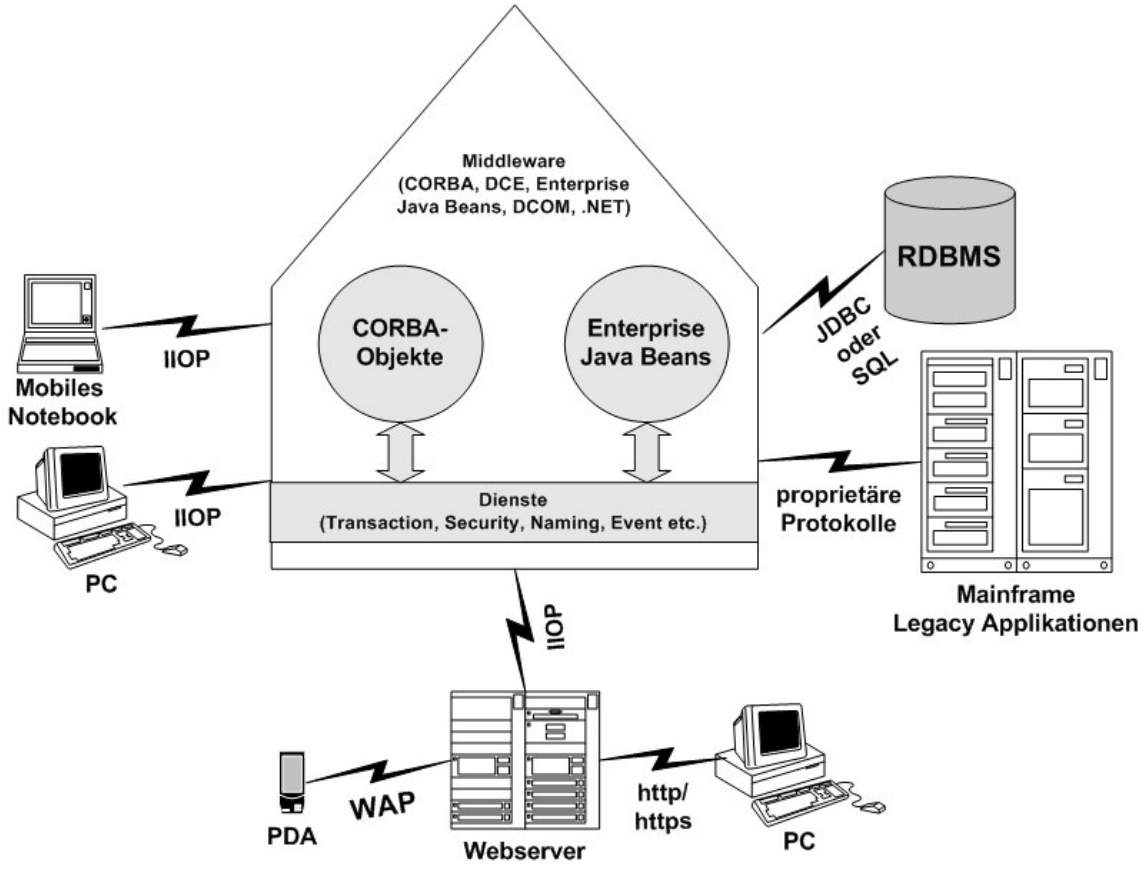


Abbildung 3: 3- bzw. n-Schichten Architektur

1.4.3.1 Unabhängigkeit vom Transportsystem

Genauso wie bei der Rechnerhardware findet man auch auf Seiten der Kommunikation eine Vielzahl verschiedener Protokolle und Dienste vor. Dazu gehören proprietäre Protokollarchitekturen wie IBM's SNA, Apple's Appletalk und Digital's DECnet, das bereits dem OSI-Referenzmodell nahe kommt; aber auch Standards bzw. Quasistandards wie OSI und TCP/IP.

Zur Beschreibung und Strukturierung der Architekturen wird i. A. das 7-Schichten OSI-Referenzmodell verwendet. Es gliedert die Kommunikationsvorgänge in einzelne Schichten, von denen jede für einen eigenen Aufgabenbereich zuständig ist.

Die Transportschicht (Schicht 4, TCP) hat vom Netz her gesehen als erste End-zu-End-Signifikanz. Die meisten Plattformen setzen hier auf. In den darunterliegenden Schichten 1 bis 3 erfolgt die Kommunikation abschnittsweise zwischen benachbarten Systemen, wobei

einige ausschließlich als Zwischensysteme fungieren und nur Funktionen der untersten drei Schichten ausführen können. Die Unterschiede bzgl. der Funktionalitäten der erwähnten Kommunikationsarchitekturen sind insbesondere in den oberen anwendungsnahen Schichten zu sehen. Es kommen dabei unterschiedliche Philosophien und Auffassungen über den Funktionsumfang und die Strukturierung dieser Schichten zum Ausdruck. Ein gemeinsamer Nenner lässt sich am ehesten an der Schnittstelle zur Transportschicht finden.

Eine Verteilungsplattform sollte unabhängig vom Transportsystem und den darunter liegenden Protokollschichten sein und somit die Komplexität der Netzsoftware vor dem Entwickler verbergen.

1.4.3.2 Kooperation

Die verteilte Verarbeitung auf Basis von Client/Server-Systemen ist das Zusammenspiel paralleler Prozesse, die sich auf einem oder mehreren Rechnern befinden. Eine Verteilungsplattform muss die Kooperation und damit auch die Synchronisation dieser Prozesse unterstützen, wobei das Grundschema des Ablaufs der Interaktion und der Programmierschnittstelle unterschiedlich ist.

Zur Kooperation gehören der Austausch von Daten, also die Kommunikation und eine Steuerung des zeitlichen Ablaufs der Interaktion, also die Synchronisation. Ferner gehört dazu eine Komponente, die das ordnungsgemäße Zusammenwirken aller Beteiligten regelt, also die Koordination. Zu berücksichtigen ist hierbei, dass mit der Verteilung von Anwendungen Probleme entstehen, die bei einem zentralistischen Ansatz entfallen. Dies sind z. B. Probleme wie Datenkonsistenz über mehrere Teilsysteme, Zugriffsschutz, Vertraulichkeit, aber auch das wichtige Thema der Administration, für das noch keine standardisierte Lösung existiert.

1.4.3.3 Nutzung gemeinsamer Ressourcen

Ressourcen sind nicht unbegrenzt verfügbar. Die unmittelbare Implikation hieraus ergibt, dass einer der Gründe, Anwendungen und/oder Systeme zu verteilen, in der gemeinschaftlichen Nutzung von Betriebsmitteln (Ressourcen) besteht. Eine Verteilungsplattform muss den parallelen Zugriff auf gemeinsame Ressourcen ermöglichen und steuern, d. h. ggf. muss die Verteilungsplattform Mechanismen für den exklusiven Zugriff auf eine bestimmte Ressource zur Verfügung stellen.

1.4.3.4 Heterogenität

In der Praxis findet sich bis zum heutigen Tag eine Reihe unterschiedlicher Systeme und Anwendungen, die in den verschiedensten Programmiersprachen implementiert wurden. Banken und Versicherungen besitzen eine Reihe von Großrechnern und eine Menge an Individualsoftware, die in Sprachen wie PL/1 oder COBOL entwickelt sind. Diese Heterogenität der Hard- und Software ist eine inhärente Eigenschaft von Client/Server-Systemen. Zum einen gehört die Integration heterogener, auf bestimmte Verarbeitungsschritte zugeschnittener Komponenten zu den großen Vorteilen verteilter Anwendungen und verteilter Systeme, andererseits trägt die Heterogenität maßgeblich zur Komplexität und somit auch zur Kompliziertheit bei.

Eine Verteilungsplattform sollte die Heterogenität weitestgehend verbergen, d. h. transparent machen. Dies beinhaltet Aspekte wie die Portabilität von Software, die Unterschiede und Verträglichkeit von Dateisystemen, die Konvertierung von zu übertragenen Daten in ein neut-

rales Format, Sicherheitsmechanismen und Vorkehrungen zum Management der Anwendung bzw. der Systeme.

Das Resultat derartiger Überlegungen führt zu offenen Systemen, die den Anwender unabhängig machen sollen von bestimmten Herstellern. In der Praxis ist dieses Ziel bisher nicht vollständig erreicht worden.

1.4.3.5 Transparenzeigenschaften verteilter Anwendungen

Angesichts der Komplexität verteilter Anwendungen und verteilter Systeme müssen Forderungen aufgestellt und erfüllt werden, die diese Komplexität vor dem Anwender und vor den Entwicklern verbergen, d. h. es bedarf einer *Verteilungstransparenz*. Das Ziel einer Verteilungsplattform ist, Transparenzeigenschaften anzubieten, um interne Abläufe vor denjenigen zu verbergen, die zur Erfüllung ihrer Aufgaben diese Details nicht benötigen. Es ist festzuhalten, dass Transparenz zu einem Effizienzgewinn bei der Entwicklung und Benutzung verteilter Anwendungen und Systeme führt.

Der Begriff *Verteilungstransparenz* beinhaltet eine Reihe einzelner Transparenzeigenschaften, wie z. B.:

1. *Ortstransparenz*: Die physische Lokation eines Prozesses/Objektes bleibt verborgen.
2. *Zugriffstransparenz*: Beim Zugriff auf lokale und entfernte Prozesse/Objekte ist kein Unterschied zu erkennen.
3. *Migrationstransparenz*: Die Verlagerung eines Prozesses/Objektes bleibt vor den Anwendern verborgen.
4. *Replikationstransparenz*: Die Mehrfachauslegung von Prozessen/Objekten bleibt verborgen und die Konsistenz der Replikate ist gewährleistet.
5. *Fehlertoleranz*: Das Auftreten von Fehlern/Störungen ist bestenfalls lokal sichtbar.
6. *Transaktionssicherheit*: Die Koordinationsmechanismen zur Sicherstellung von Transaktionseigenschaften genügen den ACID Anforderungen.
7. *Die Transparenz der Nebenläufigkeit*: Es ist nach außen nicht sichtbar, ob ein oder mehrere Prozesse/Threads die gestellte Aufgabe bearbeiten.

Ziel der Transparenzeigenschaften sollte es u. a. sein, dass ein Anwender lediglich spezifiziert, **was** er will und nicht **wer** diese Dienstleistung einbringt und **wo** sie erbracht wird. Eine totale Verteilungstransparenz kann es jedoch nicht geben.

1.4.3.6 Autonomie

In einem Client/Server-System kooperieren autonome Einheiten miteinander, um einen gemeinsamen Zweck zu erfüllen. Die Autonomie beschäftigt sich mit der Frage, wie viel eigene Kontrolle für die Zusammenarbeit im Kooperationsverbund aufgegeben wird, d. h. globale Interessen werden gegen lokale Interessen abgewogen. Ein Beispiel für Autonomie ist die Freiheit zu entscheiden, wie, wann und wo eine Ressource angeboten wird.

Eine Verteilungsplattform muss die Autonomieanforderungen unterstützen.

1.4.3.7 Adaption

Die Möglichkeit der kontinuierlichen Anpassung an veränderte Rahmenbedingungen, die sich in wirtschaftlichen Unternehmungen ständig ergeben, ist ein entscheidender Vorteil des Client/Server-Systems. I. A. ergeben sich ständig Veränderungen im Hinblick auf die Anzahl der Benutzer, der Hinzufügung von neuen Softwarekomponenten, dem Einspielen neuer

Softwareversionen, der Erweiterung von Rechnerknoten, auf die Verfügbarkeit von Komponenten und deren Auslastung usw. Eine Verteilungsplattform muss dieser Tatsache Rechnung tragen und eine geordnete, systematische Reaktion auf derartige Veränderungen anbieten. Dies beinhaltet neben den o. a. Beispielen auch die Möglichkeit des inkrementellen Wachstums, also der Skalierbarkeit.

Eine Verteilungsplattform muss die Adaptionsfähigkeit und Erweiterbarkeit gewährleisten, wobei nicht gefordert wird, dass in allen Fällen diese Veränderungen dynamisch erfolgen, sondern vielmehr, dass flexibel auf Wachstum und notwendige Rekonfigurationen reagiert werden kann.

1.5 Basisdienste einer Verteilungsplattform

Im Bereich der verteilten Anwendungen und Systeme hat sich weitgehend eine konsolidierte Auffassung darüber herausgebildet, welche Funktionalitäten eine Verteilungsplattform haben sollte.

1.5.1 Kommunikation und Kooperation

Um zwischen den einzelnen Komponenten eine Kooperation zu erreichen, bedarf es mehr als elementarer Kommunikation. Der Entwurf und die Implementierung verteilter Anwendungen und Systeme verlangt nach einem höheren Abstraktionsniveau, d. h. es sollte aus Sicht des Entwicklers ein adäquates Programmiermodell angeboten werden, das die Verteilung integriert und den Umgang mit heterogenen verteilten Systemen erleichtert.

Remote **P**rocedure **C**alls (RPCs) bzw. **R**emote **M**ethod **I**nvocation (RMI) stellen derartige Modelle zur Verfügung. Der Aufruf entfernt angebotener Dienste ist, bis auf gewisse Vorarbeiten, die ein Klient zu leisten hat, transparent, d. h. der Entwickler unterscheidet nicht zwischen einem lokalen und entfernten Methodenaufruf. Die durch die Verteiltheit erforderlichen Aktivitäten wie z. B. die Konvertierung der Daten in ein neutrales Format oder die Ermittlung der Lokation eines Dienstes bleiben weitestgehend verborgen.

1.5.2 Namens- und Verzeichnisdienste

Namen sind Bezeichner für Objekte wie z. B. Programme, Benutzer, Dateien oder Dienstleistungen. Namen dienen also der Identifikation von Objekten und geben bei suggestiver Namensbildung Aufschluss über den Bestimmungszweck des Objektes.

Der Name einer Ressource muss, vor dem Zugriff auf diesen Namen, auf eine tiefere Ebene abgebildet werden. Diese Aufgabe wird von einem Verzeichnisdienst (Directory Service oder Naming Service) geleistet. Es handelt sich hierbei i. A. um eine Entität, in der Informationen zu den im Verbund verfügbaren Objekten gespeichert sind. Ein Beispiel hierfür ist ein Applikationsserver, der seine Dienste exportiert und damit bekannt gibt, wie er für einen Dienstleistungsnehmer erreichbar ist. Konzeptionell ist ein Namens- oder ein Verzeichnisdienst eine Baumstruktur, die durchaus mit dem Aufbau eines Dateisystems z. B. unter UNIX vergleichbar ist.

Ein Verzeichnisdienst unterstützt die Entkoppelung von Client- und Serverprozessen durch dynamisches Binden zur Laufzeit. Die Vorgehensweise ist folgende:

1. Der Server gibt seine Dienstbereitschaft und die angebotenen Dienstleistungen dem Verzeichnisdienst bekannt (Export der Schnittstelle).

2. Klienten wenden sich an den Verzeichnisdienst, um für die gewünschte Dienstleistung geeignete Dienstanbieter ausfindig zu machen (Import der Schnittstelle).
3. Wird die Anfrage des Klienten an den Verzeichnisdienst positiv beantwortet, d. h. die Dienstleistung und der Dienstanbieter sind verfügbar, so kann der Klient im nächsten Schritt den ermittelten Server kontaktieren und die Dienste in Anspruch nehmen.

In einem großen verteilten System darf ein Verzeichnisdienst aus Gründen der ggf. geforderten Hochverfügbarkeit kein singulärer Punkt sein. Dies begründet sich mit der Notwendigkeit der Ausfallsicherheit und des Verlustes der Verarbeitungsgeschwindigkeit bei einem zentralen Ansatz. Deshalb sollte ein Verzeichnisdienst redundant sein und die Softwareinfrastruktur muss ggf. dafür Sorge tragen, dass diese Dienste sich gegenseitig synchronisieren.

1.5.3 Sicherheit

Aufgrund der gemeinsamen Nutzung von Betriebsmitteln in einer verteilten Anwendung oder einem verteilten System ergibt sich zwangsläufig die Frage nach dem Schutz des Zugriffs und der Sicherheit des Systems. Eine Verteilungsplattform muss die folgenden Sicherheitsbereiche abdecken:

1. Die Vertraulichkeit der Daten (confidentiality). Wird ein gesendetes Datenpaket von einem Unbefugten abgefangen, so muss sichergestellt sein, dass der Eindringling mit den Daten nichts anfangen kann.
2. Die Authentisierung und Autorisierung der Benutzer (authentication, authorisation). Bei der Authentisierung weist ein Benutzer nach, dass er der ist, für den er sich ausgibt. Ist ein Benutzer authentisiert, so kann eine Autorisierung vorgenommen werden, d. h. es wird festgelegt, welche Ressourcen der Anwender nutzen darf.
3. Die Integrität der Informationen (integrity). Eine Verfälschung der Daten wird z. B. durch eine digitale Prüfsumme verhindert. Es ist wichtig festzuhalten, dass die Datenintegrität eine schwächere Forderung darstellt als die Vertraulichkeit der Daten.
4. Die Protokollierung von Benutzereingaben (auditing). Bei verteilten Anwendungen kann es in Abhängigkeit der jeweiligen Unternehmung zu der Notwendigkeit der Revisionspflicht kommen. D. h. es muss irrtumsfrei festgestellt werden, welcher Anwender in der Vergangenheit welche Operationen mit welchen Daten ausgeführt hat.

Ohne eine ausreichende Authentifizierung und Autorisierung der Dienste wäre der gesamte Bereich des E-Commerce nicht vorstellbar.

Die Sicherheit in verteilten Anwendungen und Systemen stützt sich auf Basismechanismen der Kryptographie und vertrauenswürdigen (System-) Server.

Durch eine kryptographische Verschlüsselung soll die Vertraulichkeit und durch die Erstellung digitaler Signaturen die Integrität der Daten sichergestellt werden. Ebenfalls auf der Basis der Kryptographie werden dann Fragen der Authentisierung und Autorisierung in den dafür vorgesehenen Sicherheitsdiensten entschieden. Sicherheitsdienste sind ein wesentlicher Bestandteil einer Verteilungsplattform.

1.5.4 Uhrensynchronisation

In verteilten Systemen oder Anwendungen wird häufig mit Zeitstempeln gearbeitet. Bspw. arbeiten Sicherheitsdienste häufig mit Tickets, die über eine definierte Zeitspanne Gültigkeit haben und nach Ablauf der Frist von einem Anwender neu zu erwerben sind.

Damit kann sichergestellt werden, dass eine veränderte Autorisierung spätestens nach Ablauf der Frist greift. Ebenso kann bis zu einem gewissen Grad sichergestellt werden, dass nicht unbefugte Benutzer über einen vordefinierten Zeitraum das System nutzen.

2 Die Object Management Architecture

Die nächste Generation von Client/Server-Systemen wird mit Hilfe verteilter Objekte realisiert werden. Dies ist eine Folge der Abkehr vom Singleserver basierten LAN hin zur vielfach zitierten *Datenautobahn*, auf der jeder Rechner als Server oder Klient auftreten, ja sogar beide Rollen wahrnehmen kann.

Denkt man den Ansatz der *Datenautobahn* konsequent zu Ende, so landet man (vorausgesetzt, man bewegt sich in einer *objektorientierten Welt*) zwangsläufig beim Objekt als der Einheit, die auf dieser Datenautobahn agiert. Von da an ist es nur noch ein kleiner Schritt, das Objekt, da es als eigenständige Einheit in einem riesigen Verbund agiert, nicht mehr einem konkreten Rechner zuzuordnen, sondern das Objekt selbst als kleinste ansprechbare Einheit auf allen Ebenen eines Systems (Desktop, LAN, WAN, ...) anzusehen.

Das Objekt wird also zukünftig Gegenstand der Softwareentwicklung sein. Die Benutzung, Erweiterung oder die Aggregation von Objekten als Programmierprinzip, d. h. Software aus Komponenten, wird immer mehr in den Vordergrund rücken und die Entwicklung von Softwaresystemen *from scratch* zurückdrängen.

Daraus folgen nun zwei Notwendigkeiten:
1. Da ein Objekt zukünftig nicht mehr eindeutig einem Computer zugeordnet sein wird, muss ein Mechanismus geschaffen werden, der das Auffinden und das Management von Objekten in der verteilten Umgebung leistet.
2. Es müssen Standards für die Interaktion von Objekten geschaffen werden.

Beide Forderungen werden von der im Folgenden dargestellten **O**bject **M**anagement **A**rchitecture (OMA) erfüllt. Aus heutiger Sicht ist die Zugrundelegung des OMA-Standards bei einer Softwareneuentwicklung deshalb eine zukunftssichere und kostengünstige, sowie risikobegrenzende (durch Erwerb bereits im Einsatz befindlicher CORBA-Implementierungen) Entscheidung.

2.1 Definition der OMA

Im Folgenden wird die OMA (**O**bject **M**anagement **A**rchitecture) als Spezifikation einer Verteilungsplattform der in Kapitel 1 dargestellten Art vorgestellt [vgl. [Orfali et al., 1996]). Die OMA ist die Referenzarchitektur der OMG (**O**bject **M**anagement **G**roup), einer 1989 gegründeten Gemeinschaft von 11 Gründerfirmen, die heute bereits auf eine Mitgliederzahl von über 600 angewachsen ist, für verteilte Objektsysteme. CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture) wird häufig als Synonym für OMA verwendet, stellt genau genommen jedoch lediglich einen Teil der OMA dar (s. Abbildung 4). Die Spezifikation und Standardisierung der OMA-Komponenten ist bisher nur zum Teil abgeschlossen.

Wie aus Abbildung 4 hervorgeht, benennt die OMA die folgenden Hauptkomponenten:

1. ORB: Der **O**bject **R**equest **B**roker (ORB) stellt die zentrale Komponente der OMA dar. Er ist eine *dienstvermittelnde* Stelle: Jedes Objekt (Client) kann Dienstanforderungen (Service Requests) an den ORB stellen. Dieser vermittelt eine Anforderung an ein dienstbereitstellendes Objekt (Server). Nachdem der Server seine Arbeit beendet hat, der Dienst also erbracht ist, liefert der ORB die ihm vom Server bereitgestellten Ergebniswerte an den Klienten zurück. Bei diesem Ablauf ist weder für Client- noch für Serverobjekt von Bedeutung, wo (d. h. in welchem Prozess auf welchem Rechner in welchem Netzwerk) das jeweilige Kommunikationspartnerobjekt physikalisch residiert, in welcher Umgebung

(Betriebssystem) das Partnerobjekt residiert oder gar in welcher Programmiersprache das Partnerobjekt implementiert ist. Die Standardisierung des ORB ist abgeschlossen.

2. Object Services: Objektdienste sind wieder verwendbare Softwarebausteine, die ein Applikationsprogrammierer bei Bedarf benutzen kann. Sie sollen den Gebrauch und die Implementierung von Objekten erleichtern. Werden im Endausbau alle Object Services im Sinne der OMA-Spezifikation benutzt, so ermöglicht dies eine vom lokalen Betriebssystem und der lokalen Hardware unabhängige Applikationsentwicklung. Object Services sind z. B. für folgende Aufgaben definiert:
   - Auffinden von Objekten (Namensdienst).
   - Behandlung asynchroner Ereignismeldungen (Ereignisdienst).
   - Speichern von Objektzuständen (Persistenzdienst).
   - Sicherheitsrelevante Tätigkeiten (Security Service).
   - Synchronisation von Uhren (Timing Service).
   - Regelung der Erzeugung und Vernichtung von Objekten (Lifecycle Service).
3. Domain Services: Unter Domain Services versteht man spezielle Softwaresysteme für bestimmte Anwendungsgebiete (z. B. Telekommunikation oder Gesundheitswesen). Domain Services werden bezüglich ihres prinzipiellen Aufbaus standardisiert.
4. Application Objects: Applikationsobjekte sind konkrete, individuell gestaltete und optimierte Softwareprodukte, wie sie vom Benutzer gesehen werden: Eine solche Anwendung stellt somit eine Lösung für ein spezielles Problem dar. Application Objects werden von der OMG nicht standardisiert.

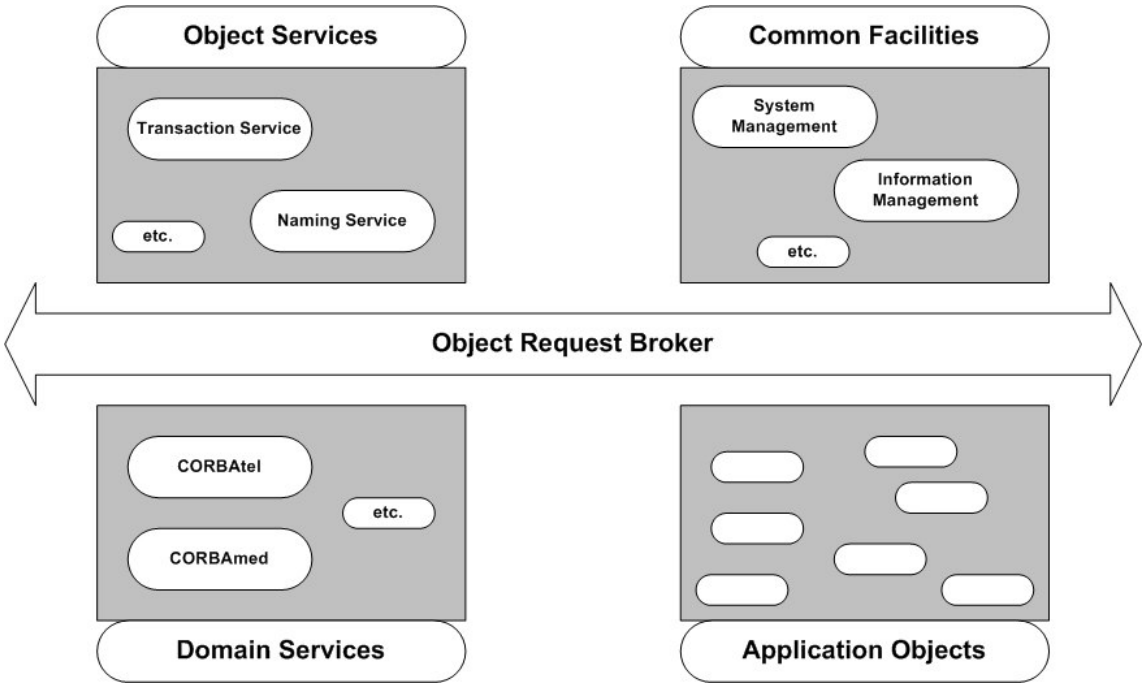


Abbildung 4: Die Object Management Architecture

Die Standardisierung der Object Services ist weitestgehend abgeschlossen. Zum ggw. Zeitpunkt existiert jedoch kein CORBA-Produkt, das alle Dienste implementiert hat. Die Hersteller beschränken sich auf die wichtigen Dienste wie Namensdienst, Transaktiondienst, Ereignisdienst und den Sicherheitsdienst.

Common Facilities: Common Facilities sind komplexe funktionale Einheiten, die auf den Object Services aufbauen. Sie unterscheiden sich von Object Services durch ihre stärkere Spezialisierung auf einzelne Teile einer Applikation, während Object Services von allen Anwendungskomponenten gleichermaßen als Grundbausteine genutzt werden können. Beispiele

für Common Facilities sind grafische Benutzeroberflächen oder email-Dienste. Durch die Verwendung standardisierter Common Facilities wird eine anwendungsübergreifende konzeptionelle Integrität erreicht. Diese führt dazu, dass sich verschiedene Systeme dem Benutzer gegenüber einheitlich darstellen, d. h. gleiche Aktionen (z. B. Doppelklick auf ein Objekt) in verschiedenen Applikationen führen zu gleichen Ergebnissen (aktiveren des Objektes, z. B. öffnen eines Textdokumentes).

3 CORBA

Für die vorliegende Ausarbeitung ist von der OMA-Spezifikation nur die CORBA-Spezifikation von Bedeutung (s. [OMG, 1995a]).

CORBA wird in drei Dokumenten beschrieben:
1. The Common Object Request Broker: Architecture and Specification.
2. CORBAservices.
3. CORBAfacilities.

Aus diesen Dokumenten gehen folgende durch CORBA spezifizierte Komponenten hervor:
1. Das CORBA-Objektmodell.
2. Der Object Request Broker (ORB).
3. Die Interface Definition Language (IDL).
4. Die Sprachabbildung zur IDL.
5. Die für die Zusammenarbeit von Klienten und Objektimplementierungen (Servern) notwendigen Mechanismen.
6. Die Standarddienste wie z. B. der Namens- oder der Transaktionsdienst.

3.1 Einführungsbeispiel

Im Folgenden wird ein einfaches Einführungsbeispiel angegeben. Der Klient kontaktiert einen Zeitgeber auf der Serverseite, der die Uhrzeit in Stunden, Minuten und Sekunden als Struktur an den Aufrufer zurückgibt.

Um das Einführungsbeispiel nicht unnötig mit Details zu belasten, wird hier auf die Verwendung eines Namensdienstes verzichtet. Stattdessen wird das Serverobjekt seine Objektreferenz als einen String in eine Datei schreiben, der Klient wird diesen String lesen und wieder in eine Objektreferenz wandeln.

Die Beschreibung eines entfernten CORBA-Objektes erfolgt über die Beschreibungssprache IDL. Festgelegt werden hier Namensräume, benutzerdefinierte Datenstrukturen und die Schnittstellen der entfernten Objekte. In dem nachfolgenden Beispiel wird in einem Namensraum *timeserver* eine Struktur *TimeOfDay* mit den Elementen *hour*, *minute* und *second* festgelegt. Der Zugang zu dem entfernten Objekt wird durch die Schnittstelle *interface Time* beschrieben. Die in der Schnittstelle aufgeführte Methode beschreibt neben dem Methodennamen und dem Rückgabewert die strikt einzuhaltende Methodensignatur, in diesem Beispiel eine leere Parameterliste. Der Rückgabewert der Methode ist die vorher deklarierte Struktur *TimeOfDay*.

```
module timeserver {
    struct TimeOfDay {
        short   hour;         // 0 - 23
        short   minute;       // 0 - 59
        short   second;       // 0 - 59
    };
    interface Time {
        TimeOfDay   getTimeOfDay();
    };
};
```

Programmbeispiel 1: IDL-Deklaration des timeservers

Die Registrierung von Objekten durch einen Server oder die notwendigen Vorarbeiten, die ein Klient leisten muss, sind zum einen eine monotone, stets wiederkehrende Aufgabe und

zumindest auf der Serverseite nicht standardisiert. Es erscheint somit sinnvoll, diese Aufgaben durch eine geeignete, separate Klasse *Init.java* zu beschreiben und in dem Klienten bzw. im Server zu verwenden. Damit ist auch eine leichte Anpassung an andere ORB-Produkte schnell möglich. In diesem Beispiel wird das CORBA-Produkt *ORBacus für Java* von der Firma Object Oriented Concepts verwendet, die zwischenzeitlich eine Tochterfirma des irischen CORBA-Herstellers Iona Technologies ist.

```
// Init.java: 1. Initialisierung des Java-ORBs ORBacus fuer den Klienten und fuer den Server
//            2. Methode zum Lesen einer 'stringified IOR' aus einer angegebenen Datei
//            3. Methode zum Schreiben einer 'stringified IOR' in eine angegebene Datei

package utilities;
import java.util.Properties;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.InvalidName;

public class Init {
    // Defaultkonstruktor:

    public Init()
    {
        // in den JDK's 1.2.x und neuer ist ein rudimentärer ORB enthalten. Um
        // Namens- und Klassenkonflikte zu vermeiden, wird durch das Setzen der
        // folgenden Eigenschaften auf die Klassen des tatsächlich verwendeten
        // ORBs umgeschaltet.

        props = System.getProperties();

        // als ORBClass wird die Klasse com.ooc.CORBA.ORB und
        // als ORBSingletonClass wird die Klasse com.ooc.CORBA.ORBSingleton
        // verwendet. Entsprechend ist eine Abbildung dieser Klassen bei Verwendung
        // anderer CORBA-Produkte anzugeben. Diese Vereinbarungen gelten nur bei
        // Verwendung der Sprache Java ab JDK 1.2.x oder neuer.

        props.setProperty("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.setProperty("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
        System.setProperties(props);
    }

    // initClient(): Initialisierung des Klienten-ORBs
    // Liefert eine RuntimeException, falls die Initialisierung fehlschlaegt.

    public void initClient(String[] args) throws RuntimeException
    {
        try {
               orb = ORB.init(args, props);
        }
        catch (SystemException e) {
               throw new RuntimeException(
                              "Init.initClient(): Der Klient kann nicht  " +
                              "initialisiert werden");
        }
    }

    // initServer(): Initialisierung des Server-ORBs und des Portable Object Adapters
    // Liefert eine RuntimeException, falls die Initialisierung fehlschlaegt.

    public void initServer(String[] args) throws RuntimeException
    {
        try {
               // initialisieren des ORBs

               orb = ORB.init(args, props);

               // POA anfordern und an den richtigen Typ anpassen. Dies ist notwen-
               // dig, da resolve_initial_references Rückgabewerte vom Typ
               // CORBA::Object liefert
               //
               // Ein POA ist ein Softwarebaustein, der zuständig ist für die Delegation
               // einer Anfrage an das richtige Objekt, für die Aktivierung und Deakti-
               // vierung von Objekten etc.
               org.omg.PortableServer.POA rootPOA =
                      org.omg.PortableServer.POAHelper.narrow(
```

```
                        orb.resolve_initial_references ("RootPOA"));
            manager = rootPOA.the_POAManager();
        }
        catch (SystemException e) {
                throw new RuntimeException(
                        "Init.initServer(): Der Server kann nicht " +
                        "initialisiert werden");
        }
        catch (InvalidName e) {
                throw new RuntimeException(
                        "Init.initServer(): "                      +
                        "resolve_initial_references (\"RootPOA\") " +
                        "fehlgeschlagen");
        }
    }


    // getPOAManager(): Referenz des 'manager's an den Aufrufer geben

    public org.omg.PortableServer.POAManager getPOAManager()
    {
        return manager;
    }

    // getORB(): Referenz des 'orb's an den Aufrufer geben

    public ORB getORB()
    {
        return orb;
    }

    // getProperties(): Liefert die fuer diesen Prozess aktuell gesetzte Umgebung zurueck.

    public Properties getProperties()
    {
        return props;
    }

    // readIOR(): Liest aus der spezifizierten Datei die 'stringified IOR' aus und liefert sie
    //            als String zurueck.

    public String readIOR(String filename) throws IOException
    {
        String ior = null;
        try {
                FileInputStream file  = new FileInputStream(filename);
                BufferedReader  in    = new BufferedReader(new InputStreamReader(file));
                ior = in.readLine();
                file.close();
        }
        catch(IOException ex) {
                throw new IOException(
                        "Die Datei '"  + filename +
                        "' existiert nicht oder ist nicht lesbar");
        }
        return ior;
    }

    // writeIOR(): Schreibt die angegebene 'stringified IOR' in die spezifizierte Datei.

    public void writeIOR(String filename, String ior)
                throws IOException
    {
        try {
                FileOutputStream      file    = new FileOutputStream(filename);
                PrintWriter           out     = new PrintWriter(file);
                out.println(ior);
                out.flush();
                file.close();
        }
        catch(IOException ex) {
                throw new IOException(
                        "Die Datei '"  + filename     +
                        "' kann nicht zum Schreiben geöffnet werden");
        }
    }

    private Properties                              props   = null;
```

```
    private ORB                                     orb     = null;
    private org.omg.PortableServer.POAManager       manager = null;
}
```

Programmbeispiel 2: Initialisierung und Hilfsmethoden des ORBacus

Im Folgenden wird das in IDL beschriebene entfernte Objekt implementiert. Von dem IDL Compiler wurde eine Klasse *timeserver.TimePOA* generiert, die hier als Basisklasse verwendet wird. Zu implementieren ist lediglich die Methode *TimeOfDay getTimeOfDay()*. Für die Implementierung werden keine CORBA-Spezialitäten benötigt.

```
// TimeImpl.java: Implementierung der entfernten Methode 'getTimeOfDay' aus Time.idl

package timeserver;

import java.util.*;

public class TimeImpl extends timeserver.TimePOA
{
    public TimeOfDay getTimeOfDay()
    {
        Calendar calendar = Calendar.getDefault();

        TimeOfDay tod = new TimeOfDay();

        tod.hour       = (short) calendar.get(Calendar.HOUR_OF_DAY);
        tod.minute     = (short) calendar.get(Calendar.MINUTE);
        tod.second     = (short) calendar.get(Calendar.SECOND);
        return tod;
    }
}
```

Programmbeispiel 3: Implementierung des Serverobjektes

Die Klasse Server initialisiert den Server-ORB, instanziiert ein Objekt vom Typ *TimeImpl* (s. o.) und einen Servant. Der Servant ist lediglich eine Programmiersprachenentität, in der das implementierte Objekt residiert und abgearbeitet wird.

```
// Server.java:

package timeserver;

import org.omg.CORBA.*;
import utilities.Init;

public class Server
{
    public static void main(String[ ] args)
    {
        String outFile = "/tmp/Time.ref";
        try {
                Init init = new Init();
                init.initServer(args);

                ORB            orb           = init.getORB();
                TimeImpl       timeServant   = new TimeImpl();
                Time           timeObject    = timeServant._this(orb);
                String         ior           = orb.object_to_string(timeObject);

                // die Objektreferenz wird für den Klienten als String in die angegebene
                // Datei geschrieben

                init.writeIOR(outFile, ior);

                // Aktivierung des Objektes und anschließend in einen (endlosen) Wartezustand
                // gehen

                (init.getPOAManager()).activate();
                orb.run();
        }
        catch (Exception e) {
```

```
                System.err.println("Serverinitialisierung fehlgeschlagen");
                System.exit(1);
        }
        System.exit(0);
    }
}
```

Programmbeispiel 4: Serverprozess für timeserver

Die einfachste Aufgabe fällt dem Klienten (*Client.java*) zu. Nach einer Reihe von Vorarbeiten scheint es so, als würde der Klient mit lokalen Objekten arbeiten. Diese Transparenzeigenschaft wird durch den vom IDL-Compiler generierten Softwarebaustein gwährleistet, der die Anfrage an die Serverseite weiterleitet und auch die Serialisierung der zu sendenden und empfangenden Daten umsetzt. Im Detail wird dies in einem späteren Kapitel erörtert.

```
// Client.java: Java-Klient, der ein Serverobjekt nach der Uhrzeit fragt.

import org.omg.CORBA.*;
import timeserver.*;
import utilities.Init;

public class Client {
    public static void main(String[ ] args)
    {
        Client client = new Client(args);
        client.run();
        System.exit(0);
    }

    public Client(String[ ] args)
    {
        try {
                if (args.length != 1) {
                        System.err.println("usage: java Client <file>, wobei " +
                                           "<file> die IOR enthaelt");
                        System.exit(3);
                }

                // nach der Initialisierung ist der Klientenorb initialisiert und somit
                // kann das benötigte ORB-Objekt durch die getORB() Methode
                // abgefragt werden. Nach Vereinbarung schreibt der Server die
                // Objektreferenz als String in eine Datei, die der Klient einlesen
                // und auswerten muss

                init = new Init();
                init.initClient(args);
                String ior = init.readIOR(args[0]);

                org.omg.CORBA.Object obj = (init.getORB()).string_to_object(ior);
                if (obj == null) {
                        System.err.println("Client(): Nil-Referenz");
                        System.exit(4);
                }

                // Anpassen an den richtigen Typ

                tm = TimeHelper.narrow(obj);
        }
        catch (Exception e) {
                System.err.println("Client(): Initialisierung fehlgeschlagen");
                System.exit(1);
        }
    }


    public void run()
    {
        // Uhrzeit vom Server erfragen. Nach den o. a. Vorarbeiten wirken die
        // die Aktivitäten des Klienten wie im lokalen Fall.

        System.out.println("Client.run()");
        TimeOfDay tod = tm.getTimeOfDay();   // entfernter Methodenaufruf
        System.out.println("Die Uhrzeit in GMT ist: " +
                        tod.hour       + ":" +
```

```
                    tod.minute    + ":" +
                    tod.second);
    }

    private Init       init    = null;
    private Time       tm      = null;
}
```

Programmbeispiel 5: Beispielclient für den timeserver

Um das o. a. Beispiel für das CORBA-Produkt ORBacus zu kompilieren und einen Testlauf zu tätigen, ist wie folgt vorzugehen.

1. Anlegen eines (temporären) Verzeichnisses für den vom IDL-Compiler generierten Code.
2. Übersetzen des generierten Codes.
3. Übersetzen des Klienten, der Implementierungsklasse und des Servers.

Es wird vorausgesetzt, dass eine Umgebungsvariable CLASSES definiert ist und dass diese Variable Teil der CLASSPATH-Variablen ist. Für den ORB ORBacus gilt also:

```
mkdir generated
jidl –output-dir generated Time.idl
javac –d $CLASSES generated/timeserver/*.java
javac –d $CLASSES Client.java TimeImpl.java Server.java
```

Der IDL-Compiler für Java (jidl) lenkt die Ausgabe der generierten Java-Klassen anhand des Kommandozeilenargumentes *--output-dir* in das angegebene Verzeichnis *generated* um. Das Subverzeichnis *timeserver* in dem Verzeichnis *generated* wird aufgrund der Modul-Vereinbarung in der IDL-Datei automatisch angelegt. Der Java-Compiler lenkt alle Kompilate in das durch CLASSES spezifizierte Verzeichnis um und legt anhand der *package*-Anweisung entsprechende Subverzeichnisse an.

Das Starten vom Server und dem Klienten erfolgt wie in Java üblich durch

```
java timeserver.Server &
java Client /tmp/Time.ref
```

Hierbei ist */tmp/Time.ref* die Datei, in die der Server die Objektreferenz als String schreibt und die dem Klienten bekannt zu geben ist.

3.2 Die Zugänge eines Klienten zu CORBA-Objekten

Zu klären sind im Vorfeld die möglichen Zugänge eines Klienten zu einem Serverobjekt. Zum ggw. Zeitpunkt existieren dazu vier Möglichkeiten:

1. Der Klient liest aus einer Datei eine so genannte *stringified IOR* (**I**nteroperable **O**bject **R**eference), also eine Objektreferenz, die von dem Server als String in eine Datei geschrieben wurde. Der Server muss dazu mit der Methode *object_to_string()* seine Objektreferenz in einen String konvertieren und der Klient muss nach dem Einlesen des Strings die inverse Operation *string_to_object()* verwenden. Es ist wichtig festzuhalten, das eine IOR von der OMG standardisiert wurde und somit die Interoperabilität zwischen verschiedenen CORBA-Produkten gewährleistet ist.
2. Der Klient verwendet den Objektdienst Trader, dessen Aufgabe es ist, Objektreferenzen inklusive einer Beschreibung zu exportieren.
3. Der Klient verwendet einen Namensdienst (vgl. Die Objektdienste von CORBA, Kapitel 5). Der Namensdienst ist hierarchisch, als Baumstruktur gegliedert und die Blätter des Baumes sind die Objektreferenzen, mit denen der Klient arbeitet. Dies setzt voraus, dass der Klient den Kontext der Objektreferenz kennt. In einem Flugunternehmen könnte der Kontext inklusive Objektreferenz lauten *Flug.LH.Auskunft*. Hierbei ist *Flug.LH* der Kontext

und *Auskunft* die begehrte Objektreferenz, die als Basisobjekt (in C++ CORBA::Object, in Java org.omg.CORBA.Object) an den Klienten geliefert wird.

4. Der Klient wendet sich an einen FactoryFinder, der Teil des LifeCycle-Dienstes ist (vgl. Die Objektdienste von CORBA, Kapitel 5). Bei Verwendung eines FactoryFinders wird, von der Spezifikation vorgegeben, serverseitig nach dem Designpattern *Fabrik* (Factory-Pattern) gearbeitet. Die Fabrik ist verantwortlich für das Erzeugen von Objekten auf der Serverseite und der Rückgabe der entsprechenden Objektreferenz an den Klienten. Die Vorzüge dieses Ansatzes sind:
   a. Das CORBA-Objekt wird von Aufgaben entlastet, die nicht zu seiner Geschäftslogik gehören. Ein Beispiel ist eine Bankanwendung. Das entfernte Objekt soll im Auftrag des Klienten Überweisungen tätigen und Kontostände abfragen können, aber es soll nicht wissen, wie ein Konto gefunden, angelegt oder gelöscht wird. Das ist die Aufgabe der Fabrik, an die sich der Klient wendet.
   b. Die Fabrik kann auf anderen Rechnern die Objekte und die hierzu gehörenden Prozesse für die weitere Verarbeitung erzeugen. Damit ist der Serverseite die Möglichkeit gegeben, die beteiligten Rechner gleichmäßig zu belasten und schonend mit Betriebsmitteln umzugehen.
   c. Die Fabrik kann aufgrund weiterer Informationen, die es von dem Klienten erhält, eine Lastverteilung basierend auf Daten vornehmen. Bspw. kann der Klient einer Bankanwendung anhand einer Filialnummer an einen bestimmten Rechner weitergeleitet werden.

Im Zusammenhang mit der Beschreibung der wichtigsten Objektdienste werden diese Zugänge vertieft.

3.3 Grundlegende Definitionen

Die Object Management Group umfasst zwischenzeitlich mehr als 900 Hersteller, mit denen gemeinsam eine Standardisierung bzgl. einer objektorientierten Softwareverteilungsinfrastruktur vereinbart wurde. Der Kern dieser Standardisierung ist CORBA.

Definition 19: Common Object Request Broker Architecture (CORBA)
Das Akronym CORBA steht für ***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*.

Hierbei bezieht sich das Attribut *common* auf zwei mögliche Zugänge zu einer verteilten CORBA-basierten Anwendung. Zum einen ein statischer Zugang, d. h. alle zu verschickenden Daten und alle Methoden des entfernten CORBA-Objektes sind in einer Beschreibungssprache bekannt gegeben und werden in genau dieser Form von dem Aufrufer verwendet.

Definition 20: Object Request Broker (ORB)
Der ***O**bject **R**equest **B**roker* (ORB) ist ein Softwarepaket für den Clienten und den Server einer verteilten CORBA-Applikation.

Für den Entwickler soll es möglichst unerheblich sein, wo sich ein entferntes Objekt befindet. Mit Ausnahme einiger weniger Vorarbeiten soll ein entfernter Methodenaufruf aussehen, als würde mit einem lokalen Objekt gearbeitet werden. Dazu ist ein „*entfernter Zeiger*“ notwendig, der die Lokation eines entfernten Objektes adressiert.

Definition 21: Interoperable Objektreferenz (IOR)
Der entfernte Zeiger zur Adressierung eines CORBA-Objektes ist die ***I**nteroperable **O**bject **R**eference* (IOR).

Der Aufbau einer IOR ist von der OMG vorgegeben:

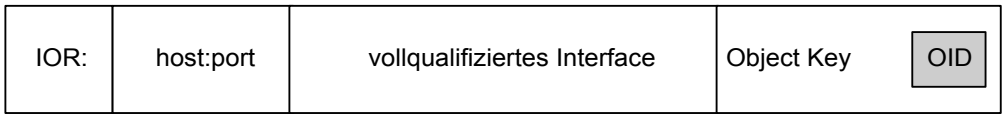


Abbildung 5: Aufbau der interoperablen Objektreferenz

Mit *host* ist ein DNS-Name oder eine TCP/IP-Adresse gemeint, der *port* ist i. A. die Portnummer eines Dämonprozesses für die Kommunikation mit dem Klienten, wie z. B. *orbixd* von Iona-Technologies oder der *ISL/ISH*-Prozess, der bei BEAs WLE/Tuxedo 8.1 verwendet wird. Ein entferntes Objekt ist immer die Instanz einer in IDL beschriebenen Schnittstelle und wird vollqualifiziert angegeben (s. Kapitel 3.5). Der letzte Teil der IOR ist herstellerspezifisch und kann z. B. genutzt werden für Lastverteilungskriterien oder für einen systemweit eindeutigen Objektidentifizierer, der als Primärschlüssel für eine Datenbank verwendet werden kann.

Ein Klient kann mit einem entfernten Objekt nur arbeiten, wenn er über die korrespondierende Objektreferenz verfügt. I. A. kann er diese über einen Namensdienst, über eine Fabrik (FactoryFinder), über einen Trader oder über eine zu einem String konvertierte interoperable Objektreferenz erhalten.

Aufgrund der Sprachunabhängigkeit und der entfernt gelegenen Objekte inklusive der zu übertragenen Daten wird von der OMG eine neutrale Beschreibungssprache zur Verfügung gestellt.

Definition 22: Interface Definition Language (IDL)
Für die Beschreibung des entfernten Objektes und der zu übertragen Daten wird die Beschreibungssprache ***I**nterface **D**efinition **L**anguage* (IDL) verwendet.

IDL ist die Sprache, in der Schnittstellen CORBA-gemäß spezifiziert werden. Sie stellt Konstrukte zur Spezifikation von objektbezogenen Operationen, deren Argumente und Rückgabewerte sowie Fehlerbehandlung bereit. IDL ist programmiersprachenunabhängig. Für die gängigen Programmiersprachen bestehen so genannte Sprachabbildungen, in denen festgelegt ist, wie IDL-Bestandteile in Konstrukte der jeweiligen Programmiersprache umgesetzt werden. Für alle kommerziellen CORBA-Implementierungen existiert zumindest eine Sprachabbildung für C++, die meisten Hersteller stellen inzwischen auch eine Java-Sprachabbildung bereit. Für andere gebräuchliche Programmiersprachen wie C oder COBOL wird nicht von allen Herstellern Unterstützung angeboten. Eine Einführung in die Beschreibungssprache IDL und die Sprachabbildung von IDL in die Zielsprache Java wird in einem nachfolgenden Kapitel beschrieben.

Eine CORBA–Implementierung enthält also mindestens einen ORB, eine Sprachabbildung und die IDL.

Durch die Standardisierung sollte eine Zusammenarbeit z. B. der Objektdienste von Hersteller A mit dem ORB von Hersteller B gewährleistet sein (dieser Punkt ist aber sicherlich im Einzelfall abzuklären).

In diesem Zusammenhang ist ein Punkt von großer Bedeutung: Die OMG produziert keine Quellcodes, d. h. Gegenstand der Standardisierung ist niemals ein *Programm*. Es werden auch keine Vorschriften darüber gemacht, wie (z. B. in welcher Programmiersprache, Datenbanken o. ä.) zu programmieren ist. Von der OMG werden im Wesentlichen Schnittstellen standardisiert.

3.4 Objektkommunikation in CORBA

Bei der Kommunikation in verteilten Objektsystemen kommen die Vorteile einer CORBA-konformen Implementierung besonders zum Tragen. Möchte ein Objekt A ein zweites Objekt B benutzen, d. h. A möchte eine Methode von B aufrufen, so hat A dafür lediglich zwei Dinge zu tun:

1. Holen einer Referenz auf Objekt B vom ORB.
2. Aufrufen der gewünschten Methode unter Zuhilfenahme der erhaltenen Referenz.

Das Holen der Objektreferenz wird dabei i. d. R. durch Aufruf eines Namensdienstes bewerkstelligt. Dies ist ein Objektdienst, dem der logische Name eines Objekts übergeben wird und der eine Referenz auf dieses Objekt zurückliefert.

Vorteil des Einsatzes von CORBA ist hier die vollständige Transparenz des *Aufenthaltsortes* der beteiligten Objekte sowie aller Aspekte der Aktivierung und Kommunikation. Selbst wenn das Serverobjekt (im obigen Beispiel Objekt B) *umzieht*, d. h. auf einen anderen Server transferiert wird, erhält A bei der nächsten Anfrage wieder eine Referenz auf B, diesmal eben auf einem anderen Server. Dieser Unterschied ist für A jedoch nicht relevant.

Durch CORBA werden hier also zwei Hauptprobleme gelöst: Lokalisierung eines (Server-) Objektes sowie Übertragung von *Anfrage* (Methodenaufruf) und *Antwort* (Ergebniswerte).

Grundsätzlich gilt: Beim Einsatz von CORBA interessiert den Software-Entwickler das benutzte Übertragungsprotokoll nicht. Er benutzt Objekte in der im vorigen Kapitel dargestellten Weise, die diese sehr tiefe Ebene der Kommunikation vor der Applikation verbirgt (vgl. Abbildung 6).

Trotzdem kommt natürlich innerhalb der CORBA-Implementierung ein Übertragungsprotokoll zum Einsatz. Außer diesem eigenen ORB-internen Protokoll muss jeder ORB ein einheitliches Protokoll unterstützen. CORBA schreibt zwei alternative Übertragungsprotokolle vor:

1. **G**eneral **I**nter-**O**RB **P**rotocol (GIOP): Dieses Protokoll spezifiziert Nachrichtenformate und Datenrepräsentationen für die Kommunikation zwischen ORBs. GIOP arbeitet direkt über jedes verbindungsorientierte Transportprotokoll. Die Implementierung von GIOP ist für jeden ORB verbindlich vorgeschrieben.
2. **DCE C**ommon **I**nter-**O**RB **P**rotocol (DCE-CIOP): DCE-CIOP wird normalerweise in einer OSF-DCE Umgebung eingesetzt. Hier können Applikationen die DCE-Sicherheits- und Managementdienste in Anspruch nehmen. CORBA-Implementierungen, die dieses Protokoll benutzen, müssen zusätzlich GIOP oder mindestens eine Bridge zu einer GIOP bereitstellenden Implementierung vorsehen.

Welches Protokoll benutzt wird, ist für den Anwendungsentwickler nicht von Bedeutung.

Ein weiteres, in Bezug auf CORBA, wichtiges Protokoll stellt IIOP dar (**I**nternet **I**nter-**O**RB **P**rotocol). IIOP definiert, wie GIOP-Nachrichten über ein TCP/IP-Netzwerk ausgetauscht werden können. Es wurde primär eingeführt, um das Internet selbst als ORB-Backbone, über den andere ORBs Nachrichten austauschen können, zu benutzen. Um CORBA 2.0-kompatibel zu sein, muss eine CORBA-Implementierung IIOP implementieren oder eine Bridge zu einem IIOP-fähigen System unterhalten.

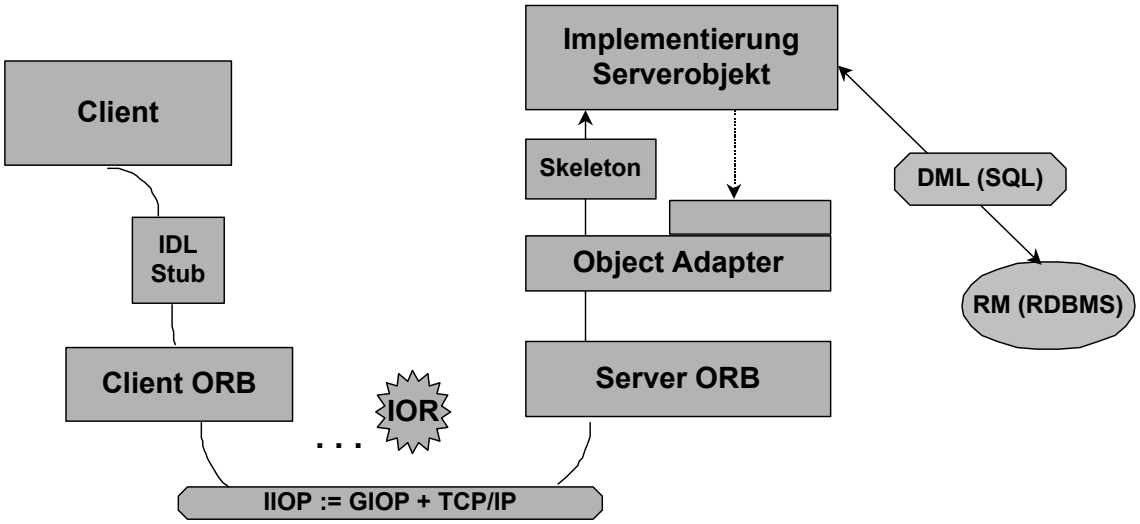


Abbildung 6: Objektkommunikation zwischen Klient und Server

3.5 IDL und die Sprachabbildung für Java

Von der OMG wurde als ein primäres Ziel die Interoperabilität zwischen Objekten, die in verschiedenen Programmiersprachen implementiert wurden, angesehen und angestrebt. Zur Beschreibung der entfernten Objekte dient die Beschreibungssprache IDL, die **I**nterface **D**efinition **L**anguage. In dem vorliegenden Kapitel wird eine Kurzeinführung in diese Beschreibungssprache gegeben, und zwar in dem Umfang, in dem es für den weiteren Verlauf notwendig ist. Detaillierte Beschreibungen finden sich u. a. in [Vinoski et al., 1999], [Redlich, 1996] oder in der IDL-Spezifikation der Object Management Group.

3.5.1 Kurzeinführung IDL

Mit IDL werden die Operationen, also die entfernten Methoden, des Objektes beschrieben und alle Daten, die zu diesem Objekt geschickt oder von dort empfangen werden.

In der folgenden Tabelle werden die IDL-Schlüsselwörter angegeben. Ein beträchtlicher Teil dieser Sprachmenge ist von den Programmiersprachen C bzw. C++ bekannt.

any	default	inout	out	switch
attribute	[long] double	interface	raises	TRUE
boolean	enum	[long] long	readonly	typedef
case	exception	module	sequence	unsigned
[w]char	FALSE	Object	short	union
const	float	octet	[w]string	valuetype
context	in	oneway	struct	void

Tabelle 1: Schlüsselwörtert von IDL

Der Datentyp *Object* bezeichnet die Basisklasse aller CORBA-Objekte und ist unerlässlich für die Rückgabe von Objektreferenzen. Der Empfänger einer Objektreferenz ist verantwortlich für die Anpassung an den richtigen Typ (vgl. Kapitel 3.5.3).

Die Schlüsselwörter *boolean*, *char*, *wchar*, *double*, *long double*, *float*, *long*, *long long*, *octet* und *short* bezeichnen die primitiven bzw. skalaren Datentypen von IDL. Im Gegensatz zu C oder C++ wird zusätzlich eine Standardisierung bzgl. der Wertebereiche vorgenommen. Ein *short* ist grundsätzlich eine 16 Bit große, vorzeichenbehaftete ganze Zahl, *long* ist eine 32 Bit große, vorzeichenbehaftete ganze Zahl und *char* repräsentiert ein einzelnes vorzeichenbehaftetes Zeichen mit 8 Bit Länge. Einige CORBA-Produkte unterstützen zurzeit bereits die Datentypen *wchar*, *wstring* und *long long*. Hierbei bezeichnen *wchar* und *wstring* den Unicode-Standard und *long long* ist ein 64-Bit Integer.

boolean bezeichnet einen Wahrheitswert und kann nur die Werte TRUE oder FALSE annehmen. Hier unterscheidet sich IDL von C bzw. C++, wo jeder von 0 verschiedene Wert als wahr angesehen wird.

octet ist ein vorzeichenbehafteter acht Bit Wert. Bei Verwendung dieses Datentyps findet kein Verpacken der Daten in einen neutralen Bytecode statt. Damit ist *octet* ein nicht interpretierter Wert. Eine Folge von *octets* eignet sich somit für die Übertragung unstrukturierter Daten wie z. B. digitalisierte Bilder.

Für die Datentypen *char*, *long* und *short* kann zusätzlich das Schlüsselwort *unsigned* verwendet werden. Hiermit wird angezeigt, dass der zu diesen Datentypen zugehörige Wert als absoluter Wert anzusehen ist.

Mit den Datentypen *float* und *double* werden Gleitkommazahlen mit einfacher oder doppelter Genauigkeit bezeichnet.

Mit dem Schlüsselwort *module* wird ein Namensraum erzeugt, wobei auch eine Schachtelung von Modulen möglich ist. Ein Namensraum ist optional anzugeben. Er dient jedoch der fein granularen Unterscheidung zwischen Deklarationen gleichen Namens. In derartigen Fällen ist eine Deklaration vollqualifiziert anzugeben.

```
module neuer_namensraum {     // Deklarationen (Schnittstellen, Strukturen etc.)
    struct S {
        // Strukturelemente
    };
};
```

Der Zugriff auf die Struktur S erfolgt dann vollqualifiziert über *neuer_namensraum::S* (C++) bzw. über *neuer_namensraum.S* in Java.

Programmbeispiel 6: Namensräume in IDL

Die Schlüsselwörter *in*, *inout* und *out* sind für die Art der Parameterübergabe in den Operationen notwendig. *in* besagt, dass ein Parameter ausschließlich vom Klienten zum Serverobjekt geschickt wird, *inout* deklariert, dass das Serverobjekt ein neues Datum als Antwort in diesem definierten Speicher ablegt und *out* zeigt an, dass ein Datum vom Serverobjekt zum Klienten geschickt wird.

Die Datentypen *struct* und *union* sind benutzerdefiniert. Eine *union* kann zu einem Zeitpunkt genau ein Datum aufnehmen. Welche Daten aufgenommen werden, wird im Vorfeld durch die Schlüsselwörter *switch* und *case* festgelegt.

```
struct Adresse {
    string str;
    long   plz;
    string ort;
};

union TestUnion switch (char) {
    case 'a': case 'A':
```

```
        Adresse anAdr;
    case 's': case 'S':
        short   aShort;
    default:
        long    aLong;
};
```

Programmbeispiel 7: Benutzerdefinierte Datenstrukturen in IDL

Von besonderem Interesse ist die Beschreibung der Schnittstellen entfernter Objekte. Dazu dient das Schlüsselwort *interface*. Innerhalb eines Interfaces können zusätzlich Vereinbarungen bzgl. benutzerdefinierter Datenstrukturen getroffen werden. D. h. eine Schnittstellenbeschreibung beinhaltet in einem übertragenen Sinn auch die Vereinbarung über einen Namensraum. Im Zusammenhang mit der Sprachabbildung für Java wird dies im Detail erläutert.

Die Deklaration einer Operation genügt dem Aufbau

```
[oneway] [void | <Returnwert>] <Methodenname>([[in | out | inout] <Parameterliste>])
[raises (<Liste möglicher Ausnahmen>])
```

Das Schlüsselwort *oneway* ist bei Verwendung der statischen Schnittstelle die einzige Möglichkeit der asynchronen Operationenaufrufe. Hierbei ist zu berücksichtigen, dass der Rückgabewert *void* ist und alle Parameter mit der Richtungsangabe *in* zu versehen sind. In der CORBA-Spezifikation ist nicht festgelegt, dass ein Klient über den Erfolg oder Misserfolg eines *oneway*-Aufrufes informiert wird. Sollte ein Klient erfolgreich einen asynchronen Operationenaufruf getätigt haben, so bleiben die Fragen offen, wie er die aus diesem Aufruf resultierenden Daten erhält. I. A. ist dies nur über den Eventservice möglich und bedeutet einen höheren Programmieraufwand.

```
interface Bestellung {
    oneway void asynchronBestellen(in Person p, in any bestellungsdaten);
    boolean     synchroneBestellung(in Person p, in any bestellungsdaten);
    any         synchronZustandAbfragen(in Person p);
};
```

Programmbeispiel 8: Schnittstellenbeschreibung in IDL

Bezüglich der Interfaces ist in IDL die Vererbung erlaubt.

Um eine Widerverwendbarkeit von IDL-Deklarationen zu ermöglichen, wurden die von C bzw. C++ bekannten Präprozessoranweisungen #*include*, #*define* und #*ifndef* mit genau der gleichen Bedeutung wie unter C bzw. C++ hinzugefügt.

```
// Datei Adresse.idl

// mehrfaches Inkludieren vermeiden

#ifndef ADRESSE_IDL
#define ADRESSE_IDL

module adressen {
    struct Adresse {
        string str;
        long   plz;
        string ort;
    };

    typedef sequence <Adresse> AdressenSeq;
};
#endif

// Datei Person.idl
```

```
#ifndef PERSON_IDL
#define PERSON_IDL

#include "Adresse.idl"

module personen {
    struct Person {
        string                name;
        string                vorname;
        adressen::AdressenSeq adr;
    };
};
#endif
```

Programmbeispiel 9: Adresse und Person in IDL

Ein häufig anzutreffendes Problem ergibt sich mit der Stabilität einer Schnittstelle. Wird in einer benutzerdefinierten Struktur lediglich ein neues Attribut hinzugefügt, bzw. ein bestehendes entfernt, so kann weder die Klienten- noch die Serverseite ohne neugenerierte Software das Verpacken bzw. Entpacken der Daten vornehmen. Eine mögliche Lösung ist die Schnittstellenversionisierung. Entfernte Objekte verschiedener Versionen werden auf der Serverseite vorrätig gehalten und der Klient muss anhand seiner IDL-Beschreibung angeben, mit welcher Version er arbeiten will. Dazu stellt IDL die Präprozessoranweisung *#pragma version* zur Verfügung.

Ein weiteres Problem ist die Vermeidung von Namenskonflikten innerhalb eines Namensraumes. Für jede IDL-Deklaration gilt, dass durch die Präprozessoranweisung *pragma prefix* den Schnittstellen ein Präfix vorangestellt wird, das zur Unterscheidung von Schnittstellen gleichen Namens sorgen kann.

```
#pragma prefix uni-mainz.de

module telefonbuch {
    interface Auskunft {
        // geeignete Operationen
    };
};
#pragma version telefonbuch::Auskunft 1.1
```

Programmbeispiel 10: Präfix in IDL

Der Klient kann anhand der IDL-Deklarationen die von ihm gewünschte Objektversion bestimmen. Dazu dient der vollqualifizierte Repository-Identifizierer, der sich wie folgt bildet:

`IDL:<Präfix>/<Modulname`$_1$`>/.../<Modulname`$_n$`>/<Schnittstellenname>:<Version>`,

wobei die Version immer den Aufbau `<Hauptnummer>.<Unternummer>` hat.

```
IDL:uni-mainz.de/telefonbuch/Auskunft:1.1
```

Programmbeispiel 11: Versionen in IDL

3.5.2 Der Datentyp valuetype

Aufgrund der verlangten Sprachunabhängigkeit ist die Beschreibungssprache IDL lediglich ein kleinster gemeinsamer Nenner bzgl. der Beschreibung der entfernten Objekte und der zu übertragenden Daten.

Mit der IDL-Spezifikation inklusive der Version 2.2 war es äußerst mühselig eine Baumstruktur zu beschreiben oder gar einen (gerichteten oder ungerichteten) Graphen. Genau diese Anforderung liegt häufig vor, wenn z. B. auf der Serverseite Daten zueinander in Beziehung gesetzt werden.

In der neuen CORBA-Spezifikation wurde deshalb ein neuer IDL-Datentyp *valuetype* aufgenommen, der es erlaubt

1. Objekte als Wert zu übertragen. Damit soll eine Minimierung der Kommunikation angestrebt werden. Das Objekt wird genau einmal zu einem Server übertragen, der füllt dieses Objekt mit Daten und anschließend wird das Objekt an den Aufrufer zurückgegeben. Genau genommen konnte dies auch schon mit benutzerdefinierten Datentypen realisiert werden. Der grundsätzliche Unterschied ist darin zu sehen, dass ein benutzerdefinierter Datentyp keine Funktionalität aufweist, während ein *valuetype* zusätzlich zu den Daten Methoden definieren kann, die auf diesen Daten lokal operieren. Das entspricht dem Unterschied zwischen einer Struktur in C und einer Struktur in C++, die genau genommen eine Klasse ist, in der die Sichtbarkeit von Daten und Methoden immer öffentlich ist.
2. Baumstrukturen und Graphen im Vorfeld durch IDL-Hilfsmittel zu beschreiben. Die Serialisierung dieser Strukturen erfolgt anhand der IDL-Beschreibung durch den Klienten- bzw. Server-ORB (s. Kapitel GIOP).
3. Ein *valuetype* kann jeden in IDL-definierten Datentypen aufnehmen.

```
typedef sequence<unsigned long> GewichteSeq;

valuetype GewichteterBinaererBaum {
    // Konstruktor

    init(unsigned long gewicht);

    // Datenelemente

    GewichteterBinaererBaum   linkerTeilbaum;
    GewichteterBinaererBaum   rechterTeilbaum;
    unsigned long             gewicht;

    // lokale Methoden

    GewichteSeqpraeOrdnung();
    GewichteSeqpostOrdnung();
};
```

Programmbeispiel 12: Ein gewichteteter, binärer Baum mit valuetype

In dem o. a. Beispiel werden drei lokale Methoden deklariert:

1. *init()*: Entspricht einem Konstruktor. Der Name ist immer *init*, gefolgt von einer Signatur. Es kann beliebig viele Konstruktoren gegen, die jedoch eindeutig, also durch ihre Signatur unterscheidbar sein müssen.
2. Die Methoden *praeOrdnung()* und *postOrdnung()* dienen der Traversierung durch den Baum und geben die Folge von Gewichten, in denen der Baum durchlaufen wurde, zurück. Die Implementierung der Methoden erfolgt in Abhängigkeit der Zielsprache lokal. Bei der Versendung eines *valuetypes* werden lediglich die Daten ausgetauscht. Der Empfänger ist verantwortlich für die Konstruktion eines lokalen *valuetypes*, der mit dem übersendeten Zustand initialisiert wird.

Wird der aufgebaute Baum an den Aufrufer zurückgegeben, so beginnt der Server-ORB mit der Serialisierung des Wurzelobjektes und des gesamten Zustandes. Da der Zustand wieder aus binären Bäumen besteht, wird rekursiv weiter serialisiert, bis der gesamte Baum als Bytefolge vorliegt und an den Empfänger gesendet werden kann. Der Empfänger konstruiert anhand der Bytefolge wieder einen binären Baum inklusive aller Zustände.

Es ist wichtig festzuhalten, dass *valuetype* kein CORBA-Objekt ist. Die unmittelbare Implikation ist, das dieser Datentyp nicht von der Basisklasse aller CORBA-Objekte, Object, abgeleitet wird. Stattdessen besitzt *valuetype* eine eigene Basisklasse *ValueBase*. Der Überhang der CORBA-Objekte wird damit vermieden, d. h. *valuetype* ist in diesem Sinne ein leichtgewichtiges Objekt.

3.5.3 Die Sprachabbildung von IDL zu Java

Nach Erstellen einer Beschreibung der entfernten Objekte via IDL wird in Abhängigkeit von der gewünschten Zielsprache ein IDL-Compiler verwendet, der die in IDL beschriebenen Schnittstellen und benutzerdefinierten Datenkonstrukte gemäß der Sprachabbildung umsetzt.

Die Umsetzung der skalaren Datentypen ist geradezu kanonisch und wird in der folgenden Tabelle zusammengefasst:

IDL	Java	Ausnahmen
boolean	boolean	CORBA::DATA_CONVERSION
[w]char	char	
double	double	
float	float	
[unsigned] long	int	
[unsigned] long long	long	
octet	byte	
[unsigned] short	short	
[w]string	java.lang.String	CORBA::DATA_CONVERSION CORBA::MARSHAL

Tabelle 2: Abbildung von IDL-Skalaren auf Java

Benutzerdefinierte Datentypen wie Strukturen, Unions werden auf abgeschlossene Java-Klassen abgebildet, und es werden jeweils ein Default- und ein Kopierkonstruktor generiert.

Für eine Union werden zusätzlich entsprechende Zugriffsmethoden und eine Diskriminatormethode generiert. Die Anzahl und Art der Zugriffsmethoden hängt von der Vereinbarung über die Datenmember in der IDL-Deklaration ab.

Sowohl für Strukturen, als auch für Unions, werden zusätzlich, wie unten beschrieben, geeignete Holder- und Helperklassen definiert.

```
struct StrukturTyp {
    long       aLong;
    string     aString;
};

union UnionTyp switch(short) {
    case 1:
        long aLong;
    case 2:
        string aString;
    default:
        char aChar;
};
```

Die Abbildung für Java:

```
public final class StrukturTyp implements org.omg.CORBA.portable.IDLEntity {
    public StrukturTyp() {}

    public StrukturTyp(int aLong, String aString)
    {
        this.aLong     = aLong;
        this.aString   = aString;
    }

    public int         aLong;
    public String      aString;
};
```

```
public final class UnionTyp implements org.omg.CORBA.portable.IDLEntity {
    public UnionTyp() { ... }
    public short      discriminator() { ... }
    public int        aLong() { ... }
    public void       aLong(int aLong) { ... }
    public String     aString() { ... }
    public void       aString(String aString) { ... }
    public char       aChar() { ... }
    public void       aChar(char aChar) { ... }
        ...
}
```

Programmbeispiel 13: Abbildung IDL-Union auf Java

IDL-Aufzählungstypen werden in eine Java-Klasse abgebildet. Intern wird für die vereinbarten Datenmember eine eindeutig bestimmte natürliche Zahl zugeordnet. Zusätzlich existiert eine Methode *value*, über die Werte abgefragt werden können.

Die folgende IDL-Deklaration

```
enum EnumTyp { rot, gruen, blau };
```

wird abgebildet auf:

```
public class EnumTyp implements org.omg.CORBA.portable.IDLEntity {
    public final static int          _rot   = 0;
    public final static EnumTyp      rot    = new EnumTyp(_rot);
    public final static int          _gruen = 1;
    public final static EnumTyp      gruen  = new EnumTyp(_gruen);
    public final static int          _blau  = 2;
    public final static EnumTyp      blau   = new EnumTyp(_blau);
    public int value()               { ... }
    public static EnumTyp            from_int(int value) { ... }
}
```

Programmbeispiel 14: Abbildung IDL-Enum auf Java

Ein IDL-*module* wird in Java in eine *package*-Definition umgesetzt, ein IDL-*interface* in eine Klasse. Werden mehrere Schnittstellen in IDL deklariert, so wird der Kompilierer entsprechend viele Java-Klassen in separaten Dateien anlegen. Dies ist darauf zurückzuführen, dass in einer Datei nur eine Java-Klasse definiert werden darf, die mit dem Attribut *public* gekennzeichnet ist.

IDL:

```
module test {
    interface X {
        ...
    };
};

module a {
    module b {
        ...
    };
};
```

Java:

```
package test;


public class X {
        ...
}
package a.b;
```

Programmbeispiel 15: IDL-Module, Interface auf Java

Bei der Parameterbeschreibung der Operationen wird für jeden benutzerdefinierten Datentyp eine Holder- und eine Helperklasse definiert, deren genereller Aufbau nachfolgend beschrieben wird. Die Notwendigkeit der Holder- und Helperklassen ist darauf zurückzuführen, dass

a. Java nur die Parameterübergabe *call-by-value* kennt. Bei Angabe von *out* bzw. *inout* für die Parameter ist jedoch eine Parameterübergabe *call-by-reference* explizit vereinbart. In Java wird dies durch eine geeignete Wrapperklasse, der Holderklasse, gelöst. Für die skalaren Datentypen sind diese Klassen bereits vordefiniert.

b. Jedes Datum muss ggf. in einen dynamischen Datentyp eingefügt bzw. ausgelesen werden. Dazu dienen die Methoden *insert* und *extract* in der generierten Helperklasse. Zusätzlich können Objektreferenzen immer nur vom Typ org.omg.CORBA.Object, also der Basisklasse aller CORBA-Objekte, übertragen werden. Der Klient muss demzufolge eine Anpassung an den richtigen Typ vornehmen. Dazu dient die Methode *narrow* in der Helperklasse. Wird ein Datum in einen dynamischen Datentyp eingestellt bzw. aus einem dynamischen Datentyp ausgelesen, so muss der Aufrufer in die Lage versetzt werden, das Datum genau zu identifizieren. Dies erfolgt über den so genannten *TypeCode* (s. Kapitel Dynamisches CORBA), der durch die Methode *type* der Helperklasse ermittelt wird.
c. Ein Klient sollte bei der Anfrage nach einer Objektreferenz immer den vollqualifizierten Repository-Namen angeben, um mögliche Namenskonflikte zu vermeiden und eine Referenz auf ein Objekt der richtigen Version zu bekommen. Dazu stellt die für alle Schnittstellen definierte Helperklasse eine Methode *id()* zur Verfügung, die genau diesen Namen inklusive Versionsnummer an den Aufrufer zurückliefert.

Bei der Generierung von Holder- und Helperklassen wurden für den Anwendungsentwickler zu viele Methoden entwickelt. Die Operationen *_read()*, *_write()* der Holderklasse und die Methoden *read()* und *write()* aus der Helperklasse sind aus Sicht eines Anwendungsentwicklers obsolet. Es wäre besser gewesen, diese Methoden in weitere, separate Klassen zu generieren.

```
public final class typeHolder implements org.omg.CORBA.portable.Streamable
{
    // der aktuelle Wert der Klasse vom Typ <type>

    public type value;
    public typeHolder() { // Defaultkonstruktor }
    public typeHolder(type value)
    {
        this.value = value;   // Kopierkonstruktor
    }
    public void _read(org.omg.CORBA.portable.InputStream in) { ... }
    public void _write(org.omg.CORBA.portable.OutputStream out) { ... }

    // liefert den TypeCode, also die Beschreibung des enthaltenen Objektes. Diese Methode ist
    // obsolet, da diese Information auch der Helperklasse entnommen werden kann (s. u.)

    public org.omg.CORBA.TypeCode _type()
    {
        ...
    }
}
```

Programmbeispiel 16: Aufbau der Holderklassen

Analog werden zu allen benutzerdefinierten Datentypen und beschriebenen Objektschnittstellen geeignete Helperklassen definiert.

```
public class typeHelper
{
    // für das Einstellen eines Datentyps in ein allgmeingültiges Objekt. Diese Methode ist
    // notwendig, da Java im Gegensatz zu C++ kein Überladen von Operatoren kennt.

    public static void insert(org.omg.CORBA.Any a, <type> t)
    {
        ...
    }

    // der inverse Vorgang. Ein Objekt wird aus dem allgemeingültigen any-Objekt extrahiert.

    public static type extract(org.omg.CORBA.Any a)
    {
        ...
    }
```

```
    public static org.omg.CORBA.TypeCode type()
    {
        // TypeCode des eingestellten Datums ermitteln und an den Aufrufer zurückgeben
    }

    public static String id()
    {
        // Ermitteln und Rückgabe des vollqualifizierten Schnittstellennamens
    }

    public void read(org.omg.CORBA.portable.InputStream in)
    {
        // für das Einlesen serialisierter Objekte.
    }

    public void write(org.omg.CORBA.portable.OutputStream out)
    {
        // für das Schreiben serialisierter Objekte.
    }

    // unbedingt notwendig zum Anpassen eines Objektes an den richtigen Typ.
    // Bspw. liefert ein Namensdienst Objektreferenzen immer vom Typ CORBA::Object
    // an den Aufrufer zurück.

    public static type narrow(java.lang.Object o)
    {
        // nur für abstrakte IDL-Schnittstellen
    }

    public static type narrow(org.omg.CORBA.Object o)
    {
        // Anpassen an den richtigen Typ
    }
}
```

Programmbeispiel 17: Aufbau der Helperklassen

Ein *valuetype* wird auf eine öffentliche Klasse gleichen Namens abgebildet, wobei ebenfalls für diesen Typ Helper- und Holderklassen generiert werden. Dies ist notwendig, da valuetypes wie jeder andere IDL-Typ auch als out- oder inout-Parameter in den Operationen verwendet werden können.

3.5.4 Die Übersetzung von IDL in eine Zielsprache

Für die jeweiligen Zielsprachen bieten die jeweiligen CORBA-Hersteller IDL-Compiler an. Welche Dateien generiert werden hängt hierbei von der Zielsprache ab. Bspw. generiert der WLE IDL-Compiler für C++ von BEA Systems für eine IDL-Datei *Test.idl* die C++-Dateien *Test_c.cpp*, *Test_c.h*, *Test_s.cpp* und *Test_h.h*.

Häufig kann bei dem Aufruf des Compilers auch die Generierung der serverseitigen Komponenten durch geeignete Optionen unterdrückt werden.

Die Datei *Test_c.cpp* enthält Informationen für die Klientenseite, also in erster Linie die benötigte Funktionalität, um einen Methodenaufruf transparent an das geeignete Serverobjekt zu delegieren, oder Informationen über die jeweiligen in IDL beschriebenen Datentypen und ihre Umsetzung von IDL in C++.

Test_s.cpp enthält entsprechende Vereinbarungen für die Serverseite.

Definition 23: Stub und Skeleton

1. Die generierte Software für den Klienten wird mit *Stub* bezeichnet.
2. Die generierte Software für die Serverseite wird mit *Skeleton* bezeichnet.

Bei der Umsetzung von IDL in Java werden entsprechend viele Dateien generiert. Neben Stub und Skeleton sind hier in Abhängigkeit der IDL-Schnittstellen und der benutzerdefinierten IDL-Datentypen die hierzu korrespondierenden Helper- und Holderklassen zu generieren.

4 Neutrale Datenformate

In einem heterogenen Umfeld ergibt sich das Problem, dass die einzelnen Rechner mit ihren unterschiedlichen Hardwareplattformen Daten auf verschiedene Arten darstellen. Dies beinhaltet die Notwendigkeit, die Daten entweder in einem neutralen Datenformat zu übertragen und der Empfänger konvertiert es in sein natives Format, oder die Daten werden in ihrem nativen Format gekennzeichnet und der Empfänger muss ggf. eine Konvertierung der Daten in sein eigenes Format vornehmen. In den folgenden Abschnitten werden diese neutralen Formate und der zu bewältigende Aufwand der Datenserialisierung untersucht. Eine detaillierte Darstellung findet sich z. B. [Ruh et al., 1999].

Definition 24: Big- und Little-Endian

Big-Endian und *Little-Endian* bezeichnen die Repräsentation der Byte-Darstellung bzgl. eines Prozessors.

Bei dem Big-Endian-Format wird das höchstwertige Byte zuerst geschrieben, bei dem Little-Endian-Format wird das höchstwertige Byte zum Schluss geschrieben.

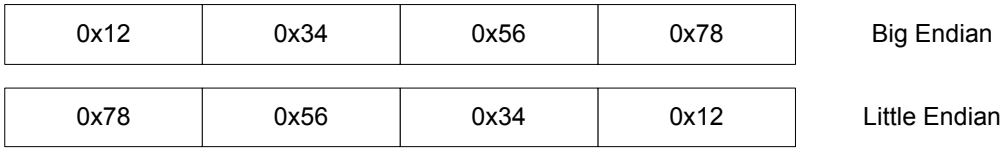


Abbildung 7: Die Folge 0x12 0x34 0x56 und 0x78 in Big- und Little Endian

Dies impliziert, dass Daten, wenn sie von einem Rechner mit Big-Endian-Darstellung auf einen Rechner mit Little-Endian-Darstellung transportiert werden sollen, in ein neutrales Format zu konvertieren sind.

Definition 25: Marshalling/Unmarshalling

1. Der Vorgang, Daten in ein neutrales Format zu konvertieren wird mit *Marshalling* bezeichnet.
2. Der inverse Vorgang wird mit *Unmarshalling* bezeichnet.

Ein weiterer Grund für das Verpacken von Daten in ein neutrales Format ist u. a. auch in der Standardisierung der Datengröße zu sehen. In C/C++ gibt es keine Vorgabe darüber, wie groß eine ganze Zahl (Datentyp int) ist, sondern es wird lediglich verlangt, dass die Relation

$8 \leq \text{char} \leq \text{short} \leq 16 \leq \text{int} \leq 32 \leq \text{long}$

erfüllt ist. Bei der Übertragung eines long-Wertes von Maschine A mit einer Wortlänge von 64 Bit auf eine Maschine B mit einer Wortlänge von 32 Bit würde ohne diese Standardisierung der Datengröße u. U. ein Fehler auftreten.

Statt bei jedem Aufruf die zu übertragenen Daten in ein neutrales Format zu verpacken, wäre es auch vorstellbar, in einem eigens vorgesehenen Feld des Datenstroms zu hinterlegen, in welcher Darstellungsform die Daten vom Sender geschickt werden. Der Empfänger der Nachricht ist dann verantwortlich, die Bytes des Datenstroms ggf. durch Byte-Vertauschung in sein natives Format zu konvertieren.

Diese Vorgehensweise verwendet CORBA mit dem CDR-Verfahren (**C**ommon **D**ata **R**epresentation).

4.1 GIOP und CDR

Die Darstellung von Daten als Skalare oder benutzerdefinierte Datenstrukturen entspricht einem natürlichen Verständnis des Entwicklers oder des Anwenders. Bevor ein Datum verschickt werden kann, wird es in einen Datenstrom konvertiert und auf der Empfängerseite wird dieser Strom wieder auf Skalare oder benutzerdefinierte Datenstrukturen abgebildet.

Definition 26: Serialisierung/Deserialisierung

1. Die Konvertierung von Skalaren und benutzerdefinierten Datenstrukturen bezeichnet man als *Serialisierung*.
2. Die umgekehrte Richtung, also aus einem Datenstrom wieder Skalare oder benutzerdefinierte Datenstrukturen zu erzeugen, bezeichnet man als *Deserialisierung*.

