**FACHHOCHSCHULE KÖLN**
**UNIVERSITY OF APPLIED SCIENCES COLOGNE**
**ABTEILUNG GUMMERSBACH**
**FACHBEREICH INFORMATIK**

Diplomarbeit

zur Erlangung

des Diplomgrades

Diplom-Informatiker (FH)

in der Fachrichtung Informatik

# – Einsatzmöglichkeiten des „Business Components for Java Frameworks“ von Oracle im Umfeld von Enterprise JavaBeans- und CORBA-Architekturen –

vorgelegt am

28. August 2001

von

**Alexander Orbach**

Ommerbornstr. 35 – 51789 Lindlar

Matr.-Nr.: 11 017 806 1 3

1. Prüfer: Prof. Dr. rer. nat. Heide Faeskorn - Woyke
2. Prüfer: Prof. Dr. rer. nat. Holger Günther

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungs- und Symbolverzeichnis

| | |
|---|---|
| **ACID** | *Atomic, Consistent, Isolated and Durable* |
| **API** | *Application Programming Interface* |
| **BOA** | *Basic Object Adapter* |
| **CORBA** | *Common Object Request Broker Architecture* |
| **DII** | *Dynamic Invocation Interface* |
| **DSI** | *Dynamic Skeleton Interface* |
| **EJB** | *Enterprise JavaBeans* |
| **GIOP** | *General Inter-ORB Protocol* |
| **IDL** | *Interface Definition Language* |
| **IIOP** | *Internet Inter-ORB Protocol* |
| **IOP** | *Inter-ORB-Protocol* |
| **IOR** | *Interoperable Object Reference* |
| **IP** | *Internet Protocol* |
| **J2EE** | *Java 2 Platform Enterprise Edition* |
| **JAR** | *Java Archive* |
| **JDBC** | *Java Database Connectivity* |
| **JDK** | *Java Development Kit* |
| **JMS** | *Java Messaging Service* |
| **JNDI** | *Java Naming and Directory Interface* |
| **JVM** | *Java Virtual Machine* |
| **OA** | *Object Adapter* |
| **OMA** | *Object Management Architecture* |
| **OMG** | *Object Management Group* |
| **ORB** | *Object Request Broker* |
| **POA** | *Portable Object Adapter* |
| **RPC** | *Remote Procedure Call* |
| **TCP** | *Transmission Control Protocol* |
| **TCP/IP** | *Transmission Control Protocol / Internet Protocol* |
| **URL** | *Uniform Resource Locator* |
| **XML** | *Extensible Markup Language* |

# 1 Einführung in das Thema der Diplomarbeit

Die Grundlagen der objektorientierten Programmierung wurden bereits in den sechziger Jahren entwickelt. Aufbauend auf diesem Grundstein hat sich diese Technologie besonders in den letzten Jahren durch ihren Einsatz in zahlreichen Projekten bewährt und durchgesetzt. Projekte, die vielfach mit prozeduralen Programmiersprachen gar nicht realisierbar gewesen wären.

Die anfänglich hohen Erwartungen, welche an diese zur damaligen Zeit als revolutionär geltende Technik im punkto Wiederverwendbarkeit der entwickelten Klassen gestellt wurden, konnten jedoch bis heute nicht vollständig erfüllt werden.

Lange Zeit galt die Vererbung als das Mittel der Wahl, mit dem Klassen im Sinne der Objektorientierung wiederverwendet werden können. Aufgrund von praktischen Einsätzen zeigt sich heute allerdings, dass sich diese Behauptung mit zunehmender Komplexität der zu entwickelnden Software nicht bestätigen konnte.

Man stelle sich eine Klassenhierarchie vor, welche über eine Superklasse und eine große Vielzahl von Subklassen verfügt. Jegliche Änderung der Superklasse führt dazu, dass auch alle Subklassen automatisch ihr Verhalten ändern. Somit entsteht eine fast nicht vorhersehbare Kaskade von Änderungen. Aus Sicht des Entwicklers bedeutet dies ein erneutes Testen und eventuelles Nacharbeiten der bestehenden Klassen, was wiederum den Nutzen der Wiederverwendbarkeit zunichte macht.

Es mussten also alternative Wege gefunden werden, um eine möglichst hohe Wahrscheinlichkeit zur Wiederverwendung von Softwarebausteinen zu erreichen. Genau an dieser Stelle setzt die Idee der Komponentenmodelle an. Dabei handelt es sich um Architekturen, welche dem Entwickler eine Infrastruktur bieten, in der er Komponenten erstellen kann, die möglichst unabhängig voneinander sind, was zur Zeit die größtmöglichen Aussichten auf eine Wiederverwendung bietet.

Des weiteren werden durch solche Modelle Umgebungen geschaffen, die bereits über Standarddienste (z.B. Persistenzmechanismen, Transaktionshandling, Sicherheitsdienst, etc.) verfügen, die der Entwickler nicht immer wieder neu erstellen muss, sondern sie einfach benutzen kann.

Ein weiterer Aspekt, der maßgeblich zur Entstehung der Komponentenmodelle beigetragen hat, ist der Trend hin zu verteilten objektorientierten Systemen. Solche

mehrschichtigen Anwendungen schaffen eine klare Trennung zwischen der Datenhaltung, der Geschäftslogik und der Darstellung der Informationen. Die Komponentenmodelle sind in diesem Zusammenhang ein Mittel zur Implementierung der serverseitigen Geschäftslogik.

Mittlerweile haben sich verschiedene de facto Standards für solche Komponentenmodelle innerhalb der Gemeinde von Softwareentwicklern etabliert.

Im Rahmen dieser Arbeit sollen zwei dieser Komponentenmodelle etwas detaillierter vorgestellt werden. Dabei handelt es sich um die Enterprise JavaBeans (EJB) Spezifikation von SUN sowie die Spezifikation der Common Object Request Broker Architecture (CORBA) der OMG.

Im Zuge der immer größer werdenden Akzeptanz dieser beiden Modelle sind parallel dazu Werkzeuge entstanden, die den Programmierer bei der Erstellung von Komponenten, die diesen Spezifikationen entsprechen, unterstützen sollen.

Auch Oracle als ein Unternehmen, welches seine Wurzeln in der Entwicklung leistungsfähiger Datenbanksysteme hat, hat diesen Trend erkannt und bietet selber ein solches Werkzeug an.

Dieses Werkzeug trägt den Namen „Business Components for Java“ (BC4J) und wird in Form eines auf Java basierenden Frameworks ausgeliefert. Es wurde vollständig in die Java Entwicklungsumgebung von Oracle namens JDeveloper integriert, was ein effizientes Erstellen und Testen der einzelnen Komponenten ermöglicht.

Durch den Einsatz dieses Frameworks können unabhängige Komponenten zunächst lokal erstellt und getestet werden, auf deren Basis dann EJB- bzw. CORBA-Objekte erzeugt werden können.

Die vorliegende Arbeit soll zunächst, nach einer Einführung in die theoretischen Grundlagen Verteilter Systeme die beiden genannten Komponentenmodelle vorstellen. Letztendlich ist es dann das Ziel, die Möglichkeiten zu diskutieren, die dem Entwickler durch den Einsatz von BC4J bei der Erstellung von Komponenten für Verteilte Architekturen zur Verfügung stehen.

# 2 Grundlagen

Diese Kapitel soll verdeutlichen, was unter den Begriffen eines „Verteilten Systems“ sowie der „Verteilten Objekte“ grundsätzlich zu verstehen ist. Auf dieser Basis bauen später die Ausführungen zu den Komponentenmodellen auf. Darüber hinaus sollen Probleme aufgezeigt werden, welche sowohl zum Zeitpunkt der Entwicklung als auch zur Laufzeit einer solcher Anwendungen auftreten können und wie man diese Probleme mit heutigen Mitteln lösen kann.

## 2.1 Verteilte Systeme

Verteilte Systeme haben zunächst einmal das Ziel, die einzelnen Bestandteile eines Geschäftssystems auf verschiedenen Computern ablaufen zu lassen. Diese Rechner sind jeweils spezialisiert auf eine bestimmte Aufgabe und können physikalisch voneinander getrennt sein. Ein Webserver, dessen einzige Aufgabe es ist, die verfügbaren Informationen für eine Internetanwendung aufzubereiten oder ein Datenbankserver, dessen Domäne wohl eher die reine Datenhaltung ist, sind nur zwei Beispiele für solche spezialisierten und zumeist sehr leistungsfähigen Computer, die in Verteilten Systemen zum Einsatz kommen.
Der immense Vorteil solcher Systeme liegt in der Tatsache, dass auf die Geschäftslogik und die dazugehörigen Daten von beliebigen Orten aus zugegriffen werden kann.

## 2.2 Verteilte Objekte

Im Zuge der wachsenden Akzeptanz objektorientierter Programmiersprachen wie Java oder C++ wird im Zusammenhang mit Verteilten Systemen immer stärker auch der Begriff der „Verteilten Objekte“ geprägt. Durch diese Technologie wird es möglich, Objekte auf einem entfernten Rechner laufen zu lassen, die dann von Client-Applikationen auf anderen Computern benutzt werden können.
Die Entwicklung Verteilter Objekte hat ihren Ursprung in der relativ alten Idee, Software in einer Architektur ablaufen zu lassen, welche aus mehreren unabhängigen Schichten besteht. Zu damaliger Zeit bestand ein Verteiltes System aus unintelligenten Terminalbildschirmen, welche die Schnittstelle zum Benutzer

bildeten, einer Schicht aus prozeduralen COBOL Applikationen, die den Programmablauf steuerten und einer Datenbankschicht für die Datenhaltung.
Der Einsatz der heutzutage modern gewordenen Verteilten Objekte innerhalb von Verteilten Systemen hat dazu geführt, dass die Terminals durch eine umfangreiche grafische Benutzeroberfläche und die COBOL Prozeduren durch flexible Geschäftsobjekte ersetzt wurden. In diesem Zusammenhang bleibt abzuwarten, ob und wenn ja wie lange es dauern wird, bis dass die heute sehr populären relationalen Datenhaltungssysteme über die Brücke der objektrelationalen Systeme letztendlich von den objektorientierten Datenbanken verdrängt werden.
Komplexe Architekturen, in denen es viele Schichten geben kann, werden oft verwendet. Die Objekte, welche auf verschiedenen Servern untergebracht sind, müssen interagieren, um eine bestimmte Aufgabe zu erfüllen. Der weitere Verlauf dieser Arbeit soll nun Aufschluss darüber geben, welche verschiedenen Typen von Architekturen in der praktischen Anwendung unterschieden werden.

## 2.3 Das Client/Server- Modell

Das Client/Server-Modell ist eine zweischichtige Architektur und bildet die einfachste Form eines Verteilten Systems. In einer solchen Umgebung bilden dabei die Server diejenigen Komponenten, die ein bestimmtes Mass an Funktionalitäten realisieren und diese als Dienste über das Netzwerk den Clients zur Verfügung stellen. Diese Clients beherrschen ihrerseits einen Mechanismus, mit dem sie diese serverseitigen Objekte nutzen können.
Die Kommunikation basiert dabei auf einem einfachen verbindungslosen Anfrage/Antwort Protokoll. Um einen Dienst zu nutzen, sendet der Client eine Anfragenachricht an den entsprechenden Server. Dieser sendet nach der Bearbeitung der Anfrage das Ergebnis an den Client zurück. Diese von den Kernen[1] durchgeführte und über das Netzwerk vollzogene Kommunikation ist graphisch in der Abbildung 2-1 veranschaulicht.
Das große Problem, welches sich jedoch hinter einem solch einfachen Modell verbirgt, resultiert aus der Notwendigkeit, dass sich Client und Server über

[1] Hiermit sind die Kerne der Betriebssysteme gemeint, die aufgrund eines bestimmten Protokolls die Daten über das Netzwerk versenden bzw. empfangen.

Nachrichten miteinander verständigen müssen. Das bedeutet, dass der Client gezwungen ist, die genaue Adresse des Servers zu kennen, was aus Sicht des Entwicklers unweigerlich zu einem Verlust der Ortstransparenz der Serverkomponenten führt.

Dieses Problem stellte in der Anfangsphase der Entwicklung Verteilter Systeme einen gravierenden Nachteil dar. Zu dieser Zeit wurde und auch heute noch wird das Ziel verfolgt, dass der Aufruf von Methoden entfernter Objekte genauso einfach zu realisieren sein sollte, wie der Aufruf von Methoden lokaler Objekte.

Erst 1984 wurde eine Lösung dieses Problems durch Birell und Nelson vorgestellt. Der sogenannte „Remote Procedure Call“ (RPC) soll im folgenden Abschnitt detaillierter vorgestellt werden, da dieser Ansatz als Grundstein für die weitere Entwicklung der Verteilten Verarbeitung angesehen werden kann.

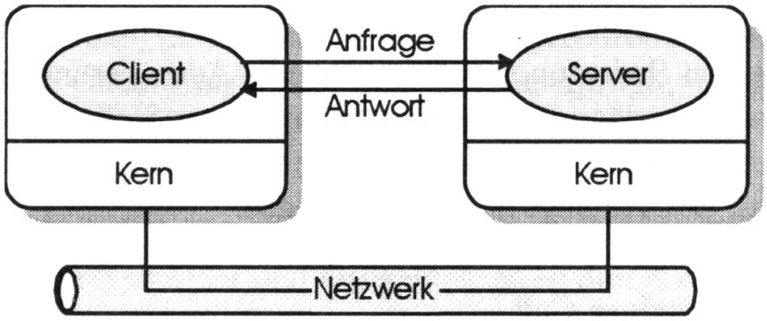


**Abbildung 2-1 Das Client/Server-Modell**

**Quelle: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 11.**

## 2.4 Remote Procedure Call

Der Remote Procedure Call beschreibt ein Verfahren, mit dem es möglich ist, Methoden entfernter Objekte aufzurufen, als wären sie lokal vorhanden. Dies kann am einfachsten verständlich gemacht werden, indem man sich zunächst die Vorgänge bei einem einfachen Aufruf einer Methode an einem lokalen Objekt vor Augen hält (→ vgl. Abbildung 2-2).

Ein solcher Methodenaufruf zeichnet sich durch den eigentlichen Namen der Methode sowie einer Liste aller notwendigen Parameter zur Erfüllung dieser Operation aus.

Bei einem Aufruf werden alle wichtigen Informationen für eine korrekte Rückkehr, d.h. die Parameter, die Rücksprungadresse sowie die lokalen Variablen auf einem Stack[2] gesichert. Anschließend wird der Prozess, welcher die Methode des anderen Objektes aufrufen möchte, vorerst suspendiert. Dies ist der Moment, in welchem dem anderen lokalen Objekt mit der gewünschten Methode die Kontrolle übertragen wird.

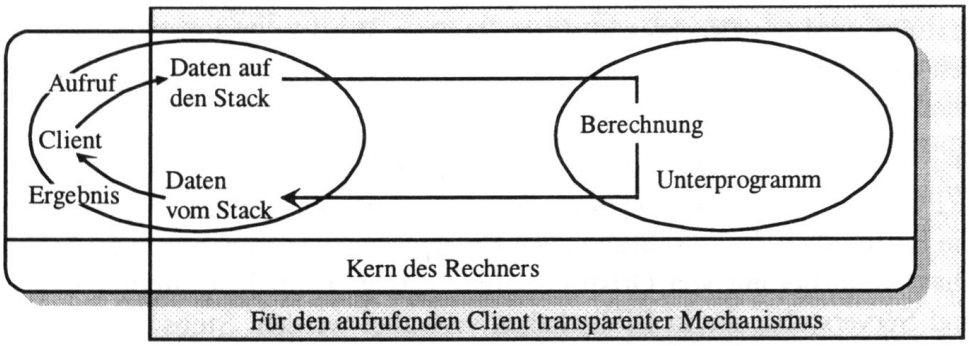


**Abbildung 2-2 Der lokale Methodenaufruf**

**Quelle: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 14.**

Dieses Objekt benutzt die übergebenen Parameter, um damit das Ergebnis zu berechnen, welches in einem Register abgelegt wird. Nachdem der aufrufende Prozess die Kontrolle wiedererlangt hat, steht diesem das Ergebnis zur Verfügung und die Parameter können an dieser Stelle vom Stack entfernt werden.

Aus Entwicklersicht soll dieser Mechanismus natürlich bei entfernten Objekten möglichst auf genau dieselbe Art und Weise funktionieren, d.h. der Ort, an dem sich das gewünschte Objekt befindet, soll verborgen bleiben.

Dies wird im Rahmen des Remote Procedure Calls durch das Einführen eines sogenannten Client- und Server-Stubs[3] gewährleistet (→ vgl. Abbildung 2-3).

---

[2] Bei einem Stack (dt. Stapel) handelt es sich um eine Datenstruktur, in der beliebig viele Daten sozusagen auf einem Stapel abgelegt werden können. Die Besonderheit eines Stacks liegt darin, dass der zuletzt gespeicherte Datensatz als erster den Stack beim Lesen wieder verläßt.

[3] In der Literatur wird der „Server-Stub“ auch oft als „Skeleton“ bezeichnet

Diese beiden Komponenten regeln den Nachrichtenaustausch zwischen Client und Server und sollen den Entwickler von dieser Aufgabe befreien.
Im Grunde funktioniert der entfernte Methodenaufruf zunächst einmal genauso wie der lokale, d.h. die Daten werden auf den Stack gesichert. Der Unterschied besteht nun darin, dass der Client-Stub diese Informationen ausliest, um daraus eine Nachricht zu erzeugen, welche er daraufhin zum Server senden kann. Dieser Vorgang wird auch als Marshalling bezeichnet. Während der Client-Stub auf eine Antwort vom Server wartet, bleibt er im suspendierten Zustand.
Auf der Seite des Servers kann die Nachricht des Client-Stubs entschlüsselt und die Parameter können entnommen werden, was als Unmarshalling bezeichnet wird. Daraus folgt nun der korrekte Aufruf der gewünschten Methode am entfernten Objekt, welches natürlich aus Sicht des Servers wie ein lokales Objekt behandelt wird. Das Ergebnis kann berechnet und über den gleichen Nachrichtenkanal zum Client-Stub zurückgeschickt werden.

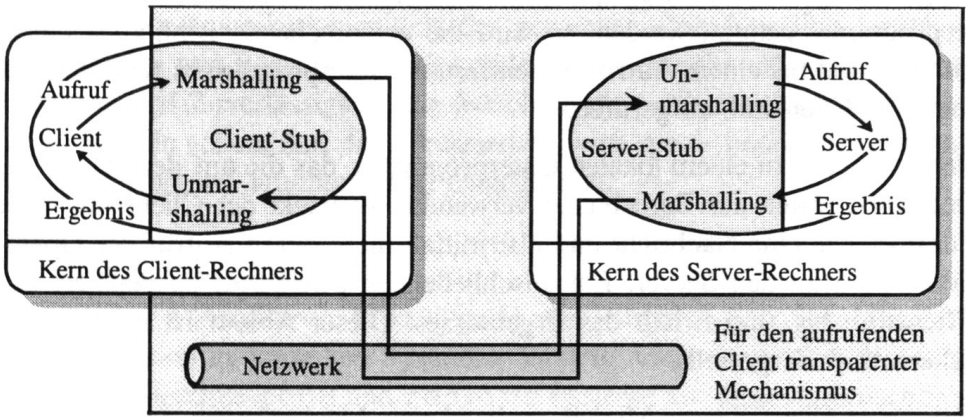


**Abbildung 2-3 Der entfernte Methodenaufruf durch RPC**

**Quelle: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 14.**

Für den Entwickler Verteilter Systeme liegt der Vorteil einer solchen Technik darin, dass sowohl der Client- als auch der Server-Stub durch einen entsprechenden Compiler basierend auf den Informationen aus den jeweiligen Objektimplementierungen vollständig generiert werden können.
Im Falle von CORBA als programmiersprachenneutrales Komponentenmodell (→ vgl. Abschnitt 3.3) wird eine eigene Sprache benutzt, um Schnittstellen

entfernter Objekte zu beschreiben. Diese unabhängige Sprache (→ vgl. Abschnitt 3.3.4) wird dann als Grundlage für die Generierung der Stubs auf unterschiedlichsten Softwareplattformen verwendet.
Letztendlich wird die Ortstransparenz bei RPCs durch die Möglichkeit des dynamischen Bindens ermöglicht.
Unter dem Begriff des dynamischen Bindens wird die Fähigkeit einer Variablen verstanden, sich erst zur Laufzeit mit einer konkreten Objektinstanz verbinden zu können. Dadurch ist ein erneutes Übersetzen der Programme nicht mehr erforderlich, falls z.B. ein Server auf einen anderen Rechner verlagert wird.

## 2.5 Drei- und Mehrschichtige Modelle

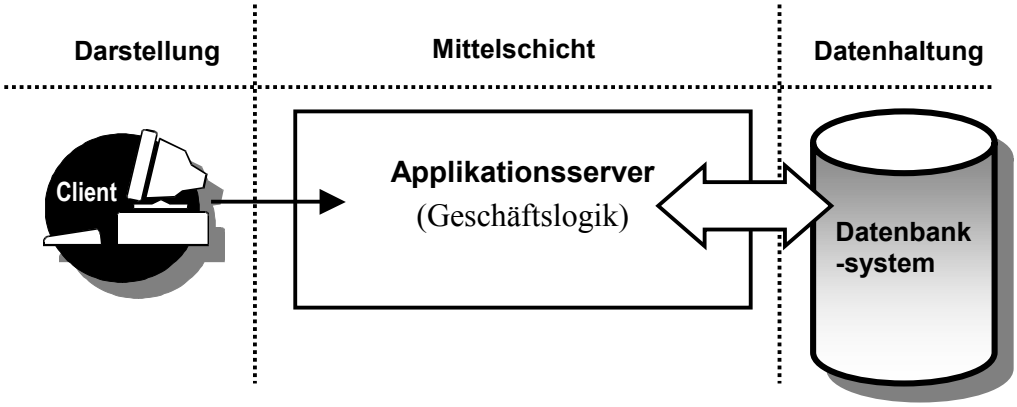


**Abbildung 2-4 Die Drei-Schichten-Architektur**

**Quelle: In Anlehnung an: Monson-Haefel Richard, *Enterprise JavaBeans*, O'Reilly 2001, 2.Auflage. S.6.**

Eine Drei-Schicht-Architektur (→ vgl. Abbildung 2-4) zeichnet sich durch eine separate Schicht zur Darstellung der Informationen aus. Diese Aufgabe wird durch die Clients (Web-Clients, Standalone Anwendungen, etc.) wahrgenommen. In der mittleren Schicht agiert der sogenannte Applikationsserver, welcher die gesamte Geschäftslogik der Anwendung beherbergt. Letztendlich werden die zu speichernden Daten in einem geeigneten Datenbanksystem in der dritten Schicht gesichert. Je nach dem welche Datenbank verwendet wird, ist es auch möglich und sinnvoll, Teile der Geschäftslogik in Form von Prozeduren, Schlüsseln oder referentiellen Integritätsprüfungen direkt in der Datenbank unterzubringen.

Einige der Vorteile, die solche Architekturen im Vergleich zu den Zwei-Schicht-Modellen des vorangegangenen Abschnitts haben, sollen an dieser Stelle diskutiert werden.

Gegenwärtig zeichnet sich wie bereits eingangs dieser Arbeit erwähnt, ein Trend ab, diese drei Schichten immer weiter aufzuspalten, um zusätzliche Schichten in das Gesamtsystem einzufügen und diesen klar definierte Aufgaben zuzuteilen. Man spricht dann von sogenannten Mehrschichtigen Architekturen.

Um den Grund für eine solche Entwicklung deutlich zu machen, muss man ein solches System aus zwei verschiedenen Blickwinkeln betrachten, der logischen und der physikalischen Sicht.

Aus der logischen Sicht heraus leuchtet schnell ein, dass je feingranularer die Schichtenaufteilung des Gesamtsystems ist, desto einfacher wird es für den Entwickler, einzelne Schichten zu modifizieren bzw. komplett auszutauschen.

Betrachtet man ein solches System von der physikalischen Warte aus, so lassen sich Systeme mit mehreren Schichten hervorragend skalieren, da theoretisch in jeder einzelnen Schicht eine separates System zur Lastenverteilung installiert werden kann.

Ein weiterer besonders wesentlicher Vorteil mehrschichtiger Modelle besteht darin, dass die Clients leichtgewichtiger[4] implementiert werden können. Durch die Tatsache, dass ein Großteil der Geschäftslogik auf dem Server in der Mittelschicht verarbeitet wird, werden auf dem Client weniger Resourcen verbraucht.

Darüber hinaus kann die Mittelschicht vom Zugriff auf die Daten abstrahieren, da das bereits durch die Datenablageschicht erledigt wird. Die primäre Aufgabe der Mittelschicht liegt darin, die Konsistenz der Daten aufrecht zu erhalten. Der Entwickler muss sich also nicht mehr um den Datenzugriff kümmern, sondern kann sich idealerweise nur auf die eigentlich zu implementierende Logik konzentrieren.

Durch den Einsatz von mehrschichtigen Architekturen kann eine bessere Skalierbarkeit garantiert werden. Das bedeutet, dass bei großen Mengen von clientseitigen Anfragen bereits in der Mittelschicht eine Lastverteilung stattfinden kann.

[4] Man spricht in diesem Zusammenhang auch oft von Thin-Clients oder Lightweight-Clients

Die Mittelschicht dient in diesem Zusammenhang als zentraler Ort zur Ablage für die eigentlich wichtige Geschäftslogik, auf die alle Clients gemeinsam zugreifen. In der Konsequenz erhält man als Entwickler ein System, welches einfach zu pflegen und zu warten ist. Im Falle einer Änderung bedeutet dies beispielsweise, dass Änderungen der Logik nur einmalig in der Mittelschicht und nicht bei jedem Client durchgeführt werden müssen.

## 2.6 Transaktionskonzepte

Immer dann, wenn mehrere Benutzer konkurrierend auf einen gemeinsamen Datenbestand zugreifen (was typischerweise in Verteilten Systemen der Fall ist), entsteht das Problem, diese parallelen Zugriffe derartig steuern zu müssen, dass die Konsistenz des Datenbestandes immer gewahrt bleibt. Heutzutage stehen allerdings Verfahren zur Verfügung, welche in der Lage sind, diese Probleme zu lösen. In diesem Zusammenhang spricht man oft von sogenannten „Transaktionskonzepten“.
Eine Transaktion zeichnet sich durch eine Menge einzelner Arbeitsschritte aus. Diese Menge von Aktionen kann im Rahmen einer Transaktion dennoch als atomare oder unteilbare Einheit gesehen werden, welche einen konsistenten Datenbestand in einen anderen konsistenten Datenbestand überführt. Schlägt einer der Schritte aus einer Transaktion fehl, so muss dafür Sorge getragen werden, dass alle vorangegangenen Aktionen wieder rückgängig gemacht werden, so als hätte die Transaktion niemals stattgefunden.
Hierbei ist allerdings zu beachten, dass es bei der Ausführung der einzelnen Aktionen durchaus zu vorübergehend inkonsistenten Zwischenzuständen kommen kann. Die einzige Bedingung, die an eine Transaktion gestellt wird ist, dass sich der Datenbestand nach Beendigung des letzten Arbeitsschrittes wieder in einem konsistenten Zustand befinden muss.
Bei parallelem Zugriff mehrerer Clients auf denselben Datenbestand kann es dadurch in vielerlei Hinsicht zu Szenarien kommen, in denen manche Clients inkonsistente Daten lesen. Solche Zustände lassen sich durch den Einsatz moderner Datenbanksysteme vermeiden, welche in der Lage sind, Daten einer laufenden

Transaktion zwischenzuspeichern[5] oder Datensätze exklusiv für einen Client zu reservieren[6].

Typischerweise werden die Eigenschaften von Transaktionen häufig durch die englische Abkürzung *„ACID“* beschrieben.

In diesem Zusammenhang steht *„A“* für Atomic (atomar), d.h. Transaktionen sind unteilbar und werden entweder ganz oder gar nicht ausgeführt.

*„C“* steht für Consistent (konsistent) und meint die Integrität des zugrundeliegenden Datenspeichers. Hier liegt es in der Verantwortung des Entwicklers dafür zu sorgen, dass es in der Datenbank passende Constraints (Primärschlüssel, referentielle Integrität, etc.) gibt und dass die von ihm implementierte Geschäftslogik keine inkonsistenten Daten zur Folge hat.

*„I“* ist die Abkürzung für Isolated (isoliert), d.h. eine Transaktion muss ausgeführt werden dürfen, ohne dass sie dabei von anderen Prozessen oder Transaktionen gestört wird. Somit dürfen die Daten, auf denen eine bestimmte Transaktion aktuell arbeitet, nicht durch eine andere Transaktion oder Anwendung verändert werden, und zwar so lange, bis dass die ursprüngliche Transaktion ihre Arbeit beendet hat.

Letztendlich steht *„D“* für Durable (dauerhaft), was bedeutet, dass im Falle einer erfolgreich durchgeführten Transaktion die Daten dauerhaft in einem physikalischen Datenspeichermedium festgeschrieben werden müssen. Erst wenn alle notwendigen Daten gesichert sind, kann die Transaktion als beendet angesehen werden, womit sichergestellt wird, dass die Änderungen nicht verloren gehen für den Fall eines plötzlichen Systemzusammenbruchs.

Der Einsatz solcher Mechanismen führt allerdings zu einem extrem hohen Verbrauch serverseitiger Ressourcen, was die Entwickler nicht selten zu einem Kompromiss zwischen der Qualität der Konsistenz und der Performance des Systems zwingt.

Beim Thema der technischen Umsetzung von Transaktionen werden in der Regel immer die Begriffe Transaktionssteuerung und Transaktionsregelung voneinander unterschieden. Die Transaktionssteuerung bestimmt in diesem Zusammenhang

---

[5] Dieses Zwischenspeichern von Daten innerhalb einer Transaktion wird auch als „Caching“ bezeichnet

[6] Das Reservieren von Datensätzen für einen Client wird auch als „Locking“ bezeichnet

wann eine Transaktion beginnt und wann sie als beendet oder als gescheitert gilt. Dazu stehen ihr drei einfache Kommandos zur Verfügung. „begin“ startet eine Transaktion, „commit“ bezeichnet eine Transaktion als erfolgreich abgeschlossen und „rollback“ veranlasst einen Abbruch einer Transaktion aufgrund eines aufgetretenen Fehlers.

Die Transaktionsregelung kontrolliert dabei den eigentlichen Ablauf einer Transaktion im technischen Sinne. Sie besteht in aller Regel aus einem Kommunikationsmechanismus für verteilte Transaktionen und einem sogenannten Transaktionsmonitor, welcher die einzelnen Arbeitsschritte überwacht.

# 3 Komponentenmodelle

Bevor in einem weiteren Schritt die wesentlichen industriellen Standards im Bereich der Komponentenmodelle beschrieben werden können, soll der folgende Abschnitt zunächst aufklären, was Komponenten bzw. Komponentenmodelle eigentlich sind und aus welchen Gründen eine solche Tendenz überhaupt entstanden ist.

## 3.1 Einführung in das Thema der Komponentenmodelle

Seit vielen Jahren ist es eines der obersten Ziele der Softwaretechnologie, eine Vorgehensweise für die Erstellung von Programmcode zu finden, der ausreichend flexibel ist, um ihn in verschiedenen Anwendungen wiederverwenden zu können. Dadurch wäre es möglich, unter qualitativen Gesichtspunkten immer bessere Software in immer kürzerer Zeit zu entwickeln.

Genau an dieser Stelle spielen Komponentenmodelle eine Rolle. Es existiert eine ganze Vielzahl unterschiedlichster Definitionen darüber, was eine Komponente ist und was nicht.

Im Grunde genommen sollte eine Komponente aus Sicht des Entwicklers als eine sogenannte Black-Box betrachtet werden können. Das bedeutet, dass er nichts über die Vorgänge innerhalb einer Komponente wissen muss. Eine Komponente sollte in der Lage sein, mit anderen Komponenten kommunizieren zu können und die eigenen Funktionalitäten, die sie selber auszeichnen über wohl definierte Schnittstellen ihrer Außenwelt zur Verfügung zu stellen. Das Ziel eines jeden komponentenbasierten Softwareprozesses sollte es sein, die Komponenten so zu entwickeln, dass sie möglichst unabhängig voneinander sind und leicht mit anderen Komponenten integriert werden können.

Durch den konsequenten Einsatz solcher Komponentenmodelle ergibt sich nicht nur die Möglichkeit der Wiederverwendung eigener Komponenten, sondern auch die Chance zur Integration eigener Komponenten mit Komponenten, welche durch Fremdanbieter entwickelt wurden. Solche Komponenten müssten dann nicht mehr selbst erstellt werden, sondern könnten einfach hinzu gekauft werden, was die Entwicklungszyklen und Testphasen erheblich verkürzen würde.

Die Strategie, die Unternehmen in diesem Bereich verfolgen sollten besteht darin, gewisse Standardkomponenten hinzu zu kaufen und zu integrieren. Die durch eine solche Zeitersparnis frei werdenden Entwicklungskapazitäten können somit in die Komponenten investiert werden, welche am ehesten zu einer Differenzierung des eigenen Produktes von anderen Konkurrenzprodukten führen könnten.

## 3.2 Entstehung der Komponentenmodelle

Zwei grundlegende Strömungen in der Geschichte der Softwaretechnologie haben zur Entwicklung von Komponentenmodellen geführt, die im Folgenden kurz vorgestellt werden sollen.

### 3.2.1 Flexibilität und Verteilung von Software

In den letzten Jahrzehnten wurde die Vorgehensweise bei der Entwicklung von Software stark durch das Bestreben geprägt, Software nicht mehr nur als rein monolithische Systeme zu entwickeln, da solche Systeme in vielen Fällen zu unflexibel sind.

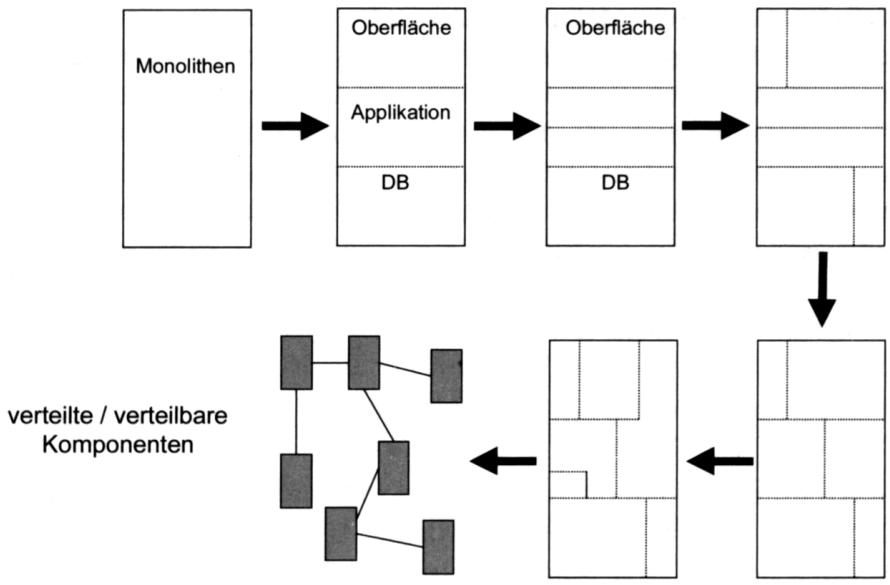


**Abbildung 3-1 Verteilung und Strukturierung von Softwaresystemen**

**Quelle: Gruhn Volker, Thiel Andreas, *Komponentenmodelle*, Addison Wesley, 2000. S. 6.**

Im Zuge der Einführung von Client-/Server-Systemen entstand die Idee der Aufteilung solcher Systeme in mehrere voneinander entkoppelte Schichten. Solche Architekturen bezeichnet man als die bereits beschriebenen Zwei-, Drei-, oder Mehrschicht-Systeme (→ vgl. Abschnitte 2.3 und 2.5).

Dieser Trend der Aufteilung bestehender Schichten zu immer feingranulareren logischen Einheiten setzt sich scheinbar immer weiter fort. Das Ergebnis sind flexible und verteilbare Komponenten (→ vgl. Abbildung 3-1). Das Ziel aus Sicht des Entwicklers besteht nun darin, diese Komponenten analog zu einem Baukasten zusammensetzen zu können, ohne sich dabei Gedanken über Verteilungsaspekte machen zu müssen. Er sollte sich ganz auf die eigentlichen fachlichen Anforderungen der Anwendung konzentrieren können.