Damit sowohl Sender als auch Empfänger von Nachrichten mit den Daten arbeiten können, bedarf es also einer verbindlichen Absprache, die genau festlegt, wie die Daten serialisiert und vom Empfänger wieder deserialisiert werden. Von Seiten der OMG wurde dazu das Protokoll GIOP (**G**eneral **I**nter**O**RB **P**rotocol) definiert. Die Spezifikation besteht aus

1. Annahmen bzgl. der Transportschicht.
2. Die Abbildung von Daten in einen Datenstrom (CDR).
3. Nachrichtenformate: Insgesamt definiert die OMG acht Nachrichtentypen, die von den Klienten und Servern für die Kommunikation verwendet werden. Lediglich zwei dieser Nachrichtentypen sind notwendig, um die grundlegende Semantik von entfernten Methodenaufrufen zu beschreiben. Die restlichen sechs Nachrichtentypen dienen der Kontrolle und Optimierung des Datenverkehrs.

4.1.1 Annahmen zur Transportschicht

Das GIOP-Protokoll setzt bei der Transportschicht die folgenden Eigenschaften voraus:

1. Die Transportschicht ist verbindungsorientiert: Ein verbindungsorientierter Datentransport erlaubt dem Sender einer Nachricht durch Angabe von Rechnername und Port eine Verbindung an den Empfänger einer Nachricht aufzubauen. Diese Angaben sind Teil der interoperablen Objektreferenz, über die der Klient entfernte Methoden aufruft. Damit ist der Verbindungsaufbau gewährleistet.
2. Die Verbindung zwischen einem Klienten und einem Server arbeitet im Vollduplex-Verfahren: Der Empfänger einer Nachricht wird bei dem Wunsch eines Verbindungsaufbaus informiert, kann den Klienten akzeptieren und kann anschließend über eine andere Verbindung, über die Klient und Server sich verständigen, Daten empfangen und senden.
3. Die Verbindung ist symmetrisch: Sobald eine Verbindung zwischen einem Sender und einem Empfänger aufgebaut ist, können beide Beteiligten diese Verbindung wieder auflösen.
4. Der Datentransport ist zuverlässig: Das Transportprotokoll stellt sicher, dass Daten garantiert und genau einmal an den Empfänger zugestellt werden.
5. Die Transportschicht stellt eine Bytestromabstraktion zur Verfügung: Aus Sicht beider an einer Kommunikation Beteiligten scheint es, dass alle Daten als genau ein Datenstrom übertragen werden. Es besteht keine Notwendigkeit, einzelne Datenpakete auf der Empfängerseite in eine gewünschte Reihenfolge zu bringen.
6. Die Transportschicht informiert die Beteiligten über den Verlust der Verbindung: Sobald eine Verbindung z. B. aufgrund eines Netzwerkproblems nicht mehr existiert, erhalten beide Beteiligten bei erneutem Schreib- oder Leseversuch eine Fehlermeldung.

Diese Anforderungen werden z. B. von TCP/IP erfüllt.

4.1.2 Common Data Representation

In Anlehnung an die Beschreibungssprache IDL spezifiziert GIOP die Big-Endian und Little-Endian-Darstellung der folgenden Skalare:

Skalare	Beschreibung	Anzahl Bytes für die Darstellung
boolean	8-Bit Wert $\in$ {0, 1}	1
char	8-Bit Wert, abgebildet auf den Zeichensatz ISO Latin-1 8859.1	1
wchar	Ein 8-, 16- oder 32-Bit-Wert, der den Unicode-Zeichensatz repräsentiert	1, 2 oder 4
octet	Eine natürliche Zahl $\in$ [0 : 255]	1
short	Eine ganze Zahl $\in$ $[2^{15} : 2^{15} - 1]$	2
unsigned short	Eine natürliche Zahl $\in$ $[0 : 2^{16} - 1]$	2
long	Eine ganze Zahl $\in$ $[2^{31} : 2^{31} - 1]$	4
unsigned long	Eine natürliche Zahl $\in$ $[0 : 2^{32} - 1]$	4
long long	Eine ganze Zahl $\in$ $[2^{63} : 2^{63} - 1]$	8
unsigned long long	Eine natürliche Zahl $\in$ $[0 : 2^{64} - 1]$	8
float	Eine 32-Bit Gleitpunktzahl nach ANSI/IEEE 754-1985	4
double	Eine 64-Bit Gleitpunktzahl mit doppelter Genauigkeit nach ANSI/IEEE 754-1985	8
long double	Eine 128-Bit Gleitpunktzahl nach ANSI/IEEE 754-1985	16

Tabelle 3: Wertebereich für Skalare

Die Skalare *boolean*, *char*, *wchar*, *octet*, [*unsigned*] *short*, [*unsigned*] *long* und [*unsigned*] *long long* werden geradezu kanonisch abgebildet. Von Interesse ist die serialisierte Darstellung von *float*, *double* und *long double* und den benutzerdefinierten Datentypen.

Sobald ein Klient eine entfernte Methode aufruft, werden die zu dieser Operation korrespondierenden Daten in eine Bytefolge serialisiert. Die Vorgabe, wie Daten serialisiert werden, wird von der OMG vorgegeben.

In den beiden folgenden Abbildungen werden die Gleitpunktzahlen vom Typ *float* und *double* jeweils in ihrer Big-, sowie ihrer Little-Endian-Darstellung angegeben.

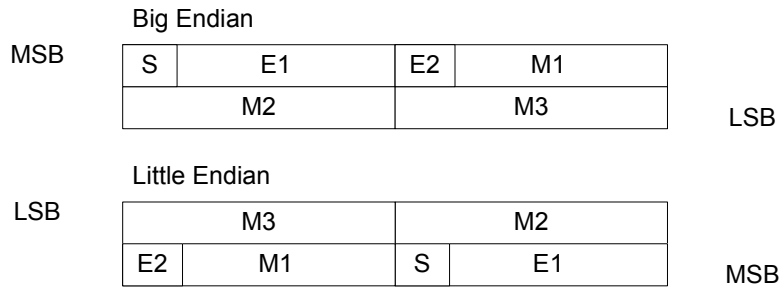


Abbildung 8: Eine float in Big- und Little-Endian Darstellung

Eine Gleitpunkzahl vom Typ *float* ist ein vier Byte großer Datentyp. Das höchstwertige Bit (in der obigen Darstellung mit *S* gekennzeichnet) bezeichnet das Vorzeichen der Gleitpunktzahl. Anschließend folgt der Exponent mit einer Gesamtlänge von 8 Bit. Die Mantisse wird mit den nachfolgenden 23 Bits dargestellt.

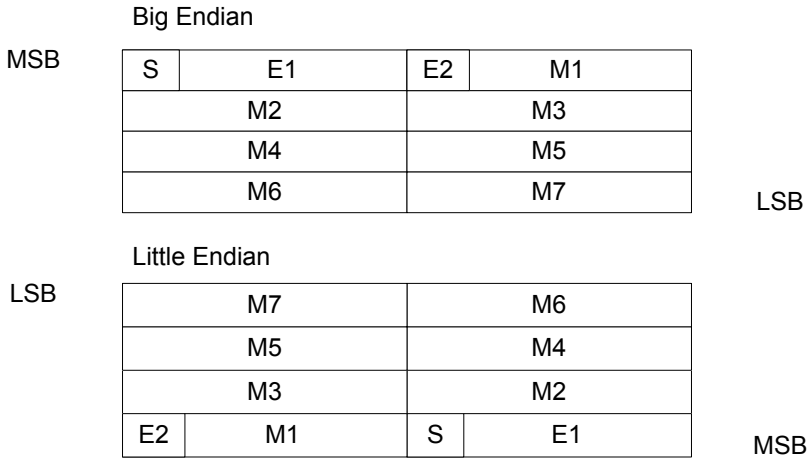


Abbildung 9: Ein double in Big- und Little-Endian Darstellung

Eine Gleitpunkzahl vom Typ *double* ist ein acht Byte großer Datentyp. Das höchstwertige Bit bezeichnet das Vorzeichen der Gleitpunktzahl. Anschließend folgt der Exponent mit einer Gesamtlänge von 11 Bit. Die Mantisse wird mit den nachfolgenden 52 Bits dargestellt.

Eine Gleitpunkzahl vom Typ *long double* ist ein 16 Byte großer Datentyp. Das höchstwertige Bit ist das Vorzeichen der Gleitpunktzahl. Anschließend folgt der Exponent mit einer Gesamtlänge von 15 Bit. Die Mantisse wird mit den nachfolgenden 112 Bits dargestellt.

In der Regel werden skalare Datentypen nicht einzeln zu dem entfernten Kommunikationspartner gesendet, sondern es handelt sich um eine Folge von unterschiedlichen Datentypen.

Definition 27: Alignment von Datenbereichen

Unter dem *Alignment* von Datenbereichen versteht man das Auffüllen einer Datenfolge durch Sonderzeichen derart, dass die Gesamtlänge der Folge ein Vielfaches einer vorgegebenen Länge ist.

In den folgenden Beispielen wird hier an Anlehnung des CDR-Format der hexadezimale Wert 0x20 (Leerzeichen) gewählt.

Gegeben: boolean b = true, long l = 132, short s = 56, char c = Q, long long ll = 234663:

Ohne Alignment

0x01	0x00 0x00 0x00 0x84	0x00 0x38	0x51	0x00 0x00 0x00 0x00 0x00 0x03 0x94 0xA7

Mit Alignment

0x01	0x20 0x20 0x20 0x20	0x00 0x00 0x00 0x84	0x00 0x38	0x51
0x20 0x20 0x20 0x20 0x20		0x00 0x00 0x00 0x00 0x00 0x03 0x94 0xA7		

Abbildung 10: Datentypen mit und ohne Alignment als Bytefolge serialisiert

Für die Serialisierung benutzerdefinierter Datentypen gelten die folgenden Regeln:

1. Strukturen: Die Serialisierung der Elemente einer Struktur erfolgt in der Reihenfolge ihrer Deklaration in der IDL-Beschreibung.

   Gegeben sei die Struktur

```
struct EineStruktur {
        long    x;
        short   y;
        long    z;
};
```

   Die Serialisierung dieser Struktur in der Big-Endian-Darstellung lautet unter der Annahme, dass alle Elemente den Wert 10 (0x0A) aufweisen:

```
0x00 0x00 0x00 0x0A 0x00 0x0A 0x20 0x20 0x00 0x00 0x00 0x0A
```

   Hierbei ist zum Auffüllen das ASCII-Zeichen Blank (0x20) verwendet worden.

Beispiel 1: Serialisierung einer Struktur

2. Unions: Bei der Union beginnt die Serialisierung immer mit dem Typ des Diskriminators, gefolgt von der Serialisierung des eingestellten Datums. Handelt es sich hierbei um einen weiteren benutzerdefinierten Datentyp, so gelten die Regeln für diesen Typ.

   Gegeben sei die Union

```
union EineUnion switch(char) {
        case 's':
                EineStruktur x;
        case 'x':
                long aLong;
        default:
                boolean b;
};
```

   Unter der Annahme, dass alle Werte der Strukturelemente 10 sind und für den Diskriminator der Wert *s* (0x73) gewählt wird, ergibt sich die folgende Serialisierung:

```
0x73 0x00 0x00 0x00 0x0A 0x00 0x0A 0x20 0x20 0x00 0x00 0x00 0x0A.
```

   Weist der Diskriminator den Wert *x* (0x78) auf und ist der long-Wert auf 10 gesetzt, so ergibt sich:

```
0x78 0x00 0x00 0x00 0x0A.
```

   Weist der Diskriminator einen von *s* und *x* verschiedenen Wert auf (z. B. *r* (0x72)), so ergibt sich:

```
0x72 0x01
```

   Der boolsche Wert wurde ohne Beschränkung der Allgemeinheit mit *true* angenommen.

Beispiel 2: Serialisierung einer IDL-Union

3. Sequenzen: Die Serialisierung einer Sequenz beginnt mit einem *unsigned long*-Wert, der die Länge der Folge wiedergibt. Die darauf folgenden Bytefolgen werden entsprechend den Regeln für die jeweiligen Typen serialisiert.

Die folgende IDL-Anweisung

```
typedef sequence<long> LongSeq;
LongSeq eineLongSequenz;
```

wird unter der Annahme, dass die Folge aus 6 Elementen besteht und die Elemente die Werte von 1 bis 6 annehmen, wie folgt serialisiert:

```
0x00 0x00 0x00 0x06          Länge der Sequenz
0x00 0x00 0x00 0x01          Erstes Element mit dem Wert 1
0x00 0x00 0x00 0x02          Zweites Element mit dem Wert 2
0x00 0x00 0x00 0x03          Drittes Element mit dem Wert 3
0x00 0x00 0x00 0x04          Viertes Element mit dem Wert 4
0x00 0x00 0x00 0x05          Fünftes Element mit dem Wert 5
0x00 0x00 0x00 0x06          Sechstes Element mit dem Wert 6
```

Beispiel 3: Serialisierung einer IDL-Sequence

4. Aufzählungstypen (enum): Ein Aufzählungstyp wird als ein *unsigned long* Wert serialisiert. Die Werte beginnen bei 0 und werden für jedes weitere Element entsprechend inkrementiert.

Die folgende IDL Anweisung

```
enum Farben { ROT, GRUEN, BLAU };
```

wird unter der Annahme, dass die Farbe BLAU mit dem Aufzählungswert 0x02 ausgewählt wurde, serialisiert durch

```
0x00 0x00 0x00 0x02
```

Beispiel 4: Serialisierung einer IDL-Enum

5. Wörter (string): Die Serialisierung eines Wortes beginnt mit einem *unsigned long*-Wert, der die Länge des Wortes festhält, gefolgt von dem Inhalt des Wortes, wobei jedes Zeichen durch ein Byte repräsentiert wird. Die Länge des Wortes beinhaltet auch das abschließende Null-Byte.

```
0x00 0x00 0x00 0x13          Länge des Wortes plus abschließendes Byte.
0x55 0x6e 0x69 0x76 0x65 0x72 0x73 0x69 0x74 0x61 0x65 0x74
 U    n    i    v    e    r    s    i    t    a    e    t

0x20     0x4d 0x61 0x69 0x6e 0x7a 0x00
Blank     M    a    i    n    z    \0
```

Beispiel 5: Serialisierung des Wortes „Universitaet Mainz“

Bei Verwendung von Unicode-Zeichen ist die Serialisierung komplizierter. Der Sender und der Empfänger einer Nachricht müssen entweder den gleichen Zeichensatz verwenden, oder der Zeichensatz des Senders muss in den verwendeten Zeichensatz des Empfängers konvertiert werden.

Von Interesse für dynamische CORBA-Komponenten ist zusätzlich die Serialisierung der Beschreibungstypen für das jeweilige Datum. In Kapitel 5 wird dieser Datentyp und seine Serialisierung im Detail beschrieben.

4.1.3 Die GIOP Nachrichtenformate

Sobald zwischen dem Klienten und einem Server eine Verbindung besteht, können beide Beteiligte Daten austauschen. Insgesamt spezifiziert die OMG acht GIOP-Nachrichtenformate, die in die beiden Kategorien Administrative und Objektaufruf-Nachrichten fallen.

Administrative Nachrichtenformate sind
a. *LocateRequest* und *LocateReply*, die dem Finden von geeigneten Objekten dienen.
b. *CancelRequest* und *CloseConnection*, um langlaufende Anfragen oder nicht länger benötigte Anfragen zu kontrollieren und ggf. zur Terminierung zu zwingen.
c. *MessageError* dient der ggf. notwendigen Fehlerbehandlung.

Nachrichtentyp	Verwendet von	Aufzählungsnummer	GIOP Version
Request	Klient	0	1.0, 1.1
Reply	Server	1	1.0, 1.1
CancelRequest	Klient	2	1.0, 1.1
LocateRequest	Klient	3	1.0, 1.1
LocateReply	Server	4	1.0, 1.1
CloseConnection	Server	5	1.0, 1.1
MessageError	Klient/Server	6	1.0, 1.1
Fragment	Klient/Server	7	1.1

Tabelle 4: Anfrage-/Antworttypen

Nachrichtenformate für den Objektaufruf sind
a. *Request*, für die Anfragedaten eines Klienten zu einem Server.
b. *Reply*, für die Antwortdaten eines Servers zu dem Klienten.
c. *Fragment*, für die Zergliederung einer Anfrage/Antwort in mehrere Teilstücke.

4.1.4 Der GIOP-Nachrichtenkopf

Sobald zwei Partizipierende anfangen Nachrichten miteinander auszutauschen, bedarf es einer Anleitung, wie mit den empfangenen Daten umzugehen ist, welches Objekt auf der Serverseite zu aktivieren ist, welche Objektmethode aufgerufen wird etc.

Die OMG beschreibt hierfür den GIOP-Nachrichtenkopf für die ggw. GIOP-Versionen 1.0 und 1.1.

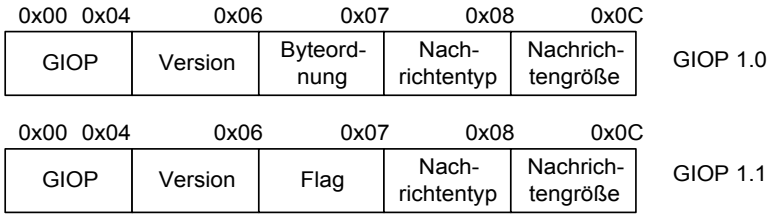


Abbildung 11: Der GIOP-Nachrichtenkopf in den Versionen 1.0 und 1.1

Der Nachrichtenkopf ist wie folgt aufgebaut:
1. Die ersten vier Bytes bezeichnen das Protokoll. Für GIOP-Nachrichten ist es immer das Wort GIOP.
2. Die beiden folgenden Bytes beschreiben die Haupt- und die Unterversionsnummer, also ggw. 1.0 oder 1.1.

3. Das folgende Byte beschreibt die Byteordnung (Version 1.0), also Big- oder Little-Endian. In der Version 1.1 ist dies durch ein Flagfeld ersetzt worden. Damit können bei späteren Erweiterungen weitere 7 Bit für eine zusätzliche Information verwendet werden. Ein Wert von 0 bezeichnet Big-Endian und ein Wert von 1 bezeichnet Little-Endian.
4. Das nächste Byte beschreibt den Nachrichtentyp, also *Request*, *Reply* etc. durch ihren Aufzählungstyp.
5. Die letzten vier Bytes geben die Nachrichtengröße als ein *unsigned long*-Wert an.

4.1.5 Der Kontext von Objektdiensten

Bei der verteilten Verarbeitung ist es notwendig, bestimmte Kontexte, z. B. den Kontext über den ggw. Zustand einer Transaktion, weiterzureichen. Diese Kontexte werden immer indirekt und implizit an die anderen an der Kommunikation Beteiligten zugestellt. Eine typische Situation, in der durch einen Dienst ein neuer Kontext erzeugt wird, ist der Sicherheitsdienst. Sobald ein Klient sich gegenüber dem System erfolgreich authentifiziert, wird ein so genanntes *SecurityCurrent*-Objekt in dem Adressraum des Klienten instanziiert und bei jedem entfernten Methodenaufruf wird der ORB des Klienten die notwendige Sicherheitsinformation der zu sendenden Nachricht hinzufügen. Damit soll sichergestellt werden, dass

a. ein Klient sich nur einmal authentifizieren muss.
b. die Sicherheitsinformation zur Serverseite gelangt, dort u. U. mit serverseitig gespeicherter Information verglichen wird und der Klient für bestimmte Tätigkeiten autorisiert werden bzw. abgewiesen werden kann.
c. der Entwickler von der Notwendigkeit, Sicherheitsaspekte programmtechnisch zu realisieren, weitestgehend entbunden wird.
d. zu einem späteren Zeitpunkt ein unsicheres System mit möglichst geringem Aufwand zu einem Sicherheitssystem umgebaut werden kann.

Um diese Dienstkontexte zu unterstützen, legt die OMG drei grundlegende Anforderungen fest:

1. Sobald von einem Objektdienst ein neuer Kontext erzeugt wird, muss er der IDL-Spezifikation genügen. Anderenfalls wird die Möglichkeit der interoperablen Dienste nicht Realität werden können und Punkt (3) wäre nicht standardisiert zu realisieren.
2. Jeder ORB muss eine Menge von Schnittstellen zur Verfügung stellen, die es allen Partizipierenden (Stub für den Klienten, Skeleton für den Server) erlaubt, kontextspezifische Informationen abzufragen bzw. zu setzen. Bis zu einem gewissen Grad kann dies auch aus der Applikation heraus geschehen.
3. Der ORB ist grundsätzlich für die Weitergabe von kontextspezifischen Informationen verantwortlich. Damit ist die Forderung erfüllt, dass Kontexte immer indirekt und implizit weitergegeben werden, d. h. sie ‚*schwimmen*' als Zusatzinformationen in den Anfrage- und Antwortnachrichten mit.

Für die implizite und indirekte Weitergabe von Kontexten schreibt die OMG die folgende IDL verbindlich vor, die von den jeweiligen Herstellern bzgl. der Implementierungssprache ihres ORBs zu übersetzen bzw. in den Quellkode einzubauen ist:

```
module IOP {
    ...
    typedef unsigned long ServiceId;
    struct ServiceContext {
        ServiceId             context_id;
        sequence<octet>       context_data;
    };
    typedef sequence<ServiceContext> ServiceContextList;
    ...
};
```

Programmbeispiel 18: Das Modul IOP

In den folgenden Subkapiteln werden die wichtigen Nachrichtenformate *Request*, *Reply*, *LocateRequest*, *LocateReply* und *CancelRequest* beschrieben.

4.1.6 Die Nachricht Request

Sobald ein Klient eine entfernte Methode aufruft, wird ein Anfrageobjekt auf der Serverseite erzeugt, in dem eine Reihe wichtiger Informationen enthalten sind.

Insgesamt besteht die Anfragenachricht aus den drei Teilen:
1. Dem GIOP Nachrichtenkopf.
2. Dem Nachrichtenkopf der Anfrage.
3. Dem Nachrichtenkörper der Anfrage mit den zu transportierenden Daten.

Der Nachrichtenkopf der Anfrage enthält spezifische Informationen darüber, welche Operation von welchem Objekt ausgeführt werden soll etc. Der Nachrichtenkopf, beschrieben in IDL für die GIOP Version 1.1, lautet:

```
struct RequestHeader_1_1 {
    IOP::ServiceContextList   service_context;
    unsigned long             request_id;
    boolean                   response_expected;
    octet                     reserved[3];
    sequence<octet>           object_key;
    string                    operation;
    Principal                 requesting_principal;
};
```

Programmbeispiel 19: Die Struktur RequestHeader

Die einzelnen Datenelemente haben die folgende Bedeutung:
1. *ServiceContext*: Enthält Informationen zu Objektdiensten wie z. B. einen Transaktionskontext oder Sicherheitsinformationen über den Klienten.
2. *RequestId*: Sobald ein Klient eine Nachricht vom Typ *Request* oder *LocateRequest* sendet, wird i. A. eine Antwort erwartet. Da sowohl Klient als auch Server multithreadingfähig sein können, ist es notwendig, jede Anfrage mit einem für diesen Klienten eindeutigen Bezeichner für die Rückantwort zu versehen. Da es sich hierbei um eine Klientenangelegenheit handelt, ist auch der Klient, bzw. der für ihn generierte Stub, für die Generierung des Bezeichners verantwortlich.
3. *Response Expected*: Dass von Seiten des Aufrufers eine Antwort erwartet wird, kann zusätzlich durch diese boolsche Variable angezeigt werden. Sinnvoll ist dieser Wert jedoch für Klienten, die keine Antwort erwarten und somit diesen Wert auf *FALSE* setzen. Damit ist die automatisch gesetzte RequestId außer Kraft gesetzt.
4. *Reserved*: Existiert seit der GIOP Version 1.1 und ist ggw. mit Nullen belegt. Dieses Datenelement ist für zukünftige Erweiterungen vorgesehen.
5. *Objektschlüssel*: Ein Klient arbeitet mit entfernten Objekten. Dies impliziert, dass er niemals direkt eine Methode aufruft, sondern er delegiert über eine geeignete Nachricht an ein serverseitiges Objekt, das für ihn die Methode implementiert hat und ausführt. Dieses Objekt residiert jedoch in einem Serverprozess, der durchaus eine Menge von Objekten enthalten kann. D. h. die Angabe in einer interoperablen Objektreferenz wie DNS-Name des Rechners und Portnummer des Serverprozesses reichen nicht aus, um ein Zielobjekt eindeutig zu beschreiben.
6. *Operation*: I. d. R. wird ein entferntes Objekt mehr als nur eine Operation zur Verfügung stellen. Um die richtige Methode zur Ausführung zu bringen wird genau diese Information in der Nachricht mittransportiert.
7. *RequestingPrincipal*: Ein Prinzipal ist in aller Allgemeinheit eine zu identifizierende Entität. I. d. R. wird es sich um eine gegenüber dem System authentifizierte Person handeln.

Es ist jedoch auch vorstellbar, dass es sich z. B. um eine Rolle, eine Gruppe von Personen oder um einen anderen Rechner handelt, der grundsätzlich autorisiert ist mit dem System zu arbeiten.

4.1.7 Die Nachricht Reply

Ein Klient, der eine Antwort von der Serverseite erwartet, weist der boolschen Variablen im Kopf der Anfrage den Wert *TRUE* zu. Die Nachricht vom Typ *Reply* ist eine Vereinfachung der Anfragestruktur. Die IDL Beschreibung für das GIOP-Protokoll in den Versionen 1.0 und 1.1 lautet:

```
enum ReplyStatusType {
    NO_EXCEPTION,
    USER_EXCEPTION,
    SYSTEM_EXCEPTION,
    LOCATION_FORWARD
};

struct ReplyHeader {
    IOP::ServiceContextList   service_context;
    unsigned long             request_id;
    ReplyStatusType           reply_status;
};
```

Programmbeispiel 20: ReplyStatusType und ReplyHeader

Die Datenelemente haben die folgende Bedeutung:
1. *Service Context*: Analog zu der Anfragestruktur.
2. *RequestId*: Analog zur Anfragestruktur.
3. *Reply Status*: War der entfernte Methodenaufruf erfolgreich, so wird der Rückgabestatus auf *NO_EXCEPTION* gesetzt. War der Aufruf nicht erfolgreich, so sind zwei Fälle zu unterscheiden:
   a. Der Aufruf wurde an ein anderes Objekt delegiert, in diesem Fall ist der Status mit dem Wert *LOCATION_FORWARD* versehen. Dies setzt voraus, dass der Server ein geeignetes Objekt kennt bzw. ermitteln kann und dass er die Möglichkeit der Delegation besitzt.
   b. Die Nutzdaten enthalten eine anwenderdefinierte oder eine Systemausnahme. In diesem Fall ist der Status auf *SYSTEM_EXCEPTION* oder auf *USER_EXCEPTION* gesetzt.

4.1.8 Die Nachricht CancelRequest

Der Nachrichtentyp *CancelRequest* hat einen extrem einfachen Aufbau. Neben dem obligatorischen GIOP-Nachrichtenkopf enthält dieser Typ lediglich den Identifizierer der Anfrage.

4.1.9 Die Nachrichtentypen LocateRequest und LocateReply

Der Sender einer Nachricht hat das Problem mit entfernten Objekten zu arbeiten und verfügt i. d. R. lediglich über das Wissen,
1. wie ein entferntes Objekt anzusprechen ist, d. h. welche Schnittstelle vereinbart wurde.
2. welche Signatur die einzelnen Operationen aufweisen.
3. welche Rückgabewerte zu erwarten sind.
4. welche Ausnahmen im Fehlerfall ausgelöst werden.

Des Weiteren verfügt der Klient über eine interoperable Objektreferenz, die er von einem Namensdienst oder einem Trader erfragt, bzw. er erhält diese Referenz in Form eines Strings und muss sie selbst in eine Objektreferenz konvertieren.

Eine Reihe von Problemen kann bei der entfernten Verarbeitung auftreten:
1. Das entfernte Objekt existiert nicht und somit ist die Objektreferenz ungültig.
2. Das Objekt existiert, ist aber nicht aktiv.
3. Das Objekt migrierte zur Laufzeit auf einen anderen Rechner u. U. in einer anderen Domäne.

Der Fall (2) ist einfach zu lösen. Ist das Objekt inaktiv, so ist es die Aufgabe der Serverseite, ein geeignetes Objekt zu aktivieren und dem Klienten zur Verfügung zu stellen.

In den Fällen (1) und (3) besitzt der Klient durch den Nachrichtentyp *LocateRequest* die Möglichkeit, Informationen über das entfernte Objekt abzufragen, also ob das Objekt noch zulässig ist oder ob eine alternative Adressierung im Falle der Objektmigration möglich ist.

Die Nachricht *LocateRequest* besteht aus dem GIOP-Nachrichtenkopf, dem Identifizierer einer möglichen Anfrage und dem Objektschlüssel. In IDL ergibt sich somit die folgende Beschreibung:

```
struct LocateRequestHeader {
    unsigned long      request_id;
    sequence<octet>    object_key;
};
```

Programmbeispiel 21: Die Struktur LocateRequestHeader

Die Antwort der Anfrage wird in dem Nachrichtentyp *LocateReply* an den Klienten geliefert. Bzgl. des Objektes sind zwei Fälle zu unterscheiden:
h. Das Objekt ist zustandslos und ggw. nicht aktiv. In diesem Fall wird ein Objekt dieses Typs instanziiert und der Klient kann seine Anfrage stellen.
i. Das Objekt ist zustandsbehaftet und befindet sich (zumindest bzgl. des CORBA 2.x Standards) im Rahmen einer Transaktion. Ist das Objekt dann nicht für den Klienten verfügbar, so liegt ein schwerwiegender Systemfehler vor.

Der Aufbau ist wie folgt:
1. Der obligatorische GIOP-Nachrichtenkopf.
2. Der Nachrichtenkopf von *LocateReply*, dessen IDL-Beschreibung lautet:

```
enum LocateStatusType {
    UNKNOWN_OBJECT, OBJECT_HERE, OBJECT_FORWARD
};

struct LocateReplyHeader {
    unsigned long         request_id;
    LocateStatusType      locate_status;
};
```

Programmbeispiel 22: LocateStatusType und LocateReplyHeader

Der Status hat die folgende Bedeutung:
a. *UNKNOWN_OBJECT*: Die Serverseite kennt das spezifizierte Objekt nicht. In diesem Fall sind die Nutzdaten der Antwort leer.
b. *OBJECT_HERE*: Der Server hat das spezifizierte Objekt lokalisiert und kann Anfragen bzgl. dieses Objektes entgegennehmen.

   c. *OBJECT-FORWARD*: Das Objekt wurde zur Laufzeit migriert, seine Lokation ist jedoch bekannt. In diesem Fall enthalten die Nutzdaten der Antwort die neue Objektreferenz des migrierten Objektes.
3. Die Nutzdaten von *LocateReply*, die entweder leer sind, oder die neue interoperable Objektreferenz des gefundenen Objektes enthalten.

5 Die Objektdienste von CORBA

Die CORBA Standard Services sind eine Sammlung von Diensten zur Lösung von Problemen, die bei der Programmierung mit verteilten Objektsystemen immer wieder auftreten. Empfehlenswerte Literatur zu diesem Thema findet sich in [OMG, 1995b], [Vinosky et. al. 1999] und [Orfali, 1996].

Die bisher standardisierten Services sind:

1. Naming Service: Dieser Dienst ermöglicht einem dienstanfordernden Objekt, durch Vorlage eines logischen Namens eine Referenz eines diensterbringenden Objekts zu erhalten. Der logische Name identifiziert dabei das diensterbringende Objekt. Ein Naming Service gehört zur Grundausstattung einer CORBA-Implementierung und wurde daher von fast jedem Hersteller implementiert.
2. Lifecycle Service: Im Rahmen dieses Dienstes werden abstrakte Schnittstellen für die Operationen *Erzeugen*, *Kopieren*, *Verschieben* und *Löschen* von Objekten definiert. Die entsprechenden Operationen müssen in den Objekten selbst implementiert werden. Ein besonders wichtiger Teil dieses Dienstes ist der FactoryFinder. Es wird hier mit dem Design-Pattern Factory gearbeitet. Diese Fabrikobjekte sind zuständig für das Erzeugen von Objekten und der hierzu korrespondierenden Objektreferenzen, die an den Klienten übermittelt werden. Damit ist ein Fabrikobjekt grundsätzlich für ein Lastverteilungsverfahren prädestiniert, das in der Literatur als *factory based routing* bezeichnet wird. Der Lifecycle-Service wurde bisher wenig implementiert.
3. Event Management Service: Dieser Ereignisdienst bietet Objekten die Möglichkeit, sich bei einem sog. Eventchannel registrieren zu lassen. Wird nun von einem ereigniserzeugenden Objekt das Auftreten eines Ereignisses in den Eventchannel eingespeist, so informiert dieser alle registrierten Objekte darüber. Ein Event Management Service gehört, ähnlich wie der Naming Service, zur Grundausstattung einer CORBA-Implementierung.
4. Persistent Object Service: Dieser Dienst unterstützt die Speicherung von Objekten. Dabei kommen sowohl relationale als auch objektorientierte Datenbanken sowie ein Dateisystem als Datenspeicher in Betracht.
5. Relationship Service: Normalerweise werden Beziehungen zwischen Objekten durch Referenzen implementiert, d. h. ein Objekt kennt eine Referenz auf das Objekt, mit dem es in Beziehung steht. Solche Beziehungen haben zwei Nachteile: Sie sind nicht attributierbar und zu ihrer Definition muss grundsätzlich der Objekttyp geändert werden. Der Relationship Service bietet eine Möglichkeit zur Definition von Objektbeziehungen ohne diese Nachteile. Mit Hilfe des Relationship Service können nicht nur Beziehungen zwischen jeweils zwei Objekten, sondern ganze Beziehungsgraphen verwaltet werden. Dieser Dienst ermöglicht außerdem die Definition zusammengesetzter Objekte.
6. Externalisation Service: Im Rahmen dieses Dienstes können komplexe Objekte in einen Stream transformiert werden, der von anderen Softwarekomponenten weiterverarbeitet werden kann. Ein anderer ORB könnte ein so transformiertes Objekt beispielsweise einlesen und wieder aufbauen.
7. Concurrency Control Service: Der Concurrency Control Service erlaubt es Objekten, ihren Zugriff auf gemeinsam genutzte Ressourcen zu koordinieren. Dies geschieht, indem jedes Objekt von diesem Dienst eine Sperre der benötigten Ressource anfordern kann und diese erst benutzen darf, wenn es die Sperre erhalten hat. Eine Sperre gilt dabei immer für eine Ressource und ein Klientenobjekt.
8. Transaction Service: Mittels dieses Dienstes ist ein Transaktionsmanagement abbildbar. In einem verteilten Objektsystem ist es notwendig, eine Transaktion von dem Klienten ausgehend, der sie auslöst, über evtl. mehrere Server und wieder zurück zum Client zu managen. Der standardisierte Transaction Service definiert die hierzu erforderlichen Schnittstellen sowohl für flache als auch für verteilte Transaktionen. Der Transaction Service ist nicht für alle CORBA-Implementierungen erwerbbar. Die bekannten Lösungen

realisieren diesen Dienst, indem ein *Transaction Processing Monitor* in die CORBA-Implementierung integriert wird.

9. Query Service: Mit Hilfe dieses Dienstes kann eine Anfrage an eine Menge von Objekten gestellt werden. Ergebnis einer solchen Anfrage ist z. B. eine Teilmenge der ursprünglichen Objektmenge, bei denen ein bestimmtes Attribut einen bestimmten Wert hat. Der Query Service ermöglicht Anfragen an eine Objektmenge in der Art, wie SQL eine Anfrage an eine relationale Datenbank ermöglicht. Als Anfragesprache muss ein Query Service SQL oder OQL unterstützen.
10. Licensing Service: Dieser Dienst bietet die Möglichkeit, Objekte gegen nichtlizensierte Benutzung zu schützen. Er bietet keine Sicherheitsfunktionalität in der vom Projekt benötigten Art und Weise, sondern ist vielmehr ein Mittel für Softwarehersteller, die unkontrollierte Benutzung Ihrer Produkte zu verhindern.
11. Change Management Service: Mit Hilfe dieses Service ist es möglich abzusichern, dass Objekte immer die gleiche Version einer Objektimplementierung benutzen, unabhängig davon, ob bereits Folgeversionen dieser Objektimplementierung existieren. Der Dienst ist zu Zeit noch selten implementiert: Die Versionsproblematik tritt im Lebenslauf eines Softwareproduktes naturgemäß erst in einem nachgelagerten Stadium auf. Es wurden außerdem Methoden etabliert, um die Grundfunktionalität des Change Management Service in anderer Weise bereitzustellen (z. B. durch Kodierung der Versionsnummer im Schnittstellennamen).
12. Property Service: Der Property-Dienst erlaubt es, dynamisch zur Laufzeit Attribute mit einem Objekt zu assoziieren. Es ist möglich, solche Attribute mit einem Namen zu versehen, ihre Werte zu setzen und zu lesen, sowie sie wieder zu löschen. Des Weiteren ist es möglich, durch Gruppen von Properties zu navigieren. Dies geschieht durch die Bereitstellung von Iteratoren für Properties. Der Property Service ist manchmal ein nützliches Instrument bei der objektorientierten Softwareentwicklung.
13. Security Service: Mittels dieses Dienstes soll die Implementierung von Datenschutz und Zugriffsrechten vereinfacht werden. Im Einzelnen werden Lösungen für die Problembereiche Authentisierung, Autorisierung und Auditing-Funktionalität in unterschiedlicher Granularität angeboten. Das bedeutet, dass z. B. auf Objektebene ebenso Sicherheitsmechanismen implementiert werden können wie auf Gruppen von Objekten.
14. Trading Service: Der Trading Service stellt einen alternativen Naming Service dar: Objektreferenzen werden nicht durch Angabe eines logischen Objektnamens, sondern auf der Basis von Signaturen exportierter Operationen gefunden.
15. Collection Service: Behandlung von Mengen, Multimengen und Listen.
16. Time Service: Dieser Dienst unterstützt die Synchronisation beteiligter Zeitgeber. Bei einer regional begrenzten Anwendung ist ein solcher Dienst nicht notwendig. Stattdessen kann ein Funkuhrsignal zur Uhrensynchronisation verwendet werden.

Zum ggw. Zeitpunkt existiert kein CORBA-Produkt, das alle Dienste zur Verfügung stellt. In den folgenden Unterkapiteln werden die wichtigsten Dienste im Detail vorgestellt.

5.1 Der Naming Service

Der von der OMG spezifizierte Namensdienst ist der einfachste der standardisierten CORBA-Dienste und ist in allen bekannten und gängigen CORBA-Produkten implementiert. Zur Verfügung gestellt wird eine Abbildung von Namen auf Objektreferenzen.

Ein Namensdienst ist organisiert als eine Baumstruktur. Die Kanten des Baumes sind die so genannten Namenskomponenten, also Kontexte oder Subkontexte, und die Blätter des Baumes sind die Objektreferenzen. Die Wurzel des Baumes wird häufig als initialer Kontext bezeichnet.

In Abbildung 12 ist ein imaginärer Baum dargestellt, der es dem Klienten erlaubt, über die Namenskomponente *Flug.Lufthansa* die Objektreferenz *Auskunft* zu bekommen und über diese Referenz eine entfernte Anfrage zu stellen.

Der Namensdienst kann, bei entsprechender Berechtigung, zur Laufzeit modifiziert werden. Dazu zählen Methoden zum
a. Entfernen von Objektreferenzen oder Teilbäumen.
b. Hinzufügen von Objektreferenzen oder Teilbäumen.
c. Ersetzen bestehender Objektreferenzen durch Neue.

Innerhalb eines Kontextes muss der Name einer Objektreferenz eindeutig sein. Dies entspricht unserem Verständnis von dem hierarchischen Aufbau eines Dateisystems unter UNIX oder Windows NT. Das Verzeichnis */opt/tuxedo* ist in diesem Kontext genauso eindeutig wie die Datei *C:\autoexec.bat*. In verschiedenen Kontexten ist es natürlich möglich, eine Objektreferenz mehrfach einzustellen.

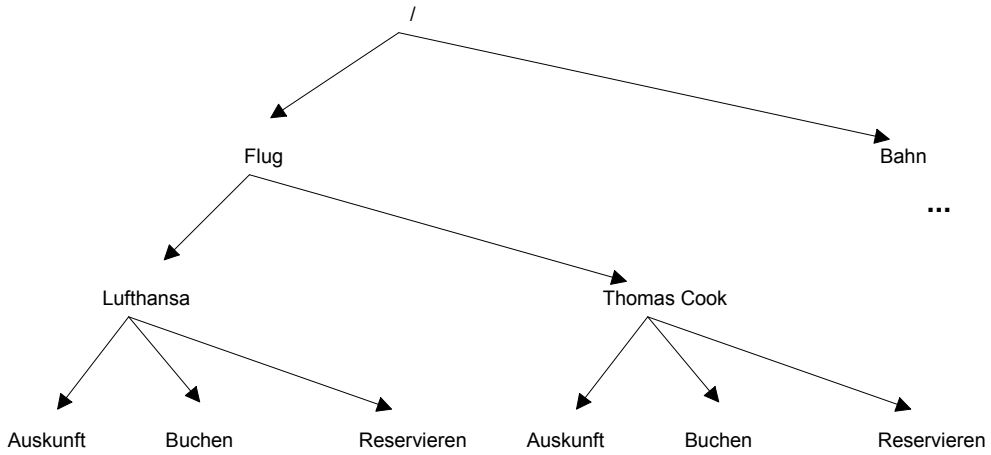


Abbildung 12: Beispiel eines Namensdienstes

Der Namensdienst wird beschrieben durch eine IDL-Datei Namens *CosNaming.idl*, wobei der Präfix Cos für CORBA Object Services steht. In dieser IDL-Vereinbarung wird ein Namensraum *CosNaming* definiert, der neben einer Reihe von Strukturen die beiden Schnittstellen *NamingContext* und *BindingIterator* deklariert.

```
#pragma prefix "org.omg"

module CosNaming {
    // Typdefinitionen

    interface NamingContext { ... };

interface BindingIterator { ... };
};
```

Programmbeispiel 23: Skeleton des Moduls CosNaming

Ein Name im Namensdienst wird durch die folgende in IDL definierte Struktur beschrieben:

```
typedef string Istring;

struct NameComponent {
    Istring id;
    Istring kind;
};
```

```
typedef sequence<NameComponent> Name;
```

Programmbeispiel 24: NameComponent und Name

Anhand dieser Definition ist es in Anlehnung an die obige Beschreibung wichtig festzuhalten, dass

a. zwei Namenskomponenten genau dann äquivalent sind, wenn das *id*- und das *kind*-Feld identisch sind.
b. zwei Namen genau dann äquivalent sind, wenn sie in ihren Namenskomponenten äquivalent sind.

Bei dem Umgang mit dem Namensdienst können Fehler auftreten. Dazu definiert die OMG in der Datei *CosNaming.idl* entsprechende Ausnahmen.

5.1.1 Mögliche Ausnahmen des Namensdienstes

Von der OMG werden eine Reihe von Ausnahmen definiert, die im Umgang mit dem Namensdienst ausgelöst werden können. Die IDL-Beschreibung der insgesamt fünf Ausnahmen ist:

```
module CosNaming {
    // ...

    interface NamingContext {
        enum NotFoundReason {
               missing_node, not_context, not_found_object
        };

        exception NotFound {
               NotFoundReason why;
               Name           rest_of_name;
        };
        exception CannotProceed {
               NamingContext  ctx;
               Name           rest_of_name;
        };
        exception InvalidName {};
        exception AlreadyBound{};
        exception NotEmpty {};
               ...
    };
};
```

Programmbeispiel 25: CORBA-CosNaming

Die Bedeutung dieser Ausnahmen ist die folgende:

1. *NotFound*: Diese Ausnahme wird ausgelöst, wenn ein Klient eine Objektreferenz anfordert, für die es keine existierende Bindung gibt. Die Ausnahme enthält zwei Datenelemente: Eine Information vom Aufzählungstyp *NotFoundReason* mit der folgenden Bedeutung:
   a. Der Grund *missing_node*: Eine der Namenskomponenten beschreibt eine Bindung, die nicht existiert.
   b. Der Grund *not_context*: Eine der Namenskomponenten beschreibt die Bindung an ein Applikationsobjekt und nicht an einen (Sub-) Kontext.
   c. Der Grund *not_found_object*: Eine der Namenskomponenten spezifiziert die Namenskomponente einer Objektreferenz, die nicht mehr gültig ist.

   Das Element *rest_of_name* liefert den nachfolgenden Teil des Namens, der nicht aufgelöst werden konnte.
2. *CannotProceed*: Diese Ausnahme indiziert, dass der Namensdienst die gestellte Anfrage nicht weiter verfolgen kann. Diese Ausnahme enthält ebenfalls zwei Datenelemente:

   a. Das Datenelement *ctx* ist die Objektreferenz, die die erste nicht auflösbare Bindung enthält.
   b. Der Grund *rest_of_name* besitzt die gleiche Bedeutung wie oben.
3. *InvalidName*: Diese Ausnahme wird ausgelöst, wenn versucht wird, einen leeren Namen, also eine leere Folge von Namenskomponenten, aufzulösen.
4. *AlreadyBound*: Diese Ausnahme wird ausgelöst, wenn versucht wird, eine bestehende Bindung neu zu setzen. Aufgrund der Eindeutigkeit eines Namens innerhalb eines Kontextes ist dies nur möglich, wenn eine bestehende Bindung gelöscht und durch eine neue Bindung ersetzt wird.
5. *NotEmpty*: Diese Ausnahme wird ausgelöst, wenn ein Kontext gelöscht werden soll, der noch bestehende und gültige Bindungen enthält.

5.1.2 Verbinden zum Namensdienst

Ein Klient verbindet sich zum Namensdienst durch die Methode *resolve_initial_references* unter Angabe des konstanten Wortes *NameService*. Diese Methode wird über das bei der Initialisierung des Klientenorbs instanziierte lokale Objekt *orb* aufgerufen

```
org.omg.CORBA.Object o = orb.resolve_initial_references ("NameService");
```

und muss die anschließend erhaltene Objektreferenz vom Typ `org.omg.CORBA.Object` an den richtigen Typ anpassen

```
CosNaming.NamingContext n = CosNaming.NamingContextHelper.narrow(o);
```

Die Methode *resolve_initial_references* wird auch für die Anforderung anderer Dienste wie z. B. dem Transaktionsdienst oder den Sicherheitsdienst verwendet und kann, falls der Dienst nicht implementiert oder nicht aktiv ist, auch eine nil-Referenz an den Aufrufer zurückgeben. Welche Dienste implementiert sind und über welchen konstanten String sie angefordert werden, kann über die Methode

```
ObjectList ol = orb.list_initial_services();
```

erfragt werden. Hierbei ist *ObjectList* definiert durch

```
typedef string ObjectId;
typedef sequence<ObjectId> ObjectList;
```

5.1.3 Modifikation des Namensdienstes

Ein Namendienst kann bei entsprechender Berechtigung zur Laufzeit modifiziert werden. Dazu zählen Methoden zum Erzeugen neuer Namenskontexte und Methoden zum Binden einer Objektreferenz an einen Namen, die Löschung dieser Referenz oder das Binden einer neuen Referenz an einen bestehenden Namen. Die Methoden sind in der Schnittstelle *CosNaming::NamingContext* deklariert. In IDL definiert erhält man

```
interface NamingContext {
    void       bind(in Name n, in Object o)
                       raises( NotFound, CannotProceed, InvalidName, AlreadyBound);
    void       bind_context(in Name n, in NamingContext nc)
                       raises( NotFound, CannotProceed, InvalidName, AlreadyBound);
    void       rebind(in Name n, in Object o)
                       raises(NotFound, CannotProceed, InvalidName);
    void       rebind_context(in Name n, in NamingContext nc)
                       raises(NotFound, CannotProceed, InvalidName);
    void       unbind(in Name n)
                       raises(NotFound, CannotProceed, InvalidName);
    Object     resolve(in Name n)
                       raises(NotFound, CannotProceed, InvalidName);
};
```

Programmbeispiel 26: NamingContext

Die Bedeutung dieser Methoden ist die Folgende:

1. *bind()*: Mit dieser Methode wird eine Objektreferenz an einen Namen, ggf. in einem Namenskontext, gebunden.
2. *bind_context()*: Es wird ein neuer Teilbaum aufgebaut. Dieser Teilbaum ist beschrieben durch die Komponente *Name* und wird unterhalb des durch *NamingContext* angegebenen Teilbaums erzeugt.
3. *rebind()*: Eine neue Objektreferenz wird an einen bereits vorhandenen Namen gebunden. Diese entspricht dem Vorgang *unbind*, gefolgt von *bind*.
4. *rebind_context()*: Analog zu *rebind* für Namenskomponenten.
5. *unbind()*: Ein Objekt, das an den übergebenen Namen gebunden ist, wird aus dem Namensdienst entfernt.
6. *resolve()*: Ein Klient baut einen Namenskontext auf, gefolgt von dem Namen der gewünschten Objektreferenz und lässt sich anhand dieser Information vom Namensdienst die hierzu korrespondierende Objektreferenz geben.

5.2 LifeCycle

Der LifeCycle-Dienst ist mit Ausnahme von BEAs CORBA-Produkt WebLogic Enterprise i. d. R. nicht implementiert. Von besonderem Interesse ist in diesem Kapitel ein Teil dieses Dienstes, der *FactoryFinder*.

Bei Verwendung dieses Teildienstes wird nach dem Designpattern *Fabrik* (Factory-Pattern) gearbeitet. Eine Fabrik ist zuständig für das Erzeugen eines Objektes auf der Serverseite und die Rückgabe der hierzu korrespondierenden Objektreferenz an den Klienten.

Für die Verwendung von Fabrikobjekten sind primär zwei Gründe zu benennen:
1. Das Objekt benötigt kein zusätzliches Wissen über anwendungsspezifische Details, die nicht zu seiner Geschäftslogik gehören. Bspw. ist es bei einer Anwendung im Bereich HomeBanking nicht erforderlich, dass ein Objekt *Bankkonto* weiß, wie ein Bankkonto erzeugt, gelöscht oder gefunden wird. Das Objekt weiß, basierend auf seiner Geschäftslogik, wie von dem Konto ein Betrag abgehoben oder gutgeschrieben wird und wie dem Klienten Umsätze eines vorgegebenen Zeitraums angezeigt werden. Die Aufgabe, anhand einer Kontonummer ein geeignetes Objekt zu instanziieren, wird an ein Objekt *BankkontoFactory* delegiert.
2. Das Fabrikobjekt kann anhand einer vom Klienten gelieferten Information ein Lastverteilungskriterium in die zu generierende Objektreferenz schreiben. Dieser Vorgang wird i. A. als *factory based routing* bezeichnet.

Bei dem Prinzip des *factory based routing* wird von dem Klienten eine zusätzliche Information an die Fabrik weitergegeben, anhand derer ein Lastverteilungskriterium bestimmt und bei der Erzeugung der Objektreferenz in den herstellerspezifischen Teil der interoperablen Referenz geschrieben wird. Dieser Teil ist lediglich für die Serverseite von Interesse und auswertbar.

Für den Klienten bedeutet die Verwendung eines FactoryFinders eine andere Vorgehensweise, um mit entfernten Objekten arbeiten zu können. Anhand der folgenden IDL-Vereinbarung wird dies illustriert:

5.2.1 Die IDL-Beschreibung für den Zugang zu den Fabriken

Die folgende IDL beschreibt eine stark vereinfachte Vereinbarung für eine Bankanwendung:

Vereinbart wird eine Ausnahme *DispositionskreditUeberzogen*, die ausgelöst werden soll, wenn der Abheber eines Betrages seinen Kreditrahmen überzogen hat. Die Struktur *Um-*

satz besteht aus einem Datum, einem Betrag, der zu diesem Zeitpunkt eingezahlt oder abgehoben wurde und einer Zusatzinformation, die z. B. den Überweisungsträger beschreiben kann. Ob eine Einzahlung vorliegt, wird durch die boolsche Variable *eingezahlt = true* beschrieben. Bei einer Abhebung ist dieser Wert auf *false* gesetzt. Bei der Umsatzabfrage ist ein Zeitraum anzugeben, wobei zu jedem Datum die Transaktionen der Kontenbewegung anzuzeigen sind. Für die Struktur *Umsatz* wird demzufolge eine Sequenz von Einträgen definiert, die für jeden Tag eine Menge von Ein- und Auszahlungen beschreiben kann und eine zweite Sequenz, die das Zeitintervall und alle Ereignisse wiedergibt.

Die Schnittstelle *Bankkonto* besteht lediglich aus drei Methoden, mit denen das Einzahlen und Abheben eines Betrages möglich ist, sowie einer Methode, die es erlaubt die Umsätze in einem bestimmten Zeitraum einzusehen.

Die Schnittstelle *BankkontoFactory* besteht aus einer Methode *create*, mit der ein neues Bankkonto angelegt werden kann und einer Methode *findByPrimaryKey*, mit der ein existierendes Konto gefunden werden kann. Beide Methoden liefern eine Objektreferenz an den Aufrufer zurück, die vom Typ *Bankkonto* ist und somit nicht mehr an den richtigen Typ angepasst werden muss.

Die IDL-Beschreibung lautet:

```
exception DispositionskreditUeberzogen {};

struct Umsatz {
    Datum      datum;
    double     betrag;
    string     zusatzinformation;
    boolean    eingezahlt;
};
typedef sequence<Umsatz> Umsaetze;          // Umsätze eines Tages
typedef sequence<Umsaetze> UmsaetzeSeq;     // Umsätze eines Zeitintervalles


interface Bankkonto {
    void               abheben(in double betrag)
                              raises(DispositionskreditUeberzogen);
    void               gutschreiben(in double betrag);
    UmsaetzeSeq        umsaetze(in Datum von, in Datum bis);
};

interface BankkontoFactory {
    Bankkonto create(in long kontonummer);
    Bankkonto findByPrimaryKey(in KontoPK p);
};
```

Programmbeispiel 27: Factory- und Schnittstellenbeschreibung in IDL

5.2.2 Der Zugang des Klienten

Basierend auf der o. a. IDL-Beschreibung werden die Arbeitsschritte des Klienten beschrieben, um über eine Fabrik gehend eine Objektreferenz zu dem gewünschten Objekt zu erhalten. Es wird vorausgesetzt, dass der Klient seinen ORB initialisiert hat und somit über das Pseudoobjekt *orb* verfügt und dass der Klient über ein existierendes Konto mit der Kontonummer *4711* verfügt.

Die auszuführenden Schritte für den Klienten sind:

1. Über das bei der Initialisierung erhaltene Objekt *orb* wird über die Methode *resolve_initial_references* der Dienst *FactoryFinder* angefordert.
   ```
   org.omg.CORBA.Object obj = orb.resolve_initial_references("FactoryFinder");
   ```

2. Das Resultat des o. a. Aufrufes ist vom Typ *Object*, der Basisklasse aller CORBA-Objekte und muss demzufolge an den richtigen Typ angepasst werden. Bei Verwendung der Sprache Java ist diese Anpassung durch die Methode *narrow* aus der Helperklasse, die für den FactoryFinder existieren muss, vorzunehmen.
   ```
   FactoryFinder f = FactoryFinderHelper.narrow(obj);
   ```
3. Der FactoryFinder verfügt laut Spezifikation über eine Methode *find_factories*. Für das BEA CORBA-Produkt WLE wurden weitere Methoden zum Auffinden von Fabrikobjekten hinzugefügt. In diesem Beispiel wird die Methode *find_one_factory_by_id* verwendet. Da Objekte in verschiedenen Versionen auf der Serverseite existieren können und um Namenskonflikte zu vermeiden, wird als Argument der vollständig qualifizierte Repository-Name des Fabrikobjektes übergeben. Bei Verwendung der Sprache Java existiert hierzu in der Helperklasse, die für das Fabrikobjekt vom IDL-Compiler generiert wurde, eine Methode *id*, die das Gewünschte leistet.
   ```
   org.omg.CORBA.Object o = f.find_one_factory_by_id(BankkontoFactoryHelper.id());
   ```
4. Der Rückgabewert ist wieder vom Typ *Object* und muss entsprechend an den richtigen Typ angepasst werden.
   ```
   BankkontoFactory bf = BankkontoFactoryHelper.narrow(o);
   ```
5. Nach der Anpassung an den richtigen Typ kann der Klient über das Fabrikobjekt die Erzeugung eines serverseitigen Objektes veranlassen und wird als Rückgabewert eine Referenz auf dieses Objekt zurückerhalten.
   ```
   Bankkonto b = bf.findByPrimaryKey(4711);
   ```
   Anschließend werden über diese Objektreferenz die Methoden des Objektes wie in CORBA üblich genutzt.

5.3 Der Event Management Service (Ereignisdienst)

Bei einem synchronen, entfernten Methodenaufruf wird der Aufrufer solange blockieren, bis er entweder ein Ergebnis erhält oder der Aufruf aufgrund eines gesetzten Zeitwertes terminiert wird. In der Praxis sind jedoch eine Reihe von Situationen vorstellbar, in denen ein Klient lediglich ein Datum für eine Verarbeitung einstellen will und anschließend andere Aufgaben wahrnimmt. Sobald die Serverseite die Verarbeitung abgeschlossen hat, möchte der Klient über dieses Ereignis informiert werden und das Verarbeitungsergebnis anfordern.

Unter Verwendung eines Ereignisdienstes kann diese Aufgabe bewerkstelligt werden. Ein Klient übergibt asynchron eine Nachricht an die Serverseite und hat gegenüber dem Ereignisdienst sein Interesse an genau diesem Ereignis bekundet. Die Serverseite ist ebenfalls derart zu entwickeln, dass ein Server bei Beendigung der Verarbeitung diesen Dienst über dieses Ereignis informiert. Die Beteiligten sind voneinander entkoppelt.

Definition 28: Produzenten/Konsumenten von Ereignissen

1. Der Produzent eines Ereignisses ist der *Erzeuger* oder *Produzent* (*supplier*) dieses Ereignisses.
2. Der *Konsument* (*receiver*) eines Ereignisses erhält dieses Ereignis.

Die beiden Beteiligten sind durch eine virtuelle Beziehung, über den Ereignisdienst gehend, miteinander verbunden.

Der von der OMG standardisierte Dienst unterstützt zwei Modelle für die Weitergabe der Ereignisse:

1. Das push-Modell: Der Produzent eines Ereignisses informiert den Ereignisdienst und dieser gibt dieses Ereignis an den Konsumenten weiter.

Abbildung 13: Das push-Modell

2. Das pull-Modell: Bei diesem Modell fragt der Konsument bei dem Ereignisdienst nach, ob das von ihm registrierte Ereignis eingetreten ist und der Dienst gibt diese Anfrage an das Serverobjekt, also den eigentlichen Produzenten dieses Ereignisses, weiter, um anschließend den Anfrager über den Sachstand zu informieren.

Abbildung 14: Das pull-Modell

Der Ereignisdienst ist, wie im CORBA-Umfeld üblich, in der Beschreibungssprache IDL beschrieben. Für das push-Modell lautet die Beschreibung:

```
module CosEventComm {
    exception Disconnected {};

    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };

    interface PushSupplier {
        void disconnect_push_supplier();
    };
};
```

Programmbeispiel 28: CosEventComm für das Push-Modell in IDL

Der Konsument eines Ereignisses implementiert die o. a. Schnittstelle *PushConsumer* und registriert eine Objektreferenz für den Produzenten eines Ereignisses. Dieser Produzent verwendet diese Objektreferenz um Daten und/oder Ereignisse zu dem implementierten Objekt vom Typ *PushConsumer* via die Methode *push* zu senden. Beide Beteiligte können die Verbindung vorzeitig trennen. Der Produzent eines Ereignisses verwendet hierzu die Methode *disconnect_push_consumer* und der Konsument ruft die Methode *disconnect_push_supplier* auf.

Bei Verwendung des pull-Modells gilt die IDL-Beschreibung:

```
module CosEventComm {
    interface PullSupplier {
        any    pull() raises(Disconnected);
        any    try_pull(out boolean has_event) raises(Disconnected);
        void   disconnect_pull_consumer();
    };

    interface PullConsumer {
        void disconnect_pull_consumer();
    };
};
```

Programmbeispiel 29: CosEventComm für das Pull-Modell in IDL

Ein Konsument kann von dem Erzeuger auf eine von zwei Möglichkeiten ein Ereignis erfragen:
1. Der Konsument ruft die Methode *pull* auf und blockiert solange, bis ein Ereignis im Sinne der angefragten Daten geliefert wird.
2. Der Konsument verwendet die Methode *try_pull* um in dem zurückgelieferten booleschen Wert zu erfahren, ob das von ihm gewünschte Ergebnis vorliegt. Bei diesem Aufruf findet eine Blockierung des Aufrufers nicht statt. Ist das Ergebnis vorhanden, so wird diese Methode zusätzlich zu dem boolschen Wert *true* das Ergebnis als Rückgabewert liefern.

Es ist wichtig, besonders darauf hinzuweisen, dass der Ereignisdienst die Ergebnisse über den dynamischen Datentyp *any* an den Aufrufer liefert. Eine detaillierte Beschreibung dieses Datentyps wird in nachfolgenden Kapiteln vorgenommen.

5.4 Der Transaktionsdienst

Der Transaktionsdienst ist ebenfalls von der OMG durch eine IDL-Beschreibung definiert worden. Die Verwendung dieses Dienstes wird durch eine kleine Menge von Methoden durch den Entwickler über ein anzuforderndes *TransactionCurrent*-Objekt wahrgenommen. Dieses Objekt erhält man i. A. über das von der ORB-Initialisierung zurückgelieferte Objekt *orb*, durch Aufruf der Methode

```
org.omg.CORBA.Object o = orb.resolve_initial_references ("TransactionCurrent");
```

und einer anschließenden Anpassung an den richtigen Typ.

Die wichtigsten Methoden sind:
1. *begin()*: Mit dieser Methode wird eine Transaktion initiiert.
2. *commit()*: Eine vorher initiierte Transaktion wird erfolgreich beendet.
3. *rollback()*: Der alte Zustand aller an der Transaktion Beteiligten wird wieder hergestellt.

Es existieren weitere Methoden, um eine Transaktion zu kontrollieren bzw. einer Transaktion bestimmte Verhaltensmuster mitzugeben.

Bei einer Betrachtung von Transaktionen wird i. A. unterschieden zwischen einer globalen Transaktion und Datenbanktransaktionen. Eine globale Transaktion beinhaltet eine Kette von Prozessen/Objekten, wobei derjenige, der die Transaktionsklammer setzt, die restlichen Beteiligten i. A. nicht kennt. Die globale Transaktion muss i. d. R. in eine Datenbanktransaktion überführt werden.

Für beide Formen von Transaktionen gelten bestimmte Charakteristiken.

Definition 29: Die ACID-Eigenschaft
Eine Transaktion muss die ACID-Eigenschaften erfüllen.

Hierbei steht das

- **A** für die Atomizität. D. h., man lässt eine Menge von Operationen so erscheinen, als wäre es genau eine einzige, nicht unterbrechbare Einheit. In der Literatur wird diesbezüglich von einem „*Alles oder nichts*“ Ansatz gesprochen.
- **C** für die Konsistenz (Consistence). Damit ist zum einen die Konsistenz der Daten einer Datenbank gemeint, zum anderen ist hier auch die konsistente Überführung des Transaktionskontextes von einem Partizipierenden zum anderen zu sehen. Dies ist insbesondere eine notwendige Forderung, da Transaktionskontexte von einem Transaktionsmanager, der Teil des Transaktionsmonitors ist, indirekt und implizit weitergegeben werden.
- **I** für die Isolation. Tatsächlich werden mehrere Transaktionen parallel ablaufen, es muss jedoch sichergestellt sein, das kein irreguläres Verhalten auftritt. Dies bedeutet, dass der Transaktionsmanager eine buchhalterische Aufgabe übernimmt und alle Prozesse/Objekte, die Teil einer Transaktion sind entsprechend sperrt. Aus Sicht des Anwenders vermittelt die Isolation den Eindruck, dass lediglich eine Transaktion zu einem Zeitpunkt ausgeführt wird und eine nachfolgende Transaktion erst gestartet wird, wenn der Vorgänger beendet ist.
- **D** für die Dauerhaftigkeit. Nach erfolgreicher Beendigung einer Transaktion soll das erzielte Ergebnis dauerhaft, also persistent, in ein Medium geschrieben werden.

Mit der Schilderung der ACID-Eigenschaften ergeben sich eine Reihe von Rollen, die von den jeweils an einer Transaktion Beteiligten, wahrgenommen werden.

Den einzelnen Partizipierenden einer Transaktion kommen i. A. unterschiedliche Rollen zu:

Definition 30: Transaktionaler Klient

Unter einem *transaktionalen Klienten* versteht man denjenigen, der eine Transaktion initiiert.

Dazu existiert in CORBA ein Pseudoobjekt *TransactionCurrent*, das Primitive wie z. B. die Methode *begin()* zur Verfügung stellt. Der transaktionale Klient ist derjenige, der durch die Primitive *commit()* bzw. *rollback()* die von ihm initiierte Transaktion beendet.

Definition 31: Transaktionaler Server

Ein *transaktionaler Server* ist ein Prozess/Objekt, der/das im Rahmen einer initiierten Transaktion nach vorgegebenen Regeln einen Geschäftsprozess abarbeitet.

Der transaktionale Server ist nicht zuständig für die Einleitung einer erfolgreichen Beendigung der Transaktion. Ebenso wenig ist es seine Aufgabe, einen bestehenden Zustand in ein anderes Medium zu schreiben bzw. alte Zustände bei nicht erfolgreichem Abschluss der Transaktion wieder herzustellen. Seine Aufgabe ist in der Abarbeitung von Regeln für einen Geschäftsprozess zu sorgen und ggf. anhand seiner Geschäftslogik die Partizipierenden zu einem Rücksetzen des alten Zustandes zu zwingen.

Definition 32: Wiederherstellbarer (recoverable) Server

Ein *widerherstellbarer Server* (recoverable Server) ist ein Prozess/Objekt,

a. der/das für das Schreiben eines neuen Zustandes in ein anderes Medium zuständig ist.
b. der/das für die Wiederherstellung des alten Zustandes zuständig ist, falls die Transaktion nicht erfolgreich abgeschlossen wurde.

Transaktionskontexte, also der Zustand und die hierzu korrespondierenden Informationen über eine Transaktion, werden immer indirekt und implizit an die Partizipierenden weitergegeben. Abbildung 15 zeigt das Zusammenspiel der jeweiligen Transaktionsteilnehmer. Der Transaktionsdienst, sowie die Verwaltung und der Zugriff auf den Transaktionskontext, sind für den äußeren Betrachter völlig transparent.

Die einzelnen Rollen werden in Abbildung 15 dargestellt. Das Szenario für diese Abbildung kann wie folgt aussehen:

1. Der Klient fordert über die CORBA-Methode
   ```
   org.omg.CORBA.Object obj = orb.resolve_initial_references ("TransactionCurrent");
   ```
   das so genannte TransactionCurrent-Objekt, das lokal im Adressraum des Klienten instanziiert wird, an und passt es mittels
   ```
   TransactionCurrent trans = TransactionCurrentHelper.narrow(obj);
   ```
   an den richtigen Typ an. *TransactionCurrent* und *TransactionCurrentHelper* sind Klassen aus dem CORBA-Umfeld und sind somit Bestandteil einer jeden Distribution. Diese Vorgehensweise ist notwendig, da alle Objektreferenzen immer von der gemeinsamen Basisklasse Object abgeleitet und als Object übertragen werden.
2. Der Klient initiiert eine Transaktion über das Transaktionsobjekt, indem die Methode
   ```
   trans.begin();
   ```
   verwendet wird.
3. Der Klient ruft im Rahmen dieser globalen Transaktion die Methoden der Serverobjekte auf. Der Transaktionskontext wird implizit und indirekt an die jeweils Beteiligten weitergereicht. Sinnbildlich gesprochen, kann gesagt werden, dass der Kontext in dem Protokoll „*mitschwimmt*“.
4. Die Anfrage gelangt durch den Methodenaufruf zur Serverseite und dort zu einem geeigneten Objekt, das diese Anfrage beantworten kann. Dieses Objekt und alle anderen Objekte, die aufgerufen werden, sind Teil dieser globalen Transaktion und somit für den Zugriff durch Dritte gesperrt. Die Aufgabe, Objekte zu sperren, Buch zu führen über die an der Transaktion beteiligten Objekte und ggf. den Ablauf mitzuprotokollieren, liegt in der Verantwortung des Transaktionsdienste und ist für die restlichen Beteiligten völlig transparent.
5. Der Klient ruft über sein Transaktionsobjekt
   ```
   trans.commit();
   ```
   bzw.
   ```
   trans.rollback();
   ```
   auf. Damit ist die Transaktion aus Sicht des Klienten beendet.

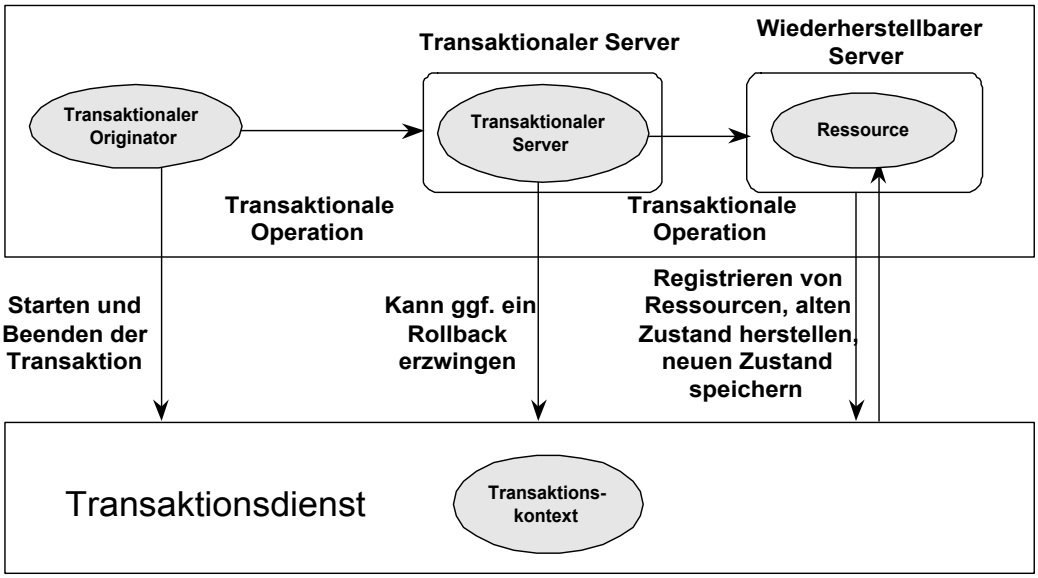


Abbildung 15: Transaktionsdienst: Implizite Weitergabe von Transaktionskontexten

Die durch Methoden *commit()* bzw. *rollback()* verlangte Beendigung der Transaktion wird an die Serverseite weitergereicht. Der Transaktionsdienst ist verantwortlich dafür, dass die Beteiligten benachrichtigt werden und den Grund für die Beendigung der Transaktion erfahren.

Dies ist notwendig, da z. B. die widerherstellbaren Serverobjekte in Abhängigkeit von einem *commit* bzw. eines *rollback* den neuen Zustand persistent machen, bzw. den alten Zustand wiederherstellen müssen.

Besonders hervorzuheben ist hierbei, dass nur Objekte, die Teil einer globalen Transaktion sind, die Eigenschaft der Exklusivität besitzen, d. h., diese Objekte sind mit einer Sperre versehen und werden erst deaktiv, wenn die Transaktion endet. Mit CORBA 3 und den dann vorhandenen CORBA-Komponenten werden auch ohne Transaktionsverarbeitung zustandsbehaftete Objekte möglich sein.

Andere Dienste sind von den CORBA-Herstellern i. d. R. nicht implementiert worden, für die vorliegende Ausarbeitung irrelevant und werden an dieser Stelle auch nicht im Detail untersucht.

6 Dynamisches CORBA

Im Allgemeinen wird der Zugang zu einer verteilten CORBA-basierten Anwendung durch die von einem IDL-Compiler generierten Softwarekomponenten Stub und Skeleton vorgenommen. Dies setzt jedoch voraus, dass der Klient über die gleiche IDL-Vereinbarung wie der Server verfügt und dass alle zu übertragenden Daten in dem Sinne statisch sind, dass sie im Vorfeld durch IDL beschrieben sind.

Die Möglichkeiten, zur Laufzeit Informationen über unbekannte Objekte zu erfragen, zu interpretieren und anschließend mit diesen unbekannten Objekten zu arbeiten, ist in der Literatur bisher stiefmütterlich behandelt worden. Es existieren keine konkreten oder nur unzureichende Angaben zu

1. wie arbeitet man mit diesen Möglichkeiten des dynamischen CORBA.
2. welche generischen Objektarten existieren für den Umgang mit unbekannten Objekten und wie werden sie genutzt.
3. welche Aufwände sind notwendig, um diese generischen Objekte zu instanziieren und mit Inhalt zu füllen.
4. welchen Verlust der Verarbeitungsgeschwindigkeit muss man einkalkulieren, wenn diese Möglichkeiten anstelle eines statischen Zugangs genutzt werden.
5. welchen Mehrwert hat man zu erwarten, wenn auch bei statisch beschriebenen Datentypen die dynamische Schnittstelle genutzt wird.

In den nachfolgenden Kapiteln wird eine detaillierte Darstellung der dynamischen CORBA-Komponenten vorgenommen.

Die Grundlagen dynamischen Verhaltens sind zu sehen in:

1. selbstbeschreibenden Objekten, d. h. erhält der Klient ein ihm unbekanntes Objekt oder eine nichtbekannte Datenstruktur, so muss er in die Lage versetzt werden, aufgrund von ggf. gekapselten Metadaten diese Objekte/Datenstrukturen zur Laufzeit zu interpretieren und geeignet darauf zu reagieren.
2. allgemeingültige Objekte, die entweder jede im Vorfeld beschriebene Datenstruktur aufnehmen können oder einen Mechanismus zur Verfügung stellen, zur Laufzeit neue Datenstrukturen zu erzeugen und zu verschicken.

Das vorliegende Kapitel beschäftigt sich mit den dynamischen CORBA-Komponenten *TypeCode*, *Any*, *dynamisches Any*, dem *Interface Repository* und der dynamischen Schnittstelle für den Zugang eines Klienten zu unbekannten Objekten. Eine sehr gute Grundlage bzgl. TypeCodes, Any und dynamischem Any wird in [Vinosky et al., 1999] vermittelt. Die dynamische Schnittstelle von CORBA und das Interface Repository werden zum Teil sehr fehlerbehaftet in [Hoque, 1998] behandelt. Neuere, jedoch zum Teil sehr oberflächliche Ausarbeitungen finden sich in [Schmidt et al., 2002a] und [Schmidt et al., 2003]. Sowohl bei Hoque, als auch bei Schmidt bleiben die meisten der o. a. Fragen unbeantwortet.

6.1 Einführungsbeispiel

In der realen Welt existieren eine Reihe von Beispielen, in denen eine verteilte Anwendung nicht immer statisch sein kann. D. h. es ist im Vorfeld nicht immer möglich die entstehenden Daten durch IDL zu beschreiben.

Ein Beispiel tritt in Polizeisystemen im Zusammenhang mit der so genannten Fallrecherche auf:

> Der Polizist stellt eine Anfrage bzgl. einer Person X, die verdächtigt ist, eine Bank beraubt zu haben. Während der Recherche stellt sich heraus, dass X bereits einschlägig bekannt

ist im Zusammenhang mit Banküberfällen und anderen kriminellen Delikten. Person X ist bekannt oder gar befreundet mit einer Person Y, die ebenfalls wegen ähnlicher Delikte erkennungsdienstlich erfasst wurde und im Verdacht steht, mit X andere Delikte begangen zu haben. Bei Y findet sich eine Telefonnummer von einem mutmaßlichen Waffen- und Drogenhändler Z. Möglicherweise ist die Waffe, die X bei seinem mutmaßlichen Überfall auf die Bank verwendet hat, von Z geliefert worden.

In Abhängigkeit weiterer vorhandener Informationen kann dies fast beliebig fortgesetzt werden. Ein grundsätzliches Problem, das mit dynamischen CORBA-Komponenten zu lösen ist, ist das Erzeugen von zur Kompilierzeit unbekannten Datenstrukturen und deren Übertragung an den Klienten.

Es ist ersichtlich, dass zwar eine Vielzahl der anfallenden Daten im Vorfeld durch IDL beschreibbar sind, aber i. d. R. sind die aufzubauenden Beziehungen zwischen diesen Daten nicht vorherbestimmbar, sondern werden zur Laufzeit auf der Serverseite aufgebaut. Im Vorfeld nicht deterministisch ist somit, welche Daten, Datenmengen und unterschiedliche Beziehungstypen an den Klienten in welcher Form zurückgeliefert werden.

Für die Software auf der Serverseite ergeben sich die Probleme

j. wie können zur Laufzeit Datenstrukturen/Objekte erzeugt werden, für die es keine IDL-Beschreibung gibt?
k. wie können Metadaten hinzugefügt werden, die es einem späteren Empfänger der Daten erlauben, diese richtig zu interpretieren?
l. wie können derartige Daten unter Berücksichtigung des Marshallings/Unmarshallings über ein Netztwerk übertragen werden?

Für die Software auf der Empfängerseite ergeben sich die Probleme

1. ein unbekanntes, nicht in IDL beschriebenes Objekt entgegen zu nehmen.
2. herauszufinden, welches Datum bzw. welche u. U. geschachtelten Daten dieses Objekt enthält, um es geeignet aus diesem Objekt extrahieren zu können.
3. das Datum anhand von Metadaten zu interpretieren, um die einzelnen Subdaten und die vorhandenen Beziehungen auflösen zu können.

Um diese Problemstellungen lösen zu können, sind zwei Ansätze denkbar:

1. Die Lösung erfolgt durch einen dynamischen Datentyp, der zur Laufzeit beliebige Datenstrukturen erzeugen kann, diese Struktur in einem Kontainerobjekt an den Klienten weiterleitet und zum anderen bedarf es einer Hilfe für den Klienten, um die erhaltenen Daten zur Laufzeit zu interpretieren.
2. Sämtliche nur erdenkliche Datenstrukturen werden modelliert und durch IDL beschrieben. Damit wird jedoch lediglich das Problem der o. a. Datenstrukturen in einer bekannten Anwendung gelöst. Das Problem, mit unbekannten Objekten ohne IDL-Beschreibung zu arbeiten wird hierdurch nicht abgedeckt.

Spätestens bei der Realisierung intelligenter Softwareagenten (s. Beispiel in Kapitel 6.6) oder dem sogenannten B2B (Business-2-Business) wird man an den dynamischen Komponenten nicht vorbeikommen. Neben CORBA ist eine Lösung dieser Probleme auch mit den Web Services (s. Kapitel 17), die letztendlich die Idee des dynamischen CORBA's internetfähig machen, möglich.

In den nachfolgenden Kapiteln wird dargelegt, dass die dynamischen CORBA-Komponenten das Problem, zur Laufzeit neue Datenstrukturen aufzubauen bzw. mit unbekannten Objekten zu arbeiten, lösen können.

Was beide Technologien, CORBA und Web Services, jedoch nicht zu leisten vermögen, ist die Vermittelung der Semantik der zu übertragenden Daten. Damit ist eine weitere Verarbeitung der Daten an anderer Stelle ohne Zusatzwissen nicht oder nur schwer möglich.

6.2 TypeCodes

Das Problem eines generischen Objektes bzw. einer generischen Datenstruktur ist darin zu sehen, dass zur Kompilierzeit nicht bekannt ist, wie dieses Objekt, diese Datenstruktur, konkret aussieht, bzw. welches aktuelle Datum zu einem Zeitpunkt in diesem Objekt enthalten ist. Es ist also unumgänglich, dass zur Laufzeit geeignete Informationen zu dynamischen CORBA-Datentypen zur Verfügung gestellt werden.

CORBA verwendet hierfür Metadaten. Die Vorgehensweise ist vergleichbar mit Kontainerschiffen in einem Hafen. Von außen kann nicht ausgemacht werden, was die Kontainer konkret enthalten; aber aufgrund eines Lieferscheins erhält man Auskunft über den Inhalt des Kontainers und damit zumindest indirekt, wie dieser Inhalt aus dem Kontainer ein- bzw. ausgeladen werden kann.

Definition 33: Pseudoobjekt

Ein Pseudoobjekt ist ein in IDL-beschriebenes Objekt, das nicht als CORBA-Objekt, also als verteiltes, serverseitiges Objekt anzusehen ist.

Grundlegend für alle statischen und dynamischen CORBA-Komponenten ist das Pseudoobjekt *TypeCode*, da dieses Objekt Informationen, also die Metadaten, über das jeweilige Objekt bzw. die jeweilige Datenstruktur kapselt, die der Klient gezielt abfragen kann. Damit entspricht der TypeCode weitestgehend dem o. a. Lieferschein.

Beispielsweise beschreibt ein *TypeCode* das aktuelle Datum eines Kontainer-Objektes und dient somit der Typsicherheit oder es liefert den Repository-Namen, d. h. einen vollqualifizierten Namen inklusive Version, von einem Objekt etc. Der Aufbau von *TypeCodes* und die einzelnen *TypeCode*-Parameter werden im Folgenden beschrieben.

Definition 34: TypeCodes

Ein *TypeCode* ist ein Pseudoobjekt, das zur Laufzeit Informationen über einen bestimmten Datentyp enthält.

Der Typecode kapselt also Metadaten zu einem bestimmten Objekt, zu einer bestimmten Datenstruktur oder zu einem einfachen Datentypen. Entsprechend stellt diese Pseudoobjekt Methoden zur Verfügung, mit denen gezielt Eigenschaften des Datums bzw. des Objektes erfragt werden können (s. u. TypeCode Interface).

Im vorliegenden Kapitel werden die TypeCodes in Zusammenhang mit einfachen oder zusammengesetzten Datentypen untersucht. In nachfolgenden Kapiteln wird dies erweitert um dynamisch erzeugte TypeCodes und ihre Bedeutung für Objekte. Im Zusammenhang mit dem generischen Datentyp DynAny wird zusätzlich ein konkretes Beispiel angeführt, das die Erzeugung dynamischer TypeCodes bzgl. unbekannter Datenstrukturen illustriert.

Ein *TypeCode* beschreibt also die jeweiligen Daten. Für die primitiven Datentypen *boolean*, [*w*]*char*, *octet*, [*unsigned*] *short*, [*unsigned*] *long*, *float*, *double* und [*w*]*string* sind die *TypeCodes* bereits vordefiniert. Für alle anderen, in IDL definierten Daten und Schnittstellen werden von dem IDL-Compiler entsprechende *TypeCodes* generiert.

TypeCodes werden analog zu den meisten CORBA-Objekten bzw. Pseudoobjekten in der Interface Definition Language beschrieben. Konzeptionell betrachtet, besteht dieses Pseudoobjekt aus einem Wertepaar, das sich zusammensetzt aus einem Beschreibungstypen und einer Beschreibung. Der Beschreibungstyp wird in der CORBA-Spezifikation als Aufzählungstyp *TCKind* beschrieben. *TCKind* legt allgemein den jeweiligen Typ fest, also *tk_string* für einen IDL-String oder *tk_struct* für einen in IDL beschriebenen benutzerdefinierten Datentyp etc.

Der Beschreibungsteil hängt naturgemäß von dem Beschreibungswert ab. Beispielsweise sind bei einer in IDL beschriebenen *Union* der Diskriminator und die Anzahl der möglicherweise aufzunehmenden Elemente und ihr Datentyp von Interesse, bei einem *String* u. U. die Länge dieses Strings.

Die von der OMG verabschiedete und standardisierte Beschreibungssprache IDL ist als Metasprache sehr gut geeignet und ist im Zusammenhang mit CORBA eine besonders gute Darstellungsform für verteilte Objekte. Im Folgenden wird die IDL-Definition für den Aufzählungstyp *TCKind* und für die *TypeCodes* angegeben.