### 3.2.2 Anwendungsnahe Bausteine

Rein technische Komponenten wie z.B. Oberflächen, Listen oder Datentypen sind aus Sicht des Entwicklers kleine, häufig wiederverwendbare Bausteine, welche jedoch mit der eigentlichen Anwendung direkt nichts zu tun haben, d.h. sie sind anwendungsfern.

Das Erstellen von anwendungsnahen Komponenten meint in diesem Zusammenhang den Einsatz und die Kombination dieser technischen Bausteine, um dadurch Komponenten zu erschaffen, welche immer fachlicher, d.h. anwendungsnaher sind.

Beispiele hierfür sind Komponenten, die eine Adresse oder eine Person repräsentieren, welche sowohl für den Entwickler als auch für den Anwender weitaus verständlicher sind.

Diese Vorgehensweise lässt sich bis hin zum Erhalt branchenspezifischer Geschäftsobjekte (z.B. Steuererklärung, Gesundheitszeugnis, etc.) fortsetzen, wodurch die Flexibilität natürlich immer stärker abnimmt.

Somit stellt sich bei Neuentwicklungen immer wieder erneut die Frage nach dem Umfang bzw. dem Grad der Anwendungsnähe der Komponenten.

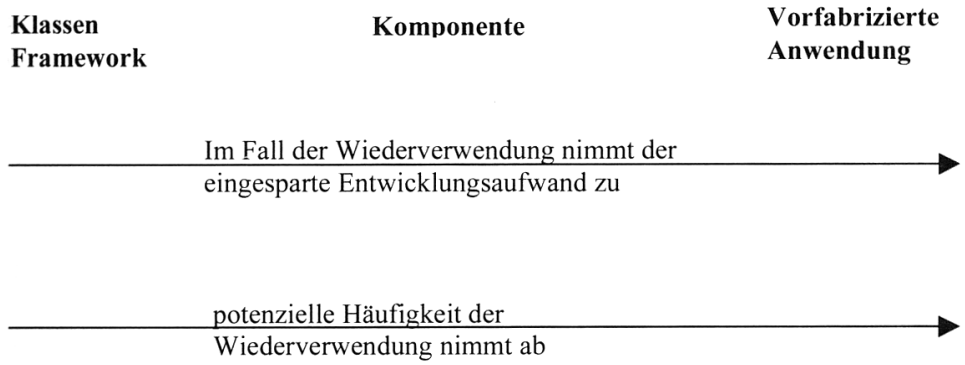


**Abbildung 3-2 Häufigkeit und Nutzen der Wiederverwendung von Komponenten**
**Quelle: Gruhn Volker, Thiel Andreas, *Komponentenmodelle*, Addison Wesley, 2000. S. 13.**

Abbildung 3-2 soll diesen Sachverhalt graphisch darstellen. Je spezialisierter die Komponenten sind, desto unwahrscheinlicher wird deren Wiederverwendung. Können solche „großen" Komponenten jedoch trotzdem wiederverwendet werden, so verkürzt dies den gesamten Entwicklungsaufwand erheblich.

## 3.3 CORBA

Der nun folgende Abschnitt soll die wesentlichen Merkmale der CORBA Spezifikation erläutern. Eine vollständige Vorstellung aller Konzepte soll im Rahmen dieser Arbeit jedoch nicht angestrebt werden. Es werden lediglich die Themenbereiche aus der CORBA Welt diskutiert, welche für das eigentliche Thema dieser Arbeit relevant sind.

### 3.3.1 Überblick

CORBA steht als Abkürzung für „**C**ommon **O**bject **R**equest **B**roker **A**rchitecture" und beschreibt in Form einer Spezifikation eine offene Architektur, welche die Interoperabilität von Anwendungen in einer verteilten Umgebung zum Ziel hat[7].
In diesem Umfeld kann CORBA durchaus als ein Industriestandard angesehen werden, welcher absolut unabhängig von den heutzutage existierenden Hard- und Softwareplattformen sowie Betriebssystemen ist.

[7] vgl. [Linnhoff-Popien, 1998]

Ein wesentlicher Grund dafür, dass sich CORBA einer breiten Akzeptanz erfreut; liegt nicht zuletzt in der Tatsache begründet, dass an der Entwicklung dieser Spezifikation sehr viele verschiedene Kräfte mitwirkten und auch heute noch mitwirken.

Die Entwicklung von CORBA als Standard wurde von der Object Management Group[8] initiiert, welche 1989 als ein Konsortium von einigen wenigen Firmen wie 3Com, American Airlines, Canon, Data General, Hewlett-Packard, Philips Telecommunications, Sun Microsystems und Unisys Corporation gegründet wurde. Ziel dieser Organisation war und ist es, durch die Verwendung objektorientierter Prinzipien die Interoperabilität von Anwendungen in heterogenen verteilten Umgebungen zu ermöglichen.

Heute stellt die OMG einen Verbund aus ca. 800 Mitgliedern dar. Es wurde stets Wert darauf gelegt, die Teilnehmer aus unterschiedlichsten Bereichen auszuwählen. Somit entstand eine Gemeinschaft aus Hardwareherstellern, Softwareentwicklern, Forschungsinstituten und Anwendern, welche alle gleichermaßen den CORBA Standard weiterentwickeln.

Das Modell, welches CORBA zugrunde liegt, sieht eine strikte Trennung zwischen den Schnittstellenbeschreibungen und deren Implementierungen vor. Jedes CORBA Objekt verfügt somit über eine separate Schnittstelle, über welche auf das Objekt zugegriffen werden kann. Auf diese Weise sollen unkontrollierbare Abhängigkeiten von Objekten vermieden werden.

### 3.3.2 Die Object Management Architecture

Bei der Entwicklung von CORBA steht die Object Management Architecture (OMA) als grundlegende Basis der Spezifikation für eine Beschreibung der Objektwelt, wie die OMG sie sieht (→ vgl. Abbildung 3-3).

Grundsätzlich besteht diese Architektur aus vier großen Hauptbestandteilen. Das Herzstück des Systems ist der Object Request Broker[9]. Er ist als zentrales Medium zum Datenaustausch und zur Kommunikation zwischen den einzelnen

[8] kurz OMG

[9] kurz ORB

Komponenten zu verstehen. Auf ihn greifen drei verschiedene Klassen von Diensten zu.

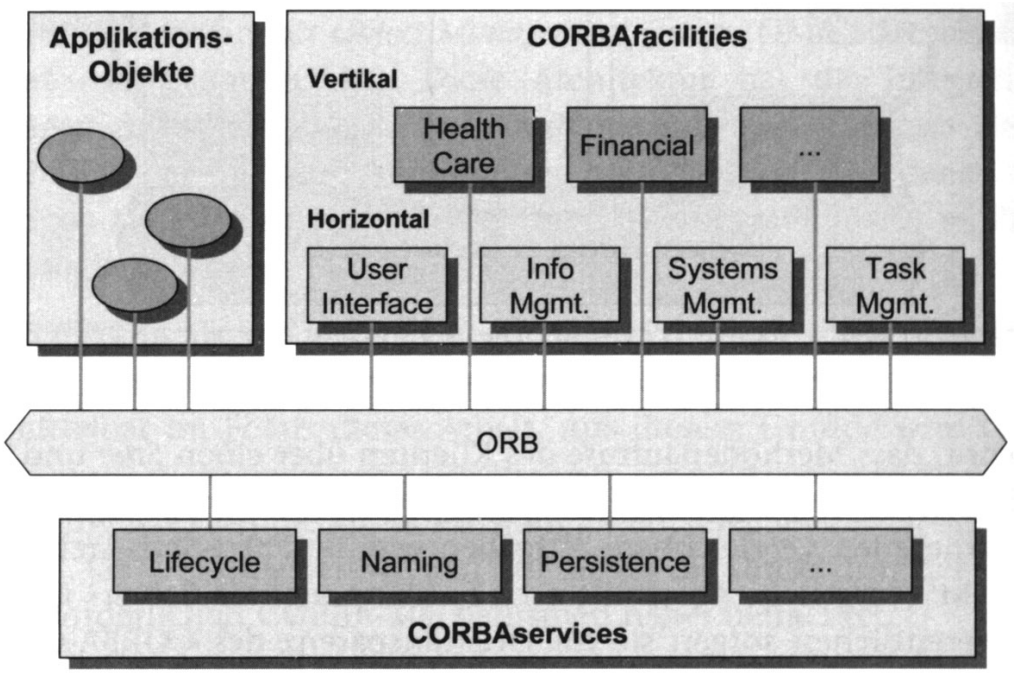


**Abbildung 3-3 Die Object Management Architecture der OMG**

**Quelle: Gruhn Volker, Thiel Andreas, *Komponentenmodelle*, Addison Wesley, 2000. S. 166.**

Die Applikationsobjekte sind die eigens vom Entwickler erstellten Objekte. Diese Objekte werden durch die sogenannten CORBAfacilities und CORBAservices ergänzt. Hierbei handelt es sich um zwei Typen von Diensten, welche weitestgehend unabhängig von der eigentlichen Kern-Spezifikation sind und standardisierte CORBA Objekte beschreiben, die häufig wiederverwendet werden können.

Da diese Bestandteile von elementarer Bedeutung für jedes OMA konforme System sind, sollen sie im weiteren Verlauf dieser Arbeit noch näher beschrieben werden.

### 3.3.3 Der Object Request Broker

Der Object Request Broker (ORB) ist ein Softwarebus, auf dem sich CORBA-Objekte registrieren können, was es ihnen möglich macht, sowohl auf die Dienste

anderer Objekte zuzugreifen als auch die eigenen Dienste über den Bus zur Verfügung zu stellen.

In diesem Zusammenhang ist interessant, dass die Schnittstellenbeschreibung eines solchen CORBA-Objektes in einer eigens für diesen Zweck entwickelten Sprache formuliert ist. Diese durch die Spezifikation definierte Interface Definition Language[10] beschreibt die Dienste des Objektes, welche durch den ORB zur Verfügung gestellt werden sollen, auf eine programmiersprachenunabhängige Art und Weise. Das führt dazu, dass die konkrete Implementierung eines CORBA Objektes für den jeweiligen Client bzw. den Dienstnehmer nicht sichtbar ist, d.h. sie kann im Grunde genommen in (fast) jeder objektorientierten Programmiersprache verfasst werden.

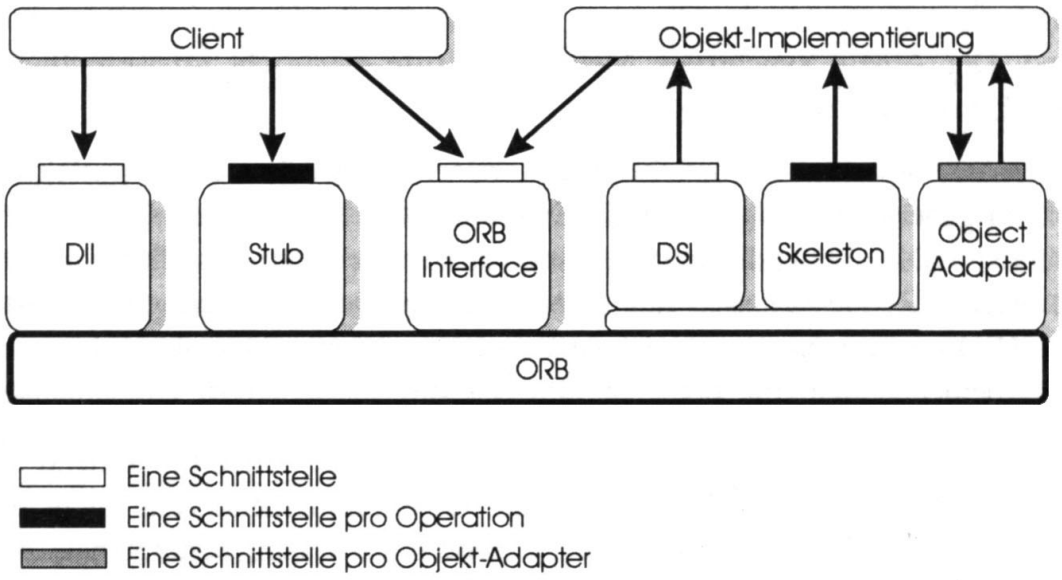


**Abbildung 3-4 Die Struktur eines Object Request Brokers**

**Quelle: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 31.**

Dadurch dass in einem verteilten System der gesamte einem Methodenaufruf zu Grunde liegende Netzwerkverkehr durch den ORB abgewickelt wird, ist es für den Dienstnehmer ebenfalls transparent, auf welchem Rechner und auf welcher Betriebssystemplattform sich das gewünschte CORBA Objekt befindet.

[10] kurz IDL

Die interne Struktur und Funktionsweise eines ORBs besteht aus mehreren Bestandteilen, welche im weiteren noch tiefergehend erläutert werden müssen, um ein Verständnis der ORB Spezifikation zu erhalten. Abbildung 3-4 soll den Aufbau des ORBs graphisch veranschaulichen und deutlich machen, welche Möglichkeiten der Client hat, um auf ein entferntes Objekt auf einem Server zuzugreifen.
Im wesentlichen haben CORBA Objekte zwei Möglichkeiten, um mit einem ORB in Verbindung zu treten, den statischen und den dynamischen Ansatz, welche im folgenden Teil der Arbeit näher beschrieben werden.

#### *3.3.3.1 Statische Nutzung des ORBs durch SII*

Abbildung 3-5 zeigt die statische Nutzung eines ORBs. Es wird deutlich, dass die Kommunikation über einen Stub bzw. ein Skeleton funktioniert, welche als stellvertretende Objekte für die eigentlichen Implementierungen stehen und der Erzeugung bzw. dem Handling von sogenannten Requests zwischen Client und Server dienen.
Der Ansatz von CORBA, den ORB auf diese Weise zu nutzen, wird auch als „Static Invocation Interface“ (SII) bezeichnet.
Ein solches Vorgehen wird als statisch bezeichnet, weil bei diesem Ansatz alle benötigten IDL-Beschreibungen der Schnittstellen zur Zeit der Anwendungsentwicklung bekannt sein müssen, welche daraufhin als Grundlage für einen IDL-Compiler dienen.

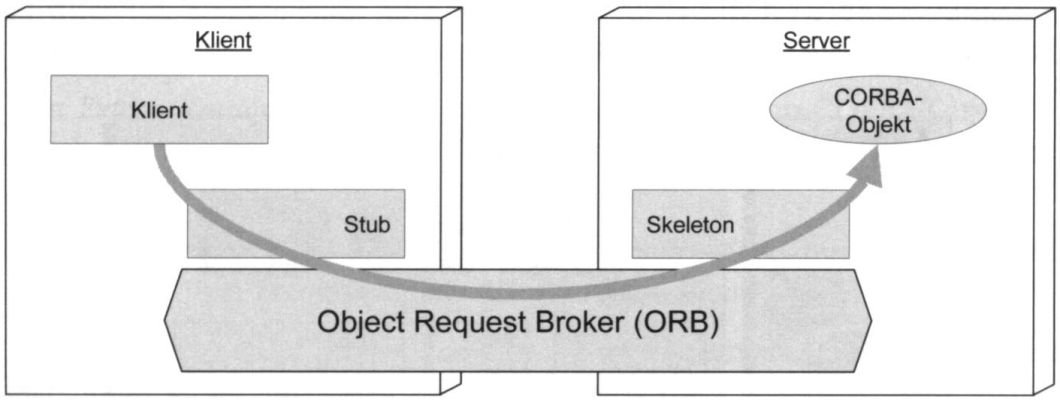


**Abbildung 3-5 Statischer Aufruf eines CORBA-Objektes**

**Quelle: Gruhn Volker, Thiel Andreas, *Komponentenmodelle*, Addison Wesley, 2000. S. 165.**

Ein solcher Compiler kann aufgrund dieser Informationen den sogenannten Stub, d.h. den clientseitigen und den Skeleton[11], d.h. den serverseitigen Anteil des CORBA Objektes entsprechend der gewünschten Zielprogrammiersprache generieren. Diese Strategie ist eine konsequente Umsetzung des Remote Procedure Calls (→ vgl. Abschnitt 2.4).

#### *3.3.3.2 Dynamische Nutzung des ORBs durch DII und DSI*

Neben der statischen Nutzung des ORBs über SII bietet CORBA auch die Möglichkeit des dynamischen Generierens von Methodenaufrufen an CORBA-Objekten zur Laufzeit. Auf der Seite des Clients wird dies als „Dynamic Invocation Interface“ (DII) und auf Seite des Servers als „Dynamic Skeleton Interface“ (DSI) bezeichnet.

##### *3.3.3.2.1 Dynamic Invocation Interface*

Ein Client kann beim Aufruf eines entfernten Objektes entweder das SII über einen Stub oder das DII nutzen, da diese beiden Schnittstellen das gleiche Protokoll zur Kommunikation verwenden. Diese beiden Varianten sind absolut gleichwertig, was bedeutet, dass die Objektimplementierung des entfernten Objektes anhand des empfangenen Methodenaufrufs nicht unterscheiden kann, welche Art des Methodenaufrufs der Client gerade verwendet hat (→ vgl. Abbildung 3-6).

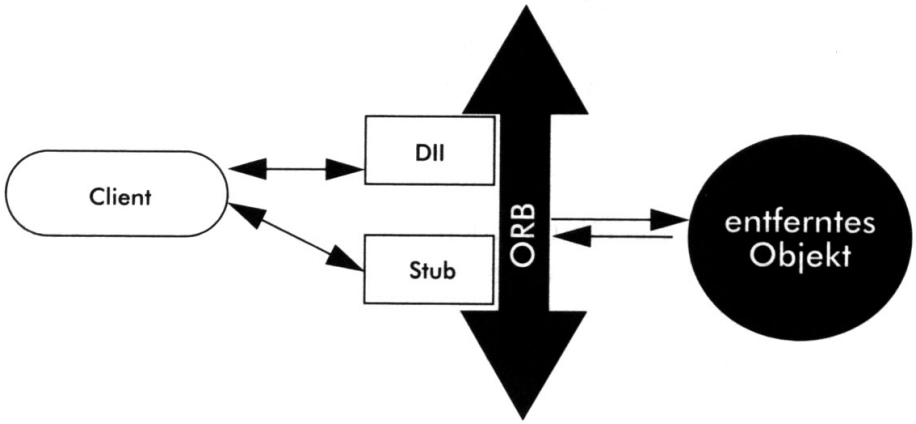


**Abbildung 3-6 Kommunikation des Clients mit einem entfernen Objekt**

**Quelle: Sayegh Andreas, *CORBA Standard, Spezifikation, Entwicklung*, O'Reilly 1999, 2.Auflage. S. 33.**

[11] Das Skeleton wird in der Literatur oftmals auch als Server-Stub bezeichnet

Wenn ein Client DII für einen Methodenaufruf an einem entfernten Objekt nutzen möchte, benötigt er den Namen der gewünschten Operation und die dazu gehörigen Parameter.
DII unterstützt in diesem Zusammenhang sowohl den synchronen als auch den asynchronen Aufruf von Methoden. Beim synchronen Methodenaufruf wird der Client solange gesperrt, bis dass die Antwort des entfernten Objektes vorliegt, während bei der asynchronen Variante die Clientanwendung umgehend nach dem Abschicken des Requests an das entfernte Objekt zurückkehrt und zu einem späteren Zeitpunkt das Ergebnis der Operation erfragt.
Die notwendigen Informationen zur korrekten Erstellung eines solchen dynamischen Requests bestimmt der Client mittels des sogenannten Interface Repositories. Das Interface Repository ist eine Datenbank bzw. ein allgemeiner Dienst, mit dem zur Laufzeit Informationen zu Objektschnittstellen abgerufen werden können.

#### *3.3.3.2.2 Dynamic Skeleton Interface*

Analog zu DII auf der Seite des Clients existiert eine ähnliche Möglichkeit auf der Seite des Servers, das sogenannte „Dynamic Skeleton Interface“.
Hiermit wird es dem Server ermöglicht, Anfragen eines Clients zu bearbeiten, welche nicht vorher statisch durch die CORBA-IDL beschrieben wurden. Der Server benötigt dazu lediglich die Informationen für einen vollständigen Methodenaufruf, d.h. den Namen der Operation und die Parameter.
Diese Technik wird häufig dazu verwendet, um eine Weiterleitung von Aufrufen an andere Systeme zu erreichen, welche sich eines anderen Komponentenmodells bedienen.

### *3.3.3.3 Der Object Adapter*

Der Object Adapter spielt innerhalb der CORBA Spezifikation eine zentrale Rolle. Er stellt die Verbindung zwischen den CORBA-Objekten und dem ORB dar. Zu den Aufgaben des Object Adapters gehören das Aktivieren bzw. Deaktivieren von Diensten, das Verteilen von Anfragen an diese Dienste und letztendlich auch das Aufrufen der konkreten Dienstoperationen.

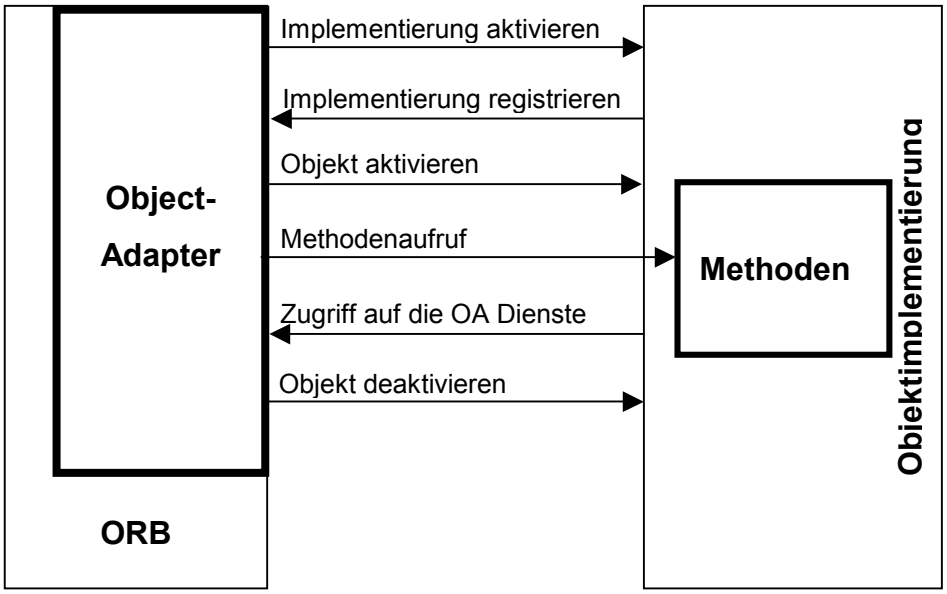


**Abbildung 3-7 Grundsätzliche Funktionsweise des Objekt Adapters**

**Quelle: In Anlehnung an: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 34.**

Abbildung 3-7 zeigt die allgemeine Funktionsweise des Adapters. Durch die CORBA-Spezifikation in der Version 2.0 wurde ein Object Adapter in Form des sogenannten „Basic Objects Adapters“ (BOA) eingeführt. Er implementierte die zuvor beschriebenen Funktionalitäten eines Object Adapters allerdings nur in sehr rudimentären Zügen.

Bereits nach kurzer Zeit wurde der OMG deutlich, dass diese erste Spezifikation eines Object Adapters nicht allen Anforderungen genügen würde. Aus diesem Grund wurde der BOA im Zuge der CORBA-Spezifikation 2.2 vollständig durch eine weiterentwickelten Object Adapter ersetzt. Dieser Adapter trägt den Namen „Portable Object Adapter“ (POA) und erweitert den BOA um einige Funktionalitäten wie beispielsweise die Unterscheidung zwischen persistenten und transienten Objekten sowie die Möglichkeit zur expliziten oder on-demand-Aktivierung von Objekten.

Im Gegensatz zum BOA kann eine serverseitige Anwendung mehrere POAs haben, um beispielsweise verschiedene Arten von CORBA-Objekten zu unterstützen.

#### *3.3.3.4 Das ORB-Interface*

Das ORB-Interface bietet dem Entwickler eine Sammlung von Methoden an, mit der er direkten Zugriff auf den ORB erhält. Diese Methoden sind sozusagen das Application Programming Interface (API) zum ORB.

Durch dieses Interface ist es unter anderem möglich, eine Liste der Kennungen aller Standarddienste des ORBs zu erhalten. Weiterhin ist das ORB-Interface in der Lage, den ORB zu initialisieren und programmiersprachliche Referenzen auf diesen ORB zurückzuliefern.

### 3.3.4 Die Interface Definition Language

Die Interface Definition Language (IDL) der OMG ist das grundlegende Instrument des CORBA-Standards, um Objektschnittstellen zu entwerfen. Diese Sprache zeichnet sich dadurch aus, dass sie unabhängig von jeglichen Programmiersprachen ist.

Des weiteren hat diese Sprache rein deskriptiven, d.h. beschreibenden Charakter. Sie enthält keine sprachlichen Elemente zur Programmierung eines Ablaufs, wie z.B. Schleifen, Variablen oder andere algorithmische Elemente.

Durch die IDL der OMG sollen lediglich die verfügbaren Methoden und die zu diesen Methodenaufrufen notwendigen Parametern kenntlich gemacht werden. Die eigentliche Implementierung der CORBA-Objekte erfolgt dann letztendlich in einer konventionellen Programmiersprache.

Die Abbildung von IDL-Konstrukten auf Konstrukte einer konkreten Programmiersprache erfolgt durch sogenannte IDL-Compiler. Man bezeichnet ein solches Vorgehen als Language Mapping. Zur Zeit existieren solche Language Mappings für Java, C++, Smalltalk, Ada, Cobol, Lisp und Python. Im weiteren Verlauf dieses Kapitels soll beispielhaft die Abbildung einfacher Datentypen (→ vgl. Tabelle 3-3) sowie das Language Mapping (→ vgl. Abbildung 3-9) für die Programmiersprache Java gezeigt werden. Abbildung 3-8 zeigt ein einfaches Beispiel einer solchen IDL-Schnittstellenbeschreibung.

Die Syntax der Interface Definition Language ist an die Syntax der Sprache C++ angelehnt. Darüber hinaus existieren einige weitere Schlüsselwörter, welche die Schnittstellenbeschreibung vereinfachen sollen.

```
// Beispiel für eine IDL-Kommentarzeile
module BeispielModul{
      interface SummenInterface{
            void berechneSumme(
                  in float x,
                  in float y,
                  out float ergebnis
            );
      };
      interface StringInterface{
            long bestimmeDieLaengeDesStrings(
                  in string parameter
            );
      };
      interface MultiplikationsInterface{
            attribute float x;
            void multipliziereParamMitX(
                  inout float parameter
            );
      };
};
```

**Abbildung 3-8 Ein einfaches Beispiel einer IDL-Schnittstellenbeschreibung**

Mit dem Schlüsselwort „module“ können mehrere Schnittstellen, welche wiederum durch das Schlüsselwort „interface“ gekennzeichnet sind, zusammengefasst werden. Methodenparameter können entweder Inputparameter („in“), Outputparameter („out“) oder bidirektionale Parameter („inout“) sein. Attribute einer Schnittstelle werden stets durch das Schlüsselwort „attribute“ kenntlich gemacht. Mittels eines Vererbungsmechanismus kann eine Schnittstelle Eigenschaften anderer Schnittstellen erben, wobei auch Mehrfachvererbungen möglich sind.

Im Abschnitt 3.3.7.1 dieser Arbeit wird das Thema der Interface Definition Language im Zusammenhang mit der Programmiersprache Java noch näher erläutert werden.

### 3.3.5 CORBA-Dienste

Parallel zur Entwicklung der eigentlichen CORBA-Spezifikation wurden auch stets eine ganze Reihe von CORBA-Diensten, den sogenannten CORBAservices, standardisiert. Welche Dienste dem Entwickler letztendlich wirklich zur Verfügung

stehen ist vom verwendeten ORB abhängig. Hier liegt die Entscheidung beim jeweiligen Hersteller, welche Dienste im Rahmen des eigenen Produkts implementiert werden und welche nicht.
Wichtig in diesem Zusammenhang ist, dass diese Dienste generell unabhängig von der zu entwickelnden Anwendung sind und dass es zwischen ihnen keinerlei Möglichkeiten zur Kommunikation gibt, d.h. es existieren keine dienstübergreifenden Schnittstellen. Sie werden dem Entwickler in Form von eigenen CORBA-Objekten zur Verfügung gestellt. Tabelle 3-1 soll eine Übersicht über die grundsätzlichen Standarddienste vermitteln, welche durch die OMG beschrieben wurden.

| **Name des Dienstes** | **Funktion des Dienstes** |
|---|---|
| **Life Cycle Service** | Der Life Cycle Service ist ein Standarddienst, der das Erzeugen, Löschen, Kopieren und Verschieben von Objekten ermöglicht. |
| **Naming Service** | Der Naming Service ist für das Abbilden von verständlichen Bezeichnungen auf Objektreferenzen zuständig. Dieser Vorgang wird auch als "Name Binding" bezeichnet. |
| **Event Service** | Durch diesen Dienst wird die Übertragung von Nachrichten zwischen Objekten ermöglicht. Hierbei werden die Objekte, welche die Nachrichten senden als Anbieter (engl. Supplier) und die Adressaten dieser Nachrichten als Verbraucher (engl. Consumer) bezeichnet. Der Event Service hat die Fähigkeit zum Broadcasten von Nachrichten, d.h. zum Versenden von Nachrichten an alle momentan bekannten Objekte.<br>Als Kanal für den Nachrichtenaustausch zwischen den Objekten dient der ORB. |
| **Persistency Service** | Dieser Dienst wird auch als Persistence Object Service (POS) bezeichnet und meint eine Unterstützung der Persistenz von Objekten, d.h. dass der Objektzustand |

| | |
|---|---|
| | dauerhaft auf einem permanenten Speichermedium gesichert wird, um ihn zu erhalten, selbst wenn die Anwendung beendet wurde. |
| **Externalization Service** | Der Externalization Service kann Objekte, bzw. deren Zustände in einen Datenstrom verwandeln. Ein solcher Strom kann daraufhin transportiert werden und sogar den ORB verlassen. Diesen Vorgang bezeichnet man als Externalisieren. Die transportierten Datenströme können durch den gegenteiligen Vorgang des Internalisierens wiederum in gültige Objekte umgewandelt werden. |
| **Relationship Service** | Der Relationship Service erlaubt die Darstellung von Entitäten und den zwischen ihnen bestehenden Relationen. Mit Entitäten sind in diesem Zusammenhang Dinge des realen Lebens gemeint, welche naturgemäß miteinander in Verbindung stehen. Typischerweise werden solche Entitäten durch entsprechende CORBA-Objekte repräsentiert. Solche Relationen zwischen den Entitäten zeichnen sich dadurch aus, dass sie bidirektional traversierbar sind und als eigene CORBA-Objekte vorliegen, d.h. sie können mit Attributen sowie Methoden versehen werden. |
| **Concurrency Control Service** | Der Concurrency Control Service regelt den Zugriff mehrerer Clients auf eine gemeinsam genutzte Ressource. Hierbei geht es insbesondere um die Auflösung bzw. die Vermeidung von diesbezüglichen Konflikten. Bei vielen parallelen Zugriffen wird dies durch ein Locking-Verfahren ermöglicht, d.h. Ressourcen können einem Client exklusiv zur Verfügung gestellt werden, solange er den sogenannten „Lock“ innehält. |

| | |
|---|---|
| **Transaction Service** | Durch den Transaction Service wird es den Objekten ermöglicht, Transaktionen auszuführen. (→ vgl. Abschnitt 2.6) |

**Tabelle 3-1 Übersicht über die gebräuchlichsten CORBA-Dienste**

**Quelle: In Anlehnung an: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 73 ff.**

### 3.3.6 Interoperabilität

Nachdem die einzelnen Komponenten eines CORBA konformen Systems beschrieben wurden, stellt sich an dieser Stelle die Frage, wie diese Bestandteile miteinander in Verbindung treten können. Die OMG hat die technischen Prinzipien hinter diesem Thema unter dem Begriff der Interoperabilität zusammengefasst.

#### *3.3.6.1 Typen der Interoperabilität*

Grundsätzlich wird unter dem Begriff der Interoperabilität die Fähigkeit verstanden, dass unterschiedliche Komponenten miteinander kommunizieren können. Man muss allerdings verschiedene Arten der Interoperabilität unterscheiden, welche durch die Tabelle 3-2 erläutert werden sollen.

| Typ der Interoperabiltät | Erläuterung |
|---|---|
| Interoperabilität zwischen Objekten | Die wohl einfachste Art der Interoperabilität ist die Kommunikation zwischen mehreren Objekten. Damit ist gemeint, dass Objekte Nachrichten in Form von Methodenaufrufen an andere Objekte schicken können. Durch den Einsatz geeigneter Komponentenmodelle spielt es dabei für den Entwickler keine Rolle, ob diese Objekte lokal oder entfernt vorliegen. |
| Interoperabilität zwischen Programmiersprachen | In manchen Fällen ist es notwendig, dass verschiedene Programmiersprachen zum Einsatz gebracht werden müssen. Die Aufgaben bei dieser |

| | |
|---|---|
| | Art der Kommunikation, wie z.B. das Konvertieren von Datentypen, werden typischerweise von einem CORBA-ORB erledigt. |
| Interoperabilität zwischen Rechnern | Im Zusammenhang mit Verteilten Systemen wird in jedem Fall eine Kommunikation zwischen Objekten benötigt, welche auf verschiedenen Rechnern liegen. Der klassische Ansatz zur Lösung dieses Problems stellt der Remote Procedure Call (→ vgl. Abschnitt 2.4) dar. |
| Interoperabilität zwischen ORBs | In seltenen Fällen kann es notwendig werden, dass ein Kommunikationskanal zwischen mehreren ORBs etabliert werden muss.<br>Gründe hierfür könnten sein, dass ein bestehender ORB mit einem anderen ORB zusammenarbeiten soll, welcher andere Language Mappings unterstützt oder andere CORBAservices zur Verfügung stellt. Ab CORBA 2.0 sind Ansätze solcher ORB übergreifenden Verbindungen in der Spezifikation enthalten. |
| Interoperabilität zwischen Kommunikationsplattformen | In allen Fällen, in denen außer einem CORBA-ORB noch andere Mechanismen zur Kommunikation in einem System vorhanden sind, spricht man von einer Interoperabilität zwischen Kommunikations-plattformen. Um solch eine Art der Interoperabiltät zu ermöglichen, werden sogenannte Gateways zum Einsatz gebracht, welche Konvertierungsaufgaben wahrnehmen wie beispielsweise das Umwandeln von Datenformaten und die Zuordnung der Objektreferenzen. Ein praktisches Beispiel hierfür ist das vom CORBA-ORB Orbix 2.0 bereitgestellte Gateway zum OLE/COM System von Microsoft. |

| | |
|---|---|
| Interoperabilität zwischen Diensten | Die allgemeinste Form der Interoperabilität besteht zwischen verschiedenen Diensten bzw. Anwendungen. Das Problem, welches sich bei einer solchen Art der Kommunikation ergibt ist, dass die auszutauschenden Daten hinsichtlich ihrer Struktur interpretiert werden müssen, um sie zueinander kompatibel machen zu können. Für ein solches Szenario gibt es zur Zeit noch keinerlei Ansätze eines allgemeinen Standards. |

**Tabelle 3-2 Verschiedene Arten der Interoperabilität**

**Quelle: In Anlehnung an: Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998, S. 37 ff.**

#### *3.3.6.2 CORBA-Protokolle zur Unterstützung der Interoperabilität*

Durch die CORBA Spezifikation in der Version 2.0 wurden erstmalig Schnittstellen für die Verbindung zwischen Objekten verschiedener Plattformen definiert.

Den Grundstein, d.h. also die Basis für alle existierenden CORBA konformen Netzprotokolle bildet das sogenannte General-Inter-ORB-Protocol (GIOP). Hierbei handelt es sich um eine sehr allgemeine und abstrakt gehaltene Protokollspezifikation, welche von der OMG entworfen wurde.

Ausgehend von diesem Protokoll existiert noch ein weiteres von der OMG definiertes Protokoll, dass eine Spezialisierung von GIOP darstellt. Diese Spezialisierung trägt den Namen Internet-Inter-ORB-Protocol (IIOP) und meint eine Ansammlung von Implementierungsregeln für ein CORBA Protokoll auf der Basis des TCP/IP Protokolls.