Die IDL-Beschreibung des Pseudoobjekts *TypeCode*:

```
enum TCKind {                                   // PIDL
#       pragma version TCKind 2.3
        tk_null,        tk_void,
        tk_short,       tk_long,                tk_ushort,      tk_ulong,
        tk_float,       tk_double,              tk_boolean,     tk_char,
        tk_octet,       tk_any,                 tk_TypeCode,    tk_Principal,  tk_objref,
        tk_struct,      tk_union,               tk_enum,        tk_string,
        tk_sequence,    tk_array,               tk_alias,       tk_except,
        tk_longlong,    tk_ulonglong,           tk_longdouble,
        tk_wchar,       tk_wstring,             tk_fixed,       tk_value,      tk_value_box,
        tk_native,      tk_abstract_interface
};
```

Programmbeispiel 30: IDL-Beschreibung der TypeCodes

Mit den *TCKind*-Aufzählungswerten werden alle nur erdenklichen in IDL definierten Daten, Schnittstellen der entfernten Objekte und Objektreferenzen erfasst. Erhält der Klient z. B. ein Datenpaket mit dem Beschreibungstyp *tk_struct*, so muss es sich um eine Struktur handeln. Mit dieser Zusatzinformation kann der Klient anhand des *TypeCodes* schrittweise dazu übergehen, die Struktur zu traversieren, die einzelnen Datenelemente und ihren Typ bestimmen usw..

Für die Interpretation der Daten existieren die in der folgenden Schnittstelle *TypeCode* aufgeführten Methoden. Eine ausführliche Beschreibung der folgenden Operationen wird nachfolgend vorgenommen; in der IDL-Beschreibung sind Kurzkommentare eingefügt.

```
interface TypeCode {                            // PIDL
# pragma version TypeCode 2.3
       exception   Bounds {};
       exception   BadKind {};

       // für alle TypeCode gelten die folgenden Methoden zum Vergleichen von TypeCodes
       // oder der Abfrage des allgemeinen Typs

       boolean  equal(in TypeCode tc);
       boolean  equivalent(in TypeCode tc);
       TypeCode get_compact_typecode();
       TCKind   kind();

       // die Methode id() macht nur Sinn für Datentypen der Kategorie
       // Objektreferenz (tk_objref), Struktur (tk_struct), Union (tk_union),
       // Aufzählungen (tk_enum), Aliasnamen (tk_alias), Ausnahmen (tk_except)
       //
       // Andere IDL-Objekte besitzen keinen Repository-Identifizierer

       RepositoryId    id() raises (BadKind);

       // ein Name ist nur sinnvoll für die folgenden IDL-Datentypen:
       // Objektreferenz (tk_objref), Struktur (tk_struct), Union (tk_union),
       // Aufzählungen (tk_enum), Aliasnamen (tk_alias), Ausnahmen (tk_except)

       Identifier name() raises(BadKind);
```

```
        // Strukturen und die anderen aufgeführten Datentypen enthalten Elemente und
        // die Elemente haben in diesem Namensraum eindeutige Namen.
        // tk_struct, tk_union, tk_enum, tk_value, tk_except

        unsigned long  member_count() raises(BadKind);
        Identifier     member_name(in unsigned long index) raises(BadKind, Bounds);

        // Für die folgenden Datentypen ist es sinnvoll, den TypeCode der Datenelemente
        // zu ermitteln:
        // tk_struct, tk_union, tk_value, and tk_except

        TypeCode member_type(in unsigned long index) raises(BadKind, Bounds);

        // Eine IDL-Union (tk_union) weist immer einen Einstiegspunkt auf (label),
        // hat einen Diskriminator und optional einen Defaultzweig

        any       member_label(in unsigned long index) raises(BadKind, Bounds);
        TypeCode  discriminator_type() raises(BadKind);
        long      default_index() raises(BadKind);

        // String (tk_string), Sequenzen (tk_sequence) und Array's (tk_array) haben
        // eine Länge bzw. eine Anzahl von Elementen

        unsigned long length() raises(BadKind);


        // Sequenzen (tk_sequence), Datenfelder (tk_array) und Wertetypen (tk_value_box)
        // können beliebige in IDL definierte Datentypen aufweisen. Diese Datentypen können
        // mit Aliasnamen (tk_alias) versehen sein.

        TypeCode content_type() raises (BadKind);

        // für tk_fixed

        unsigned short fixed_digits() raises(BadKind);
        short          fixed_scale()  raises(BadKind);

        // für den neuen IDL-Typ valuetype (tk_value)

        Visibility     member_visibility(in unsigned long index) raises(BadKind, Bounds);
        ValueModifier  type_modifier() raises(BadKind);
        TypeCode       concrete_base_type() raises(BadKind);
};
```

Programmbeispiel 31: Das Interface TypeCode

Das TypeCode-Interface beschreibt eine Reihe wichtiger Methoden, die dazu dienen, Informationen abzufragen, also einen Objektaufbau zu interpretieren.

Die folgenden Methoden gelten für alle TypeCodes:

1. *TCKind kind()*:
   Diese Methode liefert den Beschreibungswert *TCKind* des *TypeCodes* zurück und beschreibt somit in aller Allgemeingültigkeit den aktuellen Datentyp und welche Operationen auf diesem *TypeCode* möglich sind.
2. *boolean equal(in TypeCode tc)*:
   Diese Operation erlaubt den Vergleich zweier *TypeCodes*, wobei der Wert *true* bei Gleichheit zurückgeliefert wird.
3. *boolean equivalent(in TypeCode tc)*:
   Diese Operation ist mit der CORBA-Spezifikation 2.3 neu hinzugenommen. Analog zu oben werden hier ebenfalls zwei *TypeCodes* miteinander verglichen, wobei jedoch Aliase ignoriert werden.
4. *TypeCode get_compact_typecode()*:
   Eine mit der CORBA 2.3 Spezifikation hinzugekommene neue Operation. Der Rückgabewert ist ein neuer *TypeCode* mit einem leeren String für den Datentyp und leeren Strings für die Attributnamen.

Für die TypeCodes Objektreferenz, Struktur, Union, Aufzählungstyp, Aliasname und Ausnahme sind zusätzlich vorhanden:

1. *RepositoryId id()*:
   Diese Methode liefert einen vollständig qualifizierten Repository-Namen. Der Aufbau dieses Namens besteht aus dem Präfix „IDL:“, gefolgt von eventuellen Modulnamen, die durch das Zeichen ‚/‘ getrennt sind, gefolgt von dem Schnittstellennamen des Objektes, gefolgt von der Version der Schnittstelle.
2. *Identifier name()*:
   Liefert den Namen des Objektes, ohne Präfix, eventuellen Modulnamen und Version.

Für Strukturen, Unions, Aufzählungen und Ausnahmen existieren zusätzlich die Methoden:
1. *unsigned long member_count()*:
   Liefert für die o. a. Datentypen die Anzahl der Attribute.
2. *Identifier member_name(in unsigned long index)*:
   Liefert zu jedem Attribut, das über einen korrespondierenden Index angesprochen wird, den Namen des Attributes.

Speziell für Strukturen, Unions und Ausnahmen kommt die folgende Methode hinzu:
1. *TypeCode member_type(in unsigned long index)*:
   Diese Methode liefert einen TypeCode bzgl. des referenzierten Attributes zurück. Dies ist sinnvoll, um über diesen Beschreibungstypen Informationen über das enthaltene Objekt zu erhalten; beispielsweise eine Struktur in einer Struktur, oder eine rekursive Struktur. In derartigen Fällen ist es unerlässlich, auch diese Attribute weiter zu untersuchen.

Für eine Union sind die folgenden Methoden von Interesse:
1. *any member_label(in unsigned long index)*:
   Liefert den zu Index korrespondierenden Wert in einem Any-Container zurück.
2. *TypeCode discriminator_type()*:
   Liefert eine Beschreibung des Datentyps zurück, über den die switch-case-Anweisungen gesteuert werden.
3. *long default_index()*:
   Liefert den korrespondierenden Wert des Defaultzweiges zurück, bzw. –1, falls kein default in der Union vereinbart war.

Bei Sequenzen, Feldern und Strings ist die Länge bzw. die Anzahl der Elemente von Interesse:
unsigned long length():
a. Bei gebundenen Strings wird die in der IDL-Definition vereinbarte obere Schranke zurückgeliefert.
b. Bei ungebundenen Strings erhält der Aufrufer den Wert 0.
c. Für Felder wird die Dimension des Arrays zurückgegeben, die immer ungleich 0 sein muss.
d. Bei gebundenen Sequenzen wird die maximale Anzahl möglicher Elemente zurückgegeben.
e. Bei ungebundenen Sequenzen liefert die Methode den Wert 0 zurück.

Für Sequenzen, Felder und Aliastypen ist eine Beschreibung der enthaltenen Typen von Interesse:
TypeCode content_type(): Liefert den Beschreibungstyp der enthaltenen Elemente zurück.

Die o. a. IDL für das Pseudoobjekt TypeCodes ist nicht in allen ORBs vollständig implementiert.

In Abhängigkeit von dem *TypeCode* werden bestimmte *TCKind*-Werte unterschiedlich belegt. In der folgenden Tabelle findet eine Auflistung statt; Wiederholungen bei TypeCode-Parametern werden durch geschweifte Klammern begrenzt:

TCKind	Parameter
tk_fixed	Ziffern
tk_objref	Repository-Id, Interfacename
tk_struct	Strukturname, { Attributname, Attribut-TypeCode } ...
tk_union	Union-Name, TypeCode des Diskriminators, { Wert des Labels, Attributname, Attribut-TypeCode } ...
tk_enum	Name der Aufzählung, { Name des Attributs } ..., Repository-Id
tk_string	Obere Schranke oder 0
tk_wstring	Obere Schranke oder 0
tk_sequence	TypeCode der Elemente, obere Schranke oder 0
tk_array	TypeCode der Elemente, Dimension
tk_alias	Aliasname, TypeCode, Repository-Id
	Name der Ausnahme, { Name des Attributs, TypeCode des Attributs } ..., Repository-Id

Tabelle 5: Bedeutung der TCKind-Werte

6.2.1 Beispiel: Die Handhabung von TypeCodes

In dem folgenden Beispiel wird anhand einer vorgegebenen IDL-Beschreibung eine Java-Applikation entwickelt, die anhand des *TypeCodes* die IDL-Beschreibung interpretieren soll. Die IDL-Beschreibung enthält eine Struktur, einen Union-Datentyp und eine Schnittstellenbeschreibung. Es handelt sich hierbei um einen statischen TypeCode. Stellt man sich jedoch auf den Standpunkt, dass die beschriebene Datenstruktur nicht bekannt ist, auf der Serverseite erzeugt und an den Klienten gesendet wird, so entsteht die Notwendigkeit, diese Struktur zur Laufzeit zu interpretieren. Damit kann dieses Beispiel leicht auf beliebige, zur Laufzeit erstellte und unbekannte Datenstrukturen übertragen werden.

Gegeben sei die folgende IDL-Beschreibung:

```
module typecode {
    struct Struktur {
        boolean        aBoolean;
        char           aChar;
        octet          anOctet;
        short          aShort;
        long           aLong;
        float          aFloat;
        double         aDouble;
        string         aString;
    };
    union UnionType switch(char) {
        case 's':
                short s;
        case 'b':
                boolean b;
        case 'c':
                char c;
        case 'B':
                string<10> bString;
        case 'S':
                string ubString;
        default:
                Struktur st;
    };
    interface StrukturInterface {
        void send(in any a);
    };
};
```

Programmbeispiel 32: IDL-Beschreibung für das TypeCode-Beispiel

Über die vom IDL-Compiler generierten *TypeCodes* für *Struktur*, *StrukturInterface* und *UnionType* können alle Informationen über die o. a. IDL Vereinbarungen in Erfahrung gebracht werden! Dies beinhaltet den Typ des Datums, den vergebenen Namen und den Wert.

Die folgende Java-Applikation illustriert den Umgang mit TypeCodes, wobei der dynamische Aspekt erst einmal unberücksichtigt bleibt:

```
package typecode;

import org.omg.CORBA.*;       // notwendig für Java-CORBA-Objekte

public class TypeCodeTest {
    public static void main(String[ ] args)
    {
        // Die TypeCodes zu den jeweiligen IDL-Beschreibungen müssen in Java über
        // die korrespondierenden, vom IDL-Kompilierer erzeugten, Helperklassen
        // erfragt werden (s. Kapitel 3.5, IDL und die Sprachabbildung für Java)

        TypeCode tc = StrukturHelper.type();
        showInfo(tc);
        tc = StrukturInterfaceHelper.type();
        showInfo(tc);
        tc = UnionTypeHelper.type();
        showInfo(tc);
    }

    // anhand des uebergebenen TypeCodes wichtige Informationen ueber das Datum oder
    // die Schnittstelle ermitteln und ausgeben

    public static void showInfo(TypeCode tc)
    {
        try {
                // für alle Datentypen/Schnittstellen existiert immer eine Identität und
                // ein Name.

                System.out.println("RepositoryId: "  + tc.id());
                System.out.println("Name:         "  + tc.name());

                // bei Schnittstellen und benutzerdefinierten Datentypen sind weitere
                // Informationen von Interesse. Die Ermittelung dieser Informationen
                // erfolgt über den im TypeCode enthaltenen Beschreibungstyp (TCKind).

                TCKind theKind = tc.kind();

                // für Strukturen, Vereinigungen, Aufzaehlungstypen und definierten Ausnahmen

                if (theKind == TCKind.tk_struct || theKind == TCKind.tk_union ||
                    theKind == TCKind.tk_enum   || theKind == TCKind.tk_except) {
                       System.out.println("Anzahl Member: " + tc.member_count());
                       for (int i = 0; i < tc.member_count(); ++i) {
                              TCKind tcKind = tc.member_type(i).kind();
                              System.out.println("\tMembername: "  +
                                                 tc.member_name(i) +
                                                 "\t\t\tTyp: "     +
                                                 nameType[tcKind.value()]);
                       }

                       // nur eine Union kann einen Diskriminatortyp enthalten

                       if (theKind == TCKind.tk_union) {
                              System.out.println("\tDiskriminatortyp: " +
                                     nameType[tc.discriminator_type().kind().value()]);
                       }
                }
        }
        catch (Exception e) {
                System.err.println(e.getMessage());
                e.printStackTrace();
                System.exit(1);
        }
    }

    // für eine lesbare Ausgabe werden die Aufzaehlungstypen (tk_<typ>) in der vorge-
```

```
    // gebenen Reihenfolge als Strings aufgelistet (z. B. besitzt der Aufzaehlungstyp
    // tk_struct den Wert 15 und somit ist „struct“ an der Position 15 aufgelistet, wobei wie
    // in Java ueblich, bei Null angefangen wird zu zaehlen

    private final static String[] nameType = {
        "null", "void", "short", "long", "unsigned short", "unsigned long", "float", "double",
        "boolean", "char", "octet", "any", "TypeCode", "Principal", "object reference",
        "struct", "union", "enum", "string", "sequence", "array", "alias", "exception"
    };
}
```

Programmbeispiel 33: Anwenden von TypeCodes

Nach Kompilation der IDL, aller generierten Java-Klassen und des Klienten erhält man nach Aufruf der Applikation die folgende Ausgabe:

```
RepositoryId:     IDL:typecode/Struktur:1.0
Name:             Struktur
Anzahl Member:    8
   Membername:    aBoolean          Typ: boolean
   Membername:    aChar             Typ: char
   Membername:    anOctet           Typ: octet
   Membername:    aShort            Typ: short
   Membername:    aLong             Typ: long
   Membername:    aFloat            Typ: float
   Membername:    aDouble           Typ: double
   Membername:    aString           Typ: string
RepositoryId:     IDL:typecode/StrukturInterface:1.0
Name:             StrukturInterface
RepositoryId:     IDL:typecode/UnionType:1.0
Name:             UnionType
Anzahl Member:    6
   Membername:    s                 Typ: short
   Membername:    b                 Typ: boolean
   Membername:    c                 Typ: char
   Membername:    bString           Typ: string
   Membername:    ubString          Typ: string
   Membername:    st                Typ: struct
   Diskriminatortyp:                char
```

Beispiel 6: Ausgabe der TypeCode-Iterationen

Diese Ausgabe entspricht bzgl. ihrer Namen, den Identitäten und dem jeweiligen Aufbau exakt der o. a. IDL-Beschreibung.

6.2.2 Dynamisches Erzeugen von TypeCodes

In vielen Anwendungsfällen ist es ausreichend, mit den bereits vordefinierten bzw. vom IDL-Compiler erzeugten TypeCodes zu arbeiten. Für dynamische Anwendungen ist es u. U. notwendig, für völlig neue, i. d. R. nicht in IDL beschriebene benutzerdefinierte Datentypen, TypeCodes zur Laufzeit zu erzeugen. CORBA stellt verschiedene Methoden zur Erzeugung von TypeCodes zur Laufzeit zur Verfügung, die nachfolgend beschrieben werden. Dieser zur Laufzeit generierte TypeCode repräsentiert ein zur Kompilierzeit noch nicht bekanntes Objekt/Datenstruktur.

In dem o. a. Beispiel über die Fallrecherche eines Polizeisystems wäre dies z. B. der Fall, wenn anhand existierender Daten eine neue Beziehung aufzubauen ist. Die Serverseite kann hierfür zur Laufzeit wie folgt vorgehen (vgl. Kapitel 6.5):

1. Einen neuen TypeCode z. B. von einer Struktur erzeugen.

2. TypeCodes für alle Datenmember erzeugen.

Mit diesen Voraussetzungen sind anschließend die Daten in eine neue Struktur zu füllen und an den Klienten zu liefern.

Im Folgenden werden die jeweiligen Methoden zum dynamischen Erzeugen von TypeCodes beschrieben und im Zusammenhang mit dem Datentyp dynamisches Any und der dynamischen Schnittstelle werden entsprechende Beispiele angegeben.

In der CORBA Spezifikation 2 ist das Erzeugen von TypeCodes zur Laufzeit beschrieben. Der primäre Grund ist in der benötigten Interoperabilität zwischen CORBA und anderen Verteilungsplattformen wie z. B. DCOM zu sehen. Eine weitere Anwendung ergibt sich im Zusammenhang mit dem Aufbau dynamischer Datenstrukturen auf der Serverseite, die im Vorfeld nicht mit IDL beschrieben wurden. Bspw. kann für diese Struktur ein entsprechender TypeCode erzeugt werden, die Struktur selbst wird durch das Pseudoobjekt DynAny beschrieben und der Datentyp inklusive TypeCode werden serialisiert in einem Any-Container an den Klienten übergeben. Anhand des TypeCodes kann der Klient die Datenstruktur interpretieren und zu allen enthaltenen Attributen die korrespondierenden Werte ermitteln.

Von Interesse ist bei allen dynamischen CORBA-Komponenten das Einstellen von Daten in das neu erzeugte Objekt. Dazu dienen in Abhängigkeit von dem Datentyp immer die Elementfolgen, die in IDL vordefiniert sind. Z. B. bedarf es bei einer Struktur einer StructMemberSeq, die wie folgt definiert ist:

```
struct StructMember {
    Identifier name;
    TypeCode   type;
    IDLType    type_def;
};
typedef sequence <StructMember> StructMemberSeq;
```

Beispiel 7: Die Struktur/Sequenz StructMember/StructMemberSeq

Hierbei ist ein *Identifier* lediglich ein Aliasname für einen String. Der Name ist innerhalb eines Namensraumes eindeutig. Der TypeCode beschreibt das Datum und durch IDLType wird ein eventuell vorhandener Aliasname beschrieben.

Analog werden für Unions, Aufzählungstypen etc. Elementfolgen definiert.

Die Operationen zum dynamischen Erzeugen von TypeCodes sind Teil des Pseudoobjektes ORB. Zur Verfügung gestellt werden die folgenden Methoden:

a. TypeCode create_struct_tc(
   in RepositoryId id,
   in Identifier name,
   in StructMemberSeq members);

   Für die Erzeugung eines TypeCodes, der ein Struktur beschreibt, werden Angaben zum vollständig qualifizierten Namen in der Form „IDL:module$_1$/.../module$_n$/name:Version“ (RepositoryId), der Name der Struktur und eine Sequenz der Elemente benötigt. Jedes Element der Sequenz besteht aus einem Namen, einem TypeCode für dieses Element und aus einer Objektreferenz vom Typ *IDLType*, in der die Typendefinition des Elementes abgelegt ist. Die Objektreferenz ist nur im Zusammenhang mit dem Interface Repository von Interesse und muss bei der Erzeugung eines neuen TypeCodes immer auf CORBA::_nil() gesetzt werden!

b. TypeCode create_union_tc(
   in RepositoryId id,

```
                    in Identifier       name,
                    in TypeCode         discriminator_type,
                    in UnionMemberSeq   members);
```

Analog zu den Strukturen, jedoch muss der Diskriminatortyp gesetzt werden.

c. TypeCode create_enum_tc(

```
                    in RepositoryId     id,
                    in Identifier       name,
                    in EnumMemberSeq    members);
```

Analog zu oben.

d. TypeCode create_alias_tc(

```
                    in RepositoryId     id,
                    in Identifier       name,
                    in TypeCode         original_type);
```

Analog zu oben.

e. TypeCode create_exception_tc(

```
                    in RepositoryId     id,
                    in Identifier       name,
                    in StructMemberSeq  members);
```

Analog zu oben.

f. TypeCode create_interface_tc(

```
                    in RepositoryId     id,
                    in Identifier       name);
```

Erzeugt einen TypeCode für ein Interface. Von Interesse ist hier der vollständig qualifizierte Name. Über geeignete Methoden des Interface Repositories werden anhand derartiger TypeCodes die Interfaces interpretiert.

g. TypeCode create_string_tc(in unsigned long bound);
Für unbeschränkte, also Null-terminierte, Strings muss die Schranke auf den Wert 0 gesetzt werden.

h. TypeCode create_wstring_tc(in unsigned long bound);
Analog zu (g).

i. TypeCode create_sequence_tc(

```
                    in unsigned long    bound,
                    in TypeCode         element_type);
```

Für ungebundene Sequenzen ist der Wert der Grenze auf 0 zu setzen.

j. TypeCode create_recursive_sequence_tc(

```
                    in unsigned long    bound,
                    in unsigned long    offset);
```

Zusätzlich zu der Grenze ist ein Wert anzugeben, der anzeigt, wie viele Elemente zu überspringen sind, um das nächste Sequenzelement zu erhalten. Ist bspw. die folgende IDL Definition gegeben

```
struct Node {
    long                value;
    sequence<Node>      children;
};

struct TwoLevelRecursive {
    string id;
    struct Nested {
           long                          value;
           sequence<TwoLevelRecursive>   children;
    };
};
```

Um einen TypeCode für die Struktur Node zu erzeugen, muss der offset für die Erzeugung des TypeCodes für *children* auf 1 und bei der zweiten Struktur muss offset auf 2 gesetzt werden.

k. TypeCode create_array_tc(
in unsigned long length,
in TypeCode element_type);

Analog zu oben.

Die in den oben angeführten Methoden angegebenen RepositoryId-Parameter sind leider irreführend. Es vermittelt den Eindruck, dass man sich bei dem Erzeugen dynamischer TypeCodes auf eine existierende IDL-Datenstruktur bzw. ein IDL-Objekt bezieht. Tatsächlich ist diese RepositoryId lediglich ein Alias für einen String, der ggf. auch leer sein kann. In einem nachfolgenden Kapitel wird dies anhand eines Beispiels illustriert.

6.2.3 Probleme beim Vergleich zweier TypeCodes

Nach mehreren Spezifikationen im Bereich CORBA sind zum Teil grundlegende Änderungen und auch Inkompatibilitäten aufgetreten. Eine Auswirkung von Inkompatibilität betrifft den Vergleich von TypeCodes.

In dem o. a. Kapitel wurden die Methoden TypeCode::equal und TypeCode::equivalent angesprochen. Die Frage nach der Gleichheit zweier TypeCodes soll in dem vorliegenden Kapitel genauer untersucht werden. Leider ist die Beantwortung der Frage, ob zwei TypeCodes gleich sind, abhängig von der CORBA-Spezifikation. Ein ORB, der dem CORBA-Standard 2.3 genügt, liefert bei einem Vergleich mit TypeCode::equal genau dann den Wahrheitswert true zurück, wenn beide TypeCodes identisch sind. In den vorhergehenden Spezifikationen gilt dies nicht!

In CORBA 2.2 oder früheren Versionen weist TypeCode::equal in einem hohen Maße implementierungsabhängiges Verhalten auf. Eventuell wurde bei manchen ORBs die Signifikanz von Aliasnamen nicht berücksichtigt, Typ- und Attributnamen wurden u. U. als leerer String hinterlegt und u. U. war der in CORBA 2.3 als verbindlich verlangte Repository-Namen ebenfalls nicht hinterlegt oder mit dem Typnamen gleichgesetzt.

Ein ähnliches Problem ergibt sich, wenn die CORBA 2.3 Methode TypeCode::equivalent verwendet wird. Wurden die TypeCodes mit einem CORBA 2.2 ORB erzeugt, so sind die Ergebnisse nicht deterministisch.

```
struct foo {
    long       aLong;
    string     aString;
};
typedef foo AliasNameFuerFoo;
```

```
struct bar {
    long       anotherLong;
    string     anotherString;
};
```

Unter der Prämisse, dass die hierzu korrespondierenden TypeCodes mit einem CORBA 2.3 ORB erzeugt wurden liefert der Vergleich der drei TypeCodes folgendes Ergebnis:

foo und *bar* sind nicht gleich.
foo und *AliasNameFuerFoo* sind nicht gleich.
bar und *AliasNameFuerFoo* sind nicht gleich.

Bei Verwendung eines CORBA 2.2 ORBs gilt:
Wurden die TypeCodes von dem gleichen ORB erzeugt, so sind die Ergebnisse wie bei einem CORBA 2.3 ORB.

Wurden die TypeCodes mit verschiedenen CORBA 2.2 ORBs erzeugt, so sind die Ergebnisse abhängig von der jeweiligen Implementierung von TypeCode::equal und von der Vollständigkeit der enthaltenen Daten.

Bei Verwendung der CORBA 2.3 Methode TypeCode::equivalent ergibt sich, unter der Voraussetzung, dass die TypeCodes auch von einem CORBA 2.3 ORB erzeugt wurden:

foo und *bar* sind nicht äquivalent.
foo und *AliasNameFuerFoo* sind äquivalent.
bar und *AliasNameFuerFoo* sind äquivalent.

Wurden die TypeCodes durch einen CORBA 2.2 ORB erzeugt und wird mit einem CORBA 2.3 ORB TypeCode::equivalent angewendet, so erhält man als Ergebnis:

Bei den zu vergleichenden TypeCodes ist die RepositoryId korrekt gesetzt, dann erhält man die gleichen Ergebnisse wie oben.

Ist die RepositoryId zumindest bei einem TypeCode nicht korrekt gesetzt, so werden alle Vergleiche als nicht äquivalent ausgewiesen.

Beispiel 8: Vergleich von TypeCodes

6.2.4 Die Serialisierung von TypeCodes

In dem Kapitel über das Protokoll GIOP und der Serialisierung von Daten zu einer Bytefolge für die Übertragung wurde der TypeCode unberücksichtigt gelassen. Ein TypeCode, wenn er z. B. Teil eines dynamischen Datentyps ist, hat die Aufgabe, das Datum exakt zu beschreiben. Wird mit dynamischen TypeCodes gearbeitet, also TypeCodes, die erst zur Laufzeit erzeugt werden, so muss dieser Datentyp ebenfalls an den Empfänger einer Nachricht gelangen.

Für den Fall, dass ein in IDL beschriebener Datentyp sich indirekt selbst enthält, muss die Repräsentation des TypeCodes eine Indirektion enthalten. Damit verbunden ist eine Reduktion der Bytefolge bzgl. der Serialisierung.

Ist ein TypeCode rekursiv in sich selbst enthalten, so wird die Beschreibung des zweiten TypeCodes wieder auf den Anfang gerichtet.

Anhand des folgenden Beispiels wird dieser Sachverhalt illustriert:

Gegeben sei die IDL-Beschreibung

```
struct foo {
       string         tag;
       sequence<foo>  bar;
};
```

Beispiel 9: Rekursion eines TypeCodes

In der folgenden Abbildung wird die Serialisierung des zu der o. a. Struktur korrespondierenden TypeCodes dargestellt:

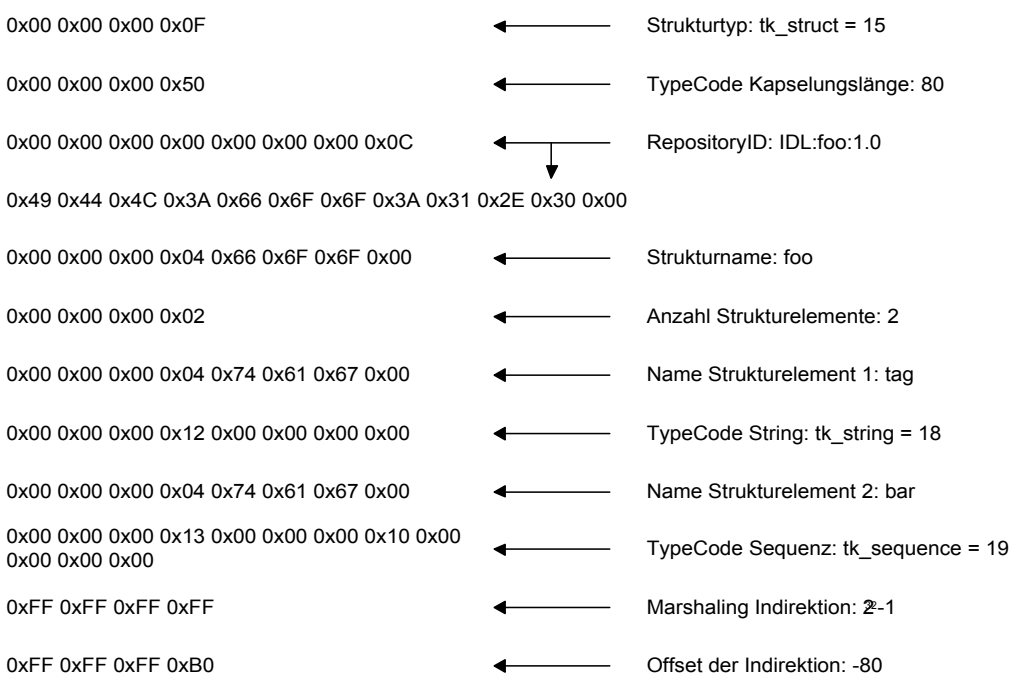


Abbildung 16: Serialisierung eines TypeCodes

Die Serialisierung von TypeCodes ist besonders für die dynamischen CORBA-Typen Any und DynAny notwendig.

6.3 Der Datentyp Any

Der in IDL vorhandene Datentyp *Any* ist allgemeingültig in dem Sinn, dass er jeden in IDL beschriebenen Datentyp aufnehmen kann. Damit ist *Any* grundsätzlich prädestiniert, um zur Kompilierzeit beliebige Datentypen aufzunehmen und zu versenden bzw. zu empfangen. Damit wäre prinzipiell die Möglichkeit gegeben eine Schnittstelle stabil, im Sinne von unveränderlich, zu halten.

Eine weitere Bedeutung kommt dem Datentyp *Any* im Zusammenhang mit den dynamischen CORBA-Komponenten zu. Bspw. ist der Aufbau von Parameterlisten für den Zugang zu unbekannten Objekten durch geeignete Listen realisiert, bestehend aus einem Beschreibungsteil des Parameters und einem aktuellen Wert, der in einen *Any* eingefügt ist. Das Gleiche gilt für den Eventservice, der erst zur Laufzeit Informationen darüber erhält, welche Events von Interesse sind.

Um Daten in ein Any-Objekt einzustellen und später wieder zu extrahieren, bedarf es einer Laufzeitbeschreibung des Datums. CORBA realisiert dies durch das o. a. Pseudoobjekt TypeCode, das für alle primitiven Datentypen existiert und für benutzerdefinierte IDL-Datentypen durch den IDL-Compiler generiert wird. Ab CORBA 2.3 wird zusätzlich das Interface für die dynamische Komposition und Dekomposition von Any-Daten als verbindlich vorgeschrieben (s. Kapitel DynAny). Damit ist es möglich, zur Laufzeit neue Datentypen zu kreieren, in ein Any-Objekt einzustellen und dem Empfänger zuzustellen. Dieser hat dann basie-

rend auf den TypeCodes die Möglichkeit, diese Datentypen zu interpretieren und konkret damit zu arbeiten.

Any wird aufgrund der Tatsache, dass dieses dynamische Objekt jeden in IDL beschriebenen Datentyp aufnehmen kann, häufig mit dem in C/C++ bekannten Zeiger `void *` verglichen. Dieser Vergleich ist nicht haltbar, da ein *Any* aufgrund der generierten TypeCodes über Typsicherheit verfügt. D. h. der void-Zeiger verlangt das Wissen darüber, welcher Datentyp sich an der referenzierten Stelle befindet; ein *Any* hingegen lässt zu, dass man sich dieses Wissen im Vorfeld via TypeCode verschafft. Intern besteht der Kontainer aus dem Datum und dem hierzu korrespondierenden Beschreibungsteil in Form des generierten bzw. vom System zur Verfügung gestellten TypeCodes.

Aufgrund seiner Allgemeingültigkeit ist das Versenden/Empfangen eines Any-Wertes aufwendiger und somit weniger performant, als das Übertragen der Daten ohne eine dynamische Kapsel. Da der Datentyp Any jedoch die Grundlage aller dynamischen CORBA-Komponenten ist, muss eine Untersuchung genau an dieser Stelle einsetzen.

Any ist bei der Umsetzung auf die jeweiligen Zielsprachen unterschiedlich. Hinzu kommt, dass von dem IDL-Compiler in Abhängigkeit der IDL-Dateien eine Reihe von Erweiterungen für den Umgang mit Any generiert wird. In C++ hat man es z. B. mit der Generierung von TypeCodes und überladenen Operatoren zu tun, in Java werden die überladenen Operatoren durch Helperklassen ersetzt, die für einen speziellen Datentyp generiert werden und die geeignete Methoden enthalten, um dieses Datum in einen Kontainer einzustellen bzw. aus einem Kontainer das Datum zu extrahieren.

Das Erzeugen eines Any-Objektes erfolgt durch das bei der Initialisierung des ORBs zurückgelieferte ORB-Objekt orb

```
org.omg.CORBA.Any orb.create_any();
```

Bei Verwendung der Sprache Java sind für benutzerdefinierte Datentypen die generierten Helperklassen von Interesse, um ein Datum in einen Any einzustellen bzw. auszulesen.

```
Gegeben sei die Struktur

    struct EineStruktur {
        ...
    };
```

Das Einstellen in ein Any-Objekt, das mit

```
org.omg.CORBA.Any einAny = orb.create_any();
```

erzeugt wurde, erfolgt durch

```
EineStrukturHelper.insert(einAny, eineStruktur);
```

und das Auslesen erfolgt durch

```
EineStruktur st = EineStrukturHelper.extract(einAny);
```

Programmbeispiel 34: Einstellen einer Struktur in ein Any-Objekt

6.4 Dynamisches Any

Das vorliegende Kapitel beschreibt den Datentyp DynAny, oder genauer das Interface DynAny und seine Implementierungen.

Das DynAny-Interface erlaubt die Komposition und Dekomposition komplexer Datenstrukturen zur Laufzeit, d. h. es ist nicht zwingend erforderlich, dass diese Datentypen bereits zur Kompilierzeit in IDL beschrieben sind. Mit dieser Fähigkeit ist DynAny ein Kandidat, um komplexe Rechercheergebnisse, wie z. B. die Fallrecherche in dem o. a. Polizeisystem, zu

beschreiben und an den Client zurück zu geben. Die Serverseite erzeugt anhand ihres Rechercheergebnisses Datenstrukturen, die im Vorfeld durch IDL-Definition beschrieben wurden und erzeugt zur Laufzeit neue Strukturen, die die Beziehungen zwischen den einzelnen Daten widerspiegeln. Dazu bedient sich die Serverseite der Möglichkeit, zur Laufzeit neue TypeCodes zu erzeugen und die recherchierten Daten inklusive der hierzu korrespondierenden TypeCodes in eine Struktur vom Typ DynAny einzustellen. Der Transport zum Klienten erfolgt durch eine vorhergehende Serialisierung des DynAny-Objektes in ein Any-Objekt.

Es wurde bereits dargelegt, dass in IDL beschriebene Datentypen durch geeignete Methoden oder geeignete überladene Operatoren in ein Any-Objekt eingestellt und ausgelesen werden können. Die nicht vorhandene Möglichkeit des Any-Objektes Daten einzustellen, die zur Laufzeit generiert und nicht im Vorfeld in IDL beschrieben waren, hat ernste Konsequenzen bei Applikationen, die auf eine gewisse Generik angewiesen sind. Das DynAny-Interface, das zum ersten Mal in der CORBA 2.2 Spezifikation als optionaler Bestandteil beschrieben wurde, soll genau die o. a. Flexibilität gewährleisten.

Das DynAny-API ist die Komposition von mehreren Schnittstellen, wobei die Schnittstelle *DynAny* als Basisklasse für die Ableitungen *DynFixed*, *DynStruct*, *DynSequence*, *DynEnum*, *DynUnion* dient. Diese Schnittstellen sind in dem Namensraum des CORBA-Moduls definiert.

DynAny ist ein lokales Interface, d. h. es können DynAny-Objekte erzeugt werden, aber diese Objekte sind lokal in dem Sinne, dass sie ihren Adressraum nicht verlassen können. Insbesondere heißt das, dass DynAny-Objekte nicht zwischen Client- und Serverobjekt ausgetauscht werden können und dass es nicht möglich ist, eine Objektreferenz eines DynAny-Objektes als so genannte *stringified IOR* wegzuschreiben. Um zur Laufzeit generierte Datenstrukturen an einen Empfänger zu senden besteht jedoch die Möglichkeit, die in 1.2 beschriebenen TypeCodes zur Laufzeit zu generieren, die Daten inklusive TypeCode in ein Any-Objekt einzustellen und an den Empfänger zu senden. Der Empfänger der Nutzdaten hat nun die undankbare Aufgabe, anhand des TypeCodes und der zusätzlichen Parameter die Daten zu interpretieren und mit ihnen konkret zu arbeiten.

Die Funktionalität eines DynAny-Objektes fällt in die Kategorien
1. Operation zum Erzeugen von DynAny-Objekten oder Ableitungen des Interfaces DynAny.
2. Operationen zum Kopieren oder Löschen von DynAny-Objekten.
3. TypeCode-Operationen zum Setzen und Auslesen der TypeCodes aus DynAny-Objekten.
4. Operationen zum Einstellen von skalaren Datentypen, um damit komplexe Objekte zu kreieren.
5. Operationen zum Auslesen von Werten aus einem DynAny-Objekt.
6. Operationen zum Iterieren über die in einem DynAny-Objekt enthaltenen Werte.
7. Operationen zur Konvertierung eines DynAny-Objektes in einen Any-Wert und umgekehrt.

Bevor mit den einzelnen DynAny-Objekten gearbeitet wird, müssen geeignete Objekte erzeugt werden. Die einzelnen Ableitungen von DynAny werden in späteren Unterkapiteln beschrieben.

6.4.1 Das Erzeugen von DynAny-Objekten

Ein DynAny-Objekt oder Ableitungen werden immer über das lokale ORB-Objekt, das bei der Initialisierung an den Aufrufer zurückgegeben wird, erzeugt. Die OMG definiert dazu in dem Namensraum ORB die folgenden Methoden:

1. DynAny create_dyn_any(in any a): Ein DynAny-Objekt wird durch ein Any-Objekt erzeugt und initialisiert. Diese Methode ist sinnvoll, da ein DynAny-Objekt lokal ist und lediglich serialisiert in einem Any-Objekt übertragen werden kann.
2. DynAny create_basic_dyn_any(in TypeCode tc): Ein Basisobjekt wird anhand eines gegebenen TypeCodes erzeugt. Ist der TypeCode z. B. die Beschreibung eines beliebigen IDL-Datentyps, so kann mit diesem Basisobjekt der hierzu korrespondierende Wert nachgebildet werden.
3. DynStruct create_dyn_struct(in TypeCode tc): Erhält z. B. der Klient einen TypeCode zu einer ihm nicht bekannten Struktur, so kann hiermit eine äquivalente dynamische Struktur erzeugt und deren Inhalt kann durch geeignete, noch zu beschreibende Zugriffsmethoden, manipuliert werden.
4. DynSequence create_dynsequence(in TypeCode tc): Analog für Sequenzen.
5. DynArray create_dyn_array(in TypeCode tc): Analog für Datenfelder.
6. DynUnion create_dyn_union(in TypeCode tc): Analog für Unions.
7. DynEnum create_dyn_enum(in TypeCode tc): Analog für Aufzählungstypen.

Grundsätzlich gilt, dass alle Methoden nur DynAny-Objekte oder eine ihrer Ableitungen erzeugen können, wenn der übergebene TypeCode genau den gewünschten Typ repräsentiert. Anderenfalls wird die Ausnahme *InconsistentTypeCode* ausgelöst.

6.4.2 Die DynAny-Schnittstellenbeschreibung

Die Schnittstelle DynAny enthält eine Reihe von Methoden, mit denen Objekte dieses Typs kopiert, zugeordnet und wieder zerstört werden können. Von besonderem Interesse sind zusätzlich die Zugriffsmethoden und die Möglichkeit, ein DynAny-Objekt in ein Any-Objekt zu überführen und wieder zu extrahieren.

Die (gekürzte) IDL-Beschreibung lautet:

```
interface DynAny {
    exception Invalid {};
    exception InvalidName {};
    exception TypeMismatch {};
    exception InvalidSeq {};

    // Zuordnung, Kopieren und Zerstören

    void       assign(in DynAny d) raises(Invalid);
    DynAny     copy();
    void       destroy();

    // Zugriffsmethoden

    void       insert_boolean(in boolean b) raises(InvalidType);
    boolean    extract_boolean() raises(TypeMismatch);

    // entsprechend für alle anderen IDL-Datentypen wie long, double etc.
};
```

Programmbeispiel 35: Das Interface DynAny in IDL-Beschreibung

Mit den Zugriffsmethoden kann die Komposition und Dekomposition unbekannter Objekte zur Laufzeit realisiert werden und mit der Möglichkeit, dieses neue Objekt in ein Any-Objekt einzustellen, kann es an einen Empfänger verschickt werden.

Der Empfänger hat nach Empfang des Any-Objektes zwei Möglichkeiten:
1. Der Aufbau des Objektes ist bekannt. In diesem Fall ist es sinnvoll, aus dem erhaltenen Any ein neues DynAny-Objekt zu erzeugen und mit den Zugriffsmethoden die Daten auszulesen.

2. Der Objektaufbau ist nicht bekannt, in diesem Fall kann über den TypeCode, der mit dem Any-Objekt geliefert wird, der Aufbau ermittelt und alle Daten können extrahiert werden.

Zusätzlich werden in der o. a. Schnittstelle Methoden zur Verfügung gestellt, die es erlauben, über den Inhalt eines DynAny-Objektes zu iterieren:

```
interface DynAny {
    ...
    // Methoden für die Iteration
    DynAny     current_component();
    boolean    next();
    boolean    seek(in long index);
    void       rewind();
};
```

Programmbeispiel 36: Methoden für die Iteration in einem DynAny-Objekt in IDL

Die Bedeutung dieser Methoden ist die folgende:
1. *current_component()*: Liefert das Objekt an der ggw. Position als DynAny-Objekt an den Aufrufer.
2. *next()*: Inkrementiert die ggw. Position und liefert den Wert true, falls die neue Position der Anfangspunkt eines gültigen, enthaltenen Objektes ist. Wird diese Methode auf ein DynAny-Objekt angewendet, das lediglich primitive Datentypen enthält (z. B. einen String), so wird der Wert false zurückgegeben.
3. *seek()*: Bewegt den Positionszeiger an die angegebene neue Position, wobei ein Wert von 0 die erste mögliche Position bezeichnet. seek liefert true, falls die neue Position gültig ist und false anderenfalls.
4. *rewind()*: Entspricht einem *seek(0)*.

6.4.3 Beispiel: Die Verwendung von DynAny-Objekten

Im Folgenden wird ein Beispiel über den Umgang mit DynAny-Objekten angegeben:

```
import org.omg.CORBA.*;
import org.omg.DynamicAny.*;
import utilities.Init;

public class DynAnyTest {
    public static void main(String[] args)
    {
        try {
                Init init = new Init();
                init.initClient(args);
                ORB orb = init.getORB();

                // Anfordern des Factory-Objektes, anpassen an den richtigen Typ.

                org.omg.CORBA.Object
                        obj = orb.resolve_initial_references("DynAnyFactory");
                DynAnyFactory df = DynAnyFactoryHelper.narrow(obj);


                // in diesem Beispiel soll erst ein long-Wert eingestellt werden. Der TypeCode
                // von long wird ueber das orb-Objekt ermittelt. Der long-Wert 20 wird
                // anschliessend direkt in das DynAny-Objekt eingestellt.

                TypeCode                    tc     = orb.get_primitive_tc(TCKind.tk_long);
                org.omg.DynamicAny.DynAny   da     = df.create_dyn_any_from_type_code(tc);
                da.insert_long(20);


                // Any-Kontainer erzeugen und das DynAny-Objekt zu einem Any-Objekt
                // serialisieren. Damit waere das Objekt ueber das Netzwerk uebertragbar.

                Any a = orb.create_any();
```

```
                a = da.to_any();


                // auf der Klientseite koennte der Wert wie folgt extrahiert werden.

                long x = a.extract_long();
                System.out.println("Es war ein long: " + x);

                // Wiederverwendung des TypeCode- und DynAny-Objektes, diesmal mit einem
                // double-Wert.

                tc = orb.get_primitive_tc(TCKind.tk_double);
                da = df.create_dyn_any_from_type_code(tc);
                da.insert_double(Math.E);
                a = da.to_any();
                double d = a.extract_double();
                System.out.println("Es war ein double: " + d);
                da.destroy();
        }
        catch (Exception e) {
                System.err.println(e.getMessage());
                System.exit(1);
        }
        System.exit(0);
    }
}
```

Programmbeispiel 37: Verwendung von DynAny-Objekten

6.4.4 Die einzelnen Ableitungen von DynAny

Von besonderem Interesse für den Aufbau komplexer Objekte ist DynStruct. DynStruct verfolgt einen ähnlichen Ansatz wie die dynamische Schnittstelle; es werden für die aufzunehmenden Daten *NamedValue*-Paare gebildet zusammen mit einem optionalen Namen für den zu den Daten gehörenden TypeCodes, dem TCKind Attribut und zwei Methoden zum Setzen bzw. Lesen von Werten.

Die IDL-Beschreibung von DynStruct lautet:

```
typedef string FieldName;
struct NameValuePair {
    FieldName  id;
    any        value;
};
typedef sequence<NameValuePair> NameValuePairSeq;
interface DynStruct : DynAny {
    FieldName          current_member_name();
    TCKind             current_member_kind();
    NameValuePairSeq   get_members();
    void               set_members(in NameValuePairSeq values) raises(InvalidSeq);
};
```

Programmbeispiel 38: DynStruct-Beschreibung in IDL

Anhand der IDL-Bechreibung wird bereits deutlich, dass eine Struktur vom Typ DynStruct eine beliebige Folge von Daten aufnehmen kann. Alle Daten der DynStruct sind vom Datentyp Any. Damit kann jedes im Vorfeld durch IDL beschriebene Datum aufgenommen werden. Für das Einstellen unbekannter, nicht in IDL beschriebener Daten kann erst ein DynAny-Objekt erzeugt, mit Daten gefüllt werden und anschließend mit der Methode to_any in ein Any-Objekt eingestellt werden, das anschließend in die Struktur DynStruct übernommen wird.

Die o. a. Schnittstelle entspricht intuitiv unserem Verständnis einer IDL-Struktur. Es können beliebig viele Elemente verschiedenen Typs aufgenommen werden und zu jedem Element

kann der Typ ermittelt werden, bzw. der tatsächliche Datentyp, falls das Element mit einem Aliasnamen versehen war.

DynUnion ist ähnlich aufgebaut wie DynStruct. Der Diskriminator der Union wird als ein Objekt vom Typ DynAny gehalten und bei Bedarf zurückgegeben.

```
interface DynUnion : DynAny {
    DynAny                  discriminator();
    TCKind                  discriminator_kind();
    attribute boolean       set_as_default;
    DynAny                  member();
    TCKind                  member_kind();
    attribute FieldName     member_name;
};
```

Programmbeispiel 39: DynUnion-Beschreibung in IDL

Das Objekt DynSequence besteht lediglich aus seiner Länge und Methoden zum Einstellen bzw. Auslesen der Sequenz:

```
typedef sequence<any> AnySeq;

interface DynSequence : DynAny {
    attribute unsigned long   length;
    AnySeq                    get_elements();
    void                      set_elements(in AnySeq value) raises(InvalidSeq);
};
```

Programmbeispiel 40: DynSequence-Beschreibung in IDL

Aufgrund der attribute Vereinbarung werden von dem IDL-Compiler die beiden Operationen

```
    void length(unsigned long l);
```

und

```
    unsigned long length();
```

generiert.

DynArray ist ähnlich aufgebaut wie DynSequence, lediglich die Längenangabe fehlt. Dies ist darauf zurückzuführen, dass die IDL-Definition eines array's immer von einer festen Länge ist. Mit den nachfolgenden Methoden kann ein Array angelegt bzw. ausgelesen werden.

```
interface DynArray : DynAny {
    AnySeq     get_elements();
    void       set_elements(in AnySeq value) raises(InvalidSeq);
};
```

Programmbeispiel 41: DynArray-Beschreibung in IDL

DynFixed arbeitet mit Datentypen, die als Octetstream übergeben werden. Die IDL von DynFixed weist lediglich Methoden für das Setzen bzw. Lesen eines derartigen Datums aus.

```
typedef sequence<octet> OctetSeq;

interface DynFixed : DynAny {
    OctetSeq   get_elements();
    void       set_elements(in OctetSeq value) raises(InvalidSeq);
};
```

Programmbeispiel 42: DynFixed-Beschreibung in IDL

DynEnum verfügt über vier Methoden. Es handelt sich hierbei um Methoden zum Setzen der Werte als String und als long, sowie Methoden zum Lesen der Werte. In Anlehnung an die

Vorgehensweise des IDL-Compilers werden diese Methoden als attribute in der Schnittstelle definiert.

```
interface DynEnum : DynAny {
    attribute string          value_as_string;
    attribute unsigned long   value_as_long;
};
```

Programmbeispiel 43: DynEnum-Beschreibung in IDL

6.5 Beispiel für die Erzeugung unbekannter Objekte zur Laufzeit

In dem nachfolgenden Beispiel werden zwei dynamische CORBA-Komponenten bzgl. einer Datenstruktur erzeugt, die im Vorfeld nicht bekannt, also nicht in IDL beschrieben ist. Um das Beispiel nicht unnötig zu verkomplizieren wird zur Laufzeit eine Struktur aufgebaut, die als Informationen einen Namen, einen Vornamen und eine Altersangabe enthält.

Dazu sind die folgenden Schritte notwendig:
1. Die aufzubauende Struktur muss in einer Sequenz von *StructMembern* beschrieben sein. Dies beinhaltet einen konstanten Namen, den das Datum innerhalb der Struktur haben soll und einen konkreten Datentyp.
2. Anhand der unter (1) angeführten Sequenz muss dynamisch ein TypeCode für eine Struktur vom Typ *DynStruct* erzeugt werden.
3. Die Werte der Struktur müssen einzeln in Any-Objekte verpackt werden.
4. Die Any-Objekte werden mit den unter (1) angeführten Namen in eine Struktur vom Typ *NamedValuePairSeq* eingestellt.
5. Das DynStruct-Objekt wird instanziiert.
6. Die NamedValuePairSeq-Sequenz wird an das DynStruct-Objekt übergeben, welches sich anschließend selbst initialisiert.
7. Das DynStruct-Objekt wird in ein Any-Objekt transformiert und kann anschließend an beliebige Empfänger versendet werden.

Der Empfänger überführt anhand der überlieferten Metadaten das Any-Objekt in ein neues DynStruct-Objekt und verwendet die enthalten Iteratoren um zu jedem einzelnen Element der Struktur zu gelangen. Für jedes Element existiert wieder rum ein separater TypeCode, der Auskunft über das aktuelle Datum erteilt.

In der seltenen Literatur, die sich mit DynAny-Objekten beschäftigt, findet man ausschließlich Beispiele für das Arbeiten mit diesen Komponenten, die auf statischem IDL basieren. Zum einen ist damit der Sinn dieser dynamischen Komponenten in Frage zu stellen und zum anderen wird ein falscher Eindruck erweckt.

Das folgende Beispiel verwendet eine einfache IDL-Beschreibung, in der lediglich die anzusprechende Methode vereinbart wird. Als Rückgabewert wird ein Any erwartet.

Die IDL-Beschreibung lautet:

```
module dynany {
    interface UnknownInterface {
        any receive();
    };
};
```

Programmbeispiel 44: IDL-Beschreibung für unbekannte Objekte

In der Objektimplementierung wird zur Laufzeit eine noch nicht in IDL beschriebene Datenstruktur aufgebaut. Bzgl. dieser Datenstruktur wird auch dynamisch ein TypeCode erzeugt.

```
// UnknowObjectCreationImpl.java: Zur Laufzeit erstellt die Methode ein Objekt vom Typ
// DynStruct und füllt die Datenmember. Dazu muss ein dynamischer TypeCode erzeugt werden, der
// den Aufbau dieser nicht in IDL beschriebenen Datenstruktur beschreibt.
//
// $Author$ $Revision$
// $Date$   $State$

package dynany;

import java.io.IOException;
import org.omg.CORBA.*;
import org.omg.DynamicAny.*;
import org.omg.CORBA.*;
import utilities.Init;


public class UnknownObjectCreationImpl extends UnknownInterfacePOA {
    public UnknownObjectCreationImpl(ORB orb)
    {
        this.orb = orb;
        System.out.println("Constructor okay");
    }


    public Any receive()
    {
        org.omg.DynamicAny.DynStruct da = null;
        TypeCode structTC = null;
        try {

                // die zur Laufzeit zu erstellende Struktur wird durch einen Array von
                // StructMember-Elementen beschrieben. Jedes Element hat einen Name
                // und einen Typ.

                org.omg.CORBA.StructMember[] members = new org.omg.CORBA.StructMember[3];
                members[0] = new org.omg.CORBA.StructMember();
                members[1] = new org.omg.CORBA.StructMember();
                members[2] = new org.omg.CORBA.StructMember();
                members[0].name = "Name";
                members[0].type = orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string);
                members[1].name = "Vorname";
                members[1].type = orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string);
                members[2].name = "Alter";
                members[2].type = orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_short);


                // TypeCode fuer die oben beschriebene Struktur zur Laufzeit erzeugen.

                structTC = orb.create_struct_tc(UnknownInterfaceHelper.id(),
                                                "KurzinfoPerson",
                                                members);
                if (structTC == null) {
                        System.err.println("tc null");
                        return null;
                }


                // Die einzelnen Daten sind immer vom Typ Any.

                Any n, v, a;
                n = orb.create_any();
                v = orb.create_any();
                a = orb.create_any();
                n.insert_string("Feinbein");
                v.insert_string("Hugo");
                a.insert_short((short) 46);


                // Zuordnung der Daten an die neu erzeugte Struktur und ihre Datenmember.

                org.omg.DynamicAny.NameValuePair[] memberSeq =
                        new org.omg.DynamicAny.NameValuePair[3];
```

```
            memberSeq[0] = new org.omg.DynamicAny.NameValuePair();
            memberSeq[1] = new org.omg.DynamicAny.NameValuePair();
            memberSeq[2] = new org.omg.DynamicAny.NameValuePair();
            memberSeq[0].id    = "Name";
            memberSeq[0].value = n;
            memberSeq[1].id    = "Vorname";
            memberSeq[1].value = v;
            memberSeq[2].id    = "Alter";
            memberSeq[2].value = a;


            // DynStruct-Objekt erzeugen

            org.omg.CORBA.Object obj = orb.resolve_initial_references("DynAnyFactory");
            org.omg.DynamicAny.DynAnyFactory df =
                        org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);
            da = (org.omg.DynamicAny.DynStruct)
                        df.create_dyn_any_from_type_code(structTC);

            if (da == null)
                   return null;


            // mit der neuen Struktur fuellen.

            da.set_members(memberSeq);
        }
        catch(Exception ex) {
               ex.printStackTrace();
               System.exit(4);
        }
        return da.to_any();
    }


    private org.omg.CORBA.ORB orb = null;
}
```

Programmbeispiel 45: Erzeugen unbekannter Objekte

Für den Klienten ergibt sich nun das Problem, dass er einen Any-Kontainer erhält, der zu interpretieren ist. Bei einfachen Datenstrukturen, die in dieses Any-Objekt eingestellt sind, kann dies, wie in Programmbeispiel 32 gezeigt wurde, durch TypeCodes geschehen. In komplizierteren Fällen bietet sich auch in diesem Fall DynAny, bzw. eine seiner Ableitungen, an, um die Datenstruktur zu erforschen.

Unterstellt man, dass in dem Any-Kontainer ein Datum vom Typ DynStruct eingestellt ist, so könnte die Klientenseite wie folgt verfahren:

```
import java.io.IOException;
import org.omg.DynamicAny.*;
import org.omg.CORBA.*;
import utilities.Init;
import dynany.*;

public class Client {
    public Client(String[] args)
    {
        init = new Init();
        init.initClient(args);
        orb = init.getORB();
    }


    // stringifiedIOR aus der angegebenen Datei lesen und in eine Objektreferenz konvertieren.

    public void readIORs()
    {
        try {
            String refFile = "/tmp/DynAny.ref";
            String inIOR   = init.readIOR(refFile);

            org.omg.CORBA.Object obj = orb.string_to_object(inIOR);
```

```
            if (obj == null) {
                System.err.println("Korrupte IOR");
                System.exit(-1);
            }
            in = dynany.UnknownInterfaceHelper.narrow(obj);
            if (in == null)
                System.err.println("Fehler bei narrow");
                System.exit(-2);
            }
        }
        catch (IOException ioe) {
               ioe.printStackTrace();
               System.exit(-3);
        }
        catch (Exception e) {
               System.err.println("Message: " + e.getMessage());
               e.printStackTrace();
               System.exit(-4);
        }
    }

    public static void main(String[] args)
    {
        Client c = new Client(args);
        c.readIORs();
        c.run();
        System.exit(0);
    }


    // das Serverobjekt ansprechen und den Rueckgabewert auswerten.

    public void run()
    {
        org.omg.DynamicAny.DynAnyFactory factory = null;
        Any a = in.receive();
        try {
               factory = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(
                                          orb.resolve_initial_references("DynAnyFactory"));
        }
        catch (Exception e) {
               e.printStackTrace();
               System.exit(-5);
        }


        // aus dem Any-Wert den TypeCode und anschliessend den Beschreibungstyp ermitteln.
        // Es kann nur ein Datum vom Typ Struktur gewesen sein. Normalerweise muesste der
        // Klient über alle bekannten Beschreibungstypen iterieren.

        TypeCode tc = a.type();
        TCKind   tk = tc.kind();

        if (tk != TCKind.tk_struct) {
               System.err.println("Eine Struktur wurde erwartet");
               System.exit(-6);
        }

        try {
               // da es eine Struktur war, wird der erhaltene Any-Wert in Objekt vom Typ
               // DynStruct eingestellt.

               org.omg.DynamicAny.DynStruct info = org.omg.DynamicAny.DynStructHelper.narrow(
                                          factory.create_dyn_any(a));


               // die Methoden von DynAny nutzen, um die einzelnen Datenmember der Struktur zu
               // erhalten.

               info.rewind();
               org.omg.DynamicAny.DynAny da = info.current_component();
               display(da);
               while (info.next()) {
                      da = info.current_component();
                      display(da);
               }
        }
        catch (Exception e) {
```

```
                e.printStackTrace();
                System.exit(2);
            }
        }
        // Ausgabe bzgl. des Typs und des aktuellen Wertes der einzelnen Datenmember der Struktur.

        public void display(org.omg.DynamicAny.DynAny da) throws Exception
        {
            TypeCode tc = da.type();
            TCKind   tk = tc.kind();
            Any      a  = da.to_any();

            if (tk == TCKind.tk_short) {
                   System.out.println("Typ: short, Wert: " + a.extract_short());
            }
            else if (tk == TCKind.tk_string) {
                   System.out.println("Typ: string, Wert: " + a.extract_string());
            }
            else
                   System.out.println("Unbekannter Typ");
        }


        //
        // private data member
        //

        private Init                     init;
        private org.omg.CORBA.ORB        orb;
        private dynany.UnknownInterface  in;
}
```

Programmbeispiel 46: DynStruct auf der Klientenseite

6.6 Eine Zukunftsvision

Möglicherweise wird es zukünftig intelligente Softwareagenten geben, die von ihrem Auftraggeber mit einem Regelwerk versehen werden und sich anschließend selbständig im Internet bewegen, um diesen Auftrag zu erfüllen. Als Anmerkung sei an dieser Stelle angeführt, dass die später zu beschreibenden Web Services diesen Weg auf Basis von Internettechnologien konsequent beschreiten.

Stößt der Agent hierbei auf einen Trader, also einen Dienst, der Dienstleistungen exportiert, so besteht die Notwendigkeit, zur Laufzeit Informationen über die Schnittstelle der exportierten Objekte in Erfahrung zu bringen. Dies beinhaltet

a. die Methodennamen und ihre Signatur.
b. einen möglichen Rückgabewert.
c. mögliche Ausnahmen, die von den Methoden ausgelöst werden können.
d. den Aufbau der benutzerdefinierten Datenstrukturen.

All diese Informationen müssen zur Verfügung gestellt werden und es muss dem Agenten möglich sein, diese Informationen lediglich über eine Objektreferenz, die er von dem Trader bekommt, zu erfragen.

Anschließend kann der Agent zur Laufzeit basierend auf diesen Informationen ein so genanntes *Request-Pseudo-Objekt* aufbauen und hierüber seine Anfragen an das entfernte Objekt stellen. Für das entfernte Objekt ist es völlig transparent, ob es dynamisch, also durch ein Request-Pseudo-Objekt oder statisch, also durch einen Klientenstub angerufen wird.

Die Informationsgewinnung bzgl. des unbekannten, entfernten Objektes kann unter CORBA durch das Interface Repository realisiert werden. Der Aufbau von Request-Pseudo-Objekten und die Transparenz der Methodenaufrufe kann durch die von CORBA spezifizierte dynamische Schnittstelle vorgenommen werden.

Das angeführte Szenario mit dem Softwareagenten ist mit heutiger Technik bereits realisierbar. Das noch zu lösende Problem ist die Semantik der Rückgabedaten. Erhält der Agent z. B. eine Struktur P mit den Datenelementen *Name* und *Vorname* als Strings, so kann man wohl mutmaßen, dass es sich um eine Person handeln kann, definitiv weiß der Agent dies jedoch nicht. Solange kein Standard existiert, der eine eindeutige Beschreibung der Semantik liefert, wird unser Agent bestenfalls mit den Daten zu seinem Auftraggeber zurückkehren und hier werden mit menschlichem Eingriff die gelieferten Daten interpretiert.

In den Kapiteln über WebServices wird dargelegt, dass man bzgl. dieser Vision Fortschritte basierend auf Internettechnologie zu verzeichnen hat.

6.7 Das Interface Repository

In geschlossenen Systemen sind alle Arten von Objekten, die anwendungsrelevant sind, bereits zur Programmentwicklung bekannt. In *offenen Systemen* ist die Situation grundsätzlich anders: Objekte, mit denen eine Anwendung während ihrer Ausführung Kontakt aufnehmen kann, sind gar nicht oder nur teilweise bekannt.

Für die Beschreibung dieser entfernten Objekte definiert die OMG ein standardisiertes Archiv für die Speicherung aller in IDL beschriebenen Objekte und Daten.

Definition 35: Interface Repository (IR)
Das von der OMG standardisierte Archiv für die zur Verfügungstellung aller in IDL beschriebenen Objekte und Daten wird als *Interface Repository* (IR) bezeichnet.

Das IR ist Bestandteil des ORB und stellt eine standardisierte Schnittstelle zur Verfügung, die es einem Klienten erlaubt, die Archivdaten zur Laufzeit abzufragen. Die in einer IDL-Definition enthaltenen Informationen werden als CORBA-Objekte repräsentiert und es kann damit von jedem CORBA-konformen Klienten oder Server darauf zugegriffen werden.

Es existiert aufgrund der engen Verwandtschaft zu einer IDL-Beschreibung immer die Möglichkeit zu einer beinahe Eins-zu-eins-Abbildung zwischen IDL und dem Interface Repository zu gelangen.

6.7.1 Objekte des Interface Repositories

Jede in IDL beschriebene Vereinbarung wird in dem Interface Repository durch eine Menge von Objekten repräsentiert (vgl. [Hoque, 1998]).

Diese Objekte sind:
1. *Repository*: Das Repository-Objekt ist für jede IDL-Beschreibung als Einstiegspunkt in das Interface Repository zuständig. Es kann je nach IDL-Beschreibung Module, Schnittstellenbeschreibungen, benutzerdefinierte Datentypen, Ausnahmen und Konstanten beinhalten.
2. *ModuleDef*: Es handelt sich hierbei um eine logische Gruppierung von Submoduln, Schnittstellen, benutzerdefinierten Datenstrukturen, Ausnahmen und Konstanten. Die Analogie zur IDL-Beschreibung ist das Schlüsselwort *module*.
3. *InterfaceDef*: Repräsentiert die IDL-Schnittstellenbeschreibung und kann somit neben den Operationen, den Signaturen für die Methoden, den möglichen Rückgabewerten und eventuellen Ausnahmen auch benutzerdefinierte Datenstrukturen und IDL-Attribute enthalten.
4. *AttributeDef*: Beschreibt die eventuellen Attribute einer Schnittstelle.

5. *OperationDef*: Beschreibt die Operationen der Schnittstelle, ihre Signaturen, Rückgabewerte und eventuelle Ausnahmen, die von diesen Operationen ausgelöst werden können.
6. *TypeDef*: Ist die Basisklasse für alle benutzerdefinierten Datenstrukturen.
7. *ConstantDef*: Die Definition von Konstanten.
8. *ExceptionDef*: Die Definition von Ausnahmen, die von Operationen ausgelöst werden können.

Der folgenden Abbildung (vgl. [Hoque, 1998]) kann entnommen werden, dass vor allem Schnittstellendefinitionen, aber auch Definitionen über Konstanten und Typen, wobei zu den Typen auch die zugehörigen TypeCodes gehören, abgelegt werden. Die Einträge im IR werden, um der Gefahr der Mehrdeutigkeit zu entgehen, jeweils mit einer global eindeutigen ID (Repository-Id) versehen.

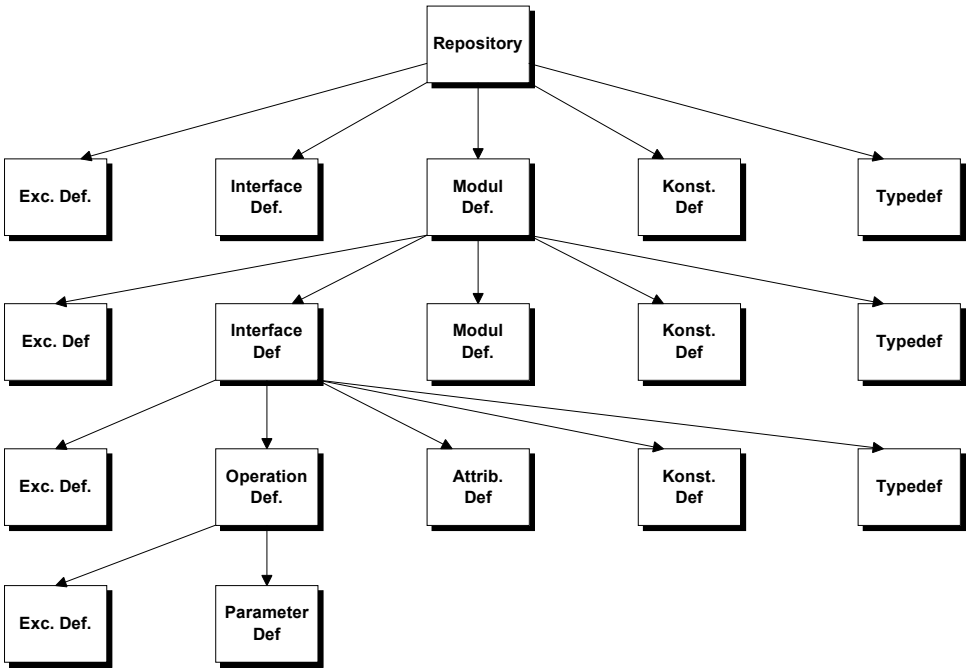


Abbildung 17: Aufbau eines Interface Repository

6.7.2 Die Klassenhierarchie des Interface Repositories

Die OMG definierte drei abstrakte Schnittstellen, die als Basis für die Traversierung durch das Interface Repository bestimmt sind. In Abbildung 18 wird die Ableitungshierarchie dargestellt.

Zu jedem in IDL benutzerdefinierten Datentyp und Schnittstelle existiert ein Objekt, wobei die Namensbildung der Regel genügt, mit einem Großbuchstaben anzufangen und das Suffix *Def* anzuhängen. Also aus *interface* wird *InterfaceDef*, aus *struct* wird *StructDef*, aus *module* wird *ModuleDef* usw.

Es ist wichtig festzuhalten, dass die Objekte ExceptionDef, AttributeDef, ConstantDef, TypeDef und ParameterDef keine weiteren Objekte enthalten, sondern in anderen Objekten enthalten sein können.

Bei den anderen Objekten wie z. B. InterfaceDef ist beides möglich. Eine IDL-Schnittstellenbeschreibung kann innerhalb eines Namensraumes enthalten sein, und es kann selbst wieder ein Namensraum z. B. für Ausnahmen, Datenstrukturen usw. sein.

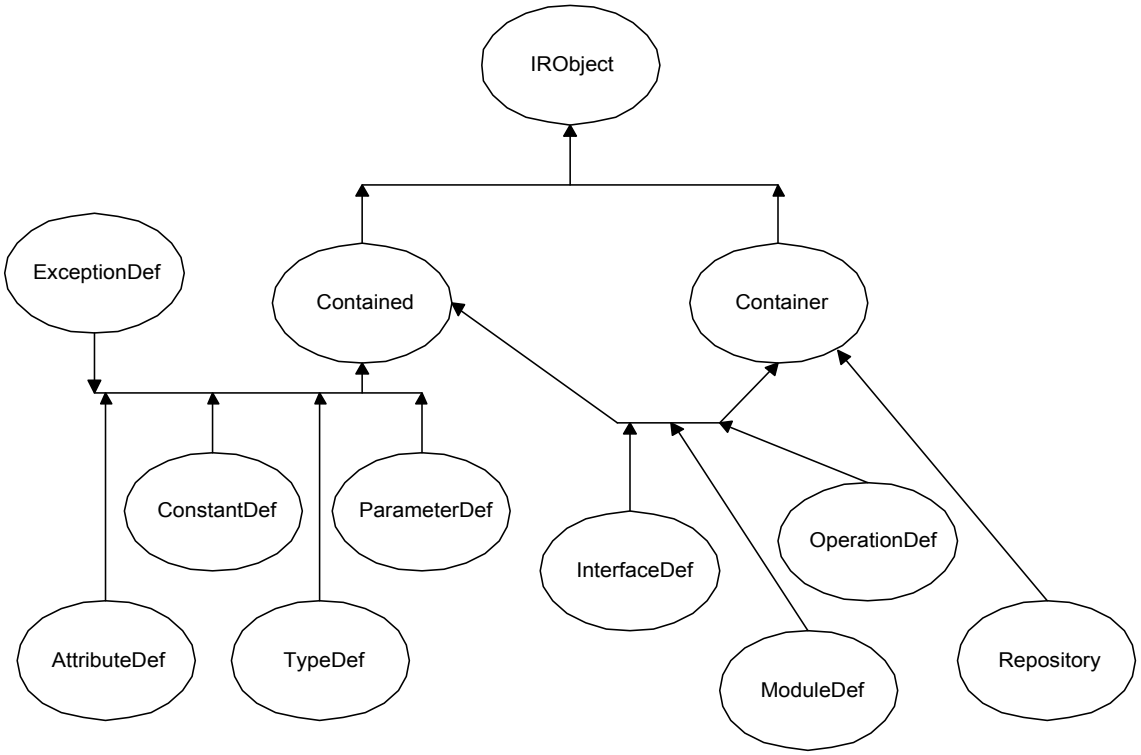


Abbildung 18: Die Klassenhierarchie des Interface Repositories

Die in Abbildung 18 (vgl. [Hoque, 1998]) angeführten Objekte haben die folgende Bedeutung:

1. *IRObject*: Dieses Objekt ist die Basisklasse aller Objekte in einem Interface Repository und somit das Wurzelobjekt.
2. *Container*: Repräsentiert alle enthaltenen Objekte eines Interface Repositories. Dieses Objekt stellt eine Methode *lookup* zur Verfügung, mit der gezielt nach anderen enthaltenen Objekten gesucht werden kann.
3. *Contained*: Ist eine Subklasse von IRObject und wird von allen Objekten beerbt, die in anderen enthalten sind. Das Objekt Contained stellt eine Reihe von Informationen über das Zielobjekt zur Verfügung. Dies sind:
   a. Eine RepositoryId mit dem Attributnamen *id*. Damit wird eine in IDL beschriebene Schnittstelle eindeutig bestimmt.
   b. Ein Bezeichner vom Typ *string* mit dem Attributnamen *name*.
   c. Eine Versionsnummer (VersionSpec) mit dem Attributnamen *version*. Dies entspricht den IDL-Möglichkeiten, eine Schnittstelle mit verschiedenen Versionen zu versehen.
   d. Eine Angabe zu dem Container-Objekt mit dem Attributnamen *contained_in*.
   e. Ein absoluter Name vom Typ ScopedName und dem Attributnamen *absolute_name*.
   f. Die Angabe zu dem Repository-Objekt mit dem Attributnamen *containing_repository*.

Neben den abstrakten Basisklassen definiert die OMG eine Reihe weiterer Klassen, die in sehr engem semantischem Zusammenhang mit dem Aufbau von IDL-Beschreibungen zu sehen sind (vgl. Abbildung 17). Dazu zählen:

1. *Repository*: Ein Repository definiert genau eine Instanz in dem Interface Repository und enthält alle dem IR bekannt gegebenen IDL-Beschreibungen. Dazu zählen

   a. die Definition von Konstanten (ConstantDef). Enthalten ist hier der TypeCode der Konstanten, sein IDL-Typ und ein Any-Objekt mit dem Wert der Konstanten.
   b. die Definition benutzerdefinierter Datentypen (TypeDef). Enthalten ist hier eine Angabe über die IDL-Typendefinition und in welchem Namensraum sie enthalten ist, also eine Angabe zu dem Objekt Contained.
   c. Angaben zu möglichen Ausnahmen (ExceptionDef). Enthalten ist hier die Definition einer benutzerdefinierten Ausnahme mit den Angaben TypeCode der Ausnahme und einer Folge von möglichen enthalten Elementen. Dies entspricht den IDL-Möglichkeiten, bei der Definition einer Ausnahme weitere Informationen wie z. B. eine Fehlermeldung oder eine Fehlernummer innerhalb der Ausnahme zu vereinbaren.
   d. Angaben zu den Namensräumen (ModuleDef). Analog zu der IDL-Beschreibung können in der Moduldefinition wieder Angaben zu den Schnittstellen, Datentypen etc. enthalten sein.
   e. Angaben zu den Schnittstellen (InterfaceDef). Ebenso wie bei der Modulbeschreibung finden sich hier u. U. weitere Angaben zu Konstanten, Datentypen, Ausnahmen usw.
2. *ModuleDef*: Definiert alle enthaltenen Informationen innerhalb eines Namensraums, also alle IDL-Definitionen innerhalb eines mit dem Schlüsselwort *module* eingeleiteten Bereiches.
3. *InterfaceDef*: Entspricht dem IDL-Schlüsselwort *interface* und enthält alle Informationen innerhalb des Gültigkeitsbereiches einer IDL-Schnittstelle. InterfaceDef enthält zusätzlich die Möglichkeit, Basisschnittstellen abzufragen durch
   InterfaceDefSeq base_interfaces
4. *AttributeDef*: Beschreibt die eventuellen Attribute, die innerhalb einer Schnittstelle vereinbart wurden. Diese Klasse enthält keine weiteren Objekte, sondern lediglich Informationen wie den TypeCode des Attributes, seinem IDL-Typ und eine Angabe darüber, ob diese Attribut in der IDL-Beschreibung auf *readonly* gesetzt wurde oder nicht.
5. *OperationDef*: Beschreibt die Operationen einer Schnittstelle, enthält selbst keine weiteren Objekte. Zu der Beschreibung der Methoden zählt:
   a. Der TypeCode des Rückgabewertes der Operation (TypeCode result).
   b. Der IDL-Typ der Rückgabe (IDLType result_def).
   c. Eine Beschreibung der Signatur vom Typ ParDescriptionSeq (ParDescriptionSeq params).
   d. Einer Angabe darüber, ob die Operation mit dem IDL-Schlüsselwort *oneway* versehen ist oder nicht.
   e. Einer Folge von möglichen Kontexten (ContextIdSeq contexts), die bei dem Methodenaufruf mitzugeben sind (s. GIOP).
   f. Eine Folge von möglichen Ausnahmen, die diese Methode auslösen kann (ExceptionDefSeq exceptions).
6. *ConstantDef*: Enthält Informationen über Konstante, die innerhalb des Repositories, in einem Namensraum oder in einer Schnittstelle definiert wurden. ConstantDef enthält keine weiteren Objekte.
7. *ExceptionDef*: Beschreibt die benutzerdefinierten Ausnahmen innerhalb des Repositories, eines Namensraumes oder einer Schnittstelle. Das IDL-Analogon hierfür ist das Schlüsselwort *exception*. Im Sinne einer IDL-Beschreibung kann eine Ausnahme ähnlich einer Struktur Datenelemente aufnehmen. Dies impliziert, dass neben dem TypeCode der Ausnahme auch eine eventuelle Folge der Datenelemente enthalten sein muss (StructMemberSeq members). Festzuhalten ist an dieser Stelle, dass hier nur die Beschreibung der einzelnen benutzerdefinierten Ausnahmen im Repository gespeichert wird. Die mögliche Folge von Ausnahmen, die eine Methode auslösen kann, wird durch *OperationDef* (s. o) festgelegt.
8. *StructDef*: Beschreibt benutzerdefinierte Datenstrukturen einer IDL-Struktur. Die Datenelemente werden genauso wie bei den Ausnahmen in einer Folge von Strukturelementen aufgelistet (StructMemberSeq members).
9. *UnionDef*: Entspricht dem IDL-Typ *union* und enthält Angaben zu
   a. dem TypeCode des Diskriminators (TypeCode discriminator_type).

   b. dem IDL-Type des Diskriminators (IDLType discriminator_type_def), also char, long etc.
   c. einer Folge von möglichen Datenelementen (UnionMemberSeq members).
10. *EnumDef*: Entspricht dem IDL-Aufzählungstypen *enum* und enthält Angaben zu den Datenelementen (EnumMemberSeq members).
11. *AliasDef*: Ist analog zu dem IDL-Schlüsselwort *typedef* zu sehen. Es beinhaltet eine Information über den originären Datentyp (IDLType original_type_def).
12. *StringDef*: Beschreibt analog zu dem IDL-Schlüsselwort *string* ein Wort und stellt die eventuelle obere Schranke des Wortes zur Verfügung.
13. *SequenceDef*: Entspricht der IDL-Vereinbarung über Sequenzen. Als Zusatzinformationen erhält man
   a. die maximale Länge der Sequenz (unsigned long bound).
   b. den TypeCode der Elemente (TypeCode element_type).
   c. den IDL-Typ der Datenelemente (IDLType element_type_def).

   Es ist wichtig, an dieser Stelle darauf hinzuweisen, das eine Sequenz aus jedem in IDL bekannten oder vom Anwender selbstdefinierten Datentypen aufgebaut werden kann.
14. *ArrayDef*: Entspricht der IDL-Vereinbarung *array*. Analog zu der Sequenz kann eine Datenliste ebenfalls aus allen in IDL bekannten oder vom Anwender selbstdefinierten Datentypen aufgebaut werden kann. Es gelten somit die gleichen Zusatzinformationen wie unter (13).
15. *PrimitiveDef*: Definiert die primitiven IDL-Datentypen wie boolean, char, octet, short etc..

6.7.3 Beispiel für den Zugriff auf das IFR

Das folgende Beispiel zeigt ausgehend von einer IDL-Beschreibung den Prozess, zur Laufzeit Informationen aus dem Interface Repository zu gewinnen.

Die IDL-Definition lautet:

```
// Interface.idl

#ifndef INTERFACE_IDL
#define INTERFACE_IDL

module ir {
    // die Basis-Interfaces sind ohne Beschränkung der Allgemeinheit leer. Sie dienen
    // lediglich der Vererbung und der Versionierung

    interface BasisInterface1 { };
    interface BasisInterface2 { };
    interface BasisInterface3 { };

    interface AbgeleiteteSchnittstelle : BasisInterface1, BasisInterface2, BasisInterface3 {
        void methode1(in short i, in any a);
        void methode2(in long l);
        void methode3(in string s, in double d);
        void methode4(in string s, in double d, out short x);

        attribute long attribut1;
        attribute long attribut2;
        attribute long attribut3;
    };
};
#pragma version ir::BasisInterface1         1.7
#pragma version ir::BasisInterface2         5.9
#pragma version ir::BasisInterface3         3.1
#pragma version ir::AbgeleiteteSchnittstelle 9.4
#endif
```

Programmbeispiel 47: Arbeiten mit dem IR. IDL-Beschreibung

Die folgende Java-Applikation kontaktiert den Interface-Repository-Server, liest die interoperable Objektreferenz des gewünschten Objektes als String aus einer Datei und konvertiert diesen String zu einer Objektreferenz. Anhand dieser Referenz versucht der Klient alle vereinbarten IDL-Beschreibungen zu erfragen und auszugeben.

Um in größerer Allgemeinheit Zugriff auf ein Interface Repository, z. B. im Internet, zu erlangen, müsste man in einem Trader neben den Objektreferenzen auch die IOR's für das zugehörige Repository ablegen. Die Zurverfügungstellung eines oder mehrerer zentraler Repositories könnte ähnlich zu den Web Services, speziell UDDI, umgesetzt werden.

```
package repository;

import ir.*;          // vom IDL-Kompilierer generiert
import org.omg.CORBA.*;
import org.omg.CORBA.portable.ObjectImpl;
import utilities.Init;

public class IRClient
{
    public static void main(String[ ] args)
    {
        // Initialisierung des Client-ORBs, Lesen und Anpassen der IOR.

        try {
                Init init = new Init();
                init.initClient(args);

                // Objektreferenz auslesen und anpassen

                String ref = null;
                try {
                        ref = init.readIOR("/tmp/Interface.ref");
                }
                catch(java.io.IOException ex) {
                        System.err.println(ex.getMessage());
                        System.exit(1);
                }

                // Konvertieren in eine Objektreferenz
                // ACHTUNG: org.omg.CORBA.Object ist bei ORBacus ein
                //          Interface. Um _get_interface() zu erhalten
                //          muss eine Abbildung auf
                //              (org.omg.CORBA.portable.ObjectImpl)
                //          erfolgen

                ObjectImpl interf = (ObjectImpl)(init.getORB()).string_to_object(ref);
                if (interf == null){
                        System.err.println("Nullreferenz");
                        System.exit(-1);
                }

                InterfaceDef idef = (InterfaceDef)(interf._get_interface_def());
                if (idef == null) {
                        System.err.println("Kein Interface Repository verfügbar");
                        System.exit(-2);
                }

                org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription desc;
                desc = idef.describe_interface();

                int i;
                System.out.println("name             = " + desc.name       +
                                   "\nid               = " + desc.id         +
                                   "\ndefiniert in     = " + desc.defined_in +
                                   "\nversion          = " + desc.version);
                TypeCode t = desc.type;
                System.out.println(nameTypeIDL[t.kind().value()]);
                System.out.println("Operationen:");
                for(i = 0; i < desc.operations.length; i++) {
                        System.out.println(i + ": " + desc.operations[i].name);
                        System.out.print("\tParameter:\n");
                        for (int j = 0; j < desc.operations[i].parameters.length; ++j) {
                                System.out.print("\t\t" +
```

```
                                        desc.operations[i].parameters[j].name);
                                org.omg.CORBA.TCKind tc =
                                        desc.operations[i].parameters[j].type.kind();
                                System.out.println("(" + nameTypeIDL[tc.value()] + ")");
                        }
                }
                System.out.println("Attribute:");
                for(i = 0; i < desc.attributes.length; i++) {
                        System.out.println(i + ": " + desc.attributes[i].name);
                }
                System.out.println("BasisInterfaces:");
                for(i = 0; i < desc.base_interfaces.length; i++) {
                        System.out.println(i + ": " + desc.base_interfaces[i]);
                }
        }
        catch(SystemException ex) {
                System.err.println(ex.getMessage());
                ex.printStackTrace();
                System.exit(-3);
        }
        System.exit(0);
    }

    public final static String[] nameTypeIDL = {
        "null", "void",         "short", "long", "unsigned short", "unsigned long", "float",
        "double", "boolean", "char", "octet", "any", "TypeCode", "Principal",
        "Objektreferenz", "struct", "union", "enum", "string", "sequence", "array",
        "alias", "exception"
    };
}
```

Programmbeispiel 48: Interface-Repository Klient

Bei Verwendung des ORBs ORBacus ist für die Kompilation und einen Testlauf wie folgt vorzugehen:

1. Anlegen eines temporären Verzeichnisses, z. B. mkdir generated.
2. Übersetzen der IDL-Beschreibung durch den IDL-für-Java-Compiler
   jidl --output-dir generated Interface.idl
3. Übersetzen der Java-Klassen
   javac -d $CLASSES generated/ir/*.java
   javac -d $CLASSES IRClient.java InterfaceServer.java InterfaceImpl.java
   Die Klassen *InterfaceServer.java* und *InterfaceImpl.java* sind in diesem Beispiel nicht angegeben. Der ORBacus ORB verlangt jedoch eine Objektimplementierung und eine Registrierung des Objektes.
4. Starten des Interface-Repository-Servers *irserv*, der Bestandteil des CORBA-Produktes ist, durch
   irserv --ior Interface.idl > /tmp/Repository.ref
   Durch das Kommandozeilenargument *--ior* wird der Server angewiesen, seine Objektreferenz als String zur Standardausgabe zu schreiben. Die Umlenkung in eine Datei dient lediglich dazu, dem Klienten diese Objektreferenz als String zukommen zu lassen.
5. Starten des Servers für die Objektimplementierung (in diesem Beispiel nicht angegeben) durch
   java repository.InterfaceServer --ORBservice InterfaceRepository
   `cat /tmp/Repository.ref`
6. Starten des Klienten
   java repository.IRClient --ORBservice InterfaceRepository `cat /tmp/Repository.ref`

Sowohl der Klient, als auch die Objektimplementierung geben durch das Kommandozeilenargument *--ORBservice* bekannt, das sie mit dem Interface-Repository-Server arbeiten wollen. Unter UNIX kann durch die Ausführung von `*cat Datei*` die Objektreferenz für das Verbinden zu dem Prozess der Applikation zur Auswertung übergeben werden. Bei Verwendung anderer Betriebssysteme ist eine entsprechende Modifikation vorzunehmen.

Das Ergebnis des Testlaufes ist:

```
name              = AbgeleiteteSchnittstelle
id                = IDL:ir/AbgeleiteteSchnittstelle:9.4
definiert in      = IDL:ir:1.0
version           = 9.4
Objektreferenz
Operationen:
0: methode1
   Parameter:
      i(short)
      a(any)
1: methode2
   Parameter:
      l(long)
2: methode3
   Parameter:
      s(string)
      d(double)
3: methode4
   Parameter:
      s(string)
      d(double)
      x(short)
Attribute:
0: attribut1
1: attribut2
2: attribut3
BasisInterfaces:
0: IDL:ir/BasisInterface1:1.7
1: IDL:ir/BasisInterface2:5.9
2: IDL:ir/BasisInterface2:5.9
```

Beispiel 10: Ausgabe der Interface-Repository-Informationen

Diese Ausgabe entspricht bzgl. aller Informationen exakt der IDL-Beschreibung.

6.8 Die dynamische Schnittstelle für CORBA-Objekte

In dem o. a. Beispiel über den intelligenten Softwareagenten wurde bereits deutlich, dass eine verteilte Anwendung nicht ausschließlich auf statischen Aufrufen und Daten basiert. In genau den Fällen, in denen der vom IDL-Compiler generierte Stub nicht verfügbar ist, wäre ein dynamischer Zugang zu der CORBA-basierten Anwendung notwendig.

Weitere Gründe für den Zugang über die dynamische Schnittstelle können sein:

1. CORBA verfügt nur begrenzt über die Möglichkeit der asynchronen Methodenaufrufe. Im Zusammenhang mit einer IDL-Beschreibung, also einem statischen Zugang zu der Anwendung, ist nur die Möglichkeit der *oneway*-Operation zu sehen. Laut Spezifikation wird der Aufrufer einer *oneway*-Operation nicht unbedingt von der erfolgreichen Zustellung der Daten zum Server informiert. Im Fehlerfall ist ebenfalls keine Benachrichtigung des Klienten vorgesehen. Ebenso schwer ist es, nach einer erfolgreichen *oneway*-Operation die von der Serverseite erstellten Rückgabedaten zu bekommen.

Laut IDL-Spezifikation sind alle Parameter einer *oneway*-Operation lediglich *in*-Parameter, d. h. sie gelangen vom Klienten zum Server, aber nicht in die umgekehrte Richtung. Ebenso ist der Rückgabewert mit *void* anzugeben.

Will der Klient eine Antwort von der Serverseite, so sind sowohl der Klient als auch der Server derart zu implementieren, dass sie über den Eventdienst gehen. Der Klient registriert sein Interesse an einer Antwort und der Server muss als Produzent genau dieser Antwort ebenfalls beim Eventdienst registriert sein. Der Entwicklungsaufwand ist für diese Form der Asynchronität überproportional hoch.

Die dynamische Schnittstelle kann hier Abhilfe schaffen, indem zur Laufzeit ein geeignetes Objekt mit statischen Daten versehen wird und die Anfrage asynchron zum Server gelangt. Zu einem späteren Zeitpunkt kann der Klient die Daten abfragen.

2. Bei langsamen Netzwerken ist es erstrebenswert, die Netzlast und somit die Kommunikation zwischen Klient und Server zu optimieren. Die dynamische Schnittstelle erlaubt es, mehrere Anfragen zu einer zusammenzufassen.

An dieser Stelle wird das Fehlen von der so genannten *Message-Oriented-Middleware* in den CORBA-Spezifikationen 2.x besonders deutlich. Es existiert zurzeit keine geeignete Möglichkeit, Daten synchron oder asynchron in eine Warteschlange einzustellen und zu einem späteren Zeitpunkt zu verarbeiten. Genau diese Vorgehensweise ist für die Praxis absolut notwendig. Will bspw. eine Unternehmung neue Kunden in ihre Datenbank einstellen, so würde es sich anbieten, die Kundendaten asynchron in eine Warteschlange zu stellen und zu einem späteren Zeitpunkt, z. B. nachts, wenn die Rechner weniger belastet sind, einer Verarbeitung zuzuführen.

Entsprechend wird im CORBA-Umfeld zurzeit immer vorausgesetzt, dass ein Dienst oder das gewünschte Objekt vorhanden und aktiv sind. Durch den Einsatz standardisierter *Message-Oriented-Middleware* kann auch dieses Manko behoben werden.

Im Zusammenhang mit asynchronen Methodenaufrufen kann die dynamische Schnittstelle Abhilfe schaffen.

Um Operationen von CORBA-Objekten aufzurufen, benutzt ein Client i. A. das vom IDL-Compiler erzeugte Interface bzw. die entsprechenden generierten Clientstubs, die jedoch eine statische Schnittstelle sind. In Bezug auf den aufzuschreibenden Programmcode erfolgt in diesem Fall der Methodenaufruf so, als ob es sich um ein lokal instanziiertes Objekt handelt. Unabänderlich, also statisch, vorgegeben ist jedoch, welches Interface zu dem Aufruf gehört. Diese Form des Methodenaufrufs bietet eine Reihe von Vorteilen, wie z. B. die o. a. Transparenz zwischen einem lokalen und entfernten Objekt, Typsicherheit bzgl. der zu übergebenden Argumenten und eine einfache, bequeme Handhabbarkeit der Schnittstelle.

U. U. kann dieser Ansatz nicht ausreichend sein. Neue Objekte werden im Netzwerk zur Verfügung gestellt, Rückgabedaten in Form von Datenstrukturen verändern sich und können vom Aufrufer nicht mehr als statisch angenommen werden. Dies beinhaltet auch die Darstellung von Daten in einer (voll-) graphischen Benutzeroberfläche, die sich dem dynamischen Verhalten des Systems automatisch anpassen muss etc.. Für derartige Problemfälle, in denen die Schnittstelle zur Zeit der Programmentwicklung noch nicht als IDL spezifiziert werden kann, existiert unter CORBA 2.x eine dynamische Schnittstelle, über die erst zur Laufzeit der Anwendung festgelegt wird,

a. welches Objekt angesprochen werden soll.
b. welche Operation(en) ausgeführt werden soll(en).
c. welche Argumente die Operation hat, also Anzahl der Parameter, Typ, Modus und Wert.

6.9 Das Dynamic Invocation Interface (DII)

In einigen Fällen, z. B. wenn der Server neue Objekte zur Verfügung stellt und die zugehörige IDL-Beschreibung dem Client nicht bekannt ist, muss ein Client ohne Kenntnis über den Objektaufbau mit entfernten Objekten arbeiten. Bekannt ist ihm nur ein Objektname oder eine *stringified IOR* des entfernten Objektes. Ist der Objektname bekannt, so kann die zugehörige Objektreferenz bspw. über den Namensdienst besorgt werden; das Interface und weitere Informationen können über das Interface Repository erfragt werden (vgl. Abbildung 19). Im Fall einer *stringified IOR* kann eine Konvertierung zu einer Objektreferenz vom Typ CORBA::Object erfolgen, mit der gearbeitet wird.

Bei der Verwendung des DII können mit Hilfe von **R**equest-**P**seudo-**O**bjekten (RPO) in folgenden Schritten beliebige Operationen aufgerufen werden (vgl. Abbildung 19):

1. Beschaffung der Objektreferenz für das aufzurufende Objekt.
2. Beschaffen der Informationen über das Interface aus dem IR.
3. Erzeugen eines RPO, wobei wenigstens das aufzurufende Objekt, der Name der auszuführenden Methode und die Liste der Argumente anzugeben sind.
4. Übergabe des RPO an den ORB. Hiermit wird der Start der Operation eingeleitet.
5. Warten auf das Ergebnis der Operation (optional).
6. Vernichten oder wieder verwenden des RPOs.

Für den Server ist es völlig transparent, ob der Client den Methodenaufruf via Clientstub oder DII ausgelöst hat; beide Vorgehensweisen führen zu der gleichen Repräsentation des Aufrufs im ORB.

Durch die Möglichkeit der Trennung von Start des Objektes und Warten auf eine Antwort ist DII zusätzlich für die Anwendungen von Interesse, die zwischenzeitlich andere Arbeiten ausführen wollen. Im Gegensatz zum Methodenaufruf via Clientstub ist bei Verwendung von DII auch ein (verzögerter) asynchroner Methodenaufruf möglich.

Für die Parameterübergabe verwendet DII den Datentyp NamedValue bzw. eine Sequenz dieses Typs.

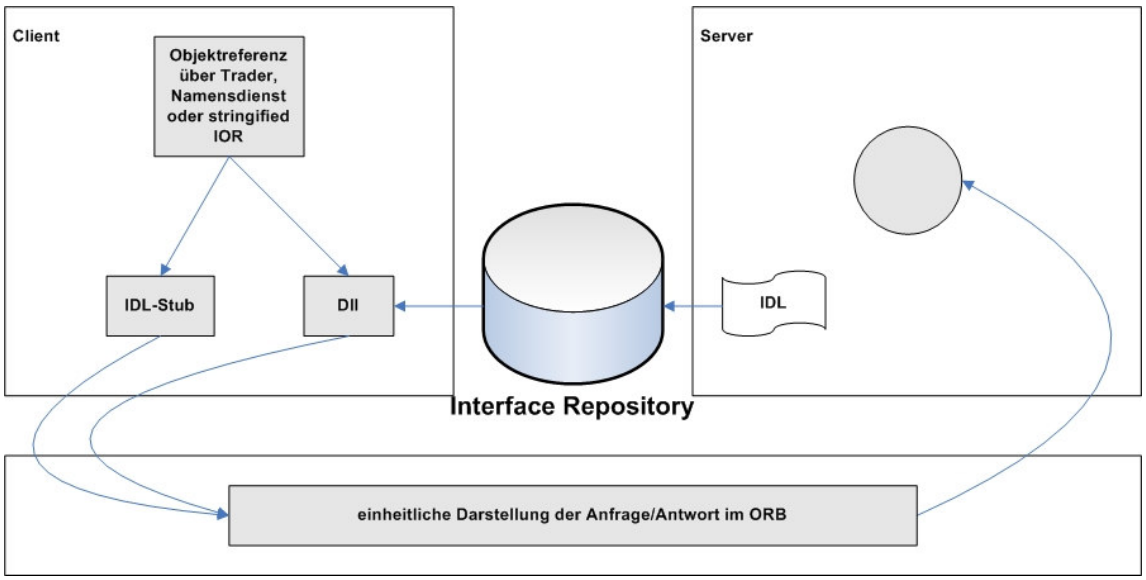


Abbildung 19: Statischer und dynamischer Methodenaufruf

In Abbildung 20 ist der statische und dynamische Zugang eines Klienten zum System dargestellt. Es ist wichtig, festzuhalten, dass der Server nicht zwischen statischem und dynamischem Zugang unterscheiden kann.

Im Falle des statischen Zugangs erhält der Klient von einem Namensdienst, einem Trader oder in Form einer Objektreferenz, Zugang zu dem System. Alle Informationen über das entfernte Objekt sind in seinem Stub vorhanden.

Bei einem dynamischen Zugang sind zwei Fälle zu unterscheiden:
1. Die einzige Information eines Klienten besteht in einer Objektreferenz. In diesem Fall ist nur der Zwischenweg über das Interface Repository möglich.

   Der Klient baut basierend auf den Informationen des Interface Repositories ein Request-Pseudo-Objekt auf, füllt es mit den entsprechenden Daten und ruft synchron oder asynchron das entfernte Objekt an.
2. Der Klient verfügt über eine IDL-Beschreibung und möchte die dynamische Schnittstelle für einen asynchronen Aufruf nutzen. In diesem Fall muss der Klient ebenfalls ein Request-Pseudo-Objekt aufbauen und basierend auf der Methodensignatur die Parameterliste über allgemeingültige Datentypen wie z. B. CORBA::Any aufbauen.

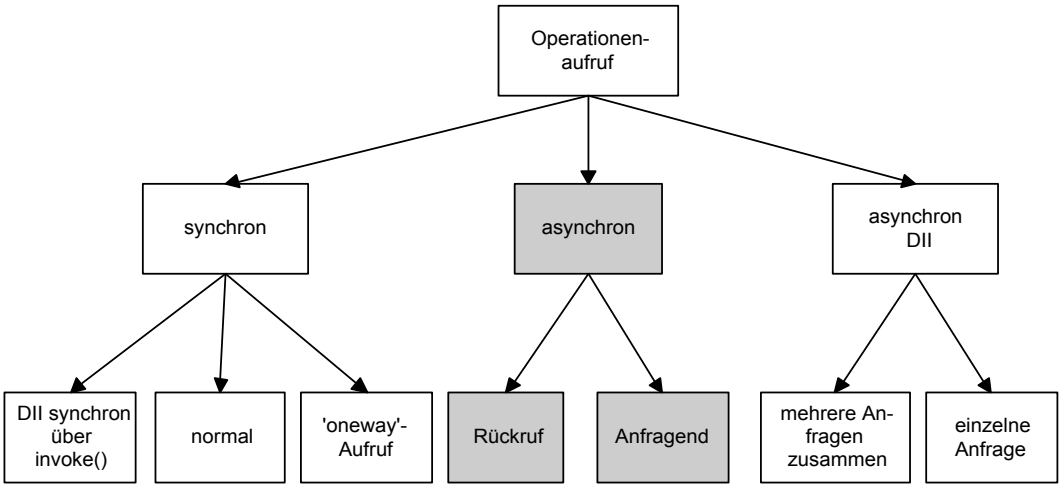


Abbildung 20: Die Kommunikationsmöglichkeiten von CORBA

Beide Vorgehensweisen werden im Folgenden im Detail erörtert. Um in beiden Fällen die Methodensignatur nachzubilden, bedarf es einer Datenstruktur, die es erlaubt in der richtigen Reihenfolge jedes Datum präzise zu repräsentieren.

6.9.1 Ein Beispiel für den Umgang mit DII

Im Folgenden wird eine einfache Applikation für die Verwendung der dynamischen Schnittstelle vorgestellt. Primär von Interesse ist hier der Aufbau von Request-Pseudo-Objekten und der eigentliche Methodenaufruf sowohl synchron als auch asynchron.

Es wird vorausgesetzt, dass auf der Serverseite ein Objekt implementiert ist, das einen übergebenen String in Groß- bzw. Kleinbuchstaben konvertiert und als Rückgabewert dem Aufrufer zurückgibt.

Als Grundlage dient die folgende IDL-Beschreibung:

```
// StringTest.idl

#ifndef DIITEST_IDL
#define DIITEST_IDL

module diitest {
    interface StringTest {
        string toLower(in string s);
        string toUpper(in string s);
    };
};
#endif
```

Programmbeispiel 49: IDL-Beschreibung für einen Zugang über DII

Die Implementierung des Klienten:

```
// DIIStringTestClient.java

package diitest;
import org.omg.CORBA.*;
import java.io.*;
import java.util.*;
import utilities.Init;

public class DIIStringTestClient {
    public static void main(String args[])
    {
        // Initialisierung des Klienten

        try {
               init.initClient(args);
        }
        catch (RuntimeException e) {
               System.err.println(e.getMessage());
               System.exit(-1);
        }
        ORB orb = init.getORB();


        // Lesen der Objektreferenz als String (wurde vom Server in die angegebene
        // Datei geschrieben) und konvertieren in eine Objektreferenz

        try {
               String ref = null;
               try {
                      ref = init.readIOR("/tmp/StringServer.ref");
               }
               catch(IOException ex) {
                      System.err.println("Fehler beim Lesen der Objektreferenz");
                      System.exit(-2);
               }

               org.omg.CORBA.Object obj = orb.string_to_object(ref);
               if (obj == null) {
                      System.err.println("obj == null");
                      System.exit(-3);
               }


               // Aufbau der Request-Pseudo-Objekte, eins für den Aufruf
               // von toUpper, das andere für die Methode toLower

               org.omg.CORBA.Request request1 = obj._request("toLower");
               org.omg.CORBA.Request request2 = obj._request("toUpper");

               String mixed = "Ein beliebiger String mit Klein- und Grossbuchstaben";


               // laut IDL-Vereinbarung sind die Parameter beider Methoden
               // in-Parameter. Der Rückgabewert ist in beiden Fällen 'string'

               request1.add_in_arg().insert_string(mixed);
```

```
                request1.set_return_type(orb.get_primitive_tc(TCKind.tk_string));
                request2.add_in_arg().insert_string(mixed);
                request2.set_return_type(orb.get_primitive_tc(TCKind.tk_string));

                // synchroner Aufruf von toLower und toUpper und Rückgabewerte extrahieren

                request1.invoke();
                request2.invoke();
                String r1 = request1.return_value().extract_string();
                String r2 = request2.return_value().extract_string();
        }
        catch(Exception ex) {
                System.err.println(ex.getMessage());
                System.exit(1);
        }
        System.exit(0);
    }
}
```

Programmbeispiel 50: DII-Klient

Für die Verwendung asynchroner Methodenaufrufe ist
    request1.invoke();
zu ersetzen durch
    request1.send_deferred();
Zu einem späteren Zeitpunkt kann der Klient durch
    boolean b = request1.poll_response();
nachfragen, ob die Bearbeitung des Servers abgeschlossen ist und ein Ergebnis vorliegt (true) oder nicht (false).

Die dynamische Schnittstelle erlaubt auch die Gruppierung von mehreren Anfragen zu einer. In diesem Fall ist wie folgt vorzugehen:

```
// Aufbau der Request-Pseudo-Objekte wie oben

Request [ ] requestSeq = new Request[2];
requestSeq[0] = request1;
requestSeq[1] = request2;
orb.send_multiple_requests_deferred(requestSeq);
```

Programmbeispiel 51: Mehrere Objekte asynchron versenden

Hierbei bezeichnet *orb* das Objekt, das bei der obligatorischen Initialisierung eines Klienten- oder Server-ORBs an den Aufrufer zurückgegeben wird.

6.9.2 Die Datentypen NamedValue und NVList

Die beiden Datentypen NamedValue und NVList sind Pseudoobjekte, die genau genommen rein lokal in einem Client/Server agierende C++- oder Java-Objekte darstellen, deren Interface in IDL beschrieben wurde. Diese beiden Objekte werden beim Aufruf des DII für die Bildung von Argumentenlisten benötigt.

Im Grunde sind Pseudoobjekte nichts anderes als Datentypen, die jedoch von derart hoher Bedeutung für CORBA sind, dass sie in jedem System und in jeder (unterstützten) Programmiersprache vorhanden und immer in derselben Weise benutzbar sein müssen.

Objektinstanzen des Typs *NamedValue* treten nur als Elemente des Datentyps *NVList* auf. *NVList* wird bei einem Request für die Darstellung der Argumentenliste eingesetzt. Ein Objekt vom Typ *NamedValue* enthält (optional) einen Namen, einen Wert vom Typ *Any* und ein Flag, das angibt, in welche Richtung der Parameter gehen (s. u.).

```
typedef string Identifier;

interface NamedValue {
    readonly attribute Identifier    name;
    readonly attribute any           value;
    readonly attribute Flags         flags; // ∈ { ARG_IN, ARG_OUT, ARG_INOUT }
};
```

Programmbeispiel 52: Pseudo-IDL Beschreibung

Für das Flag sind lediglich die Werte *ARG_IN, ARG_OUT* und *ARG_INOUT* zulässig. Hiermit wird die Übertragungsrichtung der Argumente beim Methodenaufruf festgelegt (vgl. in, out, inout der IDL-Spezifikation).

Die Methoden *name* und *value* liefern lediglich Beschreibungen auf Bestandteile des eigenen *NamedValue*-Objektes zurück.

Der Datentyp *NVList* ist eine Liste von *NamedValue*-Objekten; diese Elemente werden über die folgenden Methoden (hier C++) erzeugt:

1. *add(in Flags flag);*. Erzeugt ein *NamedValue*-Objekt ohne Namen und Wert. Lediglich flag wird gesetzt.
2. *add_item(in Identifier name, in Flags flag);*. Erzeugt ein *NamedValue*-Objekt und initialisiert den Namen und das Flag. Die Zuweisung eines Wertes kann später erfolgen durch *(nv->value()) <<= value;
3. *add_value(in Identifier name, in any value, in Flags flag);*. Erzeugt ein *NamedValue*-Objekt und initialisiert Namen, Wert und Flag.
4. *add_item_consume(in Identifier name, in Flags flag);*. Wie *add_item*, jedoch wird der vom Identifier belegte Speicher von dem erzeugten *NamedValue*-Objekt verwaltet; d. h. der Entwickler kann diesen Wert weder ändern, noch freigeben.
5. *add_value_consume(in Identifier name, in any value, in Flags flag);*. Analog zu oben.

Für Java existiert die folgende IDL-Beschreibung, die von dem jeweiligen Hersteller zu implementieren ist:

```
exception Bounds {
    ...
};

typedef unsigned long Flags;         // ∈ { ARG_IN, ARG_OUT, ARG_INOUT }
typedef unsigned long Status;

interface NVList {
    readonly attribute unsigned long count;
    NamedValue add_item(in Identifier name, in any value, in Flags flag);
    NamedValue add_value(in Identifier name, in any value, in Flags flag);
    NamedValue item(in unsigned long index) raises(Bounds);
    Status     remove(in unsigned long index) raises(Bounds);
};
```

Programmbeispiel 53: IDL-Beschreibung der NVList

Der Status gibt Auskunft über Erfolg/Misserfolg einer Operation, wobei jedoch nicht festgelegt ist, welche Werte zu diesem Typ gehören, d. h. jede CORBA-Plattform kann einen Status nach eigenem Ermessen definieren. So ist unter Iona's Orbix CORBA::Status als CORBA::Boolean definiert.

Das Einstellen von Daten in ein any-Objekt ist sprachabhängig. In der o. a. Beschreibung wurde die Möglichkeit von C++, nämlich das Überladen von Operatoren, genutzt. Für Java

existieren hierfür Methoden für die skalaren Datentypen, bzw. generierte Helperklassen für die benutzerdefinierten Datentypen.

6.9.3 Das Erzeugen von Requests

Ein Request bildet für die Benutzung des DII die Grundlage. Ein Request wird jeweils für ein spezielles Objekt erzeugt und kann beliebig oft verwendet werden.

```
// Die entsprechenden Datentypen wie z. B. ContextList etc. werden als gegeben angenommen

interface Request {
    readonly attribute Object        target;
    readonly attribute Identifier    operation;
    readonly attribute NVList        arguments;
    readonly attribute NamedValue    result;
    readonly attribute Environment   env;
    readonly attribute ExceptionList exceptions;
    readonly attribute ContextList   contexts;
    attribute context                cxt;
    Status     invoke();
    Status     send_oneway();
    Status     send_deferred();
    Status     get_response();
    boolean    poll_response();
};
```

Programmbeispiel 54: IDL-Beschreibung der Schnittstelle Request

Die Methode *_request()*, die von jedem CORBA-Objekt zu unterstützen ist, hat genau ein Argument, nämlich den Methodennamen des durch *obj* referenzierten Objekts als String.

Ein mit *_request(string methodName)* erzeugter Request enthält automatisch eine Argumentenliste vom Typ *NVList*, die anfangs leer ist und nachträglich durch die entsprechenden add-*Methoden* von *NVList* erweitert werden kann. Ein Zugriff auf die Argumentenliste ist über das Attribut *arguments* möglich (s. o.), wobei die Elemente der Liste über ihre Position in der Liste identifiziert werden; der Name bleibt unberücksichtigt.

Durch die Methode *Object::_request()* werden automatisch eine Reihe von Standardeinstellungen getätigt, die der ORB für die Bearbeitung eines Requests benötigt. Wird über die Werte dieser Standardeinstellung Kontrolle benötigt, so muss ein Request, die von jedem CORBA-Objekt bereitgestellte Methode *Object::_create_request()* erzeugt werden.

Bei Aufruf der Methode *_create_request()* müssen alle Angaben bereits in ihrer endgültigen Form vorliegen; für die Argumentenliste heißt das, dass die Allokierung und Initialisierung bereits im Vorfeld stattgefunden haben muss.

In einem Request sind die folgenden Informationen enthalten:
1. Die Objektreferenz für das aufzurufende Objekt.
2. Der Name der aufzurufenden Methode als String.
3. Die Argumentenliste als NVList.
4. Speicher für die Aufnahme des Ergebniswertes.
5. Speicher für die Bereitstellung von Informationen über mögliche Ausnahmen, die bei der Bearbeitung des Requests auftreten können (Environment)
6. Ein Context-Parameter.
7. Eine Liste für die Bereitstellung anwenderdefinierter Ausnahmen, die während der Ausführung der Methode auftreten können.
8. Eine Menge von Context-Einträgen, die der Objektimplementierung zusätzlich zu den Argumenten der Methode zur Verfügung gestellt werden.

Die o. a. Werte können beim Erzeugen eines Requests spezifiziert werden. Wird für einen dieser Werte NULL angegeben, so verwendet das System einen Standardwert.

Bei dem Typ *Context* handelt es sich um ein Pseudoobjekt, das Namen mit Zeichenketten assoziiert, also ähnlich zu den Umgebungsvariablen unter UNIX oder NT ist. Diese Angaben können von einem Client verwendet werden, um der aufzurufenden Methode zusätzlich zu den Argumenten weitere Informationen zur Verfügung zu stellen.

Das letzte Argument *flags* in *_create_request* kann die Werte *OUT_LIST_MEMORY* und *IN_COPY_VALUE* annehmen. *flags* ist dazu gedacht, die Hauptspeicherverwaltung für die *in*-, *out*- und *inout*-Argumente zu regeln.

Bei Verwendung von C++ ist dies jedoch überflüssig, da die Verwaltung der Argumente durch den Typ *CORBA::Any* erfolgt. Aus diesem Grund wird *flags* bei Verwendung von C++ ignoriert.

Bei Verwendung der ersten Variante von *_create_request* ist der ORB gezwungen, sich die benötigten Informationen über anwenderdefinierte Ausnahmen zur Laufzeit aus dem IR zu beschaffen; bei der zweiten Variante entfällt diese potentiell aufwendige Anfrage an das IR.

6.9.4 Synchrone Ausführung eines Requests

Die einfachste Art, einen Request auszuführen, besteht in dem Aufruf der Methode *Request::invoke()*. Diese Methode verhält sich ebenso wie der Aufruf einer Stubmethode, d. h. der Client wartet auf das Ende der Operation (synchroner Aufruf). Die Ergebniswerte und eventuellen Ausnahmen können anschließend abgefragt werden.

6.9.5 Asynchrone Ausführung eines Requests

Im Gegensatz zu der synchronen Ausführung kann mit der Methode *Request::send_deferred* der Request dem ORB übergeben werden, und die Methode kehrt sofort zum Aufrufer zurück. Ob die Bearbeitung der Anfrage abgeschlossen ist und die Ergebnisse bereitstehen, kann zu einem späteren Zeitpunkt über die Methode *Request::poll_response* abgefragt werden. *poll_response* blockiert nicht und kehrt sofort mit dem aktuellen Zustand zurück; hierbei gilt i. A. der Statuswert 1 als *Request beendet*, 0 sonst. Wird die Methode *Request::get_response* aufgerufen, so findet eine Blockierung statt, bis der Request abgeschlossen ist.

Die Methoden *send_deferred*, *poll_response* und *get_response* beziehen sich jeweils auf genau einen Request, d. h. es ist möglich, mittels *send_deferred* mehrere Requests hintereinander abzuschicken und gezielt für einen das Ergebnis abzuwarten.

Sinnvoll kann es sein, mehrere Methoden nebeneinander zu starten und die Ergebnisse in der Reihenfolge ihres Eintreffens zu bearbeiten. Prinzipiell kann dies mit den o. a. Methoden geschehen; eleganter ist jedoch die Verwendung der Methoden

```
Status get_next_response(RequestHolder r);
boolean poll_next_response();
```

get_next_response wartet, bis ein beliebiger, durch *send_deferred* gestarteter Request abgeschlossen ist. Das Ergebnis wird in der Der Holderklasse RequestHolder zurückgeliefert. Die Methode *poll_next_response* kehrt sofort zurück und liefert TRUE, genau dann, wenn ein nachfolgender Aufruf von *get_next_response* sofort zurückkehren würde, also wenn für einen noch ausstehenden Request bereits die Ergebniswerte oder eine Exception eingetroffen

sind. Steht von vornherein fest, dass der Client nicht auf das Ende der Methode wartet, kann eine *oneway*-Operation (*send_oneway*) gestartet werden.

Der ORB bietet zusätzlich Methoden an, die es erlauben, mehrere Requests mit einem einzigen Aufruf parallel zu starten:

```
class ORB {
    Status     send_multiple_requests_oneway(RequestHolderSeq  reqList);
    Status     send_multiple_requests_deferred(RequestHolderSeq reqList);
        ...
};
```

Programmbeispiel 55: ORB-Schnittstelle für die Gruppierung mehrere Objekte

Hierbei ist *RequestSeq* definiert als

```
typedef sequence<Request> RequestSeq;
```

6.10 Probleme der dynamischen Schnittstelle

Das Problem mit DII ist genau in der Parameterübergabe zu sehen! Für jeden Parameter der Operation existiert genau ein NamedValue in der NVList. Ein Any-Objekt ist jedoch so konstruiert, dass es lediglich bekannte, d. h. im Vorfeld in IDL beschriebene, Datentypen aufnehmen kann. Anderenfalls würden die überlagerten Operatoren in C++ bzw. die Helperklassen in Java nicht vorhanden sein (eine Ausnahme ist der ORB MICO, der fast ausschließlich mit DII arbeitet und eine (proprietäre) Erweiterung des any-Containers enthält, mit der auch unbekannte Datentypen, die zur Laufzeit ermittelt werden, eingestellt werden können). Der Klient hat lediglich eine Beschreibung der Parameterdaten aus dem Interface Repository und somit keine Möglichkeit konkrete Datentypen zu erzeugen und in das Any-Objekt einzustellen.

Die Folgerung ist, dass lediglich bekannte Daten (boolean, char, long etc.) unter Verwendung von DII zum Server transportiert werden können. Eine Lösung des Problems ist die Verwendung von DynAny und ein anschließendes Einstellen der konstruierten Daten in ein Any-Objekt. Diese Vorgehensweise ist durchaus legitim, wird jedoch nicht von jedem CORBA-Produkt unterstützt, da das lokale Objekt DynAny nicht überall implementiert ist.

6.11 Betrachtungen zur dynamischen Schnittstelle

U. a. wird in [Orfali et al., 1998] ein kurzes Szenario über den Umgang mit der dynamischen Schnittstelle angegeben. Als Schlussfolgerung ist nachzulesen:

> „Das CORBA-DII lässt Sie einen entfernten Aufruf on-the-fly erstellen. Er führt einen Stil der dynamischen Programmierung ein, der die Laufzeitentdeckung von Objekten und just-in-time-Bindungen unterstützt. Zusätzlich entfällt die Bürde, die Stubs zu verwalten und sie auf die Clients zu verteilen. Es müssen keine Versionen verwaltet werden; Sie erstellen eine Bindung immer mit der aktuellsten Schnittstelle, die ein Objekt zur Verfügung stellt. Allerdings kaufen Sie diese Freiheit zu einem hohen Preis. Zunächst erfordert es mehr Programmieraufwand. Zudem sind dynamische Aufrufe etwa 40-mal langsamer als statische. Das bedeutet, Sie tauschen Leistung und Bequemlichkeit gegen maximale Flexibilität.“

Dazu ist grundsätzlich zu bemerken, dass Orfali et al. den Beweis über die Verarbeitungsgeschwindigkeit schuldig geblieben sind. In [Orfali et al., 1998] wurde lediglich ein Versuch mit

einer parameterlosen Operation durchgeführt, der nicht als repräsentativ angesehen werden kann.

Aufgrund des engen Zusammenhangs der dynamischen Schnittstelle mit den dynamischen Objekten vom Typ Any und DynAny, sowie dem Interface Repository, werden die Untersuchungen auch auf diese dynamischen CORBA-Komponenten ausgedehnt.

Die dynamischen CORBA-Komponenten DII, Interface-Repository und DynAny wurden bisher nicht im Detail untersucht. Neben der vorliegenden Dissertation sind erst im Jahr 2002/2003 erste Artikel von Douglas Schmidt und Steve Vinoski [Schmidt et al., 2002] zu den Themen DII, Interafce Repository und DynAny erschienen, wobei festzuhalten ist, dass die Themen hier recht oberflächlich behandelt werden.

7 Konsequenzen: Grundregeln der Kommunikation

Es wird in einigen Projekten über die schlechte Verarbeitungsgeschwindigkeit von CORBA-basierten verteilten Anwendungen berichtet. Ein grundlegendes Problem ist hierbei, dass Entwickler durch die Transparenzeigenschaften von CORBA zu einer Entwicklungsform veranlasst werden, die in einem rein lokalen Umfeld angemessen, in einem verteiltem Umfeld Auswirkungen auf die Verarbeitungsgeschwindigkeit hat. Dies wird nachfolgend im Detail erläutert.

Ein weiterer wesentlicher Grund ist die völlige Transparenz bzgl. der zugrunde liegenden und von einem IDL-Compiler generierten Netzsoftware.

Aus Sicht der Verteilungsplattform sind die folgenden Gründe oft ausschlaggebend für die Herabsetzung der Verarbeitungsgeschwindigkeit in einem verteilten Umfeld:

1. Die Anzahl der entfernten Methodenaufrufe. Dies beinhaltet die Verwendung des Schlüsselwortes *attribute* bzw. *readonly attribute* in der Deklaration einer IDL-Schnittstelle, ebenso wie benutzerdefinierte Operationen, die lediglich ein einzelnes Datum versenden. Neben der Latenzzeit, die das Netzwerk für die Übertragung der Daten einnimmt, ist auf der Serverseite der Aufwand zu berücksichtigen, der benötigt wird um entsprechend viele Objekte zu erzeugen und jeweils zu aktivieren.
2. Die Datenmenge, die mit jedem Aufruf an das entfernte Objekt geschickt wird. Von Interesse ist hier die geeignete Datenrepräsentation.
3. Der Aufwand für das eventuelle Verpacken und Entpacken der zu übertragenden Daten, das in Abhängigkeit der definierten IDL-Datentypen in einem hohen Maße aufwendig sein kann.

Jeder entfernte Methodenaufruf bedingt eine Reihe von Aktivitäten sowohl auf der Seite des Klienten, als auch auf der Seite des Servers. Zum einen ist hier die Latenzzeit des Netzwerkes zu benennen, zum anderen ist auf beiden Seiten der Aufwand zu berücksichtigen, der notwendig ist, die Daten ggf. in ein neutrales Format zu verpacken und wieder auszupacken.

Verwendet ein Entwickler innerhalb einer IDL-Schnittstellendeklaration häufig das Schlüsselwort *attribute* bzw. *readonly attribute*, so werden entsprechende Zugriffsmethoden von dem IDL-Compiler generiert. Diese Methoden stellen genau ein Datum ein bzw. lesen genau ein Datum aus. Der hiermit verbundene eventuelle Aufwand des Marshallings und Unmarshalllings in Zusammenhang mit dem allgemeingültigen Protokoll IIOP ist i. d. R. nicht vertretbar.

```
struct Person {
    string name;
    string vorname;
};

interface PersonenSuche {
    attribute name;          // die Zugriffsmethoden string name() und void name(in string s)
                             // werden generiert
    attribute  vorname;      // analog zu oben
    Person     getPerson(in string s);
    void       setPerson(in Person p);
};
```

Verwendet der Entwickler die generierten Methoden
    string name() bzw. string vorname();
und
    void name(in string s) bzw. void vorname(in string s),

so werden in Abhängigkeit zu den übertragenen Daten relativ große Pakete generiert. Verwendet der Entwickler jedoch die Methoden
    Person getPerson(in string s)
bzw.

```
void setPerson(in Person p),
```

so werden in beiden Fällen vollständige Personenstrukturen übertragen bzw. empfangen. Die Größe der gesendeten bzw. empfangenen Pakete sind nur marginal größer als bei einem korrespondierenden attribute-Aufruf. Die Anzahl der Aufrufe wurde jedoch in diesem Beispiel halbiert.

Programmbeispiel 56: Attributmethoden

In den folgenden Kapiteln wird dieser Sachverhalt genauer untersucht. Zusätzlich bietet sich an dieser Stelle ein erster Vergleich mit den dynamischen CORBA-Objekten Any und Dyn-Any an.

8 Benötigte Zeit zum Datentransport

Im Folgenden wird untersucht, wie viel zeitlicher Aufwand notwendig ist, um ein Datum im statischen und dynamischen Fall an das Serverobjekt zu versenden. Gemessen werden die skalaren Datentypen und zusammengesetzten Datentypen. Die Skalare umfassen die Datentypen *boolean*, *char*, *short*, *long*, *float*, *double* und *string*.

Für die zusammengesetzten Datentypen werden die folgenden IDL-Definitionen verwendet:

1. Eine Basisstruktur bestehend aus den o. a. skalaren Datentypen:

```
module basics {
    struct BasicStruct {
           boolean        aBoolean;
           short          aShort;
           char           aChar;
           octet          anOctet;
           long           aLong;
           float          aFloat;
           double         aDouble;
           string         aString;
    };
};
```

Programmbeispiel 57: Basisstruktur in IDL

2. Eine Personenstruktur mit zusätzlichen Angaben zu einer oder mehreren Adressen, einem Geburtsort, einem Geburtsdatum und einer Folge von Wörtern, in denen die personenspezifischen Befähigungen abgelegt werden.

```
module personen {
    typedef sequence<string> BefaehigungenSeq;

    struct Adresse {
           string str;
           string plz;
           string ort;
    };
    typedef sequence <Adresse> AdressenSeq;

    struct Datum {
           short tag;
           short monat;
           long  jahr;
    };

    struct Person {
           string           name;
           string           vorname;
           AdressenSeq      adressen;
           Datum            geburtstag;
           Adresse          geburtsort;
           BefaehigungenSeq befaehigungen;
    };
    typedef sequence<Person> PersonenSeq;
};
```

Programmbeispiel 58: IDL-Strukturen für die Tests

Die jeweiligen Messungen wurden immer auf der gleichen Maschine durchgeführt und es war sichergestellt, dass neben den Systemprozessen keine weiteren Anwendungsprozesse liefen. Jeder Testlauf betrug 10 Sekunden und wurde mehrfach durchgeführt. Für die jeweils gezählten synchronen Methodenaufrufe wurde das arithmetische Mittel genommen.

In der IDL-Schnittstellenbeschreibung ist immer die Richtung für das Datum anzugeben. Aus diesem Grund wurde der Test immer mit in-, inout- und out-Parametern durchgeführt.

Für den statischen Fall werden die Schnittstellen

```
#ifndef NOANYTEST_IDL
#define NOANYTEST_IDL

#include "BasicTypes.idl"
#include "Person.idl"

module noany {
    interface NoAnyTestWithInParam {
        boolean               booleanOp(in boolean b);
        short                 shortOp(in short s);
        octet                 octetOp(in octet o);
        char                  charOp(in char c);
        long                  longOp(in long l);
        float                 floatOp(in float f);
        double                doubleOp(in double d);
        string                stringOp(in string s);
        basics::BasicStruct   basicStructOp(in basics::BasicStruct b);
        personen::Person      personenOp(in personen::Person p);
    };

    interface NoAnyTestWithOutParam {
        void   booleanOp(in boolean b, out boolean x);
        void   shortOp(in short s, out short t);
        void   octetOp(in octet o, out octet t);
        void   charOp(in char c, out char x);
        void   longOp(in long l, out long x);
        void   floatOp(in float f, out float x);
        void   doubleOp(in double d, out double x);
        void   stringOp(in string s, out string t);
        void   basicStructOp(in basics::BasicStruct b, out basics::BasicStruct x);
        void   personenOp(in  personen::Person p, out personen::Person pers);
    };

    interface NoAnyTestWithInOutParam {
        void   booleanOp(inout boolean b);
        void   shortOp(inout short s);
        void   octetOp(inout octet o);
        void   charOp(inout char c);
        void   longOp(inout long l);
        void   floatOp(inout float f);
        void   doubleOp(inout double d);
        void   stringOp(inout string s);
        void   basicStructOp(inout basics::BasicStruct b);
        void   personenOp(inout personen::Person p);
    };
};
#endif
```

Programmbeispiel 59: IDL-Beschreibung für den statischen Test

verwendet.

Für die Verwendung von Any-Objekten ist die Schnittstellenbeschreibung:

```
#ifndef ANYTEST_IDL
#define ANYTEST_IDL

#include "BasicTypes.idl"
#include "Person.idl"

module anytest {
    interface AnyTestWithInParam {
        any anyOp(in any a);
    };

    interface AnyTestWithOutParam {
        void anyOp(in any a, out any x);
```

```
    };

    interface AnyTestWithInOutParam {
        void anyOp(inout any a);
    };
};
#endif
```

Programmbeispiel 60: Schnittstelle für den any-Test

8.1 Statische Datentypen

In den folgenden Unterkapiteln wird der benötigte Zeitaufwand der Methodenaufrufe bei statischen Datentypen untersucht. Es werden jeweils die Daten als in-, inout- und out-Parameter übergeben.

Getestet wurde auf einem Pentium III Prozessor mit 700 MHz Taktfrequenz. Die Testmaschine läuft unter dem Betriebssystem SUSE LINUX 7.2 und ist mit 128 MB RAM ausgestattet. Verwendet wurde der Java Object Request Broker ORBacus for Java mit dem JDK 1.3. Getestet wurde auf genau einer Maschine. Da die einzelnen Objekte sich jedoch in verschiedenen JVMs befanden, musste die Kommunikation, wie im entfernten Fall, auf Netzwerkebene erfolgen.

8.1.1 Statische Datentypen bei in-Parametern

In der folgenden Grafik ist die Anzahl der Aufrufe bei Verwendung von in-Parametern und der Antwort als Rückgabewert angegeben. Die Laufzeit des Klienten betrug jeweils 10 Sekunden. Die Länge des Strings betrug 32 Zeichen. Gemessen wurde auf einer physikalischen Maschine. Der Klient und der Server befanden sich jedoch in verschiedenen Java Virtual Machines, so dass die Kommunikation vergleichbar ist mit der Kommunikation über verschiedenen Maschinen hinweg.

Verwendet wurde für die Messung die folgende IDL:

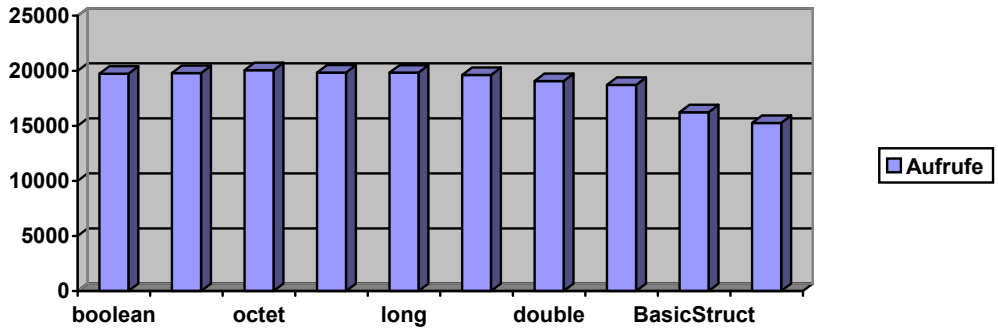
```
interface NoAnyTestWithInParam {
    boolean                 booleanOp(in boolean b);
    short                   shortOp(in short s);
    octet                   octetOp(in octet o);
    char                    charOp(in char c);
    long                    longOp(in long l);
    float                   floatOp(in float f);
    double                  doubleOp(in double d);
    string                  stringOp(in string s);
    basics::BasicStruct     basicStructOp(in basics::BasicStruct b);
    personen::Person        personenOp(in personen::Person p);
};
```

Programmbeispiel 61: Statischer Test mit in-Parametern

Die einzelnen Methoden wurden sukzessive während der Laufzeit aufgerufen. Das Ergebnis ist in der folgenden Tabelle und der hierzu korrespondierenden Grafik zusammengefasst.

Datentyp	Aufrufe in 10 Sekunden
boolean	19722
char	19791
octet	20051
short	19834
long	19825

float	19609
double	19062
string	18715
BasicStruct	16226
Person	15262



8.1.2 Statische Datentypen bei inout-Parametern

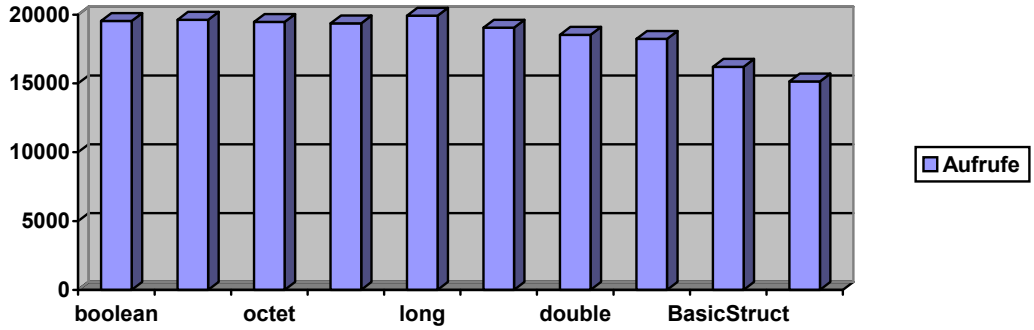
Analog wurde der Test mit inout-Parametern durchgeführt. Verwendet wurde die IDL

```
interface NoAnyTestWithInOutParam {
        void    booleanOp(inout boolean b);
        void    shortOp(inout short s);
        void    octetOp(inout octet o);
        void    charOp(inout char c);
        void    longOp(inout long l);
        void    floatOp(inout float f);
        void    doubleOp(inout double d);
        void    stringOp(inout string s);
        void    basicStructOp(inout basics::BasicStruct b);
        void    personenOp(inout personen::Person p);
};
```

Programmbeispiel 62: Statischer Test mit inout-Parametern

Als Ergebnis wurde erzielt:

Datentyp	Aufrufe in 10 Sekunden
boolean	19532
char	19628
octet	19469
short	19360
long	19916
float	19049
double	18530
string	18231
BasicStruct	16202
Person	15138



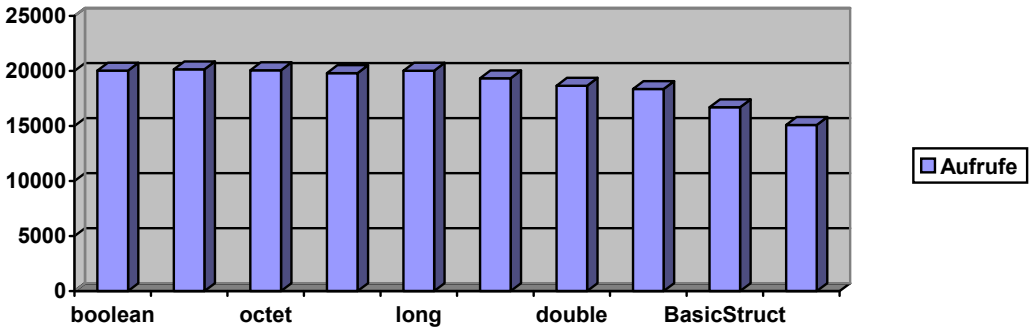
8.1.3 Statische Datentypen bei out-Parametern

Entsprechend ergab sich für out-Parameter unter Verwendung der u. a. IDL:

```
interface NoAnyTestWithOutParam {
    void       booleanOp(in boolean b, out boolean x);
    void       shortOp(in short s, out short t);
    void       octetOp(in octet o, out octet t);
    void       charOp(in char c, out char x);
    void       longOp(in long l, out long x);
    void       floatOp(in float f, out float x);
    void       doubleOp(in double d, out double x);
    void       stringOp(in string s, out string t);
    void       basicStructOp(in basics::BasicStruct b, out basics::BasicStruct x);
    void       personenOp(in  personen::Person p, out personen::Person pers);
};
```

Programmbeispiel 63: Statischer Test mit out-Parametern

Datentyp	Aufrufe in 10 Sekunden
boolean	20028
char	20136
octet	20062
short	19802
long	20022
float	19328
double	18650
string	18360
BasiscStruct	16694
Person	15085



8.1.4 Bewertung

Bei der Verwendung der o. a. statischen IDL zeigt sich, dass der Unterschied zwischen in-, inout- und out-Parametern nur marginal ist. Eine Reihe von Faktoren können die Unterschiede beeinflussen. Ist bspw. ein Objekt aktiv und kann entsprechend schnell Anfragen entgegen nehmen und verarbeiten, oder muss dieses entfernte Objekt erst aktiviert werden? Weitere Unterschiede ergeben sich durch die Entlastung der Serverseite. Sind Objekte, die für die Datenrückgabe zuständig sind, bereits auf der Klientenseite instanziiert worden (inout- bzw. out-Parameter), oder muss das Serverobjekt erst ein eigenes Objekt instanziieren und es als Rückgabewert an den Klienten liefern?

Interessanter ist der Vergleich zwischen den einzelnen Methodenaufrufen in Abhängigkeit zu den zu übertragenden Daten. Wird z. B. bei der Verwendung von inout-Parametern der boolsche Wert ca. 19532 mal übertragen, die Basisstruktur 16202 mal und die Personenstruktur lediglich 15138 mal, so ergibt sich das bereits oben angeführte Phänomen, dass einzelne skalare Datentypen zwar günstiger zu übertragen sind, aber bei mehrfachen Aufrufen die Strukturen zu bevorzugen sind.

Werden mittels der Schnittstelle sukzessive die skalaren Datentypen übertragen, so benötigt man hierzu bei acht Aufrufen im arithmetischen Mittel 15,3717 Millisekunden. Wird die Basisstruktur verwendet, so sind die gleichen Daten bereits in 1,6202 Millisekunden übertragen.

8.2 Datentypen mit Any-Objekten

In der folgenden Grafik ist die Anzahl der Durchläufe bei Verwendung von in-Parametern angegeben. Die Laufzeit des Klienten betrug jeweils 10 Sekunden. Die Länge des Strings betrug 32 Zeichen.

Gemessen wurde in der vorliegenden Untersuchung jeweils die Anzahl der Aufrufe in einem Zeitraum von 10 Sekunden. Das Erzeugen eines Containers und das Einstellen der Daten in diesen generischen Datentyp wird an anderer Stelle untersucht.

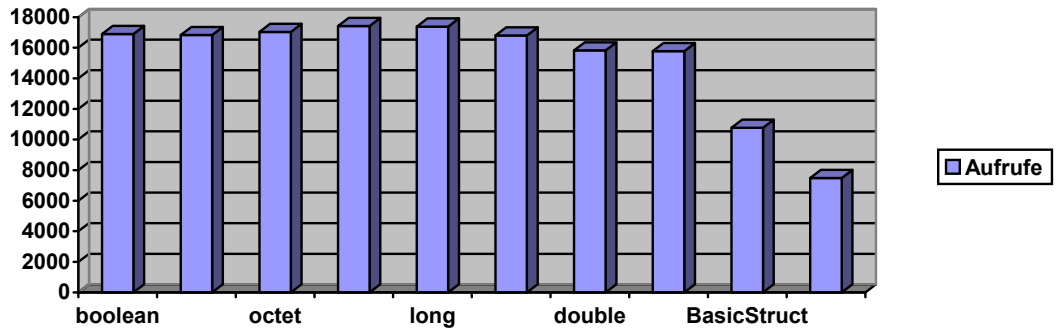
8.2.1 Any als in-Parameter

Für den Aufruf mit dem Any-Objekt als in-Parameter und der Antwort als Container per Rückgabewert ergab sich bei Verwendung der folgenden IDL-Beschreibung:

```
interface AnyTestWithInParam {
    any anyOp(in any a);
};
```

Programmbeispiel 64: Any als in-Parameter

Datentyp	Aufrufe in 10 Sekunden
boolean	16897
char	16833
octet	17033
short	17424
long	17386
float	16798
double	15831
string	15780
BasicStruct	10775
Person	7485



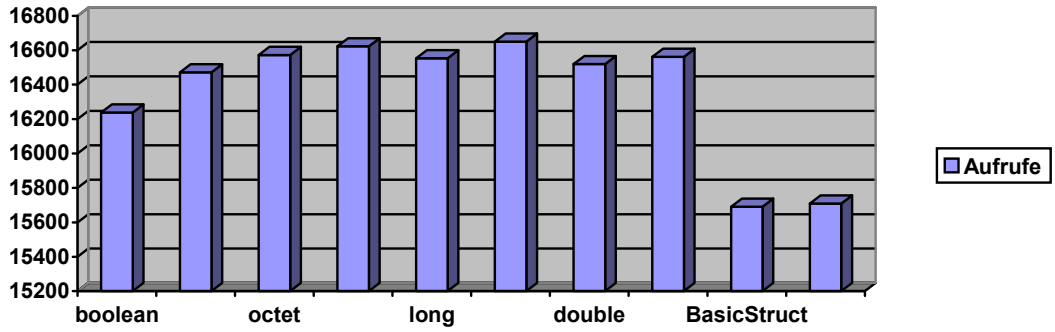
8.2.2 Any als inout-Parameter

Analog für inout-Parameter. Verwendet wurde die IDL-Beschreibung

```
interface AnyTestWithInOutParam {
       void anyOp(inout any a);
};
```

Programmbeispiel 65: Any als inout-Parameter

Datentyp	Aufrufe in 10 Sekunden
boolean	16238
char	16471
octet	16572
short	16622
long	16553
float	16649
double	16519
string	16561
Basisstruktur	15691
Person	15709



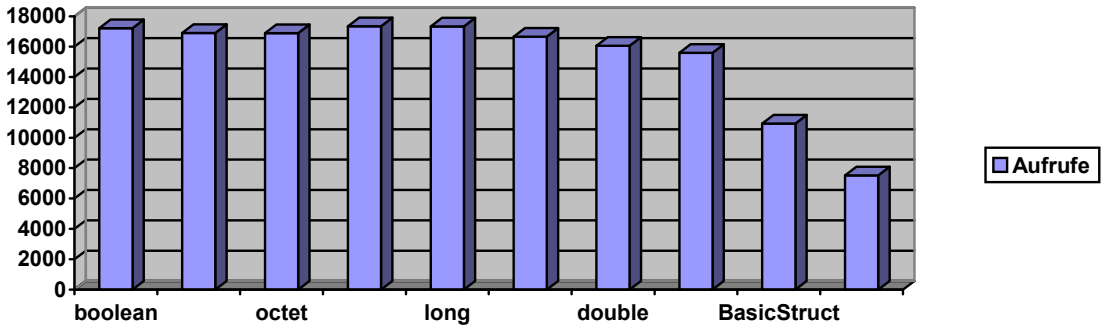
8.2.3 Any als out-Parameter

Entsprechend für out-Parameter bei Verwendung der IDL

```
interface AnyTestWithOutParam {
    void anyOp(in any a, out any x);
};
```

Programmbeispiel 66: Any-Test mit out-Parameter

Datentyp	Aufrufe in 10 Sekunden
boolean	17215
char	16903
octet	16892
short	17348
long	17332
float	16665
double	16062
string	15593
BasicStruct	10936
Person	7524



Es ergibt sich offensichtlich ein ungünstigeres Laufverhalten, als bei den statischen Datentypen. Dies ist primär auf die Allgemeingültigkeit der Kontainer zurückzuführen (vgl. Kapitel 8.2.4 und Kapitel 9).

8.2.4 Bewertung

Der Charme des dynamischen Datentyps Any ist in erster Linie in der starken Vereinfachung der Schnittstellenbeschreibung zu sehen. Eine Änderung der Signatur einer Methode im statischen Fall führt grundsätzlich zu einer Veränderung sowohl bei dem Serverobjekt, als auch auf der Klientenseite. Sind die Daten in IDL beschrieben, so kann bei Verwendung des Datentyps Any die Schnittstelle unverändert bleiben. Zu berücksichtigen ist hierbei auch, dass ein Any-Objekt immer einen TypeCode enthält, über den ggf. die einzelnen Datenmember bei benutzerdefinierten Datenstrukturen per Iteration ermittelt werden können. Zusätzlich ist mit der Einführung des TypeCodes auch eine Versionierung der Schnittstelle durch die IDL-Anweisung

```
#pragma version
```

möglich.

Bei Einsatz von dynamischen Komponenten ist grundsätzlich mit der Herabsetzung der Verarbeitungsgeschwindigkeit zu rechnen. Das basiert im günstigsten Fall lediglich in dem erhöhten Aufwand, der notwendig ist, diese Datentypen zu erzeugen und die Daten serialisiert einzustellen bzw. wieder auszulesen. Im ungünstigsten Fall sind die erhaltenen Daten von der jeweiligen Seite zur Laufzeit zu interpretieren. Dieser Ansatz wird in folgenden Kapiteln detailliert betrachtet.

Im Vergleich mit den in-Parametern im statischen Fall ergibt sich gegenüber dem Any-Ansatz eine Differenz der Methodenaufrufe von ca. 2825 Aufrufen bei einem boolschen Wert bzw. 5254 Aufrufe bei Verwendung der Basisstruktur. Dies entspricht ca. 14,3 % weniger Aufrufe bei der Übertragung des boolschen Wertes bzw. 32,4 % weniger Aufrufe bei der komplexeren Basisstruktur.

9 Zeitaufwand zum Erzeugen von dynamischen Objekten

In den o. a. Vergleichen wurde der benötigte Aufwand beim Verschicken von Daten im statischen Fall und unter Verwendung von Any-Kontainern untersucht. Von Interesse ist zusätzlich eine Betrachtung bzgl. des benötigen Aufwandes, um Daten serialisiert in einen Container einzustellen, und auch der Aufwand diese allgemeingültigen Objekte zu instanziieren.

9.1 Das Erzeugen von Any-Objekten

Das Erzeugen eines allgemeingültigen Containers ist zeitaufwendiger, als das Erzeugen einer benutzerdefinierten Datenstruktur. Zumindest auf Seiten des Klienten oder bei Serverobjekten, die sich nicht nach jedem Methodenaufruf deaktivieren, kann dieser Container in einer Initialisierungsphase einmal instanziiert und entsprechend oft wieder verwendet werden.

Ein Any-Objekt wird über das Pseudoobjekt ORB erzeugt, das man bei der Initialisierung des Objektes Request Brokers als Rückgabewert erhält

```
org.omg.CORBA.ORB  orb     = org.omg.ORB.init(args, props);
org.omg.CORBA.Any  a       = orb.create_any();
```

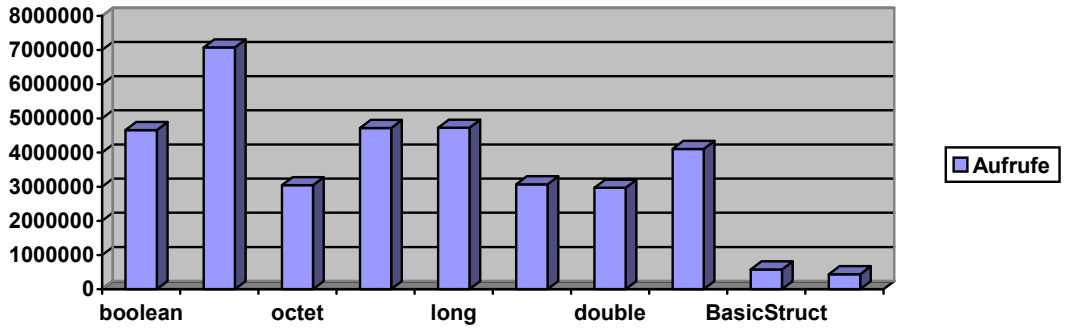
Hierbei sind *args* die übergebenen Kommandozeilenargumente und *props* bezeichnet mögliche zusätzliche Eigenschaften.

In einem Testlauf ist es auf der oben beschriebenen Hardware möglich, im Zeitraum von 10 Sekunden 6760927 Any-Objekte zu erzeugen. Dies entspricht einer Anzahl von Allokierungen in einer Größenordnung von ca. 676 Objekten pro Millisekunde und stellt eine vernachlässigbare Größe dar.

9.2 Das Erzeugen von DynAny-Objekten

Das Erzeugen des lokalen Objektes DynAny wird i. A. über TypeCodes oder durch existierende Any-Objekte realisiert. In der folgenden Tabelle und der hierzu korrespondierenden Grafik ist in einem Testlauf von 10 Sekunden dieser Vorgang in Abhängigkeit der TypeCode *_tc_boolean*, *_tc_char*, *_tc_short*, *_tc_long*, *_tc_float*, *_tc_double*, *_tc_string* und den generierten TypeCodes für die Basisstruktur bzw. für die Personenstruktur dokumentiert.

Datentyp	**Instanziierungen von DynAny-Objekten in 10 Sekunden**
boolean	4655687
char	7074951
octet	3043951
short	4718596
long	4724510
float	3070832
double	2971463
string	4107262
BasicStruct	580552
Person	439181


8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
boolean
octet
long
double
BasicStruct
Aufrufe

10 Zeitaufwand zum Einstellen von Daten in dynamische Objekte

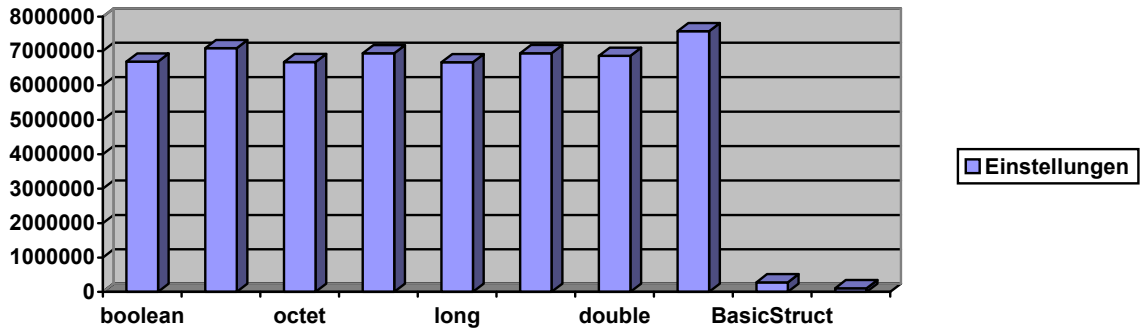
Von Interesse ist auch der Zeitaufwand, der benötigt wird, um ein Objekt serialisiert in ein Any- oder DynAny-Objekt einzustellen. Das Einstellen erfolgt bei primitiven Datentypen über die vorgesehenen Methoden des Any-Objektes *insert_boolean(boolean b)*, *insert_char(char c)* usw. Bei benutzerdefinierten Datentypen werden die Methoden *insert(…)* der jeweiligen Helperklasse verwendet. Das Auspacken der Daten aus einem Any-Objekt erfolgt bei primitiven Datentypen durch die Methoden des Any-Objektes *<return-value> extract_from()*, bzw. bei benutzerdefinierten Datentypen über die Methode der jeweiligen Helperklasse *<return-value>Helper.extract()*.

10.1 Das Einstellen von Daten in ein Any-Objekt

Das Erzeugen eines Any-Objektes ist i. d. R. ein einmaliger Vorgang, da dieser Datentyp i. A. beliebig oft wieder verwendet werden kann. Von Interesse ist der Aufwand, der notwendig ist, einen skalaren oder einen in IDL beschriebenen Datentyp in diesen Container einzustellen bzw. wieder auszulesen.

Die folgende Tabelle und die hierzu gehörige Grafik dokumentieren den jeweils benötigten Aufwand, Daten serialisiert in ein Any-Objekt einzustellen. Getestet wurde wieder in einem Testlauf von 10 Sekunden unter Verwendung der Skalare *boolean*, *char*, *octet*, *float*, *double*, *string* und der o. a. Strukturen BasicStruct, sowie Person.

Datentyp	Einstellungen in 10 Sekunden
boolean	6692122
char	7079618
octet	6679360
short	6933242
long	6672363
float	6933933
double	6856503
string	7574316
BasicStruct	279479
Person	108847



Beim Einstellen von Daten kehrt sich das Verhalten, das bei den Methodenaufrufen zu erkennen ist, in das genaue Gegenteil um. Je einfacher ein Datentyp ist, desto geringer ist der

Serialisierungsaufwand und das Einstellen dieser Daten in das Any-Objekt. Andererseits ist hier zu berücksichtigen, dass das Einstellen von benutzerdefinierten Datenstrukturen wie z. B. der recht komplexen Struktur Person im Zeitrahmen von ca. 11 Einstellungen pro Millisekunde immer noch ein recht günstiges Zeitverhalten aufweist.

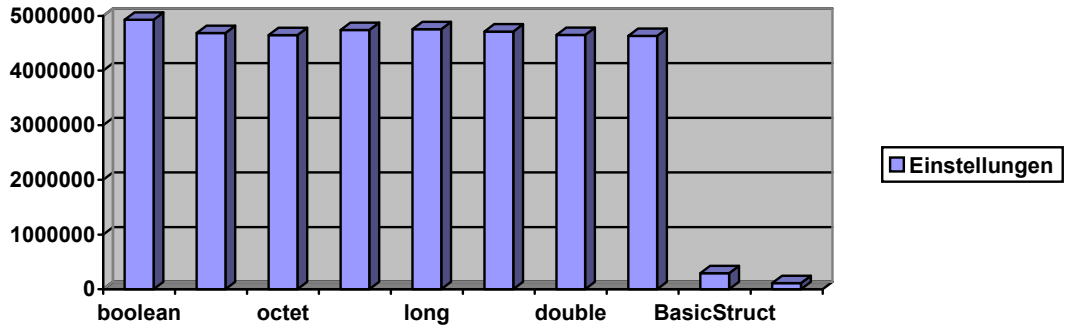
10.2 Das Einstellen von Daten in ein DynAny-Objekt

Analog zu einem Any-Objekt ist das Erzeugen eines DynAny-Objektes i. d. R. ein einmaliger Vorgang, da dieser Datentyp i. A. beliebig oft wieder verwendet werden kann. Von Interesse ist der Aufwand, der notwendig ist, einen skalaren oder einen in IDL beschriebenen Datentyp in diesen DynAny-Container einzustellen bzw. wieder auszulesen. Der Aufwand ist i. A. höher als bei einem Any-Objekt, da es sich hierbei um ein Pseudoobjekt handelt. Um Daten dieses Typs zu versenden ist eine zusätzliche Serialisierung des Inhalts zu einem Any-Objekt notwendig.

Die folgende Tabelle und die hierzu gehörige Grafik dokumentieren den jeweils benötigten Aufwand, Daten serialisiert in ein DynAny-Objekt einzustellen. Getestet wurde wieder in einem Testlauf von 10 Sekunden unter Verwendung der Skalare *boolean*, *char*, *octet*, *float*, *double*, *string* und der o. a. Strukturen BasicStruct, sowie Person.

Die Übertragung eines Pseudoobjektes wie DynAny ist nicht möglich, sondern es findet eine Serialisierung des Inhaltes in ein Any-Objekt statt. Damit ist ein Test bzgl. der Verarbeitungsgeschwindigkeiten bei der Übertragung identisch mit den Zeiten, die benötigt werden, um ein Any-Objekt zu übertragen. Programmbeispiele für das Einstellen/Auslesen von Daten in ein DynAny-Objekt sind in Programmbeispiele 37, 45 und 46 angegeben.

Datentyp	Einstellungen bei DynAny-Objekten in 10 Sekunden
boolean	4929349
char	4686480
octet	4648105
short	4741855
long	4756312
float	4712936
double	4651523
string	4635019
BasicStruct	290530
Person	108847



Beim Einstellen von Daten in ein DynAny-Objekt ist ein ähnliches Laufzeitverhalten zu erkennen, wie bei den Any-Objekten.

Damit ergibt sich ein ungünstiges Laufzeitverhalten bei der Verwendung von generischen Datentypen gegenüber den statischen. Bei einer losen Koppelung zwischen verschiedenen verteilten Anwendungen, bspw. im Bereich Business-2-Business, kann durch den Einsatz von generischen Datentypen als Return-, in-, out- oder inout-Wert die Schnittstelle zwischen beiden Anwendungen stabil gehalten werden.

In einer in sich geschlossenen, verteilten Anwendung sollte aufgrund der bisherigen Ergebnisse auf den Einsatz der dynamischen CORBA-Komponenten weitestgehend verzichtet werden.

Beim Arbeiten mit unbekannten Objekten hat man jedoch keine Alternative und muss die Möglichkeiten des dynamischen CORBA's nutzen oder auf die ebenfalls dynamischen Web Services (s. Kapitel 17) ausweichen.

11 Komponenten und ihre Charakteristiken

Um zu einem Vergleich zwischen CORBA, Enterprise JavaBeans und auch zu den Web Services zu gelangen, muss der Begriff der Komponente eingeführt werden.

Die CORBA-Produkte in der Spezifikation 2.x beinhalten i. A. keine Komponenten, sondern sind als reines Objektmodell zu betrachten. In der Spezifikation der Enterprise JavaBeans und in der neuen CORBA 3 Spezifikation sind bereits Komponenten beschrieben. Das primäre Ziel ist auch hier, den Entwickler von monotonen Programmieraufgaben zu entlasten und dem Verbraucher einen Baustein zu liefern, der durch Konfiguration an seine Bedürfnisse angepasst werden kann.

Definition 36: Komponenten

Unter einer *Komponente* versteht man eine Einheit (engl. deployment unit), die einen Dienst zur Verfügung stellt.

Eine Komponente besteht aus Objekten, in denen die Geschäftslogik implementiert ist, und aus generierter Software. Welche Softwarebausteine für eine Komponente generiert werden, wird durch eine oder mehrere externe Beschreibungsdateien bestimmt.

Definition 37: Beschreibungsdateien (Deployment Deskriptoren) für Komponenten

Die Beschreibungsdateien der Komponenten werden als *Deploymentdeskriptoren* bezeichnet.

Das Ziel der Komponententechnologie soll sein, dass fertige Bausteine entwickelt werden können, und dass ein Kunde durch die externen Beschreibungsdateien diese Bausteine an seine eigenen individuellen Bedürfnisse anpasst.

Um eine Komponente synchron oder asynchron ansprechen zu können, muss jede Komponente eine Schnittstelle bestehend aus Operationen, Eigenschaften und Ereignissen zur Verfügung stellen. Auch hier ist die Sprachunabhängigkeit von Implementierungssprachen von Interesse. Es sollte somit für den Nutzer einer Komponente transparent sein, ob dieser Baustein in C++, Java oder Visual Basic implementiert ist.

Beispiele von Komponenten sind die SessionBeans, EntityBeans und die Message Driven Beans der EJB 2.0 Spezifikation. Im CORBA-Umfeld kann das TP-Framework des BEA ORBs **W**eb**L**ogic **E**nterprise (WLE) als eine der wenigen CORBA-Komponenten angesehen werden. Das TP-Framework vereinfacht die Programmierung des **P**ortable **O**bject **A**dapters (POA) und kann bei geeigneter Konfiguration automatisch Objekte im Rahmen einer globalen Transaktionen aktivieren und bei der Deaktivierung den jeweiligen Objekten einen Grund für die Beendigung der Transaktionen liefern (commit, rollback, timeout oder Applikationsende). Dazu verwendet das TP-Framework die beiden Methoden *activate_object* und *deactivate_object* des POAs.

Wie im CORBA- oder EJB-Umfeld üblich, wird basierend auf der jeweiligen Schnittstellenbeschreibung die benötigte Netzsoftware (Stubs, Skeletons etc.) generiert.

Im Gegensatz zu den Objekten werden Komponenten nicht direkt adressiert, sondern man adressiert eigens für diese Komponente generierte Software, die verantwortlich ist für administrativ vereinbarte Vorarbeiten, der Delegation des Aufrufs an die eigentliche Komponente und an eine Nachbearbeitung. Bspw. kann eine Komponente derart konfiguriert werden, dass bei Aufruf bestimmter Methoden eine Transaktion erzeugt wird, die Komponente im Rahmen dieser Transaktion angesprochen wird und nach Beendigung der Methode automatisch die Transaktion beendet wird.

Definition 38: Container für Komponenten

Die für eine Komponente auf Basis von Konfigurationsvereinbarungen erzeugte Software wird i. A. als *Container* bezeichnet.

Die von einem Container zu leistenden Arbeiten können vielseitig sein. Zu erwähnen sind

1. automatisch Transaktionen zu erzeugen und zu beenden. Dies geschieht in Abhängigkeit von einer vorgegebenen Transaktionspolitik, die global für eine Komponente oder für jede einzelne Methode der Komponente separat gesetzt wird.
2. die Zugriffsrechte des Aufrufers zu überprüfen und ggf. die Verarbeitung zu verweigern.
3. ein Pooling der Komponenten um deren Management zu ermöglichen. Das Instanziieren von Objekten ist zeitlich eine kostspielige Angelegenheit und die Verarbeitungsgeschwindigkeit kann erhöht werden, wenn bereits existierende Objekte bzw. Komponenten vorrätig gehalten werden.
4. die Möglichkeit, bei persistenten Objekten automatisch den Objektinhalt aus einer Datenbank zu lesen, das Objekt mit diesen Daten zu initialisieren und nach erfolgter Arbeit den neuen Zustand in die Datenbank zurückzuschreiben.
5. Lastverteilung bei zustandslosen SessionBeans oder bei MessageDrivenBeans.
6. Fehlertoleranz durch die Verwendung von Replikaten, die in dem Stub des Klienten als Alternative eingetragen sind, so dass bei dem Ausfall der originären Komponente eine direkte Weiterleitung an eine andere, gleichwertige Komponente erfolgen kann.

Damit ergibt sich, dass eine Komponente respektive des hierzu korrespondierenden Containers sich auf vordefinierte Weise, die durch die Deployment Deskriptoren beschrieben werden, verhalten muss. Für eine Komponente sind die folgenden Charakteristiken sinnvoll:

1. Eine Komponente verfügt über Eigenschaften. Ist eine Komponente zustandsbehaftet, so wird dies nicht durch den Entwickler programmtechnisch festgelegt, sondern durch eine Konfigurationsmaßnahme für den Container der Komponente. Die Eigenschaft wäre in diesem Beispiel also *zustandsbehaftet* (stateful).
2. Eine Komponente muss von einem Klienten angesprochen werden. Dazu bedarf es der Schnittstellenbeschreibung, d. h. die entfernten Methoden (Operationen), sowie die zu sendenden bzw. empfangenden Daten, werden festgelegt.
3. Eine Komponente reagiert auf Ereignisse (events). Ist bspw. ein zustandsbehaftetes Bean über einen vorgegebenen Zeitraum inaktiv, so wird ein Ereignis ausgelöst und die Komponente kann u. a. passiviert oder entfernt werden.
4. Eine Komponente ist allgegenwärtig (ubiquitously deployable). Es ist somit unabhängig, mit welchem Produkt eine Komponente erstellt wurde. Durch ein ggf. neues Deployment muss diese Komponente in allen Produkten installierbar und lauffähig sein.
5. Eine Komponente ist wieder verwendbar. Hierbei handelt es sich um ein Wesensmerkmal der Objektorientierung. Die Wiederverwendbarkeit einer Komponente kann, wie in der Objektorientierung üblich, durch Aggregation oder durch Vererbung herbeigeführt werden.
6. Eine Komponente ist teilbar (shareable). Unter bestimmten Voraussetzungen kann eine Komponente mehreren Klienten dienen. Dies setzt voraus, dass dieser angebotene Dienst zustandslos ist; anderenfalls ist das Arbeiten mit genau einer Komponente mit einer Exklusivität versehen. I. d. R. kann im Falle von zustandsbehafteten Komponenten die Middleware mehrere Komponenten vorrätig halten und somit parallel mehrere Klientenaufträge bearbeiten.
7. Eine Komponente ist verteilbar. Die benötigte Netzsoftware ist analog zu CORBA-Objekten standardisiert und wird durch einen speziellen Compiler generiert. Mit der Möglichkeit zu kommunizieren ergibt sich unmittelbar die beliebige Verteilbarkeit.
8. Eine Komponente verfügt über bestimmte Automatismen. Durch geeignete Konfiguration kann eine Komponente z. B. bei Aufruf bestimmter Methoden eine Transaktion erzeugen und diese Methode im Rahmen dieser Transaktion laufen lassen, oder eine Komponente kann die Authentizität des Aufrufers prüfen und feststellen, ob dieser Klient bestimmte

Methoden nutzen darf. Von Interesse ist auch die Möglichkeit, dass eine Komponente ein Objektpooling unterstützt.

9. Eine Komponente beinhaltet genau den Programmkode, den sie tatsächlich benötigt. Diese optionale Forderung steht in engem Zusammenhang mit dem o. a. Automatismus. Eine Komponente kann die Frage der Persistenz, das Objektpooling, Sicherheit im Sinne von Authentifizierung und Autorisierung oder benötigte Transaktionsverarbeitung selbstständig übernehmen. Entsprechend wird ein Objekt während seiner Implementierung nicht derart entwickelt, ob es einen Zustand halten muss oder nicht. Diese Entscheidung wird in der Beschreibungsdatei vorgenommen, indem diese Komponente mit dem Charakteristikum *Stateful* versehen wird. Das Halten und ggf. Abspeichern des Zustandes und sein späteres Einladen ist Aufgabe der generierten Software, also die Aufgabe des Containers. Der Entwickler einer Komponente wird entsprechend entlastet und implementiert im Wesentlichen seine Geschäftslogik.
10. Eine Komponente ist selbstbeschreibend. Dies beinhaltet analog zu dem Interface Repository die Forderung, dass eine Komponente über Metadaten verfügt und ein Klient zur Laufzeit Informationen bzgl. des Aufbaus der Komponente erfragen kann.
11. Bei Verwendung von Replikaten kann für eine Komponente ein Lastverteilungsmechanismus vorgegeben werden.

Im folgenden Kapitel werden die Enterprise JavaBeans vorgestellt, die als erste Standardisierung von serverseitigen Komponenten existieren. Die Einführung der Enterprise JavaBeans wird in nachfolgenden Kapiteln für einen Vergleich mit den CORBA-Objekten im Zusammenhang mit enger Koppelung verwendet.

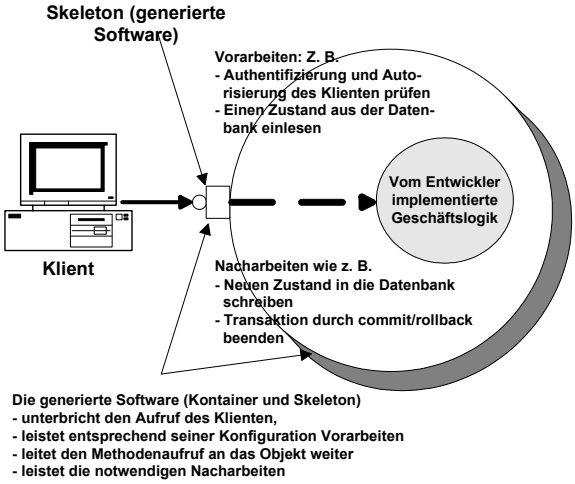


Abbildung 21: Grobaufbau einer Komponente

12 Die Beschreibungssprache XML

Die e**X**tensible **M**arkup **L**anguage (XML) ist eine mächtige, allgemeingültige Beschreibungssprache, die bzgl. ihrer Notation angelehnt ist an die **H**yper**t**ext **M**arkup **L**anguage (HTML), bzw. allgemeiner an der **S**tandard **G**eneralized **M**arkup **L**anguage (SGML).

XML ist neben den Webdiensten auch für die Enterprise JavaBeans von Interesse. Das gesamte Deployment, also die externe Beschreibung für zusätzlich generierte Software im Zusammenhang mit Komponenten, wird durch XML-Deploymentdekriptoren beschrieben.

In den folgenden Unterkapiteln wird eine kurze Einführung in die Themenbereiche
1. XML Instanzen: Die Regeln, um syntaktisch korrekte XML-Dokumente zu erstellen.
2. XML Schemata: Die syntaktische Beschreibung von XML-Datentypen mit XML.
3. XML Namensräume: Die Definition von Mechanismen, um mehrere XML Dokumente ohne Namenskollisionen in einem Dokument zu nutzen.

vorgenommen.

12.1 Dokumenten- vs. datenbasiertes XML

Grundsätzlich können zwei Anwendungsbereiche für den Einsatz von XML-Technologien identifiziert werden.

Zum einen ist es eine dokumentenbasierte Anwendung und, insbesondere im Zusammenhang mit Web Services relevant, eine datenbasierte Anwendung.

12.1.1 Dokumentenbasiertes XML

Bei einem dokumentenbasierten Ansatz wird XML für einen Datenaustausch verwendet, wobei die Daten als ein zusammenhängendes, von einem menschlichen Benutzer gelesenes, Dokument zu sehen sind. Es wird hierbei vorausgesetzt, dass die syntaktischen Regeln i. A. durch eine DTD (**D**ocument **T**ype **D**escription) oder durch ein XML Schema beschrieben sind.

Definition 39: DTD und XML Schema
1. Eine DTD (Document Type Description) ist eine syntaktische Beschreibung über den Aufbau eines XML-Dokumentes auf Basis der erweiterten Backus-Naur-Form.
2. Ein XML Schema ist eine syntaktische Beschreibung über den Aufbau eines XML-Dokumentes auf Basis der Beschreibungssprache XML.

Dokumentenbasiertes XML ähnelt der Beschreibungssprache HTML, wobei jedoch die Möglichkeit gegeben ist, eigene Tags zu definieren, die innerhalb des Dokumentes eine gewisse semantische Bedeutung haben.

```
<Person>
    <Nachname>...</Nachname>
    <!-- weitere Angaben zur Person -->
</Person>
```

Programmbeispiel 67: XML-Struktur

12.1.2 Datenbasiertes XML

Das datenbasierte XML wird für die Übertragung von Daten für entfernte Funktionenaufrufe auf Basis des Protokolls SOAP eingesetzt. Ein weiterer, wichtiger Bereich sind Konfigurationsdateien für Applikationsserver oder die Deployment Deskriptoren von Enterprise JavaBeans.