TCP/IP steht für das weltweit eingesetzte Protokoll zur Kommunikation im Internet. Es besteht aus zwei eigenständigen Protokollen, welche bei der Datenübertragung zwischen zwei Rechnern zusammenarbeiten. Das Grundprotokoll IP (Internet Protocol) ermöglicht ein Aufteilen der Daten in sogenannte IP-Pakete. Diese Pakete werden mit einer eindeutigen Adresse (IP-

Adresse) versehen und können daraufhin durch IP versendet werden. Aufgrund der Tatsache, dass die Pakete unter Umständen alle unterschiedliche Wege durch das Netzwerk zum Empfänger einschlagen, könnte es bei einer reinen IP-Datenübertragung dazu führen, dass einige der Pakete nicht beim Empfänger ankommen würden. Aus diesem Grund wurde TCP (Transmission Control Protocol) entwickelt, welches auf dem IP Protokoll aufsetzt und die Aufgabe hat, die empfangenen IP-Pakete wieder in die richtige Reihenfolge zu bringen und zu prüfen, ob alle gesendeten Pakete ordnungsgemäß beim Empfänger angekommen sind. Ist dies nicht der Fall, so kann der Absender über TCP aufgefordert werden, das fehlende Paket nochmals zu senden.

Die Trennung zwischen der allgemeinen Protokollspezifikation GIOP und der TCP/IP Spezialisierung IIOP ermöglicht es erst, dass auch andere Protokolle neben IIOP entwickelt werden können. Selbst zukünftige Protokolle, welche unter Umständen auf einem anderen Protokoll als TCP/IP basieren, wären somit in der Lage CORBA konform zu sein, sofern sie den GIOP Implementierungsregeln der OMG entsprechen.

#### *3.3.6.3 Die interoperable Objektreferenz*

Unter dem Begriff einer Objektreferenz versteht man grundsätzlich den Verweis auf einen Speicherort, an dem ein bestimmtes Objekt zu finden ist.

In der CORBA Welt werden in fast allen Fällen der Weitergabe von Objekten sogenannte interoperable Objektreferenzen (IORs) eingesetzt. Alternative Möglichkeiten zur Weitergabe eines Objektes wären beispielsweise das Versenden einer Kopie des Originalobjektes oder das Versenden des Originals selber.

Beim Versenden einer Kopie des Originals würde es dazu kommen, dass das betreffende Objekt doppelt existiert, was In der Regel nicht erwünscht ist, da die Eindeutigkeit und Einzigartigkeit eines existierenden Objektes stets angestrebt werden sollte.

Bei der Weitergabe des Originalobjektes an den Empfänger müsste das Originalobjekt ausgehend von seinem aktuellen Kontext in einen anderen überführt werden, was in aller Regel nur mit einem hohen technischen Aufwand möglich ist.

Die einfachste Variante ist also die Nutzung einer Referenz auf ein Objekt. Darüber hinaus haben die interoperablen Objektreferenzen von CORBA Objekten

den Vorteil, dass sie sich durch den Aufruf von *CORBA::ORB::object_to_string* in eine ASCII-Zeichenkette umwandeln lassen. Diese Zeichenketten können dann auf einfachste Art und Weise versendet werden, wobei der Empfänger seinerseits diese Kette durch den Aufruf von *CORBA::ORB::string_to_object* wieder in eine reguläre Objektreferenz umwandeln kann. Daraufhin stehen ihm die Methoden des entfernten CORBA Objektes zur vollen Verfügung.

### 3.3.7 CORBA & Java

Aufgrund der Tatsache, dass sich das Thema dieser Arbeit mit einem Framework beschäftigt, welches vollkommen auf der Programmiersprache Java basiert, ist es das Ziel dieses Abschnitts, exemplarisch den Zusammenhang zwischen der Java Welt und der CORBA Welt aufzuklären.

#### *3.3.7.1 Das OMG-Language Mapping von IDL nach Java*

Anhand der folgenden Tabelle 3-3 soll das Language Mapping einfacher Datentypen der Interface Definition Language auf die Datentypen der Programmiersprache Java gezeigt werden.

| IDL-Datentyp | Java-Datentyp |
|---|---|
| short | short |
| long | int |
| long long | long |
| string | java.lang.String |
| wstring | java.lang.String |
| float | float |
| double | double |
| long double | - |
| char | char |
| wchar | char |
| octet | byte |
| boolean | boolean |

**Tabelle 3-3 Abbildung einfacher IDL- auf Java-Datentypen nach CORBA 2.2**

**Quelle: Sayegh Andreas, *CORBA Standard, Spezifikation, Entwicklung*, O'Reilly 1999, 2.Auflage. S. 65.**

Komplexere Datentypen lassen sich natürlich nicht so einfach in Java überführen. Sie werden später in Form eigener Hilfsklassen abgebildet.

Übergibt man eine IDL-Schnittstellenbeschreibung (→ vgl. Abbildung 3-8) einem sogenannten IDL2Java-Compiler, so wird aus diesem IDL-Code ein Gerüst aus Java-Klassen erzeugt, welche die Infrastruktur für die Kommunikation zwischen dem Client und dem Server darstellen. Es ist nun die Aufgabe des Programmierers, basierend auf diesem Gerüst eine konkrete Implementierung für das spätere CORBA-Objekt zu entwickeln. Das Ergebnis eines solchen IDL2Java-Compilers soll die Abbildung 3-9 zeigen.

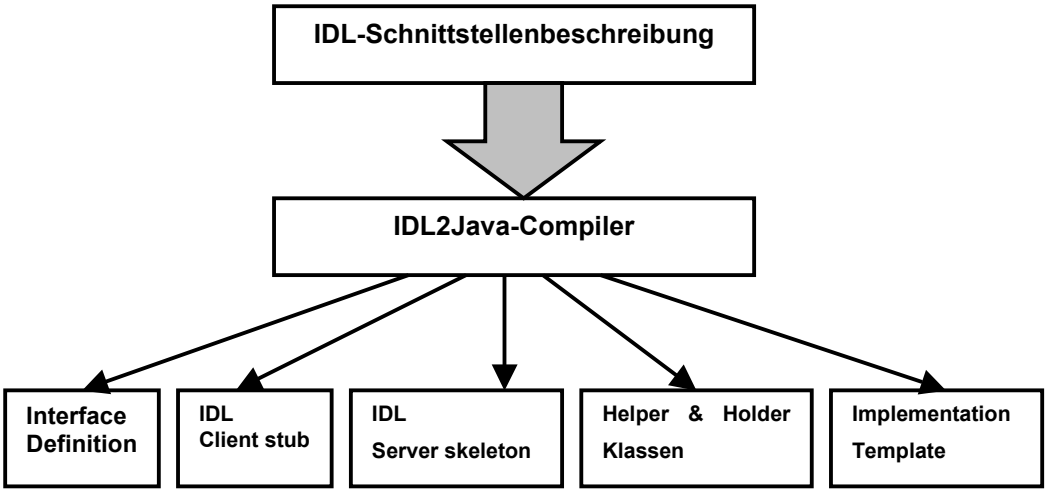


**Abbildung 3-9 Der IDL2Java-Compiler**

**Quelle: In Anlehnung an: Oracle Corporation, *Oracle8i CORBA Developer's Guide and ReferenceRelease 3 (8.1.7)*, 2000.**

Das *Implementation Template* ist eine Java-Klasse, die als Rahmen für die Implementierung des Server Objektes genutzt werden kann. Der Entwickler kann an dieser Stelle die benötigte Logik direkt in die vorgefertigten Methodenrümpfe implementieren.

*IDL Client stub* und *IDL Server skeleton* sind die bereits beschriebenen client- bzw. serverseitigen Anteile, die einen entfernten Methodenaufruf überhaupt erst ermöglichen (→ vgl. Abschnitt 2.4).

Die *Interface Definition* ist im Sinne der Programmiersprache Java ein Interface und beinhaltet eine Sammlung von Methodensignaturen derjenigen Methoden, welche später dem Client zur Verfügung gestellt werden sollen.

Die sogenannten *Helper Klassen* sind Klassen, welche über statische Methoden zur Behandlung von Datentypen verfügen. Für jeden Datentyp (egal ob elementar oder selbst definiert) existiert solch eine *Helper Klasse*. Die *Holder Klassen* benutzen diese *Helper Klassen*, um die Werte der Daten aus dem Datenstrom auszulesen und in denselben wieder hinein zu schreiben. Sie sind die Garanten dafür, dass die Werte des Datenstroms den erwarteten Datentyp aufweisen.

#### *3.3.7.2 Beispielhafte Erstellung eines CORBA-Objektes mit Java*

Nachdem in den vorangegangenen Abschnitten dieser Arbeit die wesentlichen Kernbestandteile einer CORBA Umgebung beschrieben wurden, soll nun die typische Vorgehensweise zur Erstellung eines CORBA-Objektes mit Java anhand eines einfachen Beispiels erläutert werden. Dieses Beispiel nutzt den Visibroker CORBA-ORB 3.4, dessen aktuellste Version 4.5 noch genauer in Kapitel 5.4.4.2 vorgestellt wird. Zunächst einmal soll eine einfache IDL-Schnittstellenbeschreibung (→ vgl. Abbildung 3-10) erstellt werden, welche als Ausgangsbasis für das Beispiel dienen soll.

```
// Einfaches CORBA-Beispiel
module simpleCORBAExample{
      interface HelloWorld{
            string hello();
      };
};
```

**Abbildung 3-10 IDL-Schnittstellenbeschreibung für das CORBA-Beispiel**

Diese IDL-Beschreibung ist bewusst denkbar einfach gehalten. Es existiert lediglich nur ein einziges Modul „`simpleCORBAExample`“ in dem sich eine Schnittstelle namens „`HelloWorld`“ befindet. Diese Schnittstelle verfügt über eine Methode „`hello`“, die ohne jegliche Parameter aufgerufen werden kann und als Ergebnis einen String zurückliefert. Als nächsten Schritt wird diese IDL-Schnittstellenbeschreibung durch den idl2java Compiler auf entsprechende Java Klassen abgebildet. Die Ergebnisse eines solchen Arbeitsgangs sollen nun vorgestellt werden, um ein besseres Verständnis der Abbildung 3-9 zu ermöglichen. Grundsätzlich werden in IDL definierte Module im Sinne von Java

als Packages behandelt. Zuerst einmal wird auf der Basis der obigen IDL-Beschreibung ein entsprechendes Java Interface „HelloWorld“ (→ vgl. Abbildung 3-11) erzeugt, welches die Signatur der einzigen Methode hello() beinhaltet. Diese Schnittstelle wird später durch das eigentliche CORBA-Objekt implementiert und entspricht der „Interface Definition“ aus Abbildung 3-9.

```
package simpleCORBAExample;
public interface HelloWorld extends
  com.inprise.vbroker.CORBA.Object {

    public java.lang.String hello();
}
```

**Abbildung 3-11 Generiertes Java Interface für die IDL-Schnittstelle des Beispiels**

Den clientseitigen Stub, welcher für das Erzeugen und Senden von Nachrichten an den Server benötigt wird, zeigt die Abbildung 3-12. An dieser Stelle wird deutlich, dass der Stub den entfernten Methodenaufruf in Form einer Nachricht über einen Nachrichtenstrom auslöst. Anschließend kann das Ergebnis der Methode wiederum aus demselben Strom ausgelesen werden (Marshalling bzw. Unmarshalling → vgl. Abschnitt 2.4).

```
package simpleCORBAExample;
public class _st_HelloWorld extends
com.inprise.vbroker.CORBA.portable.ObjectImpl implements
simpleCORBAExample.HelloWorld {
  protected simpleCORBAExample.HelloWorld _wrapper = null;

  public simpleCORBAExample.HelloWorld _this() {
    return this;
  }

  public java.lang.String[] _ids() {
    return __ids;
  }

  private static java.lang.String[] __ids = {
    "IDL:simpleCORBAExample/HelloWorld:1.0"
  };

  public java.lang.String hello() {
    org.omg.CORBA.portable.OutputStream _output;
    org.omg.CORBA.portable.InputStream _input;
    java.lang.String _result;
    while(true) {
```

```
      //Vorbereiten der Nachricht für den entfernten Methodenaufruf
      _output = this._request("hello", true);
      try {
        //„hello“ Aufruf
        _input = this._invoke(_output, null);
        //Ergebnis aus dem Strom auslesen...
        _result = _input.read_string();
      }
      catch(org.omg.CORBA.TRANSIENT _exception) {
        continue;
      }
      break;
    }
    return _result;
  }
}
```

**Abbildung 3-12 Client-Stub für das „HelloWorld“ CORBA-Beispiel**

Der für die Seite des Servers generierte Code, d.h. die Skeleton Klasse, ist nachfolgend abgebildet. Die eigentliche Implementierung des CORBA Objektes findet üblicherweise in einer Subklasse von `_HelloWorldImplBase` statt. Hier ist wiederum ersichtlich, dass durch diese Klasse zunächst die Methode serverseitig ausgeführt wird, worauf anschließend das Ergebnis zurück in den Datenstrom geschrieben wird, aus dem es durch den Client wieder ausgelesen werden kann.

```
abstract public class _HelloWorldImplBase extends
com.inprise.vbroker.CORBA.portable.Skeleton implements
simpleCORBAExample.HelloWorld {

  protected simpleCORBAExample.HelloWorld _wrapper = null;

  [...]

  private static java.lang.String[] __ids = {
    "IDL:simpleCORBAExample/HelloWorld:1.0"
  };

  //Ermittelt die verfügbaren Methoden des CORBA-Objektes
  public org.omg.CORBA.portable.MethodPointer[] _methods() {
    org.omg.CORBA.portable.MethodPointer[] methods = {
      new org.omg.CORBA.portable.MethodPointer("hello", 0, 0),
    };
    return methods;
  }

  [...]

  //!! Serverseitiger Aufruf der „hello“ Methode !!
  //Zurückschreiben des Ergebnisses in den Ausgabestrom zum Client
  public static boolean _execute(simpleCORBAExample.HelloWorld
```

```
    _self, int _method_id,
    org.omg.CORBA.portable.InputStream _input,
    org.omg.CORBA.portable.OutputStream _output) {
      switch(_method_id) {
        case 0: {
          //Aufruf der Methode
          java.lang.String _result = _self.hello();
          //Ergebnis in den Datenstrom schreiben
          _output.write_string(_result);
          return false;
        }
      }
    throw new org.omg.CORBA.MARSHAL();
  }
}
```

**Abbildung 3-13 Server-Skeleton für das „HelloWorld“ CORBA-Beispiel**

Aus Sicht des Entwicklers muss an den oben beschrieben Dateien nichts geändert werden. Die einzige Aufgabe, die ihm nicht abgenommen werden kann, ist die letztendliche Implementierung des CORBA Objektes. Eine mögliche Implementierung für das einfache Beispiel in diesem Abschnitt zeigt die folgende Abbildung 3-14.

```
public class _example_HelloWorld extends
simpleCORBAExample._HelloWorldImplBase {

  public _example_HelloWorld(java.lang.String name) {
    super(name);
  }

  public _example_HelloWorld() {
    super();
  }

  //!! An dieser Stelle erfolgt die serverseitige Implementierung
  // für die hello()-Methode des CORBA -Objektes !!
  // Es wird lediglich ein einfacher String zurückgegeben
  public java.lang.String hello() {
    return new String(„Hello World“);
  }
}
```

**Abbildung 3-14 Beispielhafte Implementierung des HelloWorld CORBA-Objektes**

Eine solche Implementierung des CORBA-Objektes muss im nun folgenden Schritt aus Sicht des Servers für die Clients über den ORB verfügbar gemacht werden. Um dies zu erreichen, muss eine weitere Klasse (→ vgl. Abbildung 3-15) geschrieben werden, welche auf dem Server als Prozess gestartet werden kann. Das gehört typischerweise ebenfalls zu den Tätigkeiten des Entwicklers.
Zu den Aufgaben dieser Klasse gehört es, zum einen den ORB sowie einen Object Adapter zu initialisieren. In diesem Beispiel wird dafür der Basic Object Adapter (BOA) verwendet[12].

```
public class Server {

  public static void main(String[] args) {

    // CORBA-ORB initialisieren
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);


    // Basic Object Adapter initialisieren
    org.omg.CORBA.BOA boa = orb.BOA_init();

    // CORBA-Objekt instanzieren
    simpleCORBAExample.HelloWorld remotableObject =
      new _example_HelloWorld("HelloWorld");

    // Exportieren des instanzierten CORBA-Objektes
    boa.obj_is_ready(remotableObject);
    System.out.println(remotableObject + " is ready.");

    // Warten auf eingehende Anfragen der Clients
    boa.impl_is_ready();
  }
}
```

**Abbildung 3-15 Die Server-Klasse für das CORBA-Objekt**

Bevor ein Client (→ vgl. Abbildung 3-16) auf dieses CORBA-Objekt zugreifen kann, muss er ähnlich dem Server zunächst den ORB initialisieren. Daraufhin hat er die Möglichkeit, sich eine Referenz auf das gewünschte Objekt zu holen. Hat er dies erfolgreich abgeschlossen, kann er sämtliche Methoden des entfernten Objektes nutzen, so als wären sie lokal vorhanden.

[12] Modernere CORBA-ORBs würden an dieser Stelle den POA (Portable Object Adapter) nutzen (→ vgl. Abschnitt 3.3.3.3). Für dieses Beispiel ist dies jedoch nicht von Bedeutung.

```
public class Client {
  public static void main(String[] args) {

    // CORBA-ORB initialisieren
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // Referenz auf das CORBA-Objekt holen
    simpleCORBAExample.HelloWorld remoteObject =
      simpleCORBAExample.HelloWorldHelper.bind(orb,"HelloWorld");

    // Ab jetzt können die Methoden des entfernten Objektes
    // genutzt werden.....
    System.out.println(remoteObject.hello());
  }
}
```

**Abbildung 3-16 Beispiel eines einfachen CORBA-Clients**

## 3.4 Enterprise JavaBeans

### 3.4.1 Überblick

Zunächst einmal handelt es sich bei Enterprise JavaBeans (kurz EJBs) nicht um ein fertiges Produkt, sondern um eine Spezifikation. Es gibt jedoch von SUN eine Referenzimplementierung, welche frei im Internet verfügbar ist und beispielhaft zeigen soll, wie eine mögliche Realisierung der Spezifikation aussehen könnte. Diese Implementierung ist unter der Bezeichnung Java 2 Enterprise Edition (J2EE) bekannt. Ähnlich wie im vorangegangenen Abschnitt über CORBA soll das Modell der Enterprise JavaBeans an dieser Stelle der Arbeit hinsichtlich der Architektur, der Bestandteile des Systems und der Vorgehensweise bei der Entwicklung beleuchtet werden.
Im Gegensatz zu CORBA als Komponentenmodell sind Enterprise JavaBeans nicht programmiersprachenunabhängig, sondern basieren vollständig auf der Programmiersprache Java von SUN.
SUN selber beschreibt das Modell der Enterprise JavaBeans wie folgt:

*„Die Enterprise JavaBeans-Architektur ist eine Komponenten-Architektur für die Entwicklung und Inbetriebnahme Komponenten-basierter Geschäftsanwendungen. Applikationen, die unter Verwendung der Enterprise JavaBeans-Architektur geschrieben werden, sind skalierbar, transaktionsorientiert und Mehrbenutzergeeignet. Dies Applikationen können einmal geschrieben und dann auf jeder Serverplattform in Betrieb genommen werden, die die Enterprise JavaBeans-Spezifikation unterstützt“*[13]

[13] vgl. [SUN J2EE Spezifikation, 2001]

### 3.4.2 Architektur des Enterprise JavaBean Modells

In bezug auf Verteilte Architekturen lässt sich dieses Modell dadurch charakterisieren, dass die EJBs die Rolle der serverseitigen Anwendungslogik übernehmen. Diese Logik wird in Form von Komponenten, den sogenannten Enterprise-Beans zur Verfügung gestellt. Abbildung 3-17 soll einen Gesamtüberblick der EJB-Architektur vermitteln.

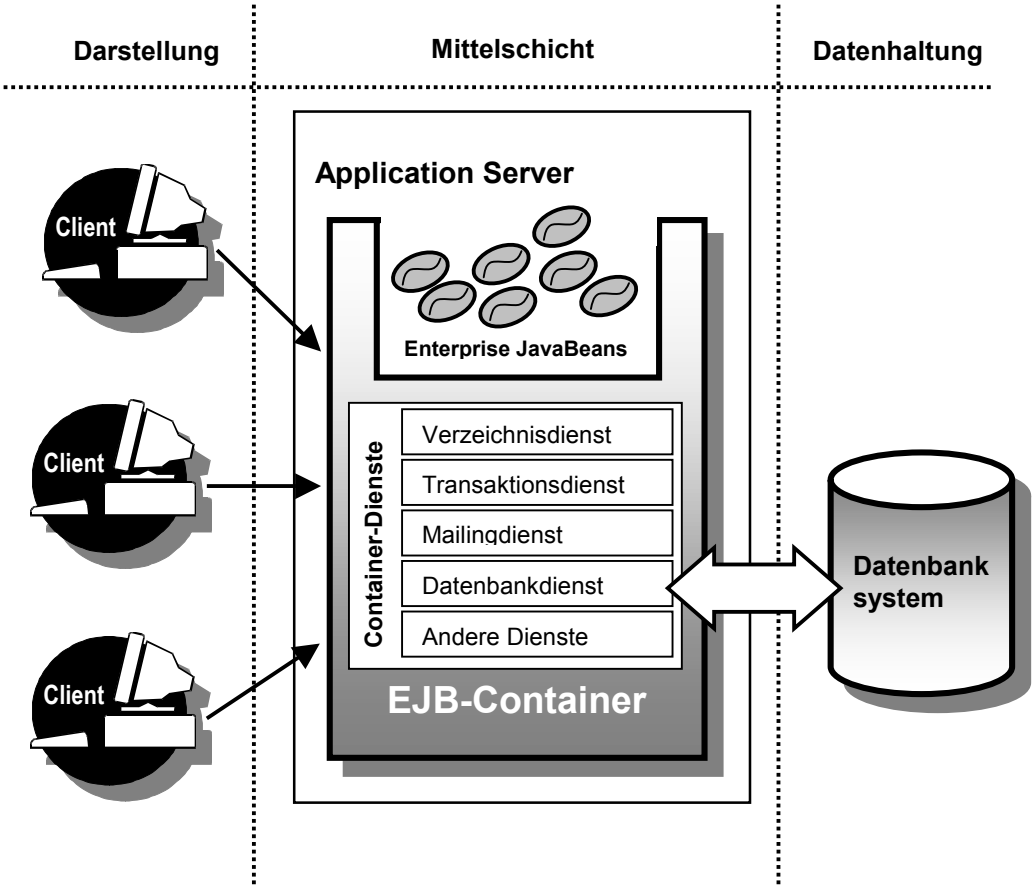


**Abbildung 3-17 Die Enterprise JavaBeans Architektur**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 8.**

An dieser Stelle sei deutlich darauf hingewiesen das die Enterprise JavaBeans Spezifikation nichts mit der ebenfalls von SUN entworfenen Spezifikation der JavaBeans gemein hat. SUN selber definiert die JavaBeans als wiederverwendbare Software-Komponenten, die mit einem Builder-Tool visuell manipuliert werden können. Das bedeutet, bei einer JavaBean handelt es sich um eine Klasse, welche

den durch die Spezifikationen festgelegten Regeln zur Implementierung folgt. Letztendlich heißt dies nur, dass eine JavaBean Klasse über einen parameterlosen Konstruktor und für die Attribute der Bean über entsprechende get<Attribut> und set<Attribut> Methoden verfügen muss. Deshalb können Entwicklungswerkzeuge die Bean nach ihren Attributen untersuchen und deren Werte daraufhin graphisch manipulieren. JavaBeans als solche haben keinerlei verteilten Charakter.
Enterprise JavaBeans dahingegen sind nicht visuelle, serverseitige Komponenten, die ihren Fokus auf die Verteilung und auf transaktionsorientierte Geschäftslogiken legen.

### 3.4.3 Der EJB-Container

Der Container spielt innerhalb der EJB-Spezifikation eine äußerst wichtige Rolle. Er ist für alle standardmäßigen Funktionalitäten (Datenbankzugriffe, Transaktions- und Sicherheitshandling, etc.) zuständig, d.h. für all jene Dienste, um die sich der Entwickler nicht mehr kümmern soll. Er bietet sozusagen eine Schale oder eine Art von Lebensraum, in der die Enterprise-Beans (also die eigentlichen logischen Komponenten) installiert werden können.
Laut der Enterprise JavaBeans Spezifikation 1.1 von SUN muss jeder EJB-Container ein gewisses Mindestmaß an Diensten bereitstellen. Diese Dienste oder APIs[14] sind in Tabelle 3-4 aufgelistet.

| **API** | **Anwendungsbereich** |
|---|---|
| **JSDK 1.1.x oder JSDK 1.x Plattform**[15] | Das eigentliche Java Software Development Kit, welches als Basis für alle Entwicklungen im Java Bereich dient. |
| **EJB Spezifikation 1.1 (J2EE Spezifikation)** | Die eigentliche Enterprise JavaBeans Spezifikation. |
| **JNDI 1.2** | (Java Naming and Directory Interface) Der Verzeichnis Dienst im Java Umfeld. |

[14] Die Abkürzung API steht für **A**pplication **P**rogramming **I**nterface

[15] Die Java-Versionen ab der Version 1.2 werden auch als Java-2-Plattformen bezeichnet

| | |
|---|---|
| **JTA 1.0.1** | (Java Transaction API). Das Interface zur Transaktionssteuerung. |
| **JDBC 2.0** | (Java Database Connectivity) Die Möglichkeit von Java, auf relationale Datenbanken zuzugreifen. |
| **Java Mail 1.1** | Die Java Schnittstelle zum Verschicken von E-Mails |

**Tabelle 3-4 Standarddienste eines EJB-Containers nach EJB 1.1[16]**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 24.**

### 3.4.4 Bestandteile einer Enterprise-Bean

Eine fertige Enterprise JavaBean besteht grundsätzlich aus mehreren Einzelteilen, welche in diesem Abschnitt vorgestellt werden sollen. Sämtliche Bausteine einer solchen Bean werden nach ihrer Fertigstellung in ein Java-Archiv (*.jar File) verpackt und sind somit immer als zusammenhängendes Bündel für die weiteren Entwicklungsschritte verfügbar.

#### *3.4.4.1 Home-Interface*

Das Home-Interface dient zur Kontrolle der Bean-Instanz, d.h. durch dieses Interface können Bean-Instanzen erzeugt, zerstört oder im Falle von Entity-Beans (→ vgl. Abschnitt 3.4.6.2) auch gesucht werden. Diese Schnittstelle ist nach außen für den Client sichtbar.

#### *3.4.4.2 Remote-Interface*

Dieses Interface beinhaltet sämtliche Methoden der Bean, welche nach außen hin für die Clients sichtbar und damit auch nutzbar sein sollen.

Das Remote-Interface bildet in Zusammenhang mit einem Home-Interface die einzige Zugriffsmöglichkeit bzw. Schnittstelle für den Client, um mit einer Bean zu

[16] vgl. [SUN J2EE Spezifikation,2000]

arbeiten. Der Client arbeitet also nie direkt mit einer Bean, sondern kann seine Anfragen nur anhand dieser Interfaces an die gewünschte Bean richten.

#### *3.4.4.3 Die Bean-Klasse*

Diese Klasse stellt letztendlich die eigentliche Bean dar, d.h. hier wird die komplette Geschäftslogik untergebracht. Es handelt sich hierbei um eine Implementierung sämtlicher Methoden, welche durch das Home- und Remote-Interface vorgegeben werden.

#### *3.4.4.4 Primärschlüssel*

Im Falle von Entity-Beans, welche die Fähigkeit besitzen, ihren Zustand dauerhaft in einem externen Speichersystem zu sichern, wird ein Primärschlüssel benötigt, um diese Beans eindeutig identifizieren zu können. Typischerweise werden zu diesem Zweck die entsprechenden Attribute[17], welche diese eindeutige Zuordnung ermöglichen sollen, in einer Primärschlüsselklasse zusammengefasst.

#### *3.4.4.5 Deployment-Deskriptor*

Der Deployment-Deskriptor liegt als Textdatei im XML-Format[18] vor und beinhaltet Informationen für den EJB-Container, wie er eine bestimmte Enterprise-Bean zur Laufzeit behandeln und wie sie mit anderen Komponenten kombiniert werden kann.

Darüber hinaus sollen durch eine solche Datei Merkmale der Bean, wie beispielsweise ihre Struktur oder ihre Abhängigkeiten zu anderen Beans festgehalten werden. Die EJB-Spezifikation sieht vor, dass die Hersteller von EJB-Containern verpflichtet sind, entsprechende Tools bereitzustellen, welche den Entwickler bei der Erstellung eines solchen Deployment-Deskriptors unterstützen sollen.

---

[17] Hierzu kann bereits ein einziges Attribut ausreichen, sofern es eindeutig ist. Theoretisch könnten jedoch alle Attribute der Bean herangezogen werden, um eine Eindeutigkeit zu erzielen.

[18] Abkürzung für E**x**tensible **M**arkup **L**anguage

### 3.4.5 Vorgehensweise zur Erstellung einer Enterprise JavaBean

In diesem Abschnitt soll beschrieben werden, welche Teile einer Enterprise JavaBean der Entwickler erstellen muss und welche Teile automatisch durch die EJB-Umgebung des Herstellers generiert werden.

Zunächst einmal ist es die Aufgabe des Entwicklers, das Home- und das Remote-Interface, die eigentliche Bean-Klasse sowie im Falle einer EntityBean die Primärschlüsselklasse zu erstellen.

Hierbei ist darauf zu achten, dass die Bean-Klasse alle Methoden[19] aus dem Home- und Remote-Interface mit deren exakter Signatur implementiert. Jedoch ist äußerst interessant, dass diese Einbindung nicht durch das Schlüsselwort ‚implements' stattfindet. Der Grund dafür liegt in der Tatsache, dass der Client laut der EJB-Spezifikation niemals direkt auf eine bestimmte Bean zugreifen kann. Um mit einer Bean arbeiten zu können, bedient sich der Client zweier zusätzlicher Objekte, welche die fehlenden Kettenglieder in der Beziehung zwischen dem Client und dem Server darstellen, da sie tatsächlich die vom Entwickler definierten Schnittstellen implementieren und zur Laufzeit die Anfragen der Clients an die Bean-Instanz weiterleiten. Die Implementierungsklasse des Home-Interfaces wird üblicherweise als Home-Objekt oder EJBHome, die des Remote-Interfaces üblicherweise als Remote-Objekt oder EJBObject bezeichnet[20].

Diese beiden Objekte sowie der Deployment-Deskriptor werden durch die Tools des Container-Herstellers aufgrund der Informationen aus der Bean generiert.

### 3.4.6 Typen von Enterprise JavaBeans

Durch die EJB-Spezifikation werden verschiedene Bean-Typen definiert, die jeweils unterschiedliche Aufgaben im System übernehmen. Im folgenden Abschnitt sollen diese Typen näher beleuchtet werden.

#### *3.4.6.1 Session Beans*

Wenn man von einem Session-Bean spricht, spricht man auch oft von einem Geschäftsobjekt. Das bedeutet, dass durch ein solches Session-Bean die Logik

[19] Mit Ausnahme der ‚findByPrimaryKey'-Methode

eines Vorgangs oder einer Funktion abgebildet wird. Typische Aufgaben solcher Session-Beans sind beispielsweise das Erstellen von Statistiken oder das Umrechnen von Währungen. Wie bereits in den vorangegangenen Kapiteln erwähnt, sind solche Beans als eine Art verlängerter Arm des Clients auf dem Server zu verstehen. Grundsätzlich sind Session-Beans Komponenten, die keinen persistenten Zustand kennen, d.h. beim Herunterfahren des Servers oder beim Beenden des Clients, gehen die Daten der Bean verloren. Durch die EJB-Spezifikation werden zwei Arten von Session-Beans definiert, die zustandslosen (engl. stateless) und zustandsbehafteten (engl. stateful) Session-Beans, deren Konzepte an dieser Stelle kurz vorgestellt werden sollen.

#### *3.4.6.1.1 Stateless-Session-Beans*

Stateless Session-Beans verfügen über keinen definierten Zustand, welcher für die Dauer mehrerer Methodenaufrufe gespeichert werden müsste.

Wichtig bei diesem Komponententyp ist, dass eine Instanz eines stateless Session-Beans nur für die Dauer eines einzigen Methodenaufrufs dem Client zur Verfügung gestellt wird. Die Verwaltung durch den EJB-Container findet durch ein sogenanntes Bean-Pooling statt, wobei die Methodenaufrufe der Clients auf einen Pool mit einer festen Anzahl von Bean-Instanzen verteilt werden. Abbildung 3-18 zeigt den Lebenszyklus solcher stateless Session-Beans. Sie können sich lediglich in zwei verschiedenen Zuständen befinden. Entweder sie existieren nicht, d.h. sie wurden noch nicht instanziert oder sie befinden sich im Bean-Pool und sind somit bereit, die Anfragen der Clients zu bearbeiten. Das Erzeugen und Entfernen der Beans ist hierbei ausdrücklich Aufgabe des EJB-Containers. Da alle Beans im Pool über dieselbe Identität verfügen, spielt es aus Sicht des Clients keine Rolle, welche Bean seine Anfrage letztendlich bearbeitet. Es wird irgendeine „freie“ Instanz dem Client verfügbar gemacht, welche nach dem Ausführen der gewünschten Methode wieder zurück in den Pool gelegt wird. Die Anzahl der benötigten Bean-Instanzen ist dadurch in der Regel viel kleiner als die Anzahl der parallelen Client-Sessions.

[20] Für die Implementierung von EJBObject und EJBHome existieren seitens der EJB-Spezifikation keinerlei Vorschriften.

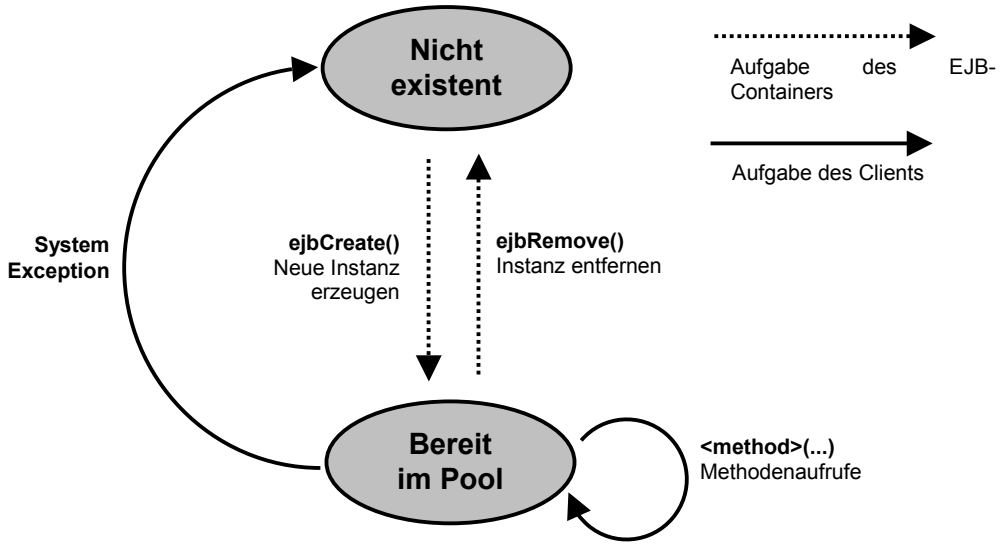


**Abbildung 3-18 Lebenszyklus der stateless Session-Beans**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 71.**

#### *3.4.6.1.2 Stateful-Session-Beans*

Im Gegensatz zu „stateless“ ist durch den Begriff „stateful“ zunächst einmal nur gemeint, dass eine statelful Session-Bean über einen definierten Zustand verfügt.

Ein solcher Zustand wird auch als „Conversational-State“ bezeichnet und beinhaltet alle Attribute einer Session-Bean-Instanz einschließlich der Menge aller anderen Java-Objekte und Beans, die durch diese Instanz referenziert werden.

Die Strategie des EJB-Containers besteht nun darin, diesen Zustand für die Dauer einer Sitzung (Session) aufrechtzuerhalten und die Bean-Instanz für diesen Zeitraum exklusiv einem einzigen Client zur Verfügung zu stellen. Erst wenn dieser Client die Instanz wieder freigibt, d.h. wenn er sie nicht mehr benötigt, geht der Zustand verloren. An dieser Stelle wird bereits deutlich, dass für den Einsatz von zustandsbehafteten Session-Beans serverseitig sehr viele Resourcen beansprucht werden, da für jeden Client auf dem Server eine Bean-Instanz existieren muss. Um dennoch den Resourcenverbrauch auf dem Server im Rahmen zu halten, bedient sich der EJB-Container der Aktivierung- bzw. Passivierung von Session-Beans, was durch die Abbildung 3-19 grafisch veranschaulicht wird.

Durch einen solchen Mechanismus werden momentan nicht mehr benötigte Beans vom Arbeitsspeicher des Servers auf ein externes Speichermedium (z.B. die

Festplatte) ausgelagert[21], was als Passivierung bezeichnet wird und die Bean in den Zustand „Passive Bean“ versetzt (→ vgl. Abbildung 3-19).

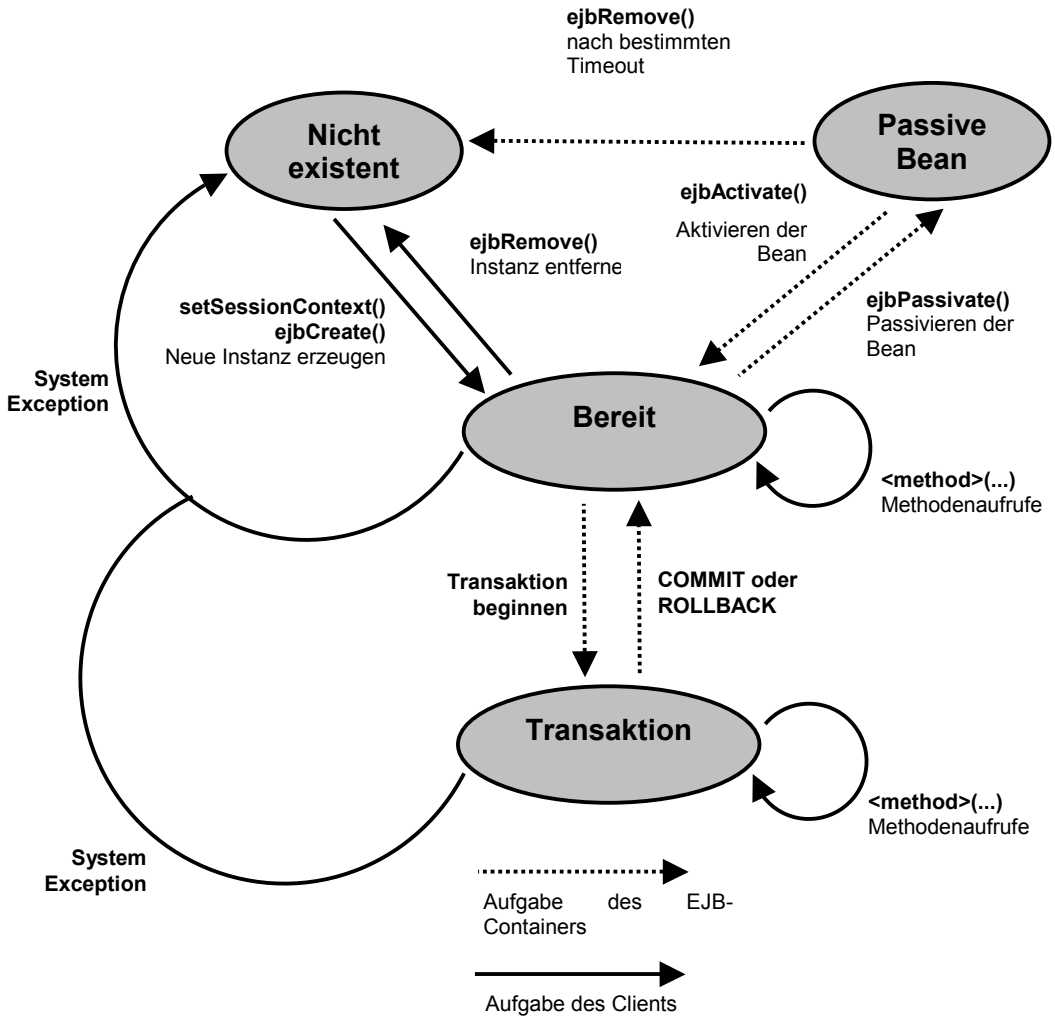


**Abbildung 3-19 Lebenszyklus der statelful Session-Beans**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 74.**

Überschreitet die Verweildauer der Bean im passiven Zustand einen fest definierten Zeitraum (Timeout), so hat der EJB-Container die Möglichkeit, diese Bean-Instanz zu entfernen. Wird diese Bean zu einem späteren Zeitpunkt wieder

[21] Dieser Vorgang wird auch als Serialisierung von Objekten bezeichnet und meint das Umwandeln von Java-Objekten in einen Byte-Strom, welcher dann in eine Datei geschrieben wird. Dieser Mechanismus ist bereits im standardmäßigen Sprachumfang von Java enthalten.

benötigt, so wird sie auf der Basis der Daten des externen Speichermediums rekonstruiert[22] (Aktivierung). Durch eine solche Technik ist es selbst bei einer großen Anzahl paralleler Clients möglich, die Menge der aktiven Bean-Instanzen im EJB-Container in Grenzen zu halten.

Diese Vorgänge der Aktivierung bzw. Passivierung von Beans sind allerdings äußerst rechenintensiv. Aus diesem Grunde werden die stateless Session-Beans auch als leichtgewichtige und die stateful Session-Beans als schwergewichtige Beans bezeichnet.

Ist eine Bean im Zustand „Bereit“ so kann der Client die Bean-Methoden entweder direkt oder im Rahmen einer Transaktion durchführen. Die notwendigen Verwaltungsmechanismen werden durch den EJB-Container bereitgestellt.

#### *3.4.6.2 Entity Beans*

Bei Entity-Beans[23] handelt es sich um Geschäftsobjekte mit eigener Funktionalität. Hauptsächlich spielen diese Beans jedoch die Rolle eines Datenspeichers, welcher in der Lage ist, mehreren Clients den parallelen Zugriff auf transaktionsgesicherte Daten zu ermöglichen. Man kann sagen, dass es die Aufgabe der Entity-Beans ist, Dinge aus der realen Welt, welche dauerhaft existieren sollen, zu modellieren. Aus diesem Grunde verfügt eine Entity-Bean im Gegensatz zu einer Session-Bean über einen persistenten Zustand.

---

[22] Das Rekonstruieren von Objekten wird ist auch als Deserialisierung bekannt und bezeichnet das gegenteilige Verhalten zur Serialisierung.

[23] Entity-Beans sind erst seit der Version 1.1 fester Bestandteil der Enterprise JavaBeans Spezifikation. Vorher war es den Server- und Container-Providern freigestellt, ob sie diesen Komponententyp anbieten oder nicht.

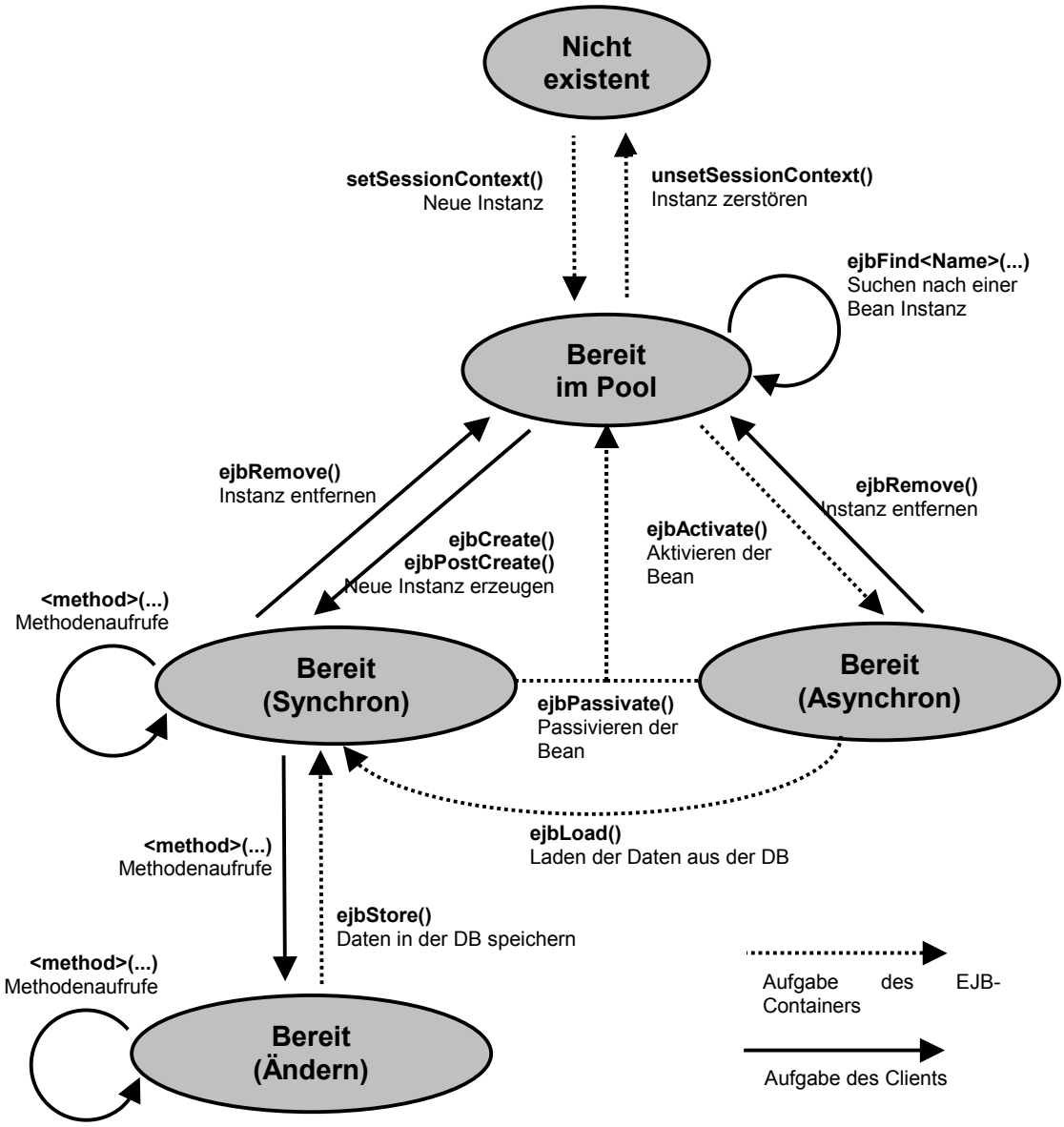


**Abbildung 3-20 Lebenszyklus der Entity-Beans**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 122.**

Abbildung 3-20 zeigt den Lebenszyklus der Entity-Beans, welcher sich in erster Linie von dem der Session-Beans dadurch unterscheidet, dass der Zustand „Bereit“ in drei Unterzustände aufteilt wird, welche durch die Tabelle 3-5 kurz erläutert werden sollen.

| Zustand der Bean | Beschreibung |
|---|---|
| „Bereit (Asynchron)“ | Die Entity-Bean wurde aktiviert. Ihre Daten müssen nicht zwangsläufig mit den Daten aus der Datenbank übereinstimmen. |
| „Bereit (Synchron)“ | Die Bean ist bereit für den lesenden Zugriff auf ihre Attribute, da die Attribute der Bean mit den Daten der Datenbank abgeglichen wurden. |
| „Bereit (Ändern)“ | Auf die Entity-Bean wird schreibend zugegriffen. Das bedeutet, dass ihre Attribute andere Werte aufweisen, die nicht synchron zu den Werten aus der Datenbank sind. Sie werden erst zu einem späteren Zeitpunkt mit der Datenbank abgeglichen. |

**Tabelle 3-5 Bereitschaftszustände von Entity-Beans (→ vgl. Abbildung 3-20)**

**Quelle: In Anlehnung an: Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000. S. 123.**

Zunächst einmal wird eine gewisse Anzahl von Bean Instanzen durch den EJB-Container im Bean-Pool vorrätig gehalten. Basierend auf diesem Bean-Pool existieren die gleichen Aktivierungs- bzw. Passivierungsmechanismen wie bei den Session-Beans.

Das bedeutet, wenn der Client beispielsweise nach einer bestimmten Bean-Instanz sucht, so wird die entsprechende Bean aktiviert. Sie befindet sich daraufhin im Zustand „Bereit (Asynchron)“, da ihre Daten nicht unbedingt mit den aktuellen Informationen aus der Datenbank übereinstimmen müssen.

Erst nach einem Synchronisationsvorgang (ejbLoad) kann davon ausgegangen werden, dass die Attribute der Bean auf dem aktuellsten Stand sind. Die Bean befindet sich zu diesem Zeitpunkt im Zustand „Bereit (Synchron)“, indem der Client auf die Attribute lesend zugreifen kann. Schreibt der Client einen neuen Wert in eines der Attribute, so führt das dazu, dass die Informationen der Bean vorübergehend nicht mit den Informationen aus der Datenbank übereinstimmen, was die Bean automatisch in den Zustand „Bereit (Ändern)“ versetzt. Der EJB-Container trägt nun die Verantwortung dafür, zu einem passenden Zeitpunkt die

Bean mit der Datenbank zu synchronisieren und sie so wieder zurück in den Zustand „Bereit (Synchron)“ zu bringen (ejbStore).
Grundsätzlich werden zwei Typen von Entity-Beans unterschieden. Auf der einen Seite existieren Entity-Beans mit container-managed-persistence, was bedeutet, dass der Container komplett für das Speichern der Bean-Attribute verantwortlich ist, so dass sich der Entwickler darum nicht mehr kümmern muss.
Auf der anderen Seite gibt es Entity-Beans mit bean-managed-persistence, wobei diese Aufgabe durch die Bean selber übernommen wird.
Ein wichtiger Unterschied zwischen Entity-Beans und den bereits vorgestellten Session-Beans liegt in der Behandlung paralleler Clients. Bei einem stateful Session-Bean wird jedem neuen Client auch eine neue Session-Bean-Instanz exklusiv zur Verfügung gestellt, während eine Entity-Bean gleichzeitig von mehreren Clients genutzt werden kann. Die daraus entstehende Vielzahl der Datenzugriffe wird durch die Transaktionskontrolle des EJB-Containers geregelt.
Bei den Attributen einer Entity-Bean wird grundsätzlich zwischen persistenten und transienten Attributen unterschieden. Persistente Attribute sind Attribute, deren Werte wirklich dauerhaft gesichert werden, während transiente Attribute[24] nur solange einen Wert besitzen, wie die Bean sich im Arbeitsspeicher befindet. Bei einer Passivierung gehen ihre Daten verloren.

### 3.4.7 Sichtweisen auf die EJB basierte Softwareentwicklung

Die Architektur der Enterprise JavaBeans kann aus unterschiedlichen Blickwinkeln oder Sichtweisen betrachtet werden, welche im nachfolgenden Abschnitt diskutiert werden sollen.

#### *3.4.7.1 EJBs aus Sicht der Applikationsentwicklung*

Aus Sicht des Entwicklers zeigen sich im Einsatz der Enterprise JavaBeans einige wesentliche Vorteile.
Zunächst einmal wird durch ein derartiges Komponentenmodell die Entwicklungszeit in der Regel verkürzt. Ein wesentlicher Grund dafür ist wohl die

[24] Transiente Attribute gehören zum standardmäßigen Sprachumfang von Java und werden durch das Schlüsselwort „transient“ gekennzeichnet. Wird dieses Schlüsselwort nicht verwendet, so ist das betreffende Attribut automatisch persistent.

Entkopplung zwischen der Schnittstelle und der eigentlichen Implementierung einer Komponente, was eine Voraussetzung für die Wiederverwendbarkeit einer Komponente ist. Zusätzlich dazu muss sich der Entwickler nur auf die fachlichen Anforderungen der Anwendung konzentrieren, da wesentliche Standardaufgaben bereits durch den EJB-Container erledigt werden.

Ein System, welches auf Basis von Enterprise JavaBeans arbeitet, besitzt zudem die Fähigkeit zur Skalierbarkeit. Das wird dadurch erreicht, dass die Beans naturgemäß auf mehreren Servern verteilt werden können, d.h. sie sind durch den zentralen Verzeichnisdienst ortstransparent. Aufgrund dieser Möglichkeit bieten einige Hersteller von Applikationsservern intelligente Lastverteilungsmechanismen in der mittleren Schicht an.

In diesem Zusammenhang trägt die Ortstransparenz der Beans auch zu einer erhöhten Verfügbarkeit bei, da bei einem Ausfall eines Servers ohne große Umstände die Beans auf einem anderen Server installiert werden können. Alle dazu notwendigen Informationen können aus den Deployment Deskriptoren bezogen werden.

Ein weiterer Vorteil des Einsatzes von Enterprise JavaBeans aus Sicht eines Entwicklers ist die Möglichkeit der Integration bestehender Systeme, wie z.B. Mainframe-Systeme. Mit entsprechenden EJB-Containern für solche Systeme, welche mittlerweile von mehreren Herstellern angeboten werden, können neu entwickelte Beans in heterogenen Systemlandschaften dieser Art integriert werden. Somit wird eine schrittweise und somit weniger risikobehaftetere Migration von alten nach moderneren Systemen ermöglicht.

Trotz dieser Vorteile unterliegen die Enterprise JavaBeans zahlreichen Einschränkungen, welche aufgestellt wurden, um eventuelle Konflikte zwischen dem EJB-Container und den Beans zu verhindern. Einige Beispiele hierfür sind nachfolgend aufgelistet:

- Eine Bean darf nicht auf das Dateisystem über das java.io Package zugreifen
- Eine Bean darf nicht versuchen, Threads zu starten, zu stoppen oder zu synchronisieren
- Eine Bean darf nicht eigenständig Socketverbindungen zu anderen Rechnern aufbauen

- Eine Bean darf weder das Schlüsselwort „this“ (also eine Referenz auf sich selber) als Parameter für einen Methodenaufruf verwenden, noch darf sie „this“ an den Aufrufer einer Methode zurückgeben

Sobald jedoch Funktionalitäten zwingend benötigt werden, die den oben gelisteten Restriktionen nicht entsprechen würden, muss die Enterprise JavaBeans Technologie als Mittel der Wahl zumindest in Frage gestellt werden. Darüber sollte sich der Entwickler stets bewusst sein.

#### *3.4.7.2 EJBs aus Sicht des Unternehmens*

Für ein Unternehmen können eine ganze Reihe von Kriterien zu einer Entscheidung für den Einsatz von EJBs führen. Der wahrscheinlich wichtigste Aspekt aus unternehmerischer Sicht ist wohl die Wirtschaftlichkeit. Das Erreichen des primären Ziels der Kostensenkung in diesem Bereich wird durch ein System, welches der EJB Spezifikation entspricht, aktiv unterstützt.
Die Tatsache, dass es sich bei EJBs um ein Komponentenmodell handelt, führt im optimalen Fall zu einem hohen Grad der Wiederverwendbarkeit einzelner Teile. Dadurch kann die Produktion neuer Software immens beschleunigt werden. Es entsteht so ein System, welches trotz kürzerer Entwicklungszyklen eine höhere Stabilität aufweist und somit weniger gewartet werden muss, was wiederum eine starke Senkung der Gesamtkosten bedeutet.
Ein weiterer Vorteil des Einsatzes eines Komponentenmodells aus Sicht eines Unternehmens liegt in der Möglichkeit, dass solche Systeme ohne großen Aufwand und hohes Risiko modifiziert und erweitert werden können. Gerade das ist zweifelsohne ein Aspekt, der zu erheblichen Kosten führen und somit die Situation eines Unternehmens unter Umständen äußerst negativ beeinflussen kann.

# 4 Das „Business Components for Java“ Framework

Nach der Erörterung zweier weit verbreiteter Komponentenmodelle im vorangegangenen Kapitel soll der weitere Verlauf der vorliegenden Arbeit zeigen, wie ein Werkzeug aussehen kann, dass den Entwickler bei der Erstellung solcher Komponenten unterstützt. Oracle, als ein namhaftes Unternehmen im Bereich der Datenbanken, bietet ein solches Werkzeug an.
Es trägt den Namen “Business Components for Java” Framework (BC4J) und wurde vollkommen in die Java Entwicklungsumgebung JDeveloper von Oracle integriert. Bevor allerdings näher auf die Struktur und die Vorgehensweise im Umgang mit diesem Werkzeug eingegangen werden kann, soll der folgende Abschnitt zunächst einmal aufklären, was unter dem Begriff der frameworkbasierten Softwareentwicklung generell zu verstehen ist.

## 4.1 Grundlagen der frameworkbasierten Softwareentwicklung

Bei der frameworkbasierten Softwareentwicklung unterscheidet man grundsätzlich zwischen technischen und fachlichen Frameworks[25].
Technische Frameworks haben einen sehr allgemeinen Aufbau und stellen aus Sicht des Anwendungsentwicklers eine Art von Werkzeugkasten dar, in welchem sich Softwarebausteine mit Basisfunktionalitäten wie z.B. Connection Pooling, Caching, Locking von Datensätzen, etc. befinden. Durch Einsatz dieser Werkzeuge hat der Entwickler die Möglichkeit, eigene Komponenten zu erstellen.
Im Gegensatz dazu stellen fachliche Frameworks eine Ansammlung von vordefinierten branchenspezifischen Geschäftskomponenten für spezielle Einsatzbereiche dar. Beispiele hierfür sind Adressverwaltungen, Lagerverwaltungen oder Buchhaltungskomponenten.

[25] In diesem Zusammenhang spricht man auch oft von horizontalen und vertikalen Frameworks.

## 4.2 Aufbau und Struktur von BC4J

Grundsätzlich handelt es sich bei diesem Framework um ein technisches Framework, welches auf Java- und XML-Technologien basiert.

Die entsprechende Entwicklungsumgebung für solche BC4J-Anwendungen ist vollständig im JDeveloper von Oracle integriert, was ein effektives Arbeiten und ein direktes Testen aller erzeugten Komponenten ermöglicht.

Eine BC4J-Anwendung zeichnet sich durch verschiedene Arten von Komponenten aus, wie in der Abbildung 4-1 zu ersehen ist. Diese einzelnen Bausteine sollen im folgenden Abschnitt kurz vorgestellt werden.

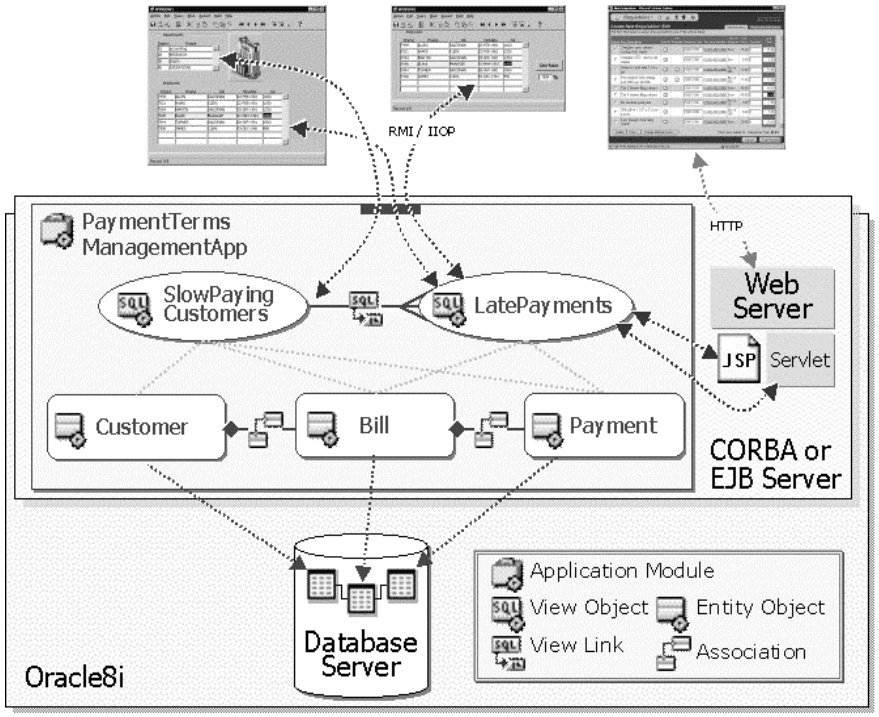


**Abbildung 4-1 Grundlegende Struktur einer BC4J Anwendung**

**Quelle: Oracle JDeveloper 3.2.3 Documentation**

### 4.2.1 Entity Objects

Entity Objects sind Java Objekte und bilden sozusagen das Fundament einer BC4J-Anwendung. Sie sind als der zentrale Ablagepunkt für die zu implementierende Geschäftslogik und als Repräsentanten eines exaktes Abbildes der Tabellen in der Datenbank zu verstehen. Die einzelnen Spalten einer Tabelle füllen hierbei die entsprechenden Attribute des Entity Objects.

Eine einzelne Instanz eines solchen Entity Objects kapselt dabei im objektorientierten Sinne die Informationen einer ganzen Zeile der Ergebnismenge einer SQL-Abfrage auf der Datenbank.

Des weiteren werden sämtliche Persistenz- sowie Caching-Mechanismen auf dieser Ebene durchgeführt. In dieser Hinsicht besteht kein direkter Zusammenhang zwischen Entity Objects und den in Kapitel 3.4 im Rahmen der EJB Konzepte vorgestellten Entity Beans. Beide haben zwar die Aufgabe Persistenzmechanismen bereitzustellen, letztendlich sind es aber in BC4J die Application Modules, welche in Form von Session- oder Entity-Beans in die Mittelschicht eingesetzt (→ vgl. 5.1) werden und nicht die Entity Objects als solche.

BC4J bietet dem Entwickler die Möglichkeit, Entity Objects mit einem Standardverhalten auf der Basis eines bestehenden Datenbankmodells generieren zu lassen (engl.: reverse generation). Liegt dieser Vorgehensweise ein hochwertiges Datenmodell zugrunde, kann dadurch die Entwicklungszeit von Standardkomponenten und die anschließende Testphase immens verkürzt werden.

Die entgegengesetzte Strategie, bei der zunächst die benötigten Entity Objects sowie deren Beziehungen durch den Entwickler erstellt werden, um auf dieser Basis ein Datenbankschema zu generieren, wird ebenfalls von BC4J unterstützt (engl.: forward generation).

Wichtig in diesem Zusammenhang ist, dass die Entity Objects sozusagen die unterste Schicht in einer BC4J-Anwendung darstellen. Das bedeutet für einen Client, dass er keinerlei direkte Zugriffsmöglichkeiten auf diese Objekte besitzt. Er muss sich der sogenannten ViewObjects (→ vgl. Abschnitt 4.2.3) bedienen, um an die Daten der Entity Objects zu gelangen.

Beziehungen zwischen mehreren Entity Objects können über Associations (→ vgl. Abschnitt 4.2.2) hergestellt werden.

### 4.2.2 Associations

Associations stellen bidirektional traversierbare Beziehungen zwischen mehreren Entity Objects her und repräsentieren in der Regel die Foreign-Key-Beziehungen zwischen den Tabellen der Datenbank. Durch sie wird es einem Entity Object ermöglicht, auf die Daten anderer Entity Objects zuzugreifen. Analog zur Datenbankwelt gibt es 1:1 und 1:n Associations. Durch den Einsatz zweier gegensätzlich gerichteter 1:n Associations kann eine n:m Beziehung erzeugt werden. Solche Beziehungen können im Sinne von UML sowohl als Aggregation, bei der ein Objekt ein anderes lediglich referenziert, als auch in Form einer Komposition, bei der ein Objekt einem anderen Objekt gehört, erzeugt werden. Im letzteren Fall ist es unmöglich, ein neues Kindobjekt zu erzeugen, ohne das dazugehörige Elternobjekt vorher angelegt zu haben.

### 4.2.3 View Objects

View Objects steuern den Zugriff auf die Daten der Applikation. In erster Linie besteht ihre Aufgabe darin, eine entsprechende SQL-Abfrage zu kapseln. Solche View Objects basieren in der Regel auf darunter liegenden Entity Objects (→ vgl. Abbildung 4-2).

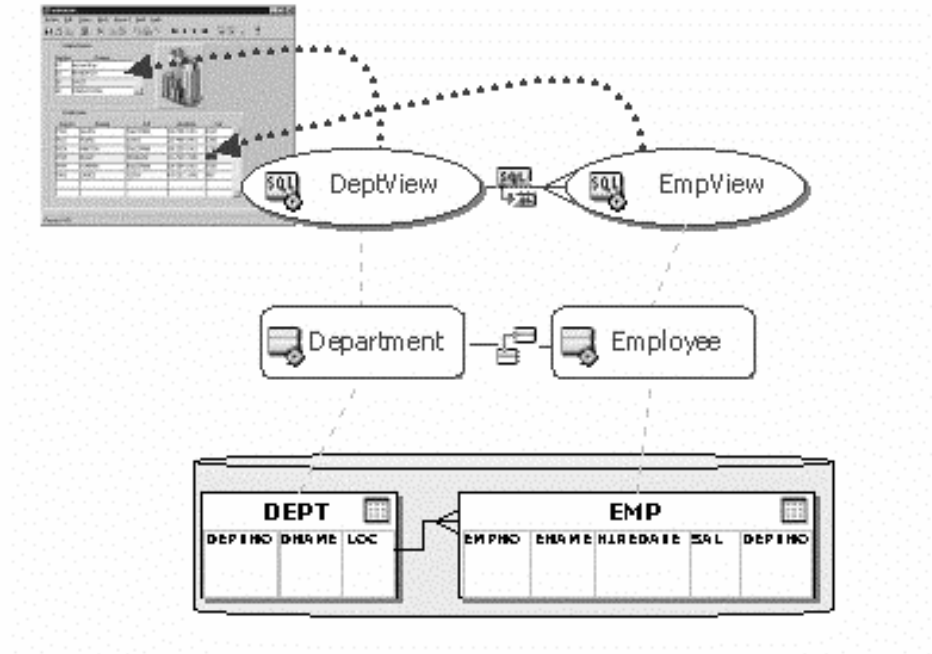


**Abbildung 4-2 Aufbau von View Objects auf der Basis von Entity Objects**

**Quelle: Oracle JDeveloper 3.2.3 Documentation**

Das bedeutet, dass nach der Ausführung des SQL-Statements entsprechend der Ergebnismenge zeilenweise Entity Objects durch das View Object erzeugt werden. Diese einmalig instanzierten Objekte werden daraufhin in einem Cache für eine eventuelle Wiederverwendung vorrätig gehalten. Durch eine solche Strategie sollen die oftmals hinsichtlich der Performance teuren Datenbankzugriffe vermieden werden.

Nach der Ausführung der Abfrage in der Datenbank stellt ein View Objekt dem Client alle Methoden zur Verfügung, um auf die Daten zugreifen sowie in der Ergebnismenge navigieren zu können. Aus Sicht dieses Clients entsteht der Eindruck, als könnten Datenmengen direkt durch das View Object manipuliert werden. In Wirklichkeit werden sämtliche clientseitigen Anfragen lediglich an die darunter liegenden Entity Objects und deren gekapselte Geschäftlogiken delegiert.

Ein View Object kann auf mehreren Entity Objects basieren, d.h. es stellt deren Attribute zur Verfügung. In solch einem Fall kann der Entwicklers höchstens ein Entity Object definieren, dessen Attribute geändert werden können. Auf alle anderen Entity Objects kann dann in der Konsequenz nur lesend zugegriffen werden, d.h. es sind im Sinne von SQL keine INSERTs, UPDATEs oder DELETEs möglich.

Unabhängig von den darunter liegenden Associations zwischen den Entity Objects werden View Objects durch sogenannte View Links miteinander verbunden.

### 4.2.4 View Links

Die sogenannten View Links definieren Beziehungen zwischen jeweils zwei View Objects. Sie können lediglich ausgehend vom Master in Richtung des Details traversiert werden und nicht entgegengesetzt, d.h. sie sind unidirektional, was sie von den Associations unterscheidet. In derselben Art und Weise wie View Objects auf Entity Objects basieren, können auch View Links auf Associations basieren (→ vgl. Abbildung 4-2 ).

### 4.2.5 Application Modules

Application Modules sind als eine Art von Schale zu verstehen, welche View Objects und View Links enthalten können. Solch ein Modul verkörpert einen bestimmten Anwendungsfall und macht diesen in Form einer entsprechenden

Schnittstelle für den Client verfügbar. Darüber hinaus können eigene Dienste als „Custom Methods“ in Application Modules untergebracht werden, welche ebenfalls direkt clientseitig genutzt werden können. In diesem Zusammenhang werden, sofern ein Application Module mit solchen „Custom Methods“ in einer EJB- oder CORBA-Umgebung betrieben werden soll, alle notwendigen Stubs und Skeletons (→ vgl. Abschnitt 2.4) seitens des JDevelopers generiert[26].
Darüber hinaus regeln Application Modules die Verwaltung von Transaktionen sowie das Locking von Datensätzen.

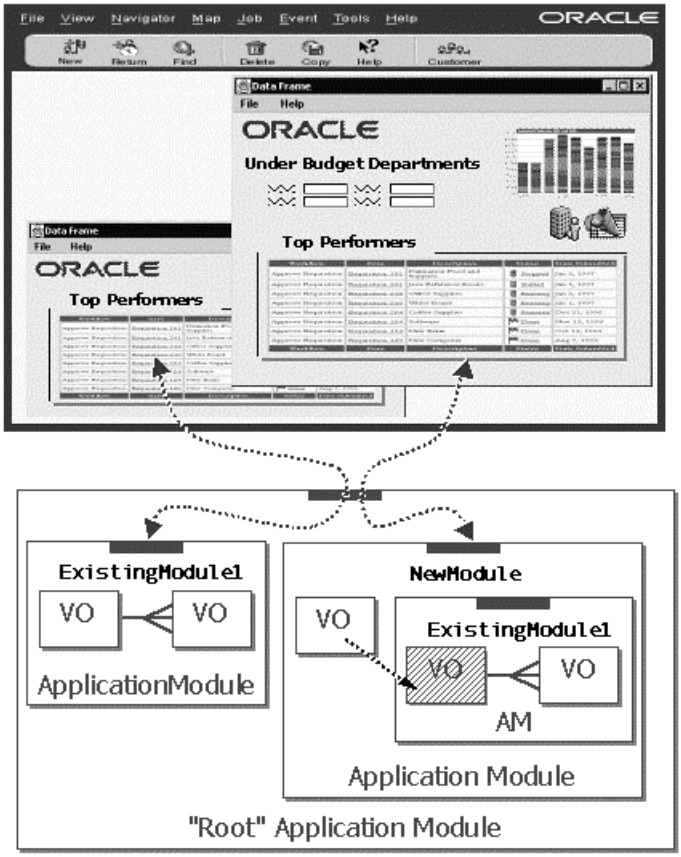


**Abbildung 4-3 Anwendungserweiterung durch verschachtelte ApplicationModules**
**Quelle: Oracle JDeveloper 3.2.3 Documentation**

[26] Der Einsatz von BC4J Komponenten im EJB- bzw. CORBA-Umfeld wird im Abschnitt 5 dieser Arbeit noch genauer beschrieben

Ein weiteres wichtiges Merkmal von Application Modules ist ihre Fähigkeit verschachtelte Strukturen zu bilden. Das bedeutet, dass ein Modul andere Module enthalten kann[27], wobei das „Root“ Application Module (d.h. das Modul, welches selber keinem anderen Modul mehr angehört) die notwendige Datenbankverbindung kapselt und diese ihren Kindern zur Verfügung stellt.
Die Vorteile solcher Möglichkeiten werden erst dann deutlich, wenn eine bestehende BC4J Anwendung erweitert werden soll (→ vgl. Abbildung 4-3). In diesem Fall kann ein neues Application Module mit der neuen Funktionalität erstellt werden, in welches ältere Modules ohne große Modifikationen einfach integriert werden können. Sowohl die alte als auch die neue Anwendung werden von einem „Root“ Application Module umgeben. Dieses Modul stellt dann wiederum die zentrale Schnittstelle sowohl zum Client als auch zur Datenbank dar.