12.2 XML Instanzen

Die Struktur und die Formatierung in einem XML Dokument muss den Regeln der XML Instanz Syntax genügen.

Ein Dokument kann einen optionalen Vorspann enthalten, gefolgt von einem Wurzelelement, das den Inhalt des Dokumentes enthält.

Der optionale Vorspann kann für die Identifikation des Dokumentes als ein XML Dokument eine XML Versionsangabe, eine interne oder externe Dokumentenbeschreibung enthalten. Zusätzlich ist die Möglichkeit gegeben, Metainformationen bzgl. des Inhaltes mit aufzunehmen.

Um als XML Dokument identifiziert zu werden, muss die spezielle Direktive *processing instruction* verwendet werden. Die Syntax lautet
<?PI-Ziel ... ?>.
Beispielsweise wird die Version des XML Dokuments angegeben durch
<?xml version="1.0" encoding="UTF-8"?>.

Wird diese XML Deklaration nicht explizit angegeben, so wird der Parser standardmäßig die Version 1.0 annehmen und die verarbeitende Einheit versucht aus dem erhaltenen Datenstrom die korrespondierende Kodierungsregel zu ermitteln.

Der Term *ELEMENT* ist ein technischer Name für die Definition einer Start- und Ende-Formatangabe (tags).

```
<ejb-jar>
    <enterprise-beans>
        <session>
                <!—Beschreibung eines SessionBeans -->
        </session>
    </enterprise-beans>
</ejb-jar>
```

Beispiel 11: XML-Tags

Die Tags <ejb-jar>, <enterprise-beans> und <session> sind Start-Formatangaben, die durch </ejb-jar>, </enterprise-beans> und </session> wieder korrekt abgeschlossen werden müssen.

Ein Starttag kann Null oder mehrere Attribute besitzen, wobei im Sinne von XML ein Attribut ein Namens-Werte-Paar ist.

```
<auftrag id="4711" versendet_am="2002-30-06"/>
```

In diesem Beispiel besitzt das Tag keinen Rumpf und schließt sich selbst wieder ab. Besitzt das Tag einen Rumpf, so könnte eine mögliche Darstellung sein:

```
<auftrag id="4711" versendet_am="2002-30-06">
```

```
    <Rechnungsanschrift>
        <Firma>...</Firma>
        <Strasse>...</Strasse>
        <Ort>...</Ort>
        <PLZ>...</PLZ>
    </Rechnungsanschrift>
</auftrag>
```

Programmbeispiel 68: XML-Tag mit und ohne Rumpf

12.3 XML Namensräume

Eine wichtige Eigenschaft bei umfangreichen Projekten mit entsprechend vielen Beschreibungsdateien ist die Vermeidung von Namenskollisionen. XML stellt hierfür analog zu Programmiersprachen wie C++ und Java, oder wie die Beschreibungssprache IDL, das Konstrukt eines Namensraumes zur Verfügung.

Ein Namensraum wird gebildet aus einem Identifizierer für diesen Namensraum, gefolgt von einem lokalen Namen.

Für die Identifizierer von Namensräumen verwendet XML häufig URIs (**U**niform **R**esource **I**dentifier). Damit kann der Identifizierer interpretiert werden als ein Name, aber auch als die Lokation eines anderen Dokuments. Im Zusammenhang mit Web Services wäre ein typisches Beispiel der Namensraum

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
```

Hiermit wird sowohl ein Namensraum vorgegeben, als auch ein Verweis angegeben, der Auskunft über die verwendete SOAP-Implementierung gibt.

Definition 40: Namensräume, Namensidentifizierer, vollqualifizierte Namen

1. Der Präfix eines Namensraumes ist ein beliebiger Name gefolgt von einem Doppelpunkt.
2. Ein Namensraumidentifizierer wird durch ein Präfix eingeleitet.
3. Ein vollqualifizierter Name ist das Präfix, gefolgt von dem obligatorischen Doppelpunkt, gefolgt von einem lokalen Namen.

```
<msg:message von="kwalpert@consavis.com" to="bestellung@beliebiges_kaufhaus"
    sent="2002-05-07"
    xmlns:msg="http://www.ecommerce_company.com/namespace/message"
    xmlns:order="http://www.beliebiges_kaufhaus.com/namespace/order"
    <msg:text>
        Bestellung von ...
    </msg:text>
    <msg:attachment>
        <msg:description>Anhang eines Auftrages</msg:description>
        <msg:item>
                <order:order id="4711" submitted="2002-05-07">
                <order:billTo id="address">
                        ...
                </order:billTo>
                <order:shipTo href="Address"/>
                <!-- weitere Angaben zum Auftrag -->
        </msg:item>
    </msg:attachment>
</msg:message>
```

Beispiel 12: XML-Dokument einer Bestellung

12.4 Die Document Type Description

Eine *Document Type Description* ist eine optionale Eigenschaft von XML Dokumenten. In dieser Typbeschreibung wird eine Menge von Regeln beschrieben, die Auskunft darüber geben, welche Elemente und Attribute in dem Dokument enthalten sind.

Jedes Element in der Dokumentenstruktur genügt einer Modellgruppe, in der z. B. die Vielfachheit von Vorkommnissen dieses Elementes usw. festgelegt wird.

Wird durch eine DTD bspw. eine Folge beschrieben, so wird durch Klammerung vorgegeben, wie eine Folge aufgebaut ist. Die Folge (X, Y, Z) besagt, dass die Folge aus einem X, gefolgt von einem Element Y, gefolgt von einem Element Z, besteht. Die Folge (X | Y | Z) besteht aus genau einem Kindelement, also aus einem X, einem Y oder einem Z. Eine Schachtelung von Folgen ist möglich. Die Folge (A | (X, Y, Z), B, (C | D)) kann u. a. expandiert werden zu
- A, B, C
- A, B, D
- X, Y, Z, B, D

usw..

In einer Folge kann die Vielfachheit von Elementen durch die Angabe von ‚*', ‚?' und ‚+' festgelegt werden. Die Bedeutung hierbei ist:
- *: Null oder beliebig viele Vorkommnisses des Elementes.
- +: Das Element hat die Vielfachheit Eins oder Null.
- ?: Das Element hat die Vielfachheit Null oder Eins.

Durch die syntaktische Beschreibung eines Dokumentes durch eine DTD besitzt ein XML-Parser die Möglichkeit, das Dokument auf seine Korrektheit zu überprüfen.

Die wichtigsten Nachteile einer Syntaxbeschreibung auf Basis einer DTD sind:
1. Die syntaktische Beschreibung eines Dokumentes erfolgt nicht durch Sprachkonstrukte von XML.
2. DTDs wurden vor der Einführung von XML-Schema's eingeführt und besitzen nur unzureichende Möglichkeiten, mit Namensräumen umzugehen. Insbesondere bei datenbasierten XML-Applikationen ergeben sich hiermit nicht tolerierbare Restriktionen. Bspw. kann eine Adresse sich zum einen auf eine Person beziehen, die die Lieferung erhält, zum anderen aber auch die Rechnungsanschrift sein.
3. Die Wiederverwendbarkeit und Erweiterbarkeit von XML-Dokumenten ist durch DTDs nur unzureichend beschreibbar.
4. DTDs besitzen keine Notation für die Beschreibung von Datentypen. In der Praxis ist es jedoch unerlässlich, auch andere Datentypen außer einem String zu übertragen.

12.5 XML Schemata

Um die o. a. Probleme der DTDs zu umgehen, wurden die XML Schema entwickelt. Hiermit ist es möglich, ein Dokument durch Hilfsmittel von XML bzgl. ihres syntaktischen Aufbaus zu beschreiben, Namensräume zu definieren und mit primitiven Datentypen bzw. benutzerdefinierten Datentypen in einem XML-Dokument zu arbeiten.

```
<?xml version="1.0" encoding="UTF-8"?>
<order:order
    xmlns:order="http://<host>/namespace/order"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://<host>/namespace/order
                        http://<host>/schema/namespace/order.xsd"

</order:order>
```

Beispiel 13: XML-Namensräume

Mit der ersten Namensraumangabe wird das Schema, das diese Applikation verwendet beschrieben. Der nachfolgende Namensraum http://www.w3.org/2001/XMLSchema-instance besitzt eine spezielle Bedeutung. Unter der angegebenen URL findet sich ein Dokument, das die Anzahl von Attributen, die Teil der Schema-Spezifikation sind, beschreibt. Per Konvention werden derartige Namensräume immer durch das Präfix *xsi* eingeleitet. Die Bindung von der Applikation an ein bestimmtes Schema erfolgt durch die Angabe von xsi:schemaLocation.

12.5.1 Die skalaren Datentypen

In der XML Schema Spezifikation werden neben den primitiven Datentypen wie int, float etc. die folgenden Datentypen definiert:

Datentyp	Beschreibung/Beispiel
String	Zeichenkette. Entspricht in Java der Klasse java.lang.String.
base64Binary	$\in$ [A-Z, a-z, 0-9, +, -]
hexBinary	0AB7
integer	entspricht in Java dem Skalar int
positiveInteger	natürliche Zahl ohne Null ($\in$ $\mathbf{N}\backslash\{0\}$).
negativeInteger	die Menge der ganzen Zahlen ohne die Menge der natürlichen Zahlen einschließlich 0 ($\mathbf{Z}\backslash\mathbf{N}_0$).
nonNegativeInteger	die Menge der natürlichen Zahlen einschließlich 0 ($\mathbf{N}_0$)
nonPositiveInteger	die Menge der ganzen Zahlen ohne die Menge der natürlichen Zahlen
decimal	Gleitpunktzahl
boolean	boolscher Wert $\in$ { true, false, 0, 1 }.
time	Zeitangabe in der Form hh:mm:ss.ms.
dateTime	Datum und Uhrzeit in der Form 2002-06-30T13:20:00.000-05:00 (der 30 Juni 2002 um 13:20 Uhr EST (Eastern Standard Time), 5 Stunden hinter der Zeitangabe CUT (Coordinated Universial Time).
duration	Zeitspanne, z. B. P1Y2M3DT10H30M12.3S (1 Jahr (1Y), 2 Monate (2M), 3 Tage (3D), zehn Stunden (10H), 30 Minuten (30M) und 12.3 Sekunden (12.3S).
date	Datum in der Form yyyy-mm-dd (Jahr-Monat-Tag).
name	XML 1.0 Namenstyp
QName	vollqualifizierter Namensraum, XML QNametyp
anyURI	Uniform Resource Identifier
ID	XML 1.0 ID, Attributetyp
IDREF	XML 1.0 IDREF, Attributetyp

Tabelle 6: XML-Datentypen

12.5.2 Benutzerdefinierte Datentypen

Durch das Element *xsd:complexType* kann die Definition eines benutzerdefinierten Datentyps eingeleitet werden. Grundsätzlich sind die folgenden Definitionen möglich:

1. xsd:sequence: Eine Folge von Elementen.
2. xsd:choice: Erlaubt die Auswahl von einem Element aus einer Menge von Elementen.
3. xsd:all: Erlaubt einer bestimmten Menge von Elementen einmal oder gar nicht in einer vorgegebenen Reihenfolge zu erscheinen.
4. xsd:group: Referenziert eine Gruppe, die an anderer Stelle definiert ist.

Zur Illustration wird nachfolgend ein benutzerdefinierter Datentyp beschrieben, der eine Adresse repräsentiert:

```
<xsd:complexType name="addressType">
    <xsd:sequence>
        <xsd:element   name="Nachname"       type="xsd:string" minOccurs="0"/>
        <xsd:element   name="Strasse"        type="xsd:string" maxOccurs="unbounded"/>
        <xsd:element   name="Ort"            type="xsd:string"/>
        <xsd:element   name="PLZ"            type="xsd:nonNegativeInteger"/>
    </xsd:sequence>
</xsd:complexType>
```

Programmbeispiel 69: Benutzerdefinierte Datentypen in XML

12.6 Das Parsen von XML

Für das Parsen von XML-Dateien existieren eine Reihe von Standards bzw. de facto Standards. Zwei wichtige Vertreter sind zum einen der SAX-Parser (**S**imple **A**PI for **X**ML) und zum anderen der DOM-Parser (**D**ocument **O**bject **M**odel). Im Zusammenhang mit den Web Services wurden zwei weitere wichtige Parser, nämlich JAXM (**J**ava **A**PI for **X**ML **M**essaging) und JAX-RPC (**J**ava **A**PI for **X**ML-based **RPC**), von der Java Gemeinde vorgestellt. Diese beiden Parser werden aufgrund ihrer engen Beziehung zu dem SOAP-Protokoll in einem späteren Kapitel vorgestellt.

12.6.1 Der SAX-Parser

Ein SAX-Parser ist ein zustandsloser Parser in dem Sinne, dass er ein XML-Dokument liest und bei Eintreten bestimmter Ereignisse entsprechende Callback-Methoden aufruft. Nach Abarbeitung dieser Methoden ist der ggw. Zustand, also das gelesene Datum, verworfen. Die zur Verfügungstellung dieser Einsprungmethoden und die Sicherung der Daten ist Aufgabe eines Entwicklers.

Der SAX-Parser gibt dazu u. a. die beiden Schnittstellen *org.xml.sax.ContentHandler* und *org.xml.sax.ErrorHandler* vor. Die folgenden Methoden sind von dem Entwickler zur Verfügung zu stellen:

```
public interface ContentHandler {
    // Über den gesetzten Locator kann in einem Fehlerfall abgefragt werden, wo dieser Fehler
    // in dem XML-Dokument auftrat.

    public void setDocumentLocator(Locator locator);


    //  startDocument wird als erste Methode im gesamten Parsing-Prozess aufgerufen.

    public void startDocument() throws SAXException;
```

```
// endDocument wird als letzte Methode im gesamten Parsing-Prozess aufgerufen.

public void endDocument() throws SAXException;


// Kann verwendet werden, um ein Präfix zu setzen und später zu berücksichtigen.

public void startPrefixMapping(String prefix, String uri) throws SAXException;


// Kann analog für die Auflösung der Präfixe in der Datenstruktur verwendet werden.

public void endPrefixMapping(String prefix) throws SAXException;


// Sobald der Parser ein XML-Tag eingelesen hat, wird diese Methode aufgerufen.
// Anhand der übergebenen Attributes kann der Wert der eventuell vorhandenen Attribute
// abgefragt werden (z. B. <application id="4711">...</applications> erlaubt die Abfrage
// des Wertes "4711" des Attributes mit dem Namen 'id'. Daten, die zwischen dem Anfangs-
// und dem Endtag stehen, können in dieser Methode nicht abgefragt werden. Von Interesse
// ist zusätzlich der Parameter 'name', der den relativen Tagnamen, in diesem Beispiel
// 'applications' enthält.

public void startElement( String         namespaceURI,
                          String         localName,
                          String         name,
                          Attributes     atts) throws SAXException;


// die folgende Methode wird verwendet, um die Daten, die zwischen Start- und Endtag
// stehen, zu ermitteln. Die relevanten Informationen stehen in einem Zeichenarray mit der
// Anfangsposition 'start' und der Längenangabe 'length'.

public void characters(char[] ch, int start, int length) throws SAXException;


// Nach Abarbeitung des ggw. XML-Tags wird die folgende Methode aufgerufen. Da XML-Tags
// beliebig geschachtelt sein können, ist hiermit die Möglichkeit gegeben, den aktuellen
// Schachtelungsgrad zu bestimmen und zwischenzuspeichern.

public void endElement(   String         namespaceURI,
                          String         localName,
                          String         name) throws SAXException;


// Zwischen den Start- und Endtags können auch Leerzeichen und Tabulatoren enthalten sein,
// die von keinerlei Interesse für den Parsingprozess sind. I. d. R. muss diese Methode
// vom Entwickler nicht implementiert zu werden, da die SAX-Parser eine sinnvolle
// Standardmethode bereits enthalten.

public void ignorableWhitespaces(char[] ch, int start, int length) throws SAXException;


// In XML-Dokumenten können weitere Ausführungsinstruktionen enthalten sein.

public void processingInstruction(String target, String data) throws SAXException;


// Möglicherweise sind beim Parsen Tags entdeckt worden, die in der Document Type
// Description nicht aufgeführt sind und deswegen übersprungen wurden.

public void skippedEntity(String name) throws SAXException;
```

Programmbeispiel 70: ContentHandler-Schnittstelle SAX-Parser

Ein SAX-Parser kann das zu lesende Dokument validieren oder ohne Validierung bearbeiten. Sobald der Parser auf Ungereimtheiten zwischen der Document Type Description und dem Dokument stößt, kann aufgrund des Fehlergrades eine Methode des Interfaces *ErrorHandler* aufgerufen werden. Wenn die Standardimplementierung dieser Methoden nicht ausreichend ist, so ist es die Aufgabe eines Entwicklers, diese Methoden entsprechend zu implementieren und anhand des vorliegenden Fehlers den Parsingprozess abzubrechen oder den Lauf weiter zu gestatten.

```
public interface ErrorHandler {
    // Bei der Implementierung der folgenden Methode wird man i. d. R. keinen Abbruch des
    // Parsing-Vorganges vornehmen, sondern lediglich eine Warninformation ausgeben. Dies ist
    // beispielsweise der Fall, wenn ein Webarchiv eine ältere Version aufweist. I. d. R. wird
    // die Verarbeitung nicht abgebrochen.

    public void warning(SAXParserException exception) throws SAXException;


    // Die folgende Methode weist auf einen Fehler des Dokumentes hin. Beispielsweise wurden
    // in der korrspondierenden DTD oder dem XML Schema für ein Attribut die Werte 'false'
    // oder 'true' mit dem Standardwert 'true' festgelegt und in dem Dokument werden
    // stattdessen die Werte 'yes' und 'no' verwendet. Die Verarbeitung wird i. d. R. nicht
    // abgebrochen und der Standardwert 'true' wird verwendet.

    public void error(SAXParserException exception) throws SAXException;


    // Die folgende Methode ist geeignet für Fehlersituationen, die nicht aufgelöst werden
    // können. I. d. R. wird nach Beendigung dieser Methode die Verarbeitung abgebrochen.

    void void fatalError(SAXParserException exception) throws SAXException;
```

Programmbeispiel 71: Schnittstelle SAX-Parser

12.6.2 DOM-Parser

Ein DOM-Parser liest nach seiner Instanziierung das gesamte Dokument ein und transformiert es in eine Baumstruktur. Es ist die Aufgabe eines Entwicklers, anschließend den Baum zu traversieren, bzw. über zur Verfügung gestellte Methoden nach den jeweiligen Daten gezielt zu fragen. Dazu sind die Schnittstellen *Node*, *NamedNodeMap*, *NodeList*, *CharacterData*, *Text*, *Document*, *Element* und *Attr* vom DOM-Parser bereits implementiert worden.

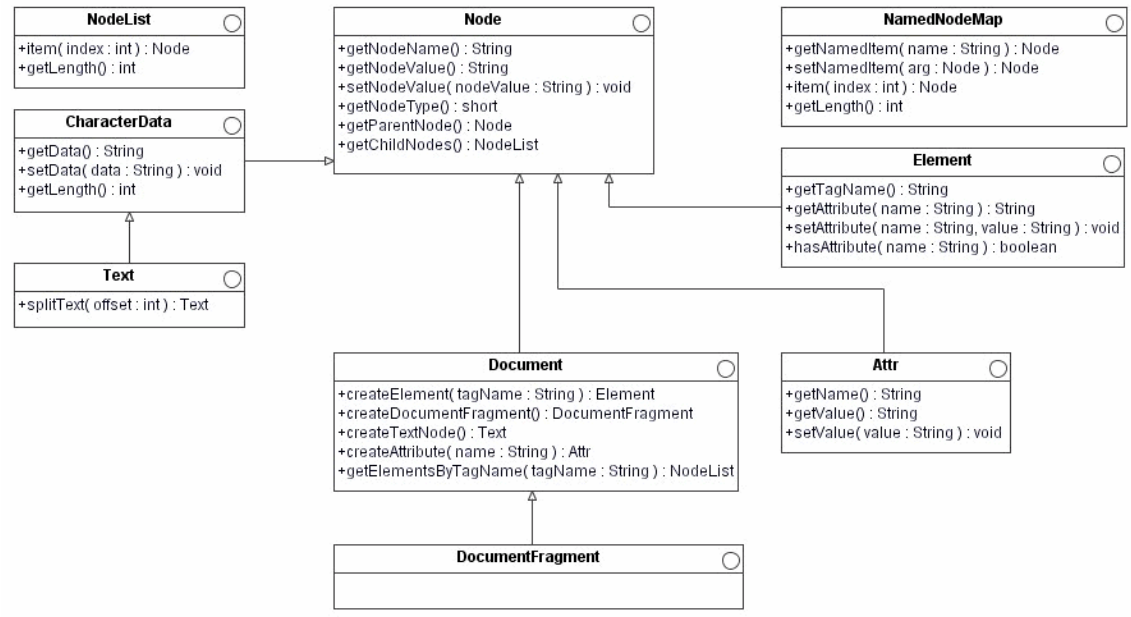


Abbildung 22: Die Schnittstellen des DOM-Parsers

13 Einführung Enterprise JavaBeans

Sun Microsystems hat mit der Spezifikation bzgl. der Enterprise JavaBeans, die ggw. in der Version 2.1 vorliegt, eine Standardisierung für die Entwicklung und die Verteilung von serverseitigen Bausteinen/Komponenten erzielt. Eine sehr gute und detaillierte Einführung in dieses Thema findet man unter [Monson-Haefel, 2003].

Die Spezifikation beinhaltet die serverseitigen Komponenten SessionBeans, EntityBeans und Message Driven Beans, wobei die SessionBeans unterteilt werden in zustandslose und zustandsbehaftete Beans. Mit der Spezifikation soll erreicht werden, dass ein Entwickler sich primär mit der Implementierung seiner Geschäftslogik beschäftigt und die notwendigen Dienste wie z. B. Transaktionen, die Frage der Persistenz oder Sicherheit, also die Authentifizierung und Autorisierung, durch zusätzlich generierte Softwarebausteine übernommen werden können.

Für alle Enterprise JavaBeans gilt:
1. Für jedes Bean existiert eine von der Spezifikation vorgegebene Schnittstelle, die von dem Bean zu implementieren ist. Für zustandsbehaftete und zustandslose SessionBeans ist es die Schnittstelle SessionBean, für EntityBeans das Interface EntityBean und für Message Driven Beans die Schnittstelle MessageDrivenBean.
2. Für jedes Bean gibt es zwei oder drei Beschreibungsdateien, in denen das Verhalten der Beans, Maßnahmen zur Steigerung der Verarbeitungsgeschwindigkeit, Sicherheit im Sinne von Autorisierung, automatische Transaktionsverarbeitung, automatische Persistenz bei speziellen EntityBeans etc. extern beschrieben werden.
3. Jedes Bean muss unter Angabe der Beschreibungsdateien durch einen speziellen EJB-Compiler übersetzt werden.

Für SessionBeans und EntityBeans gilt grundsätzlich das Fabrikmuster. Daraus folgt, das jedes Session- und jedes EntityBean zwei Schnittstellen bekannt geben muss:
1. Das HomeInterface, in dem Vereinbarungen darüber getroffen werden, wie serverseitig ein Bean erzeugt wird.
2. Das RemoteInterface, das bekannt gibt, welche entfernten Methoden das Bean implementiert hat.

Dazu gelten die folgenden Regeln:
1. Das HomeInterface wird von der Basisklasse EJBHome abgeleitet.
2. Das HomeInterface für SessionBeans enthält mindestens eine Methode
   ```
   RemoteInterface create() throws CreateException, RemoteException;
   ```
   die eine CreateException und eine RemoteExeption auslösen können. Die Rückgabe ist der Stub des RemoteInterfaces, das diese generierte Klasse repräsentiert.
3. Das HomeInterface für EntityBeans enthält mindestens die Methode
   ```
   RemoteInterface findByPrimaryKey(<Instanz des primären Schlüssels>)
          throws FinderException, RemoteException;
   ```
4. Alle Eingabedaten und Rückgabewerte müssen serialisierbar sein, d. h. ist der Rückgabewert eine Instanz einer Klasse Person, so muss die Klasse Person explizit die Schnittstelle java.io.Serializable implementieren und alle Datenmember, die nicht der Serialisierung unterliegen, sollten als *transient* gekennzeichnet werden. Die Notwendigkeit der Serialisierung ist auf zwei Gründe zurückzuführen:
   a. Bei der Kommunikation mit Enterprise JavaBeans kann der Entwickler entscheiden, welches Protokoll verwendet wird. Zur Auswahl stehen IIOP und JRMP (**J**ava **R**emote **M**ethod **P**rotocol). Bei Verwendung von JRMP ist lediglich eine Kommunikation zwischen Java-Komponenten möglich, aber der Aufwand des Marshallings/Unmarshallings entfällt. In Programmbeispiel 77 wird durch die Methode *narrow* explizit auf IIOP umgeschaltet. Hätte man statt *narrow* lediglich ein Casting auf den richtigen Typ gewählt, so wäre das Protokoll JRMP bei der Kommunikation verwendet worden.

b. Bei zustandsbehafteten SessionBeans (vgl. Kapitel 13.5.2) kann der EJB-Kontainer den Zustand eines Beans passivieren, d. h. den Zustand im Dateisystem ablegen und zu einem späteren Zeitpunkt wieder laden.

Mit Ausnahme der Message Driven Beans verfügt jedes Bean über ein RemoteInterface, in dem die zur Verfügung gestellten Methoden beschrieben werden. Das RemoteInterface entspricht somit der CORBA-IDL Beschreibung über den Zugang zu den entfernten Objekten und der zu übertragenden Daten. Auch hier gelten die Regeln
1. Jede Methode kann eine *RemoteException* und benutzerdefinierte Ausnahmen auslösen.
2. Alle Eingabedaten und alle Rückgabewerte sind serialisierbar.

Für den Fall, dass Beans untereinander kommunizieren und in der gleichen virtuellen Maschine instanziiert wurden existiert, in der Spezifikation 2.1 die Möglichkeit, mit lokalen Schnittstellen zu arbeiten. Die Unterschiede zu den entfernten Komponenten sind:
1. In dem HomeInterface wird die Ableitung von *EJBHome* ersetzt durch *EJBLocalHome*.
2. In dem RemoteInteface wird die Ableitung von *EJBObject* ersetzt durch *EJBLocalObject*.
3. Die Regeln bzgl. der Serialisierbarkeit der Rückgabewerte und der Eingabedaten kann optional entfallen.
4. Der notwendige JNDI-Name wird ersetzt durch einen lokalen JNDI-Namen.

Mit den lokalen Schnittstellen soll die Möglichkeit gegeben werden, dass zwei Komponenten im gleichen Adressraum den Remote-Methode-Mechanismus umgehen können und stattdessen über den gemeinsamen Adressraum kommunizieren. Es ist festzuhalten, dass jedes Session- und jedes EntityBean grundsätzlich einen lokalen und einen entfernten Zugang zur Verfügung stellen können.

Die einzelnen Beans und ihre Beschreibungsdateien werden in nachfolgenden Kapiteln im Detail erörtert.

In Anlehnung an das CORBA-Einführungsbeispiel wird ein vollständiges zustandsloses SessionBean aufgelistet, das dem Aufrufer die Zeit in Millisekunden oder als Zeichenkette liefert.

13.1 Beispiel eines zustandslosen SessionBeans

Das folgende Beispiel zeigt die benötigten Quellkodedateien und die Beschreibungsdateien, die für einen zustandslosen Zeitgeber notwendig sind. Entwickelt wurde dieses Beispiel unter BEA's Weblogic Server 6.1.

Im Folgenden wird der entfernte Zugang, also die Geschäftsmethoden des zustandslosen Beans, beschrieben.

```
// TimeServer.java: RemoteInterface fuer das stateless SessionBean TimeServerSLBean.

package de.consavis.time;

import java.rmi.*;
import javax.ejb.*;

public interface TimeServer extends EJBObject {
    public long       getTime() throws RemoteException;
    public String     getDate() throws RemoteException;
}
```

Programmbeispiel 72: Remote-Interface TimeServer

Durch die EJB-Spezifikation ist vorgegeben, dass SessionBeans und EntityBeans nach dem Fabrikmuster arbeiten. Die folgende Schnittstelle beschreibt, wie ein Klient serverseitig ein Bean erzeugen kann.

```
// TimeServerHome.java: HomeInterface fuer das stateless SessionBean TimeServerSLBean

package de.consavis.time;

import java.rmi.*;
import javax.ejb.*;

public interface TimeServerHome extends EJBHome {
    public TimeServer create() throws CreateException, RemoteException;
}
```

Programmbeispiel 73: Home-Interface des TimeServer

Die Implementierungsdatei muss die folgenden Methoden definieren:

1. Alle Methoden aus dem SessionBean-Interface (*ejbRemove*, *ejbPassivate*, *ejbActivate*, *setSessionContext*).
2. Alle Methoden aus dem RemoteInterface (*getTime* und *getDate*).
3. Die zu der Methode *create()* korrespondierende Methode *ejbCreate()*, die von der generierten Software aufgerufen wird.

```
// TimeServerSLBean.java: Implementierung von SessionBean- und RemoteInterface-Methoden

package de.consavis.time;
import javax.ejb.*;
import java.util.Date;

public class TimeServerSLBean implements SessionBean {

    // Implementierung der Methoden getDate und getTime des RemoteInterfaces

    public long getTime()
    {
        return System.currentTimeMillis();
    }

    public String getDate()
    {
        return (new Date().toString());
    }


    // Implementierung der zu create() aus dem HomeInterface korrespondierenden
    // ejbCreate-Methode

    public void ejbCreate()
    {
        System.out.println("\nejbCreate() called");
    }


    // Implementierung der Schnittstelle SessionBean

    public void ejbRemove()
    {
        System.out.println("\nejbRemove() called");
    }

    public void ejbActivate()
    {
        System.out.println("\nejbActivate() called");
    }

    public void ejbPassivate()
    {
        System.out.println("\nejbPassivate() called");
    }
```

```
    public void setSessionContext(SessionContext ctx)
    {
        this.ctx = ctx;
        System.out.println("\nsetSessionContext() called");
    }

    private SessionContext ctx;
}
```

Programmbeispiel 74: Implementierung des TimeServer

Der EJB-Spezifikation folgend muss für jedes Bean eine Datei *ejb-jar.xml* hinterlegt werden. Die zweite Datei *weblogic-ejb-jar.xml* ist proprietär und muss bei Verwendung anderer Applikationsserver geeignet umbenannt und modifiziert werden.

In der Datei *ejb-jar.xml* wird ein Bean allgemein beschrieben. Jedes Bean hat einen symbolischen Namen, der durch *<ejb-name>* festgelegt wird. Das nachfolgende Tag (hier *<session>* gibt an, dass es sich um ein SessionBean handelt. Mit den Tags *<home>*, *<remote>* und *<ejb-class>* werden in Abhängigkeit der Paketvereinbarungen in den Java-Quellkodes das Home-, das RemoteInterface und die Implementierungsklasse vollqualifiziert angegeben. Das Tag *<session-type>* legt fest, ob es sich um ein zustandsloses SessionBean (Stateless) oder um ein zustandsbehaftetes SessionBean (Stateful) handelt. Die letzte Taganweisung *<transaction-type>* legt fest, das eine eventuelle Transaktionsverarbeitung durch die generierte Software, dem Container, zu leisten ist.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
        <enterprise-beans>
        <session>
                <ejb-name>TimeEJB</ejb-name>
                <home>com.bea.time.TimeServerHome</home>
                <remote>com.bea.time.TimeServer</remote>
                <ejb-class>com.bea.time.TimeServerSLBean</ejb-class>
                <session-type>Stateless</session-type>
                <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Programmbeispiel 75: Deployment Deskriptor ejb-jar.xml

Die Beschreibungsdatei *weblogic-ejb-jar.xml* beschreibt für den Weblogic Server u. U. Maßnahmen wie z. B. das Einrichten eines Pools von SessionBean-Instanzen. In dem vorliegenden Beispiel wird in dieser Datei lediglich festgelegt, unter welchem Namen der Stub des HomeInterfaces in den JNDI-Baum eingebunden wird. Der Klient benötigt genau diesen Namen, um diese Information zu erfragen.

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN'
'http://www.bea.com/servers/wls60/ejb20/dtd/weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
                <ejb-name>TimeEJB</ejb-name>
                <jndi-name>de.consavis.time.TimeServerHome</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Programmbeispiel 76: Deployment Deskriptor weblogic-ejb-jar.xml

Das Deployment beinhaltet die korrekte Kompilation, das korrekte Zusammenfassen der Java-Quellkodedateien und der Deployment Deskriptoren, die Generierung der Zusatzsoftware

durch einen speziellen Compiler und die Bekanntgabe der Komponente gegenüber dem Applikationsserver.

Es wird vorausgesetzt, dass die beiden XML-Beschreibungsdateien in einem Subverzeichnis META-INF liegen.

Der Deploymentvorgang ergibt sich dann aus:
1. javac -d . *.java
2. jar -cvf x.jar de META-INF
3. rm -rf de
4. java weblogic.ejbc20 x.jar time.jar
5. cp time.jar $APPLICATION

Im Folgenden wird eine JavaServer Page angegeben, die den Klientenzugang zu dem zustandslosen SessionBean beschreibt.

```
<%@ page import="javax.naming.*,java.rmi.*,javax.rmi.*,de.consavis.time.*"
         session="true" %>
<!-- rein deklarativer Teil. Die aufgeführten Datenmember werden in dem auszuführenden
     Java-Kode benötigt. Die Instanziierung der Klasse InitialContext() bewirkt die Ver-
     bindung des Klienten zum Wurzelobjekt des JNDI-Baumes -->

<%!
        private static Context ctx;
        private TimeServerHome home;
        private TimeServer ts;

        static {
                try {
                        ctx = new InitialContext();
                }
                catch (Exception e) {
                        System.out.println(e);
                        System.out.println("Verbindung zum JNDI fehlgeschlagen");
                }
        }
%>

<HTML>
    <HEAD><TITLE>TimeServer</TITLE></HEAD>
    <BODY>
        <H1>Zeitgeber</H1>
        <HR>

        <!-- zwischen den Tags <% und %> werden Java-Anweisungen angegeben,
             deren Ausgaben/Resultate zur Laufzeit in die HTML-Seite eingefügt werden -->
    <%
        try {
                // Stub des Home-Interfaces abfragen, an den richtigen Typ anpassen.
                // Durch die Verwendung von narrow wird auf das Protokoll IIOP umgeschaltet.
                // Durch ein reines Casting, also
                //      home = (de.consavis.time.TimeServerHome)(
                //                              ctx.lookup("de.consavis.time.TimeServerHome"));
                // wird das Protokoll JRMP ausgewählt.

                home = (de.consavis.time.TimeServerHome)PortableRemoteObject.narrow(
                                              ctx.lookup("de.consavis.time.TimeServerHome"),
                                                        de.consavis.time.TimeServerHome.class);


                // Bean auf der Serverseite erzeugen lassen. Die Methode create liefert als
                // Rückgabe den Stub des Remote-Interfaces

                ts = home.create();
        }
        catch (Exception e) {
                System.err.println("Fehler beim Erzeugen des Beans");
        }
    %>
    <B>Zeit in Millisekunden:</B>
```

```
    <!-- Ergebnisse die zwischen den Tags <%= und %> stehen, werden in einen
         String konvertiert und an dieser Stelle in das HTML-Dokument eingefügt.
         'ts' repräsentiert die entfernte Komponente und durch ts.getTime()
         bzw. ts.getDate() werden die entfernten Methoden getTime und getDate aufgerufen -->

        <%=ts.getTime()%>
        <BR>
        <B>Zeit und Datum:</B>
        <%=ts.getDate()%>
    </BODY>
</HTML>
```

Programmbeispiel 77: Klient für den TimeServer

13.2 Die Beschreibungsdateien für Enterprise JavaBeans

Für alle Enterprise JavaBeans gilt, dass die zu generierende Software in Deployment Deskriptoren beschrieben werden. Dies beinhaltet Festlegungen über automatische Transaktionen, Sicherheit im Sinne von Autorisierung, automatische Persistenz bei CMP-EntityBeans usw. Für SessionBeans, Message Driven Beans und Bean Managed EntityBeans sind dies beim Weblogic Server von BEA Systems die beiden Beschreibungsdateien *ejb-jar.xml* und *weblogic-ejb-jar.xml*. Bei CMP-EntityBeans kommt eine weitere Beschreibungsdatei hinzu.

Die Datei *ejb-jar.xml* ist standardisiert und dient der allgemeingültigen Beschreibung eines Beans. In der Datei *weblogic-ejb-jar.xml* wird für jedes Bean zumindest der JNDI-Name hinterlegt, damit der Stub des HomeInterfaces im JNDI-Baum unter diesen Namen gebunden werden kann. Andere Spezialitäten des Weblogic Servers, wie z. B. Maßnahmen zur Steigerung der Verarbeitungsgeschwindigkeit, können beschrieben werden (für andere Applikationsserver existieren entsprechende proprietäre Dateien. Z. B. verwendet JBoss die Datei jboss.xml für die eigenen Spezialitäten).

Alle Beschreibungsdateien sind bzgl. ihres syntaktischen Aufbaus durch eine Document Type Description fest vorgegeben. Die Namensgebung der beiden o. a. Dateien ist fest und unveränderlich. Die zusätzliche Beschreibungsdatei für CMP-Beans kann einen beliebigen Namen besitzen.

In der folgenden Abbildung ist das Zusammenspiel zwischen der Datei *ejb-jar.xml* mit der Datei *weblogic-ejb-jar.xml* illustriert.

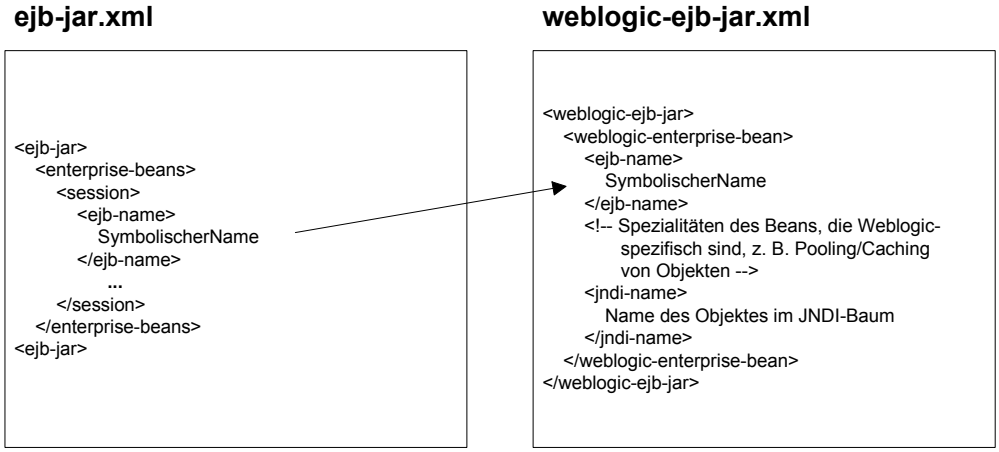


Abbildung 23: Grobaufbau und Zusammenspiel der Deployment Deskriptoren

Zu den Beispielen der jeweiligen Beans werden vollständige Deployment Deskriptoren angegeben.

Bei der Kompilation von Enterprise JavaBeans und ihrer Verpackung ist besondere Vorsicht angebracht. Der EJB-Compiler erwartet alle Informationen an definierter Stelle.

Unter der folgenden Hypothese wird dieser Vorgang beschrieben:
1. Alle Java-Sourcen, also das Home-, RemoteInterface und die Implementierungsklasse befindet sich in einem Verzeichnis.
2. Die Java-Quellkodes besitzen die Paketangabe *de.uni-mainz*.
3. Dieses Verzeichnis enthält ein Unterverzeichnis mit dem fest vorgegebenen Namen META-INF.
4. Die Deployment Deskriptoren befinden sich in dem Verzeichnis META-INF.

In der folgenden Abbildung ist dies illustriert:

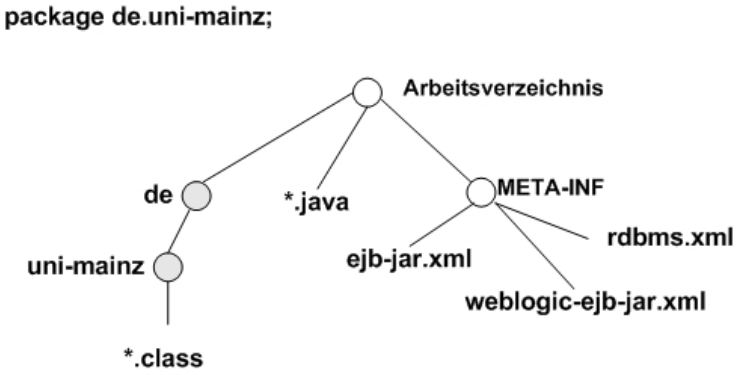


Abbildung 24: Aufbau des Arbeitsverzeichnisses für EJBs

Die Arbeitsschritte, um ein Enterprise Java Bean korrekt zu übersetzen, sind:
1. cd $Arbeitsverzeichnis
   wechseln in das Arbeitsverzeichnis. Es wird hierbei unterstellt, dass es eine Umgebungsvariable *Arbeitsverzeichnis* gibt, die als Wert das tatsächliche Verzeichnis enthält.
2. javac -d . *.java
   durch die Option -d werden, falls nicht vorhanden, die benötigten Verzeichnisse der Paketangabe in dem ggw. Verzeichnis angelegt. Die übersetzten Java-Klassen werden aufgrund der Paketangabe und der -d-Option in dem Verzeichnis *de/uni/mainz/* abgelegt (s. Abbildung).
3. jar -cvf x.jar META-INF de
   die Inhalte der Verzeichnisse META-INF und de (s. Abbildung) werden in ein Java-Archiv mit dem Namen x.jar eingebunden.
4. java weblogic.ejbc20 x.jar bean.jar
   der EJB-Compiler wird bei dem Weblogic Server als eine Java-Applikation, die in dem Namensraum *weblogic* liegt, mit ausgeliefert. Der Compiler
   a. entpackt die übergebene Datei x.jar.
   b. interpretiert die Deployment Dekriptoren.
   c. generiert basierend auf den dort getroffenen Vereinbarungen Java-Quellkode.
   d. generiert die für die Kommunikation erforderlichen Stubs und Skeletons.
   e. für CMP-EntityBeans wird ein Persistenz Manager erzeugt, der die Persistenzfrage eigenständig abarbeitet.
   f. ruft für die erzeugten Quellkodes den javac-Compiler auf.
   g. verpackt alle Informationen in einem neuen Java-Archiv.
5. mv bean.jar $APPLIKATION

der Weblogic Server verfügt in jeder Domäne über ein spezielles Verzeichnis mit dem fest vorgegebenen Namen *applications*. Die Bedeutung dieses Verzeichnisses ist, dass alle Archive, die in dieses Verzeichnis kopiert werden, automatisch vom Weblogic Server eingelesen und ausgewertet werden.

13.3 Der Zugang eines Klienten zu Enterprise JavaBeans

Ein Klient kann mit einem Session- oder EntityBean nur arbeiten, wenn er den Stub des Remote-Interfaces besitzt. Der übliche Weg führt über den Namensdienst JNDI, dem **J**ava **N**aming and **D**irectory **I**nterface, der wie üblich als Baumstruktur zu interpretieren ist.

Der Zugang vom Klienten zu dem JNDI kann bei den einzelnen Produkten unterschiedlich sein. Der allgemeine Vorgang für entfernte Klienten ist:

1. Aufbau eines Hashtable- oder Properties-Objektes mit den Angaben
   a. Mit welchem Namensdienst soll gearbeitet werden?
   b. Die Lokation des Namensdienstes in Form einer URL (**U**niform **R**essource **L**ocator).
   c. Optional die Angabe über den Benutzer, der sich verbinden will.
   d. Optional die Angabe über das Benutzerkennwort
2. Instanziierung eines Objektes vom Typ InitialContext mit Übergabe des o. a. Properties- oder Hashtable-Objektes. Nach erfolgreicher Verbindung ist der Klient mit der Wurzel des Namensdienstes verbunden.
3. Abfrage der gewünschten Informationen über die Methode

```
Object lookup(String name);
```

JNDI ist eine standardisierte Schnittstelle und lässt unberücksichtigt, welches Produkt integriert wurde, das als Namens- und Verzeichnisdienst dient. In der nachfolgenden Abbildung wird dies illustriert:

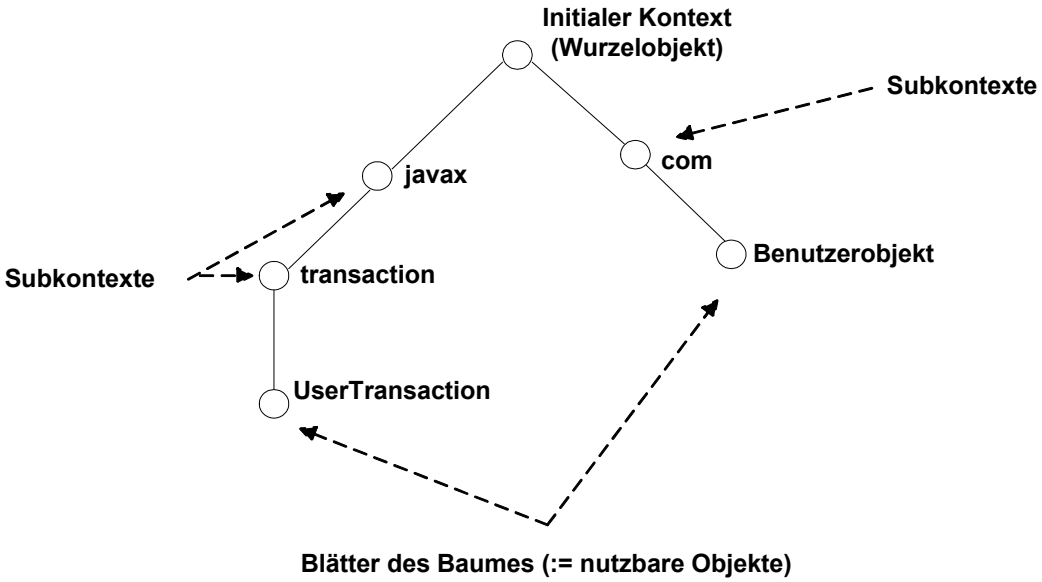


Abbildung 25: Darstellung eines JNDI-Baumes

Die Informationen, die in das JNDI eingestellt werden, sind primär

1. Der Stub der jeweiligen HomeInterfaces.
2. Datasources, die einen JDBC-Connectionpool repräsentieren.
3. Umgebungsvariablen.

4. Beliebige serialisierbare Objekte.

etc.

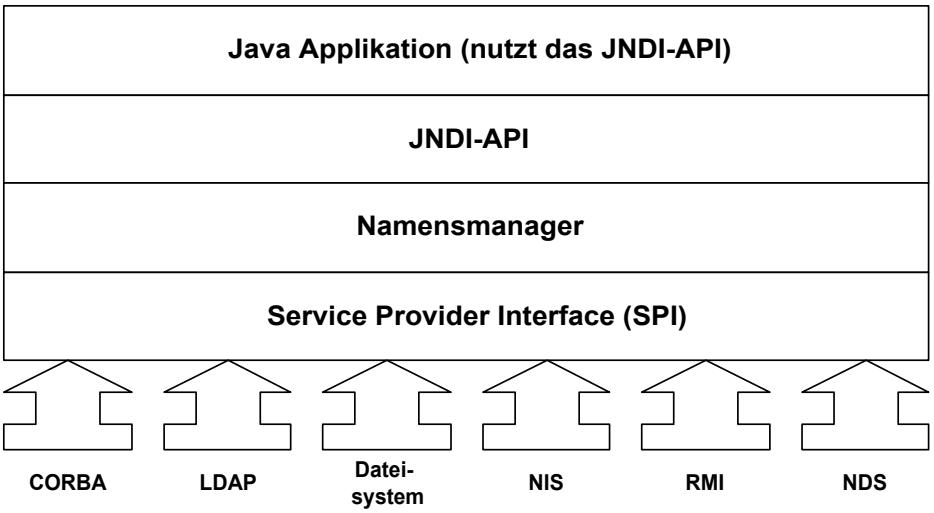


Abbildung 26: Schematische Darstellung der Arbeitsweise von JNDI

Die wichtigsten Methoden der JNDI-Schnittstelle sind:

1. *Object lookup(String name)*: Ein Klient erfragt über die Angabe Subkontext und Name des Baumblattes eine Information. Beispielsweise wird das *UserTransaction*-Object abgefragt durch

   ```
   Object o = ctx.lookup("javax.transaction.UserTransaction");
   ```

   Grundsätzlich muss ein Klient das erhaltene Objekt vom Typ java.lang.Object an den richtigen Typ anpassen. Dazu entscheidet der Klient darüber, ob die erhaltene Information ein Objekt der entfernten Verarbeitung, wie z. B. der Stub eines Home-Interfaces ist, oder nicht. Im ersten Fall muss er sich entscheiden, ob er über das Protokoll RMI-over-IIOP (**R**emote **M**ethod **I**nvocation over **I**nternet **I**nterorb **P**rotocol) oder über das Protokoll RMI-JRMP (**R**emote **M**ethod **I**nvocation over **J**ava **R**emote **M**ethod **P**rotocol) kommunizieren will. Im ersten Fall kommuniziert man über das CORBA-Protokoll IIOP mit dem bereits dargestellten Überhang, aber auch mit der Möglichkeit, Kontakt mit Objekten/Komponenten aufzunehmen, die nicht in Java entwickelt wurden. Im zweiten Fall kann lediglich mit anderen, in Java entwickelten Objekten/Komponenten, kommuniziert werden.
2. *void bind(Object object, String name)*: Das Objekt *object* wird unter dem angegebenen Namen in den JNDI-Baum als serialisiertes Objekt gebunden. Der Name setzt sich hierbei zusammen aus den Subkontexten, gefolgt von dem eigentlichen Namen, unter dem das Objekt gebunden wird. Grundsätzlich ist bei dieser Operation zweierlei zu berücksichtigen:
   a. Das Objekt darf in diesem Namensraum noch nicht unter dem angegebenen Namen gebunden sein.
   b. Der Namensraum, also der Subkontext, muss bereits existieren.
3. *void rebind(Object object, String name)*: Das Object *object* wird unter dem angegebenen Namen neu gebunden. Dies ist beispielsweise sinnvoll, wenn das Objekt unter Beibehaltung des Namens durch eine neue Version ersetzt wird.
4. *void unbind(String name)*: Das durch *name* referenzierte Objekt wird vollständig aus dem JNDI-Baum entfernt. Der Namensraum bleibt erhalten!
5. *NamingEnumeration list(String kontext)*: Mit dieser Operation wird der gesamte Baum an den Aufrufer als Aufzählungstyp zurückgeliefert.
6. *NamingEnumeration listBinding(String kontext)*: Analog zu der Methode *list* wird der gesamte Baum inklusive aller serialisierten Objekte an den Aufrufer zurückgeliefert.

7. *void destroy_subcontext(String kontext)*: Der benannte Subkontext wird unter der Voraussetzung, dass keine weiteren Blätter des Baumes existieren, aus dem JNDI-Baum entfernt.
8. *create_subcontext(String kontext)*: Ein neuer Subkontext wird unter der Voraussetzung, dass er noch nicht existiert, erzeugt.

13.4 Beispiel: Traversierung eines JNDI-Baumes

Die folgende Applikation illustriert den Umgang mit dem Namensverzeichnis JNDI, in dem über den gesamten Baum traversiert wird und alle Blätter des Baumes zur Ausgabe gebracht werden.

```
// ListJNDI.java: Java-Applikation zum Traversieren eines JNDI-Baumes. Der Import auf die
// Pakete javax.naming beinhaltet alle benötigten Klassen für den Umgang mit dem Namensdienst

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.Binding;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import java.util.Hashtable;

public class ListJNDI {
    public static void main(String[] args)
    {
        // Kommandozeilenargumente ueberpruefen

        String name;
        if (args.length < 1) {
               name = "";
        }
        else {
               name = args[0];
        }
        Hashtable  env = new Hashtable();

        // Aufbau von benoetigten Eigenschaften in einer Hashtable fuer den Zugang
        // zum JNDI. Alternativ kann auch mit java.utils.Properties gearbeitet werden
        // Eigenschaften:
        // 1. Mit welcher initialen Kontextfabrik, also JNDI-Implementierung soll ge-
        //    arbeitet werden.
        // 2. URL des Rechners, zu dessen Namensdienst der Klient sich verbinden will.
        // 3. Identität des Klienten. In diesem Beispiel der Benutzer 'system', der bzgl.
        //    seiner Berechtigung UNIX-root entspricht.
        // 4. Das Kennwort des Benutzers.

        env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "http://localhost:7001");
        env.put(Context.SECURITY_PRINCIPAL, "system");
        env.put(Context.SECURITY_CREDENTIALS, "weblogic");

        try {
               Context initialContext = new InitialContext(env);
               printContext(name,initialContext, 0);
        }
        catch (NamingException e) {
               System.out.println(e);
               System.exit(1);
        }
    }

    public static void printContext(String name, Context dir, int numLevelsDeep)
    {
        Binding               item          = null;
        NamingEnumeration     enumerator    = null;
        try {
               // Falls kein Subkontext angegeben wurde, beginnt der Durchlauf
               // bei der Wurzel des Baumes

               if (name != null)
```

```
                enumerator = dir.listBindings(name);
            else
                enumerator = dir.listBindings("");

            // Solange iterieren, bis das Ende des Baumes erreicht wurde

            while (enumerator.hasMore()) {
                try {
                    // das naechste aktuelle Objekt merken und verarbeiten

                    item = (Binding) enumerator.next();

                    // Für jeden Subkontext einmal einruecken um die
                    // Hierarchie visuell darzustellen

                    for (int i = 0; i < numLevelsDeep; i++)
                        System.out.print("    ");

                    // war das Objekt ein Blatt des Baumes, oder muss weiter
                    // iteriert werden?

                    if (item.getObject() instanceof Context) {
                        Context subContext = (Context) item.getObject();
                        System.out.println(item.getName()+"/");
                        printContext(null, subContext, numLevelsDeep + 1);
                    }
                    else {
                        // Namen des Objektes ausgeben

                        Object  ab = (Object) item.getObject();
                        System.out.println(item.getName());
                    }
                }
                catch (NamingException e) {
                    System.out.println("Inner Exception" + e.getMessage());
                }
            }
        }
        catch (NamingException e) {
            System.out.println("Outer Exception" + e.getMessage());
        }
    }
}
```

Programmbeispiel 78: JNDI-Traversierung

13.5 SessionBeans

SessionBeans unterteilen sich in zustandslose und zustandsbehaftete Beans. Die Unterschiede werden von dem Entwickler nicht programmtechnisch behandelt, sondern durch die externen Beschreibungsdateien festgelegt.

13.5.1 Zustandslose SessionBeans

Definition 41: Zustandsloses (stateless) SessionBean

Ein *zustandsloses SessionBean* ist eine gedächtnislose Komponente, die einen oder mehrere Dienste zur Verfügung stellt.

Es werden für den Klienten keine Informationen bzgl. eines bestimmten Zustandes gehalten. Ein zustandsloses Bean entspricht somit weitestgehend einem CORBA-Objekt; der Unterschied ist hier lediglich in der o. a. generierten Zusatzsoftware zu sehen. Mögliche Beispiele von zustandslosen SessionBeans sind Zugriffe auf einen Namensdienst oder mathematische Berechnungen.

13.5.2 Zustandsbehaftete SessionBeans

Definition 42: Zustandsbehaftete (stateful) SessionBeans
Ein *zustandsbehaftetes SessionBean* ist eine Softwarekomponente, die sich im Auftrag des Klienten Informationen über den ggw. Zustand merkt.

Damit verbunden sind zwei unmittelbare Folgerungen:
1. Das Bean gehört exklusiv genau einem Klienten und kann im Normalfall von anderen Klienten nicht genutzt werden.
2. Das in der Literatur bekannte Prinzip der Konversation ist gegeben. D. h., der Klient kann sich mit nachfolgenden Methodenaufrufen auf den zuletzt gemerkten Zustand beziehen.

Ein mögliches Anwendungsbeispiel für ein zustandsbehaftetes SessionBean können die Einkaufswagen bei Internet-Applikationen sein. Der Kunde füllt sukzessive seinen Einkaufswagen und das Bean verwaltet diesen Zustand.

Die Exklusivität des zustandsbehafteten SessionBeans kann nur durch die folgende Vorgehensweise unterlaufen werden:
1. Ein Klient erzeugt über das HomeInterface ein zustandsbehaftetes SessionBean und schreibt den erhaltenen Stub serialisiert in einen persistenten Speicher.
2. Ein zweiter Klient liest diesen Stub aus, deserialisiert ihn und verfügt somit über den gleichen Stub wie der erste Klient.

Von Seiten des EJB-Containers kann eine Unterscheidung bzgl. der Klienten nicht mehr vorgenommen werden, d. h. beide Klienten arbeiten serverseitig mit der gleichen Instanz.

13.6 EntityBeans

Ein EntityBean ist eine Softwarekomponente, die per Definition für das Arbeiten mit Datenbanken konzipiert wurde.

Die Aufgabe, Daten persistent in einem Speicher abzulegen, kann auch durch andere Softwarekomponenten durch die Programmierung von JDBC-Zugriffen vorgenommen werden. Werden gleiche Datenbankzugriffe von vielen verschieden Klienten ausgeführt, so kann es sinnvoll sein, den benötigten JDBC-Kode als einen zentralen Dienst in Form eines EntityBeans anzubieten, statt alle anderen Komponenten mit dem jeweiligen Kode zu versehen.

Definition 43: EntityBean
EntityBeans sind die Repräsentanten persistenter Daten.

Damit verbunden ist:
1. Jedes EntityBean verfügt über einen primären Schlüssel.
2. Das HomeInterface eines jeden EntityBeans muss die Methode
   ```
   <RemoteInterface> findByPrimaryKey(<primäre Schlüsselklasse>)
   throws RemoteException, FinderException;
   ```
   deklarieren.
3. Weitere *find*-Methoden können vereinbart werden. Die Namenskonvention und vollständige Signatur für diese Methoden ist:
   a.
   ```
   <RemoteInterface> find<Methodenname>(<Parameterliste>)
           throws RemoteException, FinderException;
   ```
   falls die Anfrage eindeutig ist.
   b.
   ```
   Collection find<Methodenname>(<Parameterliste>)
           throws RemoteException, FinderException;
   ```
   falls die Anfrage mehrere Treffer aufweisen kann.
4. Wird die Persistenz automatisiert, so sind in den Beschreibungsdateien des Beans
   a. alle Datenmember anzugeben.

b. die primäre Schlüsselklasse muss angegeben werden.
c. welcher Datenmember für den Vergleich für die *findByPrimaryKey*-Methode verwendet wird.
d. welcher *DataSource* den *JDBC-Connectionpool* repräsentiert.
e. mit welcher Tabelle der Datenbank gearbeitet wird.
f. welche Java-Datenmember auf welche Datenbanktypen umgesetzt werden.

Die Spezifikation berücksichtigt zwei Arten von EntityBeans:
1. Die benötigte Programmlogik wird von dem Entwickler zur Verfügung gestellt.
2. Die benötigte Programmlogik wird durch externe Beschreibungsdateien definiert und ein spezieller Compiler generiert den Programmkode.

Im ersten Fall spricht man von **B**ean **M**anaged **P**ersistence (BMP) und im zweiten Fall von **C**ontainer **M**anaged **P**ersistence (CMP).

13.6.1 Bean Managed Persistence

Bei den BMP-EntityBeans muss der Entwickler den gesamten JDBC-Kode für die Zugriffe auf die Datenbank implementieren.

Wichtige Gründe für den Einsatz von BMP können sein:
1. Die gesamte Kontrolle liegt beim Entwickler und alle Möglichkeiten der JDBC-Programmierung können genutzt werden.
2. Die Fehlersuche kann ggf. in Java leichter sein, als in den XML-Beschreibungsdateien.
3. Der persistente Speicher kann nicht über einen JDBC- oder ODBC-Treiber eingebunden werden.

usw.

13.6.2 Beispiel: Bean Managed Persistence

Im Folgenden wird ein EntityBean implementiert, das in genau einer Zeile einer Datenbanktabelle Informationen über Hunde, deren Namen, Rasse und Geburtsdatum verwaltet.

Das EntityBean arbeitet nach dem Prinzip Bean Managed Persistence, d. h., dass alle Datenbankanfragen ausprogrammiert werden.

In der nachfolgenden Schnittstelle werden die Methoden der Geschäftslogik deklariert.

```
// Dog.java: RemoteInterface für das EntityBean 'DogEBean'

package de.uni.mainz.bmp;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface Dog extends EJBObject {
    public String     getID() throws RemoteException;
    public String     getName() throws RemoteException;
    public void       setName(String n) throws RemoteException;
    public String     getBreed() throws RemoteException;
    public Date       getBorn() throws RemoteException;
}
```

Programmbeispiel 79: RemoteInterface für das DogEBean

Der Klient kann also zu jedem in der Datenbank erfassten Hund den Identifizierer, Namen des Hundes, seine Rasse und sein Geburtsdatum abfragen. Die Methode *setName()* erlaubt eine Umbenennung des Hundes.

Die folgende Schnittstelle beschreibt, wie das EntityBean erzeugt werden kann. Mit der Methode *create* wird ein neuer Hund mit seinen Angaben in die Datenbank eingestellt. Der notwendige, eindeutig bestimmte primäre Schlüssel wird bei der Methodenimplementierung bestimmt.

Die von der Spezifikation vorgegebene Methode *findByPrimaryKey* liefert als Rückgabewert den Stub des o. a. RemoteInterfaces zurück.

Die Methode *findAllDogs* soll die Auflistung aller erfassten Hunde ermöglichen.

```
// DogHome.java: HomeInterface des EntityBeans 'DogEBean'

package de.uni.mainz.bmp;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface DogHome extends EJBHome {
    public Dog create(String name, Date born, String breed)
                throws CreateException, RemoteException;
    public Dog findByPrimaryKey(String dogID)
                throws FinderException, RemoteException;
    public Collection findAllDogs()
                throws FinderException, RemoteException;
}
```

Programmbeispiel 80: HomeInterface für DogEBean

Die Implementierung des EntityBeans erfolgt nach der von Sun Microsystems verabschiedeten Spezifikation. Demzufolge muss das Bean

1. die Schnittstelle EntityBean mit den Methoden *setEntityContext*, *unsetEntityContext*, *ejbRemove*, *ejbLoad*, *ejbStore*, *ejbActivate* und *ejbPassivate* implementieren.
2. anstelle der *create*-Methode eine *ejbCreate*-Methode mit exakt der gleichen Signatur implementieren.
3. eine Methode *ejbPostCreate* mit exakt der gleichen Signatur implementieren.
4. für jede im HomeInterface festgelegte *find*-Methode eine *ejbFind*-Methode mit exakt der gleichen Signatur implementieren.

Nicht benötigte Methoden können als leere Methoden definiert werden.

```
// DogEBean.java: Implementierung eines EntityBeans, das eine Menge von Hunden in der
//                Datenbank verwaltet

package de.uni.mainz.bmp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

public class DogBean implements EntityBean {

    // Implementierung der Geschäftsmethoden

    public String            getID() { return id; }
    public String            getName() { return name; }
    public void              setName(String n) { name = n; }
    public String            getBreed() { return breed; }
```

```
public java.util.Date     getBorn() { return born; }

// Defaultkonstruktor

public DogBean() {}


// Implementierung der Methoden der Schnittstelle EntityBean

public void setEntityContext(EntityContext ctx)
{
    // vor jeder Beanerzeugung wird durch den Container ein Kontext gesetzt,
    // der Informationen wie z. B. Benutzeridentität, Informationen zu Trans-
    // aktionen oder für EntityBeans den benötigten Primärschlüssel enthält.

    this.ctx = ctx;
}

public void unsetEntityContext()
{
    this.ctx = null;
}

public void ejbActivate()
{
    // wird hier nicht benötigt
}

public void ejbPassivate()
{
    // wird hier nicht benötigt
 }

// getConnection(): Hilfsmethode, zum Erhalt einer Datenbankverbindung aus einem
//                  Pool. Nach außen nicht sichtbar

private Connection getConnection()
{
    InitialContext ctx = null;
    DataSource     ds  = null;

    // Verbinden zum JNDI-Baum und unter Angabe des Aliasnamens den be-
    // nötigten DataSource abfragen. Der DataSource repräsentiert einen JDBC-
    // Connectionpool und erlaubt dem Entwickler, eine Verbindung zur Datenbank
    // zu bekommen, ohne Detailwissen über die Lokation, die Eigenschaften oder
    // den benötigten JDBC-Treiber der Datenbank zu besitzen.

    try {
           ctx = new InitialContext();
           ds  = (javax.sql.DataSource) ctx.lookup("de.uni.mainz.DogDataSource");
           return ds.getConnection();
    }
    catch(Exception e) {
           System.err.println("Kein Datasource vorrätig " + e);
    }
    return null;
}

// cleanup(): Hilfsmethode zur Freigabe von Ressourcen

private void cleanup(Connection con, PreparedStatement ps)
{
    try {
           if (ps != null)
                  ps.close();
    }
    catch (Exception e) {
           System.err.println(e.getMessage());
           throw new EJBException (e);
    }

    try {
           if (con != null)
                  con.close();
    }
    catch (Exception e) {
           System.err.println(e.getMessage());
           throw new EJBException (e);
```

```
    }
}

// getNextID(): Hilfsmethode zur Bestimmung des nächstmöglichen primären Schlüssels

private String getNextID()
{
    Connection                  conn = null;
    PreparedStatement           stmt = null;
    String                      id;

    try {
            conn = getConnection();
            stmt = conn.prepareStatement( "SELECT MAX(DOGID) + 1 AS ID FROM DOG_TABLE");
            stmt.executeQuery();
            ResultSet rs = stmt.getResultSet();
            if(!rs.next())
                    id = "1";
            else
                    id = rs.getString("ID");

    }
    catch (SQLException sqe) {
            System.err.println ("SQLException: " + sqe);
            throw new EJBException(sqe);
    }
    finally {
            cleanup(conn, stmt);
    }
    return id;
}

// Die von dem ejbc-Kompiler generierte Software unterbricht den Aufruf create und
// überführt ihn in einem ejbCreate-Aufruf.

public String ejbCreate(String          name,
                        java.util.Date  born,
                        String          breed)
    throws CreateException
{
    String              id;
    Connection          conn = null;
    PreparedStatement   stmt = null;

    // Plausibilitätsüberprüfung der Eingabedaten

    if ((name == null) || name.equals(""))
            throw new CreateException("Der Hund besitzt keinen Namen!");
    if (born == null)
            throw new CreateException("Kein Geburtsdatum angegeben!");
    if ((breed == null) || breed.equals(""))
            throw new CreateException("Keine Rasse angegeben!");

    try {
            // ein create-Aufruf eines Klienten bedingt immer einen neuen Eintrag
            // im persistenten Speicher

            String sql     = "INSERT INTO DOG_TABLE VALUES(?,?,?,?)";
            conn           = getConnection();
            stmt           = conn.prepareStatement(sql);

            // primären Schlüssel bestimmen, Daten in die JDBC-Anfrage
            // einstellen und die Anfrage ausführen

            id = getNextID();
            stmt.setString(1, id);
            stmt.setDate(2, new java.sql.Date(born.getTime()));
            stmt.setString(3, breed);
            stmt.setString(4, name);
            stmt.execute();

            // Zuordnen (Initialisieren) der Datenmember

            this.id = id;
            this.name = name;
            this.born = born;
            this.breed = breed;
    }
```

```
    catch (SQLException sqe) {
           System.err.println ("SQLException: " + sqe);
           throw new EJBException(sqe);
    }
    finally {
           cleanup(conn, stmt);
    }
    return id;
}

// Nachbehandlung des Beans, bevor es vom Klienten verwendet wird.

public void ejbPostCreate(String name, java.util.Date born, String breed)
           throws CreateException
{
    // wird hier nicht benoetigt
}

// sobald der Klient die Methode remove aufruft, wird dieser Aufruf von dem Container
// in ein ejbRemove umgesetzt und bedingt ein Loeschen des Datums im persistenten
// Speicher

public void ejbRemove() throws RemoveException
{
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
           conn = getConnection();
           stmt = conn.prepareStatement("DELETE FROM DOG_TABLE WHERE DOGID=?");

           stmt.setString(1, getID());
           stmt.execute();
    }
    catch (SQLException sqe) {
           System.err.println ("SQLException: " + sqe);
           throw new EJBException(sqe);
    }
    finally {
           cleanup(conn, stmt);
    }
}

// Sobald der Klient eine Methode seiner Geschäftslogik aufruft, wird von dem
// Container die Methode ejbLoad aufgerufen. Dieser Aufruf bedingt immer eine
// Suchanfrage im persistenten Speicher. Der Klient hat bereits im Vorfeld durch
// die find-Methoden den primären Schlüssel übergeben, der in dem EntityBean-
// Kontext zwischengespeichert ist.

public void ejbLoad()
{
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
           // ermitteln des Primärschlüssels über den gesetzten Kontext.

           String id = (String) ctx.getPrimaryKey();
           this.id = id;

           conn = getConnection();
           stmt = conn.prepareStatement("SELECT * FROM DOG_TABLE WHERE DOGID=?");
           stmt.setString(1, id);
           stmt.executeQuery();
           ResultSet rs = stmt.getResultSet();
           if (! rs.next())
                  throw new EJBException(
                         "Fehler beim Lesen von Hund #" + id +
                         ". Der Hund wurde nicht gefunden");

           name  = rs.getString("Name");
           born  = rs.getDate("Born");
           breed = rs.getString("breed");
    }
    catch (SQLException sqe) {
           System.err.println ("SQLException: " + sqe);
           throw new EJBException(sqe);
    }
    finally {
           cleanup(conn, stmt);
```

```
    }
}

// nach Beendigung einer jeden Methode, die Datenmember manipuliert, ruft der
// Container ejbStore auf, um den neuen Zustand persistent zu machen.

public void ejbStore()
{
    Connection            conn = null;
    PreparedStatement     stmt = null;
    try {
           conn = getConnection();
           String sql = "UPDATE DOG_TABLE SET BORN=?, BREED=?, " +
                        "NAME=? WHERE DOGID = ?";
           stmt = conn.prepareStatement(sql);
           stmt.setDate(1, new java.sql.Date(getBorn().getTime()));
           stmt.setString(2, getBreed());
           stmt.setString(3, getName());
           stmt.setString(4, getID());
           stmt.execute();
    }
    catch (SQLException sqe) {
           System.err.println ("SQLException: " + sqe);
           throw new EJBException(sqe);
    }
    finally {
           cleanup(conn, stmt);
    }
}

// die Methode findByPrimaryKey wird von dem Container überführt in die Methode
// ejbFindByPrimaryKey mit exakt der gleichen Signatur. Die Rückgabe an den
// Container ist die Instanz des Primärschlüssels, die Rückgabe von
// findByPrimaryKey durch den Container an den Klienten ist der Stub des Remote-
// Interfaces

public String ejbFindByPrimaryKey(String id) throws FinderException
{
    Connection            conn = null;
    PreparedStatement     stmt = null;
    try {
           conn = getConnection();
           stmt = conn.prepareStatement("SELECT * FROM DOG_TABLE WHERE DOGID=?");
           stmt.setString(1, id);
           stmt.executeQuery();
           ResultSet rs = stmt.getResultSet();
           if (!rs.next())
                  throw new ObjectNotFoundException();
    }
    catch (SQLException sqe) {
           System.err.println ("SQLException: " + sqe);
           throw new EJBException(sqe);
    }
    finally {
           cleanup(conn, stmt);
    }
    return id;
}

// analog wird die Methode findAllDogs des Home-Interfaces in die Methode
// ejbFindAllDogs überführt. Die Rückgabe ist eine Menge von primären Schlüssel-
// instanzen an den Container, die Rückgabe des Containers an den Klienten ist eine
// Menge von Stubs des jeweiligen Remote-Interfaces

public Collection ejbFindAllDogs()
{
    Connection            conn    = null;
    PreparedStatement     stmt    = null;
    Vector                c       = new Vector();
    try {
           conn = getConnection();
           stmt = conn.prepareStatement("SELECT DOGID FROM DOG_TABLE");
           stmt.executeQuery();
           ResultSet rs = stmt.getResultSet();

           while(rs.next()) {
                  c.addElement(rs.getString("DOGID"));
           }
```

```
        }
        catch (SQLException sqe) {
               System.err.println ("SQLException: " + sqe);
               throw new EJBException(sqe);
        }
        finally {
               cleanup(conn, stmt);
        }
        return c;
    }

    protected String          id;
    protected String          name;
    protected String          breed;
    protected java.util.Date  born;
    private   EntityContext   ctx;
}
```

Programmbeispiel 81: Implementierung von DogEBean

Im Gegensatz zu den vorherigen SessionBeans wird in der nachfolgenden *ejb-jar.xml*-Datei ein EntityBean beschrieben, das in einigen Punkten von der Beschreibung eines SessionBeans abweicht. Im Detail wird hier festgelegt:

1. Die Beschreibung des Beans wird eingeleitet durch das Tag *<entity>*.
2. Für das zu beschreibende Bean wird ein symbolischer Name durch das Tag *<ejb-name>* festgelegt. Unter diesem Namen wird das Bean bei weiteren Beschreibungen, ggf. in anderen XML-Beschreibungsdateien, referenziert.
3. Die vollständig qualifizierten Angaben zum HomeInterface (Tag *<home>*), RemoteInterface (Tag *<remote>*) und der Implementierungsklasse (Tag *<ejb-class>*) werden festgelegt.
4. Mit *<persistence-type>* wird dem EJB-Kompilierer mitgeteilt, ob dieses Bean die Persistenz selbst durchführt oder diese Aufgabe an zu generierende Software delegiert wird. Die beiden möglichen Werte sind *Bean* oder *Container*.
5. Als primärer Schlüssel wird ein String verwendet, der als Wert des Tags *<prim-key-class>* vollqualifiziert anzugeben ist.
6. Es ist wichtig festzuhalten, dass es allen EntityBeans untersagt ist, Transaktionsverarbeitung programmtechnisch umzusetzen. Die Spezifikation 2.0 verlangt die Delegation der Transaktionsverarbeitung an die generierte Software. Die gewünschte Transaktionspolitik ist für alle Methoden dieses Beans unter *<assembly-descriptor><container-transaction>* anzugeben. Im vorliegenden Beispiel wird für das gesamte Bean die Transaktionspolitik *Required* vereinbart.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
    <enterprise-beans>
        <entity>
        <ejb-name>DogEJB</ejb-name>
        <home>de.uni.mainz.bmp.DogHome</home>
        <remote>de.uni.mainz.bmp.Dog</remote>
        <ejb-class>de.uni.mainz.bmp.DogBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction> <!-- alle Methoden des RemoteInterfaces besitzen die
                                    Transaktionspolitik 'Required'. -->
        <method>
                <ejb-name>
                        DogEJB
                </ejb-name>
                <method-intf>
                        Remote
                </method-intf>
```

```
                <method-name>
                        *
                </method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Programmbeispiel 82: ejb-jar.xml für DogEBean

Die Beschreibungsdatei *weblogic-ejb-jar.xml* ist eine proprietäre Datei, in der für den Weblogic Server weitere Vereinbarungen zu dem Bean getroffen werden können. Die minimale Angabe muss der JNDI-Name sein, unter dem der Weblogic Server den Stub des Home-Interfaces in den Baum einbindet.

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN'
'http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>DogEJB</ejb-name>
        <entity-descriptor>
        <entity-cache>
                <max-beans-in-cache>100</max-beans-in-cache>
        </entity-cache>
        </entity-descriptor>
        <jndi-name>de.uni.mainz.bmp.DogHome</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Programmbeispiel 83: weblogic-ejb-jar.xml für DogEBean

Der Klient ist in diesem Fall ebenfalls als eine JavaServer Page (JSP) implementiert.

```
<%@ page import="javax.naming.*, java.rmi.*, javax.rmi.*, de.uni.mainz.bmp.*; java.text.*,
         session="true" %>
<%!
        // Deklaration von benötigten Datenmembern

        private static Context ctx;

        Iterator iter;
        DogHome home;

    // Verbindung zum JNDI-Baum herstellen

    static {
        try {
        ctx = new InitialContext();
        }
        catch (Exception e) {
        System.out.println(e.getMessage());
        }
    }
%>
<%
{
    // Stub des Home-Interfaces abfragen, um darüber eine Beaninstanz zu erzeugen

    home = (DogHome) PortableRemoteObject.narrow(ctx.lookup("de.uni.mainz.bmp.DogHome"),
                                                    DogHome.class);

    // Eingabeparameter des HTML-Formulares verarbeiten

    if (request.getParameter("ADDDOG") != null) {
        String name = (String)request.getParameter("Name");
        String breed = (String)request.getParameter("Breed");
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
        String stringDate = (String)request.getParameter("Born");
        Date born = formatter.parse(stringDate,new ParsePosition(0));
```

```
        // Beaninstanz erzeugen und mit den Angaben initialisieren

        home.create(name,born,breed);
    }

    // wurde von dem Benutzer ‚remove' aufgerufen? Wenn ja, einen Löschauftrag
    // an das Bean schicken (home.remove())

    String remove=request.getParameter("REMOVEDOG");
    if ((remove != null) && !remove.equals(""))
        home.remove(remove);

    // alle Hundeeinträge der Datenbank für die Auflistung der HTML-Seite abfragen

    Collection c = (Collection)home.findAllDogs();
    iter = c.iterator();
}
%>

<HTML>
    <HEAD><TITLE>Hundedatenbank</TITLE></HEAD>
    <BODY>
        <H1>Liste der verf&uuml;gbaren Hunde</H1>
        <UL>
        <%
               while (iter.hasNext()) { // alle Hunde anzeigen
                      Dog d = (Dog)PortableRemoteObject.narrow(iter.next(),Dog.class);
        %>
        <LI>DOG #<%=d.getID()%>, name: <%=d.getName()%>,
                breed: <%=d.getBreed()%>,
                born on: <%=d.getBorn()%>.
                <A HREF="index.jsp?REMOVEDOG=<%=d.getID()%>">[DELETE]</A>
                <% } %>
        </LI>
        <UL>
        <HR>
        <BR>
        <FORM METHOD=POST ACTION="BMP.jsp?ADDDOG=YES">
               <B>Neuen Hund eintragen</B><BR>
        Name: <INPUT TYPE=TEXT NAME="Name"><BR>
               Breed: <INPUT TYPE=TEXT NAME="Breed"><BR>
               Birth date (YYYY-MM-DD): <INPUT TYPE=TEXT NAME="Born">
               <BR><BR>
               <INPUT TYPE=SUBMIT VALUE="Create Dog">
               <INPUT TYPE=RESET>
        </FORM>
    </BODY>
</HTML>
```

Programmbeispiel 84: Klient für DogEBean

Die Arbeitsmaske sieht wie folgt aus:

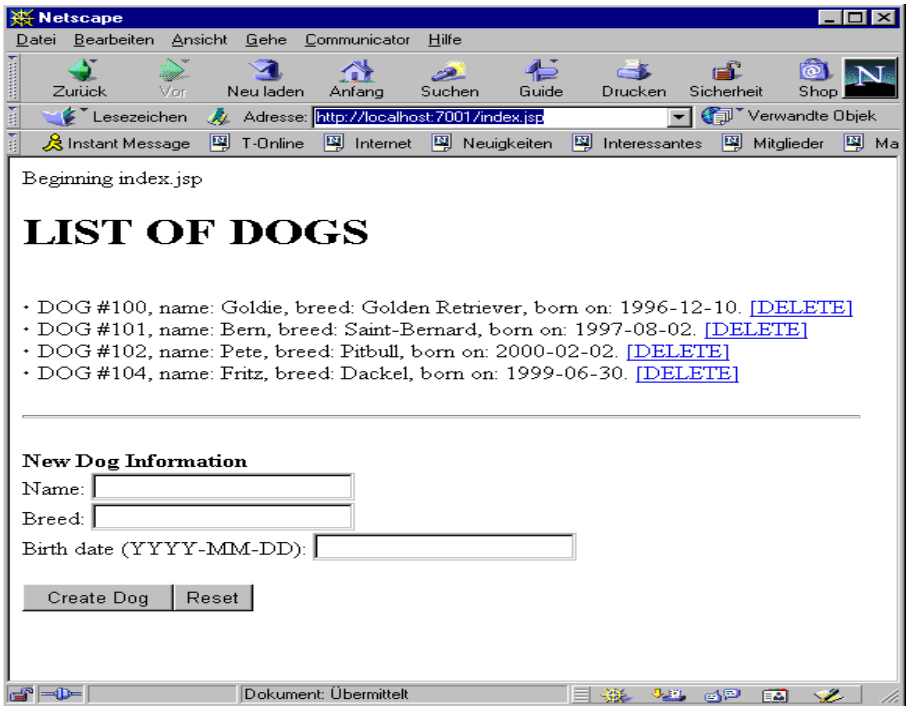


Abbildung 27: Die Klientensicht durch die angegebene JavaServer Page

13.6.3 Container Managed Persistence

Bei dem Prinzip der Container Managed EntityBeans wird Persistenz durch geeignet generierte Software abgewickelt. Gegenüber dem anderen Entity-Typ ergeben sich somit gravierende Unterschiede:

1. Die gesamte Problematik, Daten persistent in der Datenbank abzulegen, wird durch generierte Software übernommen. Dies beinhaltet, dass der Entwickler
   a. die Implementierungsklasse als abstrakt deklariert. Der EJB-Compiler wird diese Klasse als Basisklasse verwenden.
   b. keine Datenmember in der Basisklasse definiert. Stattdessen muss er für jeden Datenmember eine *set*- und eine *get*-Methode deklarieren. Dies ist darauf zurückzuführen, dass die Basisklasse abstrakt ist. Es ist die Aufgabe des EJB-Compilers, diese Methoden zu definieren und die Datenmember in der Ableitung einzubringen. Die Namenskonvention lautet

      ```
      void set<Name>(<Datentyp> x);
      ```

      und

      ```
      <Datentyp> getName();
      ```

      Beispielsweise werden für den Datenmember *kunde* die Methoden

      ```
      void setKunde(Kunde k);
      ```

      und

      ```
      Kunde getKunde();
      ```

      deklariert. Hierbei ist *Kunde* eine benutzerdefinierte Datenstruktur der Form

      ```
      public class Kunde implements java.io.Serializable { ... }
      ```

      Optional können diese Methoden in dem RemoteInterface aufgenommen werden, was jedoch dazu führt, dass set- und get-Methoden von dem Klienten verwendet werden und nicht die vollständige Information auf einmal an das Bean weitergegeben wird (vgl. Kapitel 8).

2. Der Entwickler deklariert die Datenmember in der Beschreibungsdatei *ejb-jar.xml*.
3. Für die Methode *findByPrimaryKey* wird in der Datei *ejb-jar.xml* lediglich hinterlegt, welcher Datenmember den primären Schlüssel repräsentiert.
4. Andere *find*-Methoden werden in der Datei *ejb-jar.xml* ggf. unter Angabe der Übergabeparameter bekannt gegeben. Für jede dieser Methoden wird auf Basis der Beschreibungssprache EJB-QL (**E**nterprise **J**ava**B**eans-**Q**uery **L**anguage) ein Kriterium hinterlegt, nach der bestimmte Datenmember mit den übergebenen Parametern verglichen werden. Es ist Aufgabe des EJB-Compilers, auf Basis dieser Informationen geeignete Methoden zu definieren.
5. In einer weiteren Beschreibungsdatei wird die Abbildung von Java-Datenmember auf Datenbanktypen beschrieben. Zusätzlich ist ein *DataSource*, der den *JDBC-Connection-pool* repräsentiert bekannt zugegeben und die Tabelle, mit der dieses Bean arbeitet.

13.6.4 Beispiel: Container Managed Persistence

Das o. a. Beispiel wird hier dahingehend umgestellt, dass die JDBC-Anfragen durch die generierte Software vorgenommen werden. Das Remote- und das HomeInterface sind mit Ausnahme des Paketnamens identisch mit dem vorherigen Beispiel.

Die Implementierungsklasse sieht somit wie folgt aus:

```
package de.uni.mainz.cmp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

public abstract class DogBean implements EntityBean {
    private EntityContext ctx;

    public String getID() { return getId(); }
    public abstract String getId();
    public abstract void setId(String id);

    public abstract String getName();
    public abstract void setName(String n);

    public abstract String getBreed();
    public abstract void setBreed(String b);

    public abstract java.util.Date getBorn();
    public abstract void setBorn(java.util.Date b);

    public DogBean() {}

    public void setEntityContext(EntityContext ctx)
    {
        this.ctx = ctx;
    }

    public void unsetEntityContext()
    {
        this.ctx = null;
    }

    public void ejbActivate() { }
    public void ejbPassivate() { }

    public String ejbCreate(String name, java.util.Date born, String breed)
                throws CreateException
    {
        setName(name);
        setBorn(born);
        setBreed(breed);
```

```
        return null;
    }

    public void ejbPostCreate(String name, java.util.Date born, String breed)
        throws CreateException
    {
    }

    public void ejbRemove() throws RemoveException
    {
    }

    public void ejbLoad()
    {
    }

    public void ejbStore()
    {
    }
}
```

Programmbeispiel 85: Implementierung von DogEBean als CMP

Das Deployment bei den Container Managed Persistence EntityBeans unterscheidet sich zu den anderen EntityBeans in den folgenden Punkten:

1. Alle Java-Datenmember sind in der *ejb-jar.xml*-Datei zu beschreiben. Jeder Datentyp wird unter *<cmp-field><field-name>* bekannt gegeben.
2. Alle *find*-Methoden des HomeInterfaces werden vom EJB-Compiler generiert. In der Beschreibungsdatei sind diese Methoden unter *<query><query-method>* mit Namen und Übergabeparametern bekannt zu geben.
3. Für jede *find*-Methode ist ein Vergleichskriterium unter *<query><ejb-ql>* zu hinterlegen.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd'>
<ejb-jar>
    <enterprise-beans>
        <entity>
                <ejb-name>DogEJB</ejb-name>
                <home>de.uni.mainz.cmp.DogHome</home>
                <remote>de.uni.mainz.cmp.Dog</remote>
                <ejb-class>de.uni.mainz.cmp.DogBean</ejb-class>
                <persistence-type>Container</persistence-type>
                <prim-key-class>java.lang.String</prim-key-class>
                <reentrant>False</reentrant>
                <cmp-version>2.x</cmp-version>
                <abstract-schema-name>DogEBean</abstract-schema-name>
                <cmp-field><field-name>id</field-name></cmp-field>
                <cmp-field><field-name>name</field-name></cmp-field>
                <cmp-field><field-name>breed</field-name></cmp-field>
                <cmp-field><field-name>born</field-name></cmp-field>
                <primkey-field>id</primkey-field>
                <query>
                        <query-method>
                                <method-name>
                                        findAllDogs
                                </method-name>
                                <method-params></method-params>
                </query-method>
                        <ejb-ql>
                                <![CDATA[SELECT OBJECT(d) FROM DogEBean d]]>
                        </ejb-ql>
                </query>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
                <method>
                        <ejb-name>DogEJB</ejb-name>
                        <method-name>*</method-name>
                </method>
                <trans-attribute>Required</trans-attribute>
        </container-transaction>
```

```
    </assembly-descriptor>
</ejb-jar>
```

Programmbeispiel 86: ejb-jar.xml für ein CMP-Bean

In der *weblogic-ejb-jar.xml*-Datei ist eine weitere Beschreibungsdatei bekannt zu geben, aus der hervorgeht, mit welcher Datenbank und welcher Tabelle einer Datenbank gearbeitet wird. In diesem Beispiel ist es die Beschreibungsdatei *META-INF/weblogic-cmp-rdbms-jar.xml*-Datei. Die anderen Angaben bestimmen lediglich, dass mit einem Weblogic Server der Version 6.x gearbeitet wird und dass es sich um ein Container Managed Persistence Bean handelt.