### 4.2.6 Caching Mechanismen

Komponenten, die basierend auf dem BC4J-Framework entwickelt wurden, werden zur Laufzeit der Anwendung grundsätzlich in einem Cache vorrätig gehalten.
Eine solche Strategie ist in vielerlei Hinsicht äußerst sinnvoll, da die Daten der Objekte direkt im Arbeitsspeicher geändert und erst zu dem Zeitpunkt wieder in der Datenbank gesichert werden, wenn der Benutzer die Daten explizit speichern möchte. Das führt zu einer immensen Reduzierung der sonst aus Sicht der Performance teuren Datenbankzugriffe.
Ein weiterer Vorteil dieser Vorgehensweise liegt darin, dass der Cache den Zustand der Business Components verwaltet. Bei diesen Zuständen kann es sich um neue Objekte, gelöschte Objekte oder geänderte Objekte handeln, was wiederum Konsequenzen bezüglich der entsprechenden SQL-Statements nach sich zieht. Der Entwickler muss sich jedoch nicht selber darum kümmern, ob ein INSERT eines neuen Objektes oder ein UPDATE eines bereits bestehenden Objektes durchgeführt werden muss, da ihm diese Entscheidung durch das BC4J-Caching abgenommen wird.
Das BC4J-Framework unterscheidet grundsätzlich zwischen zwei Arten des Cachings, dem Entity- und dem View Object-Cache. Für jedes Entity bzw. View

[27] engl.: Nested Application Modules

Object existiert jeweils genau ein eigener Cache. Wie diese beiden Mechanismen miteinander in Verbindung stehen, soll beispielhaft durch die Abbildung 4-4 verdeutlicht werden.

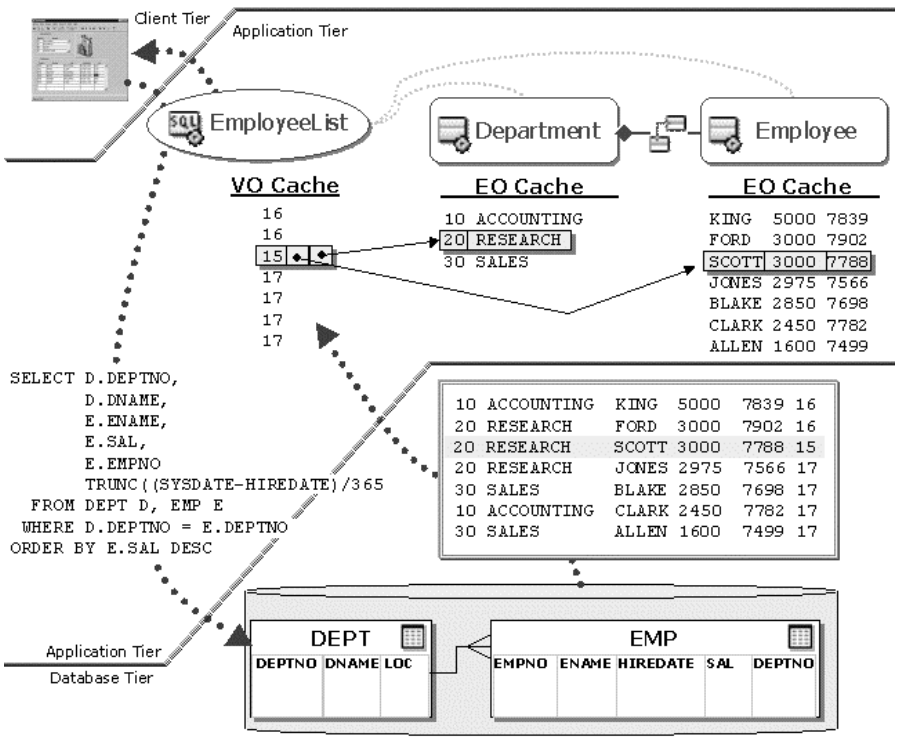


**Abbildung 4-4 BC4J Caching Mechanismen**
**Quelle: Oracle JDeveloper 3.2.3 Documentation**

Ein View Object kapselt typischerweise die auszuführende SQL-Abfrage, die zur Datenbank geschickt wird. Im obigen Beispiel geht es um eine Abfrage, welche verschiedene Abteilungen eines Unternehmens (DEPT) mit ihren jeweiligen Mitarbeitern (EMP) in Verbindung bringt.

Diejenigen Attribute der Ergebnismenge, welche sich durch Attribute der darunter liegenden Entity Objects abbilden lassen, werden in Form neuer Entity Instanzen in einem oder mehreren der verfügbaren Entity-Caches gehalten. Der View Object Cache enthält lediglich eine Referenz auf diese Objekte.

Die Daten der übriggebliebenen Attribute der Ergebnismenge, welche nicht durch Entity Objects erfasst werden können, werden direkt als Werte in der entsprechenden Zeile des View Object-Caches gespeichert. Üblicherweise wird der

Cache jedoch nicht sofort komplett mit allen Zeilen der Ergebnismenge gefüllt, obwohl dies durchaus möglich wäre. Der Cache wird statt dessen „on-demand“, d.h. auf Anfrage aufgebaut. Eine solche Anfrage tritt immer dann auf, wenn der Benutzer des Systems, die Daten der Ergebnismenge durchläuft.

Sind die Daten einmal in den Entity Object-Caches untergebracht, können die View Objects auf diese Daten zugreifen, anstatt eine neue Datenbankabfrage absetzen zu müssen. In diesem Zusammenhang ist interessant, dass sich dabei mehrere View Objects einen Entity Object-Cache teilen können. Das hat zur Konsequenz, dass die Daten aller View Objects, welche auf einem gemeinsamen Entity Object basieren, automatisch synchronisiert werden.

Für den Entwickler ist nach der Fertigstellung der Business Components und der darin enthaltenen Geschäftslogik die Arbeit an der mittleren Schicht beendet. Im nächsten Schritt ist es seine Aufgabe, eine Client-Anwendung zu erstellen, welche diese Komponenten nutzt und sie in Form einer graphischen Oberfläche dem Benutzer gegenüber darstellt. Dies soll im folgenden Abschnitt thematisiert werden.

### 4.2.7 BC4J-Clients in Form einer Swing-Anwendung

Clients in Form von Applets oder in Form sogenannter Standalone-Anwendungen[28] benutzen typischerweise die Swing-Klassenbibliothek[29] von SUN, um komplexe graphische Benutzeroberflächen darzustellen.

Es ist an dieser Stelle durchaus denkbar, eine solche clientseitige Anwendung von Grund auf selber zu entwickeln. Durch den Einsatz eines geeigneten Frameworks kann diese Aufgabe jedoch weitaus effizienter erledigt werden.

Oracle bietet im Rahmen von BC4J ein eigenständiges Framework zur Erstellung solcher Clients basierend auf fertigen BC4J Anwendungen an. Dieses Framework ist unter dem Namen „Data-Aware Control Framework“ (DACF) bekannt. Die Struktur eines solchen DACF-Clients wird durch die Abbildung 4-5 skizziert.

---

[28] Standalone-Anwendungen sind Anwendungen, die im Gegensatz zu Applets nicht in einem Browser sondern eigenständig (also „standalone“) in einer Java Virtual Machine ablaufen.

[29] Die Swing-Klassenbibliothek ist ein Framework von SUN zur Erstellung von plattformunabhängigen graphischen Benutzeroberflächen, welche auch als Graphical User Interfaces (kurz GUIs) bezeichnet werden.

Das DAC Framework etabliert auf einfache Art und Weise die Verbindung der Business Components mit den graphischen Komponenten der Benutzeroberfläche. Zu diesem Zweck besteht der Client aus zwei Hauptbestandteilen, den InfoProducern und den InfoConsumern.

Die InfoProducer haben die Aufgabe, die Daten aus den BC4J Komponenten auszulesen und diese den InfoConsumern bereitzustellen. Dabei ist anzumerken, dass die Struktur der InfoProducer der Struktur der eigentlichen Business Components sehr ähnlich ist.

Die einzig reale Verbindung, welche zwischen ihnen besteht, ist die Brücke von einem Application Module in der mittleren Schicht zum sogenannten SessionInfo-Objekt. Diese Objekte kapseln die notwendigen Informationen, um seitens des Clients auf ein bestimmtes Application Module zugreifen zu können.

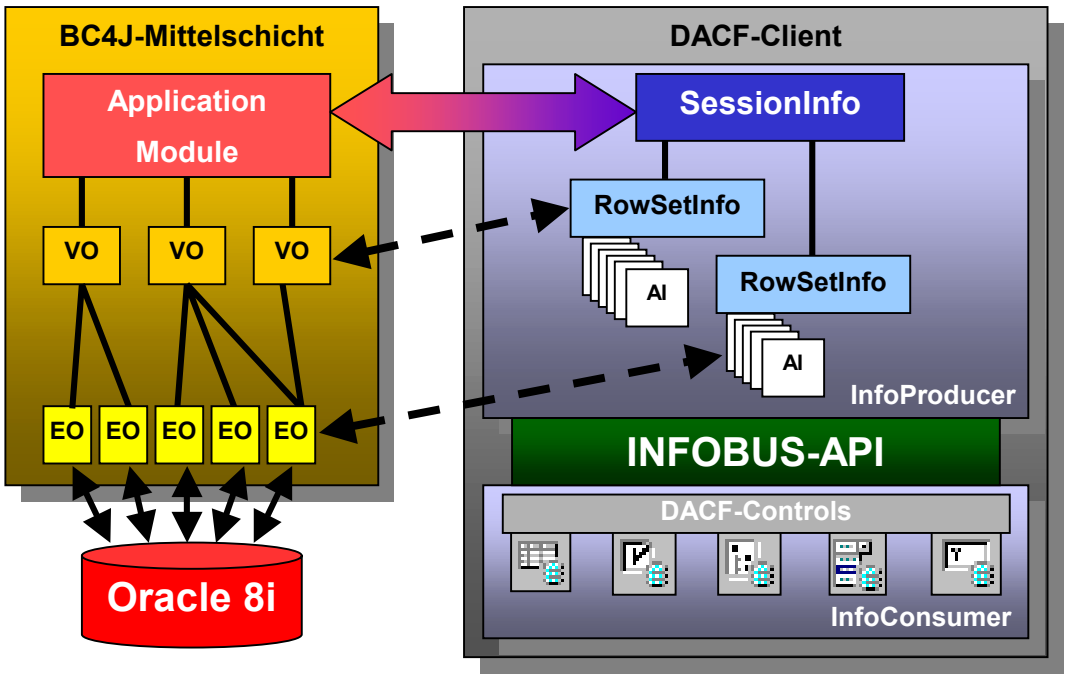


**Abbildung 4-5 Die Struktur eines DACF-Clients**

Mit diesen SessionInfo-Objekten sind die sogenannten RowSetInfo-Objekte verknüpft, welche dem Client die Möglichkeit bieten, durch die Datensätze der Ergebnismenge eines View Objects zu navigieren. Es besteht sozusagen eine virtuelle Verbindung zwischen den View Objects und den RowSetInfos (→ vgl. Abbildung 4-5). Eine ähnliche Affinität besteht zwischen den AttributeInfo-

Objekten (AIs) welche mit einem bestimmten RowSetInfo-Objekte verbunden sind und den Entity Objects der BC4J-Mittelschicht. Hier besteht beider Aufgabe darin, die Daten der einzelnen Attribute zu speichern.
Sind die Daten einmal durch die InfoProducer gekapselt, werden die Informationen über den Infobus den InfoConsumern zur Verfügung gestellt. Der Infobus ist als ein Datenbus zu verstehen, dessen API als separates Produkt bei SUN frei zur Verfügung steht. Durch den Einsatz einer solchen Technologie werden die Datenmodelle und die verschiedenen Sichten auf diese Modelle ständig synchronisiert[30]. Diese Synchronisation ist bidirektional, d.h. wenn sich die Daten in der BC4J-Mittelschicht bzw. den dazugehörigen InfoProducern ändern, werden die graphischen Komponenten der Oberfläche automatisch aktualisiert. Im Gegensatz dazu werden die Datenquellen aktualisiert, wenn der Benutzer ein graphisches Element verwendet, um einen Datensatz zu ändern.
Das führt dazu, dass alle InfoConsumer, welche auf ein und demselben RowSetInfo Objekt basieren, zu jedem Zeitpunkt automatisch synchronisierte Daten anzeigen, was aus Entwicklersicht eine immense Aufwandsersparnis darstellt.
An dieser Stelle soll die Infobus Technologie nicht näher beschrieben werden. Stattdessen sei hier auf die entsprechenden Quellen verwiesen[31].
Eine genauere Betrachtung der InfoConsumer zeigt, dass es sich dabei um Komponenten handelt, welche im technischen Sinne als eine direkte Erweiterung der jeweiligen Swing-Komponenten von SUN durch Oracle zu verstehen sind.
Diese Erweiterungen umfassen hauptsächlich die Anbindung an den InfoBus sowie die Implementierung des dazugehörigen Event-Handlings. Ihre Aufgabe besteht in der Darstellung und Manipulation der Daten, welche durch die InfoProducer über den Infobus zur Verfügung gestellt werden.

[30] Diese Strategie wird in der Literatur als Model-View-Controller Konzept (MVC) beschrieben.

[31] vgl. http://java.sun.com/products/javabeans/infobus/

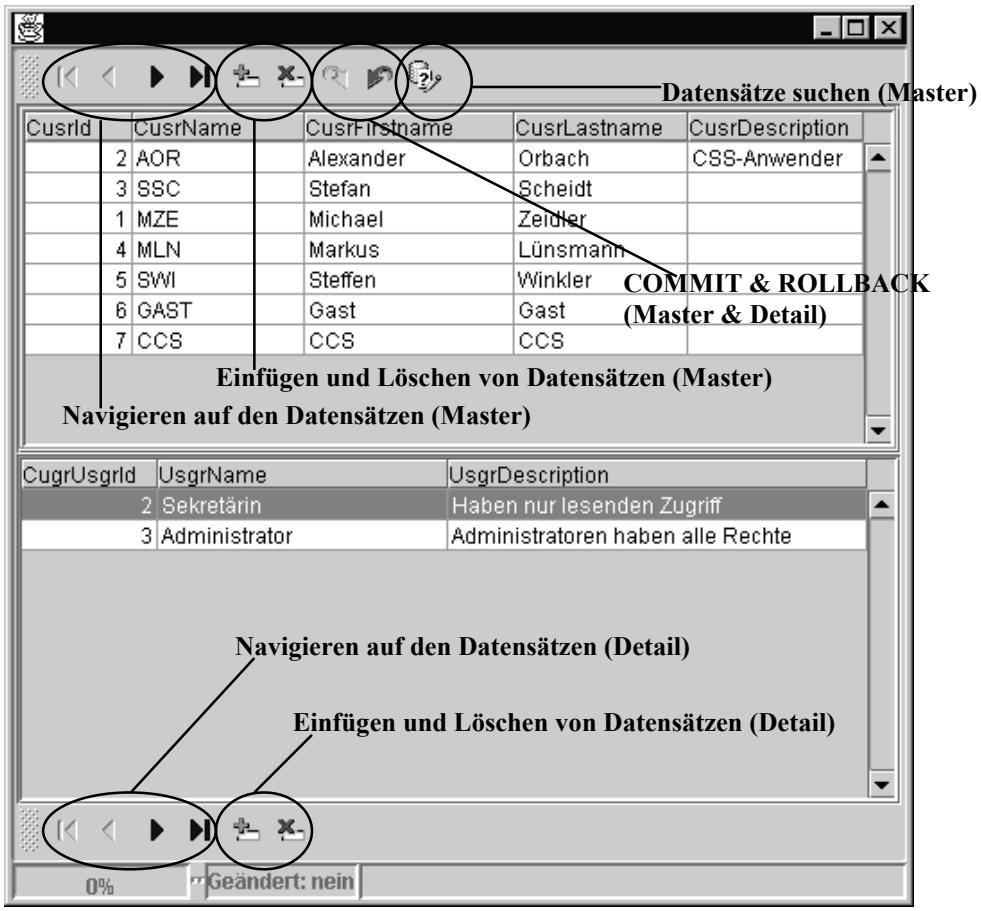


**Abbildung 4-6 Beispiel einer generierten Master-Detail Maske mit DACF**
**Quelle: Oracle JDeveloper 3.2.3**

Abbildung 4-6 zeigt beispielhaft, wie solch eine DACF-Maske aussehen kann. Diese Maske basiert auf zwei View Objects, welche in einer Master-Detail Beziehung stehen. Sie wurde vollständig durch das BC4J Framework generiert. Fachlich geht es in diesem Beispiel um Benutzer (Master), welchen verschiedene Benutzergruppen (Detail) zugeordnet werden können. Im oberen Teil der Maske kann nun durch die Master Datensätze navigiert werden, wobei sich die Detail Datensätze über den obig beschriebenen Mechanismus daraufhin aktualisieren. Zu diesem Zweck wird letztendlich jedes mal ein entsprechendes SELECT Statement gegen die Datenbank abgesetzt. Standardmäßig werden bei solchen DACF Masken die Funktionalitäten des Navigierens auf Datensätzen, des Einfügens und Löschens von Datensätzen sowie das Suchen von Datensätzen durch einen separaten

Suchdialog generiert. Darüber hinaus hat der Benutzer immer die Möglichkeit, ein COMMIT bzw. ROLLBACK durchzuführen.

### 4.2.8 BC4J-Clients in Form einer JavaServer Page Anwendung

Die JavaServer Page Technologie (JSP) wurde von SUN spezifiziert, um einen einfachen Weg zu schaffen, dynamischen Inhalt von Webseiten generieren zu können. Typischerweise wird dieser dynamische Inhalt von einem Web Server erstellt und daraufhin an den Browser des Clients geschickt.

Diese Technologie, welche als eine Erweiterung der Servlet Technologie angesehen werden kann, bietet dem Entwickler die Möglichkeit, in den gewöhnlichen Quellcode einer HTML Seite Java Quellcode in Form sogenannter Java Scriptlets zu integrieren. Durch solche Scriptlets kann auf externe Java Komponenten (z.B. Datenbanken) zugegriffen werden.

Somit eignen sich JavaServer Pages hervorragend als Web Clients, die auf die Komponenten einer mittleren Schicht und somit auf die Geschäftslogiken zugreifen. In diesem Zusammenhang können alle Möglichkeiten von HTML auch im JSP Bereich voll ausgenutzt werden.

Technisch gesehen wird im Falle einer clientseitigen HTTP Anfrage an den Web Server, eine bestimmte Java ServerPage in ein Servlet transformiert und dann auf dem WebServer gestartet. Abbildung 4-7 zeigt beispielhaft den Quellcode einer Java ServerPage. Die beschriebenen Scriptlets sind hier hervorgehoben dargestellt.

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-
)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

**Abbildung 4-7 Beispielcode einer Java ServerPage**

**Quelle: Oracle Corporation, Oracle JavaServer Pages Developer's Guide and Reference Release 8.1.7, 2000.**

Wie diese JSP letztendlich im Browser aussieht, zeigt die Abbildung 4-8. In der Regel wird der Entwickler dieser Seiten versuchen, den Umfang solcher Scriptlets so klein wie möglich zu halten.

Ziel sollte es sein, die gesamte Logik in den Komponenten der mittleren Schicht unterzubringen. Die Aufgabe einer JavaServer Page liegt dann nur noch in der Nutzung, d.h. im Aufruf der Komponenten.

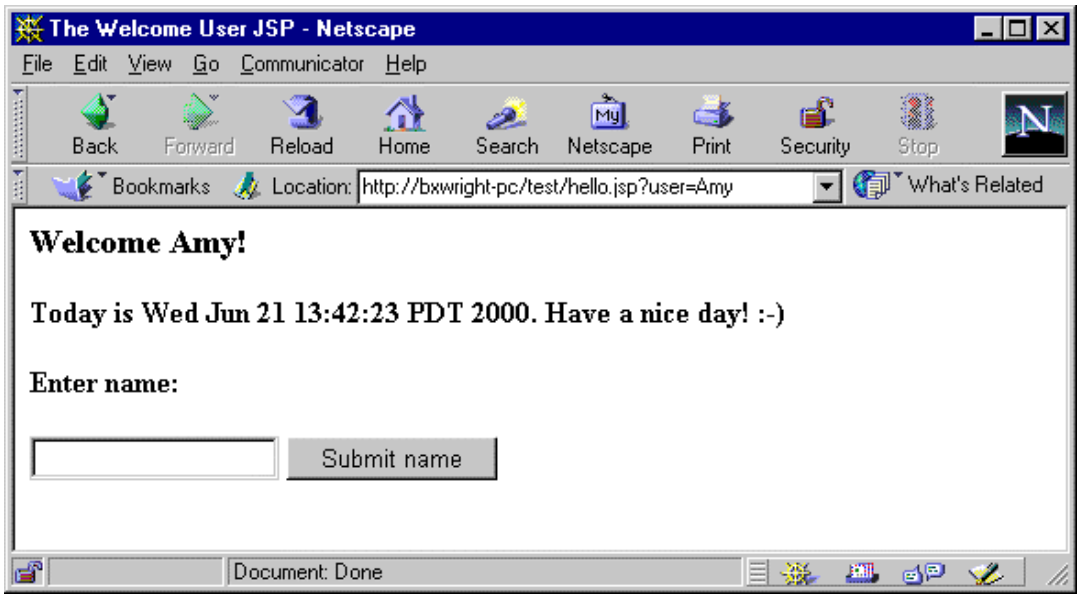


**Abbildung 4-8 Java ServerPage im Browser**

**Quelle: Oracle Corporation, Oracle JavaServer Pages Developer's Guide and Reference Release 8.1.7, 2000.**

Genau an dieser Stelle setzt das BC4J Framework von Oracle an. Um die Entwicklung eines Clients auf der Basis der JavaServer Page Technologie in Verbindung mit dem BC4J Framework zu beschleunigen, bietet Oracle dem Entwickler eine breite Palette bereits vorgefertigter Komponenten an.

Diese Komponenten sind unter der Bezeichnung DataWebBeans bekannt und haben die Fähigkeit, auf die durch BC4J entworfenen Bestandteile eines Geschäftssystems in der mittleren Schicht zugreifen zu können. Solche Beans sind in der Lage aufgrund der dort liegenden Daten zur Laufzeit dynamisch eine Antwort in Form eines HTML-Stroms zu erzeugen.

Ähnlich wie bei den Swing Clients ist BC4J auch hier in der Lage, JSP Seiten für Standardfunktionalitäten wie das Anzeigen, Editieren und Suchen von Datensätzen

komplett zu generieren. Der Entwickler kann diesen erzeugten Code dann gegebenenfalls durch eigens programmierte DataWebBeans ergänzen. Abbildung 4-9 zeigt eine solche durch BC4J generierte JavaServer Page, die eine Master-Detail Beziehung zweier Komponenten darstellt.

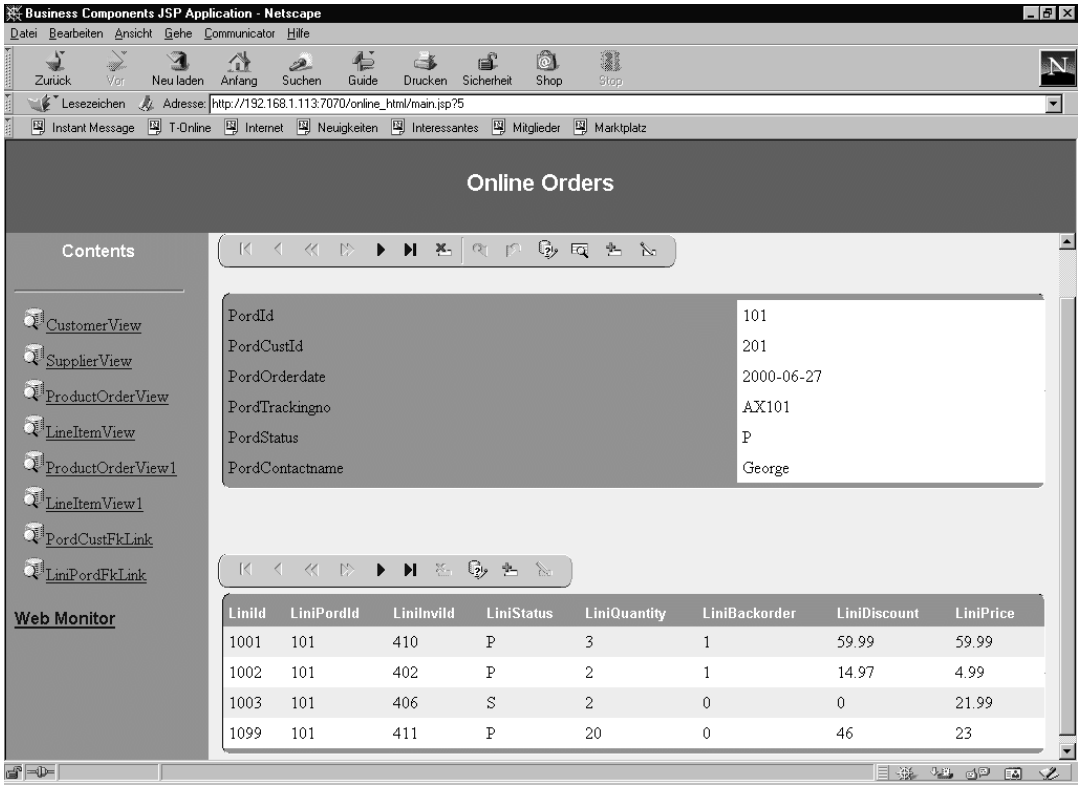


**Abbildung 4-9 Eine durch BC4J erzeugte JSP-Anwendung**

# 5 Das Deployment von BC4J-Komponenten

Der folgende Teil beschäftigt sich mit dem Thema des Deployments. Dadurch wird die fehlende Verbindung zwischen den in den ersten Kapiteln dieser Arbeit vorgestellten Komponentenmodellen CORBA bzw. Enterprise JavaBeans und dem BC4J Framework als Werkzeug zur Erstellung solcher Komponenten geschaffen.
Um die Vorgänge nachvollziehbarer machen zu können, wird in diesem Kapitel eine Beispielanwendung mit dem Namen „EDOS“ eingeführt. Dadurch wird die Erstellung der Komponenten nachvollziehbar gemacht, welche später in der mittleren Schicht eines Verteilten Systems arbeiten sollen.

## 5.1 Erläuterung des Begriffs „Deployment“

Es erweist sich als äußerst schwierig, eine genaue Definition dieses Begriffs in der Form, die im Rahmen dieser Arbeit von Bedeutung ist, in der Literatur zu finden.
Zunächst einmal bedeutet der Begriff „Deployment“ direkt übersetzt „Einsetzung“ und genau das beschreibt bereits, was damit gemeint ist.
Im Sinne von BC4J entwickelt und testet ein Programmierer die Anwendung lokal. Dieser Softwareentwicklungsprozess wird solange fortgesetzt, bis dass die Applikation fehlerfrei läuft. Erst dann werden die für die mittlere Schicht relevanten Teile, d.h. die erzeugten Application Modules und die darin enthaltenen Business Components der Anwendung, herausgenommen und in Form von Enterprise JavaBeans oder OMG konformen CORBA Komponenten in eine verteilte Umgebung „eingesetzt“, also „deployed“.
Insofern lässt sich der Begriff des „Deployments“ folgendermaßen umschreiben:

*- Mit dem Begriff „Deployment“ im Sinne der Entwicklung verteilter Software unter Einsatz von BC4J als Werkzeug ist ein Einspielvorgang bereits lokal erstellter und getesteter Komponenten in eine verteilte Systemlandschaft gemeint. -*

## 5.2 Beschreibung der Beispielanwendung „EDOS“

Die Vorgänge beim Deployment können wesentlich einfacher beschrieben werden, wenn man sie aufgrund eines praktischen Beispiels betrachtet. Da es das Ziel dieser Arbeit ist, die Vor- und Nachteile des Verteilens einer fertigen BC4J-Anwendung zu zeigen, darf dieses Beispiel nicht zu komplex sein.

Als Grundlage für die Ausführungen in diesem Kapitel soll eine exemplarische Anwendung namens „EDOS“ („Easy Deployable Online Shop“) betrachtet werden, durch welche es in erster Linie möglich sein wird, Kunden und ihre Bestellungen aller Art zu verwalten.

Diese Bestellungen bestehen jeweils aus mehreren Einzelposten. Hinter jedem der Posten verbirgt sich ein konkretes Produkt, welches bestellt werden soll. Jedes der Produkte verfügt über eine Kategorie und über einen Zulieferer, über den der Shop dieses Produkt bezieht.

## 5.3 Technische Umsetzung von „EDOS“ auf der Basis von BC4J

### 5.3.1 Entwurf der BC4J-Komponenten

Das dieser Anwendung zugrundeliegende Datenmodell ist in der Abbildung 5-1 dargestellt.

Dieses Datenmodell bildet die Basis für die Entwicklung der Business Components, d.h. es wird ein Reverse Engineering betrieben, um an die Java-Klassen zu gelangen. In einem ersten Schritt werden daher auch die Entity- und die View-Objects sowie die Beziehungen zwischen ihnen durch Associations bzw. View Links erzeugt. Die Struktur der BC4J-Anwendung soll so aussehen, dass für jede Tabelle in der Datenbank eine Klasse erzeugt wird, welche das Entity Object modelliert und eine Klasse, die darauf basierend die Implementierung des View Objects darstellt.

Zwischen den Entity Objects bestehen Associations, welche als ein direktes Abbild der Foreign Key Contraints aus der Datenbank verstanden werden können. Analog zu jeder Association soll genau ein View Link existieren.

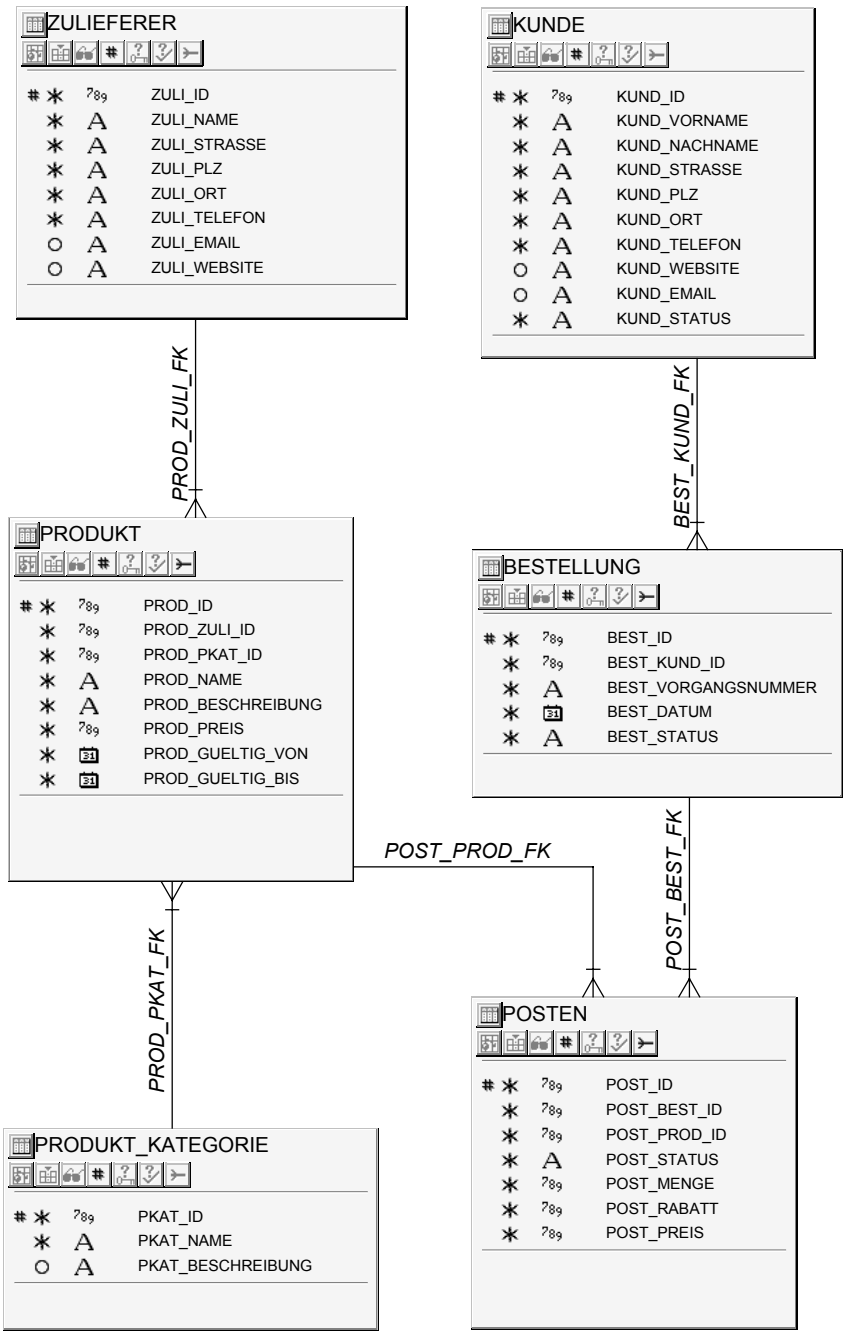


**Abbildung 5-1 Das Datenmodell für die Beispielanwendung „EDOS“**

Der nun folgende Schritt sieht die Erzeugung eines Application Modules als zentrales Element für die Beispielanwendung vor. Das Application Module, welches den Namen „EDOSModule“ tragen wird, hat die Aufgabe, die zuvor erzeugten View Object und die damit zusammenhängenden Entity Objects für die späteren Clients verfügbar zu machen.

Es ist bei der Erzeugung eines Application Modules durchaus möglich, ein und dasselbe View Object dem Application Module mehrmals hinzuzufügen, wodurch es daraufhin in Zusammenhang mit unterschiedlichen Anwendungsfällen benutzt werden kann.

Doch zunächst sollte geklärt werden, welche Anwendungsfälle zur Realisierung von „EDOS“ überhaupt benötigt werden und welche View Objects dazu in das Application Module einzubinden sind. Den derzeitige Vorrat an View Objects und deren Struktur soll durch die Abbildung 5-2 verdeutlicht werden.

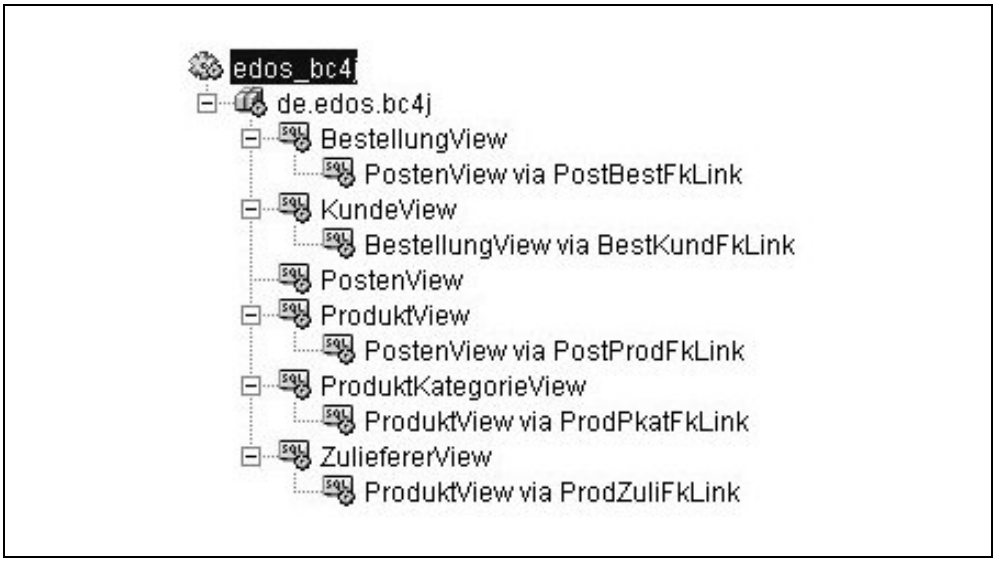


**Abbildung 5-2 Die verfügbaren View Objects der Beispielanwendung „EDOS“**

Auf der ersten Strukturebene unterhalb des Packages „de.edos.bc4j“ des oben gezeigten Baumes sind die View Objects aufgeführt, die in keinerlei Abhängigkeit zu anderen Objekten stehen. In der zweiten hierarchischen Ebene sind die View Objects zu erkennen, welche durch den angegebenen View Link in einem Abhängigkeitsverhältnis zu den Objekten der ersten Ebene stehen.

Zweifellos sind die Tabellen KUNDE, ZULIEFERER und PRODUKT_KATEGORIE wichtige Tabellen, welche Schlüsselinformationen der Anwendung beinhalten. Somit sollten auch die View Objects „KundeView“, „ZuliefererView“ und „ProduktKategorieView“ in das Application Module eingehen. Darüber hinaus sollten alle vorhandenen Master-Detail Beziehungen in die Struktur des Application Modules aufgenommen werden. In diesem

Zusammenhang spielt das View Object „KundeView“ eine Sonderrolle, da es sich hierbei in Kombination mit „BestellungView“ und „PostenView“ um eine Master-Detail-Detail Beziehung handelt. Die letztendliche Struktur des fertigen Application Modules zeigt die Abbildung 5-3.


**Abbildung 5-3 Die Struktur des Application Modules von „EDOS“**

Mit der Fertigstellung des Application Modules ist die Konstruktion der Komponenten, welche später die Mittelschicht repräsentieren sollen abgeschlossen. Die Implementierung zusätzlicher Geschäftslogik auf der Ebene der Entity Objects soll hier zunächst vernachlässigt werden.
Die bis zu diesem Zeitpunkt seitens des BC4J Frameworks generierten Quellcodes sollen im nächsten Abschnitt detaillierter erklärt werden, um ein besseres Verständnis des Frameworks zu vermitteln.

### 5.3.2 Erläuterung der von BC4J generierten Quellcodes

Exemplarisch können die nun beschriebenen Quellcodes sowie die XML-Dateien für die Komponente „Kunde“ aus der EDOS Anwendung im Anhang dieser Arbeit (Kapitel 8) nachgeschlagen werden.
Im Kapitel 4 dieser Arbeit wurde bereits erläutert, dass es sich bei BC4J um ein Persistenzframework handelt, welches vollständig auf Java und XML-Technologien basiert.

Damit ist gemeint, dass jede Komponente sowohl aus Java- als auch aus XML-Dateien besteht. Der grundlegende Unterschied besteht darin, dass die XML-Dateien rein deskriptiven Charakter haben, d.h. sie beschreiben die Eigenschaften einer Komponente, während die Java Dateien programmatischen Charakter aufweisen, d.h. sie bilden im Sinne von Java die eigentliche Implementierung der Komponente.

Die Existenz einer solchen XML-Datei ist für jede BC4J Komponente zwingend erforderlich, währenddessen eine entsprechende Java Datei nicht notwendigerweise vorhanden sein muss. Sie wird lediglich in den Fällen benötigt, in denen der Entwickler expliziten Einfluss auf die Implementierung der Komponente nehmen will oder muss. Ist er dazu nicht gezwungen, so wird seitens des BC4J Framworks eine Standardimplementierung der entsprechenden Komponente herangezogen.

In beiden Fällen wird zum Zeitpunkt der Instanzierung die passende XML-Datei eingelesen und die dazugehörige Komponente aufgrund der dort gespeicherten Informationen konfiguriert.

Im Falle eines Entity Objects beispielsweise beinhaltet die dazugehörige XML-Datei (→ vgl. „Kunde.xml“, Kapitel 8.2.2) Informationen über sämtliche dieser Komponente angehörenden Attribute, d.h. Informationen über Attributnamen, Datentypen, Mapping von Datentypen in der Datenbank auf Java Datentypen, etc.

Die implementierende Java Datei eines Entity Objects (→ vgl. „KundeImpl.java“, Kapitel 8.2.3) verfügt lediglich über Methoden zum Lesen und Ändern der Attribute. Typischerweise ist dies die Stelle, an der die gesamte Geschäftlogik des Systems untergebracht wird.

Bei View Objects wird die XML-Datei (→ vgl. „KundeView.xml“, Kapitel 8.2.4) dazu genutzt, um auf der einen Seite das zu kapselnde SQL-Statement zu speichern und auf der anderen Seite zu beschreiben, auf welchen Attributen der Entity Objects die Attribute des View Objects basieren.

Die dazugehörige Java-Klasse (→ vgl. „KundeViewImpl.java“, Kapitel 8.2.5) enthält standardmäßig nur einen parameterlosen Konstruktor.

Bei den Application Modules wird in den entsprechenden XML-Dateien (→ vgl. „EDOSModule.xml“, Kapitel 8.2.6) die durch den Entwickler definierte Struktur aus View Objects (→ vgl. 5.3.1) gesichert, während in der implementierenden Java

Klasse (→ vgl. „EDOSModuleImpl.java“, Kapitel 8.2.7) alle notwendigen Methoden generiert werden, um an die gewünschten View Objects zu gelangen.

### 5.3.3 Der EDOS-Client als Swing-Applikation

An dieser Stelle soll anhand einiger beispielhafter Screenshots gezeigt werden, wie ein Client für die Beispielanwendung EDOS in Form einer Swing-Applikation aussehen könnte.

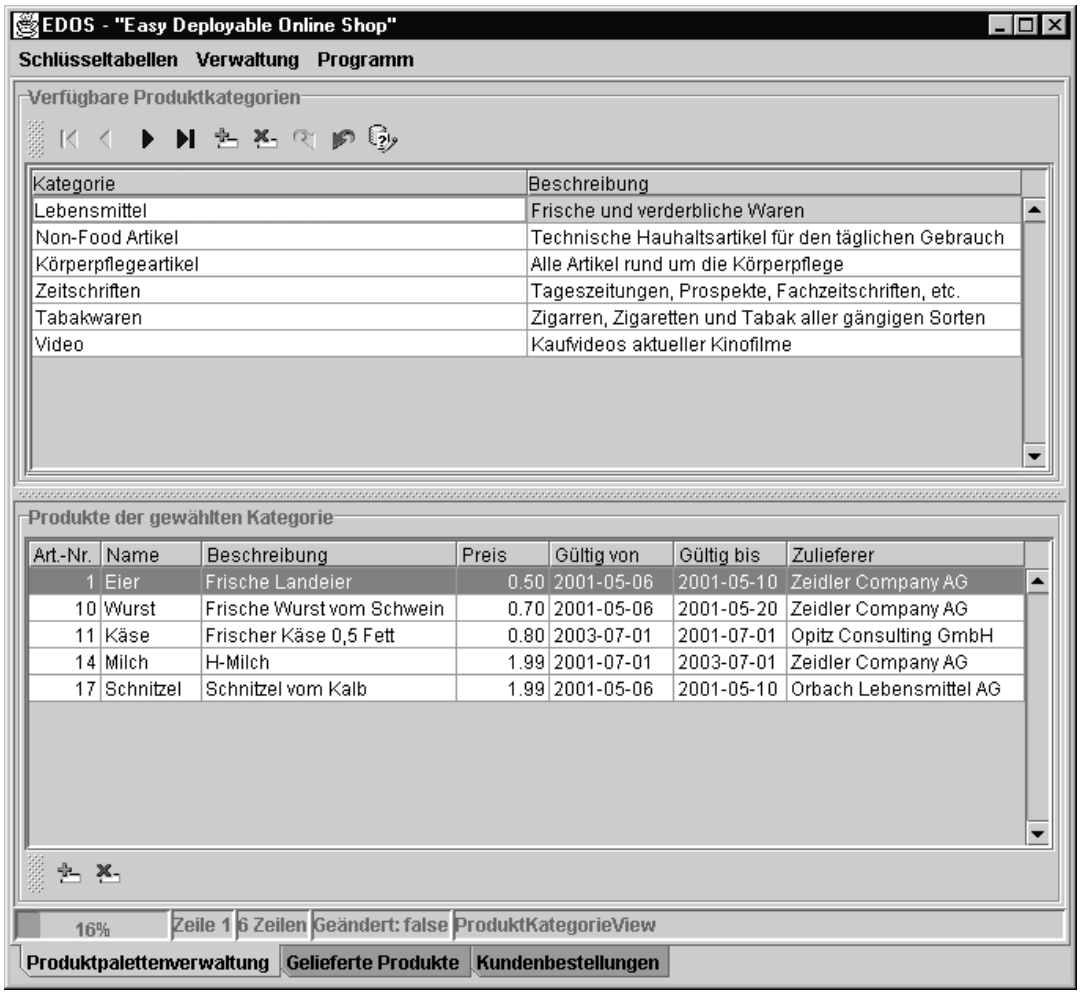


**Abbildung 5-4 Die Produktpalettenverwaltung von EDOS**
**Quelle: Screenshot der Anwendung**

Für die technische Realisierung dieses Clients wurde das DACF-Framework von Oracle (→ vgl. Abschnitt 4.2.7) genutzt. Abbildung 5-4 zeigt die DACF-Maske, durch welche die gesamte Produktpalette verwaltet werden kann. Im oberen Teil

können die Produktkategorien (Master) gepflegt werden, während im unteren Teil der Maske die zur jeweiligen Kategorie gehörigen Produkte (Detail) verwaltet werden. Im Sinne von BC4J basiert diese Maske auf den beiden View Objects *ProduktKategorieView* und *ProduktViewDetailToKategorie*. Zur Master-Detail Synchronisation wird der dazugehörige View Link *ProdPkatFkLink* verwendet.

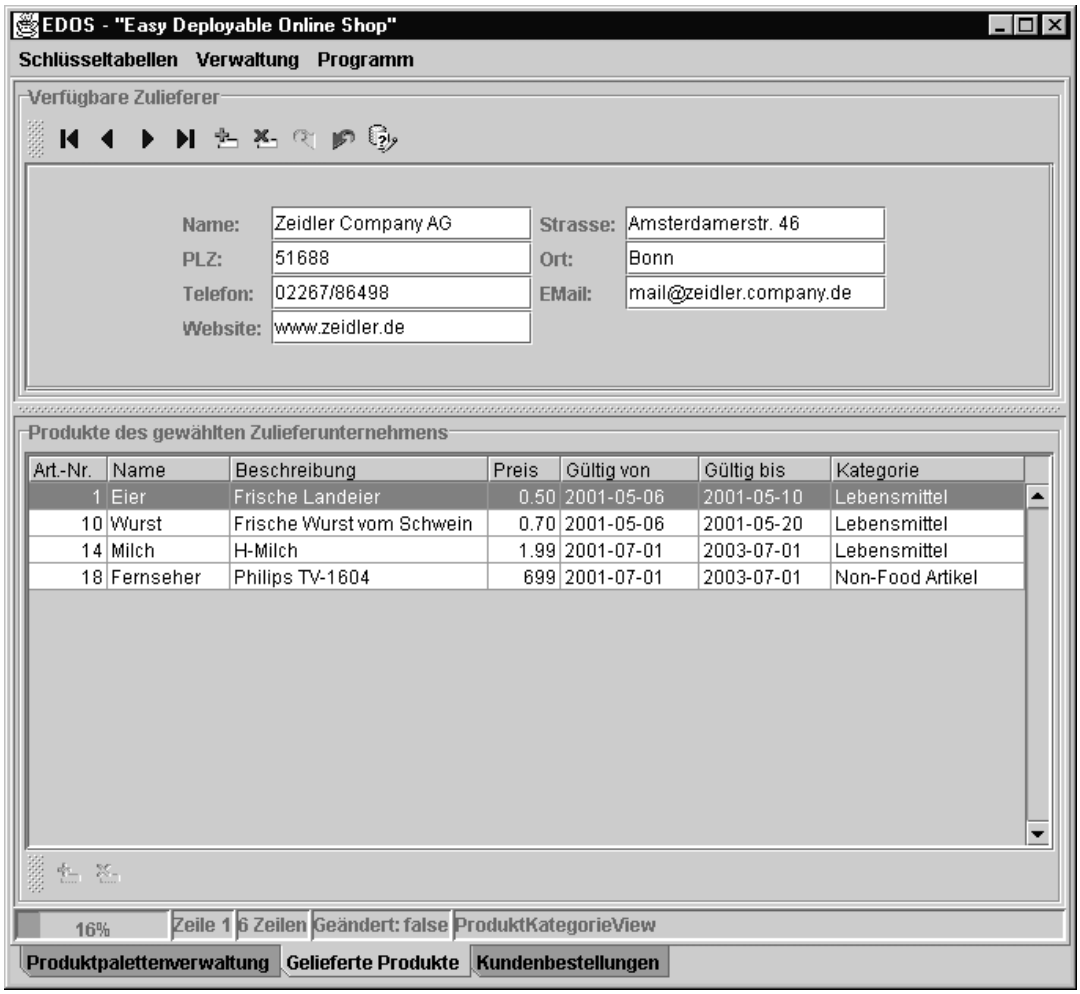


**Abbildung 5-5 Die Übersicht über die gelieferten Produkte**
**Quelle: Screenshot der Anwendung**

Abbildung 5-5 zeigt eine etwas andere Sicht auf dieselben Daten. Diesmal werden die Informationen aus der Sicht der Zulieferer erzeugt, d.h. hier wird eine Übersicht darüber gegeben, welcher Zulieferer welche Artikel liefert. Von der Datenbank aus betrachtet, bedeutet dies die Manipulation der Tabellen *PRODUKT* und

*PRODUKT_KATEGORIE* über die entsprechenden View Objects *ProduktView* und *ProduktKategorieView*.

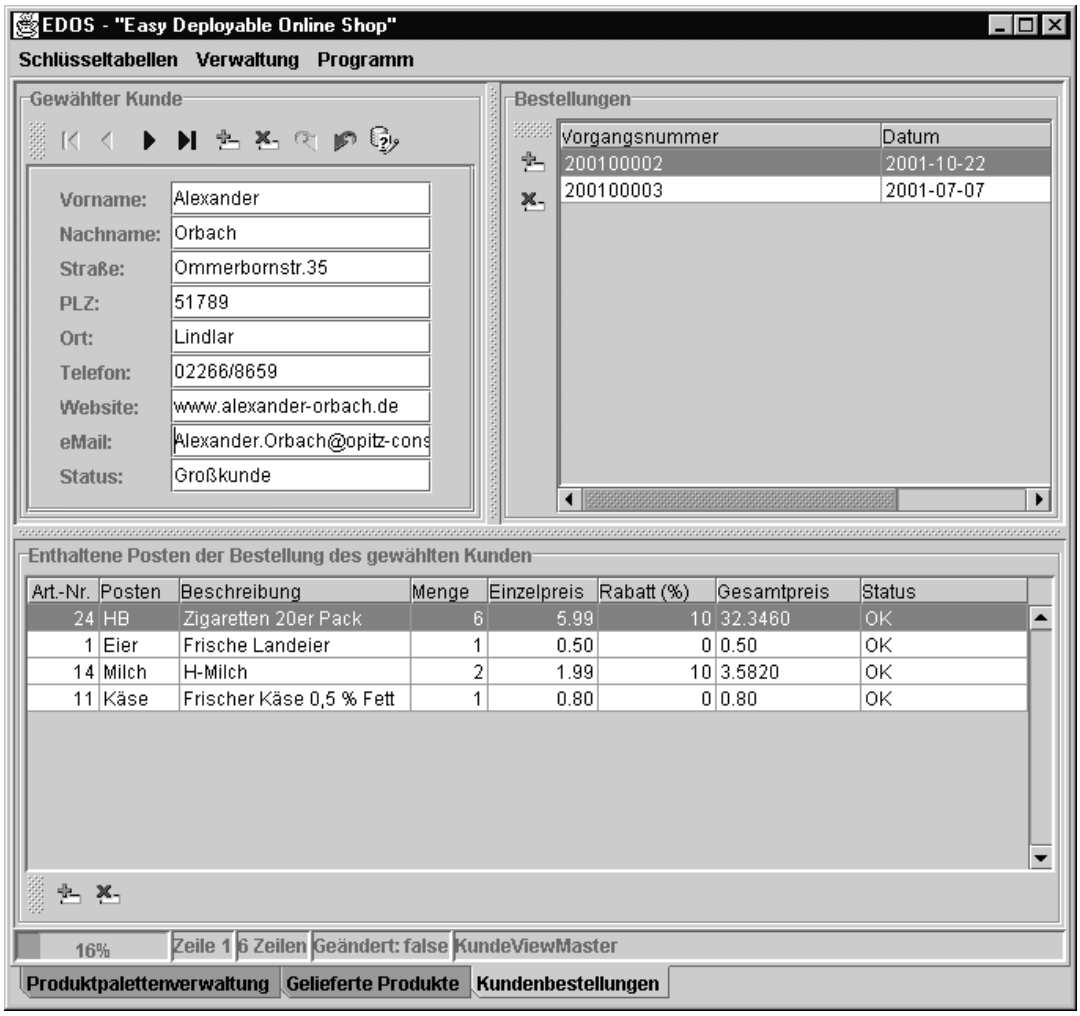


**Abbildung 5-6 Die Verwaltung der Kundenbestellungen**
**Quelle: Screenshot der Anwendung**

Abbildung 5-6 zeigt die DACF-Maske, durch die Bestellungen der Kunden bearbeitet werden können. Zunächst wird der gewünschte Kunde ausgewählt, worauf eine Liste seiner Bestellungen erscheint. Anschließend können die Einzelposten jeder dieser Bestellungen detailliert bearbeitet werden. Als Grundlage der Maske dienen in diesem Fall die drei View Objects *KundeViewMaster*, *BestellungViewDetail* und *PostenViewDetailOfBestellung*.

## 5.4 Der Applikationsserver

Innerhalb von Drei-Schicht-Architekturen spielt der Application Server als zentrale Instanz für die Bereitstellung der Komponenten eine wichtige Rolle.
Für das Deployment der Beispielanwendung, welche im vorangegangenen Kapitel beschrieben wurde, soll zunächst auf dem Server eine Oracle8i Datenbank zum Einsatz gebracht werden, die bereits standardmäßig die notwendigen Umgebungen sowohl für Enterprise JavaBeans- als auch für CORBA-Komponenten zur Verfügung stellt.
Der folgende Teil der Arbeit konzentriert sich hauptsächlich auf das Deployment der entwickelten Komponenten in diese Datenbank. Alternative EJB- bzw. CORBA-Umgebungen werden im Abschnitt 5.4.4 diskutiert.

### 5.4.1 Grundlagen für den Einsatz von Java in der Oracle8i Datenbank

Die Oracle8i Datenbank in der Version 8.1.7. bringt eine eigene Oracle Java Virtual Machine (OJVM) mit sich. Diese JVM stellt eine zu 100% der Spezifikation von SUN entsprechende serverseitige Java Umgebung dar.
Java Applikationen müssen zunächst in die Datenbank geladen werden. Daraufhin muss sich der Entwickler darüber Gedanken machen, welche Klassen oder Methoden für den Client verfügbar sein sollen. Diesen Arbeitsgang des „Sichtbarmachens“ von Klassen oder Methoden bezeichnet Oracle als „publishen“. Diese für den Client sichtbare Klassen können daraufhin in Form einer Datenbank-Session gestartet werden. Jeder Client erhält bei der Ausführung einer Applikation seine eigene Umgebung, d.h. aus der Sicht des Clients entsteht der Eindruck, als würde er eine eigene Virtual Machine auf dem Server ausführen.
Die Oracle8i Datenbank unterscheidet drei Typen von Java-Dateien, welche in die Datenbank geladen werden können (→ vgl. Tabelle 5-1).

| Bezeichnung in der Oracle8i Datenbank | Beispiele für solche Dateien |
|---|---|
| Java source schema objects | *.java<br>*.sqlj |
| Java class schema objects | *.class |
| Java resource schema objects | *.properties<br>*.ser |

**Tabelle 5-1 Java-Dateitypen, die in die Oracle8i Datenbank geladen werden können**
**Quelle: Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.**

Das Hochladen der Dateien zur Datenbank wird typischerweise durch ein Werkzeug von Oracle namens „loadjava“ bewerkstelligt. Die Syntax dieses kommandozeilengestützten Werkzeuges zeigt die Abbildung 5-7.

```
loadjava {-user | -u}
<user>/<password>[@<database>] [options]
<file>.java | <file>.class | <file>.jar |
<file>.zip |
<file>.sqlj | <resourcefile> ...
  [-debug]
  [-d | -definer]
  [-e | -encoding <encoding_scheme>]
  [-f | -force]
  [-g | -grant <user> [, <user>]...]
  [-help]
  [-nohelp]
  [-o | -oci8]
  [-order ]
  [-noverify]
  [-r | -resolve]
  [-R | -resolver "resolver_spec"]
  [-S | -schema <schema>]
  [-stdout ]
  [-s | -synonym]
  [-tableschema <schema>]
  [-t | -thin]
  [-v | -verbose]
```

**Abbildung 5-7 Die Syntax des „loadjava“-Tools von Oracle**
**Quelle: Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.**

Der wichtigsten Parameter dieses Werkzeuges sind nachfolgend in der Tabelle 5-2 erläutert.

| Parameter | Bedeutung |
|---|---|
| -user <user>/<password> [@<database>] | Hier wird der Benutzername, sein Passwort und die Zieldatenbank angegeben, in die die Java-Dateien geladen werden sollen. |
| -thin | Gibt an, dass „loadjava“ mit der Datenbank über einen sogenannten thin JDBC Treiber kommunizieren soll. |
| -verbose | Erzeugt beim Hochladen der Dateien Statusmeldungen über den Vorgang des Deployments, um den Entwickler über den aktuellen Stand zu informieren |
| -resolve | Wenn .class Dateien beim Aufruf von „loadjava“ übergeben wurden, werden alle Abhängigkeiten, welche von dieser Klasse ausgehen, geprüft.<br>Es ist ebenfalls möglich, anstatt der .class Dateien Java-Quellcodes in Form von .java Dateien beim Aufruf von „loadjava“ zu übergeben. In diesem Fall werden diese Dateien zunächst serverseitig compiliert. Daraufhin wird die entstandene .class Datei in Form eines class schema objects in der Datenbank gespeichert. |
| -grant <user> | Weist das Recht zur Ausführung (EXECUTE-Recht) der Methoden dieser Klasse einem bestimmten Benutzer zu. |

**Tabelle 5-2 Die gebräuchlichsten „loadjava“ Parameter**
**Quelle: Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.**

Im Zusammenhang mit .jar oder .zip Dateien, welche auch als Argument für „loadjava“ übergeben werden können, ist wichtig, dass diese Dateitypen nicht direkt in der Datenbank abgelegt werden. Eine Übertragung solcher komprimierten Dateitypen geschieht zunächst durch ein Extrahieren des jeweiligen Archivs und ein darauf folgendes Einfügen der einzelnen Dateien in die Datenbank.

### 5.4.2 Die Oracle8i Datenbank als CORBA-Umgebung

Im Produktumfang der Oracle8i Datenbank ist bereits ein vollständiger CORBA-ORB integriert. Dieser ORB basiert auf dem Visibroker in der Version 3.4 der Firma Visigenic, an dem von Oracle lediglich leichte Modifikationen durchgeführt wurden. Diese Modifikationen beziehen sich auf den Mechanismus zum Auffinden von CORBA Objekten sowie den Mechanismus zum Aktivieren des ORBs.
Diese Vorgänge sind durch die Änderungen seitens Oracle für den Entwickler weitgehend transparent geworden.

### 5.4.3 Die Oracle8i Datenbank als EJB-Umgebung

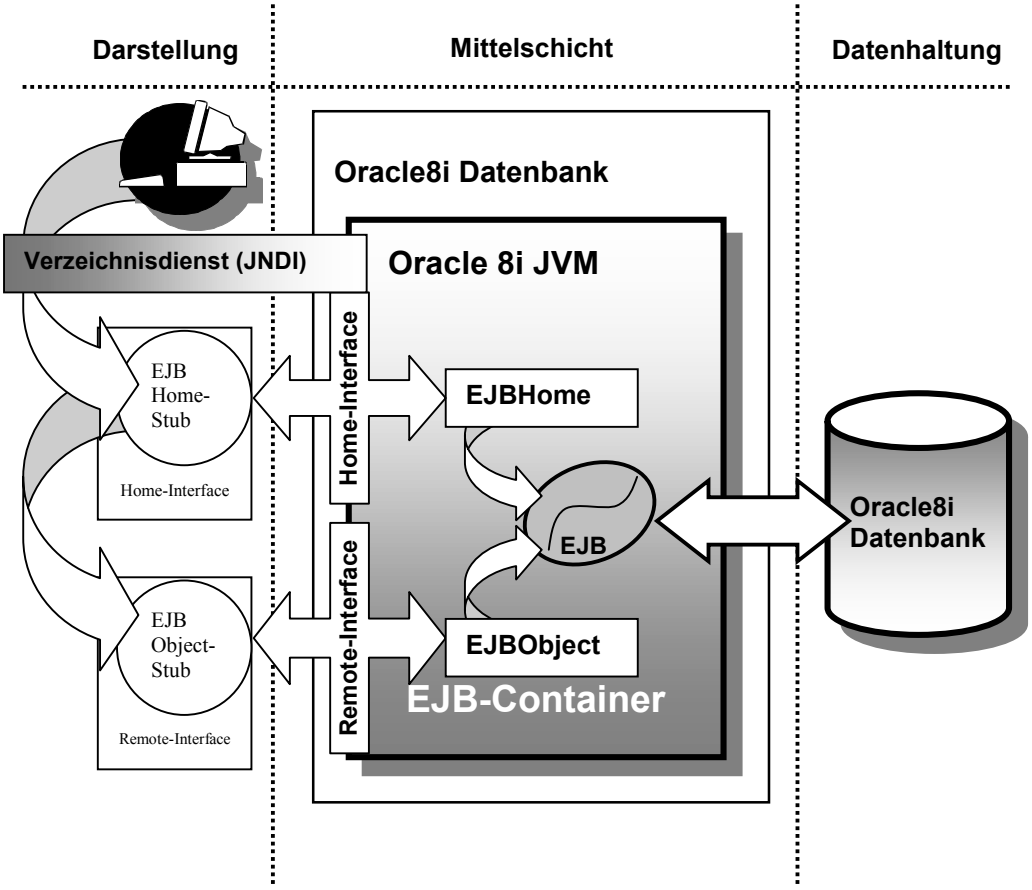


**Abbildung 5-8 Die Oracle8i Datenbank als EJB-Container**
**Quelle: In Anlehnung an: Oracle8i Documentation**

Die Oracle8i Datenbank bietet eine vollständige Laufzeitumgebung für den Einsatz von Enterprise JavaBeans. Diese Architektur wird durch die Abbildung 5-8 deutlich. Der EJB-Container befindet sich in Form der Oracle8i Java Virtual

Machine in dieser Datenbank. Durch die obige Abbildung wird besonders deutlich, dass die Clients ausschließlich über die vorgeschalteten Objekte EJBHome und EJBObject auf die eigentliche Instanzen der Enterprise JavaBeans zugreifen .

Die Abbildung 5-9 zeigt die Syntax des kommandozeilengestützten Werkzeugs „deployejb“ von Oracle, mit dem es auf einfache Art und Weise möglich ist, die einzelnen Bestandteile einer Enterprise JavaBean (→ vgl. Abschnitt 3.4.4) in die Datenbank einzuspielen.

Dabei werden die benötigten Klassen (sofern dies noch nicht geschehen ist) kompiliert und anschließend in die Datenbank geladen. Die fehlenden Klassen, also die EJBHome und die EJBObject Klasse werden ebenfalls generiert. Um letztendlich die Komponente nach außen für die Clients sichtbar zu machen, published dieses Werkzeug das Home Interface der Bean in einem sogenannten Namensraum (engl. „Namespace“).

```
deployejb {-user | -u} <username>
  {-password | -p} <password>
  {-service | -s} <serviceURL> -descriptor
<file> -temp <work_dir> <beanjar>
  [-addclasspath <dirlist>]
  [-beanonly]
  [-credsFile <credentials>]
  [-describe | -d]
  [-generated <clientjar>]
  [-help | -h]
  [-iiop]
  [-keep]
  [-oracledescriptor <file>]
  [-republish]
  [-resolver "resolver_spec"]
  [-role <role>]
  [-ssl]
  [-useServiceName]
  [-verbose]
  [-version | -v]
```

**Abbildung 5-9 Die Syntax des „deployejb“-Tools von Oracle**

**Quelle: Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.**

Eine Übersicht über die wichtigsten Parameter dieses Werkzeuges sollen in der folgenden Tabelle 5-3 erläutert werden.

| **Parameter** | **Bedeutung** |
|---|---|
| -user | Identifiziert den Benutzer in dessen Schema die Bean eingespielt werden soll. |
| -password | Das dazugehörige Paßwort für den obigen Benutzer. |
| -service | Hier wird die URL der Datenbank angegeben, in dessen Namensraum die Bean verfügbar gemacht werden soll. Die URL dazu hat folgende Syntax:<br>*sess_iiop://<host>:<lport>:<sid>*<br>Dabei ist <*host*> die Bezeichnung des Rechners, auf dem die Datenbank läuft. <*lport*> steht für den Port, auf dem dieser Rechner auf IIOP Anfragen der Clients antwortet. Die <*sid*> ist dann letzendlich die eindeutige Identifikation der Datenbankinstanz. |
| -republish | Für den Fall, dass die Bean in der Vergangenheit bereits einmal deployed wurde, wird das unter Umständen veränderte Home Interface erneut gepublished. |
| -descriptor | Hier wird die Stelle angegeben, an der das Werkzeug den Deployment Descriptor finden kann. |
| -oracledescriptor | Beim EJB-Deployment in die Datenbank generiert der Jdeveloper neben dem normalen Deployment Descriptor einen Oracle spezifischen Deployment Descriptor. |
| -generated | Mit diesem Parameter können die Namen der generierten .jar Archive angegeben werden, welche später all diejenigen Dateien enthalten, die benötigt werden, um eine Kommunikation vom Client zur mittleren Schicht aufbauen zu können. |

**Tabelle 5-3 Die gebräuchlichsten „deployejb“ Parameter**

**Quelle: Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.**

Dadurch können die Clients zur Laufzeit über den Java eigenen Verzeichnisdienst JNDI eine Referenz auf eine Enterprise Bean anfordern, welche dem Home Interface entspricht.

### 5.4.4 Alternative EJB und CORBA Umgebungen

Entwickler, welche BC4J Komponenten mit dem Ziel entwickeln, diese in einer verteilten Umgebung zu betreiben, haben nicht nur die Möglichkeit, diese durch die bereits beschriebenen Werkzeuge in die Datenbank zu laden und dort den Clients zur Verfügung zu stellen. Sowohl für den Enterprise JavaBans als auch für den CORBA Bereich soll jeweils eine Alternative vorgestellt werden.

#### *5.4.4.1 OC4J als alternativer EJB-Container*

Aktuellen Informationen von Oracle zur Folge wird die Oracle8i JVM als EJB-Container (→ vgl. vorangegangener Abschnitt) an Bedeutung verlieren. Stattdessen wird die Entwicklung eines neuen EJB-Containers vorangetrieben, welcher bereits in den Oracle9i Application Server 1.0.2.2 integriert ist.

Dieser EJB-Container trägt den Namen **O**racle9iAS **C**ontainers **f**or **J**2EE (kurz OC4J) und stellt einen zu 100% den Spezifikationen von Sun entsprechenden EJB-Container dar. Er bietet eine vollständige J2EE (Java 2 Enterprise Edition) konforme Umgebung.

Dieser Container ist selber komplett in Java entwickelt und kann auf den aktuellen standardmäßig verfügbaren Virtual Machines von SUN ausgeführt werden. Das bedeutet natürlich, dass dieser Container auf den Plattformen Solaris, HP-UX, AIX, Windows NT und Linux betrieben werden kann. Laut Oracle soll OC4J im Gegensatz zur Oracle8i JVM als EJB-Container weitaus leichtgewichtiger und auch performanter sein.

Das zukünftige Ziel, welches durch Oracle nach eigenen Aussagen im Java Umfeld verfolgt wird, ist die Integration von OC4J mit ihrem bestehenden Persistenzframework BC4J (→ vgl. Abschnitt 4). Ein erster Schritt in diese Richtung soll der demnächst verfügbare JDeveloper9i sein.

### *5.4.4.2 Standalone Visibroker als alternativer CORBA-ORB*

Anstatt den bereits in der Oracle8i Datenbank integrierten Visibroker CORBA-ORB in seiner Version 3.4 zu verwenden, bietet sich dem Entwickler die Möglichkeit, eine neuere Version des Visibrokers unabhängig von der Datenbank, d.h. also standalone auf einem anderen Server zu installieren.

Der Visibroker als CORBA-ORB liegt momentan in der Version 4.5 vor und entspricht dem CORBA Standard 2.3. Im Gegensatz zum gerade vorgestellten OC4J-Container, ist ein Deployment auf der Basis des Visibrokers bereits schon zum jetzigen Zeitpunkt durch den JDeveloper in der Version 3.2.3 möglich. Diese Möglichkeit wird noch genauer im Abschnitt 5.5.5 dieser Arbeit beschrieben.

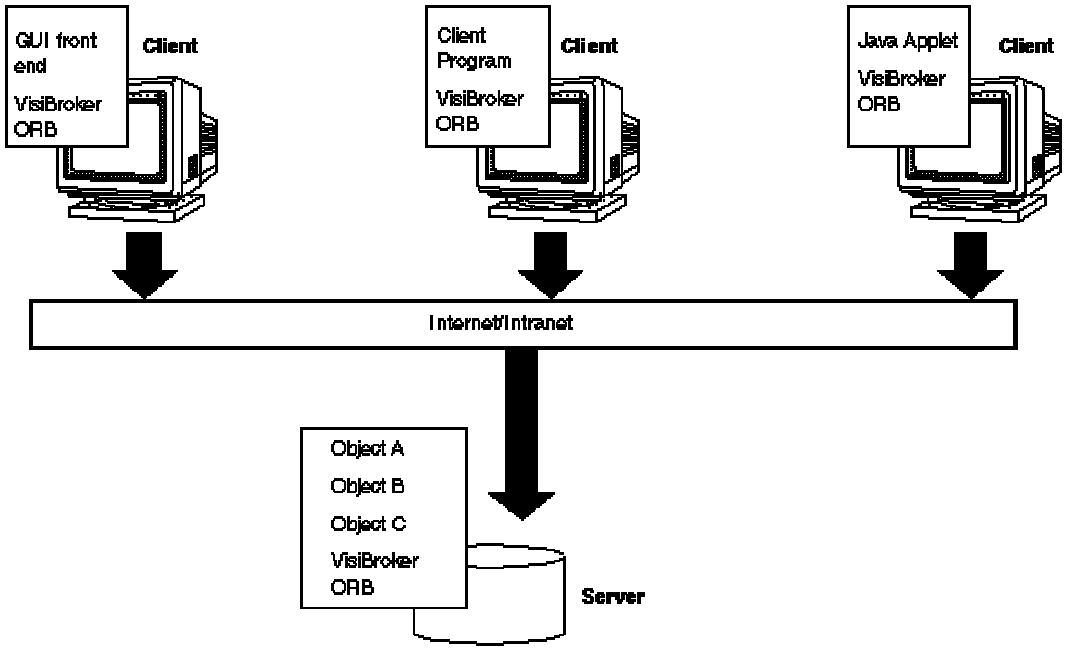


**Abbildung 5-10 Funktionsweise des Visibroker ORB**

**Quelle: Inprise Corporation Inc, *Visibroker 3.3/3.4 Programmer's Guide, http://www.borland.com/techpubs/books/vbj/vbj33/index.html*.**

Abbildung 5-10 zeigt die grundsätzliche Funktionsweise des Visibrokers. Auf jedem Rechner, der ein Clientprogramm ausführen soll, muss der Visibroker ORB installiert werden. Auf dem Visibroker Server befinden sich dann sowohl der ORB als auch die verfügbaren Objektimplementierungen.

Eine Besonderheit beim Einsatz des Visibrokers ist das Auffinden und das Binden der gewünschten CORBA Objekte an den Client. Um dies weitestgehend für den Entwickler transparent zu halten, wurde die sogenannte Smart Agent Technologie entwickelt.

Bei einem solchen Smart Agent handelt es sich um einen Verzeichnisdienst, der von den Clients in der Form genutzt wird, als dass er genaue Kenntnis darüber hat, an welcher Stelle ein bestimmtes CORBA-Objekt aufzufinden ist. Ein solcher Smart Agent muss lediglich auf einem einzigen Rechner innerhalb des lokalen Netzwerks installiert und gestartet werden. Sobald ein Client versucht ein bestimmtes CORBA Objekt an sich zu binden, wird der Smart Agent automatisch konsultiert und kann somit Auskunft über den Ort geben, an dem das Objekt zu finden ist.

Smart Agents werden vom Client durch eine das ganze Netzwerk durchdringende Suchmeldung (engl. Broadcasting) ausfindig gemacht. Da es durchaus vorkommt, dass mehrere Smart Agents innerhalb ein und desselben Netzwerks laufen, wird grundsätzlich der Smart Agent vom Client befragt, welcher zuerst auf die Broadcast Suchmeldung antwortet.

Darüber hinaus sind im Umfang des Visibrokers 4.5 noch andere Dienste enthalten, welche im Folgenden kurz vorgestellt werden sollen.