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN'
'http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>DogEJB</ejb-name>
        <entity-descriptor>
                <entity-cache>
                        <max-beans-in-cache>150</max-beans-in-cache>
                </entity-cache>
                <persistence>
                        <persistence-type>
                                <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
                                <type-version>6.0</type-version>
                                <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-storage>
                        </persistence-type>
                        <persistence-use>
                                <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
                                <type-version>6.0</type-version>
                        </persistence-use>
                </persistence>
        </entity-descriptor>
        <jndi-name>DogEJB</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Programmbeispiel 87: weblogic-ejb-jar.xml für ein CMP-Bean

In der proprietären Beschreibungsdatei *weblogic-cmp-rdbms-jar.xml* muss folgendes beschrieben sein:

1. Der symbolische Name aus der *ejb-jar.xml*-Datei, um anzuzeigen, für welches Bean diese Vereinbarungen gelten.
2. Ein *DataSource*, der genau den *JDBC-Connectionpool* repräsentiert, der Verbindungen zu genau dieser Datenbank aufgebaut hat.
3. Der Tabellenname, mit dem gearbeitet wird.
4. Für jeden Java-Datenmember ist bekannt zu geben, auf welchen Datenbankeintrag dieses Element abgebildet wird.

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-rdbms-jar PUBLIC
 '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB RDBMS Persistence//EN'
 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-persistence-600.dtd'>
<weblogic-rdbms-jar>
    <weblogic-rdbms-bean>
        <ejb-name>DogEJB</ejb-name>
        <data-source-name>wlsd21-datasource</data-source-name>
        <table-name>DOG_TABLE</table-name>
        <field-map>
                <cmp-field>name</cmp-field>
                <dbms-column>NAME</dbms-column>
        </field-map>
        <field-map>
                <cmp-field>id</cmp-field>
                <dbms-column>DOGID</dbms-column>
        </field-map>
```

```
        <field-map>
                <cmp-field>born</cmp-field>
                <dbms-column>BORN</dbms-column>
        </field-map>
        <field-map>
                <cmp-field>breed</cmp-field>
                <dbms-column>BREED</dbms-column>
        </field-map>
    </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

Programmbeispiel 88: weblogic-cmp-rdbms-jar.xml für ein CMP-Bean

Der Zugang des Klienten zu dem EntityBean ist identisch zu dem vorherigen Klienten.

13.7 Der Java Message Service

Der **J**ava **M**essage **S**ervice (JMS) gehört zur Kategorie der Message Oriented Middleware. Sun Microsystems hat auch hier keine Implementierungsdetails vorgegeben, sondern eine Menge von Schnittstellen standardisiert.

Mit JMS ist die Möglichkeit der asynchronen Kommunikation gegeben. In der EJB 2.0 Spezifikation wird zusätzlich ein neues Enterprise Java Bean, das Message Driven Bean, eingeführt, das ausschließlich diesen asynchronen Ansatz unterstützt. Ein sehr empfehlenswertes Buch zum Java Message Service ist [Monson-Haefel, 2001].

13.7.1 JMS Terminologie

Die Spezifikation verlangt die Unterstützung von

1. *Point-to-Point*: Mit diesem Punkt-zu-Punkt-Ansatz kann der Produzent einer Nachricht via JMS-Server diese Nachricht in eine Queue einstellen und der JMS-Server wird aus einer Menge von Empfängern genau einen auswählen.

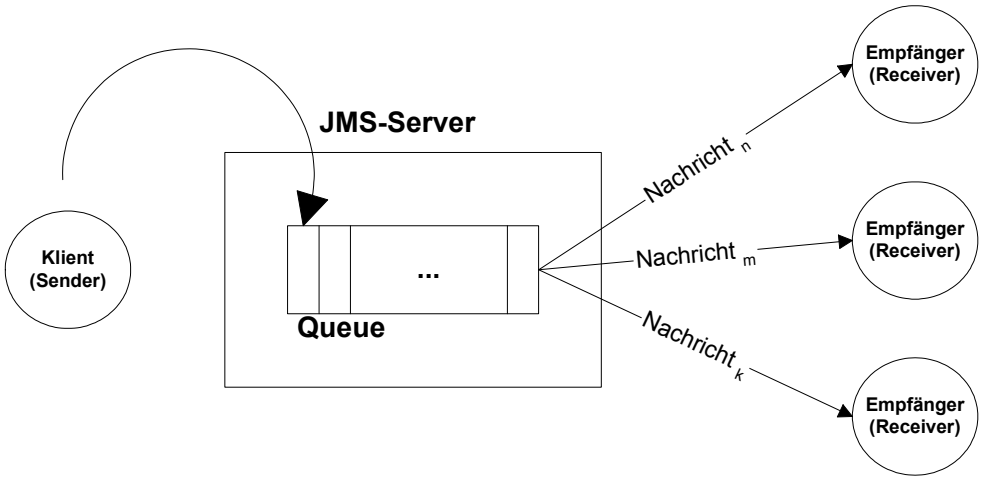


Abbildung 28: Darstellung des Punkt-zu-Punkt-Prinzips

2. *Publish/Subscribe*: Dieser Ansatz entspricht einem verteilten Ereignisdienst. Der Produzent einer Nachricht stellt diese in ein Topic ein und alle an diesem Topic Interessierten bekommen eine Kopie der Daten.

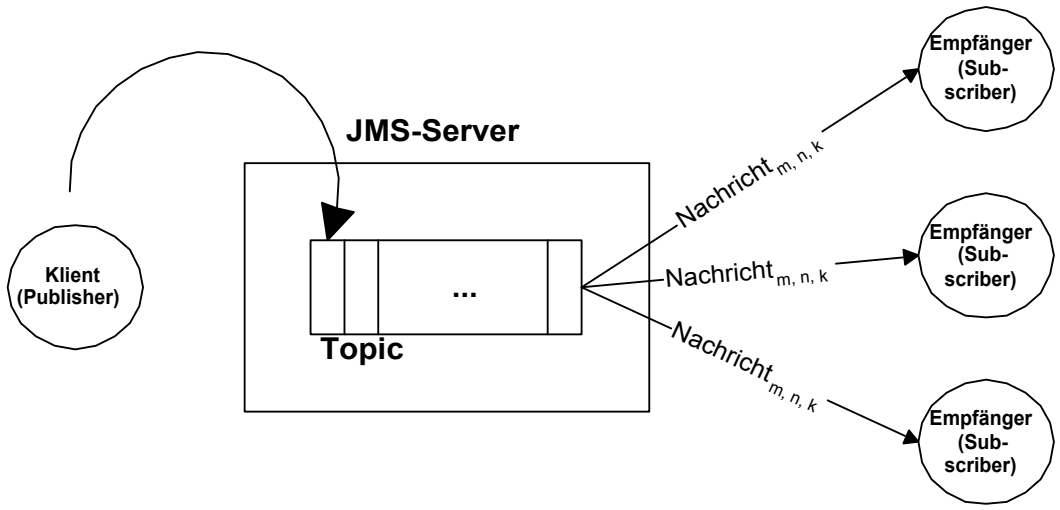


Abbildung 29: Darstellung des Publish/Subscribe-Prinzips

13.7.2 JMS-Datentypen

Die Spezifikation erlaubt für die zu übertragenen Nutzdaten die folgenden Datentypen:

1. *StreamMessage*: Dieser Datentyp erlaubt ein stromförmiges Einstellen von skalaren Datentypen. Der Sender und der Empfänger der Nachricht müssen über die Reihenfolge, in der die Daten eingestellt werden, Absprache treffen.
2. *MapMessage*: Dieser Nachrichtentyp arbeitet nach dem Prinzip *Attribut=Wert*.
3. *ByteMessage*: Im Prinzip ist eine ByteMessage eine gekapselte Folge von byte-Werten.
4. *ObjectMessage*: Ist der allgemeinste Datentyp, da eine ObjectMessage beliebige, serialisierbare Objekte aufnehmen kann.
5. *TextMessage*: Ist im Prinzip ein gekapselter String.

13.7.3 Aufbau einer JMS-Nachricht

Eine JMS-Nachricht setzt sich aus einem Nachrichtenkopf, einem Eigenschaftsteil und den eigentlichen Nutzdaten zusammen. Der Kopf besteht aus den folgenden Feldern:

1. *JMSDestination*: Hiermit wird das Ziel, also eine Queue oder ein Topic, angegeben.
2. *JMSDeliveryMode*: Der Sender einer Nachricht kann hier angeben, ob der JMS-Server seine Nachricht persistent speichern soll oder nicht. Dazu existieren die beiden Konstanten
   ```
   javax.jms.DeliveryMode.PERSISTENT;
   ```
   und
   ```
   javax.jms.DeliveryMode.NON_PERSISTENT;
   ```
   Es ist die Aufgabe des JMS-Servers, eine persistente Nachricht nach erfolgreicher Zustellung wieder aus dem persistenten Speicher zu entfernen.
3. *JMSMessageID*: Der Nachrichtenidentifizierer wird beim Senden einer Nachricht von dem System eingetragen und sollte eindeutig sein.
4. *JMSTimestamp*: Wird automatisch bei der send-Operation eingetragen.
5. *JMSExpiration*: Der Entwickler kann hiermit die Lebensdauer einer Nachricht setzen. Ist die Nachricht nicht zustellbar, so wird sie nach Ablauf der Zeitspanne durch den JMS-Server gelöscht.

6. *JMSRedelivered*: Dieses Feld indiziert, ob eine Nachricht erneut an den Konsumenten ausgeliefert wurde oder nicht.
7. *JMSPriority*: Durch den Entwickler kann für seine Nachricht eine Priorität zwischen 1 und 10 vereinbart werden. Eine Nachricht mit hoher Priorität wird vorrangig bearbeitet.
8. *JMSReplyTo*: Der Produzent einer Nachricht gibt das Ziel der Antwortnachricht vor.
9. *JMSCorrelationID*: Kann von einem Entwickler verwendet werden, um z. B. die Reihenfolge von Nachrichten und ihren Bezug zueinander zu setzen.
10. *JMSType*: Ist ein String, der den zu übertragenen Datentyp beschreiben kann.

Für alle Kopffelder existieren Zugriffsmethoden, mit denen Werte neu gesetzt oder gegenwärtig gesetzte Informationen abgefragt werden können.

Die Eigenschaften einer JMS-Nachricht werden nach dem *Attribut*=*Wert*-Prinzip gesetzt. Ein Wert wird also an einen bestimmten Namen gebunden. Bspw. könnte vereinbart werden, dass jeder Produzent einer Nachricht als Eigenschaft seinen Benutzernamen (user.name=name) bekannt gibt, damit der Empfänger ihn sofort identifizieren kann.

Von Interesse können die Eigenschaften auch sein, wenn ein potentieller Empfänger einer Nachricht eine Selektion bestimmter Nachricht anstrebt. Bspw. könnte ein Empfänger ein Interesse an allen Nachrichten eines bestimmten Benutzers haben und würde durch einen Selektorstring dem JMS-Server diesen Hinweis geben.

Der Empfänger einer Nachricht teilt bei der Registrierung sein Interesse an Nachrichten aus dieser Queue dem JMS-Server mit, das er alle Nachrichten des Benutzers *otto*, die eine Priorität ≥ 6 besitzen, zugestellt bekommen will.

```
String selektor = "user.name=otto and JMSPriority >= 6";
session.createReceiver(queue, selektor, false);
```

Beispiel 14: Selektoren für JMS

13.8 Message Driven Beans

Die Message Driven Beans sind ein neuer Typ, die mit der neuen Spezifikation 2.0 hinzugefügt werden. Die anderen Beantypen arbeiten ausschließlich mit synchronen Methodenaufrufen. Um im Zusammenhang mit Enterprise JavaBeans asynchrone Kommunikation zu ermöglichen, wurde dieser neue Typ eingeführt.
Die Eigenschaften eines Message Driven Beans sind:

1. Das Bean ist primär der Konsument von Nachrichten. Dies kann sowohl die Rolle als Empfänger einer Queue-, als auch die Rolle als Empfänger einer Topic-Nachricht sein.
2. Das Bean ist per Definition zustandslos.
3. Das Bean besitzt kein Home- und kein RemoteInterface. Dies ist darauf zurückzuführen, dass kein Klient einen direkten Zugang zu diesem Bean besitzt, sondern immer den Umweg über einen JMS-Server nehmen muss. Anstelle eines RemoteInterfaces tritt die Schnittstelle *MessageListener*, die über genau eine Methode
   ```
   public void onMessage(Message m);
   ```
   verfügt.
4. Über die Deployment Deskriptoren wird festgelegt, ob das Bean mit einer Queue oder einem Topic arbeitet und die benötigten JNDI-Namen der Queue oder des Topics werden hinterlegt. Es ist die Aufgabe der generierten Software, eine Verbindung zum JMS-Server herzustellen, das Interesse an einer Queue oder einem Topic zu registrieren, das benötigte Sitzungsobjekt zu erzeugen und in dieser Sitzung den Empfänger von Nachrichten zu erzeugen.
5. Will das Message Driven Bean auch als Produzent von Nachrichten auftreten, so muss dieser Prozess programmtechnisch in der o. a. Methode vereinbart werden.

13.9 Der Zugang eines Klienten zu JMS

JMS arbeitet grundsätzlich nach dem Fabrikmuster. Für einen Klienten bedeutet dies:
1. Über JNDI wird eine ConnectionFactory abgefragt. Diese ConnectionFactory ist zuständig, um eine Verbindung zum JMS-Server herzustellen.
2. Über JNDI wird die Queue oder das Topic erfragt, mit dem gearbeitet werden soll.
3. Über das Objekt ConnectionFactory wird eine Verbindung zum JMS-Server etabliert. Das erhaltene Connection-Objekt dient zusätzlich der Erzeugung von Sitzungen.
4. Der Klient erzeugt eine oder mehrere Sitzungen. In jeder dieser Sitzungen kann er als Konsument und als Produzent von Nachrichten auftreten.

13.10 Transaktionspolitiken für Enterprise JavaBeans

Es wurde bei den Charakteristiken von Komponenten bereits die Möglichkeit eines Automatismus angeführt. D. h., dass ein Entwickler bestimmte Aufgabe wie z. B. Sicherheit, Transaktionsverarbeitung oder die Frage der Persistenz an eigens generierte Software delegieren kann.

Insbesondere für die EntityBeans darf laut Spezifikation 2.0 die Transaktionsverarbeitung nicht mehr programmtechnisch abgewickelt werden. Für alle anderen Beantypen kann von diesem Prinzip des **C**ontainer **M**anaged **T**ransaction (CMT) ebenfalls Gebrauch gemacht werden.

Die Transaktionspolitiken bei CMT werden in einen Deployment Deskriptor, konkret in der XML-Beschreibungsdatei *ejb-jar.xml*, hinterlegt. Für jedes Bean kann entweder für das gesamte Bean eine Politik festgelegt werden, oder für jede Methode des Beans. Wird in dieser Beschreibungsdatei CMT vereinbart und keine Transaktionspolitik angegeben, so wird eine Standardeinstellung verwendet.

Im Folgenden werden die sechs Transaktionspolitiken des CMT im Detail erörtert. Es wird in der Beschreibung vorausgesetzt, dass der jeweilige Klient über das *UserTransaction*-Object verfügt, das in den Abbildungen durch die konkrete Instanz *ut* dargestellt wird.

13.10.1 Die Transaktionspolitik Never

Die Transaktionspolitik *Never* erwartet, dass ein Klient keine Transaktionsklammer gesetzt hat und in diesem Rahmen die Komponente kontaktiert. Anderenfalls wird von dem Container eine RemoteException ausgelöst und die Komponente nicht angesprochen.

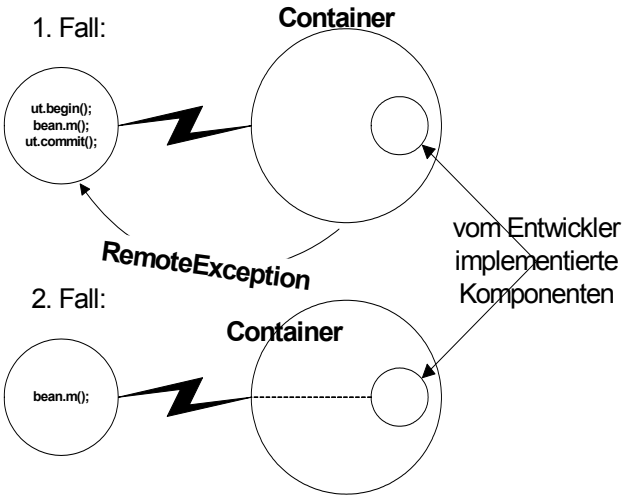


Abbildung 30: Die Arbeitsweise der Transaktionspolitik Never

13.10.2 Die Transaktionspolitik Supports

Die Transaktionspolitik *Supports* ist die Standardeinstellung, falls in der Beschreibungsdatei keine andere Vereinbarung getroffen wird. Der Container übernimmt keine Überprüfung, ob der Klient eine Transaktion eröffnet hat oder nicht. Der Methodenaufruf wird sofort an die Komponente weitergeleitet.

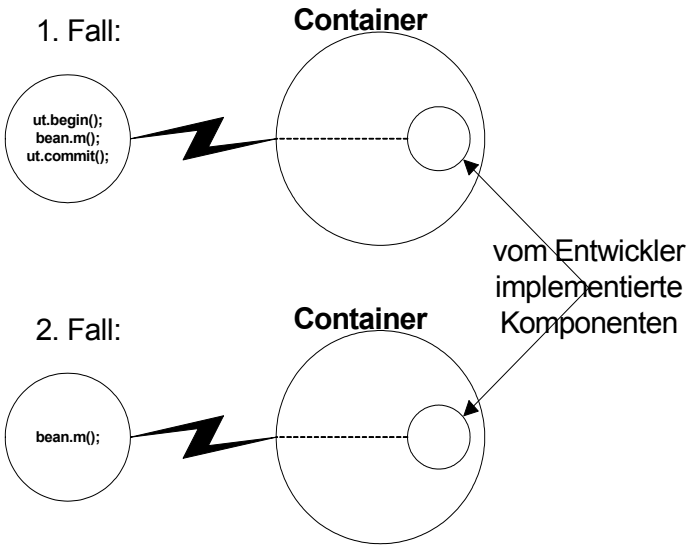


Abbildung 31: Die Transaktionspolitik Supports

13.10.3 Die Transaktionspolitik NotSupported

Die Transaktionspolitik *NotSupported* suspendiert eine durch den Klienten gesetzte Transaktion, ruft die Komponente an und wird anschließend die alte Transaktion wieder aktivieren.

Diese Politik ist z. B. bei EntityBeans, die ausschließlich lesend auf die Datenbank zugreifen, geeignet. Damit wird sichergestellt, dass lesende Zugriffe parallel ablaufen können.

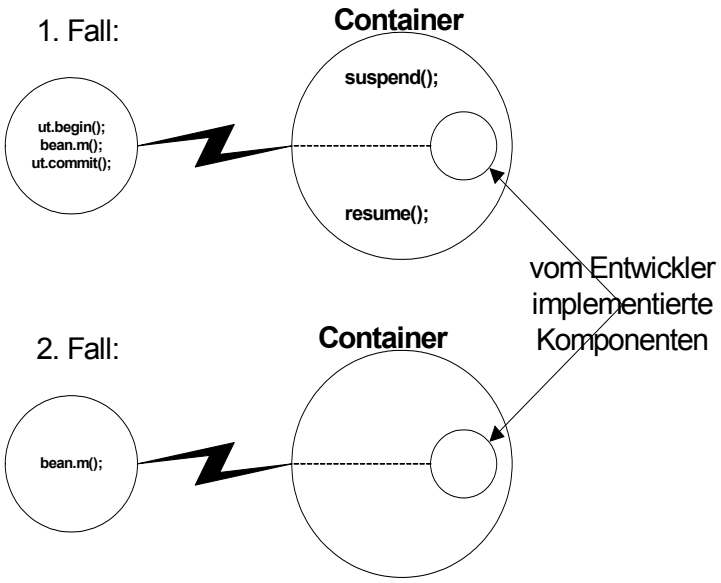


Abbildung 32: Die Transaktionspolitik NotSupported

13.10.4 Die Transaktionspolitik Required

Die Transaktionspolitik *Required* verlangt, dass die Komponente immer im Rahmen einer Transaktion angesprochen wird. Versucht ein Klient ohne gesetzte Transaktionsklammer die Komponente anzusprechen, so wird der Container von sich aus eine Transaktion initiieren und nach erfolgreicher Verarbeitung der Komponentenmethode diese Transaktion durch ein commit() oder ein rollback() wieder schließen.

Diese Transaktionspolitik ist z. B. geeignet für EntityBeans, die schreibend auf die Datenbank zugreifen.

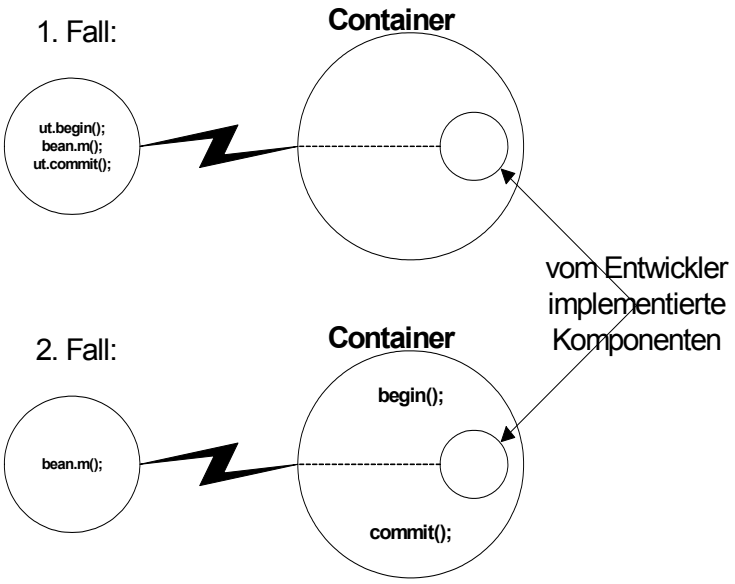


Abbildung 33: Die Transaktionspolitik Required

13.10.5 Die Transaktionspolitik RequiresNew

Die Transaktionspolitik *RequiresNew* wird grundsätzlich eine bestehende Transaktion außer Kraft setzen, eine eigene Transaktion initiieren, die Komponente ansprechen, die eigene Transaktion wieder aufheben und die vorher suspendierte Transaktion wieder aufnehmen.

Die Politik *RequiresNew* belastet demzufolge massiv den Container und sollte nur in Ausnahmefällen eingesetzt werden.

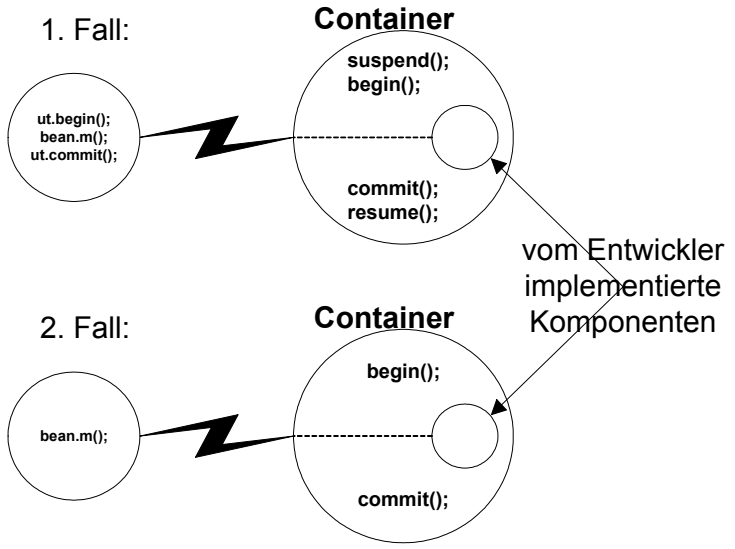


Abbildung 34: Die Arbeitsweise der Transaktionspolitik RequiresNew

13.10.6 Die Transaktionspolitik Mandatory

Die Transaktionspolitik *Mandatory* verlangt verbindlich, dass die Komponente ausschließlich im Rahmen einer Transaktion angesprochen wird. Im Fall, das der Klient diese Vereinbarung nicht einhält, wird eine Ausnahme *TransactionRequired* ausgelöst.

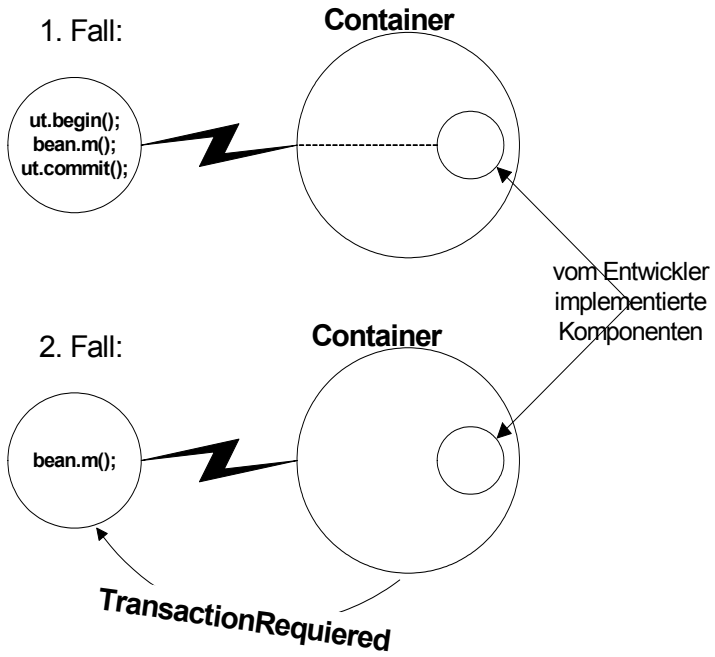


Abbildung 35: Die Arbeitsweise der Transaktionspolitik Mandatory

13.11 Komponenten vs. Objekte

CORBA ist ggw. ein verteiltes Objektmodell und stellt nur wenige und i. d. R. proprietäre Komponenten zur Verfügung. Mit der Umsetzung von CORBA 3 wird sich diese Situation ändern.

Damit verbunden ist ein höherer Entwicklungsaufwand im CORBA-Umfeld. Bei der Verwendung von Transaktionen muss, wie zuvor beschrieben, der CORBA-Entwickler ein entsprechendes Objekt, das so genannte *TransactionCurrent*-Objekt, anfordern und über dieses Objekt die notwendigen Operationen wie Transaktionsklammer setzen und beenden selbst aufrufen.

Für einen Großteil der CORBA-Produkte beinhaltet dies auch eine explizite Programmierung des Portable Object Adapters (POA), über den u. a. auch der Lebenszyklus des Objektes eingestellt wird u. v. a. m.

Als nachteilig bei den Komponenten ist zu benennen:
1. Eine Komponente, die von dem Entwickler nicht vollständig verstanden wurde, wird falsch eingesetzt. Im Zusammenhang mit Enterprise JavaBeans liegen hier bereits erste Erkenntnisse vor. Ein häufig anzutreffender Fehler im Umgang mit EJBs ist die falsch eingestellte Transaktionspolitik, die zu einer drastischen Herabsetzung der Verarbeitungsgeschwindigkeit führen kann.

2. Die Konfiguration einer Komponente, die über die Deployment Deskriptoren erfolgt, ist als statische Einstellung zu sehen. Eine Änderung in einer der Beschreibungsdateien führt zu einem neuen Kompilierlauf und zu einer neuen Bekanntgabe des geänderten Beans in dem jeweiligen Middleware-Produkt.

Auch wenn die Komponenten den Entwickler entlasten sollen und er sich primär auf die Entwicklung seiner Geschäftslogik beschränken soll, muss durch geeignete Schulungsmaßnahmen dafür Sorge getragen werden, dass Komponenten verstanden und somit beherrscht werden.

13.12 Die dynamischen Komponenten von Enterprise JavaBeans

In Bezug auf die dynamischen Komponenten ist bei den Enterprise JavaBeans lediglich die Möglichkeit gegeben, über das HomeInterface bzw. das RemoteInterface Metadaten bzgl. der jeweiligen hierzu korrespondierenden Stubs zu erhalten. Die Klasse Metadaten weist lediglich einige wenige Methoden auf, mit denen bspw. festgestellt werden kann, ob ein Bean ein zustandsloses oder zustandsbehaftetes Bean ist bzw. bei EntityBeans kann der primäre Schlüssel erfragt werden.

Diese Metadaten sind somit bestenfalls für die Entwickler von Entwicklungsumgebungen interessant, die anhand der Metadaten und des Wissens über den Entitytyp bestimmte Deploymentdeskriptoren laden bzw. erzeugen müssen. Ein Interface Repository oder Type-Codes im Sinne von CORBA existieren hier nicht und das Arbeiten mit unbekannten Objekten muss anderweitig realisiert werden.

Eine asynchroner Methodenaufruf oder die Gruppierung von mehreren Anfragen zu einem Anfrageobjekt ist ebenfalls nicht möglich. Bei asynchroner Kommunikation muss der Entwickler grundsätzlich auf den Java Message Service zurückgreifen.

13.13 Die Interoperabilität von CORBA und EJBs

Das Zusammenspiel der beiden Technologien CORBA und J2EE ist primär durch das gemeinsame Protokoll IIOP bzw. RMI-IIOP gewährleistet. Um die Interoperabilität bzgl. J2EE und CORBA 2.x zu untersuchen, ist eine Fallunterscheidung notwendig:

1. *Eine J2EE-Komponente kontaktiert ein CORBA-Objekt:* Diese Richtung ist trivial, da lediglich mit einem IDL-Compiler ein Stub für Java zu generieren ist. Die jeweilige Softwarekomponente verwendet den Namensdienst und ermittelt die Objektreferenz, passt sie an den richtigen Typ an und kann sofort Methoden des Objektes nutzen. Alle Informationen sind statisch im Stub vorhanden und die Implementierungssprache des CORBA-Objektes ist an dieser Stelle irrelevant.
2. *Ein CORBA-Objekt kontaktiert ein Enterprise Java Bean:* Will ein CORBA-Objekt, das beispielsweise in C++ implementiert wurde, ein Session- oder EntityBean kontaktieren, so müsste aufgrund des Factory-Patterns ein Stub des HomeInterfaces aus dem JNDI-Baum gelesen werden. Dieser Stub ist aber in der Programmiersprache Java entwickelt worden und C++ hat keine Möglichkeit, diesen Stub zu binden. Da J2EE ein Standard für serverseitige Java-Komponenten ist, ist die von CORBA bekannte Sprachunabhängigkeit nicht vorhanden und somit kein Stub in der nativen Sprache vorrätig.

   Auch der Einsatz von *Java-2-IDL* liefert lediglich eine Generierung einer IDL-Beschreibung aus bestehenden Java-Schnittstellen. Auch wenn mit einem geeigneten IDL-Compiler der native Stub erzeugt wird, kann keine Kommunikation mit dem EJB stattfinden, da die Spezifikation ausdrücklich verlangt, dass Stubs entweder aus dem JNDI oder per Returnwert von den create-/find-Methoden aus dem HomeInterface zu be-

ziehen sind. Zusätzlich werden bei EJBs Aufrufe durch den generierten EJB-Container unterbrochen, damit die im Deployment Deskriptor beschriebenen Vorarbeiten ausgeführt werden können, bevor das eigentliche Bean kontaktiert wird.

Zwei mögliche Lösungen bieten sich an:

a. Es wird ein Stellvertreterobjekt unter Verwendung der Sprache Java entwickelt, das die Anfrage des C++-Objektes entgegennimmt und stellvertretend an das Enterprise Java Bean weiterreicht. Da dieses Objekt ein Java-Objekt ist, kann es dynamisch den Stub des HomeInterfaces zur Laufzeit binden und anschließend den Stub des RemoteInterfaces anfordern.
b. Durch die Verwendung von valuetypes kann das Stellvertreterobjekt umgangen werden. Die von CORBA bekannte Sprachunabhängigkeit ist gewährleistet; es ist lediglich dafür Sorge zu tragen, dass der IDL-Compiler für Java die benötigten Stubs generiert hat.

14 CORBA vs. Enterprise JavaBeans

In dem vorliegenden Kapitel werden CORBA-Objekte und Enterprise JavaBeans bzgl. ihrer Eigenschaften miteinander verglichen. Dieser Vergleich ist notwendig, um bzgl. der Verarbeitungsgeschwindigkeit der CORBA-Objekte zu zuverlässigen Aussagen bzgl. der Verarbeitungsgeschwindigkeit von EJB-Komponenten gelangen zu können.

14.1 Die Gemeinsamkeiten beider Technologien

Für einen äußeren Betrachter scheinen CORBA-Objekte und die Komponenten der Enterprise JavaBeans völlig identisch zu sein. In der Tat sind sowohl CORBA, als auch die EJBs in vielerlei Hinsicht ähnlich. Die Gemeinsamkeiten bestehen aus:

1. Ein Klient spricht CORBA-Objekte, ebenso wie EJB-Komponenten über eine wohldefinierte Schnittstelle an. Diese Schnittstelle ist im statischen, ebenso wie im dynamischen Fall ein Vertrag zwischen dem Klienten und dem Server. Bei jeder Änderung der Methodennamen, der Ein- und Ausgababeparameter, des Rückgabewertes etc. muss im statischen Fall ein neuer Vertrag zwischen den Beteiligten geschlossen werden. In CORBA wird dies sprachneutral durch eine IDL-Beschreibung vorgenommen, bei den Enterprise JavaBeans erfolgt die Änderung aus Sicht der Beteiligten durch das Home-, wahrscheinlicher jedoch durch das RemoteInterface.
2. Der Mechanismus der generierten Netzsoftware im Sinne von einem Stub für den Klienten und einem Skeleton für den Server, um für transparente Kommunikation zu sorgen, ist bei beiden Technologien weitestgehend identisch. Im Sinne von CORBA wird hier mit einer interoperablen Objektreferenz gearbeitet, im Sinne der Enterprise JavaBeans handelt es sich um ein Stück Software, das zur Laufzeit dynamisch gebunden wird. Die transparente Kommunikation zwischen dem Stub und dem Skeleton ist in beiden Fällen gewährleistet; der Klient bekommt den Eindruck vermittelt, als wenn er lokal arbeiten würde. Sowohl die interoperable Objektreferenz als auch das Handle für die entfernte Kommunikation für Enterprise JavaBeans kapseln die Lokation des tatsächlichen Objektes bzw. der tatsächlichen Komponente.
3. Sowohl das CORBA-Objekt als auch das Bean repräsentieren einen Teil einer Geschäftslogik, die prinzipiell in den Bereich der Individualsoftware fallen. Aufgrund der Standardisierung durch die OMG für CORBA und die Spezifikation der Enterprise JavaBeans durch SUN Microsystems werden diese Softwarebausteine bei richtiger Erstellung portabel sein. D. h., die Heterogenität der Hardware und der verschiedenen Betriebssysteme ist für einen Entwickler grundsätzlich transparent. Speziell im CORBA-Umfeld ist bei Einhaltung der Regeln über die Sprachabbildung die Objektimplementierung auch über verschiedene CORBA-Produkte hinweg portabel. Für die Registrierung der Objekte, die letztendlich in einem Prozess residieren, können durchaus unterschiedliche Vereinbarungen gelten. Von Seiten der OMG ist dies zu keinem Zeitpunkt spezifiziert worden.

   Bei den Enterprise JavaBeans ist ebenfalls die Beanimplementierung grundsätzlich portabel. Die unterschiedlichen, proprietären Deployment Deskriptoren und die Unterschiede in der Generierung der Zusatzsoftware verhindern jedoch einen Wechsel auf einen anderen Applikationsserver. Die Portabilität von Enterprise JavaBeans zwischen verschiedenen EJB-Containern kann nur durch ein neues Deployment, also durch die Anpassung der Deployment Dekriptoren, gefolgt von einem neuen Kompilierlauf gewährleistet werden.
4. CORBA erlaubt im Gegensatz zu Enterprise JavaBeans neben einem statischen auch einen dynamischen Zugang. Im CORBA-Umfeld ist es hierfür ausreichend, die Objektreferenz eines entfernten Objektes zu kennen, um über ein geeignetes Interface Repository Informationen über den Aufbau dieses Objektes, also
   a. Methodennamen,

b. Signaturen,
c. Rückgabewerte,
d. Aufbau der Eingabedaten und
e. mögliche Ausnahmen

zu erhalten. Basierend auf diesen Informationen wird ein Request-Pseudo-Objekt aufgebaut, mit den Eingabedaten gefüllt und synchron bzw. asynchron über die dynamische Schnittstelle an das Skeleton der Serverseite geschickt.

14.2 Unterschiede der Technologien

Ein gravierender Unterschied beider Technologien ist der Umstand, das CORBA erst mit der Spezifikation 3 über Komponenten verfügen wird. Mit den CORBA-Produkten, basierend auf der Spezifikation 2.x, ist der Entwickler somit selbst verantwortlich für das Setzen und Schließen von Transaktionsklammern usw.

Die Unterschiede und Vorteile von CORBA gegenüber anderen Technologien sind:

1. Ein sehr großer und nicht zu unterschätzender Vorteil von CORBA ist die Sprachunabhängigkeit. Damit ist es grundsätzlich möglich, dass ein C++-Objekt mit CORBA-Objekten aller anderen unterstützten Sprachen kommuniziert. Durch die Sprachunabhängigkeit ist eine sehr gute Integration bestehender Altanwendungen möglich. Beispielsweise kann eine bestehende COBOL-Applikation auf einem Großrechner durch eine Kapselung für den äußeren Betrachter als CORBA-Objekt gesehen und CORBA-konform angesprochen werden. Für Sprachen, die von der OMG zurzeit nicht unterstützt werden, kann mit entsprechendem Aufwand ebenfalls eine Sprachabbildung definiert und ein IDL-Kompilierer entwickelt werden. Ein Beispiel hierfür ist ein PL/1-IDL-Kompilierer für eine Schweizer Bank, die ihre PL/1-Applikationen mit geringem Aufwand CORBA-konform gemacht hat und somit die Widerverwendbarkeit erreicht hat. Durch die Beschreibungssprache XML ist zwischenzeitlich eine sehr gute Alternative zum Datenaustausch zwischen neuen und alten Anwendungen gegeben. Andere Lösungsansätze, wie das Definieren eigener Protokolle, wie z. B. den Java-Native Code, der nach dem Big Endian-Prinzip arbeitet, ist vorstellbar. Jedoch besteht der Charme der CORBA-basierten Lösung in seiner programmtechnischen Einfachheit und der leichten Wartbarkeit.
2. Skalierbarkeit: CORBA-Objekte residieren in Prozessen. Damit ist eine feingranulare Verteilung von Objekten mit kleinen und schlanken Prozessen möglich, die durchaus auf verschiedenen Rechnern platziert sein können. Ist ein Objekt nicht aktiv, so kann der korrespondierende Prozess von dem Betriebssystem sofort suspendiert werden und belastet die CPU nicht. Ein Wechsel zwischen den einzelnen Prozessen ist bei kleinen und schlanken Prozessen nicht sonderlich kompliziert und zeitaufwendig. Das Betriebssystem UNIX ist dafür geradezu prädestiniert. Enterprise JavaBeans werden dagegen in einer virtuellen Maschine instanziiert. Diese virtuelle Maschine ist aus Sicht eines Betriebssystems ein monolithischer Block. Wenn einige der Enterprise JavaBeans ggw. nicht benötigt werden, kann diese virtuelle Maschine von dem Betriebssystem trotzdem nicht ausgelagert bzw. die Objekte suspendiert werden.
3. Stabilität der Standards: Der von der OMG verabschiedete Standard ist bisher weitestgehend stabil gewesen. Es wurde weitestgehend auf eine Abwärtskompatibilität Wert gelegt. Bei den Enterprise JavaBeans ergibt sich eine andere Situation:
   a. In der EJB-Spezifikation 1.0 durften Enterprise JavaBeans multithreading-fähig sein. In der Spezifikation 1.1 und nachfolgenden ist dies untersagt. Damit ergibt sich für einen Entwickler das Problem, seine EJBs programmtechnisch entsprechend auf die neuen Spezifikationen umzusetzen.
   b. In der EJB-Spezifikation 1.1 waren bei EntityBeans, die nach dem Prinzip des Container-Managed-Persistence entwickelt wurden, alle Datenmember mit der Sichtbarkeit public versehen. In der EJB-Spezifikation 2.0 wurde richtigerweise festgelegt, dass die Beanimplementierung als eine abstrakte Klasse zu erfolgen hat und die Da-

tenmember in der Beschreibungsdatei ejb-jar.xml zu hinterlegen sind. Die abstrakte Beanimplementierungsklasse deklariert lediglich, dass nach einem fest vorgegebenen Namensschema, Zugriffsmethoden, mit denen auf die Datenmember lesend oder schreibend zugegriffen wird. Für einen Entwickler von Container-Managed-Persistence-EntityBeans bedeutet dies ebenfalls eine Anpassung seiner Implementierung, um der neuesten Spezifikation zu genügen.

c. Für CMP-EntityBeans werden in beiden Spezifikationen 1.1 und 2.0 die benötigten find-Methoden generiert. In beiden Fällen wird in den Beschreibungsdateien hinterlegt, wie die Methoden heißen, welche Signatur sie besitzen und wie ein Vergleich stattfinden soll. In der Spezifikation 1.1 war dies jedem Hersteller selbst überlassen. Entsprechend wurden diese Beschreibungen durch die proprietäre zweite XML-Beschreibungsdatei mit ebenfalls proprietären Mitteln vorgenommen. In der EJB-Spezifikation 2.0 wird verbindlich eine EJB-QL (**E**nterprise **J**ava**B**eans **Q**uery **L**anguage) vorgegeben. Aufgrund dieser Standardisierung sind die Vergleiche und Methodenbeschreibungen jetzt in der standardisierten XML-Datei ejb-jar.xml vorzunehmen. Auch hier wird ein Entwickler seine bestehenden Beschreibungsdateien anpassen müssen, um dem neuesten Standard zu genügen.

d. In der Spezifikation 1.1 war es bei Bean-Managed-Persistence EntityBeans erlaubt, die Transaktionsverarbeitungen programmtechnisch selbst vorzunehmen. In der Spezifikation 2.0 ist dies untersagt. Alle EntityBeans müssen nach dem Prinzip des Container-Managed-Transaction arbeiten, also die Transaktionsverarbeitung in der Beschreibungsdatei ejb-jar.xml bekannt geben und durch die generierte Software vornehmen lassen.

   Es ist sehr zu hoffen, dass zukünftige Spezifikationen keine grundlegenden Veränderungen, die einen zusätzlichen Eingriff in den Quellkode oder die Beschreibungsdateien verlangen, vereinbaren werden.

4. Durch den Einsatz von maschinennahen Sprachen wie C und C++ kann ein höherer Durchsatz der Verarbeitung erreicht werden als mit interpretierenden Sprachen wie bspw. Java.
5. Viele Datenbanken lassen im Hinblick auf Sprachen wie C oder C++ eingebettetes SQL zu. Damit können Datenbankzugriffe deutlich performanter sein als bei Einsatz von zusätzlichen Treibern wie ODBC oder JDBC.

Die Nachteile von CORBA gegenüber den Enterprise JavaBeans sollen ebenfalls nicht verschwiegen werden:

1. Der Einsatz der Programmiersprache C++ beinhaltet eine recht komplizierte Sprachabbildung von IDL auf diese Zielsprache. Erfahrungsgemäß werden Entwickler mit langjähriger Erfahrung in C++ diese Sprachabbildung leicht beherrschen. Entwicklern, die sich dieser Sprache nicht so sicher sind, werden eine Reihe von potentiellen Fehlerquellen geboten.
2. Der Umgang mit dynamischen CORBA-Objekten ist ebenfalls recht kompliziert und bedingt eine sehr sorgfältige Entwicklung der Software.
3. Die Eigenschaften der EJB-Container, Objekte in einem Pool zu verwalten oder einen Cache für EntityBeans bzw. zustandsbehafteten SessionBeans zur Verfügung zu stellen, entfällt im CORBA-Umfeld und muss ggf. durch den Entwickler selbst geleistet werden. Diese Nachteile werden jedoch, wie oben erwähnt, durch die Umsetzung der CORBA-3 Spezifikation obsolet sein.

14.3 Die Gefahren von Komponenten

Die Komponententechnologie erlaubt die deskriptive Beschreibung von zu generierender Software durch externe Beschreibungsdateien. Damit ist potentiell die Möglichkeit gegeben, dass Entwickler ohne tiefgehende Kenntnis von Datenbanken oder Transaktionsverarbeitung

Software entwickeln und Zusätze generieren lassen, die Datenbankanfragen oder Transaktionsverarbeitung durch generierte Software abarbeiten lassen. Durch eine schlechte Einstellung, z. B. von den Transaktionspolitiken, kann ein ungünstiges Laufzeitverhalten und eine fehlerbehaftete Abarbeitung einer Anfrage verursacht werden. Da sich die gesamte Applikation in mehreren jar- bzw. war-Archiven befinden können ist der Aufwand, eine Verifikation beim Einspielen der Software vorzunehmen, entsprechend hoch. Zur Zeit existieren keine kommerziellen Werkzeuge, die eine Verifikation der Deployment Deskriptoren, sowohl im Einzelfall, als auch bei mehreren Paketen, vornehmen. Anhand eines Beispieles soll ein gravierender Fehler bzgl. einer falschen Transaktionspolitik dies belegt werden:

In der folgenden Abbildung wird dargestellt, dass eine Kaskade von Komponenten, die alle die gleiche Transaktionspolitik *RequiresNew* aufweisen, durch eine starke Belastung der generierten Software die Verarbeitungsgeschwindigkeit stark herabsetzt und unter bestimmten Voraussetzungen sogar zu einem Fehler führen kann.

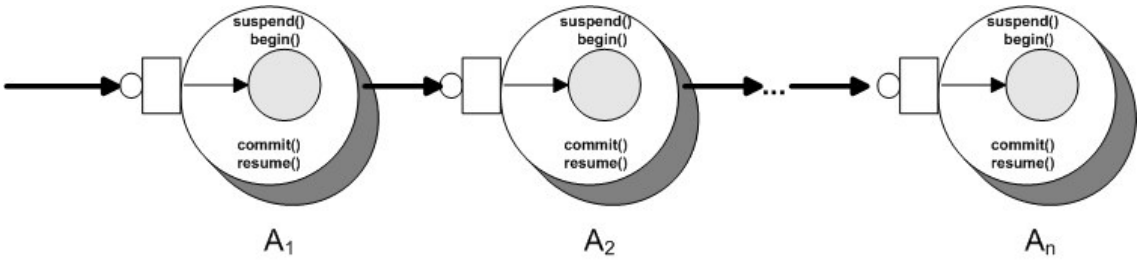


Abbildung 36: Softwarekomponente bei schlechter externer Beschreibung

Hypothetisch werde die folgende Situation angenommen:
1. Alle beteiligten Beans sind durch die externe Beschreibung mit der Transaktionspolitik RequiresNew versehen.
2. Eine Java-Applikation, ein Servlet oder eine JavaServer Page setzt eine Transaktionsklammer und ruft Bean A_1 an.
3. Bean A_1 tätigt einen Methodenaufruf in Bean A_2, A_2 ruft A_3 an usw..

In diesem Szenario ist das folgende Verhalten zu verzeichnen:
1. Die von dem Klienten gesetzte Transaktionsklammer wird von dem EJB-Container suspendiert, eine neue Transaktion wird initiiert und der Aufruf an das Bean A_1 weitergeben.
2. Bean A_1 ruft eine Methode von Bean A_2 an. Der EJB-Container wird die zuvor gesetzte Transaktion suspendieren, eine neue Transaktion initiieren und den Aufruf an das Bean A_2 weitergehen usw.
3. Das letzte Bean in der Kaskade wird angesprochen und nach dessen Verarbeitung wird der EJB-Container die zuletzt geöffnete Transaktionsklammer schließen und die vorletzte Transaktion wieder aufnehmen.
4. Diese Tätigkeiten werden fortgesetzt, bis der Erzeuger der zuerst initiierten Transaktion die Kontrolle zurückbekommt.

Damit verbunden ist ein entsprechend hoher Aufwand.
1. Eine bestehende Transaktion muss von dem Nachfolger suspendiert und eine neue Transaktion initiiert werden.
2. Die neuen Transaktionen müssen buchhalterisch festgehalten werden.
3. Eine Sperre muss auf die beteiligten Objekte gelegt werden.
4. Der jeweilige Erzeuger der neuen Transaktion muss nach Beendigung der Transaktion eine Bestätigung (commit) des Ergebnisses oder eine Ablehnung des Ergebnisses (rollback) aussprechen.

5. Der im Hintergrund laufende Transaktionsmanager muss diese Transaktion mitprotokollieren und nach erfolgreichem Abspeichern des neuen Zustandes die protokollierte Information wieder entfernen.
6. Diese Aktivitäten werden bei allen Vorgängern solange wiederholt, bis der ursprüngliche Erzeuger wieder die Kontrolle über die zuerst gesetzte Transaktion erhält.

Damit wird deutlich, dass bei unsachgemäßer Beschreibung von Komponenten ein extrem schlechtes Laufzeitverhalten verursacht werden kann.

Erschwerend kommt hinzu, dass der ursprüngliche Initiator der ersten Transaktion das Ergebnis verwerfen kann (rollback). In diesem Fall ist nicht mehr möglich, dass die anderen Beteiligten den alten Zustand wieder herstellen, da jede Zwischenstation ihr Ergebnis bereits als erfolgreich gekennzeichnet und einen neuen Zustand in den persistenten Speicher zurück geschrieben hat. Dadurch entsteht eine Fehlersituation, die bestenfalls durch geeignete inverse Operationen korrigiert werden kann.

Die automatisierte Transaktionsverarbeitung arbeitet laut Spezifikation im Standardverhalten bei der Auslösung von Ausnahmen wie folgt:

1. Wird von dem EJB-Container eine Transaktionsklammer gesetzt, so ist der Erzeuger, also der Container auch verpflichtet, diese Klammer wieder zu schließen.
2. Wird während der Verarbeitung in einem Bean eine Ausnahme ausgelöst, so sind zwei Fälle zu unterscheiden:
   a. Handelte es sich um eine Systemausnahme, also eine Exception, die von RemoteException oder RuntimeException abgeleitet wurde, so schließt der Container die Transaktion durch eine Verwerfung des Ergebnisses (rollback) und propagiert diese Ausnahme zurück an den Aufrufer des Beans.
   b. Handelte es sich um eine benutzerdefinierte Ausnahme, also eine Ausnahme die von Throwable oder Exception abgeleitet wurde, so wird von dem Container das Ergebnis der Verarbeitung positiv bestätigt (commit) und die Ausnahme wird an den Aufrufer weitergeleitet.

   Soll dieses Standardverhalten durchbrochen werden, muss ein Bean, bevor es eine benutzerdefinierte Ausnahme auslöst, über den EJB-Kontext durch das Setzen von setRollbackOnly() den Container zwingen, auch in diesem Fall den alten Zustand herzustellen.

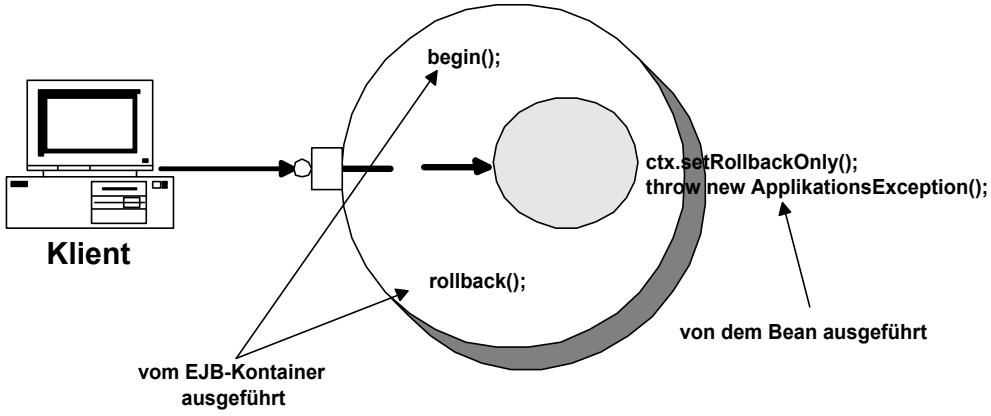


Abbildung 37: Ablehnung eines Ergebnisses durch den EJB-Container

Auch hier wird deutlich, dass ein Entwickler eine Komponente genau verstehen muss, um unerwünschte Seiteneffekte zu vermeiden. Ein weniger qualifizierter Entwickler wird u. U. Software entwickeln, die nach dem Standardmuster der generierten Software arbeitet und möglicherweise im Widerspruch zur Geschäftslogik steht.

Weitere Gefahren von Komponenten, die bei unsachgemäßer Beschreibung der generierten Zusatzsoftware entstehen können sind:

1. Bei Verwendung von Container-Managed-Persistence EntityBeans ist das Standardverhalten derart, dass alle Java-Datenmember auf genau eine Spalte einer Datenbanktabelle abgebildet werden. Damit sind notwendige Kriterien im Umgang mit Datenbanken wie die erste, zweite und dritte Normalform nicht gegeben. Die damit eventuell verbundene Redundanz von Datensätzen kann leicht zu Inkonsistenzen der Daten führen.
2. Die in der EJB-2.0 Spezifikation verabschiedeten Beziehungstypen sind in der Praxis noch nicht bewährt. Damit ist es grundsätzlich möglich, einen Objektgraphen aufzubauen, wobei jedes Bean seine Daten nach wie vor in genau eine Spalte der Tabelle schreibt. Bei sachgemäßer Verwendung können jedoch die benötigten Normalformen, aber auch alle Vereinigungen von Spalten aus verschiedenen Tabellen nachgebildet werden. Aufgrund der Bildung derartiger Beziehungsgeflechte, die durch eine Menge von Objekten in Form von Container-Managed-Persistence EntityBeans repräsentiert werden, ist ein erhöhter Aufwand an Speicher zu sehen. Das Laufzeitverhalten wird durch den höheren Kommunikationsaufwand zwischen den beteiligten Komponenten i. A. ungünstiger als bei Verwendung von selbst entwickelter Software.

   Die Verwendung von Beziehungen in der EJB-2.0 Spezifikation bedingt, dass neben den Datenmembern der EntityBeans auch alle Beziehungsfelder in den Deployment Deskriptoren beschrieben werden. Dies beinhaltet neben der Abbildung der Datentypen auf Datenbanktypen auch die Festlegung von Rollen, in welcher Vielfachheit Rollen zueinander in Beziehung stehen usw. Damit ergibt sich für einen Entwickler die Notwendigkeit, eine Vielzahl von Einträgen deskriptiv in den XML-Beschreibungsdateien zu hinterlegen. Ohne ein gutes Entwicklungswerkzeug setzt dies sehr tiefgehende Kenntnisse in XML voraus und ein Detailwissen über den syntaktischen Aufbau der jeweiligen Dokumente.

   Da bei den Container-Managed-Persistence EntityBeans und somit auch bei den Beziehungen der Beans zueinander die Informationen in drei verschiedenen XML-Dateien zu beschreiben sind, ergibt sich die Gefahr der Unübersichtlichkeit und der Inkonsistenz der Informationen. Die Fehlersuche ist auch hier als wesentlich höher anzusehen, als bei entwickeltem Programmierkode.
3. Die deskriptive Beschreibung für die zu generierende Zusatzsoftware ist ein statischer Prozess. Jede Änderung in den Deployment Deskriptoren bedingt einen neuen Kompilierlauf.
4. Enterprise JavaBeans konzentrieren sich auf genau eine Programmiersprache, nämlich Java. Die Einbindung von Altanwendungen muss über geeignete Konnektoren erfolgen, die ggw. von Sun Microsystems standardisiert werden. Inwieweit sich diese Technologie durchsetzen wird, ist abzuwarten.

Zusammenfassend kann dargelegt werden, dass beide Technologien viele Gemeinsamkeiten aufweisen und jede für sich auch Vorzüge hat. Der Einsatz muss in starker Abhängigkeit von der Anwendung und weiteren Rahmenbedingungen wie z. B. die Integration von Altanwendungen gesehen werden. Die Verknüpfung beider Technologien ist durch das Protokoll RMI-over-IIOP jederzeit möglich.

Es ist somit vorstellbar, dass eine Internetapplikation für den Zugang der Klienten einen Webserver mit Servlets und JavaServer Pages zur Verfügung stellt, seine Geschäftslogik jedoch aus Enterprise JavaBeans und CORBA-Objekten besteht.

Aufgrund der Ähnlichkeit der Vorgehensweise von entfernten Methodenaufrufen, deren Unterschied lediglich in der Unterbrechung des Aufrufs durch den Container zu sehen ist, kann bei sachgemäßer Konfiguration davon ausgegangen werden, das ein Enterprise Java Bean die gleiche Verarbeitungsgeschwindigkeit aufweist wie ein Java-CORBA-Objekt.

15 COM, DCOM und COM+

Mit DCOM bzw. COM+ entstand fast zeitgleich ein Microsoft Pendant zu CORBA. In den nachfolgenden Kapiteln wird eine kurze Übersicht zu COM/DCOM vorgenommen. Eine sehr gute Kurzeinführung hierzu findet sich bei [Emsenhuber, 2002] und [Microsoft, 2002]. Eine sehr ausführliche Behandlung von COM/DCOM ist [Abernethy et al., 1999]. In [Barke, 1996] findet sich eine exzellente Einführung in die Technologien CORBA und COM/OLE inklusive eines Vergleichs der beiden Technologien.

Mit der Initiative DNA (**D**istributed Inter**n**et **A**rchitecture) hat Microsoft eine Reihe verschiedener Technologien zu einer Einheit zusammengefasst und analog zur J2EE-Spezifikation eine Vorgehensweise bzgl. der Architektur verteilter Anwendungen beschrieben. Mit diesen zum Teil recht unterschiedlichen Komponenten kann eine verteilte Anwendung unter den Gesichtspunkten Skalierbarkeit und Hochverfügbarkeit aufgebaut werden. Durch den IIS (**I**nternet **I**nformation **S**erver) von Microsoft und den ASPs (**A**ctive **S**erver **P**ages) ist auch der Zugang über das Internet möglich (vgl. [Emsenhuber, 2002]).

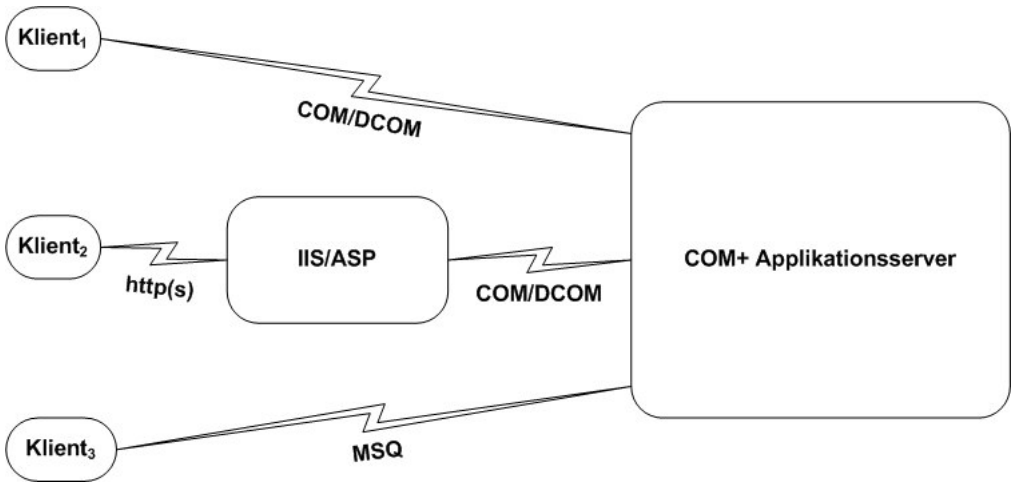


Abbildung 38: Klientzugänge zu DCOM-Applikationen

Die für diese Arbeit relevanten Komponenten sind die Möglichkeit der synchronen verteilten Verarbeitung durch einen RPC auf Basis von DCOM, die asynchronen Möglichkeiten auf Basis von MSMQ (**M**icro**s**oft **M**essage **Q**ueue) und die Möglichkeiten, in beiden Fällen Transaktionen verwenden zu können.

COM (**C**omponent **O**bject **M**odel) ist eine propritäre Entwicklung von Microsoft zum Datenaustausch zwischen Komponenten. Im Gegensatz zu CORBA und J2EE ist COM sowohl eine Spezifikation als auch eine Implementierung, die als Framework für die Integration von Komponenten gilt.

COM definiert eine Anwendungschnittstelle (API), die zur Erzeugung und/oder der Integration von verschiedenen Komponenten in benutzerdefinierten Applikationen dient.

Damit Komponenten miteinander kommunizieren können, muss eine von Microsoft vorgegebene binäre Struktur strikt eingehalten werden. Ist diese Voraussetzung erfüllt, so können Komponenten, die u. U. in unterschiedlichen Programmiersprachen entwickelt wurden, miteinander kommunizieren und sich ggf. synchronisieren. COM kann nur auf einem lokalen

Rechner zwischen zwei Komponenten, die sich durchaus in verschiedenen Adressräumen befinden können, Nachrichten austauschen.

DCOM (**D**istributed **C**omponent **O**bject **M**odel) erweitert COM um Netzwerkfunktionalität. Damit ist die Möglichkeit gegeben, verteilte Anwendungen über mehrere physikalische Rechner zu entwickeln.

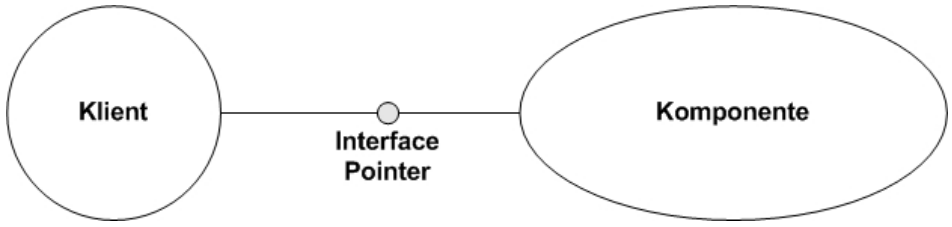


Abbildung 39: Klient-Server-Kommunikation DCOM

COM und DCOM stellen die unteren Schichten einer verteilten Anwendung dar, während OLE (**O**bject **L**inking **E**mbedded), ActiveX und MTS (**M**icrosoft **T**ransaction **S**erver) eine obere Schicht, basierend auf COM/DCOM, darstellen.

COM+ integriert die wichtigen Komponenten MTS und ein Message Passing System in den Technologien COM/DCOM.

COM/DCOM ist lauffähig unter den Microsoft Betriebssystemen Windows 95/98, Windows NT, Windows 2000/2003 und Windows XP. Von Seiten Microsoft existiert eine Portierung für MacOS-Betriebssysteme und die Software AG hat für eine Reihe von UNIX/LINUX-Derivaten eine Portierung vorgenommen. Dies beinhaltet insbesondere die Derivate SUN-Solaris, AIX, True64 und LINUX.

15.1 Historie

Microsoft hat mit dem DCOM (**D**istributed **C**omponent **O**bject **M**odell) das Rad nicht neu erfunden, sondern DCOM ist vielmehr das Ergebnis aus eigener Entwicklung und den Erfahrungen aus OLE (**O**bject **L**inking and **E**mbedding) und DDE (**D**ynamic **D**ata **E**xchange), aber auch die Erfahrungen und Ideen aus den Arbeiten/Spezifikationen der Open Software Foundation und der Object Management Group.

Man muss vorausschicken, dass MS-DOS ursprünglich darauf ausgelegt war, dass genau eine Anwendung auf einem Computer laufen konnte. Erst mit Entwicklung leistungsfähigerer Prozessoren (i80286 und i80386) begann Microsoft an das gleichzeitige Laufen mehrerer Anwendungen auf einem Computer zu denken. Mit der Einführung von Windows gab es erstmals eine Multitasking-Umgebung, die bei den Anwendern den Wunsch nach Datenaustausch zwischen den unterschiedlichsten Anwendungen aufkommen ließ. Diesem Wunsch wurde mit der Einführung der Zwischenablage und DDE (**D**ynamic **D**ata **E**xchange) Rechnung getragen. DDE war jedoch für Software-Entwickler zu kompliziert, so dass es nur sehr wenige Anwendungen gibt, die DDE erfolgreich implementieren.

Im Gegensatz zu DDE ist die Zwischenablage wesentlich einfacher für die Software-Entwickler aufgebaut. Aufgrund des einfachen Copy-Paste-Prinzips gilt die Zwischenablage heute noch als großer Erfolg und wird von der Microsoftgemeinde akzeptiert.

Ein Problem konnte die Zwischenablage jedoch nicht lösen: Wie verknüpft man beispielsweise ein Winword-Dokument mit einem Excel-Spreadsheet? Die Zwischenablage ist zu un-

flexibel, um Dokumente und deren Inhalte miteinander verknüpfen zu können. Sobald in einem eingebetteten Dokument eine Änderung notwendig wird, muss dieses bearbeitet, in die Zwischenablage kopiert und letztendlich in das Zieldokument eingefügt werden. Microsofts Antwort auf diese Problemstellung heißt OLE (**O**bject **L**inking and **E**mbedding).

OLE kam erstmals mit Windows 3.1 auf den Markt. OLE 1 sollte eine geeignete Schnittstelle bieten, um mit verknüpften Dokumenten besser umgehen zu können. Um in ein Word-Dokument ein Excel-Spreadsheet einzubetten, war es mit DDE kein Kunststück mehr. OLE 1 ermöglichte es jedoch, in einem Excel-Spreadsheet Daten zu ändern und sorgte dafür, dass diese Änderungen auch in jene Dokumente übernommen wurden, in die das betreffende Excel-Spreadsheet eingebettet war. Das eingefügte Dokument konnte sogar durch Doppelklick geöffnet werden. OLE 1 basierte trotz neuem Namen in Wirklichkeit wieder auf DDE, welches das Protokoll für die Kommunikation zwischen den einzelnen Prozessen war.

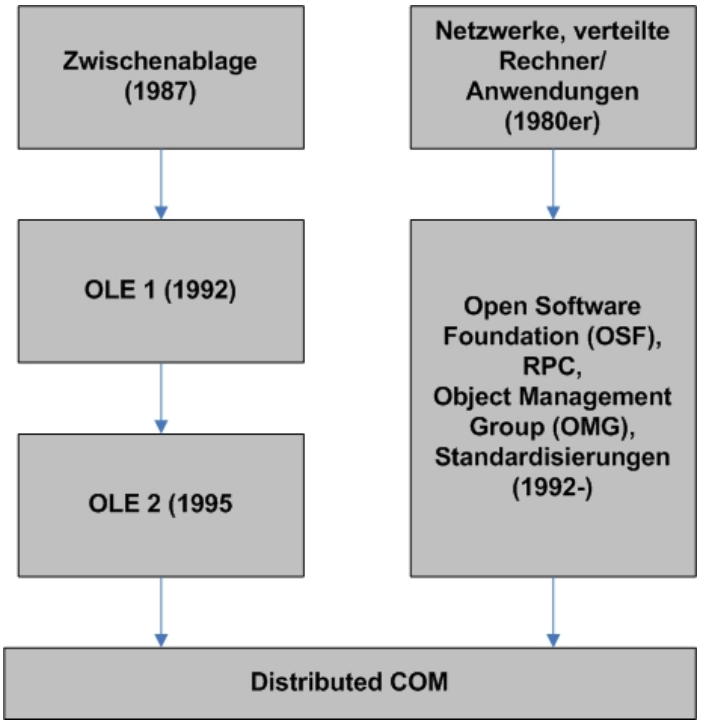


Abbildung 40: Historie DCOM

Die für damalige Zeit revolutionären Ansätze von OLE 1 wurden mit der Zeit weiter verfeinert, bis man erkannte, dass eingebettete Dokumente (Objekte) kleine Softwarekomponenten sind, die in jede beliebige Anwendung eingebaut werden können und die die Funktionalität der betreffenden Anwendung erweitern.

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

15.2 Technischer Überblick

Eine COM/DCOM-Komponente kann eine beliebige Anzahl von Schnittstellen implementieren. Analog zu CORBA ist eine Schnittstelle eine Gruppierung von Methoden und ggf. Datenstrukturen. Für die Deklaration von Schnittstellen wird die **M**icrosoft **I**nterface **D**efinition **L**anguage (MIDL), die eine Erweiterung der DCE IDL darstellt, verwendet.

Im Gegensatz zu CORBA IDL sind keine Module als Namensräume bekannt. Um Namenskollisionen zu vermeiden, muss jedes Objekt und jede Schnittstelle einen eindeutig bestimmten Identifizierer aufweisen. Analog zu DCE liefert Microsoft hierfür die Werkzeuge *uuidgen* bzw. *guidgen*. Dieser eindeutig bestimmte Identifier wird in der Regel mit UUID (**U**niversal **U**nique **ID**entifier) bezeichnet. Es handelt sich hierbei um eine 128 Bit große Zahl, die einer Schnittstelle oder einem Objekt zugeordnet wird.

Bestimmt wird diese UUID durch eine Kombination von Informationen, die sich auf Hardware-Adresse, einen Zeitstempel und eine Zufallszahl beschränken.

In der UUID existiert
1. eine Referenz auf die Hardware-Adresse, i. d. R. die MAC-Adresse der ersten Netzwerkkarte des Rechners, auf dem die UUID generiert wurde.
2. als zweite Komponente ein Zeitstempel.
3. als dritte Komponente eine Zufallszahl.

Ein Beispiel für einen eindeutig bestimmten Identifizierer liefert der Aufruf von *uuidgen* durch den Wert

```
52454bd2-3071-4041-bbf4-3601e8a9eb5b.
```

Dieser Mechanismus wurde von Microsoft vollständig von der Open Software Foundation übernommen, die in ihrer Spezifikation u. a. das Distributed Computing Environment standardisiert haben.

In der IDL-Beschreibung wird dieser eindeutig bestimmte Identifizierer zu Beginn der Datei eingestellt.

```
[
   uuid(52454bd2-3071-4041-bbf4-3601e8a9eb5b),
   version(1.7)
]
```

Analog zu CORBA IDL können in einer IDL-Beschreibung mehrere Schnittstellen deklariert werden und ein einzelner Serverprozess kann die Implementation beliebig vieler Objekte/Komponenten beinhalten.

Wie in Abbildung 41 (vgl. [SEI]) angegeben, existieren für einen Klienten drei Möglichkeiten, um mit COM/DCOM-Objekten zu arbeiten.
1. Lokales Serverobjekt: Der Klient kann sich direkt über eine Bibliothek an den Serverprozess/das Serverobjekt verbinden. Sowohl der Klient als auch der Server leben in dem gleichen Adressraum und die Kommunikation erfolgt über Funktionenaufrufe.
2. Lokales Proxyobjekt: Der Klient kontaktiert ein Serverobjekt in einem anderen Adressraum, aber auf der gleichen physikalischen Maschine. Die Kommunikation erfolgt durch leichtgewichtige Interprozesskommunikationsmechanismen.
3. Entferntes Proxyobjekt: Der Klient kontaktiert ein entferntes Objekt, das i. d. R. auf einer anderen physikalischen Maschine residiert. Die Netzkommunikation zwischen dem Klienten und dem Server erfolgt durch den adaptierten DCE RPC. Dieser Remote Procedure Call Mechanismus wurde von Microsoft sehr früh adaptiert und wird im Betriebssystemumfeld (Windows NT, Windows 2000/2003 und Windos XP) durchgängig verwendet.

Befinden sich der Klient und der Server in dem gleichen Adressraum, so ist die gemeinsame Nutzung der Daten trivial. Ist der Serverprozess von dem Klienten getrennt, so muss COM/DCOM analog zum CORBA-Umfeld die zu übertragenden Daten in ein neutrales Format verpacken bzw. entpacken. Diese Aufgabe wird ebenfalls analog zu CORBA von den generierten Komponenten Stub und Skeleton wahrgenommen, die in der DCOM-Terminologie jedoch mit Proxy und Stub, respektive, bezeichnet werden.

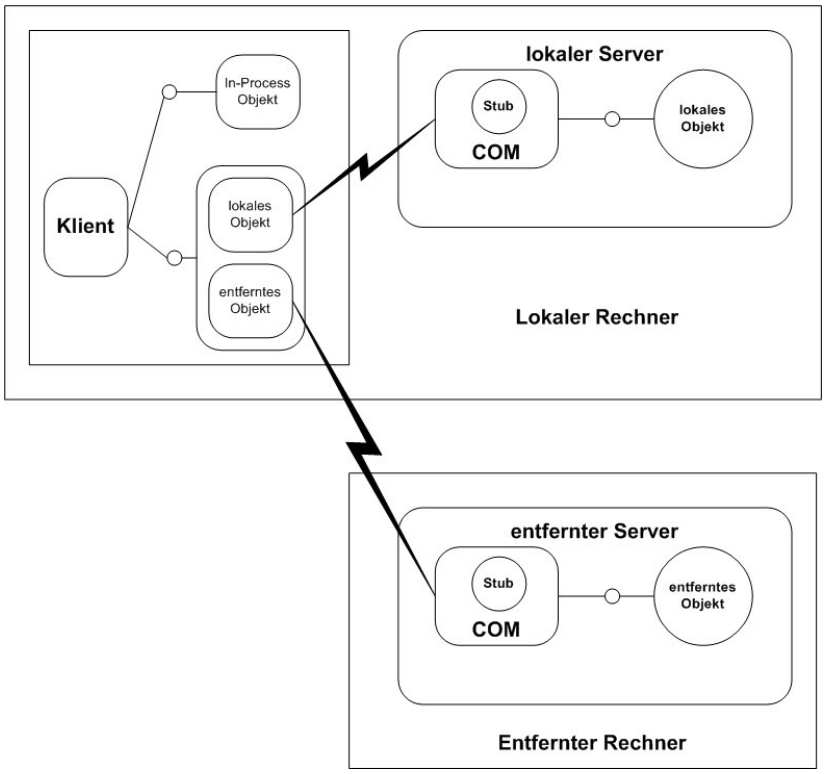


Abbildung 41: Klient-Server-Kommunikation in DCOM (vgl. [SEI])

Die COM/DCOM-Komponenten werden im Gegensatz zu CORBA nicht in einem Namensdienst, einer Fabrik (FactoryFinder) oder einem Trader gespeichert, sondern in einer Komponentendatenbank, i. d. R. die Windows-Registry, registriert.

Es ist auch nicht möglich, die Objektreferenz in einen String zu wandeln und die Referenz so dem Klienten als *stringified IOR* zugänglich zu machen.

Der Klient geht, um ein Serverobjekt zu ermitteln wie folgt vor:
1. Über das COM API instanziiert der Klient ein neues COM-Objekt.
2. Dieses neu erzeugte COM-Objekt ermittelt die Objektimplementierung und instanziiert den Serverprozess.
3. Der Server erzeugt ein Objekt und liefert einen Zeiger auf die Schnittstelle des geforderten Objektes zurück.
4. Der Klient kommuniziert über diesen Referenzzeiger mit dem Serverobjekt.

Die folgende Abbildung (vgl. [SEI]) illustriert diesen Vorgang:

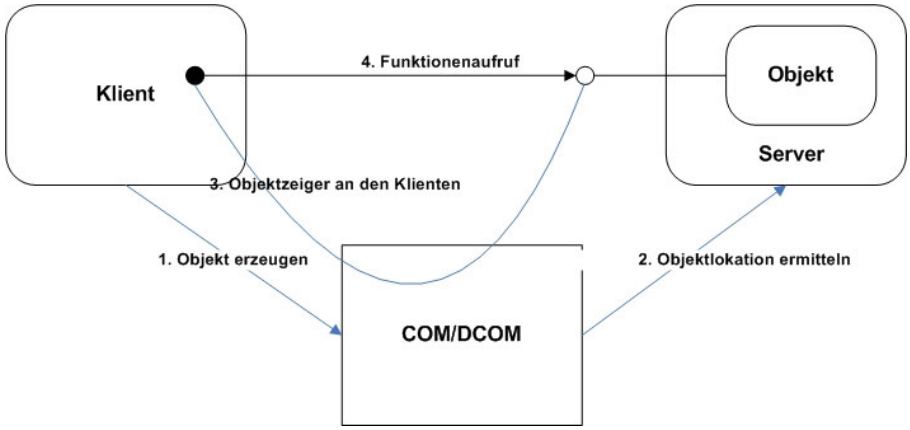


Abbildung 42: Objektermittelung mit DCOM

Ein wichtiger Aspekt von COM ist, dass Objekte keine Identität besitzen. Ein Klient kann ein zustandsloses COM-Objekt anfragen, aber nicht ein bestimmtes Objekt anfordern. Jedes Mal, wenn der Klient nach einem Objekt fragt, wird über das COM-API ein neues Objekt instanziiert. Ein Vorteil dieser Regelung ist, dass COM Objekte bereits vorrätig halten kann und eine beliebige Instanz an den Klienten gibt.

Sobald ein Klient seine Arbeit mit dem entfernten Objekt beendet hat, wird dieses Objekt von COM einem Pool zugeführt.

Bei zustandsbehafteten Objekten ist es unerlässlich, dass der Klient dieses Objekt exklusiv besitzt und mit nachfolgenden Aufrufen mit dem vom ihm gesetzten Zustand arbeitet. Dazu existieren in DCOM zwei Möglichkeiten:

1. Analog zu CORBA wird das entfernte Objekt im Rahmen einer globalen Transaktion angesprochen und erst mit einem expliziten commit oder rollback wird dieses Objekt wieder freigegeben.
2. Es wird ein so genannter *Moniker* verwendet. Ein *Moniker* ist ein spezielles COM-Objekt, das für die Erzeugung von zustandsbehafteten Objekten verwendet wird. Diese Objekte können zu einem späteren Zeitpunkt über ihren (Spitz-) Namen (Moniker) angesprochen werden.

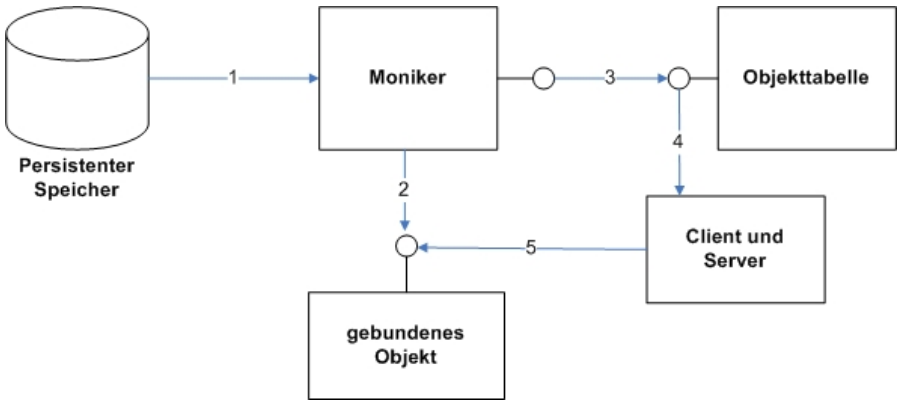


Abbildung 43: Arbeitsweise mit Monikern

Um mit Monikern zu arbeiten, muss man wie folgt vorgehen:

1. Aus einem anzugebenden persistenten Speicher wird ein Moniker erzeugt.
2. Das gewünschte Objekt wird an den Moniker gebunden.
3. Der Moniker wird in die Objekttabelle geladen.
4. Der Moniker und das gebundene Objekt können über einen vereinbarten Namen aus dem persistenten Speicher geladen werden.
5. Klient und Server arbeiten mit dem gebundenen Objekt.

In der folgenden Tabelle sind weitere Eigenschaften von COM/DCOM-Objekten aufgelistet:

Typinformationen:	U. u. benötigen Klienten zur Laufzeit Informationen bzgl. des COM-Objektes. Diese Informationen werden vom IDL-Kompiler (midl) generiert und in einer Bibliothek gespeichert. COM stellt ein API zur Verfügung, mit denen diese Informationen abgefragt, bzw. mit denen über alle Metadaten traversiert werden kann. Dieser Ansatz entspricht im CORBA-Umfeld dem Interface Repository, mit der Einschränkung, dass diese Metadaten nur lokal, aber nicht entfernt, abfragbar sind.
Persistenz:	COM/DCOM stellt analog zu den zustandsbehafteten Session-Beans einen Mechanismus zur Verfügung, der den Zustand bei einer Phase der Inaktivität im Dateisystem zwischenspeichert.
Universeller Daten Transfer	Es besteht die Möglichkeit, Datenobjekte zu übertragen und bei Änderung des Datensatzes benachrichtigt zu werden.
Verbindungsobjekte	Darunter versteht man im DCOM die Möglichkeit, dass ein verteilter Rückruf vom Server zum Klienten möglich ist.

Tabelle 7: DCOM Eigenschaften

Microsoft hat schnell erkannt, dass COM/DCOM eine Reihe von Unzulänglichkeiten aufweist. Zu benennen wären zumindest die beiden folgenden Aspekte:

1. COM ist relativ kompliziert. Der Entwickler muss gemessen an anderen Standards eine Reihe von Aufgaben selbst wahrnehmen und über ein sehr tiefgehendes Wissen bzgl. der COM aufweisen.
2. COM/DCOM ist nicht robust genug für Unternehmensapplikationen. Wichtige Dienste wie Sicherheit, Transaktionsdienst, Lastverteilung und zuverlässige Kommunikation sind nicht Bestandteil von COM.

Aufgrund dieser beiden Unzulänglichkeiten ging Microsoft dazu über, eine neue Umgebung COM+ zur Verfügung zu stellen, in der die Komplexität der Technologie gekapselt und die Robustheit durch die Integration von MTS sichergestellt wurde. Zwischenzeitlich ist Microsoft auch von COM+ wieder abgerückt und stellt mit .NET eine Technologie vor, die einfacher zu handhaben ist und die dem Entwickler mehr Freiheiten zugesteht. Zusätzlich wird .NET nicht nur eine proprietäre Microsoft-Technologie sein, sondern durch seine Freigabe auch auf anderen Plattformen in vollem Umfang und ohne Binärspezifikation zur Verfügung stehen (s. Mono-Projekt).

15.3 Die vier Säulen von COM/DCOM

Bei der Entwicklung von COM/DCOM standen für Microsoft die üblichen Charakteristiken der Objektorientierung und der verteilten Verarbeitung im Vordergrund.

15.3.1 Wiederverwendbarkeit

Erstens sollten die einzelnen Komponenten wieder verwendbar sein, da Wartung sowie die Erweiterung von Komponenten in großen Anwendungen große Probleme mit sich bringen.

Ein Aufteilen in wieder verwendbare Komponenten ist naheliegend, die die Erstellung neuer Applikationen, Erweiterung sowie Wartung erleichtern. Der Vorteil von DCOM liegt darin, dass es Klassen-Code in binären Komponenten verteilt. Das heißt, DCOM-Komponenten können ohne Abhängigkeiten vom Sourcekode wiederverwendet werden. Diese Binärkompatibilität setzt jedoch die gleiche Hardware voraus. Im gleichen Zug müssen Software-Entwickler nicht mehr Algorithmen oder Teile ihres Quellkodes preisgeben, wenn sie Komponenten an eine Entwicklergemeinde verteilen wollen. Dass durch die Wiederverwendung binärer COM/DCOM-Komponenten Probleme vermieden werden können, die zur Kompilierzeit auftreten, ist ein positiver Seiteneffekt, der nicht zu vernachlässigen ist.

15.3.2 Schnittstellenorientierte Programmierung

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

15.3.3 Unabhängigkeit von der verwendeten Programmiersprache

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

15.3.4 Client/Server-Architektur

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

15.4 Schnittstellen

COM/DCOM unterscheidet sich von anderen Komponententechnologien dadurch, dass es mit unbekannten Elementen umgehen kann. Wird eine COM/DCOM-Komponente von einem Programm instanziiert, muss das Programm Information darüber bekommen können, welche Schnittstellen in der COM/DCOM-Komponente implementiert sind. Die Methode *QueryInterface* deklariert in *IUnknown* ist die Antwort auf diese Problemstellung. Um einem Entwickler immer die Schnittstelle *IUnknown* zur Verfügung zu stellen, gilt die Vereinbarung, dass jede COM/DCOM-Klasse *IUnknown* implementieren und jede COM/DCOM-Schnittstelle davon erben muss. Die Schnittstelle von *IUnknown* definiert sich wie folgt:

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

Programmbeispiel 89: Interface IUnknown

QueryInterface wird automatisch aufgerufen, wenn eine COM/DCOM-Klasse instanziiert wird. `riid` übergibt den GUID der angeforderten Schnittstelle. Existiert für den betreffenden GUID eine Implementierung, gibt diese über den Zeiger `ppv` einen vTable-Pointer zurück. Ein vTable (engl. virtual method table) ist ein Array von Zeigern. Jeder Zeiger verweist auf eine bestimmte Methode oder eine bestimmte Eigenschaft (property) im betreffenden Objekt.

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

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

`AddRef` und `Release` sind zwei Methoden in *IUnknown*, die für die Verwaltung und die Verweiszählung verantwortlich sind.

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

Wenn eine Schnittstelle von *IUnknown* erbt, wird diese als benutzerdefinierte Schnittstelle bezeichnet. Diese haben die Eigenschaft der *frühen Bindung* gemeinsam. Das heißt, dass der vTable eines Interface schon zur Übersetzungszeit bekannt ist. Frühe Bindung basiert auf einem Vertrag, der durch die Typbibliothek vorgegeben ist.

Es ist zu beachten, dass Komponenten bei Verwendung von früher Bindung schneller sind, da schon zum Zeitpunkt der Übersetzung bekannt ist, wo die Implementierung der verwendeten Methoden ist.

Der Vorteil schnellerer Anwendungen bei früher Bindung bringt den Nachteil mit sich, dass die Implementierung den Vertrag nicht ändern darf ohne die davon abhängigen Anwendungen davon in Kenntnis zu setzen. Eine Änderung des Vertrages würde automatisch das erneute Erstellen aller auf dem Vertrag aufbauenden Programme und/oder Komponenten erforderlich machen. Andernfalls führen derartige Änderungen im vTable zu Anwendungsfehlern.

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

15.5 Microsoft Interface Definition Language (MIDL)

Wie schon erwähnt, kommunizieren COM/DCOM-Applikationen miteinander und mit dem System durch eine als Interface spezifizierte Menge von Methoden. Ein Interface ist bei COM/DCOM ein streng definierter Vertrag zwischen Software und Client, der eine Menge an Operationen zur Verfügung stellt. Dies ist absolut vergleichbar mit den Schnittstellenvereinbarungen bei CORBA-IDL oder den Home- und Remote-Schnittstellen bei Enterprise JavaBeans.

Es wurde eingangs erwähnt, dass COM und CORBA konzeptionell gewisse Ähnlichkeiten haben. Das Einzige, was COM/DCOM- und CORBA-IDL jedoch gemeinsam haben, sind ihr Name und ihre Aufgabe. In der Syntax sind sie jedoch vollkommen unterschiedlich!

15.6 Typen

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

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

Sprache	IDL	Microsoft C++	Visual Basic	Java
Basistypen	boolean	unsigned char	n/a	char
	byte	unsigned char	n/a	char
	small	char	n/a	char
	short	short	Integer	short
	long	long	Long	int
	hyper	_int64	n/a	long
	float	float	Single	float
	double	double	Double	double
	char	unsigned char	n/a	char
	wchar_t	wchar_t	Integer	short
	enum	enum	Enum	int
	Interface Pointer	Interface Pointer	Interface Reference	Interface Reference
Erweiterte Typen	VARIANT	VARIANT	Variant	ms.com.Variant
	BSTR	BSTR	String	java.lang.String
	VARIANT_BOOL	short	Boolean ∈{True, False}	boolean ∈ {true, false}

Tabelle 8: MIDL Datentypen

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

IDL zeichnet sich durch eine bestimmte Strukturierung aus. Eine COM/DCOM-Komponente ist ein Paket, das über das `library` gekennzeichnet wird. Dieses Paket kann wiederum mehrere COM-Klassen beinhalten. Sowohl Pakete als auch Co-Klassen werden durch einen GUID identifiziert; bei Co-Klassen spricht man hier von einem Class ID (CLSID). Das geschieht durch das `uuid` (**u**niversally **u**nique **id**entifier)-Attribut. Ein Library-Block kann im Gegensatz zu dem Beispiel auch andere Attribute enthalten. Der Vollständigkeit halber sollen hiermit alle Attribute aufgezählt werden:

1. uuid: ID zur Identifikation der Bibliothek. Dieser Parameter ist zwingend erforderlich.
2. version: Gibt die Versionsnummer der Bibliothek an.
3. helpstring: Gibt in Textform zusätzliche Information zur Bibliothek und kann auch in Objektbrowsern abgelesen werden.
4. lcid: Gibt die Sprache der Bibliothek an.
5. hidden: Versteckt das Objekt vor einem COM-Objektbrowser.
6. coclass: `coclass <Klassenname>` definiert die Implementierung einer COM/DCOM-Komponente. Die Definition einer Co-Klasse hat folgende Syntax:

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

Beispiel 15: Interfaceaufbau bei DCOM

Im Zusammenhang mit einer Koklassendefinition können noch weitere Attribute gesetzt werden:

1. source: Gibt an, dass das Interface Events (Ereignisse) unterstützt.
2. default : Legt fest (bei VBScripten), welches Interface standardmäßig aufgerufen wird, wenn keines angegeben ist.
3. restricted: Verhindert die Verwendung eines Interfaces durch Makroprogrammierer. Interessant erscheint hier noch, dass in jeder IDL das Statement `importlib("STDOLE2.TLB")` vorkommt. STDOLE2.TLB muss immer importiert werden, da dieses die Schnittstellen von IUnknown und IDispatch definiert.

Methoden in COM-Komponenten haben grundsätzlich immer den Rückgabewert `HRESULT`. `HRESULT` gibt an, ob ein Methodenaufruf unter COM erfolgreich aufgerufen wurde oder nicht. Wie lässt sich aber `HRESULT` mit einem beliebigen return type einer Funktion vereinbaren? Die Methode

```
long getTimeofDay();
```

wird in IDL folgendermaßen umschrieben:

```
HRESULT getTimeOfDay([out, retval] long);
```

Die Richtungsattribute `[in]`, `[out]`, `[in, out]` geben analog zu CORBA die Richtung der Parameter an. Standardwert ist `[in]`, d. h. dass Daten vom Konsumenten nur zur Implementierung geschickt werden. `[out]` steht dementsprechend für einen Datenfluss in die entgegengesetzte Richtung.

Die Kombination `[in, out]` gibt an, dass Daten von einem Konsumenten zur Komponente fließen, in der Implementierung gelöscht und neu geschrieben werden, um letztendlich wieder an den Konsumenten zurückgeschickt zu werden.

15.7 COM/DCOM Server

Allgemein ausgedrückt ist ein Server ein Teil eines Programmierkodes, der einige COM-Komponenten enthält mit der Folge, dass die COM Bibliothek und ihre Dienste diesen Code verarbeiten können. Jeder spezifische Server kann in einer von mehreren Formen implementiert sein, abhängig von der Struktur des Moduls und seines Verhältnisses zum Klientenprozess, der ihn benutzen wird. Der Server ist entweder im Status *in-process*, was bedeutet, dass sein Code in dem gleichen Prozessraum ausgeführt wird wie der Klient (als DLL), oder im Status *out-of-process* (als EXE) und somit in unterschiedlichen Prozessen auf gleichen oder unterschiedlichen Systemen.

Bei der Entwicklung eines Servers müssen die nötigen Objekte und vor allem deren Schnittstellen bekannt sein.

Eine weitere, allgemeine Methode zur Lokalisation eines Objektes wird durch sogenannte *monikers* realisiert, die wiederum Objekte darstellen, die einen Namen einkapseln. Diese dienen als Aliase für Objekte und einem Objekt einen persistenten Status verleihen. Eine QueryInterface-Operation liefert einen Zeiger zu jedem entsprechendem Objekt und ermöglicht die Prüfung auf Gleichheit aufgrund der vom Zeiger zurückgelieferten Werte.

Im Gegensatz zum objektorientierten Ansatz hat das Interface bei COM keinen Zustand und kann nicht realisiert werden, um die Erzeugung eines eindeutigen Objektes durchzuführen. Ein COM Interface ist nur eine Gruppe von zusammengehörigen Funktionen bzw. Methoden. COM Clients erhalten einen Zeiger, mit dem sie Zugriff auf eine Funktion dieses Interfaces bekommen. Dieser Zeiger enthält keine Informationen über den Zustand. Der Client kann demnach zu einer späteren Zeit nicht zur selben Objektinstanz (im gleichen Zustand) wieder verbunden werden. Es kann nur mit einem Interface-Zeiger der gleichen Klasse wieder verbunden werden. COM Objekte behalten also ihren Zustand nicht.

Wenn ein Client nach einem Objekt einer bestimmten Klasse fragt, lädt COM den Servercode und führt ihn aus, zwecks Herstellung eines Objektes dieser Klasse. Wenn dies geschehen ist, bekommt der Client einen Zeiger auf das erste Interface zurück geliefert. Der Server ist nicht selber das Objekt. Das Wort Server wird bei COM in dem Sinne genutzt, dass damit die Serviceleistung bezeichnet wird. Somit muss ein Server die notwendigen Strukturen um das Objekt herum zur Verfügung stellen, um es dem Client zugänglich zu machen.

Die folgende Abbildung soll den Vorgang der Objekterzeugung noch mal verdeutlichen. Es wird angenommen, dass die Klasse mit dem Namen MyClass in Windows registriert ist. Diese Klasse unterstützt ein Interface, genannt *IMyInterface*, und wird von einem Server - *MyServer.exe* - zur Verfügung gestellt.

Die Registrierung enthält einen Eintrag unter CLSID für MyClass, der so ähnlich aussieht wie
LocalServer32=c:\MyDir\MyServer.exe .

Unser Beispiel beginnt, nachdem COM *MyServer.exe* in dem Verzeichnis gefunden und ihn in den Speicher geladen hat. Nun erzeugt ein Client eine Instanz von MyClass und erhält einen Zeiger auf IMyInterface (vgl. Abbildung 44 oder [Abernethy et al., 1999]):

1. Der Server bewirkt ein *class factory* und lässt ihn registrieren. Sobald MyServer.exe geladen ist, muss es alle *class factories* erzeugen und diese bei COM registrieren lassen. In diesem Beispiel gibt es nur eine Klasse, MyClass. Diese Klasse stellt eine factory mit dem IClassFactory2 interface zur Verfügung. Der Server aktiviert das factory interface und übergibt COM einen Zeiger (zu sich) über das API CoRegisterClassObject. Nun weiß COM, wie es die factory für diese CLSID finden kann.
2. Der Client fordert einen Zeiger zur class factory an. Der Client ruft den COM API CoGetClassObject, um einen IClassFactory2 -Zeiger für MyClass CLSID zu bekommen.

3. Der Client fordert eine Instanz dieser Klasse an. Der Client ruft die Funktion IClassFactory2::CreateInstanceLic, um eine Instanz von MyClass zu erhalten.
4. Die factory erzeugt die angeforderte Instanz von MyClass und übergibt dem Clienteinen Zeiger auf das gewünschte Interface - IMyInterface.
5. Der Client kommuniziert mit dem Interface mit Hilfe des Zeigers und benutzt die Methoden, die das Interface zur Verfügung stellt.
6. Der Client gibt den Zeiger mit Hilfe der Release-Funktion frei und dekrementiert den Referenzzähler. Dieser stimmt mit dem AddRef überein, den man erhalten hat, als der Client den Zeiger auf IMyInterface zugewiesen bekam.
7. Der Client gibt mit Hilfe der Release-Funktion die IClassFactory2 frei, wiederum mit entsprechender Anpassung des Referenzzählerstandes.
8. Der Server wird geschlossen. Währenddessen muss der Server den APS CoRevokeClassObject aufrufen, um COM mitzuteilen, dass die zuvor registrierte class factory nicht mehr zur Verfügung steht.

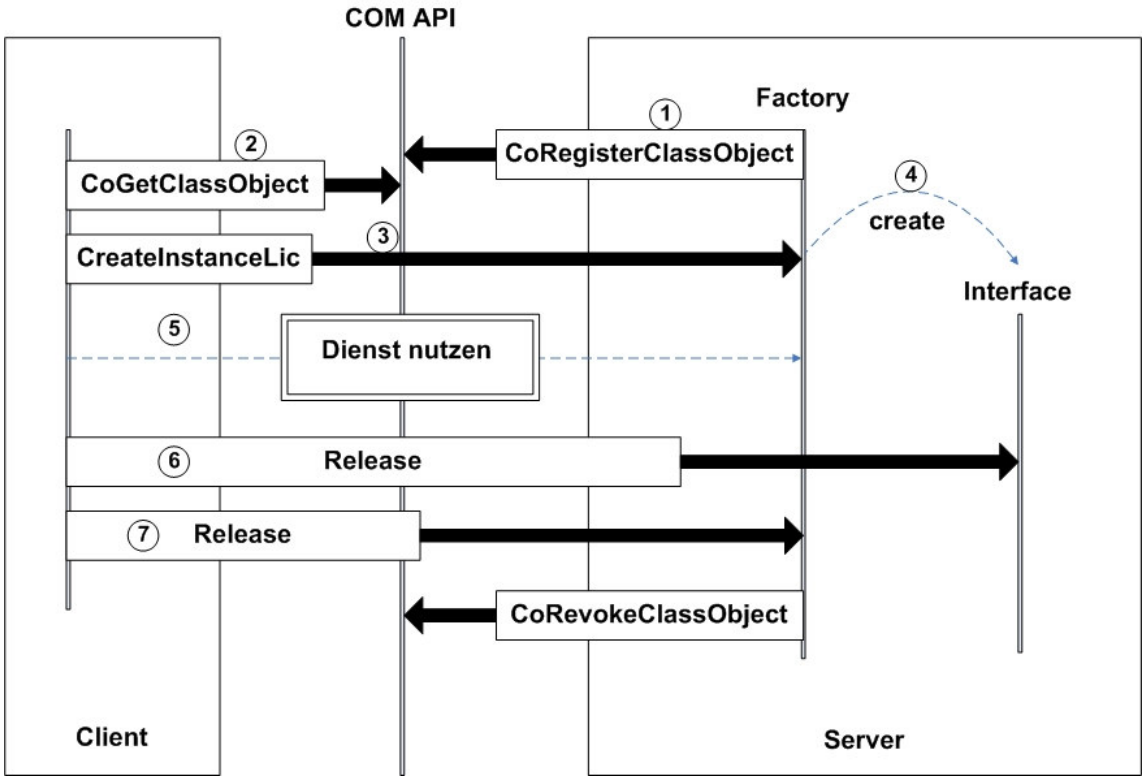


Abbildung 44: Objekterzeugung mit DCOM

15.8 Versionsverwaltung

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

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

von einer Komponente älteren Datums überschrieben wird? Das Problem dabei ist meist in Komponenten zu suchen, die von mehreren unterschiedlichen Programmen verwendet werden. Die MFC-Runtime-DLL (**M**icrosoft-**F**oundation-**C**lasses) `mfc42.dll` ist ein Klassiker für weit verbreitete DLLs. Diese DLL wird nämlich von jedem Programm verwendet, das mit Visual C++ entwickelt wurde.

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

Wenn alte Schnittstellendefinitionen in neuen Versionen nicht verändert werden, so erlaubt es alten wie neuen Klienten, die neue Komponente zu verwenden. Desweiteren können neuere Klienten alte Komponenten fragen, ob diese einen bestimmten **I**nterface **Id**entifier (IID) und mit ihm bestimmte Operationen unterstützen. Für diese Zwecke kann die Methode `QueryInterface` in `IUnknown` verwendet werden. Mittels `QueryInterface` kann ein neuer Klient sein Verhalten an das Vorhandensein einer älteren Komponente anpassen, um gegebenenfalls nur einen eingeschränkten Funktionsumfang zu unterstützen. Letztendlich sollen dadurch ungültige Funktionsaufrufe als Auslöser für Laufzeitfehler rechtzeitig abgefangen werden können.

Es wird an dieser Stelle unterstellt, dass es von einer Komponente zwei Versionen, *v1* und *v2* gibt, wobei *v2* mehr Methoden als *v1* unterstützt. Wird eine Anwendung mit der Komponente *v2* erstellt, ist dieser nicht bekannt, dass es auch eine *v1* mit einer eingeschränkten Schnittstelle gibt. Laufzeitfehler sind vorprogrammiert, sobald die Anwendung *v1* anstatt der aktuellen *v2* verwendet und eine Methode der neueren Komponente *v2* aufgerufen wird. Es gibt zwei Möglichkeiten dieses Problem zu vermeiden:

1. Entweder werden stets die neuesten Versionen von Klienten und COM/DCOM-Komponenten verwendet, oder
2. man verwendet benutzerdefinierte Schnittstellen. Hierbei werden abstrakte Klassen definiert und separat von der Klassendefinition implementiert.

15.9 Microsoft Transaction Server (MTS)

Der MTS (**M**icrosoft **T**ransaction **S**erver) ist ein vollwertiger Transaktionsmonitor mit der zusätzlichen Eigenschaft, komponentenbasiert zu sein. Damit ist analog zu den Enterprise JavaBeans die Möglichkeit verbunden, Transaktionsverarbeitung unabhängig von dem Aufrufer zu automatisieren (vgl. [Microsoft, 2002]).

Wichtige Eigenschaften des MTS sind:

1. Automatische Transaktionsverarbeitung: MTS erlaubt es, Komponenten derart zu konfigurieren, dass ggf. eine Transaktion automatisch instanziiert und bei erfolgreicher Verarbeitung automatisch mit einem commit beendet wird.
2. Interoperabilität mit anderen Transaktionsmonitoren. Durch den COM Transaction Integrator können CICS Transaktionen initiiert und kontrolliert werden.
3. Sicherheit: Ein Administrator verfügt über die Möglichkeit, Rollen zu definieren und die Schnittstellen und/oder Komponenten durch Zugriffsrechte zu schützen.
4. Datenbank-Verbindungspools: MTS erlaubt die administrative Einrichtung von mehreren, permanenten Verbindungen zu dem Ressource-Manager einer Datenbank. Damit entfällt für den Klienten der zeitaufwendige Aufbau und Abbau von Verbindungen, stattdessen verwendet man eine bestehende Verbindung aus dem Pool. Unterstützt werden eine Vielzahl von Datenbanken, wie z. B. Microsoft SQL Server, DB2, Oracle etc. Generell sollte jede Datenbank, die über einen ODBC-Treiber verfügt, einzubinden sein.

5. Automatische Threadunterstützung:
6. Management bei zustandsbehafteten Komponenten: Zum einen wird sichergestellt, dass der Zustand exklusiv einem Klienten gehört und zum anderen, das der Zustand bei Bedarf ausgelagert und wieder eingeladen werden kann.
7. Prozess-Isolation: Prozesse können einzeln oder gruppiert in eigenen Adressräumen ablaufen. Damit verbunden ist eine höhere Fehlertoleranz, da durch eine Terminierung nicht alle Komponenten gefährdet sind.
8. Sprachunterstützung: Unterstützt werden zumindest die Sprachen Microsoft Visual Basic, C++, C#, Cobol und Java.

Damit erfüllt der MTS alle Forderungen, die auch an die Enterprise JavaBeans Komponenten gerichtet werden.

15.10 Microsoft Message Queue (MSMQ)

Die Notwendigkeit, ggf. auch asynchron kommunizieren zu können, wird durch die Microsoft Message Queue erfüllt (vgl. (Microsoft, 2002]).

Dieser Dienst verfügt über entsprechende Schnittstellen für die Primitive send/receive, aber auch für das Setzen von bestimmten Charakteristiken wie z. B. die Priorität einer Nachricht. Ein wichtiger Aspekt ist die Möglichkeit, MTS vollständig zu integrieren. Damit ist einem Klienten die Möglichkeit gegeben, im Rahmen einer einzelnen Transaktion mehrere Nachrichten zu dem Dienst zu schicken, statt einzelne Pakete mit ständig neuem Verbindungsaufbau. Diese Optimierung ist gängige Praxis und wird u. a. auch für Tuxedo-Queues oder dem Java Message Service verwendet.

Wichtige Eigenschaften von MSMQ sind:
1. Automatisches Aufzeichnen von Nachrichten (automatic message journaling): Falls diese Option gesetzt ist, kann die MSMQ Kopien der gesendeten oder empfangenen Nachrichten anfertigen und somit die Ausfallsicherheit erhöhen oder zumindest das Auditing ermöglichen, also das mitprotokollieren dessen, was in der Vergangenheit geschehen ist.
2. Automatische Benachrichtigung der Klienten: Aufgrund der Asynchronität wissen Sender und Empfänger von Nachrichten nichts voneinander und i. d. R. weiß der Sender einer Nachricht nicht, ob die Daten zugestellt wurden oder nicht. MSMQ verfügt ebenso wie der Java Message Service über die Möglichkeit, dem Klienten die Entgegennahme und Weiterleitung der Nachricht zu quittieren. Im Fehlerfall hat der Klient die Möglichkeit zu reagieren, im Erfolgsfall ist damit die Sicherheit verbunden, dass die Daten korrekt weitergeleitet wurden.
3. Datenintegrität, Privatheit der Daten und digitale Signaturen: Für den Datentransfer über das Netzwerk kann MSMQ eine digitale Signierung oder eine Verschlüsselung der Daten selbständig vornehmen.
4. MSMQ erlaubt die Zuordnung von Prioritäten bzgl. der Nachrichten. Analog zu JMS ist diese jedoch mit einem erhöhten Aufwand auf der Serverseite verbunden.
5. Die Verfügbarkeit von MSMQ ist neben den Windows-Betriebssystemen auch bei einigen UNIX-Derivaten und auf IBM-Großrechnern gegeben.
6. Im Umfeld der Komponenten-Dienste von Microsoft ist MSMQ für die sogenannten "mission critical applications" von großer Bedeutung.

15.11 Kurzvorstellung von .NET

Bereits 1998 begann Microsoft an der neuen Entwicklungsumgebung .NET zu arbeiten (vgl. [Westphal, 2002]). Eine breite Palette von Microsoftprodukten, speziell für die Entwicklung verteilter Anwendungen war bereits veraltet und musste mit jeder neuen Version die bekann-

te Problematik der Abwärtskompatibilität erfüllen. Die alten Technologien wie das Win32 API oder COM waren nur noch schwer zu warten, unflexibel bzgl. ihrer Weiterentwicklung und zu eng verzahnt mit der proprietären Windows-Welt.

Microsoft, ebenso wie andere Softwarehersteller, musste sich die Frage stellen, welche möglichen Veränderungen in der Hard- und Software in den nächsten Jahren zu erwarten sind.

PDAs, Handhelds oder sogenannte Smart-Handy's werden an Bedeutung gewinnen. Ein Datenaustausch mit Desktop-Systemen, insbesondere jedoch mit Servern, ist somit eine absolute Notwendigkeit. Es kann nicht davon ausgegangen werden, dass diese neuen Geräte auf Intel-Plattformen mit dem proprietären Windows CE aufsetzen.

Angesichts dessen muss, von allen Softwareanbietern, die folgende Frage aufgeworfen und beantwortet werden (vgl. [Westphal, 2002]):

> Wie müssen Technologien, Produkte und Strategie aussehen, um in einer vernetzten, heterogenen Hard- und Softwarewelt langfristig erfolgreich zu sein?

Für Microsoft und auch andere Anbieter folgt unmittelbar eine zweite, ebenfalls existentiell wichtige Frage:

> Sind vorhandene (Microsoft) Technologien und Produkte fähig (hinsichtlich Funktionalität, Form, Flexibilität usw.), die Herausforderung anzunehmen?

Für .NET ergibt sich somit eine Reihe von zu erfüllenden Anforderungen:

1. .NET muss (auch) eine Plattform für die Nutzung und das Angebot von Internet-Komponenten bieten.
2. .NET muss als Strategie einen Weg aufzeigen, wie Microsoft auch in einer Welt von weniger Software-Paketen und mehr Software-Dienstleistungen bestehen kann.
3. .NET muss als Plattform leichtgewichtig sein, um so als attraktive Programmierschnittstelle schnell auf eine Vielzahl von Plattformen portiert zu werden.
4. .NET muss interoperabel zu anderen Standards wie z. B. CORBA oder J2EE sein.
5. .NET muss sowohl lose als auch enge Koppelung von (verteilten) Komponenten unterstützen.

Mit .NET verfolgt Microsoft das Ziel, ein leichtgewichtiges und in einem gewissen Sinn sprachunabhängiges Framework zur Verfügung zu stellen. Die unterstützten Sprachen sind C#, J#, Visual Basic, ASP (**A**ctive **S**erver **P**ages, der Microsoft Pendant zu JavaServer Pages von Sun Microsystems) und bis zu einem gewissen Grad C++. Die unterstützten Sprachen werden wie in der Java-Welt üblich in einen Zwischenkode übersetzt, der für alle gleich ist und in einer CLR (**C**ommon **L**anguage **R**untime) abgearbeitet wird.

Ebenfalls analog zu Java werden von .NET eine Reihe von Bibliotheken z. B. für Threading oder bestimmte Datentypen wie die Klasse string zur Verfügung gestellt (vgl. Abbildung 45 oder [Liberty, 2001]).

Berücksichtigt man, dass Microsoft über serverseitige Komponenten wie z. B. MTS, MSQS und den BizTalk-Server verfügt, so kann in der Kombination der einzelnen Bausteine ein Applikationsserver im Sinne von J2EE-Applikationsservern leicht aufgebaut werden und somit können verteilte, hochverfügbare Anwendungen erstellt und zentral zur Verfügung gestellt werden. Mit dem Internetserver IIS und den ASPs ist auch der Aufbau von Internetdiensten leicht möglich.

Von besonderem Interesse ist Microsoft's Unterstützung von Web Services, die in Kapitel 17 im Detail behandelt werden. In diesem Sinne wird auch kein direkter Vergleich von .NET mit CORBA angestrebt, sondern vielmehr der Vergleich CORBA mit Web Services.

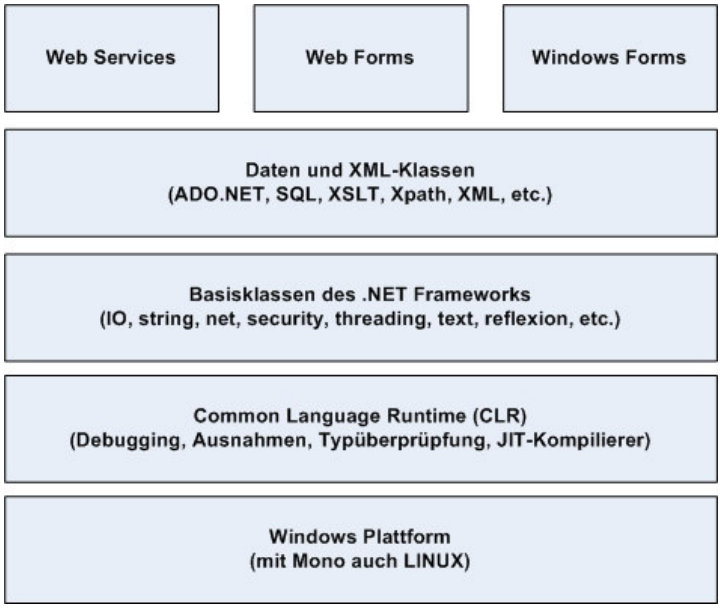


Abbildung 45: Schematischer Aufbau von .NET

16 CORBA vs. DCOM

Im Folgenden werden die beiden Technologien CORBA und DCOM miteinander verglichen. Ein komprimierter Vergleich befindet sich in Form einer Matrix in Kapitel 19. Zusätzlich ist eine detaillierte Version von [Barke, 1996] zur Verfügung gestellt worden, die hier primär verwendet wird.

Die beiden Technologien scheinen aufgrund des RMI-Kommunikationsmechanismus eng miteinander verwandt zu sein; es ergeben sich doch beträchtliche Unterschiede, die im Folgenden untersucht werden.

Der wohl signifikanteste Unterschied zwischen COM/DCOM und CORBA liegt direkt im Objektmodell. CORBA folgt dem klassischen Objektmodell und erweitert diesen Ansatz (ab CORBA 3.x) auf Komponenten. Ein CORBA-Objekt ist somit auch ein Objekt erster Klasse, d. h., es unterstützt mehrfache Vererbung, Kapselung und Polymorphie. Wie in der Objektorientierung üblich, wird ein CORBA-Objekt durch eine Klasse beschrieben, die zur Laufzeit aktiviert, also konkret werden kann. Eine Objektinstanz besitzt immer, wie von einem klassischen Objekt zu erwarten ist, eine eindeutige Referenz-ID.

Analog zu CORBA stellt auch COM/DCOM stellt universelle eindeutige IDs für Klassen bereit, die dem DCE entliehen ist. Jedoch gilt dies nicht für Objekte zur Laufzeit, d. h. konkrete Instanzen dieser Klassen. Wird eine Verbindung zu einem OLE-Objekt aufgebaut, so erhält man einen Zeiger auf eine Schnittstelle, mit dem man anschließend arbeiten kann. Möglicherweise beinhaltet das konkrete Objekt mehr Methoden als in der Schnittstelle ausgewiesen sind; nutzbar für den Anwender sind ausschließlich die Methoden der Schnittstelle. Dies ist eine gängige Praxis, die auch im J2EE-Umfeld häufig anzutreffen ist. Für einen eventuellen späteren Prozess, der das gleiche Objekt instanziiert, gibt es keine Möglichkeit der Wiederaufnahme dieser Verbindung. Dies kann unter Umständen eine große Auswirkung auf die Performanz der Server bzw. Netzwerke bedeuten.

COM/DCOM-Komponenten sind keine Klassen im eigentlichen objektorientiertem Sinne. Dies ist darauf zurückzuführen, dass eine COM/DCOM-Klasse nicht über Vererbung erweitert werden kann. Damit reduziert sich der objektorientierte Ansatz auf einen einfachen Verkapselungsmechanismus führt.

OLE unterstützt eine Funktion *aggregation*, die es einem Objekt ermöglicht, mehrere Schnittstellen zur Verfügung zu stellen; dies beinhaltet auch Schnittstellen, die von anderen Objekten unterstützt werden. Dadurch kann erreicht werden, dass eine äußere Komponente die Schnittstelle einer inneren Komponente repräsentiert. Das Merkmal der „Vererbung" wird also künstlich erzeugt.

16.1 Einbindung höherer Sprachen

Der Nachrichtenaustausch zwischen COM/DCOM-Komponenten erfolgt über ein proprietäres, binäres Protokoll, das die Formate für die zu übertragenden Daten festlegt. Analog zu CORBA liefert Microsoft mit MIDL einen Precompiler aus, der die Generierung der Stubs für die Aufrufe zwischen Prozessen erleichtert. Leider ist im Gegensatz zu CORBA die Sprachenunabhängigkeit nicht gegeben. CORBA dagegen unterstützt zumindest sechs Programmiersprachen in seinem Standard und das Hinzufügen weiterer Sprachen ist primär die Definition einer Sprachabbildung von IDL auf die Zielsprache.

Dies ist schon wegen des hohen Anspruchs der Portierung, aber auch aufgrund der in der Praxis vorhandenen Heterogenität bzgl. Betriebssystemen und Programmiersprachen not-

wendig. Durch den Einsatz einer höheren Beschreibungssprache ist die dem Objekt zugrunde liegende Middleware abgeschirmt. Das bedeutet, dass durch CORBA der Grad der Abstraktion von einem binären Standard (mit Zeigern auf Tabellen im Speicher) hin zu einem Konstrukt einer höheren Sprache verschoben wird. Dadurch kann der Entwickler in seiner Sprache mit entfernten Objekten transparent umgehen. Die Interoperabilität geht so weit, das eine CORBA Komponente, realisiert in C++, eine andere Komponente aufrufen kann, die in Java oder COBOL geschrieben wurde und von einem herstellerfremden ORB stammt. Keine andere Client/Server-Middleware hat bisher einen vergleichbaren Grad an Abstraktion und den Möglichkeiten der Portierung und Interoperabilität erreicht.

16.2 Interoperabilität CORBA, DCOM und EJBs

Eine direkte Interoperabilität zwischen CORBA und DCOM ist nicht gegeben. Von Seiten der OMG wird eine CORBA-COM-Bridge angeboten, die jedoch nicht von allen Herstellern implementiert wurde.

Die Interoperabilität zwischen CORBA und EJBs ist aufgrund des gemeinsamen Protokolls IIOP (bzw. RMI over IIOP) gewährleistet. Für die Interoperabilität zwischen J2EE-Komponenten und DCOM stellen einige Hersteller von Applikationsservern Adaptoren zur Verfügung, wie z. B. jCom von BEA Systems. Die Qualität dieser Softwarebausteine hängt in einem hohen Masse von dem Hersteller ab.

Erst mit der Einführung von .NET hat Microsoft sich einem Standard angepasst, der Interoperabilität wirklich zuläßt und von einer Vielzahl von anderen Herstellern mitgetragen wird.

16.3 Unterstützung und Fähigkeit der Portierung

Auch die Portabilität spielt bei der Kostenbetrachtung eine große Rolle, da viele Anwendungen auf den unterschiedlichsten Plattformen laufen müssen und Kunden von den Herstellern möglichst unabhängig sein wollen.

Ein damit verbundener Aspekt ist die Notwendigkeit von mehreren Versionen oder Ständen einer Software, um die Interoperabilität mit anderen Sprachen, Speichermodulen oder ähnlichen Faktoren zu unterstützen. Genau an dieser Stelle ist einer der größten Vorteile von CORBA zu sehen: Die Konsistenz der API's zwischen allen Plattformen. Die OMG IDL Sprachenabbildungen führen, nachdem sie standardisiert sind, zu einer Konsistenz für alle Plattformen untereinander. Allgemein ausgedrückt kann gesagt werden, dass CORBA keinen Grund für eine Plattformabhängigkeit liefern wird, denn kritische Punkte, wie Betriebssystem-API's und Fenster-API's, werden von anderen Standards (POSIX und OSF/MOTIF etc.) realisiert. Selbst bei der Entwicklung von Clients und Servern in unterschiedlichen Sprachen, entfällt die Notwendigkeit von mehreren Softwareständen.

COM/DCOM wird hervorragend von den Windows Systemen unterstützt. Es ist außerdem auf McIntosh verfügbar, dort jedoch schlecht dokumentiert. Hersteller wie die ehemalige Digital Equipment Corporation (DEC), heute HP, haben Teile des COM/DCOM-Standards bzw. seiner Funktionalität auf andere Systeme übertragen. Die Software AG war ebenfalls aktiv, um diesen Standard Richtung UNIX zu portieren. Allerdings beschränkten sich diese Portierungen auf ein bloßes Kernsystem; die typischen Wizards und andere Komponenten wurden nicht berücksichtigt.

Allein das von Microsoft gelieferte OLE2 API, als Teiltechnologie von COM/DCOM, verursacht eine starke Abhängigkeit von der Plattform und dies wird sich auch in Zukunft nicht ändern. Erschwerend kommt hinzu, dass OLE2 explizit Microsofts C++ als Spracheinbindung

definiert hat und dadurch ist auch eine entsprechende Sprachabhängigkeit festgeschrieben worden, was zu einem großen Aufwand führt, wenn eine Wiederverwendbarkeit des Programmkodes in anderen Spracheinbindungen erforderlich sein sollte.

16.4 Systemdienste

In der folgenden Tabelle werden die wichtigsten Systemdienste von COM/DCOM und CORBA aufgelistet. Es muss an dieser Stelle nochmals erwähnt werden, dass COM keine verteilten Dienste unterstützt. CORBA ist im Vergleich mit dem Produkt von Microsoft um einige Jahre voraus und hat z. B. Schnittstellen zu **O**bject **D**atabank **M**anagement **S**ystems (ODBMS) spezifiziert oder sämtliche ad-hoc Relationen zwischen den Komponenten entwickelt.

Zusätzlich muss bei COM/DCOM immer der lokale und der entfernte Fall unterschieden werden. Bei CORBA wäre dies obsolet; die beiden Spalten für lokal und entfernt sind lediglich aus Gründen der Konsistenz angelegt worden.

Dienste	**COM/DCOM**		**CORBA**	
	Lokal	Entfernt	Lokal	Entfernt
Event	ja	nein	ja	ja
Life Cycle	ja	nein	ja	ja
Namensdienst	ja	nein	ja	ja
Persistenz	ja	nein	ja	ja
ODBMS Integration	nein	nein	ja	ja
RDBMS Integration	nein	nein	ja	ja
Externalization	ja	nein	ja	ja
Transaktionsdienst	nein	nein	ja	ja
Concurrency	nein	nein	ja	ja
Relationship	nein	nein	ja	ja
Query	nein	nein	ja	ja
Licensing	ja	nein	ja	ja
Properties	ja	nein	ja	ja
Versioning	nein	nein	ja	ja
Security	nein	nein	ja	ja
Garbage Collection	ja	nein	ja	ja

Tabelle 9: CORBA vs. DCOM

16.5 Unterschiedliche Kommunikationswege

Die Integration der unterschiedlichen Kommunikationsmechanismen ist eine wichtige Fähigkeit für die Integration unterschiedlicher Software und ggf. die Möglichkeit, mehrere Systeme aus Sicht des Endanwenders wie ein System erscheinen zu lassen.

Die Mehrzahl der eigenentwickelten oder kommerziellen Softwareprodukte nutzen die unterschiedlichen Kommunikationsmethoden, wie Berkeley Sockets, **O**pen **N**etwork **C**omputing (ONC, ein RPC-Mechanismus von Sun Microsystems), DCE und diverse andere. Ein Merkmal der objektorientierten Softwarearchitektur besteht aus der Kapselung der Unterschiede zwischen diesen Kommunikationsmethoden. Die Spezifikation von CORBA führt explizit zu einer Unabhängigkeit von dem zugrunde liegenden Kommunikationsmechanismus, denn

dieser wird durch das OMG IDL API gekapselt. In dem Einführungskapitel zu CORBA wurden bereits die einzelnen, unterstützten Protokolle und die Transparenz für den Entwickler besprochen. Bei COM/DCOM dagegen wird der zugrunde liegende Kommunikationsmechanismus von Microsoft geliefert. Es keine Möglichkeit der Anpassung oder des Austausches.

16.6 Stabilität

Die Lebensdauer eines IT-Systems von Endverbrauchern mit z. T. selbst entwickelten, proprietären Komponenten kann durchaus 10 bis 15 Jahre betragen. In der Praxis findet man z. B. im Bankenumfeld COBOL- oder PL/1-basierte Anwendungen, die in ihrer Grundversion bereits vor 30 Jahren entwickelt wurden. Diese Zeit beschreibt ungefähr die praktische Lebensspanne einer Technologie. Dagegen ist die Entwicklung der kommerziellen Softwaretechnologie auf deutlich kürzere Zyklen komprimiert. Neue Produkte und Versionen werden alle 6 bis 12 Monate auf den Markt gebracht. Die kommerzielle Lebensdauer einer wichtigen Technologie, wie z. B. CORBA, kann sogar kürzer als 3 Jahre sein, bevor eine neue Innovationswelle, wie z. B. J2EE, den Markt verändern kann. Daher ist es im Sinne eines Investitionsschutzes von größter Bedeutung, bei der Einführung bzw. Wahl einer Technologie auf solche Faktoren wie Stabilität, Kompatibilität bzw. Portabilität und Herstellerunterstützung in einem längeren Zeitrahmen zu achten.

Bei Microsoft ist zu bereits zu beobachten, dass aufgrund der großen Konkurrenz durch Produkte mit technologischem Vorsprung gegenüber OLE2.0 und COM/DCOM der Zwang zu einer Entwicklung einer neuen (standardisierten) Technologie besteht (s. .NET).

Ein Kernproblem für viele Entwickler ist die Veralterung der APIs, die durch die hohe Innovationsrate bedingt ist. Ein Endverbrauchersystem mit einer angestrebten Lebensdauer von 10 bis 15 Jahren, das z. B. auf OLE 2.0 aufbaut, geht ein erhebliches Risiko ein. Wenn Microsoft, wie in der Vergangenheit mehrfach geschehen, sein API erneuert oder sogar ersetzt, werden signifikante Kosten für die Anpassung der Anwendungen entstehen, um mit dem technologischen Fortschritt von Microsoft Schritt zu halten.

Ein wichtiger Faktor, um die Geschwindigkeit des Veraltens von Technologien zu bremsen, ist die Standardisierung. Eine Standardisierung ist eine weit verbreiterte Marktstrategie, deren Kosten von den Soft- und Hardwareherstellern getragen werden. Standards reduzieren die Risiken sowohl für den Hersteller als auch für den Kunden, die Akzeptanz für eine Technologie wird gesteigert und das Vertrauen gefestigt. Die Standardisierung kann von einer internationalen Gruppe wie z. B. der OMG oder der X/Open-Group geleistet werden, oder auch von einzelnen Unternehmungen wie z. B. die Standardisierung von J2EE durch Sun Microsystems in Zusammenarbeit mit anderen Firmen wie IBM, BEA, etc. Wichtig in diesem Zusammenhang ist die Bildung einer gemeinsamen Infrastruktur, die zur Unterstützung von mehreren Plattformen und Funktionalitäten führt.

CORBA ist das Resultat einer solchen Vereinigung von mehreren hundert Herstellern zur OMG. Ist der Standard etabliert, kann er die Lebensdauer einer Technologie beträchtlich steigern. Wegen der

1. Unterstützung und Beteiligung von sehr vielen und vor allem den bedeutenden Herstellern,
2. den gewählten Standards und
3. der Realisierung als geschichtete Architektur,

hat CORBA alle Indikatoren, sich zu einer Technologie mit langer Lebensdauer zu entwickeln. Diese Lebensdauer wird zunehmend gefestigt und erweitert durch die Unterstützung anderer Standardsysteme, die CORBA als Basisstandard nutzen wie die International Standards Organization's Open Distributed Processing, X/Consortium, X/Open, die Standardisierungsgremien der Web Services etc.

16.7 Unterstützung von Threads

Multithreading ist die Fähigkeit, mehrere konkurrierende Prozesse innerhalb eines Programmes vom Betriebssystem ausführen zu lassen. Diese Funktion ist äußerst wichtig um zu verhindern, dass ein Client mit einer Anfrage das System blockiert, weil z. B. der Server die Antwort verzögert. Bei großen Anwendungen mit sehr vielen Clients (z. B. Abfragen oder Datenbanktransaktionen) ist die Option zum Multithreading sehr wichtig, aber nicht unverzichtbar. Bspw. verwendet BEA Tuxedo in den Versionen bis 6.5 anstelle von Threads die Erzeugung und Überlagerung von Kindprozessen.

16.8 Komplexität einer Technologie

Ein bedeutender Kostenfaktor liegt in der Integration von Kundenanwendungen in der COM/DCOM bzw. CORBA Umgebung. Die Komplexität der API's, die Qualität der Dokumentation und die Verfügbarkeit von Schulungen etc. beeinflussen direkt die Kosten beim Erlernen und Anwenden dieser Technologien. Allein OLE-2, als eine Subtechnologie von COM/DCOM, ist eine der komplexesten Technologien, die Microsoft je herausgegeben hat. Der Autor der Dokumentation *Inside OLE2* beschreibt OLE-2 als deutlich komplexer als das Windows Betriebssystem. Als Beispiel sei hier nur der Umfang der Dokumentationen genannt: *Inside OLE2 developer's guide* (925 Seiten), *The API-Reference* (650 Seiten) und *The OLE Automation Reference* (400 Seiten). Die Erfahrung von OLE-Entwicklern zeigt zudem die Notwendigkeit einer langen Lernphase (vgl. [Barke, 1996]). Im Gegensatz dazu ist die gesamte CORBA Spezifikation in IDL auf ca. 178 Seiten definiert.

Die meisten CORBA-Produkte bieten ihre komplette Dokumentation auf weiteren 200 bis 300 Seiten an. CORBA-Schulungen werden von diversen Firmen angeboten und fast alle CORBA Händler bieten einwöchige Kurse an, deren Erfahrungen zeigen, dass der Benutzer das Erlernen und Anwenden als einfach empfindet.

Ein wichtiger Aspekt bei CORBA ist die Tatsache, dass der Systementwickler die Komplexität der API's für die Anwendungen kontrollieren kann. Mit Hilfe der OMG IDL kann er eine anwendungsspezifische Softwarearchitektur definieren. Diese Architektur kann auf die ausgewogenen Kosten für die Integration zurecht geschnitten sein. OMG IDL ist ein leistungsstarkes Werkzeug für die Definition von Softwarearchitekturen, deren Dokumentation und für die Auslieferung der resultierenden API's an große Softwareprojekte, die mehrere Entwicklungsorganisationen mit einbeziehen können.

16.9 Wiederverwendung von Entwicklungen und Anwendungen

Ein sehr hohes Maß an Wiederverwendbarkeit des Programmkodes und des Designs sind wichtige Eigenschaften bei der Reduktion von Kosten. Dabei spielen die objektorientierten Merkmale der Kapselung und die Fähigkeit der Vererbung eine große Rolle.

CORBA unterstützt die (mehrfache) Vererbung zumindest auf dem Level der Schnittstellen, was eine Wiederverwendung des Designs ermöglicht. Einige CORBA Produkte unterstützen zudem die Vererbung bei Ausführung zur Laufzeit, was zu einer direkten Wiederverwendung des Programmkodes führt. Die Sprache IDL besitzt ein starkes Potential für Wiederverwendung des Designs. Wie bereits dargelegt, handelt es sich bei IDL um eine deklarative Beschreibungssprache, die frei ist von Informationen über die Implementierung, die verwendeten Programmiersprachen, die Betriebssysteme und die Verteilung der einzelnen Objekte/Komponenten. Eine CORBA-basierte Anwendung kann durchaus in einem heterogenen Rechnerverbund laufen und in unterschiedlichen Sprachen implementiert sein; lediglich der

Kompilierlauf muss aufgrund der fehlenden Binärkompatibilität für die jeweilige Plattform angestoßen werden.

COM/DCOM unterstützt, wie schon erwähnt, die mehrfache Vererbung nicht, jedoch ein damit verbundenes Konstrukt, genannt Delegation oder Tie-Approach. Um den Programmkode wieder verwenden zu können, entwickelt der Programmierer eine Methodenimplementierung, die einen Aufruf von dem wieder verwendeten Kode enthält, der von einem anderen Objekt geliefert wird. Verglichen mit der Unterstützung bei objektorientierten Systemen ist dies eine deutlich schwächere Form der Wiederverwendbarkeit, zumal die Unterstützung weniger von COM/DCOM als vom Entwickler selbst ausgeht.

Es ist jedoch festzuhalten, dass das Prinzip der Delegation eine wichtige Rolle bei der Integration von Legacy-Applikationen spielen kann. Als Beispiel wurde bereits eine Schweizer Bank zitiert, die ihre alten PL/1-Applikationen unverändert ließen und stattdessen mittels der Delegation CORBA-konforme Objekte vorgeschaltet haben, die mit diesen Applikationen arbeiten. Aus Sicht der CORBA-Entwickler erscheinen diese PL/1-Applikationen wie ein CORBA-Objekt.

17 Einführung in die Web Services

Insbesondere die Web Services ähneln sehr dem dynamischen CORBA und verfügen zusätzlich über den Vorteil, auf etablierten Internettechnologien, die für fast alle Betriebssysteme vorhanden sind, aufzusetzen.

Nach der Etablierung von DCE RPCs, CORBA, Enterprise JavaBeans bzw. allgemeiner die Java-to-Enterprise-Edition (J2EE) erfolgt ggw. eine neue Welle der Interaktion von Prozessen auf der Basis der Beschreibungssprache XML. Im Gegensatz zu den vorherigen Technologien setzen die Webdienste auf allgegenwärtig vorhanden Transportprotokollen wie z. B. HTTP auf.

Besonders empfehlenswert zu diesem Bereich sind [Monson-Haefel, 2003], [Chappell et al., 2002], [Hendricks et al., 2001] und [Newcomer, 2002].

17.1 Einführungsbeispiel

In Analogie zu den Einführungsbeispielen in Kapitel 3.1 und 13.1 soll ein Web Service mit dem BEA Workshop-8.1 entwickelt werden (s. Abbildung 46).

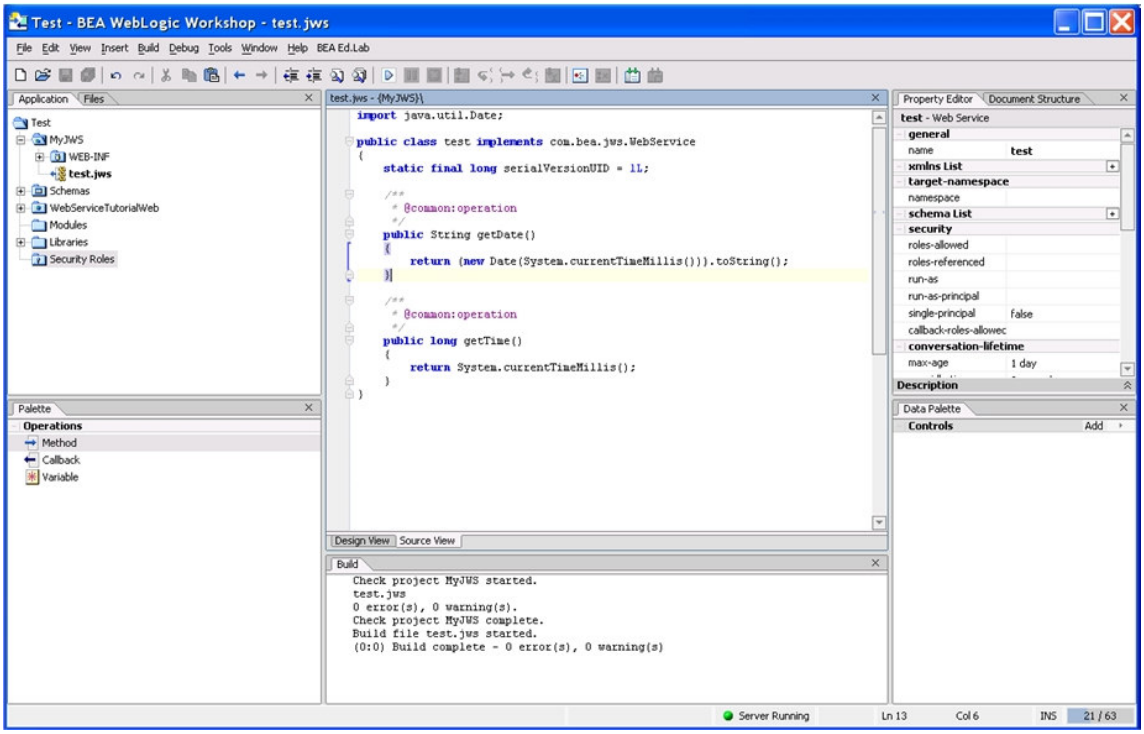


Abbildung 46: Java Web Services mit BEA Workshop

Der BEA Workshop generiert neben der Beschreibung der Schnittstellen auch einen Testclient. Nach Aufruf des Workshops wird eine neue Anwendung und in der Anwendung ein neues Projekt vom Typ Web Service angelegt. Durch ein Drag&Drop von *Method* in das Arbeitsfenster kann eine Web Service-Methode angelegt werden. In dem o. a. Beispiel sind dies die Methoden *public String getDate()* und *public long getTime()*. Die Implementierung der Methoden ist analog zu Java-Methoden und oben abgebildet.

Die vom Workshop generierte WSDL (Web Service Description Language), das Analogon zu einer IDL Beschreibung, lautet:

```
<?xml version="1.0" encoding="utf-8" ?>
    <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
                        xmlns:conv="http://www.openuri.org/2002/04/soap/conversation/"
                        xmlns:cw="http://www.openuri.org/2002/04/wsdl/conversation/"
                        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                        xmlns:jms="http://www.openuri.org/2002/04/wsdl/jms/"
                        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
                        xmlns:s="http://www.w3.org/2001/XMLSchema"
                        xmlns:s0="http://www.openuri.org/"
                        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                        targetNamespace="http://www.openuri.org/">
        <types>
                <s:schema xmlns:s="http://www.w3.org/2001/XMLSchema"
                          elementFormDefault="qualified"
                          targetNamespace="http://www.openuri.org/">
                        <s:element name="getDate">
                                <s:complexType>
                                        <s:sequence />
                                </s:complexType>
                        </s:element>
                        <s:element name="getDateResponse">
                                <s:complexType>
                                        <s:sequence>
                                                <s:element name="getDateResult"
                                                           type="s:string"
                                                           minOccurs="0" />
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                        <s:element name="string" nillable="true" type="s:string" />
                        <s:element name="getTime">
                                <s:complexType>
                                        <s:sequence />
                                </s:complexType>
                        </s:element>
                        <s:element name="getTimeResponse">
                                <s:complexType>
                                        <s:sequence>
                                                <s:element name="getTimeResult" type="s:long" />
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                        <s:element name="long" type="s:long" />
                </s:schema>
        </types>
        <message name="getDateSoapIn">
                <part name="parameters" element="s0:getDate" />
        </message>
        <message name="getDateSoapOut">
                <part name="parameters" element="s0:getDateResponse" />
        </message>
        <message name="getTimeSoapIn">
                <part name="parameters" element="s0:getTime" />
        </message>
        <message name="getTimeSoapOut">
                <part name="parameters" element="s0:getTimeResponse" />
        </message>
        <message name="getDateHttpGetIn" />
        <message name="getDateHttpGetOut">
                <part name="Body" element="s0:string" />
        </message>
        <message name="getTimeHttpGetIn" />
        <message name="getTimeHttpGetOut">
                <part name="Body" element="s0:long" />
        </message>
        <message name="getDateHttpPostIn" />
        <message name="getDateHttpPostOut">
                <part name="Body" element="s0:string" />
        </message>
        <message name="getTimeHttpPostIn" />
        <message name="getTimeHttpPostOut">
                <part name="Body" element="s0:long" />
```

```
</message>
<portType name="testSoap">
	<operation name="getDate">
		<input message="s0:getDateSoapIn" />
		<output message="s0:getDateSoapOut" />
	</operation>
	<operation name="getTime">
		<input message="s0:getTimeSoapIn" />
		<output message="s0:getTimeSoapOut" />
	</operation>
</portType>
<portType name="testHttpGet">
	<operation name="getDate">
		<input message="s0:getDateHttpGetIn" />
		<output message="s0:getDateHttpGetOut" />
	</operation>
	<operation name="getTime">
		<input message="s0:getTimeHttpGetIn" />
		<output message="s0:getTimeHttpGetOut" />
	</operation>
</portType>
<portType name="testHttpPost">
	<operation name="getDate">
		<input message="s0:getDateHttpPostIn" />
		<output message="s0:getDateHttpPostOut" />
	</operation>
	<operation name="getTime">
		<input message="s0:getTimeHttpPostIn" />
		<output message="s0:getTimeHttpPostOut" />
	</operation>
</portType>
<binding name="testSoap" type="s0:testSoap">
	<soap:binding  transport="http://schemas.xmlsoap.org/soap/http"
		style="document" />
	<operation name="getDate">
		<soap:operation soapAction="http://www.openuri.org/getDate"
			style="document" />
		<input>
			<soap:body use="literal" />
		</input>
		<output>
			<soap:body use="literal" />
		</output>
	</operation>
	<operation name="getTime">
		<soap:operation soapAction="http://www.openuri.org/getTime"
			style="document" />
		<input>
			<soap:body use="literal" />
		</input>
		<output>
			<soap:body use="literal" />
		</output>
	</operation>
</binding>
<binding name="testHttpGet" type="s0:testHttpGet">
	<http:binding verb="GET" />
	<operation name="getDate">
		<http:operation location="/getDate" />
		<input>
			<http:urlEncoded />
		</input>
		<output>
			<mime:mimeXml part="Body" />
		</output>
	</operation>
	<operation name="getTime">
		<http:operation location="/getTime" />
		<input>
			<http:urlEncoded />
		</input>
		<output>
			<mime:mimeXml part="Body" />
		</output>
	</operation>
</binding>
<binding name="testHttpPost" type="s0:testHttpPost">
	<http:binding verb="POST" />
```

```
                <operation name="getDate">
                        <http:operation location="/getDate" />
                        <input>
                                <mime:content type="application/x-www-form-urlencoded" />
                        </input>
                        <output>
                                <mime:mimeXml part="Body" />
                        </output>
                </operation>
                <operation name="getTime">
                        <http:operation location="/getTime" />
                        <input>
                                <mime:content type="application/x-www-form-urlencoded" />
                        </input>
                        <output>
                                <mime:mimeXml part="Body" />
                        </output>
                </operation>
        </binding>
        <service name="test">
                <port name="testSoap" binding="s0:testSoap">
                        <soap:address location="http://localhost:7001/MyJWS/test.jws" />
                </port>
                <port name="testHttpGet" binding="s0:testHttpGet">
                        <http:address location="http://localhost:7001/MyJWS/test.jws" />
                </port>
                <port name="testHttpPost" binding="s0:testHttpPost">
                        <http:address location="http://localhost:7001/MyJWS/test.jws" />
                </port>
        </service>
</definitions>
```

Beispiel 16: WSDL-Beschreibung des Timeservers

Im Vergleich hierzu sind Programmbeispiel 1 und Programmbeispiel 72 einfache und lesbare Schnittstellen.

Der generierte Testclient sendet das folgende Datenpaket an den Web Service:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <getDate xmlns="http://www.openuri.org/"></getDate>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Beispiel 17: Die zu sendende SOAP-Nachricht

Die Antwort vom Server ist in dem folgenden SOAP-Paket wiedergegeben:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <ns:getDateResponse xmlns:ns="http://www.openuri.org/">
                <ns:getDateResult>Mon Jan 03 12:09:17 CET 2005</ns:getDateResult>
        </ns:getDateResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Beispiel 18: Die SOAP-Antwort des Timeservers

Der tatsächliche Aufbau eines Klienten und die benötigten Technologien werden im Detail in den nachfolgenden Kapiteln beschrieben und mit weiteren Beispielen verdeutlicht.

17.2 Definition von Web Services

Der Bereich der Web Services ist zum gegenwärtigen Zeitpunkt relativ neu, verspricht aber eine der großen Entwicklungen für die nächsten Jahre im Bereich der lose gekoppelten verteilen Anwendungen zu werden. Es ist somit angebracht, die dynamischen Möglichkeiten von CORBA mit den Webdiensten zu vergleichen um Gemeinsamkeiten, aber auch Abgrenzungen zu eruieren.

Eine endgültige Definition über die Webdienste ist zurzeit nicht vorrätig. Aufgrund des großen Engagements von Firmen wie IBM, Sun Microsystems und Microsoft in diesem Bereich werden die von diesen Firmen angegebenen Definitionen bekannt gegeben und im Anschluss durch eine eigene Definition, die für die vorliegende Ausarbeitung relevant ist, ergänzt.

Sun Microsystems stellt unter [Sun, 2002] die folgende Definition für Webdienste bereit:

> „Web services are software components that can be spontaneously discovered, combined and recombined to provide a solution to the user's request/response. The Java™ language and XML are the prominent technologies for Web services."

Während Sun primär die eigenen Technologien als wegbereitend für die Webdienste beschreibt, wird IBM bereits konkreter. IBM liefert unter [IBM, 2002] eine sehr ausführliche Darstellung darüber, was ein Webdienst ist:

> „A Web service is an interface that describes a collection of operations that are network accessible through standardized XML messaging. Web services fulfill a specific task or a set of tasks. A Web service is described using a standard, formal XML notation, called service description that provides all of the details necessary to interact with the service, including message formats (that detail the operations), transport protocols, and location.
> The nature of the interface hides the implementation details of the service so that it can be used independently of the programming language in which it is written. This allows and encourages Web services based applications to be loosely coupled, component-oriented, cross-technology implementations. Web services can be used alone or in conjunction with other Web services to carry out a complex aggregation or a business transaction."

In aller Allgemeinheit erhält man bei der Definition bereits eine Vorstellung darüber, dass ein Webdienst durch eine standardisierte Beschreibungssprache spezifiziert wird und es wurde bereits erwähnt, das Webdienste in erster Linie lose gekoppelte, verteilte Anwendungen sind.

Von Seiten Microsoft werden zwei Definitionen über Webdienste zur Verfügung gestellt. Unter [Microsoft, 2002a] ist nachzulesen:

> „A Web service is a unit of application logic providing data and services to other applications. Applications access Web services via ubiquitous Web protocols and data formats such as HTTP, XML and SOAP, with no need to worry about how each Web service is implemented. Web services combine the best aspects of component-based development and the Web, and are a cornerstone of the Microsoft .NET programming model."

In dieser Definition beschreibt Microsoft bereits die verwendeten, grundlegenden Technologien, mit denen Webdienste untereinander kommunizieren. Konkreter ist die folgende, ebenfalls von Microsoft angegebene Definition, die sich unter [Microsoft, 2002b] befindet:

> „A Web service is a programmable application logic accessible using standard Internet protocols. Web services combine the best aspects of component-based development and the Web. Like components, Web services represent black-box functionality that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web services are not accessed via object-model-specific protocols, such as

the distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB-Protocol (IIOP). Instead, Web services are accessed via ubiquitous Web protocols and data formats, such as Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML). Furthermore, a Web Service interface is defined strictly in terms of the messages the Web service accepts and generates.

Consumers of the Web service can be implemented on any platform in any programming language, as long as they can create and consume the messages defined for the Web service interface."

Es ist offensichtlich, dass die o. a. Definitionen über Webdienste interessant sind, aber einen allgemeingültigen Konsens darüber, was Webdienste sind, nicht unmittelbar erkennen lassen. Für die vorliegende Ausarbeitung ist sowohl die Gemeinsamkeit von Webdiensten zum dynamischen CORBA, aber auch seine Abgrenzung von Interesse. Aus diesem Grund wird eine eigene Definition angestrebt, die für die weitere Ausarbeitung geeignet ist:

Definition 44: Web Services

Ein Webdienst (Web Service) ist eine plattform- und implementierungsunabhängige Softwarekomponente mit den folgenden Eigenschaften:

1. Die Beschreibung des Dienstes erfolgt über die standardisierte Beschreibungssprache WSDL (**W**eb**s**ervice **D**escription **L**anguage, s. Kapitel 17.3.2).
2. Der Dienst wird durch seine Registrierung veröffentlicht. Die Registrierung erfolgt auf Basis der Beschreibungssprache UDDI (**U**niversal **D**escription, **D**iscovery and **I**ntegration, s. Kapitel 17.3.4).
3. Das Auffinden des Dienstes erfolgt über einen standardisierten Mechanismus zur Kompilier-, oder zur Laufzeit.
4. Die Nutzung des Dienstes erfolgt über eine wohldefinierte, in XML beschriebene, Schnittstelle.
5. Jeder Dienst kann grundsätzlich andere Dienste benutzen.

Ein Web Service unterstützt grundsätzlich den RPC-Aufruf, also das synchrone Kommunikationsparadigma Anfrage/Antwort. In Abbildung 47 wird ein RPC-Aufruf dargestellt. Hier handelt es sich um einen Java Webservice, der Servlets und zustandstandslose SessionBeans als Endpunkte definieren kann. Die Aufgabe dieser Komponenten ist u. U. auch die Delegation an den eigentlichen Dienst.

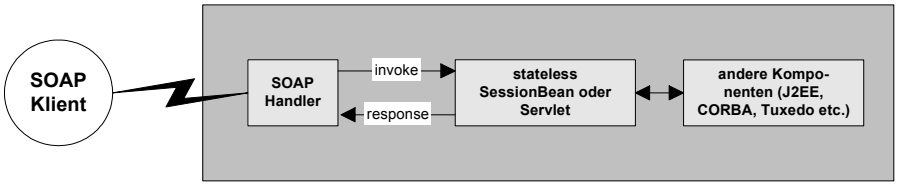


Abbildung 47: RPC-Aufruf bei Web Services

Zusätzlich besteht die Möglichkeit, ein Dokument asynchron an einen Dienst zu senden. In Abbildung 48 wird dies illustriert. Auch hier handelt es sich um einen Java Webservice, der die Komponente JMS für die asynchrone Kommunikation nutzt. Bei Verwendung von .NET, wäre stattdessen der Einsatz von MSMQ vorstellbar.

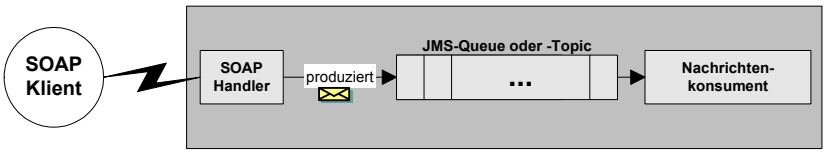


Abbildung 48: Asynchrone Kommunikation (SOAP with Attachment)

17.3 Benötigte Technologien für Webdienste

Die Idee von den Webdiensten stellt im Grunde genommen nichts Neues dar. Von den Prinzipien her betrachtet handelt es sich um die bereits im CORBA bekannten dynamischen Technologien. Neu ist lediglich, dass diese Technik internetfähig ist. Wurden im CORBA-Umfeld die entfernten Objekte in der Beschreibungssprache IDL bekannt gegeben, so findet jetzt eine Beschreibung auf Basis von WSDL statt. Diese Vorgehensweise wird konsequent fortgeführt. Anstelle der Datenübertragung auf Basis des Protokolls IIOP werden Daten durch das SOAP-Protokoll (**S**imple **O**bject **A**ccess **P**rotocol) übertragen. SOAP verwendet i. A. das Internetprotokoll HTTP und verpackt die Daten XML-konform. Das Zusammenspiel der einzelnen Komponenten ist in Abbildung 49 dargestellt.

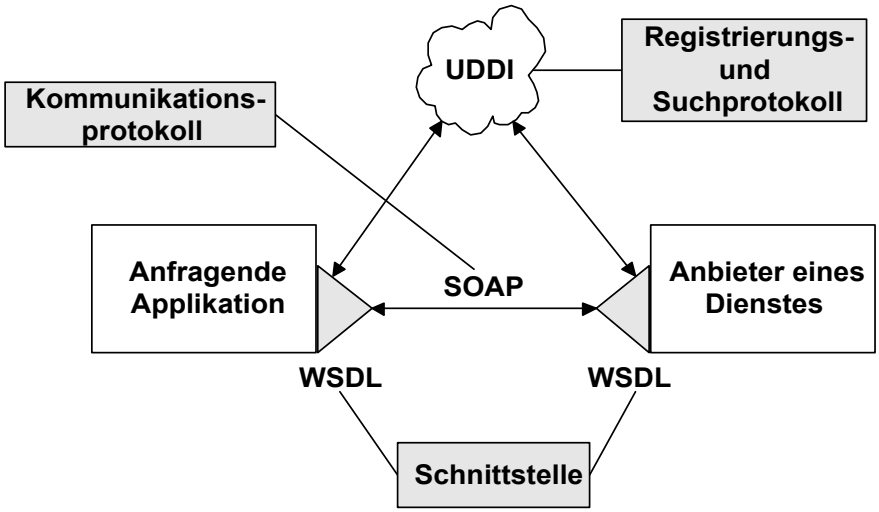


Abbildung 49: Die Technologien von Webdiensten

Das Auffinden von Webdiensten und die Beschreibung der angebotenen Dienste erfolgt anstelle der CORBA-Dienste Trader und Interface Repository durch das so genannte UDDI (**U**niversal **D**escription, **D**iscovery and **I**ntegration).

17.3.1 SOAP: Das Simple Object Access Protocol

Das **S**imple **O**bject **A**ccess **P**rotocol (SOAP) ist ein leichgewichtiges, XML-basiertes Protokoll für den Austausch von Daten in einem dezentralen, verteilten Umfeld. Die SOAP-Struktur besteht aus den folgenden Teilen:

1. Einem Umschlag (envelope), der die SOAP-Nachricht beschreibt. Insbesondere enthält der Umschlag

   a. eine Kopfstruktur (Header), in dem optional weitere Informationen für den nächstmöglichen Empfänger wie z. B. Sicherheitsinformationen, Transaktionskontexte usw. hinterlegt werden können. Dieser Kopf ist laut der Spezifikation optional. Ist er jedoch vorhanden, so muss er an der ersten Position des SOAP-Umschlages stehen.
   b. die eigentlichen Nutzdaten, eine Empfängeradresse und eine Angabe, wie die Nachricht zu verarbeiten ist.
2. Eine Menge von Darstellungsregeln, die angeben, in welchem Format die anwendungsspezifischen Daten vorliegen.
3. Eine Vereinbarung darüber, wie ein entfernter Funktionenaufruf dargestellt wird und welche Antwort zu erwarten ist.

Diese Informationen sind in einem MIME-Paket eingebettet und können auf Basis des **H**yper**t**ext **T**ransfer **P**rotocol (HTTP) oder einem anderen Webprotokoll wie z. B. SMTP (**S**imple **M**ail **T**ransfer **P**rotocol) oder FTP (**F**ile **T**ransfer **P**rotocol) verschickt werden.

Wird eine SOAP-Nachricht von einem Kommunikationspartner zum nächsten übertragen, so kann über den Kopf der Nachricht eine zusätzliche Information für einen von ggf. mehreren Empfängern hinterlegt werden. Es ist wichtig, hierbei festzuhalten, dass SOAP nicht unbedingt eine End-zu-End Kommunikation festlegt, sondern die Möglichkeit bietet, eine Kette von Empfängern zu beschreiben. Von dem Sender einer Nachricht kann festgelegt werden, welcher der Kommunikationspartner die Zusatzinformation erhalten und auswerten soll. Dazu existiert ein Attribut

SOAP:actor=URI.

Damit wird der Empfänger durch eine eindeutig bestimmte Adresse in Form eines Uniform Resource Identifiers (URI) festgelegt. Soll der nächste Kommunikationspartner die Information auswerten, so kann dies durch die spezielle URI

SOAP:actor=http://schemas.xmlsoap.org/soap/actor/next

festgelegt werden. Soll die Verarbeitung der Kopfinformation durch den Empfänger erzwungen werden, so kann der Sender das zusätzliche Attribut

SOAP:mustUnderstand=1|0

hinterlegen.

Ist z. B. eine Transaktionsinformation mit den beiden Attributen

SOAP:actor=http://schemas.xmlsoap.org/soap/actor/next
SOAP:mustUnderstand=1

hinterlegt, so muss diese Information vom nächsten Kommunikationspartner ausgewertet werden. Ist die Auswertung dieser Information nicht möglich, so muss ein so genannter SOAP-Fault gesendet werden. Diese SOAP-Fehlernachricht besteht aus den folgenden Kindelementen:

1. Fehlerkode (Faultcode): Dieser Wert ist für eine softwareseitige Auswertung vorgesehen.
2. Fehlerstring (Faultstring): Eine Information, die i. A. vom menschlichen Benutzer ausgewertet wird.
3. Fehlerverursacher (Faultactor): Die eindeutig bestimmte Adresse des Kommunikationspartners, der die Fehlermeldung verursacht hat. Die Adresse wird in Form eines URI bekannt gegeben.
4. Detail: Weitere Informationen.

Das folgende Beispiel zeigt eine gekürzte SOAP-Anfrage an eine Objektmethode bzw. eine Funktion, die in IDL deklariert ist als

float boersenkurs(in string symbol);

```
POST /StockQuote HTTP/1.1
Host: hostname
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
```

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schmas.xmlsoap.org/soap/envelope"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:boersenkurs xmlns:m="Some-URI">
               <symbol>BEAS</symbol>
        </m:boersenkurs>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Beispiel 19: SOAP-Envelope

Die Antwort könnte gekürzt wie folgt aussehen:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: nnnn
<SOAP-ENV:Envelope ...>
    <SOAP-ENV:Body>
        <m:boersenkursAntwort xmlns:m="Some-URI">
               <result>77.45</result>
        </m:boersenAntwort>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Beispiel 20: Mögliche XML-Antwort

17.3.1.1 JAXM und JAX-RPC

Die Java Gemeinde stellt mit den beiden JAXM und JAX-RPC zwei weitere Schichten für XML-Parser zur Verfügung, die speziell auf Web Services und somit für den Umgang mit SOAP und auch für die Stub-Generierung bei RPC-Aufrufen abgestimmt sind.

Die Namensgebung JAXM ist irreführend, da mit dieser Spezifikation der entfernte Methodenaufruf ebenso möglich ist, wie der asynchrone Nachrichtenaustausch.

Während ein Message Driven Bean das Interface *MessageListener* implementieren muss, hat die Gemeinde die beiden Schnittstellen *ReqRespListener* und *OnewayListener* deklariert, die ebenso wie *MessageListener* aus der einzelnen Methode *onMessage* bestehen. Diese Deklarationen sind notwendig, um dieser Methode verschiedene Rückgabewerte, die nicht Bestandteil der Methodensignatur sind, zuzuordnen. Mit der Schnittstelle *OnewayListener* wird als Rückgabewert *void* festgelegt und bei der *ReqRespListener*-Schnittstelle ist der Rückgabewert von *onMessage* vom Typ *Object*.

Die wichtigsten Eigenschaften von JAXM und JAX-RPC werden in der folgenden Tabelle zusammengefasst (vgl. [Hendricks, 2001]).

Eigenschaften	JAXM	JAX-RPC
SOAP 1.1 Unterstützung	ja	ja
SOAP with Attachment	ja	ja
Unterstützung von Industriestandards wie ebXML	ja	nein
Entfernte Funktionenaufrufe	ja	ja
WSDL-Stub-Generierung	nein	ja
Synchrone Kommunikation	ja	ja
Asynchrone Kommunikation	ja	nein
Qualität des Dienstes wie garantierte Nachrichtenzustellung	ja	nein

Tabelle 10: Eigenschaften von SOAP 1.0/1.1

Ein weiterer Unterschied zwischen den beiden Parsern ist in den unterschiedlichen Abstraktionsebenen zu sehen. JAXM verlangt von dem Entwickler mehr Aufwand als JAX-RPC.

Neben den Implementierungen der beiden Parser wird ein Servlet benötigt, das eine HTTP-Anfrage entgegennimmt und anschließend an den Empfänger weiterleitet.

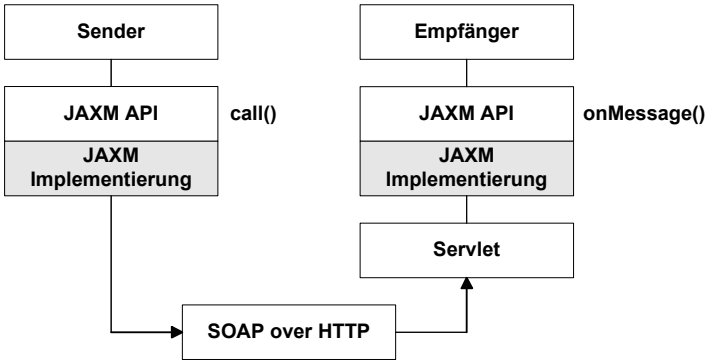


Abbildung 50: Benötigte Komponenten für JAXM

17.3.1.2 Beispiel für einen XML-RPC-Aufruf mit Web Services

Der folgende Programmkode zeigt den Aufbau eines XML-RPC-Klienten auf Basis von JAXM.

```
package de.uni.mainz.jaxm.client;

import java.io.*;
import java.util.*;
import javax.xml.soap.*;
import javax.xml.messaging.*;

public class JAXMClient {
    public JAXMClient(String[] args)
    {
        if (args.length == 1)
                hostURL = args[0];
        else
                hostURL = defaultHostURL;
    }

    public void send()
    {
        try {
                SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
                SOAPConnection        con = scf.createConnection();
                MessageFactory        mf  = MessageFactory.newInstance();
                SOAPMessage           msg = mf.createMessage();
                SOAPPart              sop = msg.getSOAPPart();
                SOAPEnvelope          env = sop.getEnvelope();
                SOAPBody              body = env.getBody();
                Name                  name = env.createName("Anfrage");
                SOAPBodyElement       bodyElement = body.addBodyElement(name);
                bodyElement.addText("Ein beliebiger Text");

                SOAPMessage reply = con.call(msg, new URLEndpoint(hostURL));

                reply.writeTo(System.out);
                con.close();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
    }

    public static void main(String[] args)
    {
```

```
        JAXMClient client = new JAXMClient(args);
        client.send();
        System.exit(0);
    }

    private String              hostURL             = null;
    private static final String defaultURI          = "urn:de-uni-mainz-jaxm-beispiel";
    private static final String defaultHostURL      =
                                    "http://localhost:8080/examples/servlets/Receiver";
}
```

Programmbeispiel 90: Klient eines Web Services

Der Empfänger dieses Aufrufs ist als ein Servlet implementiert.

```
package de.uni-mainz.jaxm.servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.xml.soap.*;
import javax.xml.messaging.*;

public class Receiver extends JAXMServlet implements ReqRespListener
{
    static MessageFactory mf = null;

    static {
        try {
                mf = MessageFactory.newInstance();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
    }

    public SOAPMessage onMessage(SOAPMessage msg)
    {
        try {
                msg.writeTo(System.out);

                SOAPMessage           msg = mf.createMessage();
                SOAPPart              sop = msg.getSOAPPart();
                SOAPEnvelope          env = sop.getEnvelope();

                SOAPBody              body = env.getBody();
                Name                  name = env.createName("Antwort");
                SOAPBodyElement       bodyElement = body.addBodyElement(name);

                bodyElement.addText("Hier kommt die Antwort");
                return msg;
        }
        catch (Exception e) {
                System.err.println("Fehler bei der Aufbereitung der Antwort");
                return null;
        }
    }
}
```

Programmbeispiel 91: Empfänger einer SOAP-Nachricht

17.3.2 WSDL: Die Web Service Description Language

Die Beschreibungssprache WSDL (Web Service Description Language) ist eine XML-basierte Spezifikation zur Beschreibung von Webdiensten. Insbesondere werden mit WSDL analog zur Interface Definition Language die entfernten Zugänge zu einer Anwendung beschrieben, also die jeweiligen Methoden/Funktionen, ihre Rückgabewerte und die Ein- und Ausgabeparameter.

```
<portType name="AddressBookPortType">
       <operation name="getPerson">
              <input message="tns:getPersonRequest"/>
              <output message="tns:getPersonResponse"/>
       </operation>
</portType>
```

Beispiel 21: PortType in einer WSDL

In aller Allgemeinheit besteht die WSDL-Beschreibung aus:

1. Typenelementen: Hier werden die Vereinbarungen über die verwendeten Datentypen/Datenstrukturen eines Webdienstes deklariert.
2. Nachrichtenelement (message element): Die Nachrichtenelemente beschreiben die Ein-, Ausgabe- und Rückgabedaten einer jeden Methode/Funktion, die ein Webdienst zur Verfügung stellt. Hierzu wird für jede Methode/Funktion eine Anfrage- und Antwortennachricht vereinbart.
3. Porttyp: Beschreibt die Menge von Operationen, die von einem oder mehreren Einstiegspunkten des Dienstes zur Verfügung gestellt werden und die hierzu korrespondierenden Anfrage- und Antwortennachrichten.
4. Bindung: Die Bindung beschreibt das Protokoll und die Datenformate für diesen Einstiegspunkt.
5. Port: Der Port ist die Lokation des Endpunktes des Dienstes und wird durch eine Bindung und durch eine Netzadresse eindeutig definiert.
6. Dienst (service): Beschreibt einen oder eine Menge von Endpunkten, die eine bestimmte Funktionalität abarbeiten.

```
<service name="AddressBook">
    <documentation>Adressbuchabfrage</documentation>
    <port name="AddressBookPort" binding="tns:AddressBookBinding">
           <soap:address location="http:localhost/address/addressuri"/>
    </port>
</service>
```

Beispiel 22: Service in einer WSDL

In Abbildung 51 wird die WSDL-Struktur dargestellt:

Im Allgemeinen werden die WSDL-Beschreibungen durch ein geeignetes Werkzeug basierend auf den vorher getätigten Vereinbarungen generiert. Beispielsweise bietet der Weblogic Server 6.1 bzw. 7.0 von BEA Systems die folgende Unterstützung an:

1. Eine SOAP-Implementierung basierend auf einem vordefinierten Servlet, das eine Integration mit zustandslosen SessionBeans oder nachrichtenorientiert mit dem Java Message Service ermöglicht.
2. SOAP Aufrufe werden automatisch auf ein zustandsloses SessionBean (RPC) abgebildet oder über JMS auf ein Message Driven Bean abgebildet.
3. Die aufgerufenen Komponenten können alle anderen Komponenten des Weblogic Servers nutzen. Damit ist potentiell die Möglichkeit gegeben, eine bereits bestehende Webanwendung mit geringem Aufwand zusätzlich als Webdienst anzubieten.

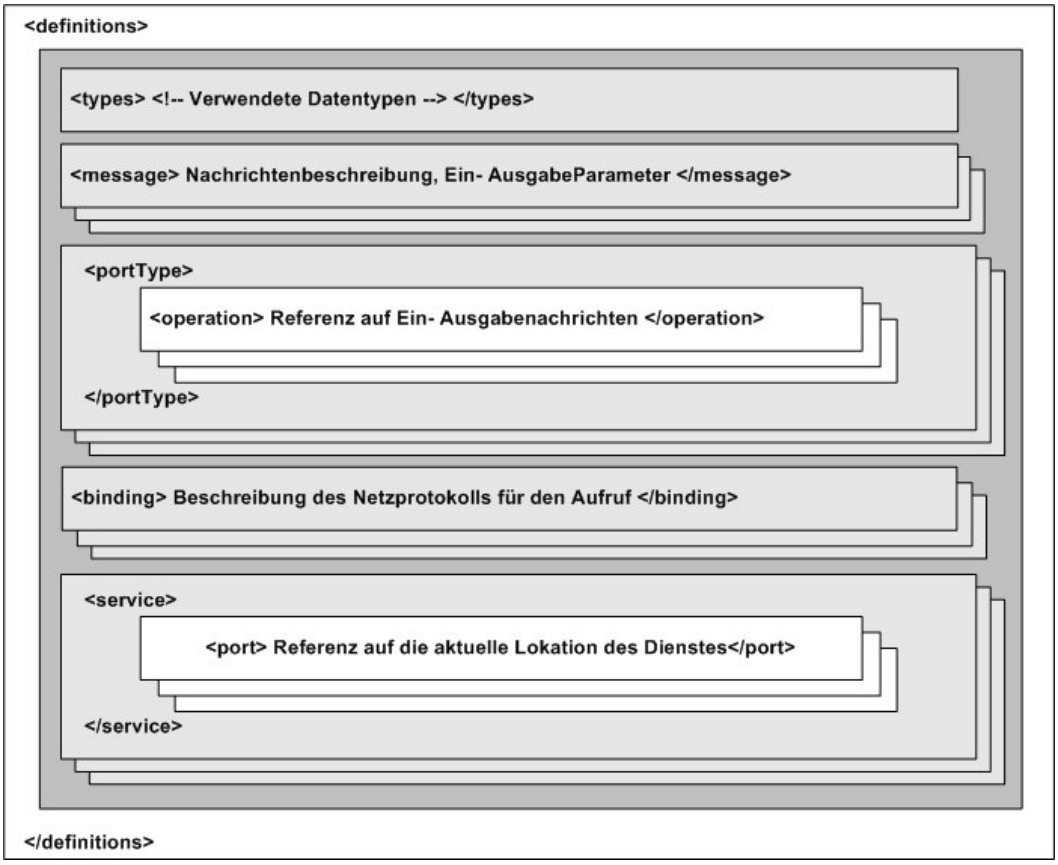


Abbildung 51: Struktur eines WSDL-Dokumentes

Das folgende Beispiel zeigt eine WSDL-Beschreibung für die Abfrage eines Adressbuches.

```
<?xml version="1.0"?>
<definitions
    targetNamespace="java:de.consavis.address"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="java:de.consavis.address">
    <types>
        <schema targetNamespace="java:de.consavis.address"
                xmlns="http://www.w3.org/1999/XMLSchema"
                xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                xmlns:soap="http://schemas.xmlsoap.org/soap/encoding/"
                xmlns:address="java:de.consavis.address">
                <complexType name="Person">
                        <attribute name="name" type="string"/>
                        <attribute name="vorname" type="string"/>
                        <attribute name="str" type="string"/>
                        <attribute name="plz" type="long"/>
                        <attribute name="ort" type="string"/>
                </complexType>
        </schema>
    </types>
    <message name="getPersonRequest">
        <part name="arg0" type="xsd:string" />
    </message>
    <message name="getPersonResponse">
        <part name="return" type="tns:Person" />
    </message>
    <portType name="AddressBookPortType">
```

```
        <operation name="getPerson">
                <input message="tns:getPersonRequest"/>
                <output message="tns:getPersonResponse"/>
        </operation>
    </portType>
    <binding name="AddressBookBinding" type="tns:AddressBookPortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getPerson">
                <soap:operation soapAction="urn:getPerson"/>
                <input>
                        <soap:body use="encoded" namespace='urn:AddressBook'
                        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
                </input>
                <output>
                        <soap:body use="encoded" namespace='urn:AddressBook'
                        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
                </output>
        </operation>
    </binding>
    <service name="AddressBook"><documentation>todo</documentation>
        <port name="AddressBookPort" binding="tns:AddressBookBinding">
                <soap:address location="http://localhost/address/addressuri"/>
        </port>
        </service>
</definitions>
```

Beispiel 23: WSDL für eine Adressbuchabfrage

17.3.3 Beispiel eines Webdienstes mit einem zustandslosen SessionBean

Das folgende Beispiel beschreibt einen Webdienst, der durch ein zustandsloses Session-Bean implementiert ist. Die Funktionalität besteht lediglich in einer einzelnen Methode, die eine Temperaturangabe an den Aufrufer zurückgibt. Der Aufruf erfolgt durch einen Klienten unter Verwendung des entfernten Methodenaufrufs.

```
// WeatherBean.java: Implementierung der Methode getTemp() des RemoteInterfaces und
//                   Implementierung der Methoden des SessionBean-Interfaces

package examples.webservices.rpc.weatherEJB;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class WeatherBean implements SessionBean {

    // Implemenierung der Methoden des SessionBean-Interfaces

    public void ejbActivate()
    {
        log("ejbActivate called");
    }

    public void ejbRemove()
    {
        log("ejbRemove called");
    }

    public void ejbPassivate()
    {
        log("ejbPassivate called");
    }

    public void setSessionContext(SessionContext ctx)
    {
        log("setSessionContext called");
        this.ctx = ctx;
    }

    public void ejbCreate () throws CreateException
```

```
    {
        log("ejbCreate called");

    }

    public float getTemp(String ZipCode)
    {
        float result;


        log("getTemp called");
        if (ZipCode.equals("90210")) {
               result = 77.0f;
        }
        else {
               result = -273.15f;
        }
        return result;
    }


    private void log(String s)
    {
        if (VERBOSE)
               System.out.println(s);
    }


    private static final boolean VERBOSE = true;
    private SessionContext ctx;
    private int tradeLimit;
}
```

Programmbeispiel 92: Stateless Bean als Web Service Endpunkt

Von BEAs Weblogic Server 6.1 wurde die folgende WSDL-Beschreibung generiert:

```
<definitions
       targetNamespace="java:examples.webservices.rpc.weatherEJB"
       xmlns="http://schemas.xmlsoap.org/wsdl/"
       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/1999/XMLSchema"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:tns="java:examples.webservices.rpc.weatherEJB">

       <types>
       <schema targetNamespace='java:examples.webservices.rpc.weatherEJB'
               xmlns='http://www.w3.org/1999/XMLSchema'>
       </schema>
       </types>

       <message name="getTempRequest">
       <part name="arg0" type="xsd:string" />
       </message>
       <message name="getTempResponse">
       <part name="return" type="xsd:float" />
       </message>
       <portType name="WeatherPortType">
       <operation name="getTemp">
              <input message="tns:getTempRequest"/>
              <output message="tns:getTempResponse"/>
       </operation>
       </portType>
       <binding name="WeatherBinding" type="tns:WeatherPortType">
       <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="getTemp">
              <soap:operation soapAction="urn:getTemp"/>
              <input><soap:body use="encoded" namespace='urn:Weather'
                           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
              </input>
              <output><soap:body use="encoded" namespace='urn:Weather'
                           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
              </output>
       </operation>
       </binding>
       <service name="Weather">
```

```
        <documentation>Kommentarzeilen ...</documentation>
        <port name="WeatherPort" binding="tns:WeatherBinding">
               <soap:address location="http://localhost:7001/weather/weatheruri"/>
        </port>
        </service>
</definitions>
```

Beispiel 24: WSDL-Beschreibung für einen zustandslosen Dienst

Der Klient ist durch die folgende Java-Applikation gegeben:

```
package examples.webservices.rpc.javaClient;

import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import examples.webservices.rpc.weatherEJB.Weather;

public class Client {
    public static void main( String[] args) throws Exception
    {
        String url   = "http://localhost:7001";
        Properties h = new Properties();

        h.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.soap.http.SoapInitialContextFactory");
        h.put("weblogic.soap.wsdl.interface", Weather.class.getName() );
        h.put("weblogic.soap.verbose", "true" );

        Context context = new InitialContext(h);
        Weather service = (Weather) context.lookup(url +
                                              "/weather/statelessSession.WeatherHome/" +
                                              "statelessSession.WeatherHome.wsdl");
        System.out.println("Temperatur: " + service.getTemp("90210"));
    }
}
```

Programmbeispiel 93: Klient für einen XML-RPC

17.3.4 UDDI: Das Universial Description, Discovery and Integration

Für den Erfolg von Web Services ist es unerlässlich, einen Dienst jedem potentiellen Klienten weltweit, betriebssystem- und sprachunabhängig zur Verfügung zu stellen.

Dazu wurde in Zusammenarbeit der Unternehmungen Ariba, IBM und Microsoft die Registrierungseinheit UDDI (**U**niversial **D**escription, **D**iscovery and **I**ntegration) entwickelt. Referenzimplementierungen u. a. für private Nutzung finden sich unter

http://www-3.ibm.com/services/uddi,
http://uddi.microsoft.com/default.aspx,
und
http://uddi.hp.com/uddi/index.jsp.

Das grundlegende Konzept der Web Service Technologie wird durch die SOA (**S**ervice **O**riented **A**rchitecture) beschrieben, in der die drei Rollen Dienstnehmer (service requestor), Dienstleister (service provider) und der Dienstvermittler (service broker) definiert werden.

In dem vorliegenden Kapitel ist der Dienstvermittler von Interesse. Repräsentiert wird der Vermittler durch Registrierungseinheiten, die von außen durch standardisierte Schnittstellen angesprochen werden.

Die Aufgabe eines UDDI besteht darin, dass verfügbare Web Services notwendige Informationen für die Klienten registrieren können und die Dienste somit auffindbar und nutzbar sind.

Das UDDI besteht aus den folgenden Spezifikationen:

1. Eine Spezifikation bzgl. der verwendeten Datenstrukturen, in der beschrieben wird, welche Daten im UDDI registriert werden. Diese Beschreibungen basieren auf XML und sind somit programmiersprachenneutral. Die Datenstrukturen sind durch XML Schemata hinterlegt, in denen der Strukturaufbau festgelegt wird.
2. Die Standardisierung der Schnittstellen, mit denen auf das UDDI zugegriffen wird. Zu unterscheiden sind grundsätzlich die beiden Zugriffsarten
   a. Veröffentlichung von Diensten (publishing). Die Erzeugung, Löschung und Modifikation von Diensten erfolgt immer auf Basis des Protokolls HTTPS und vorangegangener Authentifizierung.
   b. Suchen nach bestehenden Diensten (inquiry). Hiermit wird ausschließlich lesend auf das UDDI zugegriffen.

   Das API (**A**pplication **P**rogramming **I**nterface) ist ebenfalls sprachunabhängig. Erreicht wird dies, indem die Anfrage und die Antwort ausschließlich als XML-Dokumente erfolgen. Die ggw. Referenzimplementierungen des UDDI von HP, Microsoft und IBM gestatten einen Zugriff lediglich auf Basis des Protokolls SOAP über HTTP(S), wobei die Anfragedaten und Antworten durch den SOAP-Briefumschlag gekapselt sind.
3. Die Spezifikation über die Replizierung enthält Beschreibungen darüber, wie UDDIs untereinander Daten replizieren.
4. Zu guter Letzt existiert eine Spezifikation darüber, wie UDDI gewartet und betrieben wird.

Die Spezifikation liegt ggw. In der Version 2 vor; die Referenzimplementierungen basieren jedoch auf der Version 1.0 bzw. auf einer Betaversion von 2.

17.3.5 Die Datenstrukturen des UDDI

Das UDDI enthält Informationen über Unternehmungen und über Dienste, die bestimmte Geschäftsprozesse dem Klienten anbieten (vgl. Abbildung 52).

Die Daten, die ein UDDI beinhaltet, sind strukturiert durch:
1. Die weißen Seiten (white pages): Hier können Informationen wie z. B. Name und Adresse einer Unternehmung, sowie Telefon-, Faxnummer und Email-Adresse hinterlegt werden.
2. Die gelben Seiten (yellow pages): Die gelben Seiten enthalten Informationen über die Art der geschäftlichen Prozesse. Dem Sucher wird erlaubt, sortiert nach Unternehmensbereichen oder nach Kategorien zu suchen.
3. Die grünen Seiten (green pages): Hier werden die Dienste, die bestimmte Geschäftsprozesse anbieten, beschrieben. Dies beinhaltet technische Informationen, die notwendig sind um diesen Dienst zu nutzen. Von Interesse ist hierbei zumindest die Lokation des Dienstes und die WSDL-Beschreibung, um dem Klienten die Informationen über die Schnittstellen, Eingabeparameter und Rückgabewerte des Dienstes zu geben.

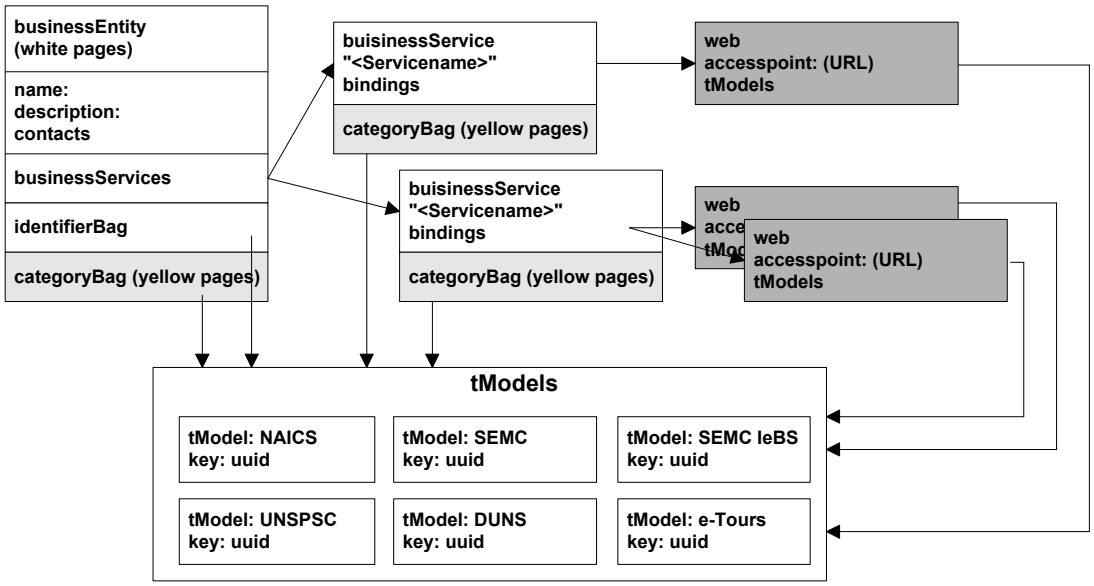


Abbildung 52: Zusammenspiel der drei UDDI-Rollen

In den XML-Dokumenten wird dies beschrieben durch die Formatangaben <businessEntity>, <businessServices>, <bindingTemplates>, <categoryBag>, <identifierBag> und <tModel>, auf die im folgenden genauer eingegangen wird.

Innerhalb eines UDDI wird immer mit eindeutig bestimmten Identifizierern gearbeitet, die i. d. R. automatisch anhand vorgegebener Information generiert werden. Für die Identifier wird das aus dem DCE bekannte UUID (**U**niversal **U**nique **Id**entifier) verwendet. Der Identifizierer wird in einem der folgenden vordefinierten Namensräume erzeugt:

1. UNSPSC (**Un**iversal **S**tandard and **P**roducts **S**pe**c**ification): Dieser Standard enthält Definitionen für unterschiedliche Produkte.
2. NAICS (**N**orth **A**merica **I**ndustry **C**lassification **S**ystem): Dieser Standard definiert ein Klassifikationssystem für geschäftliche Aktivitäten, basierend auf den Industriezweigen, auf denen operiert wird.
3. ISO 3166 (**G**eographic **C**lassification **S**ystem): Dieser Standard beschreibt verschiedene Regionen und Lokationen weltweit.
4. Andere: Ist zuständig, für allgemeine, typisierte Schlüsselwörter.

Von besonderem Interesse ist hierbei das Technologie Modell, das im UDDI stets als tModel bezeichnet wird.

17.3.6 Das Technologie Modell tModel

Das tModel ist eines der Schlüsselelemente in der UDDI-Spezifikation. In einem tModel können technische und nichttechnische Informationen hinterlegt werden. Die technischen Informationen können bspw. eine WSDL-Beschreibung eines Dienstes oder zusätzliche Informationen zum Datenaustausch beinhalten.

Jedes tModel besitzt einen eindeutig bestimmten Bezeichner und enthält einen Verweis auf eine Spezifikation, in der Detailinformationen beschrieben sind.

```
<tModel authorizedName="demo"
      operator="/operator/services/uddi"
      tModelKey="UUID:7C91BE10-EA7F-11D5-A6D4-9EAB24E10238">
      <name>AddressBuch tModel</name>
      <overviewDoc>
        <overviewURL>http://localhost/Address.wsdl</overviewURL>
      </overviewDoc>
</tModel>
```

Beispiel 25: tModel im UDDI

Ein tModel beinhaltet zusätzlich Informationen über Bezeichner und Kategorien.

17.3.7 Kategorien und Identifizierer

In einem UDDI kann eine Reihe von technischen und nichttechnischen Informationen enthalten sein. Dies impliziert, dass ein Benutzer oder ein Programm, das Informationen erfragen will, geeignete Suchkriterien anwenden muss.

Die UDDI-Elemente *identifier* und *category* sind ein Versuch, dieses Problem zu lösen. Bei diesen beiden Elementen handelt es sich nicht um tatsächlich gespeicherte Entitäten, sondern vielmehr um konzeptionelle Elemente. Eingeleitet werden Angaben zu Identifizierern und Kategorien durch die Formatangaben <identifierBag> und <categoryBag>. Der Identifizierer erlaubt, in einem UDDI-Eintrag weitere Informationen in Form von qualifizierten Namens-Werte-Paaren zu hinterlegen. D. h., jeder Identifizierer besitzt einen Schlüssel und einen hierzu korrespondierenden Wert. Diese Information ist jedoch unzureichend, da nicht ersichtlich wird, von welcher Art der Schlüssel ist.

Bspw. besitzt jede größere amerikanische Unternehmung eine DUNS-Zahl (Dun & Bradstreet **D**ata **U**niversal **N**umbering **S**ystem), durch die diese Firma bei allen geschäftlichen Transaktionen identifiziert wird. Für derartige Firmen wäre es demzufolge sinnvoll, diese eindeutig bestimmte DUNS-Zahl in einem UDDI-Eintrag zu speichern. Die Vorgehensweise wäre, einen Identifizierer zu erzeugen und den Wert des Schlüssels mit der DUNS-Zahl gleichzusetzen.

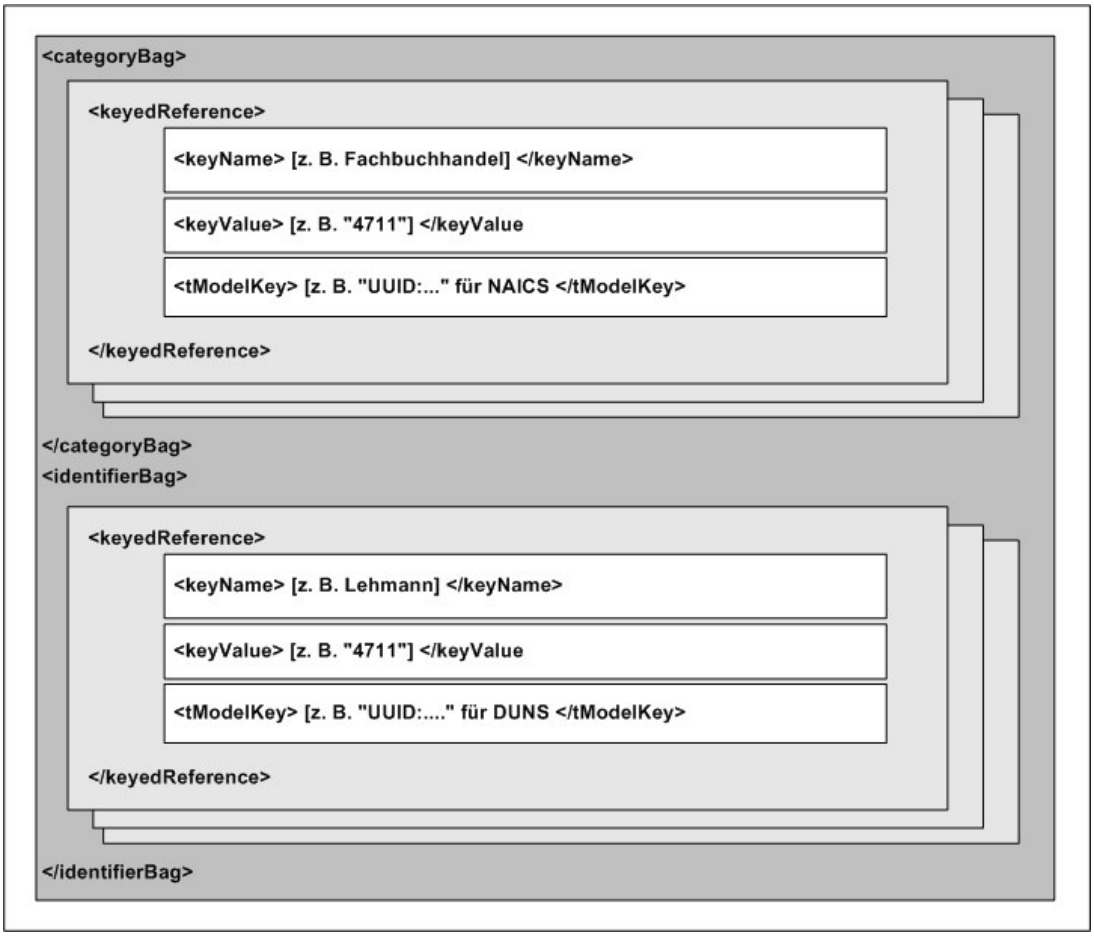


Abbildung 53: Aufbau eines Identifiers und einer Kategorie

17.3.8 Das Element <businessEntity>

Das Wurzelelement einer jeden UDDI-Hierarchie ist das Element <businessEntity>. Es enthält

1. Informationen über die Unternehmung, wie z. B. Name und Adresse der Unternehmung, Kontaktpersonen etc., also die weißen Seiten.
2. die Elemente <categoryBag> und <identifierBag>, um hiermit den Eintrag zu kategorisieren.
3. eine Liste von angebotenen Diensten.

```
<element name="businessEntity">
    <complexType>
        <sequence>
                <element ref="discoveryURLs"         minOccurs="0"/>
                <element ref="name"                  minOccurs="0"
                                                     maxOccurs="unbounded"/>
                <element ref="description"           minOccurs="0"
                                                     maxOccurs="unbounded"/>
                <element ref="contacts"              minOccurs="0"/>
                <element ref="businessServices"      minOccurs="0"/>
                <element ref="identifierBag"         minOccurs="0"/>
                <element ref="categoryBag"           minOccurs="0"/>
        </sequence>
        <attribute ref="businessKey"  use="required"/>
```

```
        <attribute ref="operator"/>
        <attribute ref="authorizedName"/>
    </complexType>
</element>
```

Beispiel 26: businessEntity im UDDI

In der nachfolgenden Abbildung wird dies illustriert:

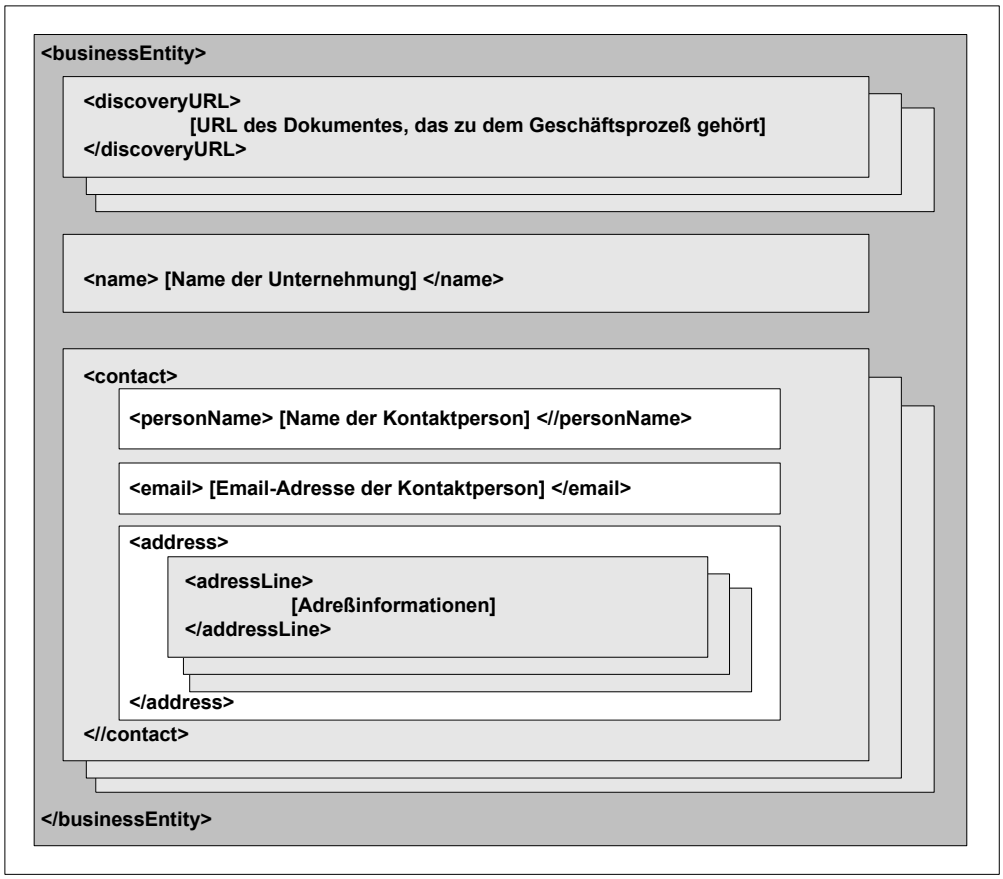


Abbildung 54: Darstellung des Geschäftsentitäten-Elementes

In der o. a. Abbildung wurde aus Gründen der Einfachheit auf die Elemente *Identifizierer* und *Kategorie* verzichtet.

In dem nachfolgenden Beispiel wird ein kurzer Auszug über eine Geschäftsentität als XML-Dokument angegeben.

```
<businessEntity businessKey="4C890B30-EB6D-11D5-A947-97BA25522F30">
    <discoveryURLs>
        <discoveryURL useType="businessEntity">
               http://localhost/services/uddi/uddiget?businessKey=...
        </discoveryURL>
    </discoveryURLs>
    <name>CONSAVIS GmbH</name>
    <description>
        IT-Beratung, Schulung und Entwicklung, Schwerpunkt Middleware
    </description>
    <contacts>
        <contact>
               <personName>Karsten Walpert</personName>
               <email>karsten.walpert@consavis.de</email>
```

```
                <address>
                        <addressLine>Lessingstr. 6</addressLine>
                        <addressLine>65232 Taunusstein</addressLine>
                        <addressLine>Germany</addressLine>
                </address>
        </contact>
    </contacts>
    <categoryBag>
        <keyedReference
                keyName="IT-Beratung, Entwicklung, Middleware etc."
                keyValue="4711"
                tModelKey="UUID: ..."/>
    </categoryBag>
<businessEntity>
```

Beispiel 27: businessEntity

Das o. a. Element <discoveryURL> wird bei der Erzeugung des UDDI-Eintrages generiert und zeigt an, wie eine Kopie des Eintrages durch eine einfache HTTP-GET-Anfrage zu erhalten ist. Zusätzliche URLs, die andere Dokumente über diesen Eintrag an den Benutzer liefern, können definiert werden.

17.3.9 Weitere Elemente des UDDI

Sobald eine Geschäftsentität registriert wird, können die zugehörigen Dienste beschrieben werden. Dazu dient die Formatangabe <businessService>; innerhalb dieses Elementes werden ein Dienstname, eine optionale Beschreibung, ein Kategorie-Element und eine Liste von Bindungselementen beschrieben.

```
<businessService
        businessKey="4C890B30-EB6D-11D5-A947-97BA25522F30"
        serviceKey="47FC56D0-ECD0-11D5-8EA6-926A0B6F3A63">
    <name>AddressBook Service</name>
    <description>Dienst fuer eine Abfrage von Adressen</description>
    <bindingTemplates/>
    <categoryBag>
        <keyedReference
                keyName="Administration des Dienstes"
                keyValue="80161501"
                tModelKey="UUID:DB77450D-9FA8-A7BC-04411D14E384"/>
    </categoryBag>
</businessService>
```

Beispiel 28: businessService

Das Attribut *serviceKey* enthält als Wert einen eindeutig bestimmten Schlüssel, der automatisch beim Erzeugen des Eintrages generiert wird und der den Eintrag in der Registrierungseinheit repräsentiert.

Das Attribut *businessKey* indiziert, zu welchen Geschäftsentitäten dieser Eintrag gehört.

Das Element *bindingTemplate* ist im Zusammenhang mit der Dienstbeschreibung optional und kann als leeres Element hinterlegt werden. Ist dieses Element nichtleer, so enthält es i. A. die beiden folgenden Kindelemente:

1. Zugriffspunkt: Wird eingeleitet durch die Formatangabe <accessPoint> und ist vom Typ String. Damit kann eine beliebige Information hinterlegt werden. In der o. a. Abbildung wurde eine URL hinterlegt, andere Informationen wie z. B. eine Telefonnummer oder eine Email-Adresse sind ebenfalls vorstellbar.
2. Detailbeschreibung des tModels: Eingeleitet wird dies durch die Formatangabe <tModelInstanceDetails>. Wie der Name vermuten lässt enthält dieses Element einen Verweis auf den zugehörigen tModel-Eintrag im UDDI. Von Interesse für den Zusammenhang eines UDDI-Eintrages mit einer WSDL-Beschreibung ist das Element <overviewURL>, das

einen Verweis auf eine zusätzliche Beschreibung eines tModels, z. B. in Form einer WSDL, ist.

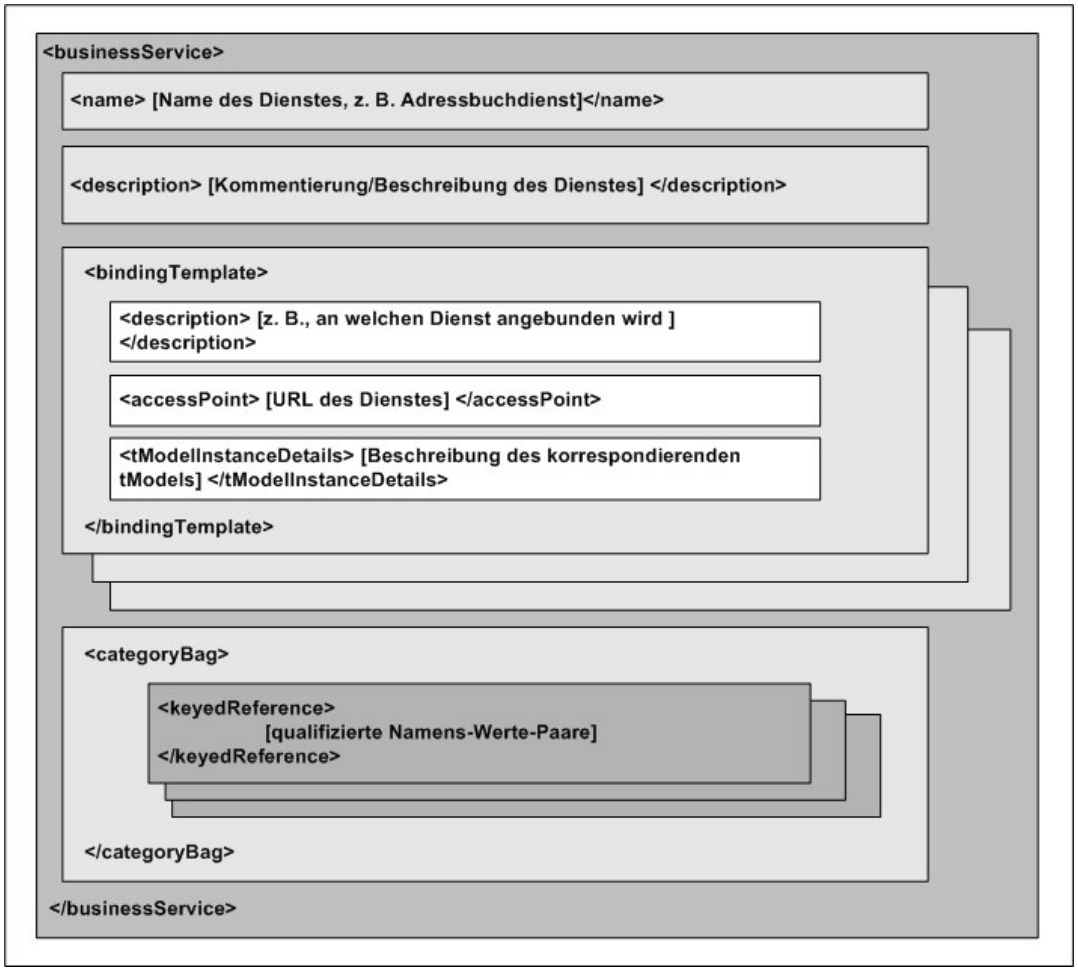


Abbildung 55: Aufbau des Geschäftsdienstes

Ein Beispiel für das Element <bindingTemplate> für das Adressbuch könnte wie folgt aussehen:

```
<bindingTemplate
       bindingKey="48A91190-ECD0-11D5-8EA6-926A0B6F3A63"
       serviceKey="47FC56D0-ECD0-11D5-8EA6-926A0B6F3A63">
    <accessPoint URLType="http">
       http://localhost/soap/servlet/rpcrouter
    </accessPoint>
    <tModelInstanceDetails>
       <tModelInstanceInfo
              tModelKey="UUID:4795F3E0-ECD0-8EA6-926A0B6F3A63">
              <instanceDetails>
                     <overviewDoc>
                            <overviewURL>
                                   http://localhost/wsdl/AddressBookService.wsdl
                            </overviewURL>
                     </overviewDoc>
              </instanceDetails>
       </tModelInstanceInfo>
    <tModelInstanceDetails>
</bindingTemplate>
```

Beispiel 29: bindingTemplate

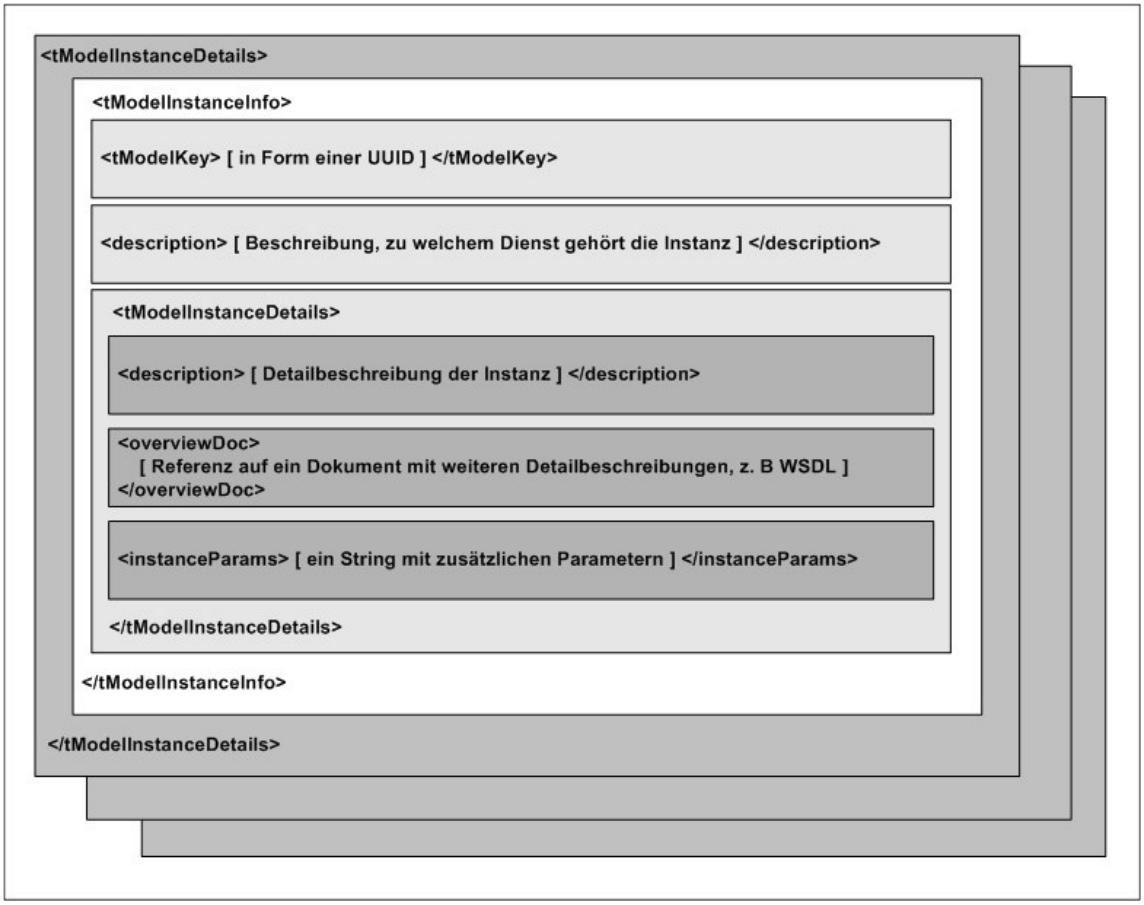


Abbildung 56: Aufbau des Elementes <tModelInstanceDetails>

18 CORBA vs. Web Services

In dem Kapitel über die Web Serivces wurde bereits dargelegt, dass diese Dienste auf Basis des Protokolls SOAP ein starkes Wachstum im Zusammenhang mit der Technologie von Verteilungsplattformen zu verzeichnen hat. Auf den ersten Blick scheinen Web Services mit CORBA weitestgehend identisch zu sein. IIOP lässt sich mit SOAP vergleichen, das UDDI entspricht einer Kombination aus Trader oder Namensdienst und einem Interface Repository und WSDL lässt sich leicht mit IDL vergleichen. Zusätzlich ist bei beiden Technologien die wichtige Forderung nach Sprachunabhängigkeit erfüllt.

Die wichtigsten Eigenschaften sind in Tabelle 11 zusammengefaßt.

CORBA	Web Services
IDL	WSDL
Stubs/Skeleton	SOAP Nachricht (XML eingebettet in den SOAP-Umschlag)
Statischer Stub	Möglich
Dynamische Komponenten durch Any, DynAny, DII, IFR	Durch WSDL, UDDI und xsd:any
CDR (Marshalling/Unmarshalling)	Unicode
IFR, DII, Trader (bzw. Naming Service)	UDDI
GIOP (IIOP)	http(s), smtp, ftp
Kommunikationart (RMI)	RPC, Soap With Attachment
I. d. R. keine Internetfähigkeit	Internetfähig

Tabelle 11: CORBA vs. Web Services

18.1 IIOP vs. SOAP

Es ergeben sich jedoch auch gravierende Unterschiede bzgl. der Protokolle und der gesamten Vorgehensweise, um einen Dienst oder ein Objekt entfernt nutzen zu können. In der Praxis werden bzgl. der Protokollunterschiede die folgenden Fragen diskutiert:

1. Nachrichtengröße: IIOP ist ein binäres Protokoll und somit effizienter in der Datenübertragung als ein zeichenorientiertes Protokoll wie SOAP. Eine ganze Zahl als long-Wert übertragen benötigt ungeachtet des Wertes genau vier Bytes. Im SOAP-Protokoll wird die Größe durch die Anzahl der Bytes bestimmt, die notwendig sind, um jede Ziffer der Zahl als ein ASCII-Zeichen darzustellen. Ein weiterer Faktor für die Nachrichtengröße bei SOAP ist die Vereinbarung über die Datenkodierung, also ob ein Byte für eine Ziffer ausreicht, oder ob mit dem Unicode-Zeichensatz gearbeitet wird.

   Es wird sicher viele Anwendungen geben, bei denen die Nachrichtengröße irrelevant ist und somit mag dieses Argument nicht unbedingt negativ für SOAP ausfallen. Die größte und populärste verteilte Anwendung ist nach wie vor das World-Wide-Web mit einer Vielzahl unterschiedlichster zu übertragender Daten auf Basis des zeichenorientierten Protokolls HTTP. Bei anderen Anwendungen, wie z. B. das Mobile-Computing, ist die Nachrichtengröße durchaus von Interesse und ein Einsatz von SOAP ist ggw. illusionär.
2. Die Komplexität der Datenserialisierung: Die Übertragung von Daten erfordert aufgrund der häufig vorhandenen Heterogenität ein Marshalling und Unmarshalling der Daten. Zumindest eine Serialisierung der Daten und Objekte ist vorzunehmen. Im Zusammenhang mit CORBA wurde bereits dieser Prozess dargelegt und eine Einführung in das CDR vorgenommen. Es wurde ersichtlich, dass es sich hierbei um einen beträchtlichen Aufwand handelt, der die Verarbeitungsgeschwindigkeit drastisch reduzieren kann. Die Komplexität von CDR ist hierbei in einem mittleren Bereich anzusiedeln. Die Serialisie-

rung von primitiven Datentypen ist trivial, die Serialisierung von komplexen und u. U. rekursiven TypeCodes oder valuetypes wird i. d. R. kompliziert sein.

Im Gegensatz hierzu ist der Aufwand der Serialisierung beim SOAP-Protokoll um ein Vielfaches höher und komplizierter. Der Klient muss seine Eingabedaten unter Berücksichtigung der syntaktischen Regeln des verwendeten XML-Dokumentes geeignet einstellen und der Empfänger einer Nachricht muss durch einen XML-Parser die syntaktische Korrektheit des Dokumentes prüfen und die Daten extrahieren. Bei der Antwort vom Dienst zu dem Klienten ist die gleiche Vorgehensweise zu sehen. Das SOAP-Protokoll erinnert hierbei stark an die Möglichkeiten des dynamischen CORBA; die Verarbeitungsgeschwindigkeit wird also auch hier stark reduziert werden.

3. Lesbarkeit der Nachrichten: Ein häufig vorgetragenes Argument gegen binäre Protokolle ist, dass die Datenpakete ohne geeignete Werkzeuge nicht für einen menschlichen Betrachter lesbar sind, während das Lesen eines XML-kodierten Datenpaketes auch ohne Kenntnis der DTD oder der XML-Schemata möglich ist. Bei Kenntnis des syntaktischen Aufbaus und der Bedeutung von Formatangaben kann der menschliche Betrachter ohne große Mühen auch die Semantik des Paketes verstehen.

   Dieses Problem ergab sich bisher bei allen binären Protokollen, die i. d. R. auf TCP/IP zurückgeführt werden. Entsprechende Programme zum Mitprotokollieren und –lesen der zu übertragenen Daten werden zum Teil von den Herstellern mit ausgeliefert oder sind im Internet frei verfügbar. Grundsätzlich ist dieser Vorteil von SOAP gegenüber IIOP nicht unbedingt von praktischer Relevanz.

Bzgl. der SOAP-Protokolle zeichnet sich ggw. ein ähnlicher Trend ab wie bei den CORBA-Produkten der Spezifikation 1.0. Zu Beginn der neunziger Jahre verfügten die vorhandenen CORBA-Produkte ausschließlich über proprietäre Protokolle. Erst mit der CORBA Spezifikation 2.x wurde das Protokoll IIOP standardisiert und somit die dringend benötigte Interoperabilität zwischen den einzelnen Produkten ermöglicht.

Im SOAP-Umfeld existieren ggw. über 60 verschiedene Implementierungen, die nicht immer vollständig oder korrekt sind. Es kann davon ausgegangen werden, dass die wichtigsten Implementierungen interoperabel zueinander sind. Es zeigt jedoch, dass dieser Standard noch nicht vollständig ausgereift ist und dass in naher Zukunft neue SOAP-Standardisierungen zu erwarten sind. Im Gegenzug hierzu ist CORBA bzgl. aller wichtigen Komponenten, wie z. B. POA oder dem ORB, standardisiert. Dies führt auch zu einem hohen Maß an Portabilität der Objektimplementierungen, jedoch nicht der Serverregistrierung.

Eine wichtige Frage die immer wieder vernommen wird ist, ob SOAP mit CORBA interoperieren kann. Dies ist grundsätzlich der Fall. Eine Reihe von ORB-Herstellern hat bereits eine Unterstützung des SOAP-Protokolls angeboten. Iona hat diesbezüglich ein SOAP-Plug-in für Orbix 2000 entwickelt und die Interoperabilität mit Microsoft nachgewiesen, BEA hat zwischenzeitlich sein proprietäres Produkt Tuxedo in den Versionen ab 7.1 mit XML-Fähigkeit ausgestattet, ebenso den WebLogic Server ab Version 6.1 und Microsoft weist mit seiner .Net-Initiative ebenfalls in die Richtung von Web Services und dem SOAP-Protokoll.

Die wirklich wesentliche Frage muss sein, welchen Mehrwert SOAP gegenüber IIOP besitzt und weshalb es sinnvoll sein kann, einen SOAP-Klienten zu entwickeln, der CORBA-Objekte oder Enterprise JavaBeans kontaktiert. Von Interesse sind hierbei die beiden folgenden Vorteile von SOAP gegenüber IIOP:

1. Die Firewall-Problematik: Es existiert ggw. kein Standardport für IIOP und somit ergeben sich die entsprechenden Nachteile bei Einsatz von CORBA in einem WAN mit vorgeschalteter Firewall. Die OMG hat in einem ersten Versuch eine Spezifikation herausgegeben, die aber in einem zweiten Ansatz überarbeitet wird und somit noch nicht verabschiedet ist. Proprietäre Lösungen des Firewall-Problems werden z. B. von Iona (Iona's WonderWall) und Inprise (VisiBroker GateKeeper) angeboten.

Hier kommt die Stärke des SOAP-Protokolls zum Tragen. Im Allgemeinen basiert SOAP auf HTTP und verwendet somit ebenfalls den Port 80, der i. d. R. von den Firewall-Administratoren freigeschaltet ist

2. Interoperabilität mit nicht CORBA-basierten Systemen: Wie bereits eine Reihe von Vorgängern (DCE, CORBA, J2EE) wird SOAP von den führenden Produktherstellern akzeptiert und unterstützt. Aufgrund der Tatsache, dass jedes neuere Betriebssystem ohnehin eine Unterstützung für HTTP anbietet, ist es ausreichend, SOAP-Unterstützung durch geeignete Bibliotheken mit auszuliefern. Die Interoperabilität mit bereits bestehenden Anwendungen hängt ggw. von dem verwendeten Produkt ab. Iona bietet mit seinem XML-Bus eine einfache Anbindung von SOAP an CORBA an, BEA's WebLogic Server 6.1 bietet Unterstützung für zustandslose SessionBeans und über JMS eine Unterstützung von Message Driven Beans an. Die Open Source Produkte Tomcat und Axis erlauben als Endpunkte des Web Services Java-Klassen oder auch Enterprise JavaBeans, wobei jedoch Zusatzprodukte benötigt werden. Die Integration von Altsystemen ist technisch ebenfalls lösbar, aber zurzeit fehlen die Werkzeuge, um die Komponenten wie WSDL usw. zu generieren, bzw. die existenten Werkzeuge wie z. B. BEA Workshop 8.1 sind hochgradig proprietär und die generierten Pakete sind lediglich im BEA WebLogic Server 8.1 ablauffähig.

18.2 Technologie-Evolution durch Web Services

Die angegebenen Definitionen von Web Services zeigen, dass es sich immer noch um einen evolutionären Prozess handelt. Festzuhalten ist lediglich, dass ein Web Service allgemein anerkannte und unterstützte Technologien wie XML und HTTP verwendet. Damit ergibt sich, dass Web Services drei wichtige Technologien repräsentieren:

1. Das World-Wide-Web: Die Webdienste repräsentieren die stetige Weiterentwicklung des World-Wide-Webs von einer Browser-orientierten interaktiven Anwendung hin zu dem wichtigen Bereich von Anwendungs-zu-Anwendungs-Koppelung.
2. Traditionelle Middleware: Web Services sind ein weiterer Schritt in der Weiterentwicklung von Middleware. Nach der Entwicklung von Sun RPCs, DCE, DCOM, CORBA, Message Oriented Middleware und J2EE sind die Webdienste eine weitere, internetfähige Technologie im Bereich der verteilten Verarbeitung.
3. Business-to-Business: Bei einer Kooperation verschiedener Unternehmungen ist es sinnvoll, wenn die Beteiligten Teile ihrer Software unter Einhaltung von Sicherheitsvorkehrungen freigeben, damit der Geschäftspartner auf Basis der Metadaten die entfernte Anwendung für sich nutzen kann.

Insbesondere der letzte Punkt B2B eröffnet im Internet eine Vielzahl von Möglichkeiten. Anhand eines einfachen Beispiels soll dies illustriert werden:

Der Wagenhersteller W hat aus Kostengründen seine Lagerhaltung abgeschafft und benötigt zu einem bestimmten Zeitpunkt n-Reifensätze, die unmittelbar zu den Montagehallen gebracht werden sollen. Der Wagenhersteller kann über ein ihm bekanntes UDDI verschiedene Reifenhersteller und deren Web Services ermitteln und auf Basis von Software können Preis-/Leistungsverhältnis und Verträge ausgehandelt werden. Sobald ein Vertrag abgschlossen ist, kann der jeweilige Reifenhersteller wieder einen Web Service von einer Spedition ermitteln und dort einen Vertrag für die Zulieferung abschließen.

Bei Verwendung von anderen Technologien müsste man voraussetzen, dass auf beiden Seiten bestimmte Softwarekomponenten der verwendeten Technologie vorhanden sind. Bei den Web Services reichen die die beschriebenen Internettechnologien, die i. d. R. immer mit der Hardware und dem Betriebssystem ausgeliefert werden, aus.

Allerdings ist mit Web Services das grundlegende Problem über die Semantik der erhaltenen Daten und somit die automatisierte Weiterverarbeitung ebenfalls nicht gelöst.

19 Zusammenfassung und Ausblick

In der vorliegenden Dissertation wurden die vier führenden Technologien und (De facto-) Standards bzgl. verteilter Anwendungen/Systeme, CORBA, COM/DCOM, J2EE und Web Services im Detail vorgestellt.

19.1 Gegenüberstellung

Die wichtigsten Eigenschaften der Middleware-Technologien CORBA, J2EE, COM/DCOM und Web Services werden in diesem Kapitel gegenübergestellt und beschrieben (s. Tabelle 12):

a. Kommunikationsmechanismen: Um einen Vergleich zwischen den einzelnen Technologien zu ermöglichen, werden nur die Kommunikationsmechanismen der EJBs aus der J2EE-Technologie berücksichtigt. Bei CORBA-, COM/DCOM-Objekten und den Enterprise JavaBeans, mit Ausnahme der Message Driven Beans, findet die Kommunikation über einen entfernten Methodenaufruf statt. Dazu werden grundsätzlich ein Stub und ein Skeleton erzeugt. Dieser Mechanismus wird auch bei entfernten Funktionenaufrufen, wie z. B. dem XML-RPC, verwendet.

   Bei der synchronen Kommunikation ist die externe Objekt- bzw. Dienstbeschreibung unerlässlich und wird von den jeweiligen Technologien immer angestrebt.

   Auch die asynchrone Kommunikation setzt bei CORBA und den Web Services ebenfalls eine externe Beschreibung der Schnittstellen und der zu übertragenden Daten voraus. Bei CORBA-basierten Anwendungen kann dies z. T. durch eine IDL-Beschreibung stattfinden oder durch Metadaten aus dem Interface-Repository.

   Bei Verwendung von JMS werden Nachrichten als Text-, Stream-, Object-, Bytes- oder MapMessages versendet und erst zur Laufzeit wird gezielt abgefragt, um welchen Nachrichtentyp es sich handelt. Welchen Inhalt die Nachrichtentypen konkret besitzen, ist im Vorfeld festzulegen.

   Bei der asynchronen Kommunikation verfügen sowohl die Web Services, als auch J2EE über einen Messaging-Dienst, wobei das Messaging für Web Services von der verwendeten Technologie abhängt. Bei der Verwendung der dynamischen Schnittstelle strebt CORBA kein Message-Passing bei der asynchronen Kommunikation an, sondern verwendet auch hier das Prinzip des entfernten Methodenaufrufs. Die asynchrone Kommunikation ist sowohl bei den Web Services als auch bei J2EE-Anwendungen gegeben. Ein Messaging-Dienst, wie z. B. JMS, ist bei CORBA Implementierungen des 2.x-Standards nicht vorgesehen. Eine Stärke bei der Verwendung von DII als asynchrone Kommunikation stellt die Gruppierung von mehreren Anfragen zu einer Anfrage zwecks Netzoptimierung dar.

   Der in CORBA 3.x beschriebene Messaging-Dienst ist gemessen an JMS allgemeingültiger und komplizierter. Implementierungen des 3.x-Standards existieren ggw. nicht.

b. Dynamisches Verhalten: Sowohl mit CORBA als auch mit Web Services ist die Möglichkeit gegeben, mit unbekannten Objekten/Diensten zu arbeiten, deren Aufbau erst zur Laufzeit durch Metadaten interpretiert wird. Enterprise JavaBeans stellen in den jeweiligen Home- bzw. RemoteInterface Metadaten zur Verfügung, jedoch ist der Inhalt beschränkt auf zusätzliche Informationen zu einem bestimmten Bean. Das Arbeiten mit unbekannten Objekten ist in der beschriebenen Art und Weise nicht möglich.

Speziell mit den generischen Datentypen von CORBA kann eine stabile Schnittstelle beschrieben werden. Bspw. können in Polizeisystemen die Schnittstellen als Rückgabewert oder für die out- bzw. inout-Parameter den Datentyp Any vereinbaren. Auch wenn grundlegende Datenstrukturen sich ändern, bleibt die Schnittstelle bzgl. ihrer Signatur stabil. Zusätzlich könnte durch die Direktive *#pragma version* die Schnittstelle mit einer Version versehen werden, so dass auch ein Klient, der nicht den neuesten Softwarestand beinhaltet, mit dieser Schnittstelle in seiner Version arbeiten. Speziell CORBA erlaubt mit den TypeCodes die Identifizierung der tatsächlichen Daten. Analog kann mit COM/DCOM-Applikationen gearbeitet werden. Die Implikation ist jedoch in einer Herabsetzung der Verarbeitungsgeschwindigkeit zu sehen.

Für J2EE-Applikationen ist lediglich die Möglichkeit gegeben, die Parameter einer Methode als *Object* anzugeben, mit der Konsequenz, dass durch *instance of* der richtige Datentyp ermittelt werden muss. Dies beinhaltet jedoch, dass zur Kompilierzeit bereits alle erdenklichen Datentypen bekannt sind.

c. Echtzeitfähigkeit. Diese Eigenschaft hängt von vielen verschiedenen Aspekten ab. Es muss an dieser Stelle unterstellt werden, dass die Architektur und die Implementierung zu keinerlei Beanstandung Anlass geben. Die Echtzeitfähigkeit bei statischem CORBA und bei J2EE-Anwendungen ist gegeben. Bei den dynamischen CORBA-Komponenten oder bei Web Services ist dies i. d. R. nicht der Fall.
d. Protokolle: CORBA verwendet ausschließlich das IIOP-Protokoll, während Java Komponenten wie Servlets oder EJBs zusätzlich das Protokoll RMI/JRMP verwenden können. Für Web Services kann in der Theorie jedes Protokoll vereinbart werden. I. d. R. werden es jedoch HTTP(S), ftp oder SMTP sein.
e. Komponententechnologie: CORBA 2.x stellt i. d. R. keine Komponenten zur Verfügung. Transaktionsverarbeitung, Persistenz etc. sind von den Entwicklern selbst zu programmieren. Bei den Web Services hängt die zur Verfügungstellung von Komponenten von der Umgebung ab. Bspw. kann der BEA WebLogic Server ab Version 6.1 zustandslose Sessionbeans oder Message Driven Beans als Endpunkte für den Web Service generieren und somit setzt dieser Dienst auf Komponenten auf. Vorstellbar ist aber auch ein Servlet als Endpunkt. In diesem Fall kann der Dienst keine für Komponenten vorhandenen Automatismen verwenden.
f. Einsatzgebiet: Aufgrund der Echtzeitfähigkeit im statischen Fall eignen sich CORBA, COM/DCOM und J2EE grundsätzlich für eine enge Koppelung, also für verteilte Anwendungen/Systeme mit hohem Durchsatz und der Forderung nach geringen Antwortzeiten.

   Durch den Messaging Service bei J2EE und der dynamischen Schnittstelle von CORBA können beide Technologien auch bei loser Koppelung zum Einsatz kommen.

   Web Services schränken sich durch ihren generischen Ansatz bzgl. verteilter Anwendungen mit Echtzeitanforderung stark ein. Das bevorzugte Einsatzgebiet sollte der Bereich Business-to-Business (B2B) oder bei geringen Anforderungen an die Verarbeitungsgeschwindigkeit der Bereich Enterprise-Application-Integration (EAI) sein.
g. Sprachunabhängigkeit: CORBA unterstützt bereits sechs Sprachen, für die eine standardisierte Sprachabbildung von IDL auf diese Sprache definiert ist. Andere Sprachabbildungen, wie z. B. IDL auf PL/1 oder IDL auf C#, sind zwischenzeitlich realisiert, aber nicht standardisiert. J2EE verwendet als einzige Entwicklungssprache Java, während die Web Services vollständig sprachunabhängig sind. Die einzigen Voraussetzungen bei den Web Services sind, dass die Implementierungssprache XML und das vereinbarte Protokoll, i. d. R. HTTP(s), unterstützen.
h. Plattform- und Betriebssystemunabhängigkeit: CORBA, J2EE und die Web Services sind i. d. R. von den jeweiligen Plattform/Betriebssystemen unabhängig. Welche Plattformen unterstützt werden, hängt speziell bei CORBA von dem jeweiligen Produkt ab. Bspw. ist BEA Tuxedo 8.1 für alle gängigen UNIX-Derivate inklusive LINUX, die Microsoft Betriebssysteme NT und Windows 2000/2003, aber auch für Exoten wie Open VMS verfügbar. Iona Technologies unterstützt zusätzlich auch IBM-Großrechner etc. J2EE ist

auf allen Plattformen lauffähig, für die es ein geeignetes JDK in der geforderten Version gibt. Bspw. läuft der BEA WebLogic Server neben den Windows Betriebssystemen und allen erdenklichen UNIX-Derivaten auch unter Open VMS, das u. a. vom Bundeskriminalamt oder der Schweizer Börse eingesetzt wird. Für COM/DCOM ist die Plattform i. d. R. auf Windows-Betriebssysteme beschränkt. Die von der Software AG vorgenommene Portierung von COM/DCOM auf UNIX beinhaltet nur das Kernsystem und ist für die Praxis eigentlich bedeutungslos. Die Web Services sind plattformunabhängig! Beschränkt man sich speziell auf Microsofts .NET und die neue Sprache C#, so sind bereits erste Anstrengungen einer Portierung auf UNIX unternommen worden (vgl. Mono-Projekt).

i. Einfachheit der Entwicklung: Bei CORBA, COM/DCOM oder den EJBs ist die Entwicklung i. d. R. einfach. Speziell bei CORBA hängt dies auch von der verwendeten Implementierungssprache ab. Die Sprachabbildung von IDL auf C++ ist deutlich komplizierter als die Abbildung auf Java. Man muss jedoch unterstellen, dass jeder Entwickler, der C++ beherrscht, auch in einem kurzen Zeitraum die Sprachabbildung erlernt und fehlerfrei anwenden kann. Bei den Web Services ist man aufgrund der komplexen XML-Beschreibungen von SOAP, UDDI und der WSDL von einem Werkzeug wie z. B. BEA Workshop 8.1 abhängig, das die Komplexität von XML und den XML-Schemas verbirgt. Andere Werkzeuge wie z. B. Eclipse oder Borlands JBuilder verfügen über Plugins, mit denen die Komplexität handhabbar wird.

j. Einfachheit der Administration: Die Administration und ihre Einfachheit ist von den jeweiligen Produkten abhängig. Ein Musterbeispiel für mächtige und zugleich einfache Administration stellt BEA Tuxedo dar. Hier existiert genau eine Konfigurationsdatei, in der relativ einfach festgelegt wird,
   - welche globalen Parameter für alle Beteiligten gelten (z. B. die Anzahl zulässiger, globaler Transaktionen zu einem Zeitpunkt etc.).
   - welche Rechner an der Anwendung beteiligt sind.
   - welche Prozesse/Objekte auf welchen Rechnern platziert werden.
   - welche Eigenschaften diese Prozesse/Objekte besitzen (Anzahl der Replikate, bestimmte Umgebungsvariablen etc.).

   usw.. Alle Prozesse laufen unter der Kontrolle von Tuxedo, so dass ein Ausfall sofort erkannt wird und entsprechend reagiert werden kann. Zusätzlich hat der Administrator die Möglichkeit mit normalen UNIX-Skripten die Administration weitestgehend zu automatisieren.

k. Einfachheit der Erlernbarkeit: CORBA verwendet für die Beschreibung von Objekten und Diensten eine einfache Beschreibungssprache, die von jedem, der Grundkenntnisse in C oder C++ besitzt, leicht erlernbar ist. Die Einfachheit dieser Sprache ist zusätzlich eine sehr gute Dokumentationsmöglichkeit und auch als Nachschlagewerk für Entwickler in hohem Maße geeignet. Problematisch ist für CORBA u. U. die Sprachabbildung von IDL auf eine Zielsprache. Die Erlernbarkeit dieser sprachlichen Abbildung hängt eindeutig von der Zielsprache ab. Bei Verwendung von COM/DCOM sieht ein Entwickler sich mit einer Unmenge von API's konfrontiert, deren Handhabbarkeit aufgrund der Komplexität nur schwer zu meistern ist. Für die Erlernbarkeit von J2EE muss die jeweilige Subtechnologie betrachtet werden. Bei Enterprise JavaBeans sind die Restriktionen für einen Entwickler lediglich in den zu implementierenden Interfaces und den Deployment Deskriptoren zu sehen. Ansonsten verfügt der Entwickler analog zu CORBA über alle Freiheiten. Bei anderen Subtechnologien wie z. B. Servlets oder JavaServer Pages ist eine von SUN standardisierte Spezifikation, die auch eine Vielzahl vordefinierter Schnittstellen beinhaltet, zu berücksichtigen. Web Services sind i. d. R. nur mit geeigneten Werkzeugen handhabbar. Ein einfacher Vergleich zwischen einer IDL-Beschreibung und der hierzu korrespondierenden WSDL-Beschreibung zeigt deutlich den Unterschied zwischen der Lesbarkeit und somit Erstellbarkeit von IDL-Beschreibungen versus WSDL-Beschreibungen.

l. Transparenzeigenschaften: Sind eine grundlegende Voraussetzung für verteilte Verarbeitung und sind bei allen Technologien gegeben.

m. Dienste: Zumindest die notwendigen Dienste wie Namensdienst usw. sind in der einen oder anderen Form bei allen Technologien vorhanden.
n. Internetfähigkeit: Ist bei J2EE und den Web Services uneingeschränkt gegeben. Bei CORBA und COM/DCOM werden Binärprotokolle verwendet, die bei einem Einsatz im Internet problematisch sein können.
o. Laufzeitumgebung für den Klienten: Ist bei CORBA und COM/DCOM unerlässlich, bei Web Services und J2EE kann ein einfacher Webbrowser u. U. ausreichend sein.
p. Standard: CORBA und J2EE sind standardisierte Technologien. Bei den Web Services ist eine endgültige Standardisierung durch das *Web Service Basic Profile 1.0* eingeleitet, aber noch nicht endgültig verabschiedet. COM/DCOM muss als nicht standardisiert gelten.
q. Clustering: Diese Eigenschaft hängt von den jeweiligen Produkten ab. Bspw. ist eine CORBA-basierte Anwendung mit BEA Tuxedo 8.1, die sich auf mehrere Rechner verteilt, automatisch in einem Cluster. D. h. ein Ausfall von Prozessen/Objekten oder eines physikalischen Rechners wird von Tuxedo erkannt und nachfolgende Anfragen werden automatisch an entsprechende Replikate weitergeleitet. Ebenso wird erkannt, ob der ausgefallene Rechner wieder verfügbar ist und ob diese Maschine wieder als Teilnehmer dem Cluster zugefügt wird. Diese Vorgänge können durch einen Administrator auch manuell eingeleitet werden. Bei anderen Produkten wie z. B. ORBacus von Iona Technologies sind derartige Eigenschaften nicht gegeben.
r. Integration: Aufgrund der Sprachunabhängigkeit ist CORBA für die Integration von Altanwendungen i. d. R. gut gerüstet. Als ein herausragendes Beispiel gilt die CORBA-konforme Einbindung von PL/1-Anwendungen, die auf einem Großrechner laufen. Dazu hat eine Schweizer Bank von Iona Technologies einen IDL-Kompiler entwickeln lassen, der IDL auf PL/1 abbildet. Damit kann die Altanwendung als CORBA-Objekt interpretiert werden und über das Protokoll IIOP sowohl von J2EE-, als auch CORBA-Klienten transparent adressiert werden. Die Java-Gemeinde setzt bzgl. der Integration auf JCA, die Java-Connector-Architecture. Beispiel von Konnektoren sind die Einbindung von SAP/R3 oder die Einbindung von CICS-Anwendungen auf einem IBM-Großrechner.

	CORBA	**COM/DCOM**	**J2EE**	**Web Services**
Kommunikationsmechanismen:	Remote Method Invocation.	Remote Procedure Call oder Messaging	Remote Method Invocation bei EJBs oder Messaging. Ansonsten abhängig von der Teiltechnologie.	Remote Procedure Call bzw. Messaging.
Asynchrone Kommunikation möglich?	CORBA 2.x bedingt (nur DII). Ab CORBA 3.x	ja (MSQ)	ja (JMS).	ja (SOAP with Attachment).
Externe Beschreibung der Schnittstelle und der zu übertragenden Daten bei synchroner Kommunikation?	ja (IDL).	ja (MIDL)	ja (RemoteInterface).	ja (WSDL).
Externe Beschreibung der Schnittstelle und der zu übertragenden Daten bei asynchroner Kommunikation?	ja (IDL, oder ggf. Metadaten aus dem Interface Repository).	nein (Vereinbarung oder zur Laufzeit Objekttyp ermitteln)	nein (Vereinbarung oder zur Laufzeit Objekttyp ermitteln).	ja (WSDL).
Dynamisches Verhalten?	ja.	nur lokal	nein, mit Ausnahme der Java-Mechanismen.	ja.
Metadaten für das Arbeiten mit unbekannten Objekten oder Diensten?	ja.	nur lokal	nein.	ja.
Echtzeitfähigkeit				
bei synchroner Kommunikation	ja	ja	ja	bedingt
bei asynchroner Kommunikation	bedingt		ja	bedingt

Protokolle	IIOP	ORPC (Object Remote Procedure Call)	HTTP(S), IIOP	HTTP(S), ftp, sftp, SMTP, ggf. IIOP
Versionierung der Schnittstelle/Objekte	ja	ja	nein	hängt von der verwendeten Technologie ab
Komponententechnologie	nein (erst ab CORBA 3.0)	ja	bedingt (reduziert auf EJBs, JMS u. a.)	bedingt
Zustandsbehaftete Objekte	nur mit Transaktionen oder CORBA 3	ja	ja	hängt von der verwendeten Technologie ab.
automatisierte Persistenz	nein (erst ab CORBA 3.0)	ja	ja	hängt von der verwendeten Technologie ab
automatisierte Transaktionsverarbeitung	nein (erst ab CORBA 3.0 oder BEA Tuxedo)	ja	ja	hängt von der verwendeten Technologie ab
automatisierte Autorisierung	nein	nein	ja	hängt von der verwendeten Technologie ab
Einsatzgebiet	verteilte Anwendungen/ Systeme	verteilte Anwendungen/ Systeme	verteilte Anwendungen	B2B, EAI
Geeignet für enge Koppelung	ja	ja	ja	nein
Geeignet für lose Koppelung	ja	ja	bedingt	ja
Sprachunabhängigkeit	ja	ja	nein	ja
Plattformunabhängigkeit	ja	nein	ja	ja
Betriebssystemunabhängigkeit	ja	bedingt	ja	ja
Einfachheit der Entwicklung	mittel	gering	einfach	nur mit geeigneten Werkzeugen
Einfachheit der Administration	produktabhängig	hoch	produktabhängig	produktabhängig
Einfachheit der Erlernbarkeit	mittel bis hoch	hoch	mittel	mittel bis hoch
Umfang der Spezifikation	mittel	hoch	mittel	mittel
Transparenzeigenschaften	gegeben	gegeben	gegeben	gegeben
Dienste:				
Namensdienst	ja	Registry	ja	nein
Sicherheitsdienst	ja	nur MTS	ja	nein
Transaktionsdienst	ja	nur MTS	ja	nein
Trader	ja (produktabhängig)	nein	nein	UDDI
Interface Repository	ja	nein, nur lokal	nein	UDDI
Internetfähigkeit	nein	nein	ja	ja
Laufzeitumgebung für den Klienten erforderlich?	ja	ja	nein	nein
Standard	ja	nein	ja	noch nicht abgeschlossen
Stabilität des Standards	hoch	n/a	gering bis mittel	n/a
Skalierbarkeit	hoch	gering	hoch	hoch
Clustering	produktabhängig	nein	produktabhängig	i. d. R. nein
Integration von Legacy-Applikationen	i. d. R. hoch	ja	i. d. R. hoch (JCA)	hoch

Tabelle 12: Gegenüberstellung CORBA, DCOM, J2EE und Web Services

19.2 Bewertung

Im Zusammenhang mit CORBA wurde die enge Koppelung verteilter Anwendungen mit statischem Stub und Skeleton untersucht. Ein zentrales Ergebnis war die nachweisbare Erkenntnis, dass die Kommunikation zwischen dem Sender einer Nachricht und dessen Empfänger hochgradig optimiert werden muss. Dies wurde belegt, in dem der Aufwand, der für das Marshalling und Unmarshalling der Daten benötigt wird, im Detail dargelegt wurde.

Ferner wurden Messungen vorgenommen, die eindeutig belegen, dass eine komplette Datenstruktur bzgl. der Übertragungszeit günstiger ist, als das Senden mehrerer kleinerer Pakete. Da heraus resultierten Richtlinien, um eine verteilte Anwendung echtzeitfähig zu gestalten.

Von besonderem Interesse waren zusätzlich die bisher noch nicht hinreichend genug untersuchten dynamischen CORBA-Komponenten DII, Interface Repository, sowie die generischen Datentypen Any und DynAny. Ziel war es, eine genaue Analyse über
a. die Arbeitsweise dieser Komponenten zu erstellen.
b. den Aufwand, der notwendig ist, um generische Datentypen zu erzeugen.
c. die Verarbeitungsgeschwindigkeit zu erstellen.
d. die Verwendbarkeit dieser Komponenten in einer engen und einer losen Koppelung zu erstellen.

Die erzielten Resultate beweisen eindeutig, dass die Verbindung von DII mit dem Interface Repository im Zusammenhang mit unbekannten Objekten nicht echtzeitfähig ist und somit nur in einer losen Koppelung zu Einsatz kommen sollte. Die Verwendung von DII mit statischen Datentypen ist weniger kritisch bzgl. seiner Verarbeitungsgeschwindigkeit und eignet sich für die asynchrone Kommunikation, die im Zusammenhang mit langläufigen Anfragen wünschenswert und notwendig ist. Als nachteilig wurde lediglich der Umstand identifiziert, dass eine Dienstqualität, wie z. B. die garantierte Nachrichtenzustellung, erst ab CORBA 3 gewährleistet ist.

Der Mehraufwand, der durch das Erstellen von Request-Pseudoobjekten entsteht, kann bei richtiger Anwendung minimiert werden und findet primär auf der Seite des Klienten statt. Für die Serverseite ist es völlig transparent, ob ein Objekt statisch oder dynamisch angesprochen wird. Der zusätzliche Mehraufwand, der in dem serverseitigen Skeleton zu leisten ist, stellt eine weitestgehend vernachlässigbare Größe dar.

Damit verbunden waren detaillierte Analysen über den Aufbau dieser Komponenten und Vorgehensweisen, die gezeigt haben, wie diese Bausteine genutzt werden können. Illustriert wurde dies jeweils durch Beispiele.

Die erzielten Ergebnisse bzgl. der statischen Methodenaufrufe lassen sich unmittelbar auf die Enterprise JavaBeans übertragen, da hier das gleiche Kommunikationsprotokoll und das gleiche Kommunikationsparadigma verwendet wird. Im Zusammenhang mit dynamischen Komponenten hat J2EE nichts Äquivalentes zu DII und Interface Repository vorzuweisen.

Bzgl. des statischen Verhaltens sind die Ergebnisse ebenfalls auf COM/DCOM übertragbar. In beiden Fällen werden Binärprotokolle verwendet, bei denen die Kommunikation optimiert werden muss. Ebenfalls in beiden Fällen ist bei einem Einsatz im Internet die Firewall-Problematik zu berücksichtigen.

Bzgl. der asynchronen Kommunikation ist COM/DCOM mit dem MSQS gleichwertig zu der J2EE-Subtechnologie JMS. CORBA bietet in der Spezifikation 2.x lediglich das DII für die asynchrone Kommunikation an. Die von CORBA beschriebene Optimierung durch das

Versenden mehrerer Nachrichten in einer einzelnen Anfrage kann mit J2EE oder MSQS ebenfalls erzielt werden. Der Sender von Nachrichten eröffnet vor dem Senden eine Transaktion und stellt die Nachrichten mit der zur Verfügungstehenden Primitive *send* ein. Sobald der Sender der Nachrichten die Beendigung der Transaktion mit *commit* bestätigt, werden alle Nachrichten in genau einer Socketverbindung zum Server übertragen und in jeweilige die Queue bzw. das jeweilige Topic eingestellt.

Bei der asynchronen Kommunikation erlaubt CORBA i. d. R. keine Priorisierung (eine Ausnahme ist lediglich BEA Tuxedo 8.1. Hier besteht die Möglichkeit, dass ein CORBA-Objekt eine Anfrage entgegennimmt und diese Anfrage versehen mit einer Priorität in eine Tuxedo-Queue schreibt). Hier ist der Vorteil der asynchronen Kommunikation eindeutig auf Seiten von JMS bzw. MSQS zu sehen. Beide Queing-Systeme erlauben dem Sender eine Priorität festzulegen und somit die Bedeutung von Nachrichten hervorzuheben. In einem Polizeisystem kann diese Eigenschaft durchaus von Interesse sein. Werden bspw. für die Person X Punkte in das Flensburger Kraftfahrtbundesamt System eingestellt, so kann dies durchaus mit geringer Priorität geschehen, damit andere, wichtigere Anfragen, schneller bearbeitet werden.

CORBA verfügt erst ab der Spezifikation 3.x über verbesserte Möglichkeiten der asynchronen Kommunikation. Ob es jedoch Implementierungen von CORBA 3.x geben wird, ist ggw. nicht abzusehen. Von Seiten des Marktführers Iona Technologies scheint der Fokus ggw. mehr auf J2EE-Applikationsserver und auf den Komplex des EAI's gerichtet zu sein. Dito für den Mitbewerber BEA Systems.

Der Charme von CORBA besteht in seiner Spezifikation, die vollständig und in sich konsistent ist. Mehrere hundert Softwarehersteller haben an diesem Standard mitgearbeitet und ihn gemeinsam beschlossen. Auch wenn CORBA ggw. nicht mehr die führende Middleware-Technologie ist und weitestgehend durch J2EE ersetzt wird, so sind die zugrundeliegenden Ideen bahnbrechend und spiegeln sich in der Spezifikation der Web Services wieder.

Insbesondere der dynamische Anteil von CORBA, nämlich das Interface Repository, das Dynamic Invocation Interface und Dienste wie z. B. ein Trader sind die konzeptionellen Säulen der Web Services. In der Tat ist das UDDI eine Kombination aus DII, IFR und Trader. Aufgrund seiner zentralen Adresse kann das UDDI im Internet sofort gefunden werden und die Beschreibung von Diensten zur Verfügung stellen. Die Interpretation der WSDL ist durchaus vergleichbar mit der Interpretation von Metadaten aus einem IFR und dem Erstellen von Request-Pseudoobjekten für das DII.

Web Services erlauben sowohl die synchrone Kommunikation, als auch die asynchrone. Durch ihren generischen Ansatz sind sie jedoch in erster Linie für eine lose Koppelung, wie z. B. B2B, geeignet. Auch wenn ein Klient sich einen Stub merkt und ihn nicht zu Laufzeit neu erstellen bzw. beziehen muss, ist zu berücksichtigen, dass die Daten in ein XML-Dokument eingestellt bzw. gelesen werden müssen. Der damit verbundene Aufwand ist durchaus vergleichbar mit dem Aufbau von Request-Pseudoobjekten, dem Erzeugen von Any-Kontainer und das Einstellen/Extrahieren von Daten in/aus diesem Kontainer.

Für die Entwicklung von in sich abgeschlossenen verteilten Anwendungen sind CORBA, COM/DCOM bzw. .NET und J2EE grundsätzlich geeignete Technologien sowohl für eine lose, als auch für eine enge Koppelung. Ein unzureichend gelöstes Problem dieser Technologien ist in der Interoperabilität zwischen CORBA und COM/DCOM bzw. J2EE und COM/DCOM zu sehen. Die von der OMG spezifizierte COM-CORBA-Bridge ist ebenso wie das jCom von J2EE äußerst unhandlich und nicht bei allen Produkten vorhanden. An dieser Stelle können die Web Services durchaus Abhilfe schaffen. Dank der .NET-Initiative von Microsoft und den Web Service Aktivitäten der J2EE-Gemeinde kann das SOAP-Protokoll leicht diese Lücke schließen. Für CORBA und andere, proprietäre Technologien wie z. B. BEA Tuxedo sind ebenfalls Implementierungen für Web Services möglich. Bspw. weist BEA

Tuxedo ab Version 7.1 eine XML-Unterstützung für C (und COBOL) auf und ist bereits damit bedingt Web Service fähig.

Index

Literaturverzeichnis

[Abernethy et al., 1999]
Abernethy, R., Morin, R., Chahin, J.
COM/DCOM Unleashed
SAMS, 1999

[Barke, 1996]
Barke, Dr. Michael
Studie über die beiden Standards für objektorientierte Entwicklungen - CORBA und COM/OLE
http://www.bkm-dv.de/Technische_Umsetzung/317.htm

[Brecht, 1991]
Brecht, W.
Verteilte Systeme unter UNIX
Vieweg Verlag, 1991

[Chappell et al., 2002]
Chappell, D. A., Jewell, T.
Java Web Services
O'Reilly 2002

[Cobb, 1998]
Cobb, E. E.
Issues when making object middleware scalable
Middleware Spectra, Mai 1998

[Coulouris et al., 1994]
Coulouris, G. F., Kindberg, T. , Dollimore, J.
Distributed Systems: Concepts and Design
Addison-Wesley 1993, 2nd Edition

[Dannegger et al., 1991]
Dannegger, C., Geugelin-Dannegger, P.
Parallele Prozesse unter UNIX: Simulation und Anwendung bekannter Synchronisationsmethoden in C unter UNIX
Hanser Verlag 1991

[Edwards, 1998]
Edwards, J.
Let's get serious about distributed objects
Distributed Computing, Februar 1998

[Emsenhuber, 2002]
Emsenhuber, C.
COM+ in der Prozeßautomatisierung
Diplomarbeit, Universität Linz, 2002

[Geihs,1993]
Geihs, K.
Infrastrukturen für heterogene verteilte Systeme
Informatik Spektrum, Bd. 16, Nr. 1, 1993

[Geihs,1995]
Geihs, K.
Client/Server-Systeme: Grundlagen und Architekturen
Thomson Publishing, 1995

[Guglielmo et al., 1998]
Guglielmo, E., Nainu, M.
Activation management of buisiness objects
Distributed Computing, April 1998

[Hendricks et al., 2001]

Hendricks, M., Galbraith, B., Irani, R., Milbery, J., Modi, T., Tost, A., Toussaint, A., Jeelani Basha, S., Cable, S.
Professional Java Web Services
WROX, 2001

[Hoque, 1998]
Hoque, R.
CORBA 3
IDG Books, 1998

[IBM, 2002]
http://www4.ibm.com/software/solutions/Webservices/pdf/WSCA.pdf

[Langendörfer et al., 1994]
Langendörfer, H., Schnor, B.
Verteilte Systeme
Hanser Verlag 1994

[Liberty, 2001]
Liberty, J.
Programming C#
O'Reilly, 2001

[Microsoft, 2002]
http://www.microsoft.com/com/wpaper/compsvcs.asp
Microsoft Component Services, A Technology Overview
Microsoft Corporation, White Paper 2002

[Microsoft, 2002a]
http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contendid=28000442
Microsoft Corporation, 2002

[Microsoft, 2002b]
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnWebsrv/html/Websvcs_platform.asp
Microsoft Corporation, 2002

[Monson-Haefel, 2000]
Monson-Haefel, R.
Java Message Service
O'Reilly 2000

[Monson-Haefel, 2002]
Monson-Haefel, R.
Enterprise Java Beans 3rd Edition
O'Reilly 2002

[Monson-Haefel, 2003]
Monson-Haefel, R.
J2EE Web Services
Addison-Wesley 2003

[Newcomer, 2002]
Newcomer, E.
Understanding Web Services
Addison Wesley 2002

[Nickel et al., 1999]
Nickel, W., Walpert, K.
Erfahrungsbericht: Der BEA ORB M3
GI-Frühjahrstreffen bei IBM, Heidelberg

[OMG, 1995a]
Object Management Group
The Common Object Request Broker: Architecture and Specification
OMG

[OMG, 1995b]
Object Management Group
Object Services Architecture

OMG
[Orfali et al., 1996]
Orfali, R., Harkey, D., Edwards, J.
The Essentiel Distributed Objects Survival Guide
John Wiley & Sons, Inc., 1996
[Redlich, 1996]
Redlich, J. P.
CORBA 2.0
Addison Wesley 1996
[Ruh et al., 1999]
Ruh, W., Herron, Th., Klinker, P.
IIOP Complete: Understand CORBA and Middleware Interoperability
Addison-Wesley, 1999
[Schmidt et al., 2002a]
Schmidt, D., Vinoski, S.
http://www.cuj.com/documents/s=7981/cujcexp2007vinoski
Object Interconnections: Dynamic CORBA, Part 1: The Dynamic Invocation Interface
2002
[Schmidt et al., 2002b]
Schmidt, D., Vinoski, S.
http://www.cuj.com/documents/s=7981/cujcexp2007vinoski
Object Interconnections: Dynamic CORBA, Part 2 — Dynamic Any
2002
[Schmidt et al., 2002c]
Schmidt, D., Vinoski, S.
http://www.cuj.com/documents/s=7981/cujcexp2007vinoski
Object Interconnections: Dynamic CORBA, Part 3: The Dynamisc Skeleton Interface
2002
[Schmidt et al., 2003]
Schmidt, D., Vinoski, S.
http://www.cuj.com/documents/s=7981/cujcexp2007vinoski
Object Interconnections: Dynamic CORBA, Part 4: The Interface Repository
2003
[SEI]
http://www.sei.cmu.edu/str/descriptions/com_body.html
[Siegel, 2000]
Siegel, J.
CORBA 3: Fundamentals and Programming. 2nd Edition
John Wiley & Sons, Inc., 2000
[Slama et al.]
Slama, D., Garbis, J., Russel, P.
Enterprise CORBA
Prentice Hall, 1999
[Sun, 2002]
http://www.sun.com/software/sunone/faq.html#42
Sun Microsystems, 2002
[Suresh, 1999]
Suresh Raj, Gopalan
http://members.tripod.com/gsraj/misc/ejbmts/ejbmtscomp.html
[Suresh, 1999a]
Suresh Raj, Gopalan
http://my.execpc.com/~gopalan/misc/compare.html
[Stevens, 1990]
Stevens, R. W.
UNIX Network Programming
Prentice-Hall 1990

[Stewart, 1998]
Stewart, R. L.
Distributed Services vs. Distributed Objects
Distributed Computing, Juli 1998
[Tanenbaum 1990]
Tanenbaum, A. S.
Computer Netzwerke
Wolfrahm's Fachverlag 1990
[Tanenbaum, 1995]
Tanenbaum, A. S.
Verteilte Betriebssysteme
Prentice Hall, 1995
[Troelsen, 2001]
Troelsen, A.
C# and the .NET Platform
.NET Developer Series, 2001
[Vinoski et al., 1999]
Vinoski, S., Mitchi, H.
Advanced CORBA Programming with C++
Addison Wesley, 1999
[Walpert, 1997]
Walpert, K.
Modell einer allgemeingültigen, frei konfigurierbaren Verteilungsplattform
4 ITG/GI Fachtagung Arbeitsplatz-Rechensysteme (APS'97) vom
21-22.05.1997, Universität Koblenz-Landau, S. 206-213
[Westphal, 2002]
Westphal, R.]
.NET kompakt
Spektrum, Akad. Verlag, 2002