#### *5.4.4.2.1 Naming Service*

Dieser Namensdienst, welcher mit dem Visibroker 4.5 ausgeliefert wird, ermöglicht ein Assoziieren von einem oder mehreren logischen Namen mit einer serverseitigen Objektimplementierung. Dieser Service wird daraufhin von den Clients genutzt, um anhand der logischen Namen Referenzen auf diese CORBA Objekte zu erhalten (→ vgl. Abschnitt 3.3.5).

#### *5.4.4.2.2 Event Service*

Genauso wie der Naming Service wurde der Event Service bereits im Abschnitt 3.3.5 vorgestellt. Grundsätzlich geht es um eine Endkopplung von Objekten, welche durch ein Prinzip von Publishern und Subscribern hergestellt wird. Publisher sind in dem Fall die Objekte, welche eine Nachricht erzeugen und diese

versenden können. Diese Nachrichten werden von den Subscribern aufgefangen, die dann darauf reagieren können.

#### *5.4.4.2.3 Gatekeeper*

Dieser Dienst ermöglicht es den Clients mit Servern in Verbindung zu treten, auf die sie ansonsten aufgrund von Java Sicherheitsrichtlinien oder Firewalls keinerlei Zugriff hätten. Der Gatekeeper ist in diesem Zusammenhang als ein Gateway zu verstehen, welches zwischen Clients und Servern agiert.

Wird ein Server beispielsweise durch eine Firewall geschützt, hat dies zur Folge, dass die Firewall nur HTTP Verkehr zwischen dem Server und der Außenwelt zuläßt, was Anfragen von clientseitigen Applets über das IIOP Protokoll zunächst einmal unmöglich macht.

Der Gatekeeper bietet für dieses Problem eine Lösung in Form des sogenannten HTTP Tunnelings an. In seiner Funktion als Gateway hat er die Möglichkeit die eingehenden IIOP Requests in gültige HTTP Anfragen zu konvertieren, welche durch die Firewall hindurch zum Server geschickt werden können. Das Ergebnis kann auf demselben Kommunikationskanal wieder zurückgeschickt werden.

Im technischen Sinne ist der Gatekeeper ein vollständig der OMG CORBA Firewall Spezifikation entsprechender Proxy Server.

## 5.5 Das Deployment der Beispielanwendung

Der nun folgende Teil der Arbeit soll die technischen Hintergründe schildern, die notwendig sind, um die Beispielanwendung „EDOS“ in einer Drei-Schicht-Architektur ausführen zu können. Es soll jeweils geklärt werden, wie das Deployment der Komponenten sowohl als Enterprise Beans als auch in Form von CORBA-Objekten unter Zuhilfenahme des JDevelopers vonstatten geht. In einem weiteren Schritt wird darauf eingegangen, wie die notwendigen Verbindungen zwischen den einzelnen Schichten etabliert werden können.

### 5.5.1 Grundlagen für das BC4J-Deployment mit dem JDeveloper

Grundsätzlich sind es im Sinne von BC4J die Application Modules, welche in Form von Enterprise JavaBeans oder CORBA Objekten in der mittleren Schicht deployed werden. Alle anderen Komponenten eines solchen BC4J Projektes (Entity

Objects, View Object, Associations, View Links, etc.) liegen als Java Archive auf dem Server vor und werden gegebenenfalls durch das entstandene CORBA-Objekt oder die Enterprise Bean genutzt.

Um den Entwickler beim Deployment dieser Komponenten in eine verteilte Umgebung zu unterstützen, wurde das Prinzip der sogenannten Deployment Profiles entwickelt. Bei diesen Profilen handelt es sich um separate Dateien, welche die Endung .prf tragen. In diesen Dateien werden Informationen über den Typ des Projektes sowie über die innerhalb des Projektes zu deployenden Dateien gehalten. Darüber hinaus werden dort Daten über den Ort und den Typ des entfernten Systems gespeichert.

Es ist hierbei nicht die Aufgabe des Entwicklers, solche Deployment Profiles selber zu verfassen. Sie werden vielmehr aufgrund seiner Angaben automatisch generiert und können daraufhin aufgerufen werden. Wird ein solches Profil gestartet, so wird dass Deployment nach dessen Vorgaben durchgeführt.

Für den Fall, dass ein Deployment der Komponenten in die Oracle Datenbank stattfinden soll, werden die notwendigen Bestandteile direkt durch den JDeveloper in die Datenbank eingespielt und die benötigten Klassen nach außen für die Clients sichtbar gemacht (Publishing). Alle CORBA- bzw. EJB-spezifischen Informationen werden ebenfalls durch die Deployment Profiles verwaltet.

Im Falle eines Deployments auf andere Plattformen wie beispielsweise einen Webserver, werden die Komponenten durch den JDeveloper zunächst lokal in Form von .jar Archiven zur Verfügung gestellt. Danach müssen diese Dateien per Hand auf den Webserver übertragen werden. Exemplarisch ist ein solches Deployment Profil, welches für das Deployment der Beispielanwendung „EDOS“ als CORBA Objekte genutzt werden kann, im Anhang dieser Arbeit abgedruckt (→ vgl. Abschnitt 8.2.8).

Wie oben bereits erwähnt, werden solche Profile seitens des JDevelopers im Falle eines Deployments der Komponenten als CORBA Objekte in eine Oracle8i Datenbank dazu genutzt, um die korrekten „loadjava“ bzw. im Falle eines EJB Deployments die „deployejb“ Kommandos zu erzeugen und auszuführen (→ vgl. Abschnitte 5.4.1 und 5.4.3).

Eine detailliertere Beschreibung dieser Vorgänge sowie die notwendige Konfiguration der Verbindungen innerhalb eines solchen verteilten Systems soll der folgende Abschnitt geben.

### 5.5.2 Deployment als EJB Session Bean

Bevor eine Anwendung als EJB Session Bean in die Datenbank deployed werden kann, muss zunächst einmal auf der Seite des Servers eine Laufzeitumgebung für die Enterprise JavaBeans geschaffen werden.
Diese Umgebung besteht aus einer ganzen Reihe von Java Klassen, welche in Form von Archiven (.jar Dateien) gebündelt vorliegen. An dieser Stelle ist der JDeveloper von Oracle in der Lage, alle benötigten Dateien anhand von entsprechenden „loadjava“ Kommandos automatisch in die Datenbank einzuspielen und dort auch verfügbar zu machen. Im Anschluss daran kann die eigentliche Anwendung in die Datenbank deployed werden.
Tabelle 5-4 soll Aufschluss darüber geben, welche Archive beim Anlegen eines Deployment Profiles erzeugt werden und welche Rollen sie im einzelnen spielen.

| **Erzeugtes Java Archiv** | **Beschreibung** |
|---|---|
| <Projektname>_bc4j.jar | In diesem Archiv werden alle im Rahmen der BC4J-Anwendung entstandenen XML-Files sowie die Implementierungen der Komponenten in Form von kompilierten .class Dateien abgelegt. |
| <Projektname>_bc4jCommonEJB.jar | Dieses Archiv beinhaltet die notwendigen Home- bzw. Remote Interfaces für die zu deployenden Enterprise Beans. Diese Interfaces werden durch den JDeveloper generiert und später für die Clients sichtbar gemacht. |
| <ApplicationModule> ServerOracleEJB.jar | Dieses Archiv beinhaltet eine Standardimplementierung für ein Application Module als Session Bean im Sinne der EJB-Spezifikation. |

| | |
|---|---|
| <ApplicationModule>EJBClient.jar | Dieses Archiv enthält alle notwendigen Klassen, die der Client benötigt, um mit Instanzen der Application Modules in Form von SessionBeans in Verbindung treten zu können. Neben diversen Hilfsklassen gehören hierzu insbesondere die clientseitigen Stubs, welche für den entfernten Methodenaufruf benötigt werden (→ vgl. Abschnitt 2.4). |
| <Projektname>_bc4jBcEJB.jar | Dieses Archiv beinhaltet eine Datei namens *connections.properties*. Hier liegen alle Informationen bereit, die für das Etablieren von JDBC oder IIOP Verbindungen notwendig sind. Neben dem Rechnernamen, sowie den Angaben zum benutzten Treiber werden hier auch die Benutzernamen und deren Passwörter gespeichert. Letztendlich muss dann nur noch der sprechende Name der Verbindung im Programmcode angegeben werden. Alle weiteren Informationen werden daraufhin aus dieser Datei bezogen. |

**Tabelle 5-4 Generierte Java Archive beim EJB-Deployment**
**Quelle: JDeveloper 3.2.3**

Nachdem auch die aus Tabelle 5-4 beschriebenen Archive seitens des JDevelopers in die Datenbank geladen wurden (mit Ausnahme des Client-Archivs), ist das Beispielmodul namens „de.edos.bc4j.EDOSModule“ bereits in Form einer SessionBean verfügbar.

Im nächsten Schritt muss nun sowohl die IIOP Verbindung vom Client zur mittleren Schicht als auch dessen JDBC Verbindung zur Datenbankschicht eingerichtet werden. Da es sich im Falle der Beispielanwendung „EDOS“ um einen Client handelt, welcher das DACF Framework von Oracle nutzt, müssen beide Verbindungen durch das bereits beschriebene SessionInfo Objekt (→ vgl.

Abschnitt 4.2.7) etabliert werden. Abbildung 5-11 zeigt den clientseitigen Programmcode, welcher notwendig ist, um diese Verbindungen innerhalb einer Drei-Schicht-Architektur aufzubauen.

```
SessionInfo sessionInfo = new SessionInfo();
[…]
//IIOP Connection zur mittleren Schicht.
//Hierzu wird der Verbindungsname und der JNDI Pfad angegeben, unter dem
//das Application Module deployed wurde.
EjbConnection ejbConn = new EjbConnection("AOREJB_IIOP",
"/test/aorejb/ejb");
//Die CustomConnection ist die JDBC Verbindung zwischen der mittleren
//Schicht und der Datenbankschicht.
ejbConn.setCustomConnection(new MyCustomEJBSessionBeanConnection());
sessionInfo.setConnectionInfo(ejbConn);
[…]
```

**Abbildung 5-11 Konfiguration des DACF-Clients für den Zugriff auf EJBs**

Die JDBC Verbindung zwischen der mittleren Schicht und der Datenbankschicht wurde im Rahmen der Beispielanwendung „EDOS“ durch eine separate Klasse mit dem Namen `MyCustomEJBSessionBeanConnection` gekapselt (→ vgl. Abbildung 5-12).

```
public class MyCustomEJBSessionBeanConnection implements CustomConnection
{
     public NamedConnection loginConnection(NamedConnection lc)
     {
        lc.setConnectionURL("jdbc:oracle:thin:@wntdb:1521:dbfe");
        lc.setUserName("aordb");
        lc.setPassWord("diplom");
        return lc;
     }
}
```

**Abbildung 5-12 Die Klasse MyCustomEJBSessionBeanConnection**

Nach Abschluss dieser Konfigurationen lässt sich der Client unter Zuhilfenahme des erzeugten clientseitigen Archivs ausführen.
Für den Fall, dass der Client nicht auf der Basis des DACF-Frameworks entwickelt wurde, müssten die benötigten Verbindungen auf eine andere Art und Weise eingerichtet werden. Am technischen Prinzip ändert sich jedoch nichts Wesentliches.

### 5.5.3 Deployment als EJB Entity Bean

Ein Deployment von Application Modules in Form von Entity Beans ist im Zusammenhang mit BC4J gar nicht möglich.
Der Grund dafür wird schnell verständlich, wenn man sich vor Augen führt, was BC4J eigentlich ist. Es handelt sich dabei um ein sogenanntes Persistenzframework, d.h. es können durch den Einsatz dieses Frameworks Komponenten basierend auf Geschäftsprozessen geschaffen werden, welche in Verbindung mit einer Datenbank die Fähigkeit besitzen, ihren Zustand dauerhaft in diesem Speichermedium zu sichern.
Exakt dieselbe Funktionalität bieten im Sinne der Enterprise JavaBeans Spezifikation von SUN die Entity-Beans. Somit würde man als Entwickler diese Funktionalität doppelt in ein System einbauen, was natürlich keinerlei Vorteile in sich birgt.
Darüber hinaus haben Application Modules den Charakter einer Dienstleistung. Der Client holt sich für die Dauer einer bestimmten Tätigkeit ein solches Module vom Server, um damit ein bestimmtes Ziel zu erreichen. Hat er seine Aufgabe erledigt, benötigt er das Module nicht mehr. Ein solches Szenario wird typischerweise durch den Einsatz von Session Beans umgesetzt.
Aus diesem Grunde wurde ein Deployment als EJB Entity Beans im Umfeld von BC4J aus Sicht von Oracle nicht weiterverfolgt.

### 5.5.4 Deployment als Oracle8i CORBA Server

Das Deployment der Application Modules als Oracle8i CORBA Server Objekte ist prinzipiell dem vorangegangenen EJB-Deployment sehr ähnlich.
Zunächst einmal muss ebenfalls eine eigene CORBA Laufzeitumgebung in der Datenbank des Servers etabliert werden.

Ist dieser Vorgang erfolgreich abgeschlossen, werden wiederum die erzeugten Archive der eigentlichen Anwendung in die Datenbank geladen. Diese unterscheiden sich nicht wesentlich von den Archiven des EJB-Deployments. Der einzige Unterschied besteht darin, dass die Deployment Profiles sowie die erzeugten Archive in diesem Fall CORBA-spezifischen Informationen beinhalten.
Verfügt das zu deployende Application Module über keinerlei „Custom Methods“[32], so werden auch keine Stubs und Skeletons für den entfernten Methodenaufruf generiert. Zu diesem Zweck genügt es für das CORBA-Deployment die standardmäßig in BC4J implementierten Stubs und Skeletons heranzuziehen, welche bereits Teil der CORBA Laufzeitumgebung sind.
Die anschließende Konfiguration der Verbindungen vollzieht sich auf gleiche Art und Weise, was die Abbildung 5-13 verdeutlicht. Die Klasse `MyCustomCORBA8iConnection` verweist dabei auf die gleiche Datenbank wie beim EJB-Deployment.

```
SessionInfo sessionInfo = new SessionInfo();
[…]
CorbaConnection corbaConn = new CorbaConnection("AOR_IIOP", "/test/aor");
corbaConn.setCustomConnection(new MyCustomCORBA8iConnection());
sessionInfo.setConnectionInfo(corbaConn);
[…]
```

**Abbildung 5-13 Konfiguration des DACF-Clients für den Zugriff auf Oracle8i CORBA Objekte**

### 5.5.5 Deployment als Visibroker CORBA Server

Die Funktionsweise des Visibrokers als alternativer CORBA-ORB wurde bereits im Rahmen dieser Arbeit beschrieben (→ vgl. Abschnitt 5.4.4.2). Für das Deployment der Beispielanwendung dieser Arbeit wurde der Visibroker ORB in seiner zur Zeit aktuellsten Version 4.5 verwendet.

[32] Unter „Custom Methods“ versteht man vom Entwickler eigens dem Modul hinzugefügte Dienste, die innerhalb einer verteilten Architektur dem Client zugänglich gemacht werden sollen (→ vgl. Abschnitt 4.2.5).

Bei dieser alternativen Methode des Deployments handelt es sich nicht um ein Deployment in eine Datenbank.
Stattdessen wird das Application Module, welches als CORBA Objekt verfügbar gemacht werden soll in Form einer Klasse auf dem Server gestartet. Diese Serverklasse wird im Rahmen des Visibroker Deployments durch den JDeveloper erzeugt und muss daher nicht vom Entwickler programmiert werden.
Die bereits beschriebene Smart Agent Technologie des Visibrokers trägt die Verantwortung dafür, dass das Application Module netzwerkweit verfügbar ist. Solch ein Smart Agent ist also als zentrale Anlaufstelle zu verstehen, wenn es um das Auffinden benötigter CORBA Objekte geht.
Das Etablieren der Verbindungen erweist sich daher gegenüber dem Deployment in eine Datenbank als unkomplizierter (→ vgl. Abbildung 5-14).

```
SessionInfo sessionInfo = new SessionInfo();
[…]
VBBindingConnection vbConn = new VBBindingConnection("AORDB_JDBC");
sessionInfo.setConnectionInfo(vbConn);
[…]
```

**Abbildung 5-14 Konfiguration der Verbindungen bei einem Deployment als Visibroker CORBA Server**

An dieser Stelle muss lediglich nur noch der Name der JDBC Verbindung der mittleren Schicht zur Datenbank angegeben werden. Die Adresse des Smart Agents bekommt die JVM zum Zeitpunkt der Ausführung des Clients als Parameter übergeben. Das Aussehen eines solchen Clientaufrufs zeigt die Abbildung 5-15.

```
java
-Dorg.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB
-DORBagentAddr=wntas.opitz-consulting.de
de.edos.client.applet.MainApplet
```

**Abbildung 5-15 Aufruf eines DACF-Clients für den Visibroker CORBA-ORB**

## 5.6 Diskussion der Ergebnisse hinsichtlich verschiedener Kriterien

Dieser Teil der vorliegenden Arbeit soll die praktischen Ergebnisse des Deployments der Beispielanwendung unter verschiedenen Geschichtspunkten bewerten.

Insbesondere soll an dieser Stelle der Einsatz der anhand von BC4J entwickelten Komponenten als CORBA Objekte in der Datenbank dem Einsatz auf der Basis des bereits beschriebenen Visibroker-ORB 4.5 gegenübergestellt werden.

Der Einsatz des OC4J-Containers als Alternative zum EJB-Deployment in eine Datenbank wird im Rahmen dieser Arbeit nicht berücksichtigt, da sich der JDeveloper9i als die neueste Version der Entwicklungsumgebung von Oracle momentan noch im Teststadium befindet. Zusätzlich dazu kann der DACF-Client der Beispielanwendung „EDOS“ nicht im Zusammenhang mit dem OC4J-Container betrieben werden.

Zukünftig wird es an dieser Stelle möglich sein, die Application Modules direkt in Form von Enterprise JavaBeans in diesen neuen Container zu deployen, was in der jetzigen Version des JDevelopers noch nicht möglich ist.

### 5.6.1 Performance

Beim Thema der Performance sind deutliche Unterschiede zwischen dem Deployment der Business Components in eine Oracle8i Datenbank und dem Deployment als Visibroker CORBA Objekt erkennbar.

Es hat sich gezeigt, dass die Anwendung sofern sie auf dem neuen Visibroker 4.5 basiert um einige Größenordnungen schneller arbeitet als wenn sie in der Datenbank abläuft. In Zahlen ausgedrückt benötigt die Anwendung unter dem Visibroker ORB etwa nur 20-30% der Zeit für den kompletten Aufbau einer DACF-Maske im Vergleich zu ihrer Datenbankversion.

Im Bereich der Enterprise JavaBeans soll sich eine ähnliche Tendenz abzeichnen. Auch hier läßt laut Oracle das Deployment der Application Modules auf der Basis des neuen OC4J-Containers einen deutlichen Perfomancegewinn erwarten.

### 5.6.2 Skalierbarkeit

Beim Thema der Skalierbarkeit geht es darum, welche Maßnahmen bei einer verteilten Anwendung getroffen werden können, um selbst bei einer großen Menge clientseitiger Anfragen noch ein akzeptables Laufzeitverhalten garantieren zu können.

Für den Fall, dass die Anwendung in die Datenbank deployed wurde, können die standardmäßig verfügbaren Mechanismen von Oracle genutzt werden, um eine Skalierbarkeit des Systems zu erreichen. In diesem Bereich bieten Oracle8i Datenbanken die Möglichkeit des Clusterings, d.h. mehrere Rechner können zu einem Verbund zusammengeschlossen werden. Für die Verteilung der Anfragen sorgt dann ein Produkt von Oracle namens Parallel Server, welches dafür Sorge trägt, die verfügbaren Ressourcen des Clusters möglichst optimal auszunutzen.

Eine Anwendung, welche auf der Basis des Visibroker CORBA-ORB deployed wurde, zeichnet sich ebenfalls durch die Fähigkeit der Skalierbarkeit aus. Der Visibroker ORB 4.5 ermöglicht es dem Server, auf dem er installiert ist, mehrere Clients parallel zu bedienen. Das bedeutet, es ist nicht notwendig, dass ein Client mit seiner Anfrage warten muss, bis dass die Ergebnisse für die anderen Clients errechnet wurden. Um dies zu erreichen wurden entsprechende Multi-threading Mechanismen für den Visibroker ORB entwickelt.

Auf der Seite des Clients verhält es sich auf ähnliche Art und Weise. Hier ist ebenfalls eine parallele Verarbeitung in der Form möglich, als dass mehrere Clients ihre Anfrage gleichzeitig über den ORB an den Server stellen können, ohne dass es dabei zu Problemen kommt. Durch ein solches Verhalten werden sowohl client- als auch serverseitig die vorhandenen Ressourcen optimal ausgenutzt.

Des weiteren bietet der Visibroker ORB die Möglichkeit, dass Objekte auf mehreren Servern repliziert werden können. Aus Sicht der Clients wirken diese Kopien alle wie ein und dasselbe Objekt. Sie benutzen den ORB auf genau die gleiche Art und Weise wie zuvor.

### 5.6.3 Vergleich der deployten Objekte mit den Spezifikationen

Dieser Teil der Arbeit soll Aufschluss darüber geben, inwieweit die Ergebnisse des Deployments aus dem JDeveloper mit den entsprechenden Spezifikationen von SUN bzw. der OMG übereinstimmen.

#### *5.6.3.1 Enterprise JavaBeans Spezifikation*

Grundsätzlich entsprechen die Enterprise JavaBeans, die auf der Basis der Application Modules durch den JDeveloper generiert wurden, der EJB-Spezifikation von SUN. Das bedeutet, es werden die erforderlichen Interfaces, die Stubs und Skeletons sowie die notwendigen Objekte (EJBObject und EJBHome) erzeugt.

Standardmäßig werden beim EJB-Deployment in die Oracle8i Datenbank ausschließlich stateful Session Beans erzeugt. Aufgrund der Diskussion der verschiedenen Typen von Session Beans im Abschnitt 3.4.6.1 dieser Arbeit bedeutet dies, dass diese Vorgehensweise beim Deployment wegen dem erhöhten Aufwand der Aktivierung bzw. Passivierung der Beans grundsätzlich immer die rechenintensivere Variante darstellt.

Beim Deployment der Application Modules als Enterprise JavaBeans in eine Datenbank werden grundsätzlich zwei Deployment Deskriptoren erzeugt. Der eine enthält die allgemeine Beschreibung der Enterprise Bean während der andere Oracle-spezifische Informationen für die Oracle8i Datenbank als Server beinhaltet. Dort sind beispielsweise die JNDI Namen für die einzelnen Beans gespeichert. Der Einsatz eines solchen speziell für den Server bestimmten Deskriptors ist durchaus im Rahmen der EJB-Spezifikation von SUN möglich.

Momentan haben die während des Deployments durch den JDeveloper erzeugten Archive allerdings nicht die erforderliche Struktur, damit sie direkt in einer J2EE konformen Umgebung eingesetzt werden könnten. Um ein Deployment auf einen anderen EJB-Container als die Datenbank durchführen zu können, müssten die seitens des JDevelopers generierten Archive ausgepackt und neu strukturiert werden. Darüber hinaus müsste der entsprechende server-spezifische Deployment Deskriptor unter Umständen von Hand geschrieben werden, was sich aus Sicht des Entwicklers als äußerst umständlich erweist.

Wie bereits im Abschnitt 5.4.4.1 dieser Arbeit beschrieben, wird diese Art des Deployments auf der Basis der Oracle8i JVM in der Datenbank an Bedeutung verlieren. Im Zuge der Weiterentwicklung des JDevelopers, welcher demnächst in der Version 9i erscheinen wird, findet eine immer stärker werdende Anlehnung an den J2EE Standard von SUN statt. Dies wird in der Form geschehen, als dass der

JDeveloper eine Möglichkeit besitzen wird, Enterprise JavaBeans direkt für den OC4J-Container[33] zu deployen.

#### *5.6.3.2 CORBA Spezifikation*

Die Oracle8i Datenbank nutzt für das Deployment eines BC4J Projektes als CORBA Server Objekte wie bereits beschrieben den Visibroker ORB 3.4 in einer seitens Oracle leicht modifizierten Version.

Insofern ist das Ergebnis des Deployments des JDevelopers in eine Oracle8i Datenbank konform zur CORBA 2.0 Spezifikation der OMG.

Das bedeutet für die deployten Business Components, dass für das Auffinden und Aktivieren eines CORBA-Objektes auf dem Server ein Basic Object Adapter (→ vgl. Abschnitt 3.3.3.3) genutzt wird. In diesem Zusammenhang ist interessant, dass der ORB in der Oracle8i Datenbank grundsätzlich weder über ein Interface Repository noch über ein Implementation Repository verfügt (→ vgl. Abschnitt 3.3.3.2.1), was die Nutzung des Dynamic Invocation Interfaces (DII) zum dynamischen Aufruf eines entfernten Objektes für den Client unmöglich macht. Die Kommunikation zwischen den Clients und dem Server kann somit nur über entsprechende Stubs und Skeletons statisch durchgeführt werden.

Eine weitere Auffälligkeit beim CORBA-Deployment der Business Components in eine Datenbank ist, dass es aus Sicht des Entwicklers nicht notwendig ist, eine Server-Klasse (→ vgl. Abschnitt 3.3.7.2) im Sinne von CORBA zu programmieren. In diesem Zusammenhang sorgt die Datenbank beim Deployment automatisch dafür, dass alle Objekte, welche gepublished werden, auch tatsächlich gegenüber den Clients verfügbar gemacht werden. Dies wird durch eine standardmäßige Klasse auf dem Datenbankserver bewerkstelligt.

Im Gegensatz zum Oracle8i Deployment des JDevelopers, entspricht ein Deployment der Komponenten auf der Basis des Visibroker ORB in seiner aktuellsten Version 4.5 dahingegen vollständig der CORBA Spezifikation 2.3.

Diese Version des Visibroker ORB verfügt über Implementierungen der durch die OMG definierten Dienste Naming Service und Event Service (→ vgl. Abschnitt

[33] Dieser Container ist bereits fester Bestandteil des Oracle9i Application Servers in seiner aktuellsten Version.

3.3.5). Darüber hinaus werden durch Visigenic noch weitere diverse Dienste im Rahmen des Visibrokers 4.5 zur Verfügung gestellt (Gatekeeper, etc.).

Entgegen dem Oracle ORB nutzt der Visibroker einen weiterentwickelten Object Adapter, den bereits vorgestellten Portable Object Adapter für die Aktivierung der Instanzen.

Abschließend ist noch bemerkenswert, dass aufgrund eines fertigen Application Modules generell keine IDL Beschreibungen im Sinne der CORBA-Spezifikation beim Deployment der BC4J-Komponenten erzeugt werden.

Solche Beschreibungen wären auch nur dann nützlich, wenn ein Entwickler die Absicht verfolgen würde, die Business Components aufgrund einer solchen IDL-Schnittstelle in einer anderen Programmiersprache erneut zu realisieren, was zwar theoretisch möglich jedoch praktisch nicht sinnvoll wäre.

# 6 Ausblick und Schlussbetrachtung

Das Ziel der vorliegenden Arbeit war es zunächst einmal, einen Einblick in die theoretischen Grundlagen Verteilter Systeme zu vermitteln. Darüber hinaus wurden in diesem Zusammenhang die industriellen Standards der Enterprise JavaBeans- sowie der CORBA-Spezifikation als Plattformen zur Umsetzung solcher Verteilten Architekturen beschrieben.

Aus Sicht eines Entwicklers, der das erste Mal mit einer solchen Umgebung in Berührung kommt, mögen solche Programmiermodelle als zu kompliziert und überdimensioniert erscheinen.

Bei einer genaueren Betrachtung zeigt sich allerdings, dass es durchaus sinnvoll sein kann, die Bestandteile eines Softwaresystems nach den Richtlinien eines solchen Modells zu entwerfen. Ist der Programmierer erst einmal mit der grundsätzlichen Vorgehensweise zur Erstellung eines CORBA-Server Objektes bzw. einer Enterprise Bean vertraut, so kann er sich im besten Falle vollständig auf die eigentliche Logik seiner Anwendung konzentrieren. Transaktionskontrollen, Sicherheitsprüfungen und Datenbankzugriffe sind nur einige Beispiele für Aufgaben, deren technische Umsetzung sonst Wochen oder Monate in Anspruch nehmen würde. Durch den Einsatz eines der oben genannten Komponentenmodelle können diese Funktionalitäten durch einen EJB-Container bzw. durch entsprechende CORBA-Dienste als fertige Implementierungen bereitgestellt werden, was zu einer immensen Verkürzung der Entwicklungszeiten führen kann.

Letztendlich läßt sich zu diesem Thema sagen, dass die vorgestellten Komponentenmodelle in der Evolution der Softwaretechnologie als wichtiger Schritt in Richtung skalierbarer, wiederverwendbarer und sicherer Systeme im industriellen Rahmen anzusehen sind.

Das „Business Components for Java Framework“ von Oracle als Entwicklungswerkzeug ermöglicht ein komfortables Erstellen und Testen der einzelnen Komponenten auf dem lokalen Rechner. Erst in einem zweiten Schritt können die entstandenen Komponenten in Form von EJBs oder CORBA-Objekten durch den JDeveloper bequem eingespielt werden.

Das Laufzeitverhalten von Anwendungen, welche in die Datenbank deployed wurden, hat sich allerdings als so langsam erwiesen, als dass man damit kaum produktiv arbeiten kann. Es zeichnen sich jedoch bereits heute Alternativen ab, die dieses Problem lösen können, was der Visibroker als CORBA-ORB beweist und zukünftig OC4J als EJB-Container beweisen wird.
Die praktischen Erfahrung bei der Erstellung der Beispielanwendung „EDOS“ hat eindeutig gezeigt, dass sowohl das Erzeugen der Business Components als auch das anschließende Deployment nicht den Großteil der Zeit in Anspruch genommen haben. Als wesentlich aufwendiger hat sich die Entwicklung des Clients in Form einer Swing-Anwendung erwiesen. An dieser Stelle wurde deutlich, dass sich das DACF-Framework doch in einem verhältnismäßig unausgereiften Zustand befindet. Oracle selber spricht in diesem Zusammenhang davon, dass die zukünftigen Versionen des JDevelopers in diesem Bereich ein anderes Verfahren anbieten werden.
In diesem Sinne bleibt abzuwarten, was die Weiterentwicklungen des JDevelopers tatsächlich bringen werden. Grundsätzlich läßt sich jedoch sagen, dass das Framework als solches sehr gut in diese Entwicklungsumgebung integriert wurde, was sich in einer äußerst komfortablen Bedienung niederschlägt.
Ein solches Werkzeug in Verbindung mit den im Rahmen dieser Arbeit gezeigten Komponentenmodelle läßt die Vermutung zu, dass diese Kombination aus Werkzeug und Komponentenmodell auch zukünftig im Bereich großer industrieller Anwendungen eine äußerst wichtige Rolle einnehmen wird.

Im Abschluss an diese Diplomarbeit möchte ich mich bei der OPITZ CONSULTING GmbH für die nette Unterstützung während dieser Zeit bedanken.

# 7 Literaturverzeichnis

[Brose, 2001] Brose Gerald, Vogel Andreas, Keith Duddy, *Java Programming with CORBA*, Wiley & Sons inc., 2001, Third Edition.

[BorlandV33, 2000] Inprise Corporation Inc, *Visibroker 3.3/3.4 Programmer's Guide, http://www.borland.com/techpubs/books/vbj/vbj33/index.html*.

[BorlandV45, 2000] Inprise Corporation Inc, *Visibroker 4.5 Programmer's Guide, http://www.borland.com/techpubs/books/vbj/vbj45/framesetindex.html*.

[Denninger, 2000] Denninger Stefan, Peters Ingo, *Enterprise JavaBeans*, Addison Wesley, 2000.

[Gruhn, 2000] Gruhn Volker, Thiel Andreas, *Komponentenmodelle*, Addison Wesley, 2000.

[Linnhoff-Popien, 1998] Linnhoff-Popien Claudia, *CORBA Kommunikation und Management*, Springer 1998

[Monson-Haefel, 2001] Monson-Haefel Richard, *Enterprise JavaBeans*, O'Reilly 2001, 2.Auflage.

[OracleCORBA, 2000] Oracle Corporation, *Oracle8i CORBA Developer's Guide and ReferenceRelease 3 (8.1.7)*, 2000.

[OracleEJB, 2000] Oracle Corporation, *Oracle8i Enterprise JavaBeans Developer's Guide and ReferenceRelease 3 (8.1.7)*, 2000.

[OracleJava, 2000] Oracle Corporation, *Oracle8i Java Developer's Guide Release 3 (8.1.7)*, 2000.

[OracleTools, 2000] Oracle Corporation, *Oracle8i Java Tools Reference Release 3 (8.1.7)*, 2000.

[OracleJSP, 2000] Oracle Corporation, *Oracle JavaServer Pages Developer's Guide and Reference Release 8.1.7*, 2000.

[Sayegh, 1999] Sayegh Andreas, *CORBA Standard, Spezifikation, Entwicklung*, O'Reilly 1999, 2.Auflage.

[SUN J2EE Spezifikation, 2001] The Java 2 Platform, Enterprise Edition (J2EE), *http://java.sun.com/j2ee*.

[SUN J2EE-Blueprints, 2001] The Java 2 Platform, Enterprise Edition Blueprints, *http://java.sun.com/j2ee/blueprints*.

# 8 Anhang

## 8.1 SQL-Quellcodes zur Erzeugung der Tabellen

```
CREATE TABLE BESTELLUNG
 (BEST_ID NUMBER(11)                    NOT NULL
 ,BEST_KUND_ID NUMBER(11)               NOT NULL
 ,BEST_VORGANGSNUMMER VARCHAR2(240)     NOT NULL
 ,BEST_DATUM DATE                       NOT NULL
 ,BEST_STATUS VARCHAR2(240)             NOT NULL
 )
/

CREATE TABLE KUNDE
 (KUND_ID NUMBER(11)                    NOT NULL
 ,KUND_VORNAME VARCHAR2(240)            NOT NULL
 ,KUND_NACHNAME VARCHAR2(240)           NOT NULL
 ,KUND_STRASSE VARCHAR2(240)            NOT NULL
 ,KUND_PLZ VARCHAR2(240)                NOT NULL
 ,KUND_ORT VARCHAR2(240)                NOT NULL
 ,KUND_TELEFON VARCHAR2(240)            NOT NULL
 ,KUND_WEBSITE VARCHAR2(240)
 ,KUND_EMAIL VARCHAR2(240)
 ,KUND_STATUS VARCHAR2(240)             NOT NULL
 )
/

CREATE TABLE POSTEN
 (POST_ID NUMBER(11)                    NOT NULL
 ,POST_BEST_ID NUMBER(11)               NOT NULL
 ,POST_PROD_ID NUMBER(11)               NOT NULL
 ,POST_STATUS VARCHAR2(240)             NOT NULL
 ,POST_MENGE NUMBER(11)                 NOT NULL
 ,POST_RABATT NUMBER(5,3)               NOT NULL
 ,POST_PREIS NUMBER(5,3)                NOT NULL
 )
/

CREATE TABLE PRODUKT_KATEGORIE
 (PKAT_ID NUMBER(11)                    NOT NULL
 ,PKAT_NAME VARCHAR2(240)               NOT NULL
 ,PKAT_BESCHREIBUNG VARCHAR2(1000)
 )
/

CREATE TABLE PRODUKT
 (PROD_ID NUMBER(11)                    NOT NULL
 ,PROD_ZULI_ID NUMBER(11)               NOT NULL
 ,PROD_PKAT_ID NUMBER(11)               NOT NULL
 ,PROD_NAME VARCHAR2(240)               NOT NULL
 ,PROD_BESCHREIBUNG VARCHAR2(1000)      NOT NULL
 ,PROD_PREIS NUMBER(8,2) DEFAULT 0      NOT NULL
 ,PROD_GUELTIG_VON DATE                 NOT NULL
 ,PROD_GUELTIG_BIS DATE                 NOT NULL
 )
/

CREATE TABLE ZULIEFERER
 (ZULI_ID NUMBER(11)                    NOT NULL
 ,ZULI_NAME VARCHAR2(240)               NOT NULL
 ,ZULI_STRASSE VARCHAR2(240)            NOT NULL
```

```
 ,ZULI_PLZ VARCHAR2(240)                NOT NULL
 ,ZULI_ORT VARCHAR2(240)                NOT NULL
 ,ZULI_TELEFON VARCHAR2(240)            NOT NULL
 ,ZULI_EMAIL VARCHAR2(240)
 ,ZULI_WEBSITE VARCHAR2(240)
 )
/
```

## 8.2 Exemplarische Quellcodes der Business Components

### 8.2.1 Das Source-File „MainApplet.java“

```
package de.edos.client.applet;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import oracle.dacf.dataset.*;
import oracle.dacf.dataset.connections.*;
import java.util.*;
import java.lang.reflect.*;

/**
 * Diese Klasse gehört zur Beispielanwendung "EDOS"
 * der Diplomarbeit zum Thema
 * -
 * Einsatzmöglichkeiten des "Business Components for Java Frameworks"
 * von Oracle im Umfeld von Enterprise JavaBeans- und CORBA-Architekturen
 * -
 * <P>
 * Sie bildet die Hauptklasse, welche den Rahmen der eigentlichen
 * Anwendung ausmacht. Ihre wichtigste Aufgabe besteht darin, die
 * SessionInfo Objekte für die ApplicationModules und die DACF-Masken zu
 * verwalten.
 * <P>
 * @author  Alexander Orbach (1101780613)
 * @version 1.0, 22/08/2001
*/
public class MainApplet extends JApplet {

    boolean isStandalone = false;
    JPanel jPanel1 = new JPanel();
    BorderLayout borderLayout1 = new BorderLayout();
    JMenuBar jMenuBar1 = new JMenuBar();
    JMenu jMenuSchluesselTabellen = new JMenu();
    JMenuItem jMenuItemKunde = new JMenuItem();
    JMenuItem jMenuItemZulieferer = new JMenuItem();
    JMenuItem jMenuItemProduktkategorien = new JMenuItem();
    JMenu jMenuVerwaltung = new JMenu();
    JMenuItem jMenuItemKundenbestellungen = new JMenuItem();
    JMenuItem jMenuItemGelieferteProdukte = new JMenuItem();
    JMenuItem jMenuItemProduktpalettenverwaltung = new JMenuItem();
    JMenu jMenuProgramm = new JMenu();
    JMenuItem jMenuItemUeber = new JMenuItem();
    JMenuItem jMenuItemBeenden = new JMenuItem();
    JTabbedPane jTabbedPane1 = new JTabbedPane();
    BorderLayout borderLayout2 = new BorderLayout();
    Vector visiblePanels = new Vector();

    private static Hashtable activeSessions = new Hashtable(10);
```

```
/**
 * getParameter
 * @param key
 * @param def
 * @return java.lang.String
 */
public String getParameter(String key, String def) {
    if (isStandalone) {
        return System.getProperty(key, def);
    }
    if (getParameter(key) != null) {
        return getParameter(key);
    }
    return def;
}

/**
 * Parameterloser Konstruktur
 */
public MainApplet() {
}

/**
 * Initialisierung des Applets
 */
public void init() {
    try  {
        jbInit();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

private void jbInit() throws Exception {
    this.getContentPane().setLayout(borderLayout1);
    this.setSize(new Dimension(448, 461));
    jPanel1.setLayout(borderLayout2);

    /*
    * JMenue initialisieren
    */
    jMenuSchluesselTabellen.setText("Schlüsseltabellen");
    jMenuItemKunde.setText("Kunde");
    jMenuItemKunde.addActionListener(new
      java.awt.event.ActionListener() {

        public void actionPerformed(ActionEvent e) {
            jMenuItemKunde_actionPerformed(e);
        }
    });
    jMenuItemZulieferer.setText("Zulieferer");
    jMenuItemZulieferer.addActionListener(new
      java.awt.event.ActionListener() {

        public void actionPerformed(ActionEvent e) {
            jMenuItemZulieferer_actionPerformed(e);
        }
    });
    jMenuItemProduktkategorien.setText("Produktkategorien");
    jMenuItemProduktkategorien.addActionListener(new
      java.awt.event.ActionListener() {

        public void actionPerformed(ActionEvent e) {
            jMenuItemProduktkategorien_actionPerformed(e);
        }
```

```
        });
        jMenuVerwaltung.setText("Verwaltung");
        jMenuItemKundenbestellungen.setText("Kundenbestellungen");
        jMenuItemKundenbestellungen.addActionListener(new
          java.awt.event.ActionListener() {

            public void actionPerformed(ActionEvent e) {
                jMenuItemKundenbestellungen_actionPerformed(e);
            }
        });
        jMenuItemGelieferteProdukte.setText("Gelieferte Produkte");
        jMenuItemGelieferteProdukte.addActionListener(new
          java.awt.event.ActionListener() {

            public void actionPerformed(ActionEvent e) {
                jMenuItemGelieferteProdukte_actionPerformed(e);
            }
        });

jMenuItemProduktpalettenverwaltung.setText("Produktpalettenverwaltung");
        jMenuItemProduktpalettenverwaltung.addActionListener(new
          java.awt.event.ActionListener() {

            public void actionPerformed(ActionEvent e) {
                jMenuItemProduktpalettenverwaltung_actionPerformed(e);
            }
        });
        jMenuProgramm.setText("Programm");
        jMenuItemUeber.setText("Über...");
        jMenuItemUeber.addActionListener(new
          java.awt.event.ActionListener() {

            public void actionPerformed(ActionEvent e) {
                jMenuItemUeber_actionPerformed(e);
            }
        });
        jMenuItemBeenden.setText("Beenden...");
        jMenuItemBeenden.addActionListener(new
          java.awt.event.ActionListener() {

            public void actionPerformed(ActionEvent e) {
                jMenuItemBeenden_actionPerformed(e);
            }
        });
        jTabbedPane1.setTabPlacement(JTabbedPane.BOTTOM);
        this.getContentPane().add(jPanel1, BorderLayout.CENTER);
        jPanel1.add(jTabbedPane1, BorderLayout.CENTER);
        jMenuBar1.add(jMenuSchluesselTabellen);
        jMenuBar1.add(jMenuVerwaltung);
        jMenuBar1.add(jMenuProgramm);
        jMenuSchluesselTabellen.add(jMenuItemKunde);
        jMenuSchluesselTabellen.add(jMenuItemZulieferer);
        jMenuSchluesselTabellen.add(jMenuItemProduktkategorien);
        jMenuVerwaltung.add(jMenuItemKundenbestellungen);
        jMenuVerwaltung.add(jMenuItemGelieferteProdukte);
        jMenuVerwaltung.add(jMenuItemProduktpalettenverwaltung);
        jMenuProgramm.add(jMenuItemUeber);
        jMenuProgramm.add(jMenuItemBeenden);
        this.setJMenuBar(jMenuBar1);
    }

    /**
     * start
     */
    public void start() {
    }
```

```
/**
 * stop
 */
public void stop() {
}

/**
 * destroy
 */
public void destroy() {
}

/**
 * getAppletInfo
 * @return java.lang.String
 */
public String getAppletInfo() {
    return "Applet Information";
}

/**
 * getParameterInfo
 * @return java.lang.String[][]
 */
public String[][] getParameterInfo() {
    return null;
}

/**
 * main Methode des Applets
 * @param args
 */
public static void main(String[] args) {

    //Sprache der Anwendung auf Deutsch setzen
    System.setProperty("jbo.default.language", "de");
    System.setProperty("jbo.default.country", "DE");

    //MainApplet instanzieren
    MainApplet applet = new MainApplet();
    applet.isStandalone = true;
    JFrame frame = new JFrame();
    frame.setTitle("EDOS - \"Easy Deployable Online Shop\"");
    frame.getContentPane().add(applet, BorderLayout.CENTER);
    applet.init();
    applet.start();
    frame.setSize(600, 620);
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation(
      (d.width - frame.getSize().width) / 2, (d.height -
      frame.getSize().height) / 2
    );
    frame.setVisible(true);
    frame.addWindowListener(
      new WindowAdapter() { public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
}
```

```
/*
* Hinzufügen neuer Panels zum JTabbedPane
*/
private void addNewPanelToTabbedPane(String panelName, String label){
    if(!this.visiblePanels.contains(panelName)){
        //Panel existiert noch nicht und muss daher instanziert werden
        JPanel newPanel = null;
        try{
            newPanel = (JPanel)Class.forName(panelName).newInstance();
        }catch(ClassNotFoundException cnfe){
        }catch(IllegalAccessException iae){
        }catch(InstantiationException ie){
        }
        this.jTabbedPane1.addTab(label,newPanel);
        int selectedComponentIndex =
          this.jTabbedPane1.indexOfTab(label);
         this.jTabbedPane1.setSelectedComponent(
        this.jTabbedPane1.getComponentAt(selectedComponentIndex));
        this.visiblePanels.add(panelName);
    }else{
        //Panel existiert bereits und wird in den Vordergrund geholt
        int selectedComponentIndex =
          this.jTabbedPane1.indexOfTab(label);
        Component cachedComponent =
          this.jTabbedPane1.getComponentAt(selectedComponentIndex);
        this.jTabbedPane1.setSelectedComponent(cachedComponent);
    }
}

/*
* Event Handling für die MenueItems
*/
void jMenuItemKunde_actionPerformed(ActionEvent e) {
  addNewPanelToTabbedPane(
    "de.edos.client.applet.KundePanel","Kunden"
  );
}

void jMenuItemZulieferer_actionPerformed(ActionEvent e) {
    addNewPanelToTabbedPane(
      "de.edos.client.applet.ZuliefererPanel","Zulieferer"
    );
}

void jMenuItemGelieferteProdukte_actionPerformed(ActionEvent e) {
    addNewPanelToTabbedPane(
      "de.edos.client.applet.ZuliefererProduktPanel","Gelieferte
      Produkte"
    );
}

void jMenuItemProduktkategorien_actionPerformed(ActionEvent e) {
    addNewPanelToTabbedPane(
     "de.edos.client.applet.ProduktKategoriePanel","Produktkategorien"
    );
}

void jMenuItemProduktpalettenverwaltung_actionPerformed(ActionEvent e)
{
    addNewPanelToTabbedPane(
      "de.edos.client.applet.KategorieProduktPanel",
      "Produktpalettenverwaltung"
    );
}
```

```
void jMenuItemKundenbestellungen_actionPerformed(ActionEvent e) {
    addNewPanelToTabbedPane(
      "de.edos.client.applet.KundeBestellungPanel",
      "Kundenbestellungen"
    );
}

void jMenuItemUeber_actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this,
    "EDOS - \"Easy Deployable Online Shop\"\n\n"+
    "Dieser Swing-Client soll beispielhaft
     das Laufzeitverhalten einer Anwendung\n"+
    "zeigen, dessen Geschäftslogik durch das
    \"Business Components for Java\" Framework (BC4J)\n"+
    "von Oracle entwickelt wurde. Diese Anwendung
   läuft im Rahmen einer 3-Schicht-Architektur\n"+
    "in Form von Enterprise JavaBeans oder CORBA-Objekten,
    welche auf der Basis von BC4J in die\n"+
    "Mittelschicht deployed wurden.\n\n"+
    "(c) Alexander Orbach 2001",
    "Über diesen Prototypen...",
    JOptionPane.INFORMATION_MESSAGE);
}

void jMenuItemBeenden_actionPerformed(ActionEvent e) {
    Object[] options = new Object[2];
    options[0] = "Ja";
    options[1] = "Nein";
    int z = JOptionPane.showOptionDialog(
        this,
        "Wollen Sie die Anwendung wirklich beenden?",
        "Beenden...",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        null);
    if(z==JOptionPane.YES_OPTION){
        System.exit(0);
    }
}

/*
* Die Methode prüft, ob für ein bestimmtes ApplicationModule bereits
* ein SessionInfo Objekt existiert. Für den Fall, dass ein SessionInfo
* Objekt bereits existiert, wird dessen Referenz zurückgegeben.
* Im anderen
* Fall wird ein neues erzeugt und entsprechend der Architektur
* (2- oder 3-Schicht) konfiguriert. In diesem Beispiel handelt es sich
* um eine 3-schichtige EJB-Architektur
*/
public static SessionInfo getSession
  (String appPackage, String appModule) {
    if (activeSessions.containsKey(appPackage+appModule)) {
        return (SessionInfo)activeSessions.get(appPackage+appModule);
    }
    else {
        SessionInfo sessionInfo = new SessionInfo();
        sessionInfo.setAppModuleInfo(
          new ModuleInfo(appPackage, appModule)
        );

        //Instanzieren der EJBConnection
        EjbConnection ejbConn = new EjbConnection(
          "AOREJB_IIOP",
          "/test/aorejb/ejb"
```

```
            );
            ejbConn.setCustomConnection(
              new MyCustomEJBSessionBeanConnection()
            );
            sessionInfo.setConnectionInfo(ejbConn);
            sessionInfo.setName("Session1");

            activeSessions.put(appPackage+appModule,sessionInfo);
            return sessionInfo;
        }
    }

    /*
    * Zerstört ein SessionInfo Objekt zu einem bestimmten
    * ApplicationModule
    */
    public static void removeSession(String appPackage, String appModule)
    {
        if (activeSessions.containsKey(appPackage+appModule)) {

((SessionInfo)activeSessions.get(appPackage+appModule)).revokeSession();

((SessionInfo)activeSessions.get(appPackage+appModule)).close();
            activeSessions.remove(appPackage+appModule);
        }
    }
}
```

### 8.2.2 Das XML-File „Kunde.xml“

```
<?xml version="1.0" encoding='WINDOWS-1252'?>
<!DOCTYPE Entity SYSTEM "jbo_03_01.dtd">

<Entity
   Name="Kunde"
   DBObjectType="table"
   DBObjectName="KUNDE"
   AliasName="Kunde"
   BindingStyle="Oracle"
   CodeGenFlag="5"
   RowClass="de.edos.bc4j.KundeImpl" >
   <DesignTime>
      <Attr Name="_isCodegen" Value="true" />
      <AttrArray Name="_publishEvents">
      </AttrArray>
   </DesignTime>
   <Attribute
      Name="KundId"
      Type="oracle.jbo.domain.Number"
      ColumnName="KUND_ID"
      ColumnType="NUMBER"
      SQLType="NUMERIC"
      IsNotNull="true"
      Precision="11"
      Scale="0"
      TableName="KUNDE"
      PrimaryKey="true" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundVorname"
      Type="java.lang.String"
      ColumnName="KUND_VORNAME"
      ColumnType="VARCHAR2"
```

```
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundNachname"
      Type="java.lang.String"
      ColumnName="KUND_NACHNAME"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundStrasse"
      Type="java.lang.String"
      ColumnName="KUND_STRASSE"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundPlz"
      Type="java.lang.String"
      ColumnName="KUND_PLZ"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundOrt"
      Type="java.lang.String"
      ColumnName="KUND_ORT"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundTelefon"
      Type="java.lang.String"
      ColumnName="KUND_TELEFON"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
```

```
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundWebsite"
      Type="java.lang.String"
      ColumnName="KUND_WEBSITE"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundEmail"
      Type="java.lang.String"
      ColumnName="KUND_EMAIL"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <Attribute
      Name="KundStatus"
      Type="java.lang.String"
      ColumnName="KUND_STATUS"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      IsNotNull="true"
      Precision="240"
      TableName="KUNDE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="240" />
      </DesignTime>
   </Attribute>
   <AccessorAttribute
      Name="Bestellung"
      Association="de.edos.bc4j.BestKundFkAssoc"
      AssociationEnd="de.edos.bc4j.BestKundFkAssoc.Bestellung"
      AssociationOtherEnd="de.edos.bc4j.BestKundFkAssoc.Kunde"
      Type="oracle.jbo.RowIterator"
      IsUpdateable="false" >
   </AccessorAttribute>
   <Key
      Name="KundPk" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundId" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="KUND_PK" />
         <Attr Name="_isPrimary" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010635" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundId" />
```

```
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010635" />
         <Attr Name="_checkCondition" Value="&#34;KUND_ID&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010636" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundVorname" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010636" />
         <Attr Name="_checkCondition" Value="&#34;KUND_VORNAME&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010637" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundNachname" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010637" />
         <Attr Name="_checkCondition" Value="&#34;KUND_NACHNAME&#34; IS
NOT NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010638" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundStrasse" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010638" />
         <Attr Name="_checkCondition" Value="&#34;KUND_STRASSE&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010639" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundPlz" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010639" />
         <Attr Name="_checkCondition" Value="&#34;KUND_PLZ&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010640" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundOrt" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010640" />
         <Attr Name="_checkCondition" Value="&#34;KUND_ORT&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
```

```
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010641" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundTelefon" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010641" />
         <Attr Name="_checkCondition" Value="&#34;KUND_TELEFON&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
   <Key
      Name="SysC0010642" >
      <AttrArray Name="Attributes">
         <Item Value="de.edos.bc4j.Kunde.KundStatus" />
      </AttrArray>
      <DesignTime>
         <Attr Name="_DBObjectName" Value="SYS_C0010642" />
         <Attr Name="_checkCondition" Value="&#34;KUND_STATUS&#34; IS NOT
NULL" />
         <Attr Name="_isCheck" Value="true" />
      </DesignTime>
   </Key>
</Entity>
```

### 8.2.3 Das Source-File „KundeImpl.java“

```
package de.edos.bc4j;

import oracle.jbo.server.*;
import oracle.jbo.RowIterator;
import oracle.jbo.domain.Number;
import oracle.jbo.Key;
import oracle.jbo.server.util.*;
import oracle.jbo.AttributeList;

public class KundeImpl extends oracle.jbo.server.EntityImpl {
    protected static final int KUNDID = 0;
    protected static final int KUNDVORNAME = 1;
    protected static final int KUNDNACHNAME = 2;
    protected static final int KUNDSTRASSE = 3;
    protected static final int KUNDPLZ = 4;
    protected static final int KUNDORT = 5;
    protected static final int KUNDTELEFON = 6;
    protected static final int KUNDWEBSITE = 7;
    protected static final int KUNDEMAIL = 8;
    protected static final int KUNDSTATUS = 9;
    protected static final int BESTELLUNG = 10;

    private static EntityDefImpl mDefinitionObject;
    /**
    * This is the default constructor (do not remove)
    */
    public KundeImpl() {
    }

    /**
    * Retrieves the definition object for this instance class.
    */
    public static synchronized EntityDefImpl getDefinitionObject() {
        if (mDefinitionObject == null) {
            mDefinitionObject =
(EntityDefImpl)EntityDefImpl.findDefObject("de.edos.bc4j.Kunde");
```

```
        }
        return mDefinitionObject;
    }

    /**
    * Gets the attribute value for KundId, using the alias name KundId
    */
    public Number getKundId() {
        return (Number)getAttributeInternal(KUNDID);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundId
    */
    public void setKundId(Number value) {
        setAttributeInternal(KUNDID, value);
    }

    /**
    * Gets the attribute value for KundVorname, using the alias name
KundVorname
    */
    public String getKundVorname() {
        return (String)getAttributeInternal(KUNDVORNAME);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundVorname
    */
    public void setKundVorname(String value) {
        setAttributeInternal(KUNDVORNAME, value);
    }


    /**
    * Gets the attribute value for KundNachname, using the alias name
KundNachname
    */
    public String getKundNachname() {
        return (String)getAttributeInternal(KUNDNACHNAME);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundNachname
    */
    public void setKundNachname(String value) {
        setAttributeInternal(KUNDNACHNAME, value);
    }

    /**
    * Gets the attribute value for KundStrasse, using the alias name
KundStrasse
    */
    public String getKundStrasse() {
        return (String)getAttributeInternal(KUNDSTRASSE);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundStrasse
    */
    public void setKundStrasse(String value) {
        setAttributeInternal(KUNDSTRASSE, value);
    }

    /**
    * Gets the attribute value for KundPlz, using the alias name KundPlz
```

```
    */
    public String getKundPlz() {
        return (String)getAttributeInternal(KUNDPLZ);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundPlz
    */
    public void setKundPlz(String value) {
        setAttributeInternal(KUNDPLZ, value);
    }

    /**
    * Gets the attribute value for KundOrt, using the alias name KundOrt
    */
    public String getKundOrt() {
        return (String)getAttributeInternal(KUNDORT);
    }
    /**
    * Sets <code>value</code> as the attribute value for KundOrt
    */
    public void setKundOrt(String value) {
        setAttributeInternal(KUNDORT, value);
    }


    /**
    * Gets the attribute value for KundTelefon, using the alias name
KundTelefon
    */
    public String getKundTelefon() {
        return (String)getAttributeInternal(KUNDTELEFON);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundTelefon
    */
    public void setKundTelefon(String value) {
        setAttributeInternal(KUNDTELEFON, value);
    }

    /**
    * Gets the attribute value for KundWebsite, using the alias name
KundWebsite
    */
    public String getKundWebsite() {
        return (String)getAttributeInternal(KUNDWEBSITE);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundWebsite
    */
    public void setKundWebsite(String value) {
        setAttributeInternal(KUNDWEBSITE, value);
    }

    /**
    * Gets the attribute value for KundEmail, using the alias name
KundEmail
    */
    public String getKundEmail() {
        return (String)getAttributeInternal(KUNDEMAIL);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundEmail
```

```
    */
    public void setKundEmail(String value) {
        setAttributeInternal(KUNDEMAIL, value);
    }

    /**
    * Gets the attribute value for KundStatus, using the alias name
KundStatus
    */
    public String getKundStatus() {
        return (String)getAttributeInternal(KUNDSTATUS);
    }

    /**
    * Sets <code>value</code> as the attribute value for KundStatus
    */
    public void setKundStatus(String value) {
        setAttributeInternal(KUNDSTATUS, value);
    }

    /**
    * Gets the associated entity oracle.jbo.RowIterator
    */
    public oracle.jbo.RowIterator getBestellung() {
        return (oracle.jbo.RowIterator)getAttributeInternal(BESTELLUNG);
    }

    /**
    * Add attribute defaulting logic here.
    */

    // Diese Methode wurde editiert, um beim Anlegen eines neuen
    // Datensatzes die Id des neuen Kunden aus einer Datenbank Sequence zu
    // ermitteln.
    public void create(AttributeList attributeList) {
        super.create(attributeList);
        SequenceImpl s = new SequenceImpl("kund_seq",
this.getDBTransaction());
        Integer next = (Integer)s.getData();
        this.setKundId(new Number(next.intValue()));
    }

    /**
    * Creates a Key object based on given key constituents
    */
    public static Key createPrimaryKey(Number kundId) {
        return new Key(new Object[]{kundId});
    }

}
```

### 8.2.4 Das XML-File „KundeView.xml“

```
<?xml version="1.0" encoding='WINDOWS-1252'?>
<!DOCTYPE ViewObject SYSTEM "jbo_03_01.dtd">

<ViewObject
   Name="KundeView"
   SelectList="Kunde.KUND_ID,
       Kunde.KUND_VORNAME,
       Kunde.KUND_NACHNAME,

       Kunde.KUND_STRASSE,
       Kunde.KUND_PLZ,
       Kunde.KUND_ORT,
```

```
    Kunde.KUND_TELEFON,
    Kunde.KUND_WEBSITE,

Kunde.KUND_EMAIL,
    Kunde.KUND_STATUS"
FromList="KUNDE Kunde"
BindingStyle="Oracle"
CustomQuery="false"
ComponentClass="de.edos.bc4j.KundeViewImpl" >
<DesignTime>
   <Attr Name="_codeGenFlag" Value="20" />
</DesignTime>
<EntityUsage
   Name="Kunde"
   Entity="de.edos.bc4j.Kunde" >
   <DesignTime>
      <Attr Name="_ReadOnly" Value="false" />
      <Attr Name="_EntireObjectTable" Value="false" />
      <Attr Name="_queryClause" Value="false" />
   </DesignTime>
</EntityUsage>
<ViewAttribute
   Name="KundId"
   EntityAttrName="KundId"
   EntityUsage="Kunde"
   AliasName="KUND_ID" >
   <DesignTime>
      <Attr Name="_DisplaySize" Value="0" />
   </DesignTime>
</ViewAttribute>
<ViewAttribute
   Name="KundVorname"
   EntityAttrName="KundVorname"
   EntityUsage="Kunde"
   AliasName="KUND_VORNAME" >
   <DesignTime>
      <Attr Name="_DisplaySize" Value="0" />
   </DesignTime>
</ViewAttribute>
<ViewAttribute
   Name="KundNachname"
   EntityAttrName="KundNachname"
   EntityUsage="Kunde"
   AliasName="KUND_NACHNAME" >
   <DesignTime>
      <Attr Name="_DisplaySize" Value="0" />
   </DesignTime>
</ViewAttribute>
<ViewAttribute
   Name="KundStrasse"
   EntityAttrName="KundStrasse"
   EntityUsage="Kunde"
   AliasName="KUND_STRASSE" >
   <DesignTime>
      <Attr Name="_DisplaySize" Value="0" />
   </DesignTime>
</ViewAttribute>
<ViewAttribute
   Name="KundPlz"
   EntityAttrName="KundPlz"
   EntityUsage="Kunde"
   AliasName="KUND_PLZ" >
   <DesignTime>
      <Attr Name="_DisplaySize" Value="0" />
   </DesignTime>
</ViewAttribute>
```

```
   <ViewAttribute
      Name="KundOrt"
      EntityAttrName="KundOrt"
      EntityUsage="Kunde"
      AliasName="KUND_ORT" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </ViewAttribute>
   <ViewAttribute
      Name="KundTelefon"
      EntityAttrName="KundTelefon"
      EntityUsage="Kunde"
      AliasName="KUND_TELEFON" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </ViewAttribute>
   <ViewAttribute
      Name="KundWebsite"
      EntityAttrName="KundWebsite"
      EntityUsage="Kunde"
      AliasName="KUND_WEBSITE" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </ViewAttribute>
   <ViewAttribute
      Name="KundEmail"
      EntityAttrName="KundEmail"
      EntityUsage="Kunde"
      AliasName="KUND_EMAIL" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </ViewAttribute>
   <ViewAttribute
      Name="KundStatus"
      EntityAttrName="KundStatus"
      EntityUsage="Kunde"
      AliasName="KUND_STATUS" >
      <DesignTime>
         <Attr Name="_DisplaySize" Value="0" />
      </DesignTime>
   </ViewAttribute>
   <ViewLinkAccessor
      Name="BestellungView"
      ViewLink="de.edos.bc4j.BestKundFkLink"
      Type="oracle.jbo.RowIterator"
      IsUpdateable="false" >
   </ViewLinkAccessor>
</ViewObject>
```

### 8.2.5 Das Source -File „KundeViewImpl.java“

```
package de.edos.bc4j;

import oracle.jbo.server.*;
import oracle.jbo.RowIterator;

public class KundeViewImpl extends oracle.jbo.server.ViewObjectImpl {

    /**
    * This is the default constructor (do not remove)
    */
    public KundeViewImpl() {
```

```
    }
}
```

### 8.2.6 Das XML-File „EDOSModule.xml“

```
<?xml version="1.0" encoding='WINDOWS-1252'?>
<!DOCTYPE AppModule SYSTEM "jbo_03_01.dtd">

<AppModule
   Name="EDOSModule"
   ComponentClass="de.edos.bc4j.EDOSModuleImpl" >
   <DesignTime>
      <Attr Name="_isCodegen" Value="true" />
      <Attr Name="_deployType" Value="2" />
      <Attr Name="_exportName" Value="EDOSModule" />
   </DesignTime>
   <ViewUsage
      Name="ZuliefererViewMaster"
      ViewObjectName="de.edos.bc4j.ZuliefererView" >
   </ViewUsage>
   <ViewUsage
      Name="ProduktViewDetail"
      ViewObjectName="de.edos.bc4j.ProduktView" >
   </ViewUsage>
   <ViewUsage
      Name="ProduktKategorieViewMaster"
      ViewObjectName="de.edos.bc4j.ProduktKategorieView" >
   </ViewUsage>
   <ViewUsage
      Name="KundeViewMaster"
      ViewObjectName="de.edos.bc4j.KundeView" >
   </ViewUsage>
   <ViewUsage
      Name="BestellungViewDetail"
      ViewObjectName="de.edos.bc4j.BestellungView" >
   </ViewUsage>
   <ViewUsage
      Name="PostenViewDetailOfBestellung"
      ViewObjectName="de.edos.bc4j.PostenView" >
   </ViewUsage>
   <ViewLinkUsage
      Name="ProdZuliFkLink"
      ViewLinkObjectName="de.edos.bc4j.ProdZuliFkLink"
      SrcViewUsageName="de.edos.bc4j.EDOSModule.ZuliefererViewMaster"
      DstViewUsageName="de.edos.bc4j.EDOSModule.ProduktViewDetail" >
      <DesignTime>
         <Attr Name="_isCodegen" Value="true" />
      </DesignTime>
   </ViewLinkUsage>
   <ViewLinkUsage
      Name="ProdPkatFkLink"
      ViewLinkObjectName="de.edos.bc4j.ProdPkatFkLink"

SrcViewUsageName="de.edos.bc4j.EDOSModule.ProduktKategorieViewMaster"
      DstViewUsageName="de.edos.bc4j.EDOSModule.ProduktViewDetail" >
      <DesignTime>
         <Attr Name="_isCodegen" Value="true" />
      </DesignTime>
   </ViewLinkUsage>
   <ViewLinkUsage
      Name="BestKundFkLink"
      ViewLinkObjectName="de.edos.bc4j.BestKundFkLink"
      SrcViewUsageName="de.edos.bc4j.EDOSModule.KundeViewMaster"
      DstViewUsageName="de.edos.bc4j.EDOSModule.BestellungViewDetail" >
      <DesignTime>
         <Attr Name="_isCodegen" Value="true" />
```

```
      </DesignTime>
   </ViewLinkUsage>
   <ViewLinkUsage
      Name="PostBestFkLink"
      ViewLinkObjectName="de.edos.bc4j.PostBestFkLink"
      SrcViewUsageName="de.edos.bc4j.EDOSModule.BestellungViewDetail"

DstViewUsageName="de.edos.bc4j.EDOSModule.PostenViewDetailOfBestellung" >
      <DesignTime>
         <Attr Name="_isCodegen" Value="true" />
      </DesignTime>
   </ViewLinkUsage>
   <Remote
      Name="O8"
      ServerClassName="de.edos.bc4j.server.o8.EDOSModuleServerO8" >
      <DesignTime>
         <Attr Name="_profileName" Value="bcdeploy\EDOSModuleServerO8.prf"
/>
      </DesignTime>
   </Remote>
</AppModule>
```

### 8.2.7 Das Source-File „EDOSModuleImpl.java“

```
package de.edos.bc4j;

import oracle.jbo.server.*;
import oracle.jbo.ViewObject;

public class EDOSModuleImpl extends
oracle.jbo.server.ApplicationModuleImpl {

    protected ZuliefererViewImpl ZuliefererViewMaster;
    protected ProduktViewImpl ProduktViewDetail;
    protected ProduktKategorieViewImpl ProduktKategorieViewMaster;
    protected KundeViewImpl KundeViewMaster;
    protected BestellungViewImpl BestellungViewDetail;
    protected PostenViewImpl PostenViewDetailOfBestellung;
    protected ViewLinkImpl ProdZuliFkLink;
    protected ViewLinkImpl ProdPkatFkLink;
    protected ViewLinkImpl BestKundFkLink;
    protected ViewLinkImpl PostBestFkLink;
    /**
    * This is the default constructor (do not remove)
    */
    public EDOSModuleImpl() {
    }

    /**
    * Container's getter for ZuliefererViewMaster
    */
    public ZuliefererViewImpl getZuliefererViewMaster() {
        if (ZuliefererViewMaster == null) {
            ZuliefererViewMaster =
(ZuliefererViewImpl)findViewObject("ZuliefererViewMaster");
        }
        return ZuliefererViewMaster;
    }


    /**
    * Container's getter for ProduktViewDetail
    */
    public ProduktViewImpl getProduktViewDetail() {
        if (ProduktViewDetail == null) {
```

```
            ProduktViewDetail =
(ProduktViewImpl)findViewObject("ProduktViewDetail");
        }
        return ProduktViewDetail;
    }


    /**
    * Container's getter for ProduktKategorieViewMaster
    */
    public ProduktKategorieViewImpl getProduktKategorieViewMaster() {
        if (ProduktKategorieViewMaster == null) {
            ProduktKategorieViewMaster =
(ProduktKategorieViewImpl)findViewObject("ProduktKategorieViewMaster");
        }
        return ProduktKategorieViewMaster;
    }


    /**
    * Container's getter for KundeViewMaster
    */
    public KundeViewImpl getKundeViewMaster() {
        if (KundeViewMaster == null) {
            KundeViewMaster =
(KundeViewImpl)findViewObject("KundeViewMaster");
        }
        return KundeViewMaster;
    }


    /**
    * Container's getter for BestellungViewDetail
    */
    public BestellungViewImpl getBestellungViewDetail() {
        if (BestellungViewDetail == null) {
            BestellungViewDetail =
(BestellungViewImpl)findViewObject("BestellungViewDetail");
        }
        return BestellungViewDetail;
    }


    /**
    * Container's getter for PostenViewDetailOfBestellung
    */
    public PostenViewImpl getPostenViewDetailOfBestellung() {
        if (PostenViewDetailOfBestellung == null) {
            PostenViewDetailOfBestellung =
(PostenViewImpl)findViewObject("PostenViewDetailOfBestellung");
        }
        return PostenViewDetailOfBestellung;
    }


    /**
    * Container's getter for ProdZuliFkLink
    */
    public ViewLinkImpl getProdZuliFkLink() {
        if (ProdZuliFkLink == null) {
            ProdZuliFkLink = (ViewLinkImpl)findViewLink("ProdZuliFkLink");
        }
        return ProdZuliFkLink;
    }
```

```
    /**
    * Container's getter for ProdPkatFkLink
    */
    public ViewLinkImpl getProdPkatFkLink() {
        if (ProdPkatFkLink == null) {
            ProdPkatFkLink = (ViewLinkImpl)findViewLink("ProdPkatFkLink");
        }
        return ProdPkatFkLink;
    }


    /**
    * Container's getter for BestKundFkLink
    */
    public ViewLinkImpl getBestKundFkLink() {
        if (BestKundFkLink == null) {
            BestKundFkLink = (ViewLinkImpl)findViewLink("BestKundFkLink");
        }
        return BestKundFkLink;
    }


    /**
    * Container's getter for PostBestFkLink
    */
    public ViewLinkImpl getPostBestFkLink() {
        if (PostBestFkLink == null) {
            PostBestFkLink = (ViewLinkImpl)findViewLink("PostBestFkLink");
        }
        return PostBestFkLink;
    }


    /**
    * Sample main for debugging Business Components code using the tester.
    */
    public static void main(String[] args) {
        launchTester("de.edos.bc4j", /* package name */
      "EDOSModuleLocal" /* Configuration Name */);
    }

}
```

### 8.2.8 Deployment Profile für das „EDOSModule“ als CORBA Objekt

```
deployExcludedClasses=
deploySynonym=0
deployManifest=<auto generate>
deployPublishAs=/test/aor/de.edos.bc4j.EDOSModule
deployCorbaHelperName=oracle.jbo.common.remote.corba.RemoteApplicationModu
leHomeHelper
deployIncludedFilesString=
deployRepublish=1
deploySSL=0
deployJDBCConnection=AOR_JDBC
deployIncludedClasses=
deployDependencyAnalyzeAugmentedClasspath=0
deployIncludedLibraries=
deployGenerateScriptForSettingClasspath=0
deployResolver=
deployAbstract=0
deployIncludedSinglePackages=
deployExcludedSources=
deployGrants=
deployProfileFormatVersion=3.0
```

```
deployVerbose=1
deployAsJar=1
deployConnection=AOR_IIOP
deployIncludedPackages=de.edos.bc4j.server.o8
deployIncludedSources=latte:java//d:\java\diplomarbeit\edos_bc4j\sources\d
e\edos\bc4j\server\o8\EDOSModuleServerO8.java
deployExcludedSinglePackages=
deployExcludedPackages=oracle.jbo;oracle.jdbc;oracle.sql;oracle.aurora;jav
ax;org;de.edos.bc4j
deployCompressed=0
deployAutoInclude=0
deployIncludedArchives=D:\java\Diplomarbeit\edos_bc4j\classes\edos_bc4jCom
monO81.jar;D:\java\Diplomarbeit\edos_bc4j\classes\edos_bc4j1.jar
deployResolve=1
deployExcludedArchives=
deployCorbaServerName=de.edos.bc4j.server.o8.EDOSModuleServerO8
deployIncludeConnectionProperties=0
deployMainClass=
deployRedirectDiagnostics=1
deployArchive=D:\java\Diplomarbeit\edos_bc4j\classes\EDOSModuleServerO81.j
ar
deployType=3
```

# 9 Stichwortverzeichnis

# Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift