Untersuchungen zur effizienten Kommunikation in komponentenbasierten Client/Server-Systemen

Dissertation

der Fakultät für Informatik
der Eberhard-Karls-Universität zu Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. **Klaus Beschorner**
aus Reutlingen

Tübingen
2002

Tag der mündlichen Prüfung:	**17. Juli 2002**
Dekan:	**Prof. Dr. A. Zell**
1. Berichterstatter:	**Prof. Dr. W. Rosenstiel**
2. Berichterstatter:	**Prof. Dr. W. Spruth**

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Enterprise JavaBeans (EJB) [DeM01] ermöglichen die Erstellung von leistungsfähigen und mehrschichtigen Client/Server-Anwendungen auf Basis der Programmiersprache Java. Anwendungen werden dabei aus Server-Komponenten zusammengesetzt, die Basisdienste einer Laufzeitumgebung in Anspruch nehmen können. Die Laufzeitumgebung stellt dabei technische Notwendigkeiten, wie z.B. eine Kommunikationsinfrastruktur zwischen Komponenten und ihren Clients zur Verfügung und gewährleistet zusätzlich die Transaktionssicherheit von aufgerufenen Operationen. Dies entlastet Anwendungsentwickler von technischen Fragestellungen und ermöglicht die stärkere Fokussierung auf fachliche Probleme der Anwendungsdomäne. Die als Laufzeitumgebung benötigten Applikations-Server werden dabei zu einer der wichtigsten Technologie- und Integrationsplattformen in Unternehmen [Ale01]. Die *Giga Information Group* schätzt, daß der Markt für EJB-Applikations-Server von 585 Millionen Dollar im Jahr 1999 auf einen Umfang von 9 Milliarden Dollar im Jahr 2003 ansteigen wird [Gil00].

Eine grundlegende und wichtige Frage, die bei der Entwicklung von EJB-Anwendungen häufig kaum Beachtung findet, besteht darin, in welcher Form Daten zwischen Client und Server transportiert werden sollen. Während der Implementierung führt ein fehlendes Übertragungskonzept zu der Situation, daß ein Teil der Anwendungsentwickler kaum Überlegungen anstellt, wie Daten übertragen werden sollen und naheliegende Lösungen gewählt werden, die auf den komfortablen Mechanismen und Klassen der Java-Entwicklungsumgebung basieren. Dagegen investiert ein anderer Teil mehr Zeit in Überlegungen, wie die Datenübertragung für einen bestimmten Fall erfolgen soll. Daraus resultieren viele unterschiedliche Formen der Datenübertragung, die aus unterschiedlichen Motiven gewählt werden und die Verfolgung eines Gesamtzieles bei der Anwendungsentwicklung vereiteln. Mögliche Ziele sind z.B. die Optimierung der Anwendungsleistung oder die Reduktion des Entwicklungsaufwands. Der Entwicklungsaufwand ist jedoch erhöht, da jeder Anwendungsentwickler das Problem der Datenübertragung neu löst und umsetzt. Das fehlende Gesamtkonzept führt bei der Wartung und Erweiterung der Anwendung zu Problemen, da die unterschiedlichen Kommunikationsansätze erkannt, verstanden und angepaßt oder gar ersetzt werden müssen.

Laut Untersuchungen der *Standish Group* [The95] stellen unvollständige und geänderte Anforderungen mit am häufigsten die Ursache für Probleme in Software-Projekten dar. Dies kann zu kostenintensiven Verzögerungen oder gar zum Abbruch des Projekts führen. Eine nicht zu unterschätzende Anzahl von Anforderungsänderungen verursacht auf technischer Ebene eine Änderung von Datenart und -menge und stellt damit hohe Anforderungen an die Flexibilität

eines verwendeten Datenübertragungskonzepts. Als Konsequenz muß die Datenübertragung möglichst universell gestaltet werden, um auf Anforderungsänderungen zeitnah und mit geringem Aufwand reagieren zu können. Gleichzeitig muß allerdings eine feste Vorgehensweise eingeführt werden, um die Vielfalt möglicher Datenübertragungsmechanismen einzuschränken und den Anwendungsentwickler von Fragen, die damit zusammenhängen, zu entlasten.

1.1 Ziele

In dieser Arbeit sollen Fragen der Datenübertragung in EJB-Systemen untersucht werden. Dabei wird die Datenübertragung in unterschiedliche Teile aufgespalten und eine allgemeine Struktur für universelle Daten-Container entworfen, die in verschiedensten EJB-Architekturansätzen und mit unterschiedlichsten Java-Objekten, deren Daten zwischen Client und Server transportiert werden müssen, verwendet werden können. Die Konzeption der Daten-Container soll dabei derart erfolgen, daß auf deren Basis ein einheitliches Übertragungskonzept etabliert werden kann, das über alle Schichten einer mehrstufigen Client/Server-Anwendung hinweg Verwendung findet. Dabei sollen Anwendungsentwickler entlastet und implizit Entwurfsmuster verwendet werden, die sich in verteilten Systemen bewährt haben. Zusätzlich soll aufgezeigt werden, wie weitere Systemkomponenten aufgrund der universellen Container ebenfalls verallgemeinert und zentral bereitgestellt werden können, um zu verhindern, daß diese unter hohem Aufwand mehrfach für verschiedene Fälle implementiert werden müssen. Ein weiteres Kernziel der Datenübertragungskonzepte ist es, die Datenübertragung teilweise von der Anwendung in dem Sinne abzukoppeln, daß der komplette Übertragungsmechanismus nachträglich erweitert, konfiguriert oder gar komplett ausgetauscht werden kann. Dies sichert die nachträgliche Änderung der Datenübertragung und damit die Möglichkeit, auf geänderte oder neue Anforderungen, die sich auf die Datenübertragung auswirken, zu reagieren. Die Daten-Container selbst können bestehende Verfahren vollständig ersetzen oder diese mit den genannten Zielsetzungen erweitern. Im Rahmen dieser Arbeit werden prototypisch zentrale Teile von Daten-Containern entwickelt, die zur Veranschaulichung und Untersuchung der Konzepte dienen. Dabei soll als ein weiteres Ziel der Arbeit aufgedeckt werden, welche Einflußfaktoren bei der Datenübertragung in EJB-Systemen maßgeblich sind, da in der gängigen EJB-Literatur nur sehr wenige Aussagen darüber zu finden sind.

1.2 Aufbau

In Kapitel 2 werden zunächst die Grundlagen erläutert, die zum Verständnis dieser Arbeit und ihrem Umfeld notwendig sind. Dabei liegt der Schwerpunkt auf der Darstellung moderner Client/Server-Konzepte, die dem objektorientierten Paradigma folgen. Dazu gehören die *Common Object Request Broker Architecture (CORBA)* sowie die *Remote Method Invocation (RMI)*, um eine verteilte Kommunikation zwischen Objekten zu ermöglichen. Diese Standards bilden die Basis für das serverseitige Komponentenmodell *Enterprise JavaBeans (EJB)*, das den Schwerpunkt in dieser Arbeit bildet. Die darauf folgenden Kapitel spiegeln die Ziele dieser Arbeit wider. In Kapitel 4 werden universelle Datenübertragungskonzepte in Form von *Aktiven Daten-Containern* vorgestellt, die bestehende Datenübertragungsformen ersetzen und er-

weitern. Dabei werden auch die wesentlichen Einflußfaktoren, die bei der Datenübertragung maßgeblich sind, genannt. Deren Anwendung in unterschiedlichen EJB-Architekturen wird im darauf folgenden Kapitel erläutert. Sie zeigt, daß die Datenübertragungskonzepte unabhängig von der Anwendungsarchitektur eingesetzt werden können. In Kapitel 6 werden die erzielten Ergebnisse und der Einsatz der im Rahmen dieser Arbeit entwickelten Konzepte anhand eines Industrieprojekts erläutert. Zur Untersuchung der im Rahmen dieser Arbeit entwickelten Konzepte in unterschiedlichen Applikations-Servern wurden Anforderungen an ein Testwerkzeug formuliert und prototypisch implementiert. Anhang B ermöglicht einen Überblick über dieses Werkzeug.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für diese Arbeit wesentlichen Client/Server-Technologien beschrieben. Der Schwerpunkt liegt dabei auf *Enterprise JavaBeans*.

2.1 Client/Server-Systeme

Dieser Abschnitt beschreibt die wichtigsten Grundbegriffe im Client/Server-Umfeld, die für das Verständnis dieser Arbeit notwendig sind. Wichtige Begriffe wurden dabei aufgrund ihrer Prägnanz und Verbreitung im Englischen belassen. Ebenso wurden aus Spezifikationen entnommene Abbildungen im Englischen belassen.

2.1.1 Grundbegriffe

Das Client/Server-Modell unterscheidet zwischen zwei Komponentenarten in verteilten Systemen. Die Server-Komponente bietet Dienste zur Nutzung an. Die Client-Komponente nimmt diese Dienste in Anspruch. Die Inanspruchnahme von Diensten erfolgt durch das Stellen einer Anfrage mittels einer Nachricht über das Netzwerk an den Server. Nachdem der Server die Anfrage abgearbeitet hat, schickt er das Ergebnis an den Client zurück [Sha98]. Das Modell ist in Abbildung 2.1 schematisch dargestellt.
Bei der Abarbeitung von Anfragen in einem Client/Server-System spielt der Begriff *Transaktion* eine wesentliche Rolle. Nachfolgend wird dieser Begriff grundlegend erklärt. Für eine tiefere Auseinandersetzung mit diesem Thema kann z.B. auf [Gra93] zurückgegriffen werden.
Eine Transaktion faßt mehrere Arbeitsschritte zu einer Einheit zusammen und garantiert dafür die unter dem englischen Akronym ACID bekannten Eigenschaften [Rom02]:

- **Atomicity (Atomizität)** garantiert, daß mehrere Operationen nach dem Alles-oder-nichts-Prinzip ausgeführt werden. Die Operationen erscheinen so nach außen hin als eine Einheit, die nur erfolgreich beendet ist, wenn alle Einzeloperationen fehlerfrei durchgeführt werden. Tritt bei einer Operation eine Fehlersituation auf, werden alle bisher erfolgten Operationen rückgängig gemacht.

- **Consistency (Konsistenz)** garantiert, daß sich das System nach einer durchgeführten Transaktion in einem konsistenten Zustand befindet. Ein konsistenter Zustand wird durch

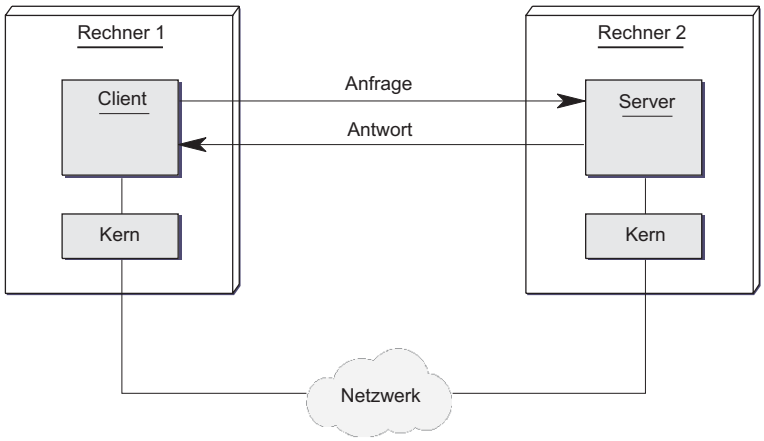


Abbildung 2.1: Client/Server-Modell

eine invariante Menge von Regeln definiert, die nach der Beendigung einer Transaktion erfüllt sein müssen. In einem Banksystem kann z.B. die Regel festgelegt sein, daß nach einer Transaktion alle beteiligten Kontostände positiv sein müssen. Während einer Transaktion können aufgrund der Aufteilung in mehrere Operationen temporär inkonsistente Zustände auftreten, die wegen der Atomizität nach außen hin nicht sichtbar sind.

- **Isolation (Isolation)** garantiert, daß nebenläufige Transaktionen voneinander getrennt ablaufen, d.h. die Zwischenzustände einer Transaktion können von keiner anderen Transaktion herangezogen werden. Es können sonst inkonsistente Zustände auftreten. Beim Zugriff auf gemeinsame Daten in einer Datenbank wird z.B. mittels Sperrung der Daten verhindert, daß andere Transaktionen auf diese Daten zugreifen können.

- **Duration (Dauerhaftigkeit)** garantiert, daß die Auswirkungen einer erfolgreich beendeten Transaktion nicht verloren gehen. Das Ergebnis einer Transaktion kann nur durch eine weitere Transaktion rückgängig gemacht werden. Die durch eine Transaktion herbeigeführte Zustandsänderung des Systems überdauert einen Systemabsturz oder einen Neustart des Systems.

Um die Isolation von Transaktionen untereinander zu gewährleisten, werden Sperrverfahren eingesetzt, die verhindern, daß bestimmte Ressourcen von mehreren Transaktionen gleichzeitig verwendet werden. Durch die Wartephasen auf gesperrte Ressourcen wird das Leistungsverhalten der Anwendung negativ beeinflußt. Um dies in dafür geeigneten Szenarien zu mildern, kann der Isolationsgrad (*Isolation Level*) zwischen Transaktionen beeinflußt werden. Dabei werden in unterschiedlichem Maße Inkonsistenzen hingenommen. Der Isolationsgrad läßt sich durch die folgenden Kriterien klassifizieren [Rom02]:

- **Dirty Read.** Eine Transaktion darf die Daten einer anderen, noch nicht beendeten Transaktion lesen.

- **Non-repeatable Read.** Eine Transaktion kann bei einer wiederholten Leseoperation andere Daten wie beim ersten Mal erhalten, falls zwischenzeitlich eine andere Transaktion Änderungen vorgenommen hat.
- **Phantom Read.** Eine Transaktion erhält bei einer weiteren Abfrage eine größere Treffermenge, weil eine andere Transaktion zwischenzeitlich neue Elemente eingefügt hat.

Isolierungsgrad	Dirty Read	Non-repeatable Read	Phantom Read	Beschreibung
Read Uncommitted	ja	ja	ja	Transaktionen können Datenänderungen von nicht beendeten Transaktionen lesen.
Read Committed	nein	ja	ja	Transaktionen können nur Datenänderungen von anderen, bereits beendeten Transaktionen lesen.
Repeatable Read	nein	nein	ja	Transaktionen können keine Daten ändern, die von anderen Transaktionen gelesen werden.
Serializable	nein	nein	nein	Transaktionen arbeiten nacheinander mit den Daten.

Tabelle 2.1: Isolationsgrade

Entsprechend den Kriterien lassen sich die Isolationsgrade in Tabelle 2.1 (in Anlehnung an [Rom02]) unterscheiden. Der Typ *Serializable* ist der strengste Isolierungsgrad und schließt dabei Konsistenzprobleme bei ggf. schlechterem Leistungsverhalten aus. Aufgrund der Tatsache, daß an einer Transaktion mehrere Ressourcen beteiligt sind, die auch in einem Netzwerk verteilt sein können, müssen verteilte Transaktionen stattfinden. Die ACID-Eigenschaften werden dabei durch das *Two-Phase Commit Protocol (2PC)* sichergestellt. Ein Transaktionskoordinator steuert anhand eines Protokolls mit den folgenden zwei Phasen den Ablauf einer Transaktion:

- **Phase 1.** Alle Transaktionsmanager der teilnehmenden Ressourcen werden abgefragt, ob sie die vorliegende Transaktion erfolgreich beenden können (*before commit*). Falls die Antwort einer Ressource negativ ausfällt, wird die komplette Transaktion abgebrochen und es finden keine Änderungen statt. Sonst wird die Transaktion durchgeführt und kann nur noch durch einen außergewöhnlichen Fehler unterbrochen werden.
- **Phase 2.** Diese Phase wird nur durchlaufen, wenn Phase 1 erfolgreich durchlaufen wird und weist alle teilnehmenden Transaktionsmanager an, die von der Transaktion geforderten Änderungen durchzuführen (*commit*).

Die technische Umsetzung eines verteilten Systems ist mit extrem hohem Aufwand verbunden, falls der Entwickler alle damit verbundenen Aufgaben selbst auf niedrigem Niveau implementieren muß [Emm00]. So erfordert z.B. die Umsetzung des Kommunikationsvorgangs eine detaillierte Auseinandersetzung mit Netzwerkprotokollen und Nachrichtenformaten. Um dies zu

verhindern, können Produkte verwendet werden, die unter dem Begriff *Middleware* zusammengefaßt werden. Die *Middleware* ist dabei zwischen den eigentlichen Anwendungskomponenten und dem Netzwerkbetriebssystem angesiedelt und bietet dem Entwickler eine Schnittstelle auf hohem Abstraktionsniveau, um ein verteiltes System zu entwickeln. Die *Middleware*-Produkte können dabei in die folgenden vier Kategorien eingeteilt werden [Emm00]:

- **Transaktionale Middleware:** Erlaubt die transparente Verteilung von Komponenten in heterogenen Netzwerken und die Durchführung von transaktionssicheren Operationen zwischen ihnen. Zur Realisierung von verteilten Transaktionen wird das 2PC-Protokoll eingesetzt. Durch Lastverteilungs- und Replikationsmechanismen können sehr leistungsfähige Systeme erstellt werden. Produkte, die zu dieser Kategorie gehören, sind z.B. *CICS* von *IBM* [IBMa], *Tuxedo* von *Bea* [Bea] und *Encina* von *Transarc* [IBMb].

- **Nachrichtenorientierte Middleware**[1]: Erlaubt den asynchronen Nachrichtenaustausch zwischen den Systemkomponenten. Dabei erfolgt eine starke Entkopplung zwischen den Komponenten, da nach der Übergabe einer Nachricht an die *Middleware* keine Blockierung des Aufrufers erfolgt. In diese Kategorie fällt z.B. *MQSeries* von *IBM*.

- **Prozedurale Middleware:** Erlaubt den Aufruf von entfernten Prozeduren zwischen Komponenten, die sich auf unterschiedlichen Rechnern befinden. Die verwendeten Parameter werden automatisch vor der Kommunikation verpackt und nach der Kommunikation entpackt (*Marshalling/Unmarshalling*). Diese *Middleware* wurde in Form des *Remote Procedure Calls (RPCs)* von *Sun Microsystems* entwickelt und ist Bestandteil der meisten UNIX-Systeme.

- **Objekt- und Komponenten-Middleware:** Erlaubt den Aufruf von entfernten Methoden zwischen verteilten Objekten und Komponenten. Vertreter dieser Art stellen eine konsequente Ausweitung des RPCs auf objektorientierte Paradigmen dar. Beispiele für solche Produkte sind die *Remote Method Invocation (RMI)*, die *Common Object Request Broker Architecture (CORBA)* und Implementierungen des Standards *Enterprise JavaBeans* von *Sun Microsystems*.

Aufgrund der wesentlichen Bedeutung von Objekt- und Komponentenorientierter *Middleware* für diese Arbeit, erfolgt im nächsten Abschnitt eine nähere Betrachtung der Grundideen dieser Technologien.

2.1.2 Modelle

Im Laufe der letzten Jahre haben sich objektorientierte Konzepte neben prozeduralen Konzepten in der Client/Server-Welt fest etabliert. Wie aus Abbildung 2.2 (in Anlehnung an [Orf98]) ersichtlich ist, wird in prozeduralen Konzepten eine entfernte Prozedur aufgerufen. Die Daten werden dabei separat vom Funktionscode gehalten.
Im Gegensatz dazu können mit einem objektorientierten Ansatz entfernte Methoden aufgerufen werden, die sich auf ein bestimmtes Objekt beziehen, das seine Daten kapselt. Abbildung 2.3

[1] *Message-oriented Middleware (MOM)*

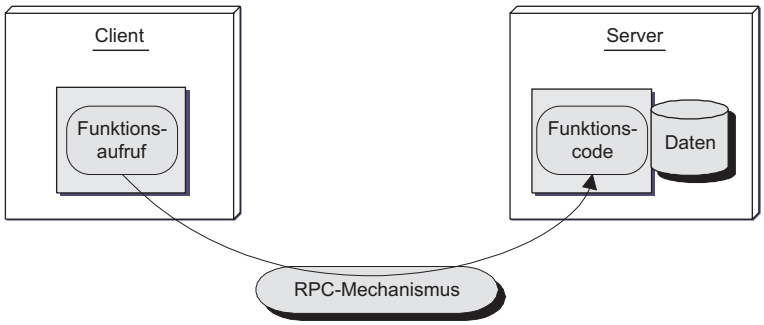


Abbildung 2.2: RPC-Mechanismus

(in Anlehnung an [Orf98]) stellt diesen Zusammenhang dar. Die gleiche Methode kann sich dabei auch bei unterschiedlichen Objekten verschieden verhalten (Polymorphismus). Allgemein werden objektorientierte Konzepte, wie Datenkapselung, Vererbung, Polymorphie und objektbasierte Ausnahmebehandlung in verteilten Systemen ermöglicht. Der Anwendungsentwickler wird hier signifikant entlastet, da er sich nicht mit Programmierdetails, die eine Kommunikation über das Netzwerk betreffen, auseinandersetzen muß. Ein entfernter Methodenaufruf sieht genauso wie ein lokaler Methodenaufruf aus und es ist nicht erforderlich die Daten zum Versand in ein spezielles Nachrichtenformat zu überführen. Zu den OO-Mechanismen gehören die in Abschnitt 2.2 vorgestellte *Common Object Request Broker Architecture (CORBA)* und die in Abschnitt 2.3 illustrierte *Remote Method Invocation (RMI)*.

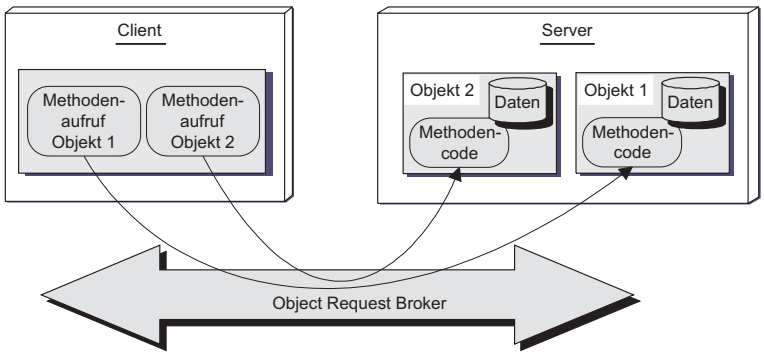


Abbildung 2.3: OO-Mechanismus

Obwohl die OO-Kommunikationsmechanismen eine leistungsfähige Kommunikationsinfrastruktur in der Client/Server-Welt bereitstellen, ergeben sich bei deren Anwendung eine Reihe von Problemen, die nach wie vor vom Anwendungsentwickler zu lösen sind [Sta00b]:

- **Lebenszyklusmanagement.** Objekte müssen zum richtigen Zeitpunkt erzeugt, aktiviert und deaktiviert werden.
- **Zugriff auf die Anwendungslogik.** Es muß festgelegt werden, wie der Zugriff auf die Logikschicht (*Middle-Tier*) der Anwendung stattfindet (vgl. folgender Abschnitt).

- **Sicherheit.** I.d.R. sind Sicherheitsanforderungen beim Zugriff auf die Anwendung zu erfüllen.

- **Installation und Administration.** Zur Bereitstellung und Pflege der Anwendung sind unterstützende Werkzeuge wünschenswert.

- **Zustandssicherung.** Der in verteilten Objekten vorhandene Zustand muß dauerhaft gespeichert werden und über Fehlervorfälle hinweg konsistent gehalten werden.

- **Plattformunabhängigkeit.** Der Anwendungsentwickler sollte unabhängig von der zugrundeliegenden Systemplattform entwickeln können.

- **Web-Anbindung.** Häufig muß eine Web-Anbindung der Anwendung erfolgen, die ebenfalls zu realisieren ist.

Die Lösung all dieser Probleme verursacht bei der Entwicklung einen signifikanten Aufwand, der in [Sta00b] auf mehr als 50% des Gesamtaufwandes geschätzt wird. Damit beschäftigt sich ein Entwickler hauptsächlich mit technischen Fragen, die sich mit der Bereitstellung einer Infrastruktur zum Ablauf der entstehenden Anwendung befassen. Zur einheitlichen und wiederverwendbaren Lösung der geschilderten Probleme wurden auf Basis der OO-Technologien serverseitige Komponententechnologien geschaffen. Grundidee dabei ist die Bereitstellung einer Laufzeitumgebung, die Basisdienste, wie z.B. Transaktionalität, Lebenszyklusmanagement und Sicherheit anbietet, die von Objekten genutzt werden können. Die Laufzeitumgebung wird auch als *Container* bezeichnet. Um die Auseinandersetzung mit den Diensten des Containers auf ein Minimum zu beschränken, müssen diese lediglich mit einem Deskriptor deklariert werden und nicht mehr explizit durch den Anwendungsentwickler implementiert werden. Somit steht eine fertige Infrastruktur bereit, die zur Realisierung von Applikationen verwendet werden kann und zu einer Zeiteinsparung in der Anwendungsentwicklung führt. Das Konzept ist in Abbildung 2.4 dargestellt.

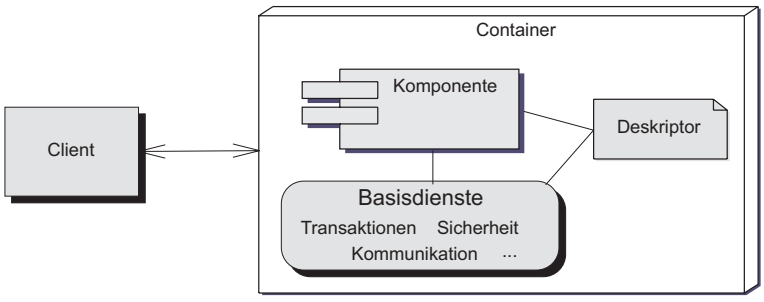


Abbildung 2.4: Komponententechnologie

Vertreter dieser Technologie sind *Enterprise JavaBeans (EJB)*, die den Schwerpunkt in dieser Arbeit bilden und in Abschnitt 2.5 erläutert werden. Alternative Umsetzungen sind *Microsoft Component Object Model+ (COM+)* und das *CORBA Component Model (CCM)*. Aufgrund der

unterschiedlichen technischen Realisierung des Komponentengedankens und den verschiedenen Standards, kann eine Interaktion zwischen Komponenten aus den unterschiedlichen Modellen nur über *Bridges* erfolgen, die zu einer komplexeren Architektur des Anwendungssystems führen [Göb00]. Ein sehr guter Gesamtüberblick über alle drei Technologien findet sich in [Sta00b]. Kompakte weiterführende Informationen zu COM+ finden sich z.B. in [Wey99] und zum CCM in [Sta00a, Sta01].
Mit der Einführung von verteilten, standardisierten Komponenten ist ebenfalls der Wunsch verbunden, qualitativ hochwertige Anwendungen in möglichst kurzer Zeit zu entwickeln. Dabei soll auf bewährte Komponenten von Drittherstellern zurückgegriffen werden, die innerhalb eines Projekts zu einem größeren Anwendungssystem zusammengesetzt werden. Dabei existieren jedoch u.a. die folgenden Probleme [Göb00]:

- **Funktionale Standards**. Es gibt keine Standards, die festlegen, welche Funktionalität eine Komponente umfassen muß. Es liegt im Ermessen der Komponentenhersteller, welche Funktionalität sie im Umfeld ihrer Kunden für sinnvoll halten. Aufgrund des Fehlens funktionaler Standards, werden solche Komponenten unterschiedlicher Herkunft nicht zusammenarbeiten, obwohl sie auf der selben Technologie basieren. Dies führt auch dazu, daß auf dem Markt eher sehr allgemein einsetzbare Komponenten, wie z.B. grafische Elemente für Benutzerschnittstellen oder Basiskomponenten, wie z.B. relationale Datenbanksysteme (die als große Komponenten angesehen werden können), angeboten werden.

- **Technische Standards**. Aufgrund des Fehlens eines einheitlichen Standards muß eine Auseinandersetzung mit mehreren Technologien erfolgen, die evtl. zur aufwendigen Implementierung von *Bridges* führt, um unterschiedliche Komponenten zusammenarbeiten zu lassen.

- **Auffinden von Komponenten**. Es ist ein großes Problem das Verhalten von Komponenten so zu beschreiben, daß sie vom Benutzer gefunden werden können. Vor dem Kauf einer Komponente muß klar sein, ob sie den Bedürfnissen des Kunden entspricht. D.h. sie muß das gewünschte Verhalten aufweisen und evtl. mit Komponenten anderer Hersteller zusammenpassen. Der Prozeß des Auffindens muß aus Produktivitätsgründen schneller durchführbar sein, als die Eigenproduktion einer äquivalenten Komponente.

Aufgrund der vorliegenden Probleme ist trotz der Existenz eines Rahmens durch die genannten Komponententechnologien in absehbarer Zeit nicht damit zu rechnen, daß eine solche Anwendungsentwicklung zum gewünschten Erfolg führt.

2.1.3 Architekturen

Unabhängig davon, ob eine Applikation verteilt ist oder nicht, besteht sie aus drei grundlegenden funktionalen Elementen, die in Abbildung 2.5 dargestellt sind [Sha98]. Das Präsentationselement dient der Interaktion mit dem Benutzer. Das Datenelement liefert die Persistenzmechanismen zum dauerhaften Erhalt von Daten, die in der Applikation bearbeitet werden und das Logikelement sorgt dafür, daß Benutzereingaben sowie Daten entsprechend dem Zweck der Applikation verarbeitet werden können.

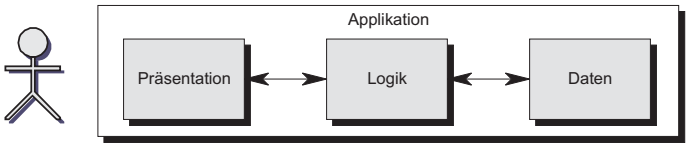


Abbildung 2.5: Elemente einer Applikation

Mehrschichtige Architekturen verfolgen grundsätzlich das Ziel, ähnliche Funktionalität oder Komponenten mit ähnlichen Eigenschaften in einer Schicht zusammenzufassen, die ihre Dienste umgebenden Schichten anbietet [Mye00]. Eine dreischichtige Architektur folgt einer benutzerorientierten Sichtweise, bei der sich die Präsentation, die Logik sowie die Datenhaltung jeweils in einer eigenen Schicht befinden. Zwischen den Schichten erfolgt die Kommunikation. In einer Client/Server-Anwendung können die in Abbildung 2.5 dargestellten Elemente einer Anwendung auf verschiedene Rechner verteilt werden. Dabei sind die folgenden Partitionierungen möglich (in Anlehnung an [Sha98]):

- **Verteilte Präsentation.** Client und Server teilen sich die Aufgaben, die zur Präsentation von Daten notwendig sind. Beispiele hierfür sind das System X-Window und Systeme, die zur Darstellung HTML in Web-Browsern verwenden.
- **Entfernte Präsentation.** Der Client enthält keinerlei Anwendungslogik, ist aber allein für die Präsentation verantwortlich. Beispiele hierfür sind Clients, die nur den Bildschirminhalt eines entfernten Rechners darstellen.
- **Verteilte Logik.** Die Anwendungslogik befindet sich teilweise im Server und teilweise im Client. Ein Beispiel hierfür sind Java-Applets.
- **Entfernter Datenzugriff.** Anwendungslogik und Präsentation befinden sich im Client, der auf Daten eines entfernten Servers zugreift. Beispiel hierfür sind entfernte Datenbankzugriffe über SQL.
- **Verteilte Daten.** Die Aufgaben der Datenhaltung werden zwischen dem Client und ein oder mehreren Servern aufgeteilt. Ein Beispiel hierfür ist die *Distributed Relational Database Architecture (DRDA)* von IBM.

Abbildung 2.6 (in Anlehnung an [Sha98]) faßt die fünf Arten der Partitionierung zusammen.
Mehrschichtige (*N-Tier*) Architekturen bilden die Basis für auf *Enterprise JavaBeans* basierende Anwendungen. Abbildung 2.7 [Dee01, Kas00] stellt die typische Architektur in diesem Umfeld dar. Die Präsentationsschicht besteht dabei aus der Schicht, in der sich die Anwendungs-Clients befinden und der passenden Gegenschicht auf dem Server, die Daten entsprechend den Erfordernissen des Clients aufbereitet. Die Logikschicht wird durch *Enterprise JavaBeans* repräsentiert, die wiederum mittels entsprechenden Konnektoren und Treibern mit Datenspeichern und Fremdsystemen kommunizieren.
Eine dienstbasierte Architektur faßt mehrere Operationen in einem Dienstobjekt zusammen, das in einer Dienstschicht plaziert wird, um Anfragen von Clients entgegenzunehmen [Tea00]. Der

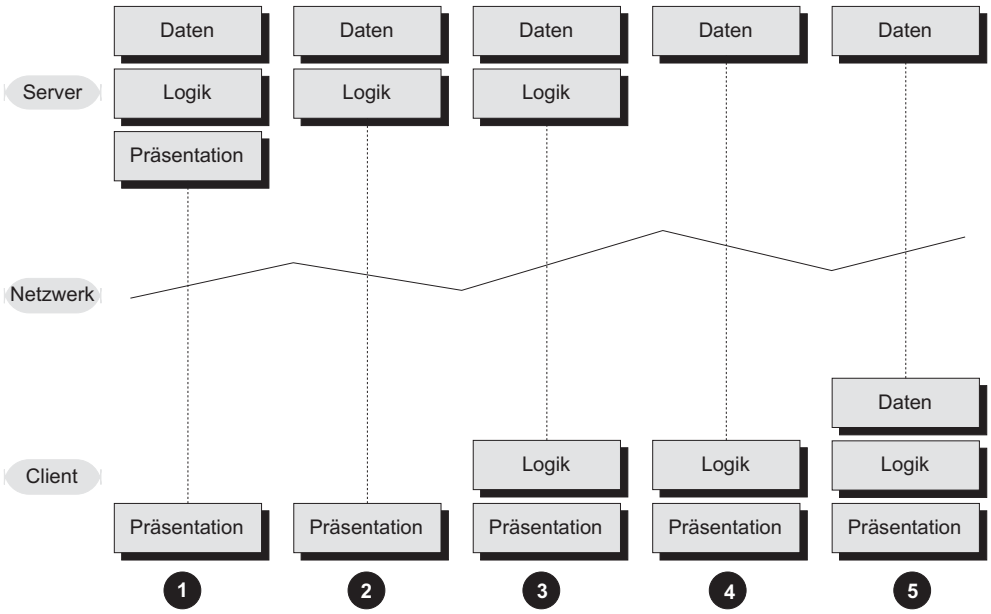


Abbildung 2.6: Partitionierung einer Client/Server-Applikation

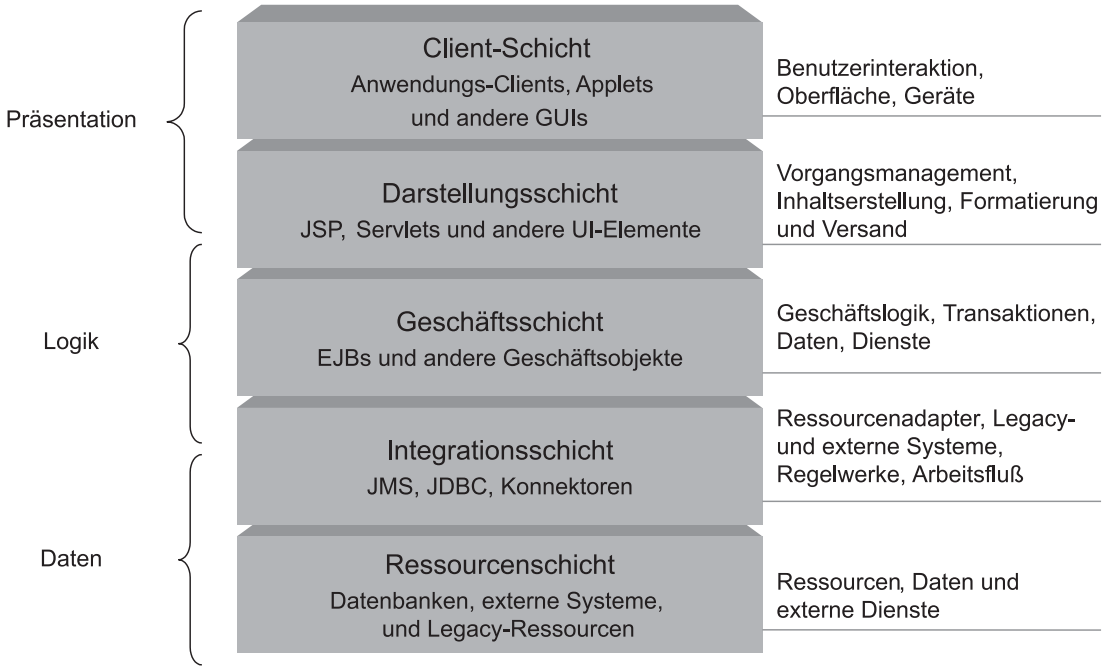


Abbildung 2.7: Mehrschichtige Client/Server-Architektur

Betrachtungsschwerpunkt liegt dabei auf der Schnittstelle des Dienstes und nicht auf dessen Implementierung (Black-Box) [Mye00]. Die bereitgestellten Dienste können dabei sehr unterschiedlicher Natur sein und die folgenden Aufgaben umfassen [Tea00]:

- Kontrolle des Lebenszyklus eines Geschäftsobjekts.
- Umsetzung von mehreren Geschäftsprozessen und -vorgängen.
- Kapselung der externen Schnittstelle der Fremd- oder Legacy-Systeme.

Die dienstbasierte Sicht einer Architektur lenkt den Betrachtungsschwerpunkt auf die Funktionalität der einzelnen Komponenten, die in einem Anwendungssystem vorhanden sind und auf den verschiedenen Schichten verteilt sind. Im Rahmen von Komponententechnologien wird die Kontrolle des Lebenszyklus der Geschäftsobjekte bereits vom Container erbracht und muß somit nicht mehr schwerpunktmäßig betrachtet werden. Es kann eine Konzentration auf die Umsetzung der Geschäftsprozesse und die evtl. notwendige Integration von Fremd- oder Altsystemen erfolgen.

2.2 Common Object Request Broker Architecture

Die *Common Object Request Broker Architecture (CORBA)* ist Bestandteil der *Object Management Architecture (OMA)*, die Objekte als grundlegende Bausteine verteilter Systeme definiert. Sie ist der Ausgangspunkt für alle Standardisierungsaktivitäten der *Object Management Group (OMG)* [Red96]. Die OMG ist eine unabhängige, nicht profitorientierte Organisation, die die Interessen von über 800 Mitgliedern vertritt. Es handelt sich dabei um Hardware-Hersteller, Software-Hersteller, Netzwerkbetreiber und kommerzielle Anwender von Software. Einige davon sind *AT&T*, *Borland*, *Intel*, *IBM*, *Siemens Nixdorf*, *Software AG* und *Sun Microsystems*.

2.2.1 Object Management Architecture

Die Architektur der OMA setzt sich aus zwei Teilen zusammen [Gro97b, Red96]. Das Objektmodell enthält im wesentlichen Begriffsdefinitionen, um Objekte in einem verteilten System beschreiben zu können. Das Referenzmodell beschreibt die Komponenten der Architektur und wie diese interagieren. Beim Objektmodell handelt es sich um ein klassisches Objektmodell. In einem System werden Aktionen hervorgerufen, indem Nachrichten (*Requests*) an Objekte, die bestimmte Dienste anbieten, geschickt werden. Einige wichtige Begriffsdefinitionen im Objektmodell sind:

- **Objekt:** Eindeutig identifizierbare Entität, die bestimmte Dienste (Operationen) anbietet. Das Objekt verfügt über ein Interface, das der Außenwelt bekannt gibt, welche Dienste angeboten werden, wie diese aufgerufen werden und was diese zurückliefern.
- **Request:** Der Client nimmt Dienste durch das Senden eines Requests in Anspruch. Der Request enthält die gewünschte Operation, die zugehörigen Parameter (falls vorhanden), das Zielobjekt und einen optionalen Kontext.

- **Objektreferenz:** Durch eine Objektreferenz kann jedes Objekt eindeutig identifiziert werden.
- **Interface:** Das Interface eines Objekts enthält eine Menge von Operationen und Attributen. Interfaces werden mit einer eigens dafür entworfenen Sprache, der *Interface Definition Language (IDL)*, beschrieben. Ziel ist eine Trennung von Interface und Implementierung.
- **Operation:** Eine Operation erbringt einen bestimmten Dienst. Sie besteht aus einem Namen und der Spezifikation von Parametern, Rückgabewerten sowie möglichen Exceptions.
- **Objektimplementierung:** Die Objektimplementierung führt Aktionen aus, die erforderlich sind, um die angebotenen Dienste zu erbringen.
- **Client und Server:** Die Begriffe sind als Rollen zu verstehen, die von den CORBA-Objekten zu bestimmten Zeitpunkten eingenommen werden. Ein Client, der einen Dienst eines Servers in Anspruch nimmt, kann auch selbst ein Server für andere Clients sein (*Peer-to-Peer*). Es kann aber auch reine Clients geben, die selbst keine Schnittstelle exportieren und damit keine CORBA-Objekte sind.

Das Referenzmodell besteht aus fünf, in Abbildung 2.8 dargestellten Komponenten. Für jede Komponente existieren eigene Standards, die sich in entsprechenden Dokumenten, die von der OMG herausgegeben werden, befinden.

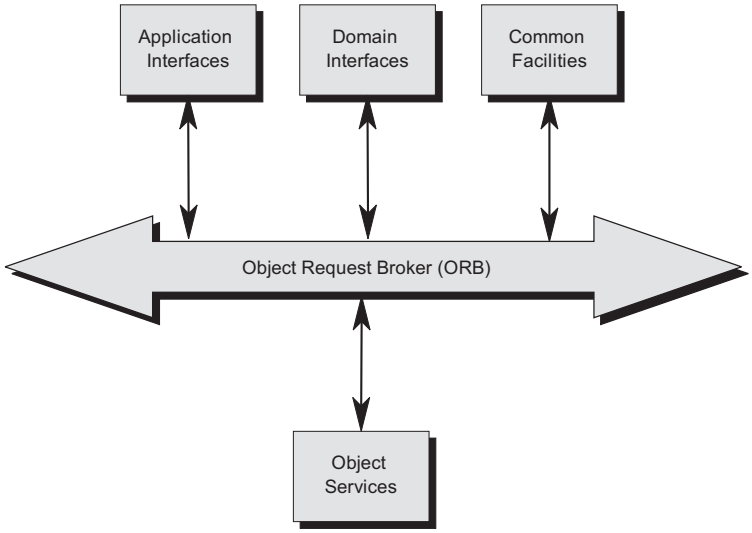


Abbildung 2.8: OMG-Referenzmodell (OMA)

Nachfolgend werden die Komponenten der OMA kurz charakterisiert:

- **Object Request Broker (ORB):** Der ORB stellt als zentrale Komponente die Kommunikationsinfrastruktur für Objekte bereit und sorgt dafür, daß es unerheblich ist, für welche

Maschinen, Betriebssysteme und in welcher Programmiersprache die Objekte implementiert sind. Der zugehörige Standard heißt *Common Object Request Broker Architecture (CORBA)* [Gro97c].

- **Object Services:** Die *Object Services* fassen Basisdienste zusammen, die von vielen verteilten Anwendungen benötigt werden. Es handelt sich dabei um Komponenten, die sich aus CORBA-Objekten zusammensetzen und deren Interface in IDL beschrieben ist [Red96]. In Abschnitt 2.2.5 sind zwei wichtige Basisdienste beschrieben. Für eine vollständige Übersicht sei an dieser Stelle auf [Gro97a, Gro97c] verwiesen.

- **Common Facilities (CORBAfacilities):** Die *Common Facilities* beinhalten Dienste, die der Endbenutzer in vielen Anwendungen benötigt. Durch Konfigurationseinrichtungen können sie an verschiedene Anforderungen angepaßt werden. Beispiele für *Common Facilities* sind Dokumentenverwaltung, Druckersteuerung und E-Mail [Red96]. Die *Common Facilities* werden häufig als horizontale *Common Facilities* (auch horizontale CORBA-*Facilities*) bezeichnet.

- **Domain Interfaces:** Die *Domain Interfaces* beinhalten Dienste für spezielle Anwendungsbereiche. U.a. beinhalten sie auch Beschreibungen bereits vorhandener Software in IDL [Red96]. Unterstützte Anwendungsgebiete sind z.B. das Finanzwesen (CORBAfinance), das Gesundheitswesen (CORBAhealth) und Telekommunikation (CORBAtel). Die *Domain Interfaces* werden auch als vertikale *Common Facilities* (auch vertikale CORBA-*Facilities*) bezeichnet.

- **Application Interfaces:** Die *Application Interfaces* beschreiben die Schnittstellen von Objekten, die ein spezielles Problem lösen, also die eigentliche Anwendung bilden. Die OMG nimmt hier keine Klassifikation oder Standardisierung vor. Die OMG stellt außerdem selbst keine Anwendungen her.

2.2.2 Architekturüberblick

Abbildung 2.9 enthält einen Überblick über die CORBA-Architektur [Gro97c]. Spezifiziert werden der Aufbau und die einzelnen Komponenten eines *Object Request Brokers (ORBs)*, der für die Kommunikation von verteilten Objekten untereinander zuständig ist. Die einzelnen Bestandteile sind:

- **IDL-Stubs und -Skeletons:** Die *Stubs* und *Skeletons* werden aufgrund der Objektbeschreibung, die mittels der IDL definiert wird, generiert. Sie sorgen dafür, daß die Übergabe- und Rückgabeparameter zwischen Client und Objektimplementierung verpackt und mit Hilfe des ORBs über das Netzwerk verschickt werden.

- **Object Adapter (OA):** Der OA sorgt dafür, daß eine Anfrage (*Request*) an das richtige Objekt weitergeleitet wird.

- **ORB-Core:** Der Kern des ORBs stellt das Kommunikationsprotokoll für Objekte zur Verfügung und übernimmt das Versenden von Anfragen und Antworten über das Netzwerk.

- **ORB-Interface:** Die Schnittstelle des ORBs ist standardisiert und erlaubt den Zugriff auf zentrale Dienste.
- **Dynamic Invocation Interface (DII) und Dynamic Skeleton Interface (DSI):** Das DII und das DSI stellen eine dynamische Alternative zur statischen Kommunikation mittels *Stubs* und *Skeletons* zur Verfügung. Zur Laufzeit können Anfragen erstellt, abgesendet und ausgewertet werden, ohne über eine IDL-Beschreibung der betreffenden CORBA-Objekte zu verfügen.
- **Interface Repository (IR):** Das IR stellt die Schnittstellenbeschreibungen von Objekten zur Verfügung. Die Daten des IR können u.a. zur dynamischen Kommunikation verwendet werden.
- **Implementation Repository:** Das *Implementation Repository* hält Informationen über die vorhandenen Objekte und dient zu deren Verwaltung durch den Objektadapter.

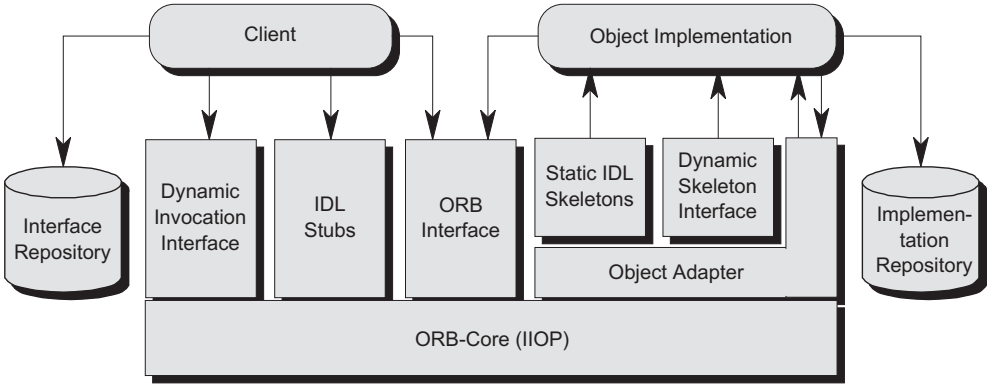


Abbildung 2.9: CORBA-Architektur

In den nachfolgenden Abschnitten erfolgt eine nähere Erläuterung der einzelnen Bestandteile.

2.2.3 Interface Definition Language

Aufgrund des Postulats der Trennung von Interface und Implementierung kommt der *Interface Definition Language (IDL)* im Rahmen des CORBA-Standards ein hoher Stellenwert zu. Bei der IDL handelt es sich um eine rein deklarative Sprache, die vollkommen programmiersprachenunabhängig ist. Die IDL dient zur Definition der Schnittstelle eines CORBA-Objekts. Die Schnittstelle gibt darüber Auskunft, welche Funktionalität von einem CORBA-Objekt bereitgestellt wird und enthält alle Informationen, die zur Entwicklung von Clients, die diese Funktionalität nutzen wollen, notwendig sind. Aus der IDL-Definition können von einem IDL-Compiler *Stubs* und *Skeletons*, je nach verwendeter Programmiersprache, für die Implementierung des Objekts erzeugt werden (z.B. C++, Java). Dabei werden die IDL-Definitionen gemäß der Sprachanbindung (*Language Mapping*) in fest vorgegebene Konstrukte der verwendeten Programmiersprache übersetzt. In Anhang C werden die in der IDL vorhandenen Datentypen und einige Beispiele für die Java-Sprachanbindung aufgelistet.

2.2.4 Object Request Broker

Der *Object Request Broker (ORB)* stellt die Kommunikationsinfrastruktur für verteilte Objekte zur Verfügung. Die einzelnen Bestandteile des ORBs werden nachfolgend kurz erläutert.

Kern

Der ORB-Kern ist für die Kommunikation der Objekte untereinander und alle daraus resultierenden Aufgaben zuständig. D.h. er muß sich mit zugrundeliegenden Netzwerken, Protokollen, Hardware-Plattformen und Betriebssystemen auseinandersetzen. Unterschiede zwischen den Implementierungen des Kerns werden durch darüberliegende Schichten, die eine standardisierte Schnittstelle besitzen, verdeckt. So können die Hersteller eines ORBs ihre Implementierung an eine bestimmte Systemumgebung optimal anpassen und bleiben dabei trotzdem standardkonform. Eines der Schlüsselkonzepte von CORBA-konformen ORBs ist die Interoperabilität zwischen Produkten verschiedener Hersteller. Hierzu ist die Definition von einheitlichen Objektreferenzen und einem einheitlichen Kommunikationsprotokoll der ORBs untereinander notwendig. Das *General Inter-ORB Protocol (GIOP)* ist ein solches Protokoll. Die Definition eines einheitlichen Datenformats und einer Menge von Nachrichtenformaten bilden zusammen ein abstraktes Protokoll, das relativ einfach auf existierende verbindungsorientierte Kommunikationsprotokolle, wie z.B. TCP/IP, Novell SPX und SNA Protokolle abgebildet werden kann. Allgemein steht hinter GIOP ein ähnliches Konzept wie hinter IDL. GIOP definiert ein gewünschtes Verhalten, läßt aber das zugrundeliegende Transportprotokoll offen. Diese Trennung wird in Abbildung 2.10 veranschaulicht.

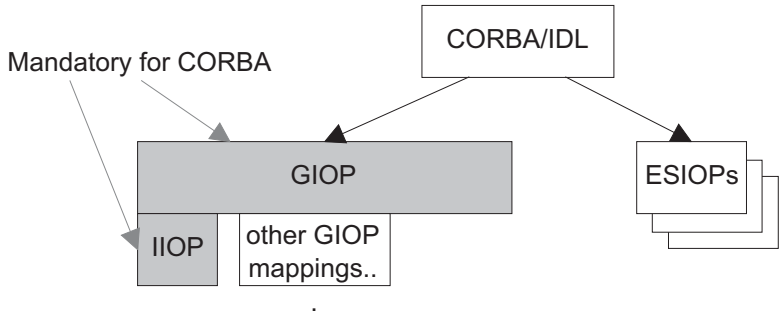


Abbildung 2.10: Konzept für herstellerunabhängige ORB-Kommunikationsprotokolle [Gro97c]

Die Abbildung von GIOP auf TCP/IP wird als *Internet Inter-ORB Protokoll (IIOP)* bezeichnet, da TCP/IP das Standardprotokoll des Internets darstellt. Zusätzlich sind sog. *Environment-Specific Inter-ORB Protocols (ESIOPs)* möglich, die es erlauben, einen ORB in bestehende Netzwerkstrukturen und Umgebungen für verteiltes Rechnen zu integrieren. Damit CORBA-Objekte verschiedener ORBs kommunizieren können, ist zusätzlich ein einheitliches Objektreferenzformat in Form von *Interoperable Object References (IORs)* notwendig. Jede IOR enthält ein oder mehrere Profile, die beschreiben, wie ein Client ein Objekt über ein bestimmtes Protokoll kontaktieren kann. Jede korrekte IOR muß ein IIOP-Profil besitzen, das die Internetadresse

des Servers für das gewünschte Objekt und einen Schlüsselwert zur Identifikation eines bestimmten Objekts auf dem Server enthält. Das IIOP-Protokoll hat sich als Industriestandard etabliert und spielt auch bei der Kommunikation in RMI-Systemen (Abschnitt 2.3) und EJB-Systemen (Abschnitt 2.5) eine große Rolle. Für eine ausführliche Beschreibung von IIOP sei hier auf [Gro97c, Ruh00] verwiesen.

IDL-Stubs (Static Invocation Interface)

Um die in Schnittstellen definierten Operationen von Server-Objekten in Anspruch nehmen zu können, benötigt der Client einen *Stub* für jedes dieser Objekte. Die *Stubs* werden aus den IDL-Schnittstellendefinitionen von Objekten generiert (IDL-Compiler) und bei der Übersetzung des Clients eingebunden. Sie werden deshalb auch als statisch bezeichnet, da die vom Client genutzten Schnittstellen bereits zur Übersetzungszeit feststehen. Der *Stub* enthält die notwendigen Definitionen und überführt die Operationsaufrufe und Parameter in ein flaches Nachrichtenformat (*Marshalling*), das über das Netzwerk versendet werden kann [Orf98]. Aus Sicht des Clients stellt der *Stub* also einen *lokalen Stellvertreter* für ein entferntes Objekt dar. Er wird deshalb auch treffend als „Proxy“, „Proxy-Objekt“ oder „Surrogat“ bezeichnet [Vog97]. Es handelt sich dabei um eine Umsetzung des Entwurfsmusters *Proxy*, das beschreibt, wie der Zugriff auf ein anderes Objekt kontrolliert werden kann [Gam95].

Dynamic Invocation Interface

Im Gegensatz zu den statischen *Stubs* steht beim *Dynamic Invocation Interface (DII)* zur Übersetzungszeit nicht fest, welche Objekte angesprochen werden sollen. Vielmehr kann der Client zur Laufzeit durch Metadaten das Interface eines Server-Objekts erkunden. Dabei identifiziert er die angebotenen Operationen samt Übergabe- und Rückgabeparameter, um anschließend eine Anfrage mit der gewünschten Operation und Parametern zu erzeugen, abzuschicken und die Ergebnisse zu empfangen. Dies erklärt auch die Notwendigkeit des *Interface Repositories*, das Metadaten über Schnittstellen von Server-Objekten bereitstellen muß. Für den Server stellen sich Anfragen über das DII oder IDL-*Stubs* gleich dar.
Ein weiterer, wesentlicher Unterschied zwischen dem DII und den statischen *Stubs* stellt der mögliche Kommunikationsablauf zwischen Client und Server dar. Bei der statischen Schnittstelle erfolgt die Kommunikation grundsätzlich synchron (*Synchronous Invocation*), d.h. der Client setzt seine Anfrage ab und blockiert so lange, bis das Ergebnis vom Server ankommt. Die Ausnahme von dieser Regel sind *Oneway*-Operationen (*Oneway Invocation*), die sofort nach dem Absetzen der Anfrage zurückkehren. Sie unterliegen aber starken Einschränkungen (z.B. keine Rückgaben).
Die dynamische Schnittstelle bietet zusätzlich die Möglichkeit eine Anfrage abzusenden, weiterzuarbeiten und die Antwort erst später entgegenzunehmen (*Deferred Synchronous Invocation*) [Vin97]. Allgemein ist anzumerken, daß die dynamische Schnittstelle um ein vielfaches langsamer als die statische sein kann. Vor allem, wenn Informationen über die Schnittstelle des entfernten Objekts erst aus dem ebenfalls entfernten *Interface Repository* übertragen werden müssen. Demgegenüber steht der Vorteil, daß keine statischen *Stubs* benötigt werden. Dies ist insbesondere dann ein Vorteil, wenn sehr viele verschiedene Objekttypen durch einen Client mit sehr wenig Speicherplatz kontaktiert werden müssen.

Das DII eignet sich auch gut zur Entwicklung von Fehlersuchwerkzeugen [Fis99].

Object Adapter

Durch den Objektadapter (OA) nimmt die Objektimplementierung Funktionen des ORBs in Anspruch. Umgekehrt kann der OA Leistungen, wie die Erzeugung von Objektreferenzen, Methodenaufrufen, Sicherheitsmechanismen, Abbildung von Objektreferenzen auf Implementierungen, Registrierung von Implementierungen, Implementierungsaktivierungen und -deaktivierungen erbringen. Jeder ORB muß einen Standardadapter besitzen, der für eine Vielzahl von Fällen ausreichend ist. Ein Objektadapter kann jedoch nicht für alle denkbaren Objekte adäquat sein. Objekte können sich z.B. hinsichtlich ihrer Granularität, Lebensdauer und Sicherheitsanforderungen unterscheiden. Somit können weitere Objektadapter vorhanden sein, die diesen Objekteigenschaften gerecht werden. Als Standardadapter wurde ursprünglich der *Basic Object Adapter (BOA)* angesehen. Dieser ist jedoch nicht ausreichend spezifiziert und führte in den ORBs unterschiedlicher Hersteller zu unterschiedlichem Verhalten. Seit Anfang 1998 ist der sog. *Portable Object Adapter (POA)* der Standardadapter und sollte statt dem BOA verwendet werden. Eine ausführliche Betrachtung der BOA-Probleme und dem neuen POA befindet sich in [Sch97].

IDL-Skeletons

Die IDL-*Skeletons* stellen auf der Server-Seite das Gegenstück zu den IDL-*Stubs* auf der Client-Seite dar. Sie werden ebenfalls aus den IDL-Definitionen von CORBA-Objekten erzeugt (IDL-Compiler) und zur Übersetzungszeit dem Server statisch hinzugebunden.

Dynamic Skeleton Interface

Das *Dynamic Skeleton Interface (DSI)* stellt das Gegenstück auf Server-Seite zum DII auf Client-Seite dar. Es wird so möglich, die in einer ankommenden Anfrage enthaltene Operation und Parameter erst zur Laufzeit an eine dafür bestimmte Objektklasse zu übergeben, ohne daß dafür ein IDL-*Skeleton* existiert. Es ist also zur Übersetzungszeit nicht festgelegt, welche Anfragen das betreffende Objekt verarbeiten kann. Die Anfragen von einem Client können dabei statisch (IDL-*Stub*) oder dynamisch (DII) sein.
Das DSI ermöglicht u.a. die Entwicklung von Brücken zu anderen Technologien, wie z.B. *Microsoft DCOM* [Fis99].

ORB-Interface

Das ORB-Interface stellt die Programmierschnittstelle zum lokalen ORB dar. Sie stellt z.B. Funktionen zur Initialisierung des ORBs und zum Erzeugen einer dynamischen Anfrage (siehe DII) bereit. Weitere Beispiele sind die Funktionen `object_to_string()` zum Umwandeln einer Objektreferenz (IOR) in einen String (z.B. zum Speichern in einer Datenbank) und `string_to_object()` für die umgekehrte Operation.

Interface Repository

Das Interface Repository stellt eine zur Laufzeit verteilte Datenbank dar [Orf98]. Die Schnittstellendefinitionen von CORBA-Objekten können darin gespeichert und abgefragt werden. Seit CORBA 2.0 gibt es sog. *Repository IDs*, die Komponenten und ihre Schnittstellen über unterschiedliche ORBs und Interface Repositories verschiedener Hersteller hinweg identifizieren. Es handelt sich dabei um Zeichenketten, die eine dreistufige Namenshierarchie beinhalten. `IDL:DogCatInc/MyAnimals/Cat/:1.0`, bezeichnet z.B. das Interface `Cat` im Modul `MyAnimals` in der Version `1.0`. `DogCatInc` ist ein eindeutiger Präfix.

Implementation Repository

Das *Implementation Repository (IR)* speichert sämtliche Informationen, die der ORB benötigt, um die vom Client gewünschte Objektimplementierung aufzufinden. Auf einem Server können z.B. mehrere Exemplare vom selben Objekttyp erzeugt worden sein. Bei einem eintreffenden Request muß das darin geforderte Objekt aus dieser Menge angesprochen werden. Außerdem kann das IR zur Speicherung von weiteren Informationen für eine Objektimplementierung, wie z.B. Debug-Informationen, Sicherheitsinformationen und belegte Ressourcen genutzt werden.

2.2.5 Basisdienste

Die im Rahmen der OMA klassifizierten *Object Services* spezifizieren Basisdienste, die von vielen verteilten Anwendungen benötigt werden. Es handelt sich dabei um Komponenten, die sich aus CORBA-Objekten zusammensetzen und deren Interface in IDL beschrieben ist [Red96]. Im folgenden werden einige Beispiele für Services aufgezählt [Gro97b], [Gro97a].

Namensdienst

Bevor CORBA-Objekte miteinander kommunizieren können, müssen sie voneinander erfahren, d.h. sie benötigen eine Objektreferenz des zu kontaktierenden Objekts. CORBA-Objekte können sich überall in einem Netzwerk befinden. Jedes dieser Objekte besitzt eine eindeutige Objektreferenz (IOR), die bei dessen Erzeugung vom ORB und OA vergeben wird. Für die Kontaktaufnahme zu einem bestimmten CORBA-Objekt benötigt der Client dessen Objektreferenz. Bei einer Anfrage des Clients kann der ORB anhand der Objektreferenz das gewünschte Objekt im Netzwerk finden. Es verbleibt allerdings das Problem, wie der Client in den Besitz der Objektreferenz für das Objekt, welches er verwenden will, kommt. Die OMG hat hierfür den Namensdienst spezifiziert (*COSS-Naming*), der Objektnamen Objektreferenzen zuordnet und somit im übertragenen Sinne ein Telefonbuch für CORBA-Objekte darstellt. Der CORBA-Standard sieht hierfür zwei ORB-Funktionen vor. Mit `list_initial_services()` kann eine Liste mit verfügbaren Diensten angefordert werden, in der u.a. der Namensdienst enthalten sein muß. Mit `resolve_initial_references()` kann schließlich die Objektreferenz für einen angegebenen Dienst ermittelt werden. Die grundsätzliche Kontaktaufnahme ist im folgenden Codefragment vereinfacht skizziert:

```
// Server-Objekt CORBAObjekt wird erzeugt
```

```
// und im Namensdienst eingetragen
...
// Suchen des Namensdienstes
namingobjref=resolve_initial_references("NameService");
...
CORBAObjektImpl coImpl=new CORBAObjektImpl();
...
// Binden im Namensdienst
namingobjref.bind( "CORBAObjekt", coImpl );
...

// Client benoetigt Operationen des
// CORBA-Objekts CORBAObjekt
...
// Suchen des Namensdienstes
namingobjref=resolve_initial_references("NameService");
...
// Suchen im Namensdienst
objref=namingobjref.resolve("CORBAObjekt");
// Operationen von CORBAObjekt sind durch objref
// nun verfuegbar.
...
```

Transaktionsdienst

Der CORBA-Transaktionsdienst ermöglicht eine transaktionale Zusammenarbeit von CORBA-Objekten in einem heterogenen Netzwerk. Dabei spielt die Plattform und Programmiersprache der Objekte keine Rolle. Es ist hervorzuheben, daß der Dienst neben flachen Transaktionen auch ein auf geschachtelten Transaktionen basierendes Modell ermöglicht. Dadurch können CORBA-Aufrufe sehr feingranular kontrolliert werden. Das Grundprinzip zur Anwendung des Transaktionsdienstes ist in Abbildung 2.11 dargestellt.
Ein Client kann Transaktionen auf verschiedenen Servern unter der Kontrolle des Transaktionsdienstes ausführen. Dieser ordnet dem Client zu diesem Zweck einen eindeutigen Transaktionskontext zu. Transaktionale Objekte sind Objekte, deren Verhalten beeinflußt wird, wenn sie innerhalb einer Transaktion aufgerufen werden. Typischerweise handelt es sich um Objekte, die Operationen anbieten, um persistente Daten zu manipulieren. Die Daten können dabei direkt im Objekt enthalten sein oder es kann nur ein Verweis darauf erfolgen. Ein transaktionales Objekt, das seine Daten direkt verwaltet und vom Ausgang einer Transaktion (*Commit* oder *Rollback*) abhängig ist, durch die Daten manipuliert werden, wird wiederherstellbar genannt (*Recoverable*). Wiederherstellbare Objekte kommunizieren mittels eines Objekts, das als *Ressource* bezeichnet wird, mit dem Transaktionsdienst. Die dafür vorgesehenen Protokolle koordinieren den Ausgang einer Transaktion und stellen sicher, daß sich alle Transaktionsteilnehmer über den Ausgang einer Transaktion einig sind (*Commit* oder *Rollback*). Transaktionale Server fassen ein oder mehrere transaktionale Objekte zusammen, die selbst keinen wiederherstellbaren Zustand besitzen und damit nicht beim Abschluß der Transaktion mitwirken. Ein wiederherstellbarer Server (*Recoverable Server*) faßt wiederherstellbare Objekte zusammen, die mittels

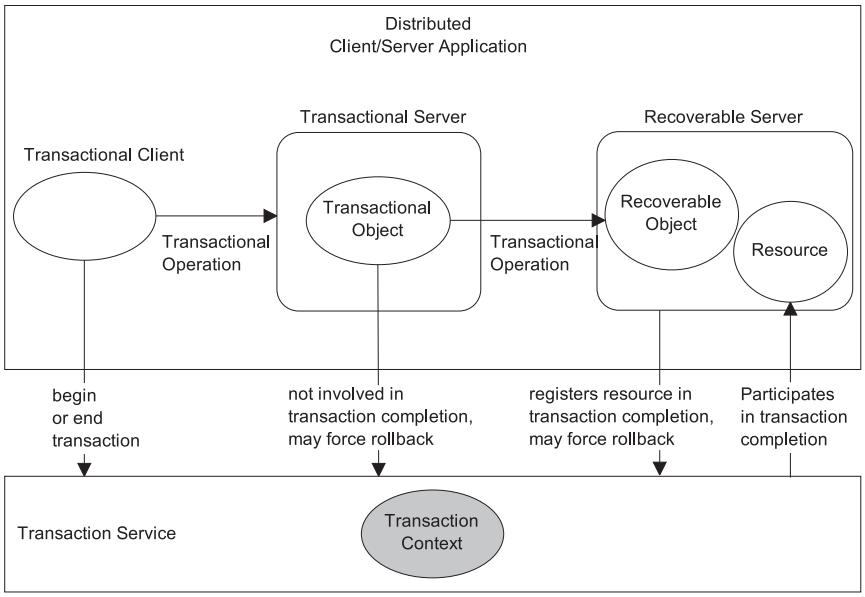


Abbildung 2.11: CORBA-Transaktionen [Gro97a]

Ressourcen-Objekten aktiv am Abschluß einer Transaktion teilnehmen.
Die Schnittstelle des CORBA-Transaktionsdienstes ist in IDL definiert. Das wichtigste Hauptmodul `CosTransactions` definiert Schnittstellen, die zur Verwendung von CORBA-Transaktionen herangezogen werden können:

- **Current** definiert Operationen des Clients zum Beginnen, Beenden und zur Statusabfrage von Transaktionen (z.B. `begin()`, `commit()`, `rollback()`, `status()`).
- **TransactionFactory** erzeugt übergeordnete Transaktionen und gibt `Control`-Objekte dafür zurück.
- **Control** erlaubt den expliziten Zugriff auf den Transaktionskontext, durch den Erhalt der Schnittstellen `Terminator` und `Coordinator`.
- **Terminator** definiert die Methoden `commit()` und `rollback()` zum Beenden einer Transaktion.
- **Coordinator** definiert Operationen, die von Transaktionsteilnehmern genutzt werden, um z.B. neue Untertransaktionen zu erzeugen oder externe Ressourcen zu registrieren.
- **RecoveryCoordinator** definiert die Operation `replay_completion()`, die von wiederherstellbaren Objekten genutzt wird, um ihren Zustand in einer Fehlersituation wiederherzustellen.
- **Resource** definiert die Operationen, die jede Ressource unterstützen muß, um das 2PC-Protokoll umzusetzen.

- **Synchronization** definiert die Methode `before_completion()`, die vor dem Start des 2PC-Protokolls aufgerufen wird und `after_completion()`, die nach dem Ablauf des Protokolls ausgeführt wird.

- **SubtransactionAwareResource** wird von wiederherstellbaren Objekten benutzt, um festzustellen, ob bei geschachtelten Transaktionen eine Untertransaktion erfolgreich beendet wurde.

- **TransactionalObject** besitzt keine Operationen, sondern dient zur Kennzeichnung von Objekten, die bei Anfragen den Transaktionskontext des Clients benötigen.

Zur Entwicklung von transaktionalen CORBA-Objekten kann zwischen impliziter und expliziter Kontextpropagierung gewählt werden. Bei der impliziten Variante wird der jeweilige Transaktionskontext automatisch im *Stub-Code* propagiert. Hierzu muß die Schnittstelle des CORBA-Objekts von `TransactionalObject` abgeleitet werden. Bei dieser Vorgehensweise sind alle Operationen der Schnittstelle transaktional, was zur Folge hat, daß immer der Transaktionskontext übertragen werden muß. Dies kann ein Nachteil sein, wenn nicht alle Operationen Transaktionalität benötigen. Zusätzlich muß an jedes Objekt eine ORB-spezifische OTS-Laufzeitbibliothek gebunden werden, die den Anwendungscode aufbläht, das Leistungsverhalten negativ beeinflußt und bei verschiedenen ORBs zu Interoperabilitätsproblemen führen kann [Fle]. Bei expliziter Propagierung muß jeder transaktionalen Operation eines CORBA-Objekts ein `Control`-Objekt übergeben werden. Dies erfordert keine Hinzubindung von proprietären OTS-Bibliotheken und kann das Leistungsverhalten positiv beeinflussen, da auch nur ausgewählte Operationen die Propagierung des Transaktionskontextes erfordern. Diese Methoden müssen allerdings früh in der Anwendungserstellung bekannt sein. Eine nachträgliche Propagierung des Transaktionskontextes in Operationen, die ursprünglich ohne Transaktionalität konzipiert wurden, kann zu Problemen führen [Fle]. Zur Entwicklung von wiederherstellbaren Objekten müssen CORBA-Objekte neben dem Interface `TransactionalObject` noch die Schnittstelle `Resource` implementieren.
Die Deklaration einer Transaktion im Anwendungscode kann direkt mit Hilfe der dafür vorgesehenen Schnittstelle `TransactionFactory` erfolgen oder indirekt mit dem `Current`-Interface. Die indirekte Alternative wird hauptsächlich bei impliziter Transaktionspropagierung verwendet [Fle]. Das nachfolgende Codefragment skizziert die Verwendung des OTS mit Hilfe eines `Current`-Objekts.

```
// Initialisierung des OTS in der Variable ots
// Deklaration von Current als Variable current

current = ots.get_current();

// Beginne Transaktion
current.begin();

// Fuehre transaktionale Operationen durch
transactionalObject.doSomething();
...
```

```
// Beende Transaktion
current.commit();
```

Für eine ausführliche Auseinandersetzung mit den OTS-Transaktionskonzepten sei hier auf die Spezifikation [Gro97a] und begleitende Literatur, wie z.B. [Vog99] verwiesen. In [Bau00] findet sich eine Projektfallstudie über den Einsatz von OTS.
In diesem Abschnitt wird trotz der kurzen Abhandlung des OTS deutlich, daß der CORBA-Anwendungsentwickler stark mit der Implementierung von Transaktionen in Berührung kommt. Die in Abschnitt 2.5 thematisierten *Enterprise JavaBeans* setzen auf einer Untermenge des OTS auf und reduzieren diese Komplexität aus Sicht des Anwendungsentwicklers deutlich. Die Transaktionalität von Methoden wird deklarativ beschrieben und mittels automatisch generiertem Code, der den Transaktionsdienst in Anspruch nimmt, realisiert.

2.3 Remote Method Invocation (RMI)

Die *Remote Method Invocation* [Mic99d] stellt einen einfachen Mechanismus zur Kommunikation zwischen Java-Objekten, die sich in unterschiedlichen virtuellen Maschinen befinden, zur Verfügung. Die virtuellen Maschinen können sich dabei auch auf unterschiedlichen Rechnern in einem Netzwerk befinden. Der Aufruf von Methoden eines entfernten Objekts unterscheidet sich grundsätzlich nicht von einem lokalen Aufruf und entlastet Entwickler somit von Details der Netzwerkprogrammierung. Aufgrund der nahtlosen Integration von RMI in die Sprache Java existieren kaum Einschränkungen in Bezug auf die Parametertypen, die in Methoden verwendet werden können.

2.3.1 Architekturüberblick

In Abbildung 2.12 ist die Architektur von RMI dargestellt (in Anlehnung an [Bog99]). Die Anwendung besteht aus den Client- und Server-Klassen, die mittels *Stubs* und *Skeletons* miteinander kommunizieren.

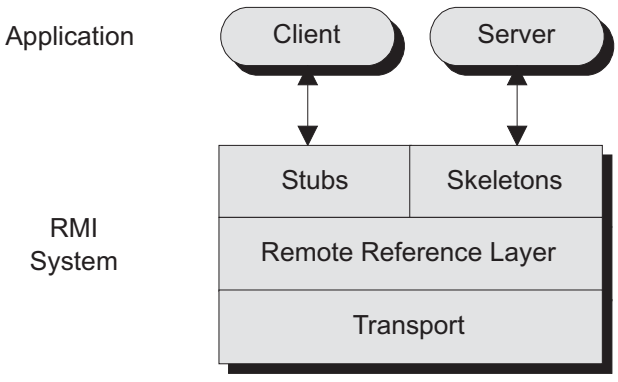


Abbildung 2.12: RMI-Architektur

Um Java-Objekte netzwerkfähig zu machen, muß eine Remote-Schnittstelle definiert werden, die dem Compiler *rmic* zur Generierung der *Stubs* und *Skeletons* dient. Die generierten Kommunikationsklassen sorgen automatisch für den Kommunikationsvorgang. D.h. Über- und Rückgabeparameter, die in den Methoden der Remote-Schnittstelle definiert sind, werden automatisch in ein Format gebracht, daß innerhalb einer Nachricht von der RMI-Transportschicht über das Netzwerk transportiert werden kann. Als Parameter kommen Objekte vom Typ `Serializable` oder Remote-Objekte selbst in Frage. Die Serialisierung ist ein in der Java-Bibliothek implementierter Mechanismus, der den Zustand eines Objekts in eine Folge von Bytes überführt, die anschließend weiterverarbeitet werden kann. Der Zustand von serialisierbaren Objekten wird komplett übertragen (Übertragung *copy-by-value*). Bei Remote-Objekten wird nur ein *Stub* auf das entfernte Objekt übertragen (Übertragung *copy-by-reference*).
Der Entwicklungsprozeß für RMI-Objekte ergibt sich, wie in Abbildung 2.13.

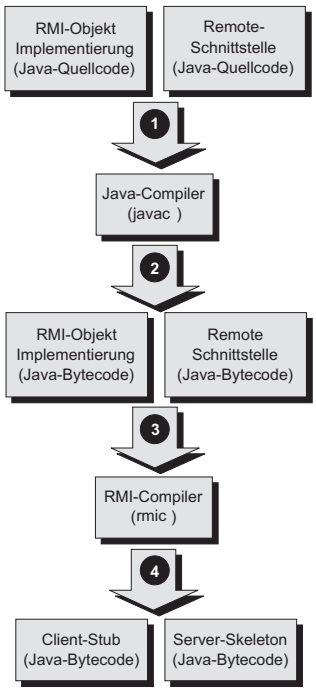


Abbildung 2.13: RMI-Entwicklungsprozeß

Für das RMI-Objekt muß ein Remote-Interface erstellt werden, das von `java.rmi.Remote` abgeleitet wird und die angebotenen Operationen definiert sowie die zugehörige Objektimplementierung, die von `java.rmi.UnicastRemoteObject` abgeleitet werden muß und die Remote-Schnittstelle implementiert (1). Anschließend müssen diese Klassen in Java-Bytecode übersetzt werden (2). Durch den RMI-Compiler `rmic` kann danach aus der Objektimplementierung (3) der Bytecode für den *Stub* und das *Skeleton* erzeugt werden (4). Um die Anfragen von Clients entgegennehmen zu können, muß die *RMI-Registry* gestartet werden, die als Namensdienst zum Auffinden von Objekten im Netzwerk dient. Danach kann das RMI-Objekt gestartet werden. Als Alternative zu `UnicastRemoteObject` kann auch `javax.rmi.PortableRemoteObject` oder `java.rmi.activation.Activatable` abgeleitet werden. `Ac-`

`tivatable` zeigt an, daß der Aktivierungsdienst verwendet werden soll (vgl. dazu 2.3.5). Die Verwendung von `PortableRemoteObject` zeigt an, daß mittels des IIOP-Protokolls kommuniziert werden soll (vgl. dazu Abschnitt 2.3.4).

2.3.2 Stubs und Skeletons

Der *Stub* ist der Stellvertreter für ein entferntes RMI-Objekt und bietet dem Client dessen Schnittstelle an. Der *Stub-Code* überführt Methodenparameter in ein Format, das über das Netzwerk verschickt werden kann (*Marshalling*). Er verwendet hierzu ein *Stream*-Objekt, das er aus der Referenzschicht erhält. Umgekehrt wird die Antwort des entfernten Objekts, vom *Stub*-Code wieder aus dem Netzwerkformat in korrespondierende Java-Objekte und -Datentypen umgesetzt (*Unmarshalling*). Abschließend informiert der *Stub* die Referenzschicht über den Abschluß der Anfrage [Dow98]. Das *Skeleton* ist der serverseitige Stellvertreter für ein RMI-Objekt und leitet Anfragen von Client-*Stubs* an die entsprechende Methoden weiter. Dabei werden die in der Anfrage enthaltenen Parameter aus dem Netzwerkformat rekonstruiert. Umgekehrt wird das Ergebnis der Anfrage zum Versand in ein transportfähiges Format gebracht [Dow98]. In der geänderten RMI-Implementierung von Java 2 ist kein *Skeleton* mehr erforderlich. Dessen Aufgaben werden dort mit einem *Reflection*-Mechanismus[2] gelöst [Ses00].

2.3.3 Referenzschicht

Die Referenzschicht stellt eine Abstraktionsebene zwischen den tatsächlich verwendeten Kommunikationsprotokollen und den Stellvertreterobjekten dar. Die Verbindung zu einem entfernten Objekt wird durch ein `RemoteRef`-Objekt repräsentiert. Die *Stubs* rufen auf diesem Objekt die `invoke`-Methode auf, um einen entfernten Methodenaufruf durchzuführen [Ses00]. Die *Stubs* und *Skeletons* kommunizieren dadurch immer mit *Stream*-Objekten der Refernzschicht, die unabhängig davon bereitgestellt werden, ob tatsächlich ein verbindungsorientiertes Kommunikationsprotokoll verwendet wird [Dow98]. Zusätzlich dient diese Schicht dem Auffinden des jeweiligen Partners, mit dem Daten ausgetauscht werden sollen. Auf dieser Schicht ist auch der Namensdienst in Form der *RMI-Registry* angesiedelt [Bog99].

2.3.4 Transportschicht

Die Transportschicht wickelt die eigentliche Kommunikation ab. Die Schicht verwaltet dabei Verbindungen, die zu anderen Maschinen bestehen. Standardmäßig wird eine *Stream*-basierte TCP/IP-Verbindung zur Kommunikation verwendet. Aufgrund der Einteilung in Schichten, kann die Transportschicht komplett ausgetauscht werden, ohne die anderen Schichten zu beeinflussen. Grundsätzlich basiert die Kommunikation auf dem Java-Serialisierungsprotokoll zur Überführung von Parametern in ein transportfähiges Format [Mic99d]. Die Serialisierung überführt den Zustand von Objekten in ein Format, das ausreichend ist, um die Objekte wieder rekonstruieren zu können [Mic99c]. Das Konzept der Serialisierung ist fest in der Java-

[2] Es handelt sich dabei um einen Mechanismus, der es erlaubt Objekte und deren Klassen zur Laufzeit auf die verwendeten Attribute, Konstruktoren und Methoden hin zu analysieren sowie diese zu verwenden. Java-Reflection wird z.B. ausführlich in [Mic01, McM97, McC98] beschrieben.

Klassenbibliothek verankert und fußt auf der Schnittstelle `java.io.Serializable`, die Objekte implementieren müssen, um serialisierbar zu sein. Der Objektzustand kann dabei auch andere Objekte umfassen, die ebenfalls mitgesichert werden müssen. Damit werden automatisch ganze Objektgraphen erfaßt. In Abbildung 2.14 ist ein Beispiel vorhanden. Die Serialisierung eines Objekts vom Typ `Klasse_1` führt zur Serialisierung aller direkt und indirekt referenzierten Objekte. Jede Referenz auf eine andere Klasse wird bei der Serialisierung aufgelöst. Um eine Exception zu verhindern, müssen diese Klassen ebenfalls die Schnittstelle `java.io.Serializable` implementieren. Objekte, die diesem Kriterium nicht entsprechen oder aus anderen Gründen nicht serialisiert werden sollen, können mittels des Schlüsselworts `transient` von der Serialisierung ausgeschlossen werden. Zu dieser Kategorie gehört das `String`-Objekt `ref3`. Neben den primitiven Datentypen ist die Mehrheit der in der Java-Bibliothek vorhandenen Klassen serialisierbar. Der Vorgang der Serialisierung kann grundsätzlich beeinflußt werden, indem Objekte ihre eigenen privaten Methoden `writeObject()` und `readObject()` implementieren. Dabei kann in den Methoden direkt auf den Ausgabe- bzw. Eingabe-*Stream* zugegriffen werden, der zur Laufzeit benutzt wird, um die Daten von Objekten zu serialisieren bzw. zu deserialisieren. Eine noch weitergehende Beeinflussung der Serialisierung/Deserialisierung kann durch Implementierung der `java.io.Externalizable`-Schnittstelle erreicht werden, die durch die öffentlichen Methoden `writeExternal()` und `readExternal()` eine vollständige Steuerung des Serialisierungsvorgangs erlaubt. Die Serialisierung wird z.B. in [Mica, Hal00b] beschrieben.

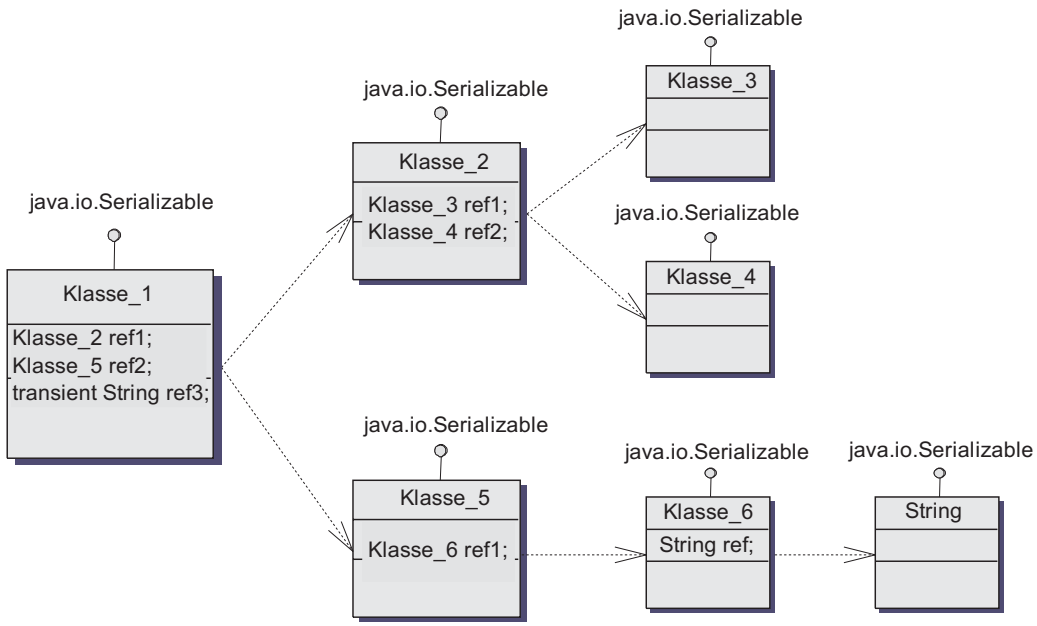


Abbildung 2.14: Serialisierung

Auf der Basis von TCP/IP wird das proprietäre *Java Remote Method Protocol (JRMP)* standardmäßig zur Kommunikation zwischen RMI-Objekten verwendet. Mit RMI-IIOP kann er-

reicht werden, daß RMI-Objekte auch das CORBA-Protokoll IIOP zur Kommunikation verwenden (vgl. 2.2) und damit beide Systemwelten verbunden werden können [And99]. Diese Thematik wird in Abschnitt 2.4 näher erläutert.

2.3.5 Basisdienste

Neben der Möglichkeit zum transparenten Aufruf von Methoden bietet RMI noch weitere Dienste an, die vom Anwendungsentwickler genutzt werden können.

Namensdienst

Die *RMI-Registry* kann als einfacher Namensdienst zum Auffinden von RMI-Objekten im Netzwerk verwendet werden und wird als Bestandteil des *Java Development Kit (JDK)* ausgeliefert (`rmiregistry`). Um ein RMI-Objekt Clients verfügbar zu machen, wird es zunächst an das RMI-System exportiert, um dafür zu sorgen, daß ein Dienst erzeugt wird, der die Anfragen von Clients an dieses Objekt entgegennimmt. Anschließend kann das Objekt unter einem beliebigen Namen in die RMI-Registry gebunden werden. Die Schnittstelle zur Registry wird durch die statische Klasse `Naming` bereitgestellt. Mittels der Methode `rebind()` kann sich ein RMI-Objekt in den Namensdienst eintragen. Clients können durch die Methode `lookup()` nach Objekten suchen, die in der Registry eingetragen sind. Hierzu müssen Namen in Form eines *Uniform Resource Locators (URL)* angegeben werden, der neben dem Objektnamen noch zusätzliche Angaben über den Rechnernamen auf dem die RMI-Registry gestartet ist und ggf. den Port auf dem sie Anfragen entgegennimmt, spezifiziert. Im nachfolgenden Codefragment ist die Verwendung der RMI-Registry veranschaulicht.

```
// Objektregistrierung auf dem Server
RMIObjekt rmiobj = new RMIObjekt();
String name = "//host/Objektname";
Naming.rebind(name, rmiobj);

// Objektsuche auf dem Client
String name = "//" + host + "/Objektname";
RMIObjekt obj = (RMIObjekt) Naming.lookup(name);
```

Die Verwendung der RMI-Registry ist auf Objekte beschränkt, die mittels JRMP kommunizieren. Für interoperable Objekte, die RMI-IIOP benutzen, muß der ebenfalls im JDK enthaltene Namensdienst `tnameserv` verwendet werden [Inc99]. Dabei handelt es sich um eine Implementierung des CORBA-Namensdienstes COSNaming (vgl. dazu Abschnitt 2.2.5), die über eine standardisierte Schnittstelle verfügt (JNDI, vgl. dazu Abschnitt 2.5). Die Verwendung ist im nachfolgenden Codefragment skizziert;

```
// Objektanmeldung beim Namensdienst auf dem Server
MyObjectImpl objRef = new MyObjectImpl();
Context initialNamingContext = new InitialContext();
initialNamingContext.rebind( "MyObjectService", objRef );
```

```
// Objektsuche im Client
Context ic = new InitialContext();
objref = ic.lookup( "MyObjectService" );

MyObjectInterface service =
  (MyObjectInterface) PortableRemoteObject.narrow(
            objref, MyObjectInterface.class);

// entfernte Methodenaufrufe
service.doSomething(...);
```

Aktivierungsdienst

Der Aktivierungsdienst *Remote Object Activation* verhindert, daß jedes RMI-Objekt explizit gestartet werden muß, bevor es durch Clients genutzt werden kann. Statt dessen werden RMI-Objekte in Abhängigkeit von vorliegenden Client-Anfragen gestartet. Dieses Verhalten wird durch *Lazy Activation* umgesetzt, d.h. die Klasse des RMI-Objekts wird erst bei der ersten Anfrage durch einen Client geladen, vorher nicht. Für Clients ist dieser Vorgang völlig transparent. Die Aktivierung auf Anforderung führt zu einem besseren Verhalten in Fehlersituationen, wenn ein Objekt abgestürzt ist und bei erneuten Anfragen wieder automatisch gestartet wird. Außerdem ermöglicht der Dienst die Realisierung sehr großer, aus sehr vielen RMI-Objekten bestehenden Systemen, die nicht gleichzeitig im Speicher gehalten werden können und verhindert, daß selten genutzte Objekte den Speicher unnötig belasten [Ses99]. Zur Umsetzung des soeben beschriebenen Verhaltens muß ein RMI-Objekt die Schnittstelle `Activatable` ableiten und den RMI-Aktivierungsdämon `rmid` verwenden.

Verteilte Speicherbereinigung

Die verteilte Speicherbereinigung stellt eine konsequente Ausdehnung des Java-Prinzips der automatischen Speicherbereinigung auf verteilte Umgebungen dar. RMI-Objekte werden dabei durch den *Distributed Garbage Collector (DGC)* aus dem Speicher entfernt, wenn keine Referenzen von Clients mehr existieren. Der DGC bedient sich eines sog. *Leasing-Prinzips*. Wenn ein Client eine Referenz auf ein RMI-Objekt erhält, ist diese für eine beschränkte Zeit, die mit der Umgebungsvariable `java.rmi.dgc.leaseValue` gesteuert werden kann, gültig. Solange der Client ausgeführt wird, erneuert er den Lease automatisch, wenn die halbe Zeit von `leaseValue` vergangen ist. Bei Beendigung des Clients entfällt die Erneuerung des Leases und der DGC erkennt dies als weggefallene Referenz auf das RMI-Objekt. Falls ein RMI-Objekt benachrichtigt werden soll, bevor es vom DGC aus dem Speicher entfernt wird, kann das Interface `Unreferenced` implementiert werden. Die Methode `unreferenced()` der Objektimplementierung wird dann aufgerufen, sobald keine aktive Referenz mehr auf das Objekt existiert. In dieser Methode kann dann das Schließen von Ressourcen, wie z.B. Netzwerk- und Datenbankverbindungen erfolgen [Ses99].

2.4 CORBA oder RMI

Die Existenz von RMI und CORBA führt zunächst zu einem Dilemma in der Wahl eines Mechanismus zur verteilten Kommunikation in Java, da zwischen einem proprietären, sprachspezifischen Modell und einem auf breiter Interoperabilität beruhenden Modell gewählt werden muß [And99]. RMI fügt sich nahtlos in die Programmiersprache Java ein und unterstützt deren Eigenschaften, wie z.B. Code-Mobilität und automatische Speicherverwaltung. Die Anwendung von RMI ist im Sinne von Java sehr einfach gehalten, basiert jedoch grundsätzlich auf dem proprietären Protokoll JRMP. Im Gegensatz dazu steht das sprachneutrale Modell von CORBA, das neben objektorientierten Sprachen, wie z.B. Java und C++ auch prozedurale Sprachen, wie z.B. C und COBOL unterstützt. CORBA bietet somit eine sehr gute Möglichkeit heterogene Systeme mittels des als Industriestandard anerkannten Protokolls IIOP zu verbinden, wenn keine Neuimplementierung in Java erfolgen soll. Dies wird aufgrund der zahlreich verfügbaren CORBA-Implementierungen für unterschiedliche Plattformen möglich. Ein ausführlicher Vergleich beider Technologien findet sich z.B. in [Orf98].
Zur Überwindung des Dilemmas, entstand 1999 aus einer Zusammenarbeit von IBM, Sun und der OMG eine RMI-IIOP-Programmierschnittstelle, die beide Welten auf einfache Weise verbindet [And99]. Zur Umsetzung der Verbindung zwischen RMI und CORBA wurde der CORBA-Standard um die Spezifikationen *Objects by Value* [Gro98] zur Bereitstellung eines zur Java-Serialisierung ähnlichen Mechanismus in CORBA und *Java-to-IDL Mapping* [Gro99] zur Umsetzung von Java-Schnittstellen in IDL-Schnittstellen, erweitert. Mit RMI-IIOP können Objekte erstellt werden, die JRMP und IIOP unterstützen. Ein Protokollwechsel erfordert dabei keine Neuübersetzung des Codes. Ein RMI-IIOP-Objekt, das gleichzeitig durch RMI-Clients (JRMP) und CORBA-Clients (IIOP) kontaktiert werden soll, muß dabei lediglich für beide erforderlichen Kommunikationsprotokolle auf dem Server *exportiert* werden (*Dual Export*). Daraus ergeben sich die in Abbildung 2.15 [And99] dargestellten Kommunikationsmöglichkeiten.

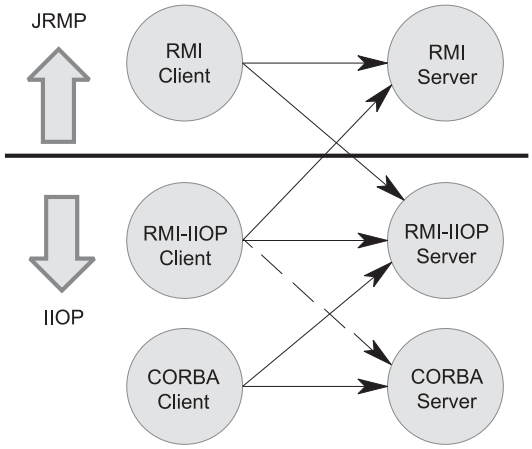


Abbildung 2.15: RMI-IIOP

Die Verbindung zwischen RMI und CORBA wird dabei durch die diagonalen Pfeile dargestellt. Ein RMI-IIOP-Objekt kann gleichermaßen von RMI-Clients, RMI-IIOP-Clients und CORBA-

Clients genutzt werden. Ein RMI-IIOP-Client kann auf ein RMI-Objekt zugreifen und mit Einschränkung auch auf ein CORBA-Objekt (gestrichelte Linie). Die Einschränkung besteht darin, daß die von RMI-IIOP realisierten Java-Schnittstellen nur eine Untermenge aller möglichen CORBA IDL-Konstrukte anbieten. Zur Kommunikation von RMI-Objekten mit dem IIOP-Protokoll ändert sich der Entwicklungsprozess aus Abbildung 2.13 grundsätzlich nicht. Der `rmic`-Compiler muß lediglich mit dem Parameter `iiop` gestartet werden. In Folge dessen werden *Stubs* und *Skeletons* erzeugt, die RMI-Objekte über einen ORB mittels IIOP kommunizieren lassen (Abbildung 2.16).

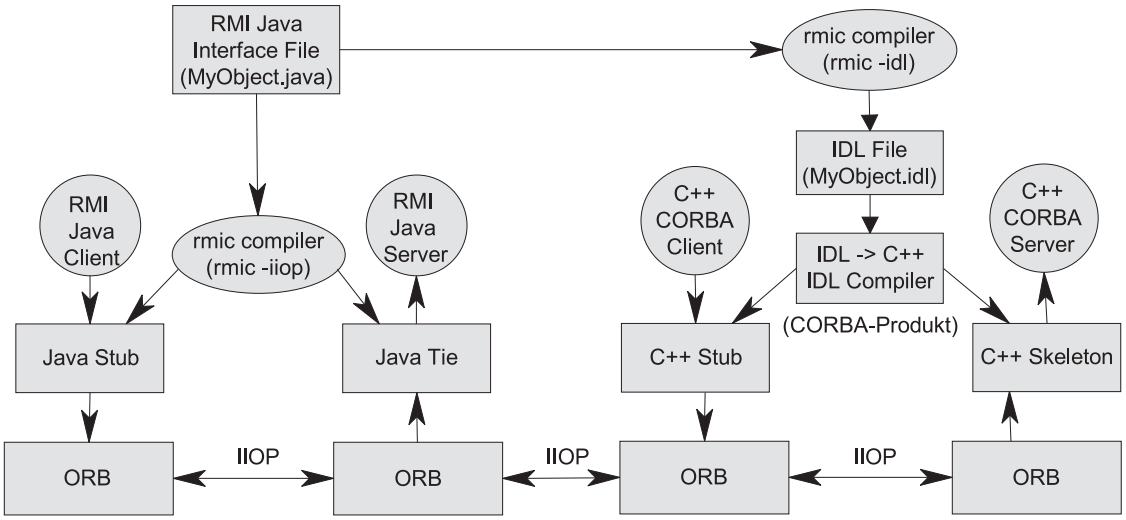


Abbildung 2.16: RMI-IIOP-Entwicklungsprozeß mit Java und C++ (in Anlehnung an [And99])

Mit Hilfe des `rmic`-Compilers und dem Parameter `idl` kann auch eine IDL-Beschreibung einer Java-Schnittstelle erzeugt werden, die anschließend zur Implementierung eines CORBA-Objekts in einer anderen Programmiersprache herangezogen wird. In Abbildung 2.16 ist dies am Beispiel der Programmiersprache C++ dargestellt. Nachdem aus der Java-Schnittstelle eine IDL-Beschreibung erzeugt wurde, werden C++-*Stubs* und C++-*Skeletons* mittels eines entsprechenden IDL-Compilers für C++ generiert. Somit kann der entstandene C++-Client mit dem RMI-Server über IIOP kommunizieren. Falls der Server mittels C++ implementiert wird, kann der RMI-Client diesen ebenso über IIOP ansprechen.
Als Fazit kann festgehalten werden, daß sich CORBA aufgrund seiner Plattform- und Programmiersprachenunabhängigkeit besser zur Integration bestehender Systeme in heterogenen Umgebungen eignet. RMI bietet sich aufgrund seiner einfacheren Anwendung in Java-Systemen an. Durch RMI-IIOP besteht die Möglichkeit beide Vorteile zu nutzen.
Der im folgenden Abschnitt erläuterte Komponentenstandard *Enterprise JavaBeans* basiert auf der Programmierung im RMI-IIOP-Stil. Das tatsächlich zur Kommunikation verwendete Protokoll kann dadurch von der Laufzeitumgebung, in der die Komponenten zur Ausführung kommen, gewählt werden. Somit ist es z.B. in der Laufzeitumgebung möglich, einen CORBA-ORB für die Kommunikation einzusetzen, der auf bestehende CORBA-Basisdienste zurückgreift, um

z.B. die Transaktionssteuerung zu übernehmen.

2.5 Enterprise JavaBeans (EJB)

Bei Enterprise JavaBeans handelt es sich um ein serverseitiges Komponentenmodell, das die Entwicklung von objektorientierten Client/Server-Anwendungen vereinfachen und beschleunigen soll. EJBs sind Bestandteil der *Java 2 Enterprise Edition (J2EE)*, die eine Reihe von Standards zusammenfaßt, die zur Realisierung von unternehmensweiten Java-Anwendungen benötigt werden. Im wesentlichen beruht das Modell auf den folgenden Spezifikationen:

- ***Enterprise JavaBeans (EJB) [DeM01].*** Die Spezifikation definiert Struktur und Verhalten der Server-Komponenten und den damit verpflichtenden Leistungsumfang, den ein Laufzeitsystem in Form eines *Applikations-Servers* erbringen muß, um eine Ablaufumgebung für die Komponenten bereitzustellen.

- ***Remote Method Invocation (RMI) und RMI-IIOP [Inc99].*** Die Spezifikation definiert die Möglichkeit von Methodenaufrufen über Prozeßgrenzen hinweg mittels JRMP (vgl. Abschnitt 2.3). RMI-IIOP ermöglicht die Interoperabiltität zwischen RMI- und CORBA-Systemen, die mit dem IIOP-Protokoll kommunizieren (vgl. Abschnitt 2.2 und Abschnitt 2.3).

- ***Java Naming and Directory Interface (JNDI) [Mic99a], [Mic99b].*** Die JNDI-Spezifikation definiert einen standardisierten Namensdienst, der das Auffinden von Komponenten und Ressourcen im Netzwerk erlaubt.

- ***Java Database Connectivity (JDBC) [Ell01].*** Die JDBC-Spezifikation legt den Zugriff auf relationale Datenbanken mittels Java fest.

- ***Java Transaction Service (JTS) [Che99] und* Java Transaction API (JTA) [Che01].** Die beiden Spezifikationen definieren die Eigenschaften eines Transaktionsdienstes und dessen Verwendung in Applikationen.

- ***Java Messaging Service (JMS) [Hap01].*** JMS spezifiziert einen asynchronen verteilten Kommunikationsmechanismus zwischen Objekten.

- ***Java Servlets und Java Server Pages (JSP) [Cow01b], [Cow01c]***. Die beiden Spezifikationen legen den Aufbau und das Verhalten von auf Java basierenden Web-Komponenten fest, die in einem Applikations-Server dynamischen Inhalt, wie z.B. HTML-Seiten, generieren.

- ***Java IDL* [Micb].** Mit *Java IDL* stellt Sun Microsystems eine CORBA-Implementierung für Java bereit.

- ***JavaMail [Mic00].*** Mittels der JavaMail-Spezifikation wird das plattform- und protokollunabhängige Versenden von E-Mails mit Java definiert.

- ***Java Connector Architecture (JCA) [Sha01b].*** JCA liefert einen Standard für Konnektoren, die zur Integration von Fremd- und Altsystemen in J2EE-Anwendungen herangezogen werden können.

- ***Java API for XML Processing (JAXP) [Mor01].*** Spezifiziert eine Java-Schnittstelle zur standardisierten Verarbeitung von XML[3]-Dokumenten in Java-Programmen.

Die *Java 2 Enterprise Edition Specification* [Sha01a] spezifiziert den Aufbau einer Laufzeitumgebung (Applikations-Server), die zur Ausführung von J2EE-Anwendungen geeignet ist. Auf Basis der Spezifikation erstellte J2EE-Anwendungen sollen möglichst herstellerunabhängig sein. Kommerzielle und nichtkommerzielle Implementierungen von Applikations-Servern können mittels einer von Sun bereitgestellten Test-Suite auf ihre Standardkonformität hin überprüft werden. Standardkonforme Produkte erhalten ein J2EE-Logo.

2.5.1 Architekturüberblick

Abbildung 2.17 (in Anlehnung an [Rom02]) enthält eine schematische Darstellung der EJB-Architektur.

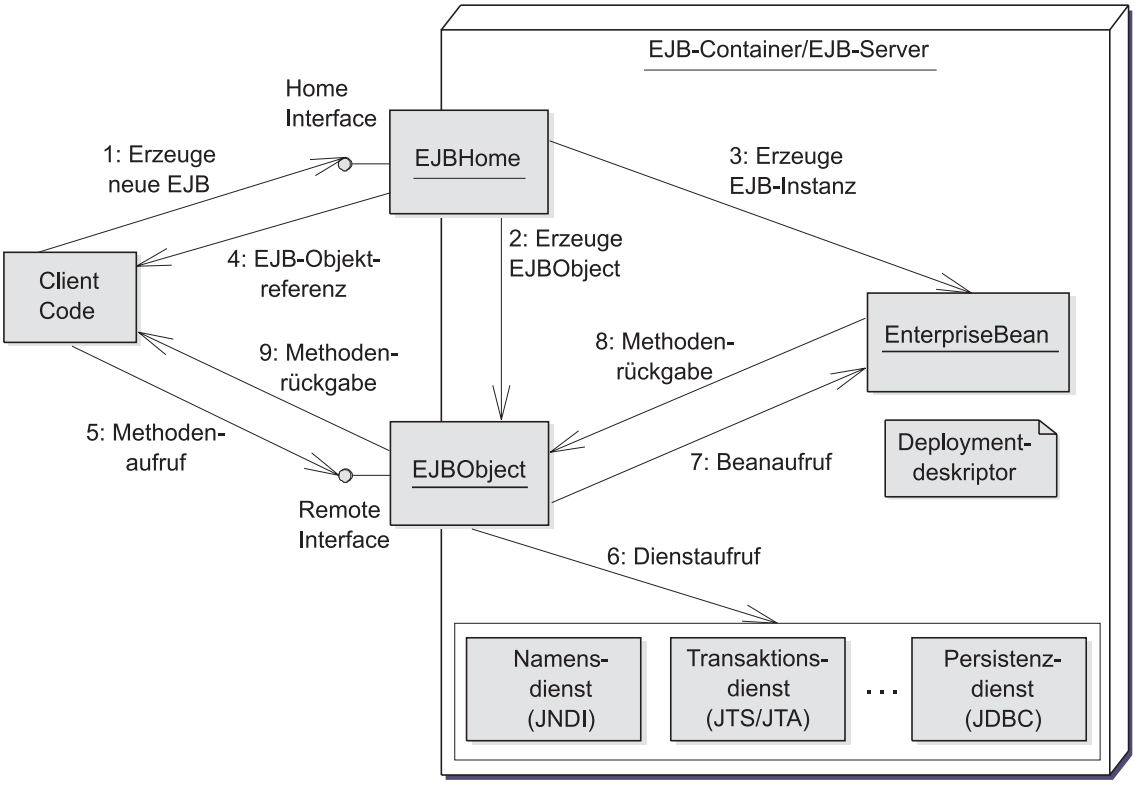


Abbildung 2.17: EJB-Architektur

[3]Extensible Markup Language

Die Abbildung zeigt den EJB-Server und EJB-Container, die den EJBs eine Ablaufumgebung zur Verfügung stellen, die aus einer Reihe von Basisdiensten, wie z.B. dem Namensdienst und Transaktionsdienst, bestehen. Die Basisdienste können mittels des *Deployment-Deskriptors* für jede einzelne EJB konfiguriert werden. Dieses *deklarative Programmierkonzept* ist ein wesentlicher Bestandteil der EJB-Umgebung.
Die EJBs selbst implementieren die Geschäftslogik einer Anwendung und bestehen aus einem Home-Objekt, einem EJB-Objekt und der EJB-Implementierung selbst. Das Home-Objekt kann Exemplare der entsprechenden Bean erzeugen und vernichten. Es erlaubt damit die Einflußnahme auf den Lebenszyklus einer Bean durch Clients. Das EJB-Objekt stellt ein technisches Schlüsselkonzept der EJB-Umgebung dar. Ein Client greift nie direkt auf eine EJB-Instanz zu, sondern immer über das EJB-Objekt. Durch diese Indirektion wird es möglich Zugriffe zu kontrollieren und entsprechende Basisdienste, wie sie im Deployment-Deskriptor definiert sind, mit einzubeziehen. Das `EJBObject` und das `HomeObject` müssen nicht direkt vom Anwendungsentwickler implementiert werden, sondern die jeweils zugehörigen `Home`- und `Remote`-Schnittstellen. Die Implementierungen der Objekte werden vom EJB-Container anhand der Schnittstellen und der im Deployment-Deskriptor vorhandenen Informationen erzeugt. Dadurch wird eine Konzentration des Anwendungsentwicklers auf die fachliche Logik in der eigentlichen Bean-Implementierung ermöglicht. Das Zusammenspiel der einzelnen Objekte ist ebenfalls in Abbildung 2.17 skizziert. Der Client erzeugt mit Hilfe des Home-Objekts eine neue EJB (1). Das Home-Objekt erzeugt ein zugehöriges EJB-Objekt (2) und eine zugehörige EJB-Instanz (3). Der Client erhält eine Objektreferenz auf die von ihm erzeugte EJB (4). Der Client ruft eine Methode der Remote-Schnittstelle der EJB auf (5). Das EJB-Objekt nutzt die Schnittstelle zu den Basisdiensten (6) und leitet die Anfrage an die eigentliche Bean-Instanz weiter (7). Die Bean-Instanz arbeitet die Methode ab und gibt das Ergebnis zurück (8). Das EJB-Objekt gibt das Ergebnis an den Client zurück (9). Falls die EJB nicht entfernt aufgerufen werden soll, sondern nur innerhalb des selben Prozesses, können statt der `Home`- und `Remote`-Schnittstellen `LocalHome`- und `Local`-Schnittstellen verwendet werden, die zur Generierung von korrespondierenden `LocalHome` und `EJBLocalObject` führen. Es handelt sich dabei um eine Erweiterung des EJB 2.0-Standards.
Zur Installation von EJBs in einem EJB-Container wird der sog. *Deployment-Prozeß* ausgeführt. Bei diesem Prozeß können verschiedene EJBs zusammengestellt und anhand ihres Deployment-Deskriptors konfiguriert werden. Neben der Konfiguration der Basisdienste können auch noch anwendungsspezifische Informationen in Form von Schlüssel/Wert-Paaren im Deployment--Deskriptor für jede einzelne Bean abgelegt werden, die nach der Installation in der Container-Umgebung durch die Bean abgerufen werden können. Dies ermöglicht eine hohe Flexibilität der Server-Komponenten und erlaubt die Erstellung anwendungsunabhängiger Komponenten, die ohne Quellcode weitergegeben werden können. Zur Weitergabe und Installation müssen ein oder mehrere Beans in einem Java-Archiv (`jar`-Datei) mit ihrem Deployment-Deskriptor untergebracht werden.

2.5.2 Komponentenarten

Die EJB-Spezifikation definiert verschiedene Komponentenarten, um serverseitige Anwendungen aufzubauen. In Abbildung 2.18 sind die unterschiedlichen Komponenten und ihre Aus-

prägungen dargestellt.

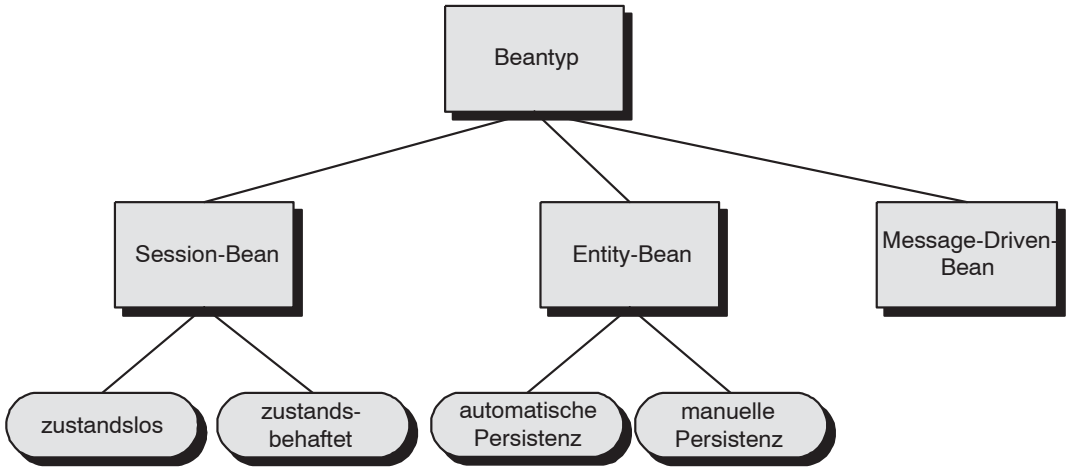


Abbildung 2.18: EJB-Arten

Die Charakteristika der verschiedenen Bean-Arten werden nachfolgend kurz beschrieben:

- **Session-Beans** bieten Dienste an, die von Clients genutzt werden können. Zustandslose Session-Beans (*Stateless Session Beans*) können vom Applikations-Server in einem Instanzen-Pool gehalten werden und jedem beliebigen Client zum Aufruf einer Methode bereitgestellt werden. Somit können Beans dieses Typs keinem Client eindeutig zugeordnet werden und damit keine clientspezifischen Daten halten. Eine eindeutige Zuordnung kann jedoch bei zustandsbehafteten Session-Beans (*Stateful Session Beans*) erfolgen. Hier können Daten gehalten werden, die dem Client auch über mehrere Methodenaufrufe hinweg zur Verfügung stehen.

- **Entity-Beans** sind persistente Komponenten, die ihren Zustand in einem dauerhaften Speicher halten. Bei Entity-Beans mit automatischer Persistenz (*Container-managed Persistence*) kümmert sich der EJB-Container vollständig um die Anweisungen, die erforderlich sind, um den Zustand in einem dauerhaften Speicher zu speichern und von ihm zu lesen. Der Anwendungsentwickler wird dabei von der Entwicklung eines Persistenzmechanismus entlastet. Im Gegensatz dazu werden bei Entity-Beans mit manueller Persistenz (*Bean-managed Persistence*) Persistenzmechanismen vom Anwendungsentwickler selbst entwickelt und umgesetzt.

- **Message-Driven Beans** sind Komponenten, die JMS-Nachrichten konsumieren können. Im Gegensatz zu Session- und Entity-Beans, die synchron mit Clients kommunizieren und deshalb blockieren bis sie eine Antwort erhalten, kommunizieren Message-Driven Beans durch die JMS-Mechanismen mit ihren Clients asynchron. Als Folge davon blockieren Clients nicht, wenn sie indirekt durch JMS Dienste einer Message-Driven Bean in Anspruch nehmen.

Aufgrund des deklarativen Programmiermodells werden wichtige Eigenschaften jeder Komponente mittels des Deployment-Deskriptors festgelegt und konfiguriert. Nachfolgend wird der

Aufbau des Deployment-Deskriptors und die wesentlichen Merkmale der unterschiedlichen Komponenten erläutert.

Session-Beans

Eine Session-Bean bietet serverseitige Dienste an, die von Clients genutzt werden können. Sie implementiert Geschäftslogik- und -regeln, Algorithmen und Geschäftsabläufe (*Workflow*) [Rom02]. Beim Aufruf einer Methode der Session-Bean garantiert der EJB-Container, daß kein anderer Client die selbe Bean-Instanz verwendet. Der EJB-Container kann dabei mehrere Instanzen des selben Bean-Typs bereitstellen, um mehrere Clients simultan zu bedienen (*Pooling*). Zwischen Session-Bean und Client existiert ein dialogorientierter Zustand (*Conversational State* [DeM01]), der je nach verwendeter Unterart von Session-Beans von unterschiedlicher Dauer ist:

- **Zustandsbehaftete Session-Bean.** Der Zustand existiert über *viele Methodenaufrufe* des Clients hinweg. Der EJB-Container hält den Zustand für jeden individuellen Client. Beispiel hierfür ist ein Warenkorb im Internet, der von einem Kunden über einen längeren Zeitraum hinweg gefüllt wird.

- **Zustandslose Session-Bean.** Der Zustand existiert nur während der Dauer *eines Methodenaufrufs*. Beans dieser Art können sich keinen Client-spezifischen Zustand merken. Jede Instanz kann jedem beliebigen Aufrufer zugeordnet werden. Dies bedeutet aber auch, daß andere allgemeingültige Zustände erlaubt sind. Dazu gehört z.B. das Halten von initialisierten Home- und Remote-Objekten, um mit anderen Beans zu kommunizieren. Ein Anwendungsbeispiel für eine zustandslose Session-Bean ist z.B. ein Kreditkartendienst, der anhand der übergebenen Kreditkartendaten die Gültigkeit der Karte überprüft.

In Abbildung 2.19 sind die unterschiedlichen Verhaltensweisen veranschaulicht. Ein Methodenaufruf von einer zustandslosen Session-Bean wird an eine beliebige Bean-Instanz delegiert, die der Container einem Pool entnimmt. Der Aufruf einer Methode von einer zustandsbehafteten Session-Bean findet immer auf einer Instanz mit dem selben, vom Client hervorgerufenen, Zustand statt. Zur Reduzierung von Bean-Instanzen, die zur Laufzeit im Speicher gehalten werden müssen, kann der EJB-Container den Zustand von Bean-Instanzen passivieren, d.h. in einem Sekundärspeicher zwischenlagern. Sollte aufgrund einer weiteren Anfrage des betreffenden Client eine Instanz mit diesem Zustand benötigt werden, wird durch die Aktivierung eine solche Instanz erzeugt und mit dem zugehörigen Zustand versehen. Aufgrund der durch die EJB-Spezifikation vorgeschriebenen Serialisierbarkeit von zustandsbehafteten Session-Beans wird bereits eine einfache technische Alternative der Zustandssicherung möglich. Nach welchen Kriterien die Passivierung erfolgt und mit welchem Mechanismus die Zustände persistent gemacht werden, liegt aber letztlich im Ermessen des Container-Herstellers [MH99].
Zur Implementierung einer Session-Bean müssen die folgenden Elemente bereitgestellt werden:

- **Implementierung der Session-Bean:** Enthält die Geschäftsattribute und -methoden. Zur Verwaltung des Lebenszyklus durch den EJB-Container muß zusätzlich die Schnittstelle `javax.ejb.SessionBean` implementiert werden. In Tabelle 2.2 sind die zu implementierenden Methoden aufgeführt.

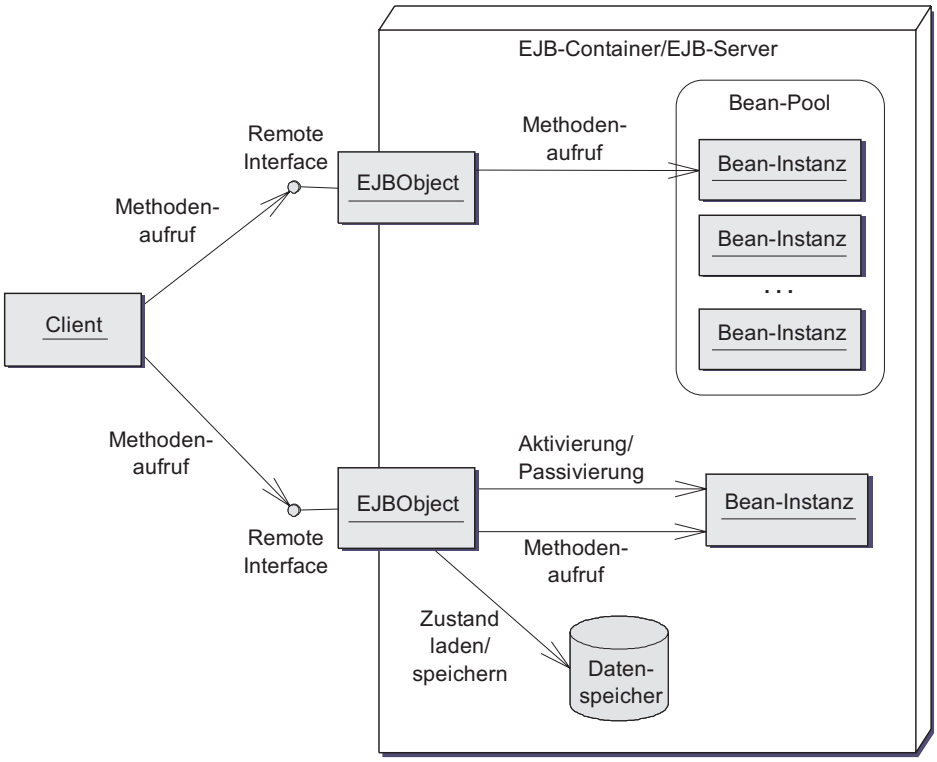


Abbildung 2.19: Kernkonzepte von Session-Beans

- **Remote/Local-Schnittstelle:** Enthält die öffentliche Schnittstelle mit den Geschäftsmethoden der Komponente, die Clients zur Verfügung steht. Bei Verwendung einer `Local`-Schnittstelle, müssen sie sich im selben Prozeß, wie die Bean befinden. Dadurch kann der Mehraufwand, der durch *Stubs* und *Skeletons* sowie durch das Netzwerkprotokoll entsteht, vermieden werden.

- **Home/LocalHome-Schnittstelle:** Enthält Methoden zum Anlegen und Löschen von Beans. Die `LocalHome`-Schnittstelle muß verwendet werden, wenn die Beans über ihre `Local`-Schnittstelle angesprochen werden sollen.

- **Deployment-Deskriptor:** Spezifiziert die Eigenschaften der *Session-Bean*. Neben den allgemeinen Informationen (vgl. dazu 2.5.3), ist hier vor allem die Angabe der verwendeten Unterart (zustandslos oder zustandsbehaftet) hervorzuheben.

Zusätzlich muß eine Session-Bean `ejbCreate`-Methoden implementieren, die den korrespondierenden `create`-Methoden aus der Home-Schnittstelle entsprechen. Die Home-Schnittstelle wird vom Home-Objekt implementiert und wird zum Auffinden einer Bean sowie zur Steuerung deren Lebenszyklus aus Anwendungssicht benötigt. Die `ejbCreate`-Methode wird immer aufgerufen, wenn ein Client eine Bean mittels der gewünschten `create`-Methode im Home-Objekt anfordert. Eine Session-Bean muß mindestens eine `create`-Methode besitzen. Zu den eben genannten technisch motivierten Methoden kommen schließlich noch die fachlich mo-

Methode	Beschreibung
`setSessionContext()`	Wird einmal bei Erzeugung der Bean ausgeführt. Übergibt den Kontext, mit dem die Bean Dienste des Containers nutzen kann.
`ejbPassivate()`	Zustandsbehaftete Session-Beans können passiviert werden, falls zu viele Exemplare existieren. Dabei wird der Zustand in einer Datenbank oder im Dateisystem zwischengespeichert. Methode wird vor der Passivierung aufgerufen und kann z.B. zum Schließen von Ressourcen verwendet werden.
`ejbActivate()`	Umgekehrter Vorgang zur Passivierung zustandsbehafteter Session-Beans. Methode wird unmittelbar nach der Aktivierung aufgerufen und sollte zur Wiederherstellung von Ressourcen, die bei der Passivierung freigegeben wurden, genutzt werden.
`ejbRemove()`	Methode wird am Ende des Lebenszyklus aufgerufen, bevor sie vernichtet wird und kann z.B. zum Freigeben von belegten Ressourcen verwendet werden.

Tabelle 2.2: Interface einer Session-Bean

tivierten Geschäftsmethoden hinzu, die in der Remote-Schnittstelle (bzw. Local-Schnittstelle) der Session-Bean definiert sind.

Entity-Beans

Eine Entity-Bean repräsentiert ein serverseitiges Geschäftsobjekt, dessen Zustand sich in einem dauerhaften Speichermedium, wie z.B. einer Datenbank, befindet. Damit unterscheiden sie sich wesentlich von Session-Beans, die selbst nicht persistent sind und eher Geschäftsprozesse repräsentieren. Das Kernkonzept dieser Komponentenart ist in Abbildung 2.20 dargestellt (in Anlehnung an [Rom02]).
Zur Präzisierung des Begriffs *Entity-Bean* wird dabei zwischen *Instanz* und den zugehörigen *Daten* unterschieden [Rom02]. Die Instanz wird aus der Entity-Bean-Klasse erzeugt und repräsentiert zur Laufzeit der Anwendung das Speicherabbild der Daten, die sich physikalisch in einem Datenspeicher befinden. Die Daten einer Entity-Bean werden mittels `ejbLoad()` von der Datenbank gelesen (1, 2) und der Entity-Bean-Instanz zugewiesen. Anschließend stehen die Daten als Attributausprägungen (Zustand) der Entity-Bean-Instanz beim Aufruf von Geschäftsmethoden (3) zur Verfügung. Vorgenommene Attributänderungen in der Instanz müssen durch `ejbStore()` persistent gemacht werden (4, 5), um dauerhaft erhalten zu bleiben. Das Laden und Speichern der Daten findet je nach Bedarf mehrfach statt. Die Entscheidung darüber, wann der Aufruf der Load/Store-Methoden erfolgt, obliegt unter Aufrechterhaltung des transaktionalen Kontextes vollständig dem EJB-Container. Zusätzlich sorgt der Container dafür, daß immer nur ein Client (Thread) zu einem Zeitpunkt mit einer Entity-Bean-Instanz arbeitet. Um das Leistungsverhalten der Anwendung zu verbessern, kann der Container mehrere Instanzen des gleichen Bean-Typs bereitstellen, um mehrere Clients simultan zu bedienen (z.B. durch *Pooling* von

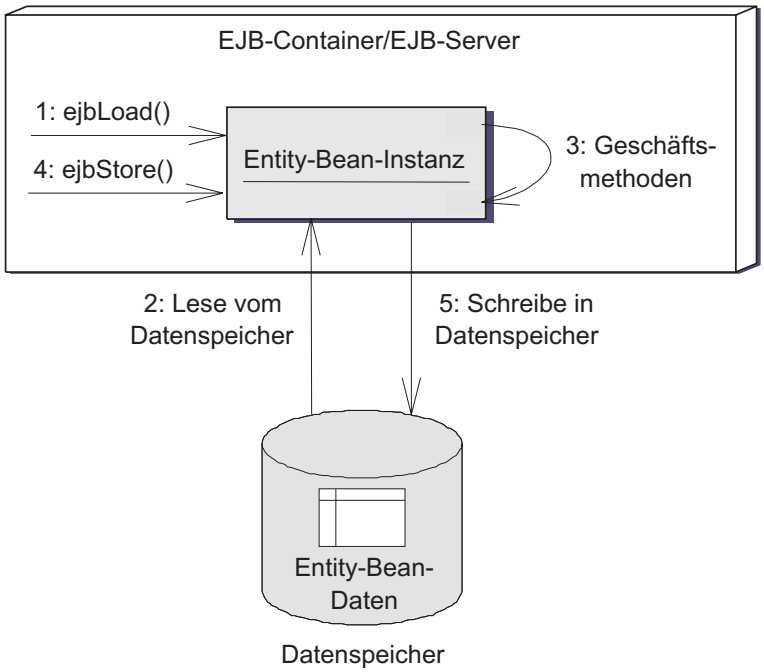


Abbildung 2.20: Kernkonzept von Entity-Beans

Entity-Bean-Instanzen). Falls dabei mehrere Beans die selben Daten repräsentieren, wird die Konsistenz der Daten bei der Synchronisation mit dem Datenspeicher (`ejbLoad(), ejbStore()`) durch das Transaktionsmanagement des Containers sichergestellt [Rom02]. In den Load/Store-Methoden werden die erforderlichen Zugriffe auf den Datenspeicher implementiert, um die Daten der Entity-Bean-Instanz persistent zu machen. Anhand dieser Implementierung können zwei unterschiedliche Unterarten von Entity-Beans unterschieden werden:

- **Manuelle Persistenz (*bean-managed persistence*)**: Der Anwendungsentwickler muß die Methoden vollständig selbst implementieren. Hier müssen z.B. SQL-Zugriffe auf eine relationale Datenbank entwickelt werden, die mittels JDBC abgesetzt werden. Das zurückgelieferte Ergebnis muß ausgelesen und in den Attributen der Bean-Instanz abgelegt werden. Durch manuelle Persistenz erhält der Entwickler die Möglichkeit nahezu alle in Java integrierbaren Persistenzmechanismen zu nutzen und Fälle zu implementieren, die durch die automatische, vom Container angebotene Persistenz, nicht gelöst werden.

- **Automatische Persistenz (*container-managed persistence)***: Der EJB-Container implementiert die Methoden vollständig selbst. Der Anwendungsentwickler muß mittels des Deployment-Deskriptors deklarieren, wie die Attribute einer Entity-Bean persistent gemacht werden sollen. Dazu gehört z.B. bei einer relationalen Datenbank die Zuordnung von Entity-Beans zu Tabellen und Bean-Attributen zu Tabellen-Attributen (*Object-Relational Mapping*, vgl. dazu z.B. allgemein [Fus97] und speziell für EJBs [Rom02]). Der Container generiert daraufhin die notwendigen SQL-Statements, JDBC-Befehle und Routinen zur Belegung der Bean-Attribute. Es muß keine Auseinandersetzung mit JDBC-Details durch den Anwendungsentwickler erfolgen. Codierungsaufwand wird eingespart

und die Bean-Implementierung bleibt frei von technisch motivierten JDBC-Konstrukten.

Zur Implementierung von Entity-Beans müssen die folgenden Elemente bereitgestellt werden:

- **Implementierung der Entity-Bean:** Enthält die Geschäftsattribute und -methoden. Zur Verwaltung des Lebenszyklus durch den EJB-Container muß zusätzlich die Schnittstelle `javax.ejb.EntityBean` implementiert werden. In Tabelle 2.3 sind die zu implementierenden Methoden aufgeführt. Entity-Beans, die nach dem EJB 2.0-Standard entwickelt werden, dürfen direkt keine persistenten Attribute mehr deklarieren, sondern nur noch `get/set`-Methoden, die vom EJB-Container in einer abgeleiteten Klasse implementiert werden. Diese Klasse enthält dann auch die tatsächlichen Attribute [DeM01].

- **Remote-Schnittstelle:** Enthält die öffentliche Schnittstelle mit den Geschäftsmethoden der Komponente, die Clients zur Verfügung steht. Seit EJB 2.0 existiert zusätzlich die Möglichkeit lokale Schnittstellen zu definieren, die anzeigen, daß die Bean nur von Clients, die sich auf der selben Maschine bzw. dem selben Applikations-Server befinden, aufgerufen werden kann. Dadurch kann der Mehraufwand, der durch *Stubs* und *Skeletons* sowie durch das Netzwerkprotokoll entsteht, vermieden werden.

- **Home-Schnittstelle:** Enthält Methoden zum Suchen, Anlegen und Löschen von Beans (Tabelle 2.4). Entity-Beans, die nach dem EJB 2.0-Standard implementiert sind, können ebenfalls eine lokale Home-Schnittstelle definieren, falls sie über eine lokale Schnittstelle verfügen. Zusätzlich ermöglichen `home`-Methoden die Implementierung datenunabhängiger Funktionalität für alle Entity-Beans. Die Suche wird bei Entity-Beans durch die standardisierte, an SQL angelehnte Sprache *Enterprise JavaBeans-Query Language (EJB-QL)* ermöglicht. Der erweiterte Standard soll dabei dazu beitragen, daß herstellerunabhängige Entity-Beans implementiert werden können.

- **Primärschlüsselklasse:** Dient zur eindeutigen Identifizierung von Entity-Beans. Ein Primärschlüsselobjekt kann dabei so viele Attribute enthalten, wie notwendig sind, um Entity-Beans voneinander eindeutig zu unterscheiden. Das Objekt muß serialisierbar sein.

- **Deployment-Deskriptor:** Spezifiziert die Eigenschaften der *Entity-Bean*. Neben den allgemeinen Informationen (vgl. dazu 2.5.3) sind bei automatischer Persistenz vor allem Informationen über den Datenspeicher und die Zuordnung der Entity-Beans zu diesem enthalten.

Message-Driven-Beans

Die *Message-Driven Beans* sind mit dem EJB 2.0-Standard neu eingeführt worden und beruhen auf dem in der J2EE-Plattform integrierten JMS. Sie ermöglichen somit die asynchrone Kommunikation zwischen Client und Server. D.h. ein Client kann Server-Dienste benutzen, ohne anschließend zu blockieren. Dies kann dazu genutzt werden, andere Funktionen auszuführen, während der Client auf das Ergebnis der ursprünglichen Anfrage wartet. Zur Verwendung von asynchroner Kommunikation muß zunächst eine *Domäne* zur Bestimmung der Kommunikationsart gewählt werden. Es existieren die folgenden Domänen [Hap01, Rom02]:

Methode	Beschreibung
`setEntityContext()`	Wird einmal bei Erzeugung der Bean ausgeführt. Übergibt den Kontext, mit dem die Bean Dienste des Containers nutzen kann.
`unsetEntityContext()`	Wird aufgerufen, bevor eine Entity-Bean-Instanz aus dem Instanzen-Pool entfernt wird.
`ejbPassivate()`	Wird aufgerufen, bevor eine Entity-Bean-Instanz zurück in den Instanzen-Pool gelegt wird. Der Vorgang wird als Passivierung bezeichnet.
`ejbActivate()`	Umgekehrter Vorgang zur Passivierung. Wird aufgerufen, nachdem die Instanz einer Entity-Bean aus dem Pool entnommen und mit dem vom Client geforderten Zustand versehen wurde.
`ejbRemove()`	Methode wird am Ende des Lebenszyklus aufgerufen und löscht den Zustand der Entity-Bean im permanenten Speicher.
`ejbLoad()`	Wird aufgerufen, um den Zustand einer Bean-Instanz mit den Werten aus dem Datenspeicher zu belegen. Muß nur bei manueller Persistenz implementiert werden.
`ejbStore()`	Wird aufgerufen, um den Zustand der Bean-Instanz im Speicher abzulegen. Muß nur bei manueller Persistenz implementiert werden.

Tabelle 2.3: Interface einer Entity-Bean

- **Publish/Subscribe**. Viele Nachrichtenproduzenten können über *Topics* mit vielen Nachrichtenkonsumenten kommunizieren. Dabei wird jede Nachricht an jeden Konsumenten geschickt und von ihm verarbeitet.

- **Point-to-Point**. Viele Nachrichtenproduzenten können Nachrichten in eine Nachrichtenschlange (*Queue*) stellen. Es kann zwar mehrere Konsumenten geben, die Nachrichten entnehmen, dabei wird jede Nachricht aber nur genau einmal durch einen Konsumenten verarbeitet.

Die unterschiedlichen Modelle sind in Abbildung 2.21 veranschaulicht [Rom02].
Eine Message-Driven-Bean (MDB) kann Nachrichten entsprechend der erläuterten Domänen auf dem Applikations-Server konsumieren. Sie besitzt dabei die folgenden Eigenschaften [Rom02, Hap01]:

- **Keine Home- und keine Remote-Schnittstellen**, da ein Client nur indirekt über JMS-Mechanismen Dienste der Bean in Anspruch nehmen kann.

- **Nur eine schwach typisierte Geschäftsmethode**. Die Methode `onMessage()` nimmt Objekte vom JMS-Typ `Message` entgegen. Konkrete Implementierungen dieser Schnittstelle sind `BytesMessage`, `ObjectMessage`, `TextMessage`, `StreamMessage`

Methode	Beschreibung
`ejbFind`$<\ldots>$`(`$<\ldots>$`)`	Methode wird aufgerufen, um Entity-Beans im Datenspeicher zu suchen. Muß nur bei manueller Persistenz implementiert werden.
`ejbSelect()`	Hilfsmethode zur Durchführung von EJB-QL-Anfragen bei autom. Persistenz. Nicht durch Clients aufrufbar.
`ejbHome`$<\ldots>$`()`	Implementiert eine globale Operation, die unabhängig von den Daten ist. Z.B. das Zählen von vorhandenen Datensätzen.
`ejbCreate(`$<\ldots>$`)`	Zur Erzeugung und Initialisierung von Entity-Beans und damit verbundenen Datensätzen im permanenten Speicher. Falls Clients keine Datensätze erzeugen sollen, kann auf die Implementierung verzichtet werden.
`ejbPostCreate(`$<\ldots>$`)`	Muß für jede ejbCreate-Methode implementiert werden. Wird unmittelbar nach ejbCreate() aufgerufen und kann zur Fortsetzung der Initialisierung verwendet werden.

Tabelle 2.4: Interface eines Home-Objekts für Entity-Beans

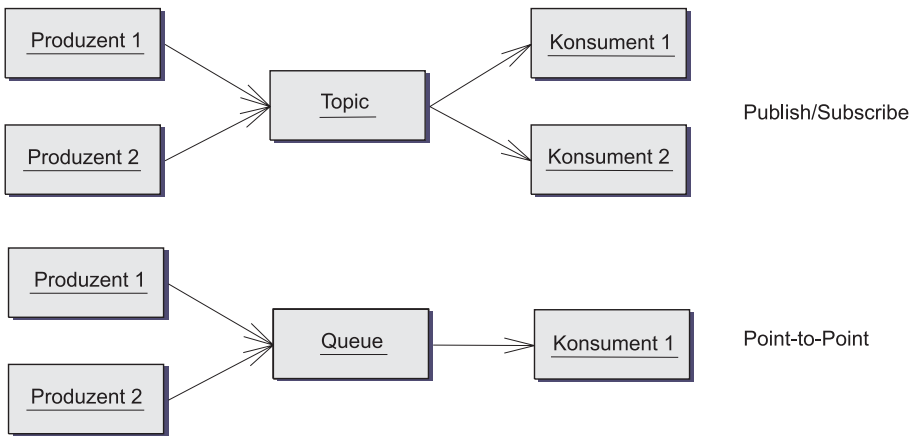


Abbildung 2.21: JMS-Domänen

und `MapMessage`. Zur Laufzeit muß die eingegangene Nachricht auf ihren Typ und Inhalt analysiert werden.

- **Keine Rückgabewerte und keine Exceptions**, da der Produzent der Nachricht vom Konsumenten der Nachricht entkoppelt ist und nicht blockiert, um auf eine Antwort zu warten. Die EJB-Spezifikation erlaubt bei diesem Bean-Typ nur das Erzeugen von System-Exceptions, die vom EJB-Container verarbeitet werden können. Verboten ist das Erzeugen von anwendungsspezifischen Ausnahmen.

- **Zustandslosigkeit**. Alle Bean-Instanzen sind anonym und besitzen keine Identität, die für Clients sichtbar ist. Die Instanzen eines Bean-Typs werden vom EJB-Container als identisch angesehen.

- **Dauerhafte (*durable*) oder nicht dauerhafte (*non-durable*) Konsumenten**. Durch die Deklaration von dauerhaften Konsumenten ist es möglich die Auslieferung von Nachrichten an diese auch dann zu gewährleisten, wenn sie nicht aktiv sind. Alle Nachrichten werden dann erhalten bis der Konsument wieder verfügbar ist und an ihn ausgeliefert. Die Lebensdauer einer Nachricht kann dabei allerdings beschränkt werden und ohne ausgeliefert zu werden, verfallen.

- **Keine EJB-Sicherheitsmechanismen**. Im Gegensatz zu Session- und Entity-Beans können hier keine automatischen Zugriffsberechtigungsprüfungen erfolgen. Dies muß von der Anwendung selbst erledigt werden.

Zur Umsetzung einer MDB müssen die Schnittstellen `javax.ejb.MessageDrivenBean` und `javax.jms.MessageListener` implementiert werden. Dabei sind die in Tabelle 2.5 aufgeführten Methoden zu implementieren.

Methode	Beschreibung
`setMessageDrivenContext()`	Wird einmal bei Erzeugung der Bean ausgeführt. Übergibt den Kontext, mit dem die Bean Dienste des Containers nutzen kann.
`onMessage()`	Übergibt eine Nachricht und stellt damit die Methode zur Implementierung der Geschäftslogik dar. Methode wird für jede konsumierte Nachricht aufgerufen.
`ejbRemove()`	Methode wird am Ende des Lebenszyklus vor der Vernichtung aufgerufen und kann z.B. zum Freigeben von belegten Ressourcen verwendet werden.

Tabelle 2.5: Interface einer Message-Driven-Bean

Obwohl keine Home-Schnittstelle existiert, muß dennoch eine `ejbCreate`-Methode implementiert werden, die vom EJB-Container bei der Erzeugung der Bean aufgerufen wird. Im Deployment-Deskriptor werden u.a. die folgenden Eigenschaften einer Message-Driven-Bean spezifiziert:

- **Logischer Name der Bean.** Legt den Namen der Bean fest, unter dem sie im Deployment-Deskriptor referenziert wird.
- **Bean-Klasse.** Vollqualifizierter Name der Bean-Klasse.
- **Transaktionssteuerung.** Legt fest, ob die Transaktionssteuerung durch den Container oder durch die Bean selbst erfolgt.
- **JMS-Domäne.** Legt fest, in welcher Domäne die Bean Konsument ist (*Topic* oder *Queue*).

Die namentliche Zuordnung zwischen Bean und *Queue* oder *Topic* erfolgt *nicht* im durch die EJB-Spezifikation festgelegten Deployment-Deskriptor, sondern in einem vom Hersteller des Applikations-Servers spezifizierten zusätzlichen Deployment-Deskriptor. Dieses Vorgehen erhöht die Portabilität von Beans, da die Benennung von *Queues* und *Topics* auf verschiedenen Applikations-Servern unterschiedlich sein kann [Rom02]. Aufgrund der Entkopplung zwischen Clients und Beans kann eine MDB nicht an der Transaktion des Produzenten teilnehmen. Bei dauerhaften (durable) Nachrichten sind grundsätzlich zwei Transaktionen beteiligt. Die Transaktion des Clients reiht die Nachricht in die Warteschlange ein und die Transaktion der Bean entnimmt die Nachricht aus der Warteschlange.

2.5.3 Deployment-Deskriptor

Beim Deployment-Deskriptor handelt es sich um eine Datei im XML-Format, die alle deklarativen Informationen beinhaltet, die nicht direkt im Code der EJBs festgehalten sind. Die Informationen lassen sich in zwei grundlegende Arten unterteilen [DeM01]:

1. **Struktur:** Beschreibt den Aufbau einer Bean und ihre externe Abhängigkeiten. Diese Information muß enthalten sein, wenn ein Java-Archiv ausgeliefert wird. Beispiele für strukturelle Informationen sind:
   - Logischer Name für die Bean, Name für Bean-Klasse, Remote-Schnittstelle und Home-Schnittstelle.
   - Art der EJB: `session`, `entity` oder `message-Driven`.
   - Transaktionstyp für Session- und Message-Driven Beans.
   - Zustandstyp von Session-Beans.
   - Persistenztyp für Entity-Beans und zugehörige Primärschlüsselklasse.
   - Ressourcen-Referenzen, die von der EJB verwendet werden.
   - Referenzen auf andere EJBs.
2. **Anwendungszusammensetzung:** Beschreibt, wie eine oder mehrere EJBs zu einer größeren Anwendung zusammengesetzt werden. Diese Angaben sind optional, da Beans auch ohne diese Informationen funktionieren. Beispiele für Informationen über die Anwendungszusammensetzung sind:
   - Umgebungsvariablen können geändert oder neu hinzugefügt werden.

- Beschreibungsfelder können geändert oder neu hinzugefügt werden.
- Namen für Beziehungen.

In Abbildung 2.22 ist exemplarisch ein Auszug eines Deployment-Deskriptors dargestellt. Es wird eine zustandsbehaftete Session-Bean `BankEJB` zur Kontoführung definiert. Es wird eine Fließkommazahl `Zinssatz` mit dem Wert `3.5` als Umgebungsvariable definiert. Die Transaktionssteuerung der Bean-Methoden wird vom EJB-Container übernommen und die dargestellte Methode `get()` muß immer innerhalb einer Transaktion ablaufen (Transaktionstyp `Required`, vgl. dazu Transaktionsdienst in Abschnitt 2.5.4).

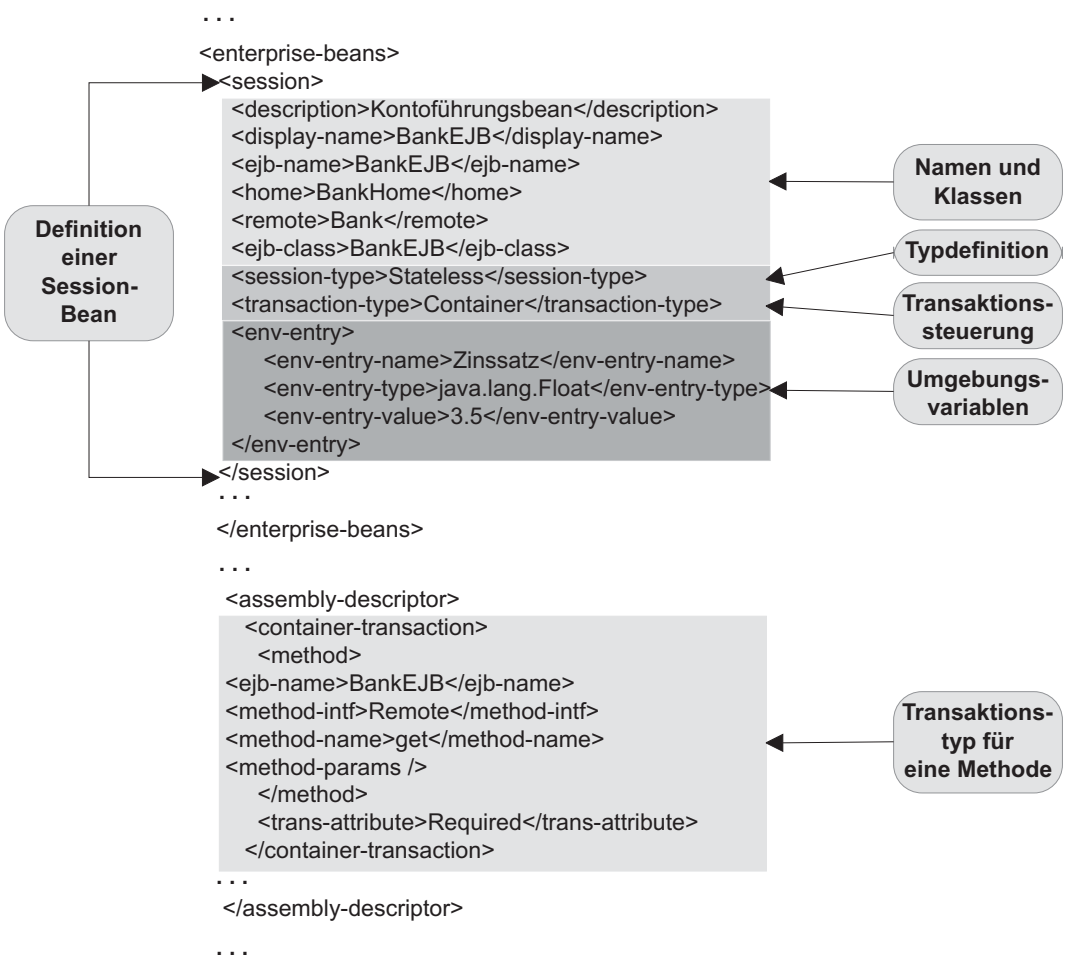


Abbildung 2.22: Auszug aus einem Deployment-Deskriptor

2.5.4 Laufzeitsystem

Wie bereits im vorigen Abschnitt erläutert, stellt das Laufzeitsystem das Fundament für serverseitige EJB-Komponenten dar. Laufzeitsysteme für EJBs sind in Form zahlreicher kommerzieller und nichtkommerzieller Implementierungen von Applikations-Servern verfügbar. Beispiele

für kommerzielle Server sind *Bea Weblogic*, *IBM Websphere* und *Iona iPortal*. Nichtkommerzielle Implementierungen stellen z.B. *JBoss* der *JBoss Group*, die *J2EE-Referenzimplementierung (J2EE-RI)* von *Sun* und *Jonas* von *Evidian* dar. Ein hervorzuhebender Aspekt dabei ist, daß die Implementierung eines Applikations-Servers, der die Laufzeitumgebung (Container) für EJBs bereitstellt, weitestgehend frei durch die Hersteller erfolgen kann. Lediglich das Verhalten und die Programmierschnittstelle werden durch den EJB-Standard festgelegt. Als Resultat daraus existieren eine Reihe von Applikations-Servern, deren Implementierung auf CORBA-Produkten basiert, die zur Kommunikation einen ORB, zur Transaktionssteuerung einen CORBA-Transaktionsdienst und zum Auffinden von EJBs einen CORBA-Namensdienst verwenden. Aufgrund der einheitlichen Programmierschnittstelle bleiben diese Aspekte jedoch für den Anwendungsentwickler im Verborgenen. Der *Borland Applicationserver* ist ein Vertreter dieser Gattung, dessen Aufbau in [Vog, Cor98] beschrieben wird. Nachfolgend werden Aufgaben und Eigenschaften wichtiger Basisdienste, die in den Implementierungen enthalten sein müssen, kurz erläutert.

Kommunikationsdienst

Der Kommunikationsdienst sorgt dafür, daß Clients Methoden von EJBs aufrufen können. Aus der Remote-Schnittstelle von EJBs werden auf Basis von Werkzeugen des verwendeten Applikations-Servers *Stubs* und *Skeletons* erzeugt, die zur Kommunikation über den jeweils implementierten Mechanismus verwendet werden. Als Mechanismen kommen z.B. RMI (Abschnitt 2.3) und CORBA (Abschnitt 2.2) in Betracht und damit auch eine Reihe unterschiedlicher Protokolle, die zur Kommunikation verwendet werden können. Eine Reihe von neu entwickelten Applikations-Servern, wie z.B. *JBoss* [JBo], basieren auf dem RMI-Protokoll JRMP oder auf proprietären RMI-Protokollen, wie z.B. *Jeremie* von *Jonas* [Evi]. Einige andere Applikations-Server sind aus weit verbreiteten CORBA-Produkten entstanden, die IIOP zur Kommunikation verwenden. Zu dieser Gattung gehören z.B. der *Borland Applicationserver*, der auf dem ORB *Visibroker* basiert [Bor] und der Server *iPortal* der Firma *IONA*, der auf dem ORB *OrbixWeb* basiert [Ion]. Durch das Komponentenparadigma (vgl. Abschnitt 2.1.2) bleiben diese Details vor dem Anwendungsentwickler verborgen. Er verwendet die vom Kommunikationsprotokoll unabhängige RMI-IIOP-API zur Implementierung von EJBs [Sha01a].

Namensdienst

Der Namensdienst ist eine Schlüsselkomponente eines Applikations-Servers und besitzt eine durch die JNDI-Spezifikation gesicherte Standardschnittstelle, die Java-basierten Programmen die Verwendung von Namens- und Verzeichnisdiensten erlaubt. Damit ist eine Trennung zwischen Schnittstelle und Implementierung gewährleistet, die eine einheitliche Nutzung von verschiedenen Namensdiensten ermöglicht.
Die JNDI-Architektur besteht aus der *Client-API* und dem *Service Provider Interface (SPI)* und ist in Abbildung 2.23 schematisch dargestellt.
Beispiele für integrierte Namens- und Verzeichnisdienste sind [Rom99]:

- *Object Naming Service (COSNaming),*
- *RMI Registry,*

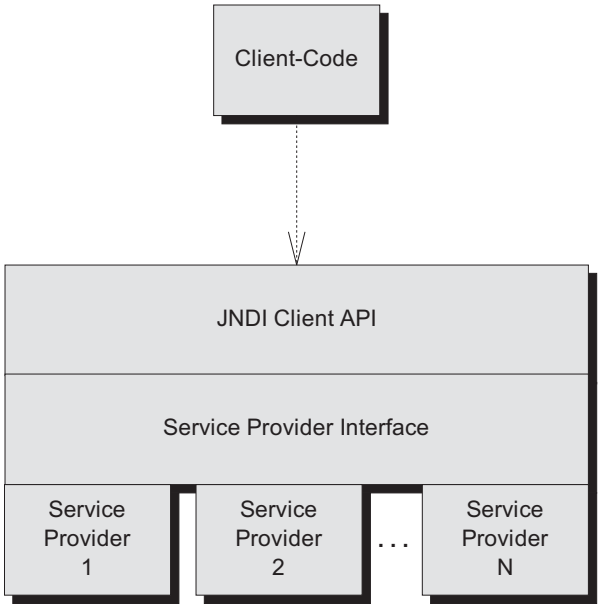


Abbildung 2.23: JNDI-Architektur

- *Lightweight Directory Acces Protocol (LDAP),*
- *Network Information Service (NIS).*

Somit kann z.B. ein Applikations-Server, der auf Basis von CORBA entwickelt wurde, den CORBA-Namensdienst COSNaming verwenden. Die Verwendung dieses Dienstes ist für Anwendungsentwickler transparent, da für sie nur die JNDI-Programmierschnittstelle relevant ist. Der JNDI-Dienst ist hierarchisch aus Kontexten (*Contexts*) und Bindungen (*Bindings*) zusammengesetzt. Bindungen ordnen Namen Objekte zu. Bindungen sind Bestandteil von Kontexten und Kontexte können ineinander geschachtelt werden (Subkontexte (*Subcontext*)). Als Zugang zu einem solchen Namenssystem dient der Initiale Kontext (*Initial Context*). Daraus ergibt sich verallgemeinert die in Abbildung 2.24 dargestellte Struktur eines JNDI-Namensystems.
Als eine Schlüsselfunktion ermöglicht der Namensdienst das Auffinden von EJBs durch die Vorhaltung zugehöriger `EJBHome`-Objekte unter bestimmten Namen, die beim *Deployment* festgelegt werden können. Darüber hinaus werden auch wichtige Ressourcen im Namensdienst eingetragen und verfügbar gemacht. Dazu gehört z.B.:

- Die Umgebung einer EJB in Form von Schlüssel/Wert-Paaren.
- Referenzen auf Objekte, die den Zugriff auf Datenquellen ermöglichen.
- Transaktionsdienstschnittstelle, die von Anwendungsprogrammen manuell genutzt werden kann.

Die Verwendung in EJBs folgt aufgrund der JNDI-Schnittstelle und der RMI-IIOP-Programmierschnittstelle immer dem selben Schema, das dem nachfolgenden Code-Fragment entnommen werden kann:

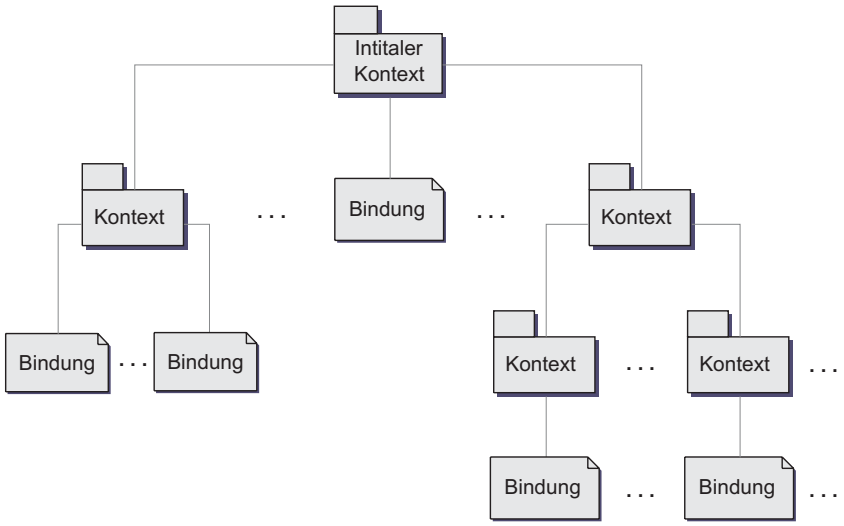


Abbildung 2.24: JNDI-Struktur

```
Properties env = new Properties();
env.setProperty (
 "java.naming.factory.initial",
 "org.jnp.interfaces.NamingContextFactory");

env.setProperty (
 "java.naming.provider.url",
 "localhost:1099");

env.setProperty (
"java.naming.factory.url.pkgs",
"org.jboss.naming");

Context initial = new InitialContext(env);

Object objref = initial.lookup("MyAccount");

AccountHome home =
      (AccountHome) PortableRemoteObject.narrow(
            objref,
            AccountHome.class);
```

Mit Hilfe des im Code-Fragment dargestellten `Properties`-Objekts und den darin enthaltenen Schlüsseln `java.naming.factory.initial`, `java.naming.provider.url` und `java.naming.factory.url.pkgs` wird spezifiziert, wie der JNDI-Dienst des verwendeten Applikations-Servers benutzt werden kann und wo sich dieser im Netzwerk befindet. Hier wird der lokal verfügbare Applikations-Server JBoss verwendet, dessen Namensdienst Anfragen auf dem Netzwerk-Port 1099 entgegennimmt. Mit dieser Information kann der initiale Kontext ermittelt werden und mittels der Methode `lookup()` nach dem Home-Objekt der

gewünschten Bean gesucht werden. Die gefundene Objektreferenz muß gemäß der RMI-IIOP-Spezifikation mit der Methode `narrow()` des Objekts `PortableRemoteObject` in das Home-Objekt der Bean umgewandelt werden. Danach können mit Hilfe der Methoden dieses Objekts auf dem Server Beans erzeugt und verwendet werden. Die Namen unter dem die Home-Objekte von Beans im Namensdienst verfügbar sind, werden im Deployment-Deskriptor definiert. Eine Anmeldung durch den Anwendungscode beim Namensdienst entfällt damit. Bei proprietären Protokollen, kann die Kontaktaufnahme zu Beans anders aussehen und erschwert damit die Portabilität der Anwendung.

Transaktionsdienst

Der Transaktionsdienst realisiert Transaktionen mit den bereits erläuterten ACID-Eigenschaften (vgl. Abschnitt 2.1) innerhalb der J2EE-Umgebung. Unterstützt werden lokale und globale Transaktionen sowie das 2PC-Protokoll. Der Dienst basiert auf den Spezifikationen JTS [Che99] und JTA [Che01]. JTS spezifiziert den Aufbau eines Transaktionsmonitors, der zur Umsetzung des Transaktionskonzepts notwendig ist. Bei den Funktionen handelt es sich um eine kompatible Untermenge zum OTS 1.1 der OMG [Gro97a], wie er im Rahmen des CORBA-Standards definiert ist [Den00] (vgl. Abschnitt 2.2). Im Gegensatz zum CORBA-Transaktionsdienst OTS werden z.B. keine geschachtelten Transaktionen unterstützt und nur sehr wenige Annahmen über den Ressourcenmanager gemacht. JTS und JTA sind vornehmlich auf die Zusammenarbeit mit Datenbanken ausgelegt [Til99]. Die JTS-Spezifikation stellt eine Verpflichtung für die Hersteller von Transaktionsdiensten dar. Um die geforderte Funktionalität zu erbringen, kann auch auf eine Implementierung von CORBA-OTS zurückgegriffen werden. Dies bietet sich insbesondere dann an, wenn ein Applikations-Server auf Basis eines ORBs realisiert wird. Für den Anwendungsentwickler ist i.d.R. nur die in JTA spezifizierte Schnittstelle zur manuellen Transaktionssteuerung relevant. Aufgrund des deklarativen Programmiermodells von EJBs beschränkt sich bei automatischer Transaktionssteuerung durch den EJB-Container die Auseinandersetzung mit Transaktionen auf die korrekte Deklaration von Transaktionsattributen für die Methoden der implementierten EJBs im Deployment-Deskriptor. Der Anwendungscode bleibt dabei völlig unberührt. Der für jede Methode notwendige Transaktionskontext wird, wie bereits in Abschnitt 2.5.1 beschrieben durch das EJB-Objekt mit dem Transaktionsdienst vereinbart. Als Transaktionsattribute stehen die folgenden Größen für Bean-Methoden zur Verfügung:

- **NotSupported.** Die Methode unterstützt keine Transaktionen und wird damit nicht innerhalb einer Transaktion ausgeführt. Eine evtl. vom Client propagierte Transaktion wird ignoriert. Bei eventuellen Fehlersituationen kann kein *Rollback* erfolgen.
- **Required.** Die Methode wird immer innerhalb einer Transaktion ausgeführt. Eine vom Client propagierte globale Transaktion wird übernommen, sonst wird eine neue globale Transaktion erzeugt. Aufgrund der Vererbung des Transaktionskontextes ist der Zugriff auf weitere EJBs und transaktionale Systeme ebenfalls transaktionsgesichert. Globale Transaktionen sind teure Operationen. Deren Vermeidung kann zur Steigerung des Leistungsverhalten beitragen [Den00].
- **Supports.** Die Methode wird mit oder ohne Transaktion ausgeführt. Bei Aufruf der Methode mit einer Transaktion, wird diese Transaktion übernommen. Liegt keine Transak-

tion vor, läuft die Methode ohne Transaktion. Supports kann zur Optimierung des Leistungsverhaltens herangezogen werden. Es muß allerdings sichergestellt werden, daß die Methodenimplementierung ihre Aufgabe in beiden Fällen erfüllt [Den00].

- **RequiresNew.** Die Methode hat ihre eigene globale Transaktion, die vor dem Beginn der Methode neu gestartet wird. Nach Ablauf der Methode erfolgt immer ein *Commit* durch den EJB-Container. Eine evtl. vom Client erzeugte Transaktion bleibt isoliert. Bei Aufruf von anderen transaktionalen Systemen und Beans, wird die Transaktion vererbt.

- **Mandatory.** Die Methode muß vom Client mit einer globalen Transaktion aufgerufen werden, sonst tritt ein Fehler auf. Die Transaktion des Clients wird zur Methodenausführung verwendet und an andere Beans und transaktionale Systeme weitergegeben.

- **Never.** Die Methode darf vom Client niemals innerhalb einer Transaktion aufgerufen werden und wird immer ohne Transaktion ausgeführt. Dies hat wiederum zur Folge, daß bei Fehlersituationen kein *Rollback* erfolgen kann.

Tabelle 2.6 listet auf, wie Transaktionsattribute mit den verschiedenen Bean-Arten verwendet werden können.

Transaktions-attribut	Zustandslose Session-Bean	Zustandsbeh. Session-Bean mit Synchronisation	Entity-Bean	Message--Driven-Bean
Required	Ja	Ja	Ja	Ja
RequiresNew	Ja	Ja	Ja	Nein
Mandatory	Ja	Ja	Ja	Nein
Supports	Ja	Nein	Nein	Nein
NotSupported	Ja	Nein	Nein	Ja
Never	Ja	Nein	Nein	Nein

Tabelle 2.6: Transaktionsattribute und Bean-Typ [Rom02]

Nachfolgend ist ein Beispiel für die Deklaration eines Transaktionsattributs dargestellt. Die Methode `searchAccounts()` der Bean `BankEJB` verwendet das Attribut `Required`.

```
 <container-transaction>
      <method>
<ejb-name>BankEJB</ejb-name>
<method-intf>Remote</method-intf>
<method-name>searchAccounts</method-name>
<method-params>
  <method-param>int</method-param>
  <method-param>int</method-param>
</method-params>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
```

Zur Synchronisation von Session-Beans mit den Container-Transaktionen kann das Interface `javax.ejb.SessionSynchronisation` implementiert werden, das aus drei Methoden besteht, die vom EJB-Container in Abhängigkeit von bestimmten Transaktionszuständen aufgerufen werden. Die `afterBegin`-Methode teilt der Bean mit, daß die folgenden Methodenaufrufe in einer neuen Transaktion ausgeführt werden. Dies kann z.B. für Caching-Mechanismen genutzt werden [Den00]. Die `beforeCompletion`-Methode wird vor dem Ende einer Transaktion aufgerufen, wenn noch nicht bekannt ist, ob ein *Commit* oder *Rollback* erfolgt (es kann auch ein *Rollback* mittels `setRollbackOnly()` erzwungen werden). Die Methode kann z.B. zum Zurückschreiben von gepufferten Daten oder zur Prüfung zusätzlicher Konsistenzbedingungen verwendet werden [Den00]. Nach Beendigung der Transaktion wird die Methode `afterCompletion()` aufgerufen. Die Bean kann so erfahren, ob ein *Commit* oder ein *Rollback* erfolgte. Die Methode läuft außerhalb der Transaktion ab und kann z.B. zur Ressourcenfreigabe genutzt werden. Beans, die das *SessionSynchronisation*-Interface implementieren, müssen Transaktionen verwenden und dürfen daher nicht die Transaktionsattribute *NotSupported*, *Supports* und *Never* verwenden.

Alternativ zur deklarativen Transaktionssteuerung, kann der Anwendungsentwickler den Transaktionsdienst manuell nutzen, d.h. er greift auf dessen Schnittstelle mittels JTA zu, um Transaktionen zu starten und zu beenden. Zum Zugriff auf den Dienst wird ein Objekt benötigt, das die Schnittstelle `javax.transaction.UserTransaction` implementiert. Die Schnittstelle bietet u.a. Methoden zum Starten (`begin()`), Beenden (`commit()`), Zurücknehmen `rollback()`) und zum Abfragen des Transaktionsstatus (`getStatus()`) an. Das entsprechende Objekt kann vom Applikations-Server aus dem Context, der jeder EJB am Anfang ihres Lebenszyklus übergeben wird, mittels der Methode `getUserTransaction()` bezogen werden. Die manuelle Steuerung von Transaktionen ist aber nicht nur auf Beans beschränkt, sondern kann grundsätzlich von beliebigem Java-Code aus erfolgen. So kann z.B. ein Java-Servlet, das dem Systembenutzer eine HTML-Schnittstelle bietet, auf Beans innerhalb einer Transaktion zugreifen. Ebenso können vom Anwendungs-Client direkt Transaktionen gesteuert werden. Um diese Transaktionsanwendungen zu gewährleisten, kann das Schnittstellenobjekt auch vom Namensdienst (JNDI) des Applikations-Servers bezogen werden. Bei einer expliziten Transaktionssteuerung im Client ist allerdings kritisch zu bemerken, daß die Wartung der Applikation schwieriger werden kann, da sehr viel Programmlogik im Client realisiert werden muß. Vorteilhaft an dieser Vorgehensweise ist, daß Beans auf dem Server einfach in unterschiedlichem Umfang verwendet werden können, ohne sie zu erweitern und zu ändern. Somit ist die manuelle Transaktionssteuerung vom Client aus besonders zur Erstellung von Prototypen und in Anwendungen interessant, die hauptsächlich aus zugekauften Beans bestehen [Den00]. Die Verwendung der manuellen Transaktionsverwaltung ist allerdings bei Entity-Beans durch die Spezifikation ausgeschlossen.

Die Definition des Isolierungsgrades (vgl. Abschnitt 2.1) muß durch vom Server-Hersteller gegebene Konfigurationsmöglichkeiten definiert werden. Dies wird z.B. mittels eines weiteren, herstellerspezifischen Deployment-Deskriptors gelöst, der auch andere Einstellungen, wie z.B. die Größe von verwendeten Bean-Pools erlaubt.

2.6 Entwurfsmuster

Entwurfsmuster beschreiben Probleme und Lösungen, die im Entwurf von objektorientierter Software auftreten. Nachfolgend wird kurz das Prinzip von Entwurfsmustern sowie die dafür erforderlichen Beschreibungselemente dargestellt. Für eine ausführliche Beschreibung von elementaren Entwurfsmustern wird [Gam95] empfohlen.

2.6.1 Prinzip

Entwurfsmuster verfolgen das Ziel, Probleme, die beim Entwurf objektorientierter Software häufig auftreten, zu charakterisieren und die Kernelemente einer Lösung dafür zu beschreiben. Die so entstandene Problemlösung kann von anderen Entwicklern im Rahmen ihrer Designentscheidungen mit einbezogen und unter Anpassungen in der vorliegenden Systemumgebung realisiert werden. Entwurfsmuster stellen damit auch eine Möglichkeit dar, Expertenwissen zu erfassen und zugänglich zu machen.

2.6.2 Aufbau

Um eine einheitliche Strukturierung von Problemen und Lösungsansätzen zu gewährleisten, folgt die Beschreibung von Entwurfsmustern i.d.R. einem bestimmten Schema. Generell wird jedes Entwurfsmuster anhand von vier Elementen beschrieben [Gam95]:

1. **Mustername:** Identifiziert ein Entwurfsmuster und dient der Dokumentation und Kommunikation zwischen Entwicklern.

2. **Problem:** Beschreibt, wann ein Entwurfsmuster angewendet werden kann. Dazu gehört auch die Beschreibung der vorliegenden Probleme, des Umfelds und evtl. auch der Bedingungen, die erfüllt sein müssen, um dieses Entwurfsmuster anwenden zu können.

3. **Lösung:** Beschreibt die Elemente eines Entwurfs, ihre Beziehungen, Verantwortlichkeiten und ihre Zusammenarbeit auf hohem Abstraktionsniveau. Das Muster soll als eine Art Schablone in verschiedenen Situationen angewendet werden können.

4. **Konsequenzen:** Beschreiben das Ergebnis und die Kompromisse, die bei Anwendung des Entwurfsmusters vorliegen. Diese Ausführungen dienen der Beschreibung von Vor- und Nachteilen und dem Verständnis des Kosten/Nutzen-Effekts.

Die verschiedenen Elemente werden als Schablone zur Beschreibung von Entwurfsmustern eingesetzt (*Pattern Template*), um jedes einzelne Muster zu beschreiben und anschließend zu einem Entwurfsmusterkatalog zusammenzufassen. Mit [Gam95] existiert ein umfangreicher Entwurfsmusterkatalog für objektorientierte Systeme, die vielfach auch in verteilten objektorientierten Systemen angewendet werden können. Mittlerweile existieren für den Entwurf von verteilten Systemen ebenfalls eine Reihe von Entwurfsmustern. In [Mow97] findet sich z.B. ein Entwurfsmusterkatalog für CORBA-Systeme. Mit [Dee01] existiert ein Entwurfsmusterkatalog für Systeme, die auf Enterprise-JavaBeans beruhen. Jeder Entwurfsmusterkatalog verwendet eine Entwurfsmusterschablone, die im wesentlichen den vier in [Gam95] genannten Elementen entsprechen. Die Entwurfsmusterschablone in [Dee01] besteht aus den folgenden Elementen:

- Name des Musters (*pattern name*).
- Zusammenhang (*context*). Beschreibt die Umgebung, für die das Muster bestimmt ist.
- Problem (*problem*). Erläutert die adressierten Entwurfsfragen.
- Gründe (*forces*). Beschreibt die Motivation und Gründe zum Einsatz des Musters.
- Lösung (*solution*). Beschreibt den Lösungsansatz und dessen Elemente. Die Lösung besteht aus den folgenden Teilen:
  - Struktur (*structure*). UML-Diagramme zeigen die grundsätzliche Struktur und Sequenzdiagramme beschreiben dynamische Aspekte der Lösung. Alle Bestandteile und ihr Zusammenwirken werden beschrieben.
  - Vorgehensweisen (*strategies*). Beschreibt unterschiedliche Implementierungsansätze des Entwurfsmusters.
- Folgen (*consequences*). Beschreibt die Vor- und Nachteile des Mustereinsatzes.
- Beziehungen zu anderen Mustern (*related patterns*). Erläutert Beziehungen zu anderen Mustern.

2.7 Unified Modeling Language

Die *Unified Modeling Language (UML)* ist eine Sprache zur Beschreibung objektorientierter Modelle, die federführend von Grady Booch, Ivar Jacobson und Jim Rumbaugh als Nachfolgerin verschiedener OO-Modellierungssprachen wie Booch, OMT und OOSE entwickelt wurde. Mittlerweile unterliegt sie den Standardisierungsaktivitäten der OMG und wird in der Spezifikation [Gro00] publiziert. Die UML selbst ist keine Methode, sondern eine reine Sprache und Notation zur Modellierung. Eine Methode muß zusätzlich weitere Faktoren, wie z.B. Rahmenbedingungen des Anwendungsbereichs und das organisatorische Umfeld berücksichtigen. Die UML kann aber als Basis für Methoden verwendet werden [Oes98]. Die visuellen Beschreibungselemente der UML werden in den folgenden Diagrammtypen verwendet [Oes98]:

- **Anwendungsfalldiagramm:** Beschreibt Beziehungen zwischen einer Menge von Anwendungsfällen und den daran beteiligten Akteuren. Es dient damit der Erfassung und Visualisierung von Geschäftsprozessen und -vorfällen.
- **Klassendiagramm:** Stellt die Klassen eines Systems dar und wie sie untereinander in Beziehung stehen.
- **Verhaltensdiagramme**
  - **Aktivitätsdiagramm:** Beschreibt mit Hilfe von Aktivitäten die Ablaufmöglichkeiten eines Systems.
  - **Kollaborationsdiagramm:** Beschreibt eine Menge von Interaktionen zwischen ausgewählten Objekten in einem bestimmten Kontext.

  - **Sequenzdiagramm:** Beschreibt den Nachrichtenaustausch zwischen einer Menge ausgewählter Objekte in einer zeitlich begrenzten Situation.
  - **Zustandsdiagramm:** Beschreibt den Lebenszyklus eines Objektes durch eine Folge von Zuständen und Stimuli, die zu Zustandsübergängen führen.

- **Implementierungsdiagramme**
  - **Komponentendiagramm:** Stellt die Komponenten eines Systems dar und wie sie untereinander in Beziehung stehen.
  - **Verteilungsdiagramm:** Stellt dar, wie Objekte und Komponenten auf unterschiedliche Knoten (Prozesse, Rechner) verteilt sind und welche Kommunikationsbeziehungen zwischen ihnen bestehen.

Für eine ausführliche Diskussion der einzelnen Diagramme sei an dieser Stelle auf die einschlägige Literatur verwiesen ([Oes98, Gro00]). In den nachfolgenden Abschnitten erfolgt eine kurze Illustration, der für diese Arbeit wichtigen Diagramme.

2.7.1 Klassendiagramm

Das Klassendiagramm liefert eine statische Sicht auf die in einem System verwendeten Klassen und ihre Beziehungen untereinander. Einige wesentliche Merkmale eines Klassendiagramms sind in Abbildung 2.25 dargestellt. Im einzelnen sind die folgenden Beschreibungselemente sichtbar:

- **Klassen.** Klassen werden durch Rechtecke dargestellt, die in drei Bereiche unterteilt sind. Auf den Klassennamen folgen die in der Klasse enthaltenen Attribute und Methoden. Die Sichtbarkeit von Attributen und Methoden wird durch Symbole ausgedrückt. Das Kennzeichen '+' bedeutet `public`, '#' `protected` und '-' `private`. Abstrakte Klassen werden durch kursive Schreibung des Namens hervorgehoben.

- **Schnittstellen.** Schnittstellen können analog zu Klassen dargestellt werden. Um sie von Klassen unterscheiden zu können werden sie zusätzlich durch `interface` gekennzeichnet. Als Alternative zu dieser Darstellungsform kann die bei `Klasse 4` dargestellte Notation verwendet werden.

- **Beziehungen.** Klassen können durch Linien und Pfeile untereinander verbunden werden, um auszudrücken, das Beziehungen zwischen ihnen existieren. Folgende Beziehungen sind in Abbildung 2.25 dargestellt:
  - **Assoziation.** Beziehungen zwischen Klassen können durch eine einfache Linie ausgedrückt werden. Assoziationen können zusätzlich mit Kardinalitäten versehen werden, um die Anzahl der beteiligten Objekte anzugeben (hier: 1 Objekt der `Klasse 1` enthält n Objekte der `Klasse 2`).

  - **Gerichtete Assoziation.** Durch eine offene Pfeilspitze können Assoziationen gerichtet werden, um mögliche Navigationsrichtungen festzulegen (hier: Von Objekten der `Klasse 5` kann auf Objekte der `Klasse 1` navigiert werden, aber nicht umgekehrt).
  - **Aggregation.** Die Aggregation beschreibt eine Ganzes-Teile-Hierarchie und wird durch eine Raute dargestellt (hier: Objekte der `Klasse 1` bestehen aus einer nicht näher spezifizierten Menge von Objekten der `Klasse 3`).
  - **Komposition.** Die durch eine ausgefüllte Raute dargestellte Komposition ist eine stärkere Bedingung als die Aggregation und besagt, daß die Teile nur existieren, wenn das Ganze existiert (hier: Objekte der `Klasse 4` existieren nur, wenn zugehörige Objekte der `Klasse 1` existieren).
  - **Vererbung.** Die Vererbung von Eigenschaften einer Klasse an eine andere wird durch eine geschlossene Pfeilspitze dargestellt (hier: `Klasse 1` erbt die Eigenschaften von `Basisklasse`.)
  - **Schnittstellenimplementierung.** Die Implementierung einer Schnittstelle wird durch eine gestrichelte Linie mit geschlossener Pfeilspitze dargestellt.

- **Notizen.** Mit dem Notizsymbol können Kommentare ohne semantische Wirkung in das Diagramm aufgenommen werden. Durch die gestrichelte Linie kann einem oder mehreren Modellelementen ein Kommentar zugeordnet werden. Das Notizsymbol kann in allen UML-Diagrammen verwendet werden.

2.7.2 Sequenzdiagramm

Das Sequenzdiagramm stellt den zeitlichen Ablauf von Nachrichten dar, die eine Menge von Objekten in einer zeitlich begrenzten Situation untereinander austauschen. Die Basiselemente dieses Diagrammtyps sind in Abbildung 2.26 veranschaulicht.
Innerhalb des Diagramms werden die Objekte und ihre Interaktionen beschrieben. Der zeitliche Ablauf wird durch die Lebenslinie repräsentiert. Der Steuerungsfokus befindet sich auf der Lebenslinie und gibt an, welches Objekt gerade aktiv ist. Nachrichten werden als Pfeile zwischen den Lebenslinien von Objekten dargestellt. Mehrfach ausgelöste Nachrichten können dabei durch eine Iteration gekennzeichnet werden. Antworten werden als gestrichelte Pfeile repräsentiert. Durch die Selbstdelegation können auch Nachrichten dargestellt werden, die das betreffende Objekt an sich selbst schickt. Die Objektkonstruktion wird durch das Herabsetzen des erzeugten Objekts auf die Höhe des Nachrichtenpfeils dargestellt und das Entfernen eines Objekts wird durch das Kreuz am Ende des Steuerungsfokus dargestellt.

2.7.3 Zustandsdiagramm

Das Zustandsdiagramm beschreibt einen *Endlichen Automaten*, der sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet. Beschrieben wird dabei eine Folge von Zuständen, die ein Objekt während seines Lebenszyklus einnehmen kann und aufgrund welcher Stimuli

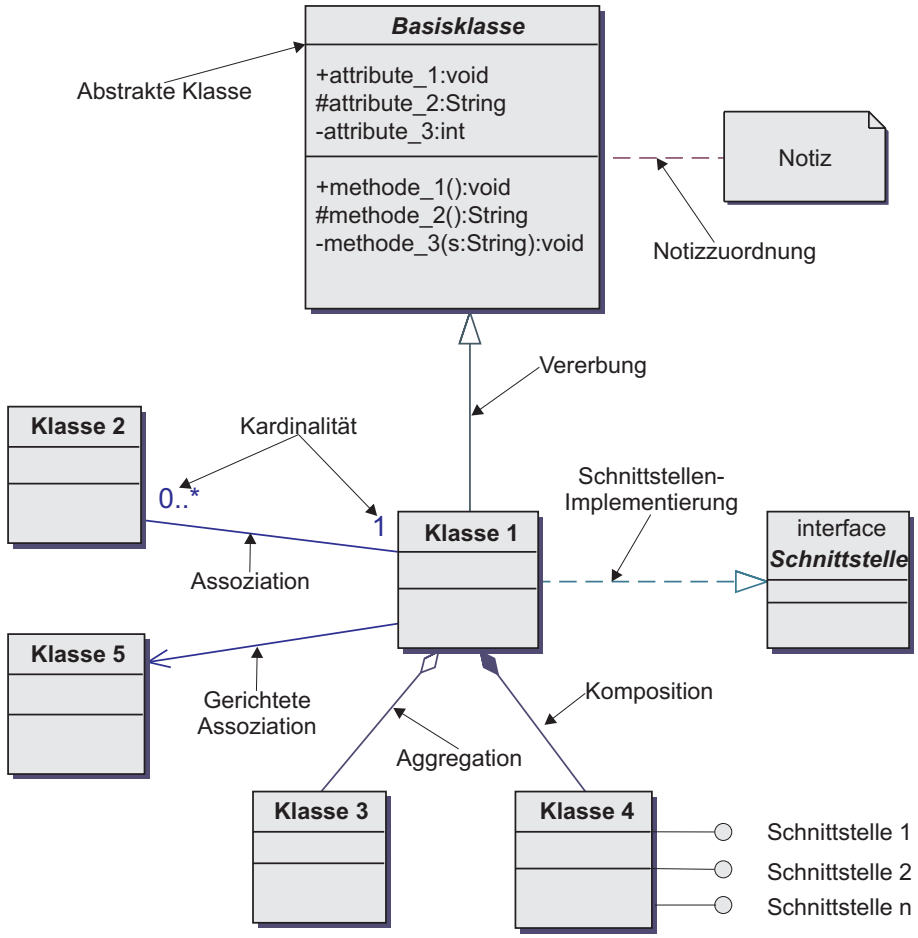


Abbildung 2.25: Klassendiagramm

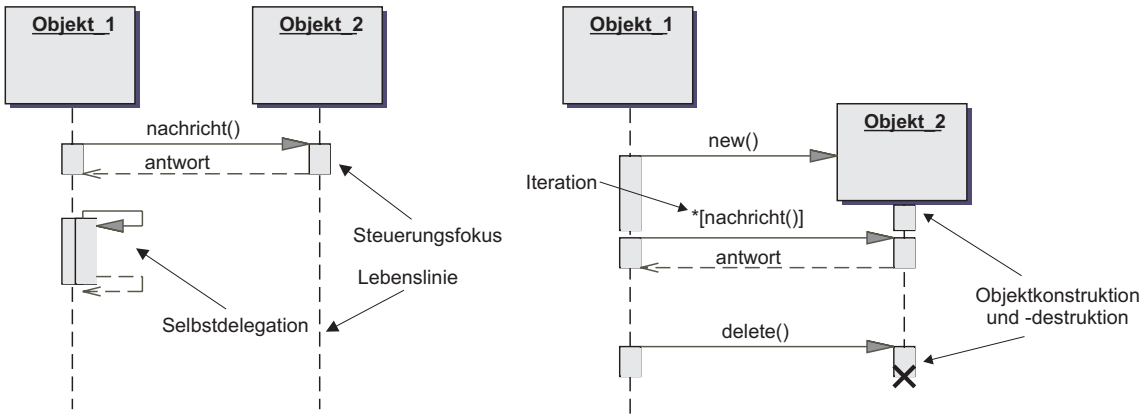


Abbildung 2.26: Sequenzdiagramm

Zustandsänderungen stattfinden. In Abbildung 2.27 ist ein einfaches Zustandsdiagramm dargestellt. Vom Startzustand aus wird der `Zustand 1` durch das `Ereignis 1` erreicht. Ein Übergangswechsel zu `Zustand 2` findet durch das `Ereignis 2` statt. Bei Auftreten von `Ereignis 3` bleibt der `Zustand 2` erhalten. Bei Eintritt von `Ereignis 4` ist der Endzustand erreicht.

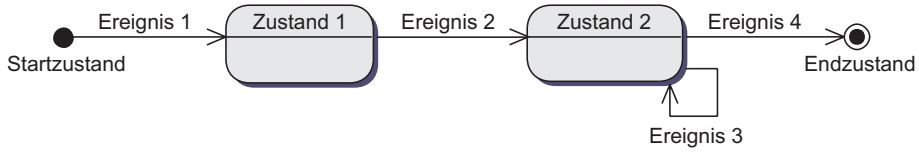


Abbildung 2.27: Zustandsdiagramm

2.7.4 Verteilungsdiagramm

Mit Verteilungsdiagrammen kann dargestellt werden, wie Komponenten und Objekte über Knoten (z.B. Computer, Prozeß) verteilt sind. Abbildung 2.28 stellt wesentliche Gestaltungselemente dieses Diagrammtyps dar.

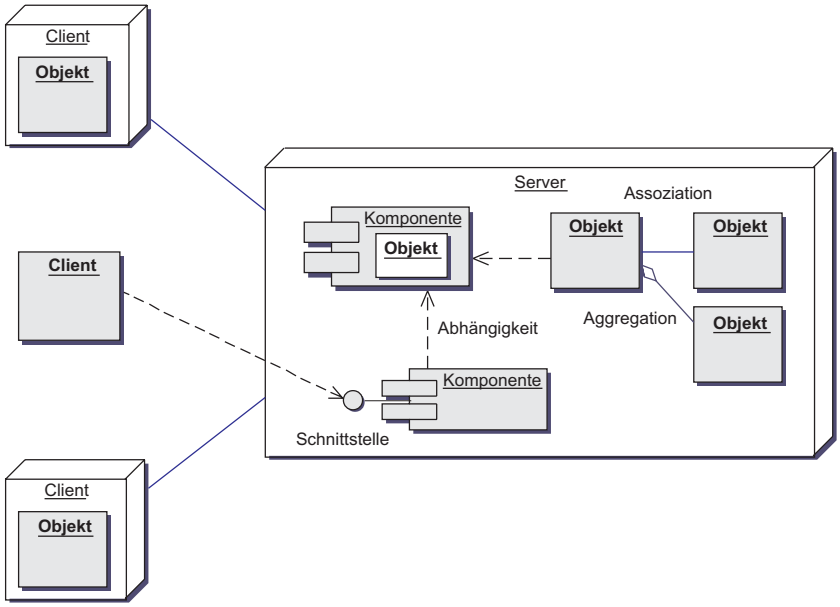


Abbildung 2.28: Verteilungsdiagramm

Knoten werden durch Quader dargestellt, Komponenten durch Rechtecke mit zwei kleinen Rechtecken am linken Rand und Objekte als Rechtecke. Komponenten können zusätzlich mit einer oder mehreren Schnittstellen versehen werden. Zwischen den einzelnen Elementen können Abhängigkeiten allgemein durch eine gestrichelte Linie mit offener Pfeilspitze dargestellt werden. Zwischen Objekten können auch Assoziationen und Aggregationen dargestellt werden.

2.7.5 Stereotypen

Die in UML vorhandenen Elemente können durch Stereotypen mit projekt-, unternehmens- oder methodenspezifischen Erweiterungen versehen werden [Oes98]. Zur Darstellung von J2EE-Komponenten werden die Diagramme im Rahmen dieser Arbeit mit UML-Stereotypen gemäß Tabelle 2.7 erweitert (in Anlehnung an [Dee01]). Dabei wird eine Vereinfachung der Darstellung angestrebt, indem mehrere Klassen durch eine Klasse mit Stereotyp-Erweiterung ersetzt werden. Dies ist z.B. bei EJB-Komponenten sinnvoll, die aus mehreren Klassen, Schnittstellen und einem Deskriptor bestehen.

Stereotyp	Bedeutung
EJB	Repräsentiert eine EJB-Komponente, die mit einem Geschäftsobjekt assoziiert ist.
SessionEJB	Repräsentiert eine Session-Bean als Ganzes. Dabei wird das Remote- und Home-Interface sowie die Bean-Implementierungsklasse nicht näher spezifiziert.
EntityEJB	Repräsentiert eine Entity-Bean als Ganzes. Dabei wird das Remote- und Home-Interface sowie die Bean-Implementierung und die Primärschlüsselklasse nicht näher spezifiziert.
MessageDrivenEJB	Repräsentiert eine Message-Driven-Bean.
Servlet	Repräsentiert ein Java-Servlet.

Tabelle 2.7: Verwendete UML-Stereotypen

Ein Beispiel illustriert in Abbildung 2.29, wie die Stereotypen angewendet und in Diagrammen dargestellt werden. Aus dem dargestellten Klassendiagramm geht hervor, daß die Klasse `Vorgang` ein Servlet ist und die Klassen `Vorgang1` und `Vorgang2` Session-Beans sind. Das Servlet versorgt den HTML-Client, der hier als normale Klasse dargestellt ist, mit Daten und nutzt hierfür die beiden Session-Beans.

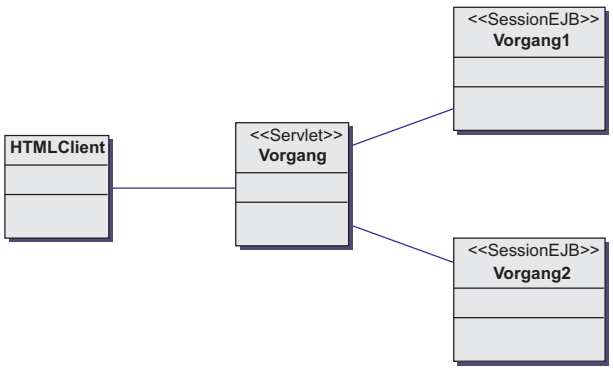


Abbildung 2.29: Verwenden von Stereotypen

Kapitel 3

Stand der Technik

Es liegt in der Natur von Client/Server-Anwendungen, daß Daten zwischen Client und Server übertragen werden müssen. Die Form, in der diese Daten vorliegen unterliegt weitestgehend der freien Gestaltung des Entwicklers, der das Software-System implementiert. Im EJB-Umfeld fußt die Kommunikation zwischen Server-Komponenten und ihren Clients auf dem Java-Serialisierungsmechanismus. D.h. jede serialisierbare Datenstruktur kann zwischen Client und Server übertragen werden (vgl. dazu Kapitel 2, Abschnitt 2.3). Bestehende Datenübertragungskonzepte lassen sich gemäß Abbildung 3.1 in statische und dynamische Ansätze einteilen.

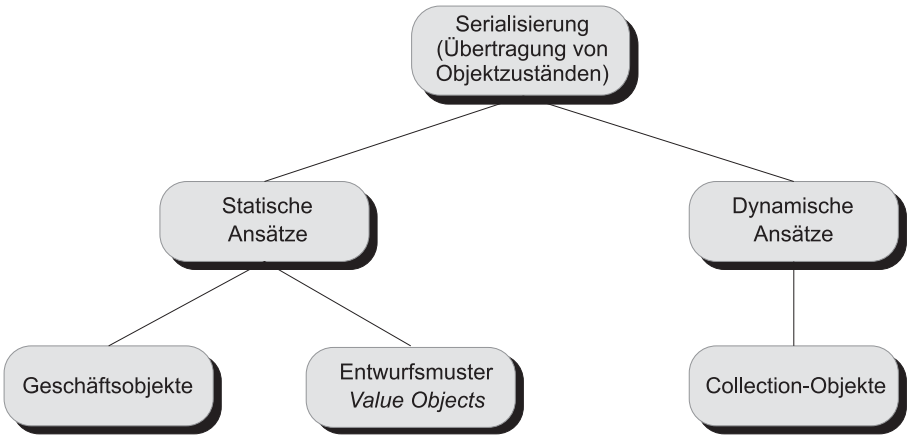


Abbildung 3.1: Bestehende Datenübertragungskonzepte

Statische Ansätze beruhen auf der Verwendung von serialisierbaren Transportobjekten, die Daten in Form ihrer Attribute deklarieren. Somit steht bereits zur Übersetzungszeit statisch fest, welche Daten in Objekten dieser Klassen transportiert werden können. Dies ist mit einer strengen Prüfung des Java-Compilers verbunden, ob bei lesenden und schreibenden Zugriffen auf das Objekt und den damit verbundenen Zuweisungen eine korrekte Typisierung der Datentypen vorliegt. Zusätzlich führen Zugriffe auf nicht vorhandene Attribute zu einem Übersetzungsfehler.
Dynamische Ansätze verwenden serialisierbare Objekte, die Daten unterschiedlichen Typs zur Laufzeit aufnehmen können. Zu dieser Klasse gehören alle `Collection`-Objekte, die Bestandteil der Java-Bibliothek sind und zur Aufnahme von beliebigen Java-Objekten genutzt

werden können. Diese Objekte sind zur Laufzeit in der Lage beliebige Java-Objekte vom Basistyp `Object` aufzunehmen und können damit ebenfalls als Transportobjekt genutzt werden, solange die übergebenen Objekte selbst serialisierbar sind. Bei der Entnahme von Objekten aus diesen Datenstrukturen muß i.d.R. wieder eine Typumwandlung in den ursprünglichen Typ erfolgen. Aufgrund der beschriebenen Eigenschaften kann diese Datenstruktur vom Compiler nicht zur Übersetzungszeit überprüft werden. Fehler, die bei der Entnahme von Objekten durch eine falsche Typumwandlung entstehen, führen zur Laufzeit zur Erzeugung einer Ausnahme.
In diesem Kapitel werden die gängigen statischen und dynamischen Datenübertragungsverfahren beschrieben. Dabei muß berücksichtigt werden, daß in einem EJB-System viele unterschiedliche Komponentenarten, Objekttypen und Datenstrukturen existieren, die aufgrund von objektorientierten Analyse- und Designprozessen ermittelt und erstellt werden. Dies führt zu vielfältigen Anforderungen an Datenübertragungsmechanismen. Im folgenden Abschnitt erfolgt deshalb ein kurzer Umriß des Vorgehens bei der Erstellung einer EJB-Anwendung.

3.1 Anwendungsarchitekturen

Bei der Erstellung eines Systems werden während der Analyse typischerweise Anwendungsfälle und die benötigten Objekte in der jeweiligen Anwendungsdomäne identifiziert. Anhand dieser Ergebnisse muß nun in einem Designschritt für jeden Anwendungsfall und jedes identifizierte Geschäftsobjekt entschieden werden, ob sein Zustand dauerhaft persistent sein muß und ob eine Abbildung in eine EJB oder ein normales Java-Objekt erfolgen soll. Wird ein Geschäftsobjekt oder ein Anwendungsfall als EJB repräsentiert muß jeweils entschieden werden, welche Bean-Art und -Unterart verwendet wird. Bei der Repräsentation als Java-Objekt muß zusätzlich entschieden werden, wo sich das Objekt befinden soll, d.h. z.B. innerhalb welcher EJB es verwendet werden soll (Partitionierung). In Abbildung 3.2 ist die grundsätzliche Herausforderung dargestellt, die bei der Entwicklung einer EJB-Architektur zu bewältigen ist.
Der Entscheidung darüber, wie die Repräsentation eines Geschäftsobjekts in einem EJB-System erfolgen soll, geht die folgenden Klassifikation voraus:

- **Verteilte Objekte:** Es handelt sich dabei um Geschäftsobjekte, die für Clients über Rechnergrenzen hinweg verfügbar sein müssen und deshalb als EJBs implementiert werden.

- **Persistente Objekte:** Es handelt sich dabei um Objekte, deren Zustand dauerhaft in einer Datenbank gespeichert werden muß. Objekte mit diesen Eigenschaften können mit den Mechanismen des EJB-Standards in Form von Entity-Beans realisiert werden. Alternativ kann auf Persistenz-Frameworks[1] von Drittherstellern zurückgegriffen werden, die diesen Prozeß vereinfachen[2]. Persistenz-Frameworks lösen auch die Abbildungsproblematik zwischen objektorientierten Strukturen und relationalen Datenbanken (vgl. dazu

[1] Ein Framework besteht aus einer Menge von Objekten, die eine generische Lösung für eine Reihe verwandter Probleme implementieren [Bir95]. Ein Persistenz-Framework befaßt sich schwerpunktmäßig mit Fragen der Persistenz von Daten. In dieser Arbeit sollen darunter insbesondere Frameworks verstanden werden, die eine dauerhafte Speicherung der Daten von Objekten und Komponenten ermöglichen.
[2] Beispiele für solche Produkte sind *Avantis Unisuite* [Ava], *Cocobase* [Tho] und *TOPLink* [Web].

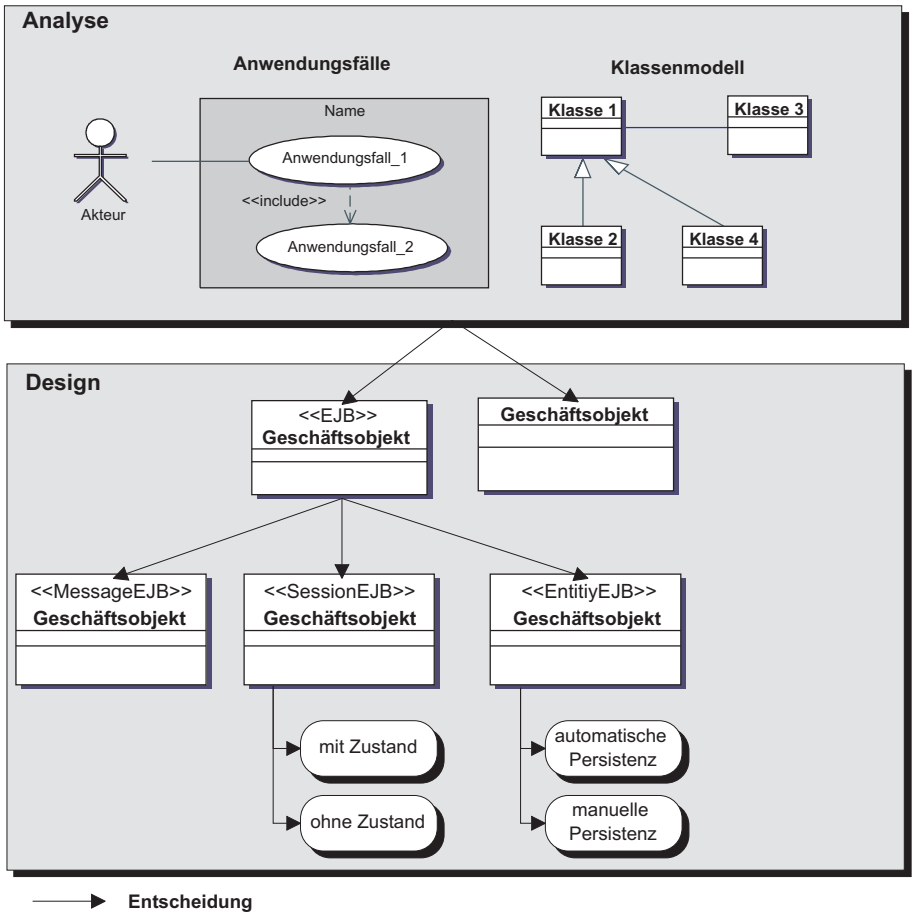


Abbildung 3.2: Entwicklungsprozess

z.B. [Fus97, Amb00, Küh01]). Alternativ kann die Persistenz durch eine objektorientierte Datenbank gewährleistet werden[3]. Ein herstellerunabhängiger Standard für einen Persistenzmechanismus ist mit *Java Data Objects (JDO)* [Gro01b] verfügbar. Ein sehr guter Überblick über JDO befindet sich in [Frö01].

- **Flache Datenstruktur:** Aus Leistungsgründen kommt häufig auch der Verzicht auf objektorientierte Strukturen in Betracht. Dabei wird eine flache Datenstruktur beibehalten, wie sie z.B. relationale Datenbanken beim Zugriff mittels JDBC liefern. Dabei wird der Aufwand eingespart, der existiert, um solche flachen Datenstrukturen in objektorientierte Strukturen zu überführen.

  Bei der Anbindung von Alt- und Fremdsystemen muß häufig auf die Verwendung von objektorientierten Strukturen verzichtet werden, da diese z.B. auf Text basierende Nachrichtenformate verwenden.

Nachdem das Ergebnis der Klassifizierung in verteilte und persistente Objekte vorliegt, müssen weitere Designentscheidungen getroffen werden, die darin bestehen die zu verwendenden EJB-

[3] Ein Beispiel für ein solches Produkt ist *EnJin* [Ver].

Arten und ihr Zusammenspiel festzulegen. In der Literatur finden sich zahlreiche Entwurfs- und Implementierungshinweise. Empfehlenswert sind [Kas00, Mye00, Lar00, Suc01, Rom02]. In Abbildung 3.3 ist das am häufigsten in der EJB-Literatur reflektierte Architekturkonzept dargestellt, das auf Session- und Entity-Beans beruht. Die Architektur setzt sich dabei aus Session-Beans zusammen, die auf Entity-Beans zugreifen. Clients führen durch die Session-Beans Geschäftsprozesse aus. Die Session-Beans manipulieren dabei persistente Objekte, die als Entity-Beans modelliert sind. Ein solcher Aufbau wird in der Literatur häufig als Session-Fassade bezeichnet [Dee01, Rom02].

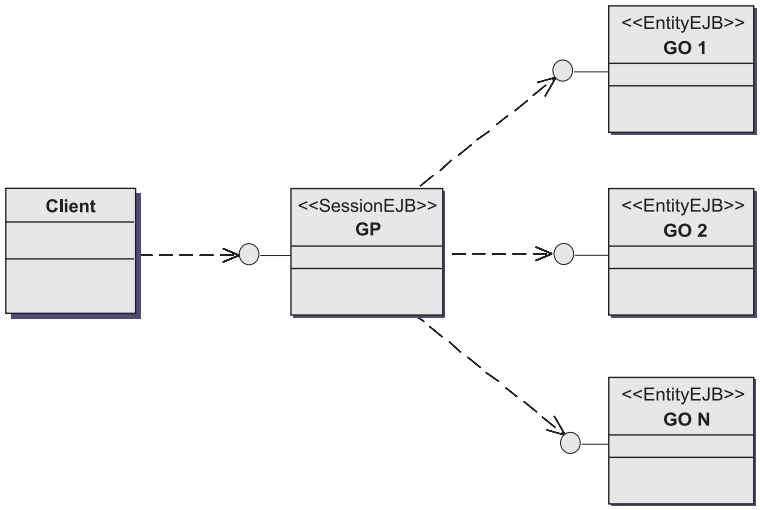


Abbildung 3.3: Session-Fassade

Die Verwendung von Entity-Beans unterliegt in der industriellen Praxis allerdings einer heftigen Diskussion [Ale01]. Diese bezieht sich u.a. auf deren Schwergewichtigkeit als potentiell verteilte Komponenten, aber auch auf fehlende Konzepte wie z.B. Vererbung [Gro01a]. Dies führt zu höheren Entwicklungskosten, da die Implementierung einer Komponente mehrere Klassen und Deskriptoren für die Laufzeitumgebung benötigt und keine ausreichende Wiederverwendung durch Vererbung stattfinden kann. Weiterhin ist häufig das Laufzeitverhalten aufgrund der Schwergewichtigkeit von Entity-Beans negativ tangiert. Als Folge der Problematik mit Entity-Beans existieren verschiedene Entwurfsansätze, um deren Nachteile zu reduzieren. Das Entwurfsmuster *Aggregate Entity* dient dazu die Anzahl der Entity-Beans zu reduzieren [Lar00]. Eine Entity-Bean besitzt dabei mehrere abhängige Geschäftsobjekte, die als Java-Objekte implementiert sind. Ein Geschäftsobjekt ist dabei abhängig, wenn sein Lebenszyklus vollständig durch die Entity-Bean kontrolliert wird. In [Lar00] werden zwei Lösungsstrukturen vorgeschlagen. Die erste beruht auf dem Entwurfsmuster *Bridge* [Gam95] und ist in Abbildung 3.4 dargestellt.
Die Entity-Bean besitzt dabei als Bridge-Abstraktion eine Referenz auf ein unabhängiges Objekt (Bridge-Implementierung), das als Wurzel einer Hierarchie abhängiger Objekte dient. Die zweite, in Abbildung 3.5 dargestellte Möglichkeit, besteht darin, das unabhängige Objekt direkt als Entity-Bean zu implementieren und dabei auf das Bridge-Muster zu verzichten.
Die Verwendung des Bridge-Musters bietet den Vorteil, daß die Bridge-Implementierung samt

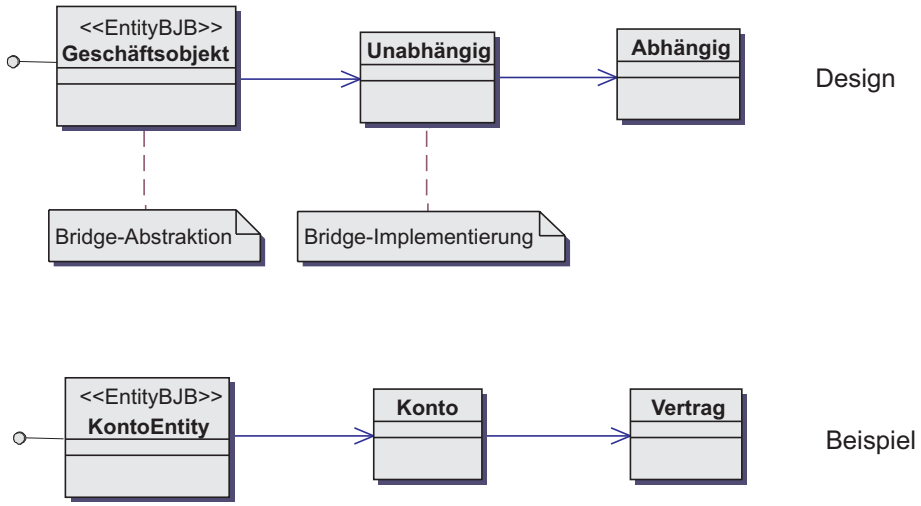


Abbildung 3.4: Entwurfsmuster *Aggregate Entity* mit *Bridge*-Muster

dem dahinterliegenden Objektgraphen in anderen Systemen, die nicht auf EJBs basieren, wiederverwendet werden kann. Falls eine Umstrukturierung dahingehend erforderlich wird, daß das unabhängige Objekt abhängig wird, kann es einfach in die Hierarchie des neuen unabhängigen Objekts integriert werden. Nachteilig bei der Verwendung des Bridge-Musters ist, daß ein zusätzliches Objekt benötigt wird und damit ggf. entsprechende Zugriffsmechanismen auf Objekte, die in der Hierarchie tiefer liegen, implementiert werden müssen.
Um die Erzeugung vieler Entity-Beans bei großen Anfragen zu verhindern, besteht eine Architekturvariante darin, eine Methode in der Session-Bean zur Verfügung zu stellen, die eine direkte Anfrage über JDBC startet und das Ergebnis in Listenform an die Clients übergibt. Erst bei Auswahl einer oder mehrerer Datensätze zur Bearbeitung durch den Client, werden die entsprechenden Entity-Beans erzeugt und bieten die gewohnten Eigenschaften hinsichtlich Transaktionalität und Fehlertoleranz. Die beiden unterschiedlichen Ansätze sind in Abbildung 3.6 und 3.7 in Form eines Verteilungsdiagramms und der zugehörigen skizzierten Aufrufsequenz dargestellt. Beim ersten Architekturansatz (Abbildung 3.6) bewirkt eine Suchanfrage des Clients

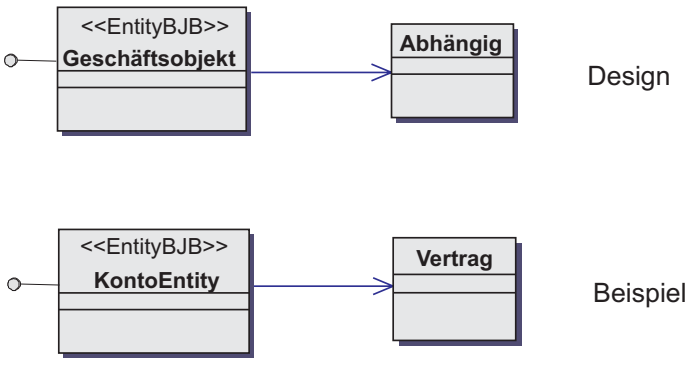


Abbildung 3.5: Entwurfsmuster *Aggregate Entity* ohne Bridge-Muster

(1) den Aufruf einer entsprechend zugeordneten `finder`-Methode auf dem `Home`-Objekt der zugehörigen Entity-Bean (1.1). Falls Daten vorhanden sind, werden die entsprechenden Entity-Beans bereitgestellt und der Zugriff über `EJBObject(s)` und zugehörige `Stubs` ermöglicht (Rückgabe von 1.1). Beim Zugriff auf die Entity-Beans durch die Session-Bean muß eine Entnahme der gewünschten Daten für die Trefferliste erfolgen (Vorgang wird nicht im Sequenzdiagramm dargestellt). Die so entstandene Liste muß nun in einer dafür vorgesehenen Form zum Client zurückgeschickt werden (Rückgabe von 1). Dieses Vorgehen stellt hohe Anforderungen an die Ressourcen, da sehr viele schwergewichtige Komponenten erzeugt werden, um darauf zuzugreifen und eine Liste darzustellen. Dies ist insbesondere dann unnötig, wenn nur wenige Elemente der Liste tatsächlich bearbeitet werden sollen.

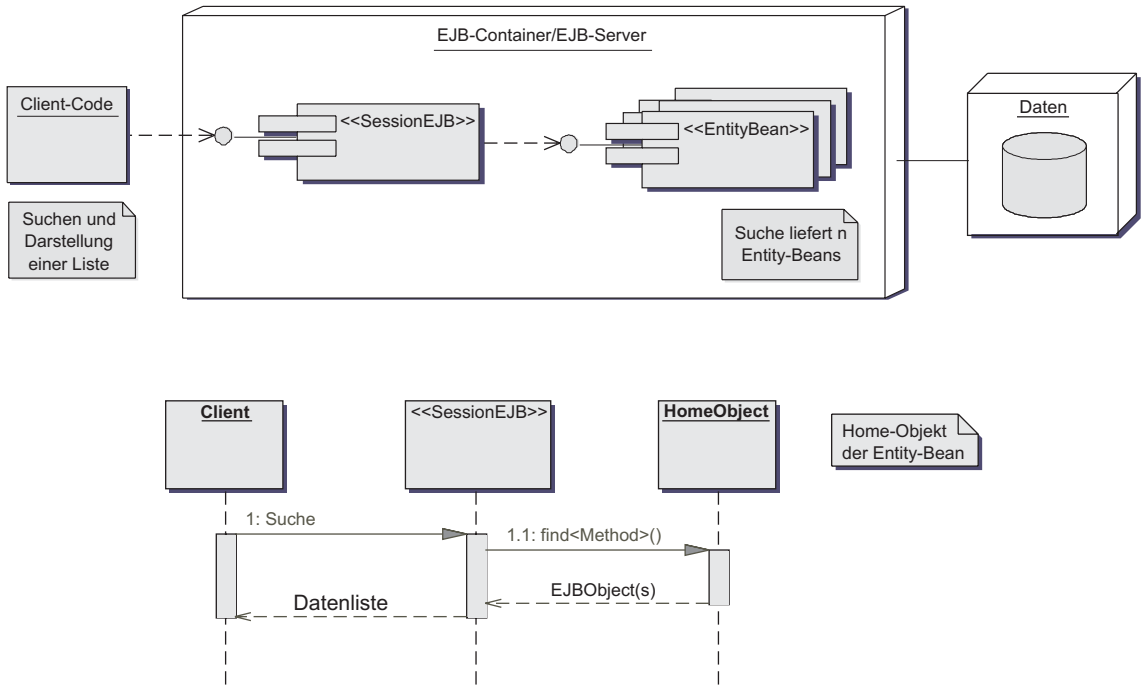


Abbildung 3.6: Architektur ohne ResultSet-Objekt

Im Gegensatz dazu ruft die Suchanfrage eines Clients (1) die Erzeugung eines zugehörigen JDBC-Statements zum direkten Absetzen einer SQL-Anfrage auf der Datenbank hervor (1.1, `createStatement`-Methode). Nachdem die Anfrage mittels der `executeQuery`- Methode des `Statement`-Objekts abgesetzt wurde (1.2), wird ein `ResultSet-Objekt` mit den Ergebnissen der Datenbankanfrage erzeugt (1.2.1) und an die Session-Bean zurückgegeben (Rückgabe von 1.2). Die Treffer müssen nun aus diesem Objekt ausgelesen werden und in das dafür vorgesehene Listenformat überführt werden. Bei jeder Auswahl eines Listenelements durch den Benutzer auf dem Client erfolgt ein Aufruf an die Session-Bean (2, 3), der die Suche und Erzeugung der jeweiligen Entity-Bean zur Folge hat (2.1, 3.1) sowie die für die Bearbeitung notwendigen Daten der Entity-Bean entnimmt und in der dafür vorgesehenen Form an den Client zurückgibt. Dieser Ansatz reduziert die Erzeugung von schwergewichtigen Komponenten auf diejenigen, deren Daten tatsächlich weiterverarbeitet werden.

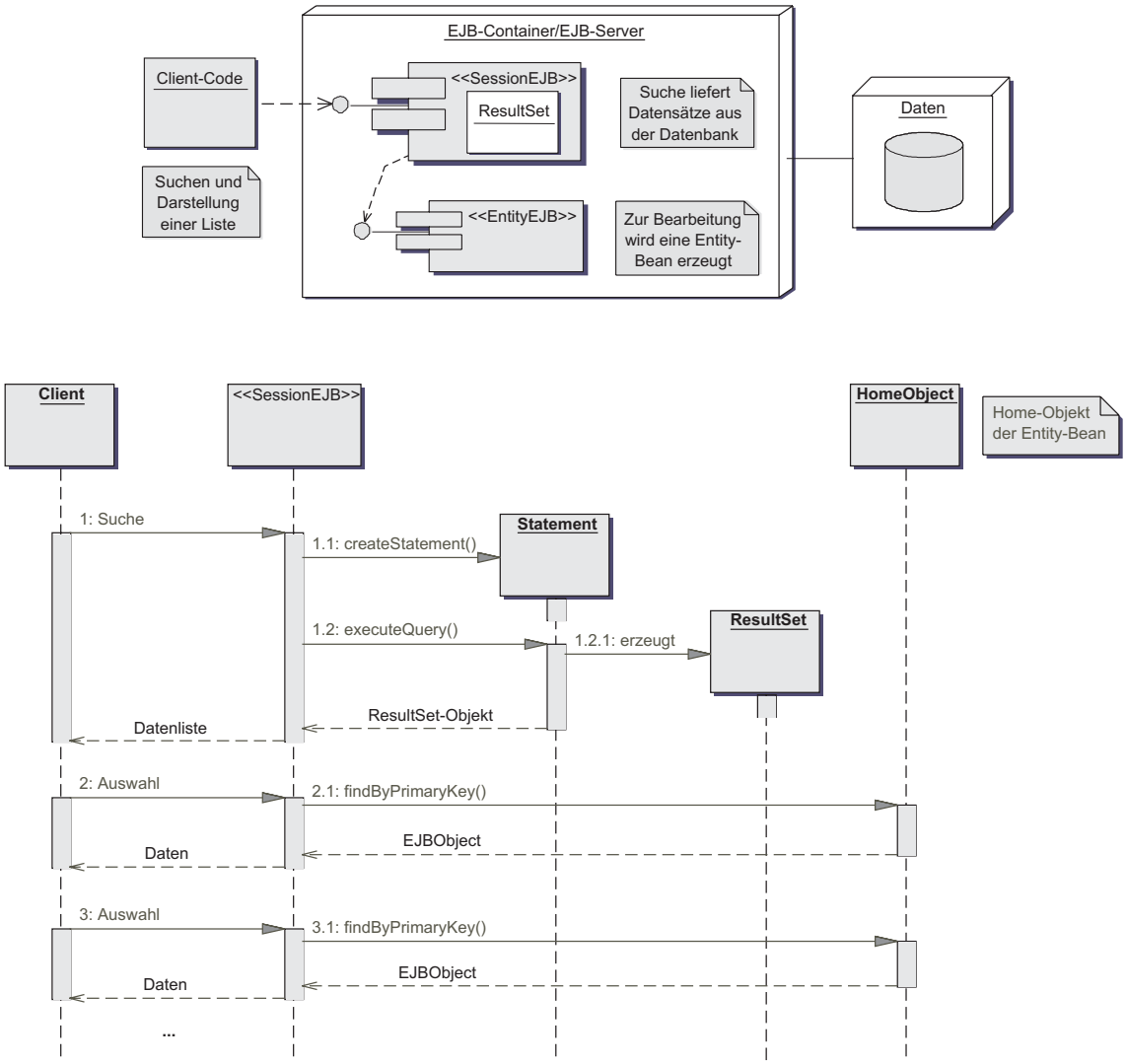


Abbildung 3.7: Architektur mit ResultSet-Objekt

Schließlich kann auf die Verwendung von Entity-Beans völlig verzichtet werden, was zu dem in Abbildung 3.8 dargestellten Architekturansatz führt. Die Clients führen dabei Geschäftsprozesse aus, die durch Session-Beans modelliert werden und ihrerseits auf persistente Objekte zurückgreifen, die als normale Java-Objekte modelliert sind. Zur Realisierung dieses Ansatzes werden Persistenz-Frameworks verwendet. Dabei kann bei den persistenten Objekten auch auf Vererbungsmechanismen zurückgegriffen werden. Dieser Ansatz kann analog wie bei Entity-Beans wiederum mit einem direkten Zugriff auf die Datenbank gekoppelt werden. Dadurch kann bei der Verarbeitung großer Datenmengen wiederum der Aufbau objektorientierter Strukturen umgangen werden, der einen Mehraufwand zur Laufzeit darstellt und damit das Leistungsverhalten negativ beeinflußt.

Insgesamt muß festgehalten werden, daß sich bis heute keine einheitliche Meinung über die Struktur von EJB-Architekturen herausgebildet hat. Die tatsächliche Struktur wird maßgeb-

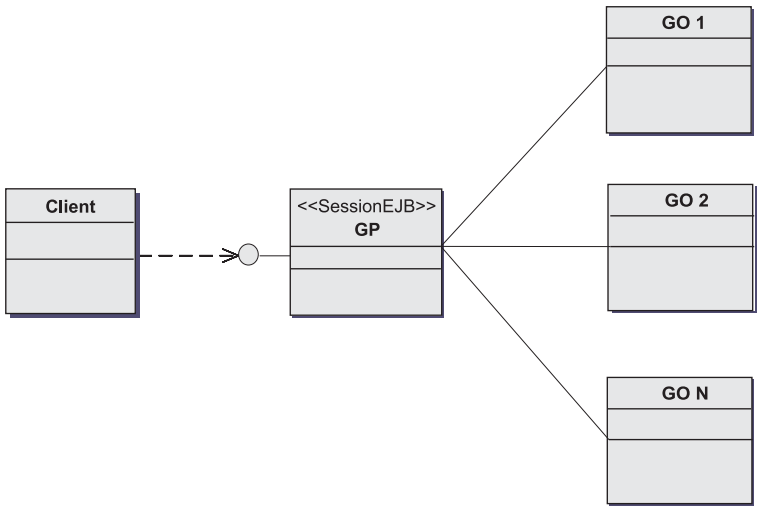


Abbildung 3.8: Session-Fassade mit Java-Objekten

lich von den Anforderungen an ein neu zu entwickelndes Anwendungssystem bestimmt. In jedem Fall muß jedoch festgelegt werden, wie die verschiedenen Komponenten untereinander und mit ihren Anwendungs-Clients kommunizieren. Dies untermauert die Notwendigkeit eines möglichst flexiblen und universellen Datenübertragungskonzepts, das unabhängig von der gewählten Architektur eingesetzt werden kann.

3.2 Datenübertragungskonzepte

3.2.1 Statische Konzepte

Statische Konzepte beruhen auf der Verwendung von Daten-Containern, die bereits zur Übersetzungszeit festlegen, welche Datentypen zum Transport übergeben werden können.

Serialisierung von Geschäftsobjekten

Die Funktionsweise des Java-Serialisierungsmechanismus erlaubt die Erstellung beliebiger anwendungsbezogener serialisierbarer Java-Objekte, die zwischen Client und Server übertragen werden können (vgl. Kapitel 2, Abschnitt 2.3). Somit ermöglicht dieser Mechanismus ein Datenübertragungskonzept, das darauf beruht, die verwendeten Geschäftsobjekte als serialisierbar zu definieren und gleichermaßen auf dem Client und Server zu verwenden. Abbildung 3.2.1 veranschaulicht dies.
Dieses Konzept wird in der Literatur auf unterschiedliche Weise zur Lösung der Datenübertragung eingesetzt. In [MH99, Lar00, Kas00] wird die Datenübertragung eng an Design- und Architekturfragen gekoppelt. Dabei sollen sog. abhängige Java-Objekte, die serialisierbar sind, verwendet werden. Ein Objekt ist dabei abhängig, wenn es von der Existenz einer Entity-Bean abhängt und selbst kaum Logik enthält, sondern hauptsächlich nur Datenfelder besitzt. Zusätzlich wird in [MH99, Kas00] betont, daß abhängige Objekte nur lesend auf dem Client verwendet

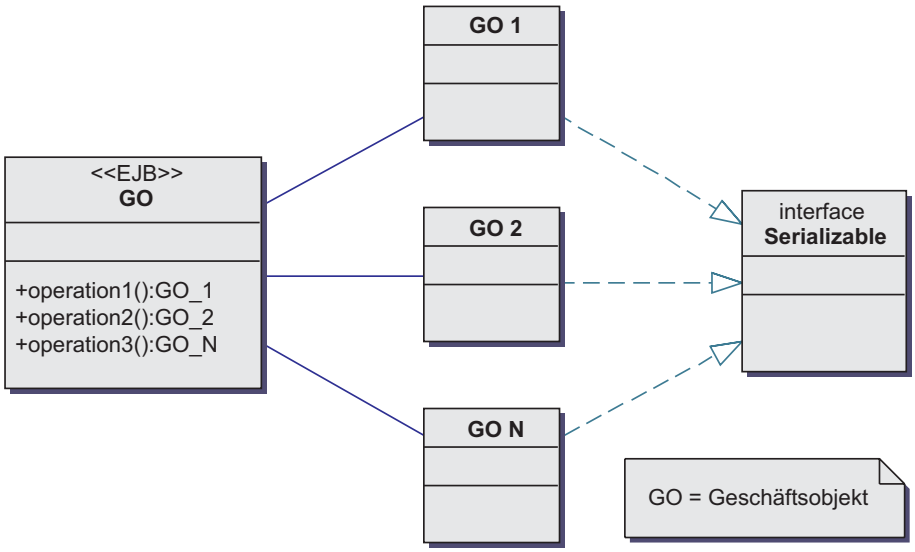


Abbildung 3.9: Serialisierungskonzept

werden sollen, da sie eine Kopie eines serverseitigen Objekts darstellen und Änderungen nur direkt auf dem Server vorgenommen werden sollten, um Synchronisationsprobleme zu verhindern. In [Tea00] wird darauf verwiesen, daß nach Möglichkeit die Geschäftsobjekte selbst serialisiert werden sollten, da dies mit dem geringsten Aufwand verbunden ist. In [Res00] wird die Serialisierung direkt auf Instanzen von Entity-Beans angewendet, um sie zum Client zu übertragen. Die serialisierten Instanzen werden als „Astrale Klone“ bezeichnet, da sie außerhalb der Kontrolle des EJB-Containers verwendet werden. Dabei müssen EJB-Container-spezifische Attribute, wie z.B. der `EntityContext` ersetzt oder mittels `transient` ausgeblendet werden. Zum Abgleich von im Klon geänderten Attributen, muß er zur Entity-Bean zurückgeschickt werden und Attribut für Attribut ausgelesen werden.

Value Objects

Das Konzept der (*Value Objects*) wird in den J2EE-Blueprints [Kas00] und dem J2EE-Entwurfsmusterkatalog [Dee01] beschrieben. Dabei werden für als EJBs modellierte Geschäftsobjekte serialisierbare Daten-Container-Klassen implementiert, die alle Geschäftsattribute der Beans zum Transport enthalten. Objekte dieser Klasse werden im System zum Transport der Daten zwischen Client und Server verwendet. Die Abbildung 3.2.1 und 3.2.1 [Dee01] zeigen die grundlegende Struktur und Verwendung der *Value Objects*. Eine Anfrage eines Clients (1, `GetData`) an ein Geschäftsobjekt, das als EJB modelliert ist, ruft die Erzeugung eines Daten-Containers (1.1, `ValueObject`) hervor, der die Attribute des Geschäftsobjekts enthält. Der Daten-Container wird als Kopie an den anfragenden Client zurückgeschickt (1.2). Die Kopie `ClientValueObject` kann anschließend mittels dafür vorgesehenen `GetValue`-Methoden ausgelesen werden (2, 3).
Konzepte, die weitgehend den *Value Objects* entsprechen sind:

- In [Mye00] wird die Verwendung eines Detailobjekts für Entity-Beans vorgeschlagen,

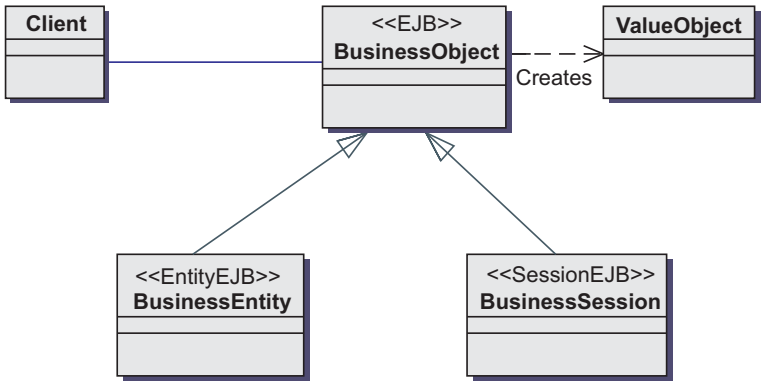


Abbildung 3.10: Struktur von Value Objects

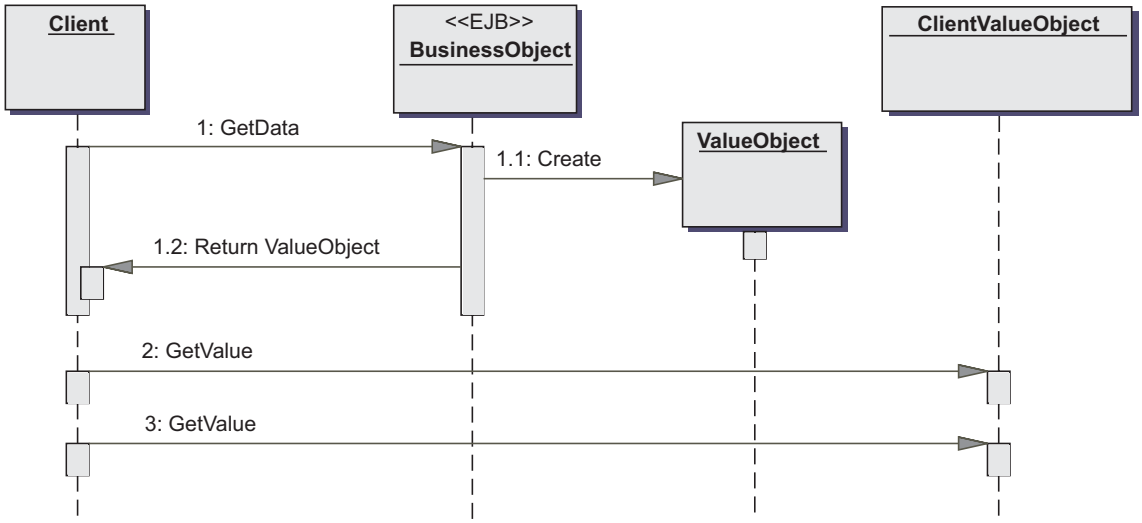


Abbildung 3.11: Verwendung von Value Objects

das als nur lesbare Java-Bean implementiert wird. Die Verhinderung des Schreibens auf das Detailobjekt soll verhindern, daß der Client Datenänderungen vornimmt, die nicht systemweit sichtbar sind.

- In [Dav99] wird ein serialisierbares Properties-Objekt empfohlen, das die Attribute einer Entity-Bean enthält. Erlaubt ist lesender und schreibender Zugriff. Das Objekt nimmt zusätzlich einen Zeitstempel auf, um Konflikte zu erkennen, wenn mehrere Clients auf die selbe Bean zugreifen und die lokal geänderten Attribute des Properties-Objekts zurückschicken.
- In [Tea00, Lar00] werden *Domain Object State Holder* beschrieben, die als Container-Objekte den Zustand von Geschäftsobjekten direkt oder davon abgeleitete Informationen transportieren.

In [Dee01] werden mit *Value Object Assembler* und *Value List Handler* weitere Entwurfsmuster

auf sehr hohem Abstraktionsniveau beschrieben, die sich mit *Value Objects* auseinandersetzen. Die Kernaussage von *Value Object Assembler* ist dabei, daß mehrere *Value Objects* zu einem neuen, komplexeren *Value Object* zusammengesetzt werden müssen, falls vom Client mehrere Daten aus unterschiedlichen Quellen angefordert werden. Der *Value List Handler* befaßt sich im Kern mit Anfragen, die eine große Treffermenge zur Folge haben. Die dabei anfallende hohe Anzahl von *Value Objects* soll laut Aussage des Musters als Liste zwischengepuffert werden und dem Client mittels eines Iterators (vgl. dazu [Gam95]) in kleineren Mengen zur Verfügung gestellt werden.

3.2.2 Dynamische Konzepte

Dynamische Konzepte beruhen auf der Verwendung von flexiblen Datenstrukturen, die in der Lage sind, beliebige Java-Objekte abzuspeichern. In der Java-Bibliothek sind eine Reihe solcher Klassen vorhanden, deren Objekte das beschriebene Verhalten ermöglichen. Es handelt sich dabei vor allem um Klassen, die das Interface `Collection` oder `Map` implementieren [Mica]. `Collection`-Objekte stellen eine Gruppe von Objekten dar, die als Elemente bezeichnet werden. Objekte vom Typ `Map` ordnen Schlüssel-Objekten, Wert-Objekte zu, wobei jeder Schlüssel nur einmal enthalten ist und genau ein Wert zugeordnet werden kann. In Abbildung 3.2.2 ist die Anwendung verallgemeinert dargestellt.

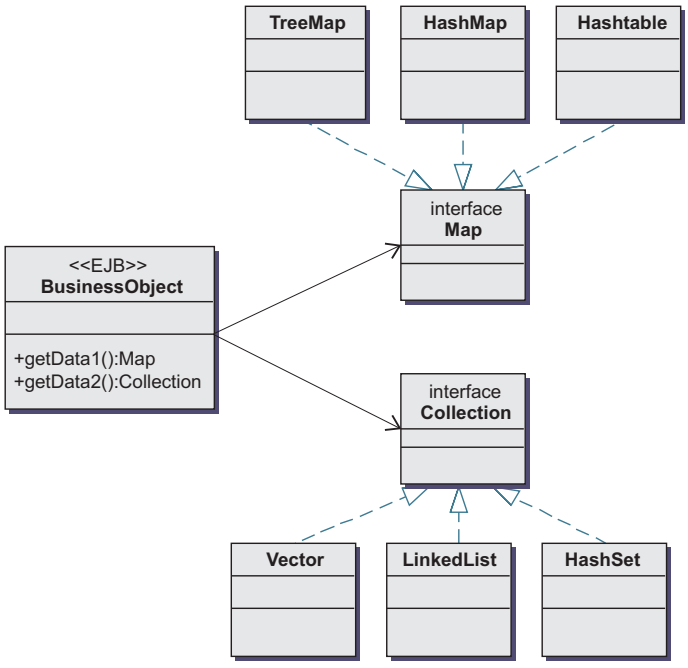


Abbildung 3.12: Dynamische Transportobjekte

Einige Beispiele für konkrete Objekte sind:

- **HashSet:** Realisiert eine Menge von Objekten, die keine doppelten Elemente enthält. Die Basisoperationen Hinzufügen (`add()`), Entfernen (`remove()`) und Existenzprüfung

(`contains()`) von Elementen sowie die Größenbestimmung (`size()`) erfolgen aufgrund des Hash-Algorithmus in konstanter Zeit.

- **HashMap:** Implementierung des `Map`-Interfaces, die auf einem Hash-Verfahren beruht, um die Basisoperationen Lesen (`get()`) und Schreiben (`put()`) von Werten in konstanter Zeit zu ermöglichen.

- **Vector:** Realisiert eine Folge von Objekten, die zur Laufzeit dynamisch wachsen und schrumpfen kann.

Die Datenstrukturen implementieren zusätzlich das Interface `Serializable` und können damit zur Kommunikation in EJB-Systemen verwendet werden [Bes00].
Neben den `Collection`- und `Map`-Objekten existiert eine weitere Klasse von Objekten, die bei der direkten Verwendung von SQL-Anfragen mittels JDBC, auftreten. In Form von `ResultSet`-Objekten wird das Ergebnis einer Datenbankanfrage zurückgegeben und kann mittels generischer `get`-Methoden ausgelesen werden. Diese Objekte sind nicht serialisierbar und können nicht direkt zum Client übertragen werden. Der Typ `CachedRowSet` ist eine Implementierung der `RowSet`-Schnittstelle und besitzt grundsätzlich die selben Eigenschaften, benötigt jedoch keine Verbindung zur Datenbank. Zusätzlich sind diese Objekte serialisierbar und können somit immer dann zum Einsatz kommen, wenn tabellarische Daten zum Client geschickt werden müssen [Cow01a]. Diese Anwendung von `CachedRowSet` ist in Abbildung 3.2.2 skizziert.

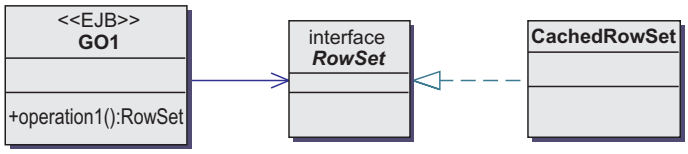


Abbildung 3.13: Dynamische Datenübertragung mit CachedRowSet

3.2.3 Verwandte Konzepte

Es gibt weitere Konzepte, die mit der Frage nach der Datenübertragung verwandt sind. Nachfolgend wird ein kurzer Überblick gegeben.

Smart-Stubs

Mit *Smart Stubs* oder *Smart Proxies* wird im CORBA-Umfeld versucht, die Datenübertragung zu optimieren, indem z.B. ein clientseitiger Cache etabliert wird [Vog98]. Das Grundprinzip besteht darin, den *Stub* durch ein weiteres Objekt zu kapseln und Aufrufe zu delegieren. Dabei bewirkt nicht jeder Aufruf einen Durchgriff auf den Server. In [Wil00] wird ein solcher Ansatz für RMI-Systeme beschrieben. In [Neu99] wird ein solcher Ansatz in einem EJB-System umgesetzt.

Intelligente Agenten

Eines der Ziele des Agenten-Paradigmas ist es, mobile Agenten, die sich reaktiv und proaktiv in ihrer Umgebung verhalten können von Server zu Server wandern zu lassen und dabei Daten zu filtern, die für den Auftraggeber interessant sind. Dabei steuert das Forschungsgebiet der Künstlichen Intelligenz (KI) einen signifikanten Anteil bei, um ein intelligentes Verhalten des Agenten zu ermöglichen. Technisch gesehen verhindert der Agentenansatz die Übertragung einer großen Datenmenge über das Netzwerk, die aufgrund fachlicher Merkmale nicht vollständig benötigt wird. In [Jen98] befindet sich ein guter Gesamtüberblick des Forschungsgebiets. Internet-bezogene Anwendungen werden z.B. in [Etz95] beschrieben.

3.3 Bewertung

Die Erläuterungen im vorigen Abschnitt machen deutlich, daß bestehende Verfahren der Datenübertragung entweder auf der Serialisierung der Geschäftsobjekte selbst beruhen oder auf der Serialisierung von Daten-Container-Objekten, die für jedes Geschäftsobjekt implementiert werden, um dessen Attribute zum Transport aufzunehmen. Als grundlegende Alternative hierzu werden dynamische Klassen der Java-Bibliothek verwendet.
Die Serialisierung ganzer Geschäftsobjekte ist grundsätzlich zu kritisieren, da Client und Server dadurch mit dem selben Objektmodell arbeiten. Dies führt zu einer engen Kopplung zwischen Server und Client, was dem Grundprinzip der Trennung zwischen Geschäftslogik und Präsentation einer mehrschichtigen Architektur widerspricht. Falls die Geschäftsobjekte als Enterprise JavaBeans implementiert sind, ist zusätzlich zu kritisieren, daß die direkte Serialisierung von Bean-Instanzen zur Nutzung auf dem Client deren Konzeption als Server-Komponenten, die der Kontrolle des Containers unterliegen und vom Client nie direkt aufgerufen werden können, zuwiderläuft. Darüber hinaus kann bei der Serialisierung die zu übertragende Datenmenge nur schwer kontrolliert werden. Grundsätzlich werden die Zustände ganzer Objekte und Objektgraphen übertragen. Ein weiteres Problem kann die Tatsache sein, daß in den Geschäftsobjekten wichtige Algorithmen implementiert sind, die nicht an externe Clients weitergegeben werden sollen. Java-Klassen können dekompiliert werden und darin enthaltenes Know-how ist nicht geschützt. Diese allgemein in Java vorhandene Problematik wird z.B. in [Nol97] thematisiert.
Verfahren, die eine Verwendung von statischen Daten-Container-Objekten vorschlagen, sind insoweit zu kritisieren, als

- die Ansätze eher auf die Anwendung mit Architekturen, die auf Entity-Beans basieren, ausgelegt sind und nicht ausdrücklich berücksichtigen, daß Geschäftsobjekte grundsätzlich als reine Java-Objekte modelliert sein können, die der Kontrolle eines Persistenz-Frameworks unterliegen. Dieser Ansatz wird jedoch häufig in der Praxis angewendet.
- für jedes Geschäftsobjekt eine eigene Container-Klasse implementiert werden muß, die dessen Attribute enthält. Je mehr Geschäftsobjekte in einem Anwendungssystem benötigt werden, desto mehr Container-Klassen werden benötigt. Dieses Vorgehen ist mit hohem Aufwand zur Erstellung und zur Pflege der Container-Klassen verbunden. Außerdem benötigen diese Klassen Speicherplatz und müssen im Falle eines Java-Applets zum Client übertragen werden.

- die Container-Klassen alle Attribute enthalten, wodurch immer alle Attribute zwischen Client und Server übertragen werden, obwohl diese nicht immer komplett benötigt werden. Dieser Umstand ist besonders kritisch, falls die Geschäftsobjekte sehr viele Attribute besitzen. Selbst wenn einige Attribute nicht belegt sind oder vor der Übertragung durch Löschen deren Inhalts „ausgeblendet“ werden, müssen sie durch die Systemumgebung berücksichtigt werden.

- pro Geschäftsobjekt mehrere Container-Klassen implementiert werden müssen, um die Anzahl der übertragenen Attribute tatsächlich zu beschränken. Damit existieren noch mehr Klassen, die mit relativ hohem Aufwand implementiert und gewartet werden müssen. Zusätzlich besteht die Gefahr der Unübersichtlichkeit, da es im Ermessen jedes einzelnen Anwendungsentwicklers liegt, wieviele Transport-Container bereitgestellt werden und zu welchem Zweck diese erforderlich sind.

- keinerlei zusätzlich Funktionalität im Sinne der Optimierung der Datenübertragung vorgesehen ist, die zu verschiedenen Zeitpunkten, vor, während oder nach der Datenübertragung in unterschiedlicher Granularität angewendet werden kann. Des weiteren ist keine dynamische Konfiguration, die zur Laufzeit oder beim Deployment die Datenübertragung beeinflußt vorgesehen. Damit geht ein hohes Maß an Flexibilität verloren, um auf Anforderungsänderungen zu reagieren und auch noch nachträglich das Übertragungsverhalten zu modifizieren.

Die Verwendung von dynamischen JDK-Daten-Container-Objekten ist insoweit zu kritisieren, als

- keine Zusatzfunktionalität während der Übertragung ausgeführt werden kann und die Datenstruktur festliegt, ohne noch geändert werden zu können.

- der Umgang mit den Objekten teilweise sehr mühsam ist und die bessere Datenstrukturierung nur durch zusätzlichen Aufwand und eigene Überlegungen jedes einzelnen Anwendungsentwicklers realisiert werden kann.

- die Datenspeicherung aufgrund der Struktur der Objekte mehr Platz benötigt, was insbesondere dann negativ zum Tragen kommt, wenn Listen und Massendaten übertragen werden müssen.

- die Anforderung von Attributen, die nicht im Daten-Container enthalten sind zu keinem Fehler führen, sondern zur Rückgabe des Wertes `null`. Dies ist z.B. bei `HashMap`-Objekten der Fall. Der Wert `null` kann jedoch eine fachliche Bedeutung haben und muß deshalb vom Fall des Nicht-Enthalten-Seins abgegrenzt werden. Bei einem `HashMap`-Objekt müßte dies umständlich durch Aufruf einer weiteren Methode geschehen.

- die Verwendung von speziellen dynamischen Klassen, die z.B. im Rahmen einer Datenbankanfrage zurückgegeben werden, zur Verwendung eines Objekts der Datenschicht auf dem Client führt und darüber hinaus eine Abkehr von anderen Konzepten, die im System vorhanden sind, bedeutet.

- die Verwendung als Datentransportobjekt nicht unmittelbar aus den Klassen selbst hervorgeht. Sie werden hauptsächlich für andere Datenspeicherungszwecke verwendet, die von der Verwendung als Transportklasse zu unterscheiden sind. Insgesamt ist der Verwendungszweck dieser Klassen nicht auf die Datenübertragung ausgelegt.

Allgemein etabliert keines der Verfahren ein universelles Konzept, mit dem beliebige Daten, unabhängig von der Architektur des Anwendungssystems, strukturiert übertragen werden können. Die Übertragungskonzepte gehen i.d.R. davon aus, daß eine Architektur verwendet wird, die aus Session und Entity-Beans bestehen. Dieser Architekturansatz wird jedoch häufig nicht verwendet oder durch andere Ansätze ergänzt. Aufgrund der fehlenden Einheitlichkeit ist es nur schwer und mit hohem Aufwand möglich, die Verfahren universell und zentral bereitzustellen und darauf weitere Funktionalitäten, die häufig während der Entwicklung benötigt werden, abzustimmen. Damit bleibt ein hohes Potential zur Einsparung von Entwicklungsaufwand und zur zentralen, wiederverwendbaren Bereitstellung von Funktionen, die mit der Datenübertragung zusammenhängen oder direkt an diese anschließen, ungenutzt. Insgesamt wird der Datenübertragung nur ein geringer Stellenwert eingeräumt, obwohl viele Systemanforderungen, die sich im Laufe der Zeit auch ändern können, damit verbunden sind. Dies drückt sich häufig auch darin aus, daß die Form der Datenübertragung überhaupt nicht festgelegt wird, was zur Mischung der unterschiedlichsten Konzepte führt.
Insgesamt sind in der Literatur nur wenig Hinweise über leistungsbeeinflussende Faktoren bei der Datenübertragung im Zusammenhang mit EJBs und deren Laufzeitumgebung in Form von Applikations-Servern vorhanden. Dies macht es schwierig zu entscheiden, worauf bei der Implementierung von Transportobjekten zu achten ist. Ein vollständiges Datenübertragungskonzept muß diese Aussagen aber treffen und berücksichtigen. Dies ermöglicht dann die Anpassung der Datenübertragung an die Anforderungen, die an die Gesamtanwendung gestellt werden.
Trotz der negativen Kritik, die an dieser Stelle an Java geübt wird, eignet sich diese Sprache aufgrund ihrer Plattformunabhängigkeit und dem EJB-Standard für Server-Komponenten sehr gut zur Entwicklung von unternehmenskritischen Anwendungen [Wol01, Rom00]. Dabei existieren zahlreiche Studien, die bei der Verwendung von Java signifikante Produktionskosteneinsparungen nennen [Wol01]. Dabei werden Kosteneinsparungen von bis zu 50% durch die Plattformunabhängigkeit und Produktivitätssteigerungen zwischen 10% und 20% gegenüber der Sprache C genannt. Dies ist auch aufgrund der um 75% gesunkenen Fehlerquote in Java-Programmen der Fall. Diese Eigenschaften wirken sich positiv auf die Entwicklung von Server-Anwendungen aus. Zusätzlich kann eine Java-Anwendung auf jeder Plattform zum Einsatz gebracht werden, die über eine Java-Laufzeitumgebung verfügt. Somit können Anwendungen z.B. auf eine leistungsfähigere Plattform übertragen werden, falls die gegenwärtige nicht mehr ausreicht. Diese Eigenschaft fördert auch die Integration von bestehenden Systemen, die im unternehmensweiten Einsatz in neue Lösungen eingebunden werden müssen. Insgesamt ist die Sprache von Grund auf objektorientiert und verzichtet auf fehleranfällige Techniken, wie z.B. Zeigerarithmetik in C/C++ [Fla98]. Dies führt zu einer leichteren Erlernbarkeit der Sprache. Außerdem sind viele fertige Basis-Komponenten verfügbar bzw. bereits in der Java-Bibliothek integriert. Durch die Verfügbarkeit von *Just-In-Time Compilern*, die ein Java-Programm zur Laufzeit in Maschinencode der zugrundeligenden Plattform übersetzen, wird in vielen Anwendungsbereichen die selbe Performanz oder nur eine geringfügig darunter liegende, wie mit C++ erreicht [Man98, Rom00]. Aufgrund der zahlreichen Vorteile von Java haben sich

Java-Applikations-Server stark verbreitet [Wol01] und zahlreiche kommerzielle Anwendungen werden in diesem Umfeld entwickelt. Die in dieser Arbeit entwickelten Konzepte adressieren daher eine Vielzahl von Anwendungen und zeigen Lösungskonzepte sowie Optimierungs- und Kosteneinsparungspotentiale in deren Entwicklung auf. Dabei werden auch kleine Nachteile von Java mittels anderen Konzepten ausgeglichen. So werden z.B. im Rahmen dieser Arbeit Generatoransätze als Ersatz für den in Java fehlenden Präprozessor verwendet, um statische Optimierungen vorzunehmen.

Kapitel 4

Universelle Datenübertragungskonzepte

In diesem Kapitel werden die im vorigen Kapitel beschriebenen Datenübertragungskonzepte gemäß Abbildung 4.1 durch eigene, neue Konzepte ersetzt bzw. erweitert. Hierzu wird eine zusätzliche Klassifikation eingeführt, die zwischen aktiven und passiven Datenübertragungskonzepten unterscheidet [Bes00]. Passive Konzepte beruhen auf Daten-Containern, die sich darauf beschränken Daten zu speichern, um diese zwischen Client und Server zu übertragen. Im Gegensatz dazu etablieren aktive Konzepte Daten-Container, die eng mit der Datenübertragung verbundene Aufgaben mit übernehmen und über die bloße Speicherung von Daten hinausgehen, indem sie aktiv auf die übertragenen Daten einwirken oder aufgrund von Zugriffen eigenständig Aktivitäten auslösen. Wegen dieser Beschaffenheit werden die Daten-Container zu aktiven Elementen einer Anwendung, die flexibel an wechselnde Anforderungen angepaßt werden und zur Systemoptimierung beitragen können. Es handelt sich dabei nicht um mobile Agenten, da sich die Daten-Container nicht autonom im Netzwerk bewegen können und keine künstliche Intelligenz bei der Datenbeschaffung einsetzen. Ebenso handelt es sich nicht um einen *Smart-Proxy*-Ansatz (vgl. dazu Kapitel 3, Abschnitt 3.2.3), da die Container-Objekte selbst als Datentransportbehälter zur Kommunikation verwendet werden. Während ihrer Verweildauer im Client können die Transportobjekte im Rahmen der Zusatzfunktionalität *zusätzlich* das Verhalten eines *Smart-Proxies* übernehmen.

4.1 Zielsetzung

Nachfolgend soll ein universelles und umfassendes Gesamtkonzept zur Datenübertragung vorgestellt werden. Die Aufgabe der Datenübertragung wird dabei nicht als isolierte Aufgabe in einem Client/Server-System gesehen, sondern als integrativer Bestandteil aller Schichten, der möglichst flexibel, leicht anwendbar und konfigurierbar bzw. jederzeit änderbar sein muß. Dabei wird nicht nur die Übertragung von Daten zwischen zwei Schichten einer mehrstufigen Client/Server-Anwendung angeboten, sondern auch eine Integration auf der Seite von Java-Clients, die eine grafische Darstellung der Daten vornehmen. Dies stellt eine Erweiterung bestehender Konzepte insofern dar, als die Client-Seite nicht ausgespart wird. So enthält z.B. der J2EE-Entwurfsmusterkatalog keine Entwurfsmuster für Java-Clients. Ein weiteres Hauptziel der vorgestellten Konzepte besteht darin, daß Daten unabhängig von ihrer Quelle einheitlich repräsentiert und verarbeitet werden können. Als weitere Ziele soll eine systemweite Verwen-

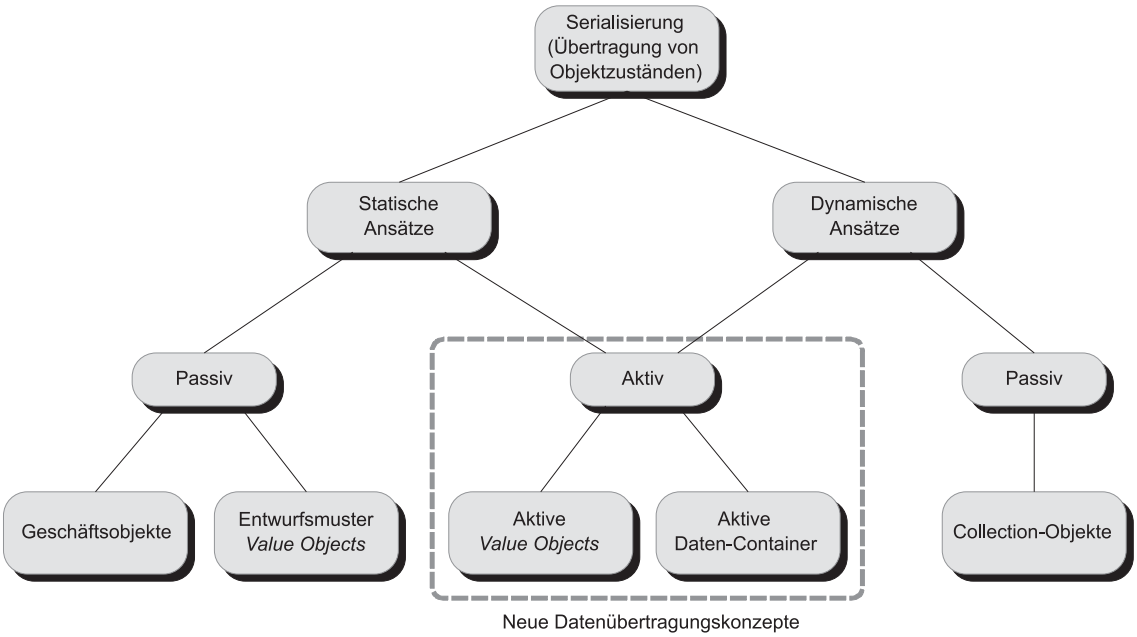


Abbildung 4.1: Neue Datenübertragungskonzepte

dung von Entwurfsmustern, die auf die Datenübertragung anwendbar sind, erreicht und eine einfache Lesbarkeit des erstellten Programmcodes gewährleistet werden. Das Konzept ist in Anhang A als Entwurfsmuster formuliert, da es universell eingesetzt und an spezifische Anforderungen angepaßt werden kann. Die nachfolgende Diskussion orientiert sich an einer Entwurfsmusterbeschreibung. Zentrale Aspekte werden dabei ausführlicher erläutert. Insbesondere geht die Diskussion über eine reine Beschreibung als Entwurfsmuster hinaus, da zunächst die bei der Datenübertragung wirkenden Einflußfaktoren bestimmt werden. Dabei werden neben den Anwendungsaspekten eines Datenübertragungskonzepts auch wesentliche Einflußfaktoren, die das Leistungsverhalten der Datenübertragung im Umfeld von Applikations-Servern bestimmen, untersucht. Diese Erkenntnisse werden benötigt, um einfach anwendbare und optimierte, an die jeweilige Anwendung angepaßte Transportobjekte zu implementieren. Dies ist vor allem in komplexen Geschäftsanwendungen erforderlich, die mit aufwendigen Clients arbeiten und damit hohe Anforderungen an die von der Logikschicht benötigten Daten stellen. Die aktiven Konzepte werden im Rahmen dieses Kapitels zunächst in Form von dynamischen Umsetzungen als *Aktive Daten-Container* vorgestellt. Aufbauend darauf werden statische Formen erläutert, die als *Aktive Value Objects* bezeichnet werden. Die dynamischen Formen beruhen auf der Verwendung eines Daten-Containers für alle Kommunikationsvorgänge. Dabei muß der Daten-Container zur Laufzeit in der Lage sein, alle im System anfallenden Daten für einen Übertragungsvorgang aufzunehmen. Um dieses Ziel zu erreichen, muß der Daten-Container generisch aufgebaut sein, was eine schwächere Typisierung der Datenübergabe- und Datenentnahmemethoden nach sich zieht. Im Gegensatz dazu stehen bei den statischen Formen die zu übertragenden Datentypen bereits fest, d.h. es kann eine strenge Typisierung der zu übergebenden und zu entnehmenden Daten erfolgen. Dies hat jedoch zur Folge, daß ein Daten-Container nicht mehr ausreicht. Statt dessen muß für jeden Datenübertragungsfall, der in einer Anwen-

dung existiert, eine passende Daten-Container-Klasse implementiert werden, die eine Definition der zu übertragenden Attribute und ihres Datentyps vornimmt. Dies führt in Anwendungen dazu, daß pro Geschäftsobjekt mindestens ein spezieller Daten-Container implementiert werden muß. Darüber hinaus muß i.d.R. pro Anwendungsfall, in dem das Geschäftsobjekt verwendet wird, ein weiterer Daten-Container implementiert werden, der nur die für diesen Fall notwendigen Attribute transportiert. Dies verhindert die unnötige Übertragung von Attributen. Solche Entwurfsansätze werden durch das Entwurfsmuster *Value Objects* erfaßt (vgl. Kapitel 3). In dieser Arbeit erfolgen jedoch wesentliche Erweiterungen und Verbesserungen dieses statischen Grundkonzepts durch aktive Elemente. So kann z.B. die Anzahl der benötigten Daten-Container auf maximal ein Exemplar pro Geschäftsobjekt beschränkt werden, das zur Laufzeit in der Lage ist, nicht benötigte Attribute bei Übertragungsvorgängen auszublenden. Um sowohl den bestehenden statischen Konzepten als auch den neuen aktiven Konzepten gerecht zu werden, wird das im Rahmen dieser Arbeit entwickelte Konzept als *Aktive Value Objects* bezeichnet.

4.2 Motivation

In diesem Abschnitt werden wesentliche Probleme der Datenübertragung identifiziert.

4.2.1 Implementierungsaspekte

Bei der Entwicklung von Client/Server-Anwendungen müssen fachlich relevante Daten zwischen Client und Server übertragen werden. Bei einer mehrstufigen Architektur, wie sie in Abschnitt 2 erläutert wurde, werden die Daten aus der dafür vorgesehenen Persistenzschicht bezogen. Als Datenquelle kommen in der Persistenzschicht allerdings sehr viele Systeme in Frage. Neben den üblichen Datenbankmanagementsystemen sind häufig auch Fremdsysteme angebunden. Bei diesen Systemen handelt es sich z.B. um Warenwirtschafts- und Buchhaltungssysteme, aber auch um Systeme anderer Unternehmen, die z.B. Dienstleistungen erbringen oder Lieferanten sind. Häufig handelt es sich bei den Fremdsystemen auch um Altsysteme (sog. *Legacy-Systeme*), die in neue, modernere Anwendungen integriert werden müssen. Die anwendungsbezogenen Daten werden mit verschiedenen Mechanismen in die Anwendungsschicht übernommen und dort in ein Modell aus Komponenten und Objekten abgebildet. Zur Verarbeitung der Daten müssen diese nun zwischen Client und Server übertragen werden. Dabei wird jeder einzelne Entwickler mit den in Abbildung 4.2 dargestellten Aufgaben konfrontiert:

1. **Datenbeschaffung**. Die vom Client angeforderten Daten müssen beschafft werden, d.h. es muß z.B. die Formulierung einer Anfrage, die Daten aus der Persistenzschicht beschafft, erfolgen.

2. **Datenformat**. Nachdem die Daten aus der Persistenzschicht beschafft wurden, muß die Struktur bekannt sein, in der sie vorliegen. Es kann sich um objektorientierte Strukturen handeln, deren Objekte entsprechend der fachlichen Domäne untereinander in Beziehung stehen oder es kann sich um technisch motivierte Objekte handeln, die anhand ihrer Schnittstelle Zugriff auf Datensätze gewähren.

3. **Datenentnahme.** Das Datenformat muß vollständig verstanden werden, damit Daten entnommen werden können.

4. **Übertragungsformat.** Nachdem Daten entnommen werden können, muß das Übertragungsformat festgelegt werden. D.h. es muß eine Form festgelegt werden, die zum Client geschickt wird, um sie dort weiterzuverarbeiten.

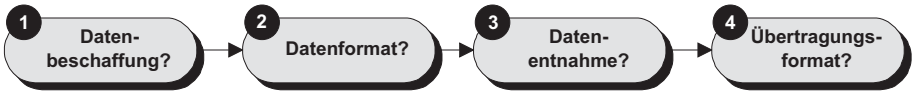


Abbildung 4.2: Aufgaben bei der Datenübertragung

Sendet der Client Daten zum Server zurück, muß der ganze Vorgang umgekehrt stattfinden. Aus den Aufgaben der Datenübertragung ergeben sich eine Reihe von Problemen und daraus resultierende Anforderungen, die von bestehenden Datenübertragungsmechanismen nicht abgedeckt werden:

- **Durchgängigkeitsproblematik.** Der gewählte Übertragungsmechanismus wird häufig in Abhängigkeit des Problemfelds und den Kenntnissen des Entwicklers gewählt. Somit werden in einem großen System viele unterschiedliche Lösungsansätze gewählt, die bei der Erweiterung, beim Test und bei der Wartung des Systems analysiert und durchschaut werden müssen. Dies kann insbesondere dann zu Problemen führen, wenn der Anwendungsentwickler, der das Konzept umgesetzt hat nicht mehr greifbar ist. Ein direkt damit zusammenhängendes Problem ist, daß Anwendungsentwickler ihre Übertragungskonzepte isoliert voneinander verwenden und damit eine Reihe von Einsparungspotentialen verlorengehen, die aufgrund der Existenz von allgemeingültigen Funktionen zwar existieren, aber mehrfach implementiert werden, weil kein einheitliches Datenübertragungskonzept vereinbart wird. Als weitere Folge davon, kann nur schwer auf Probleme reagiert werden, die sich aus geänderten Anforderungen an das Anwendungssystem ergeben und sich unmittelbar auf die Datenübertragung auswirken. Änderungen müssen dann an sehr vielen Stellen im Anwendungssystem auf individuelle Weise durchgeführt werden. Aus den selben Gründen ist es auch schwierig wiederverwendbare, allgemeingültige Standardkomponenten zu implementieren, die anschließend zum Kauf für andere Anwendungen angeboten werden. Hierbei besteht häufig das Problem, daß die Daten von den Komponenten nicht in der Form geliefert werden, wie sie in einer individuellen Anwendung, die in der selben Anwendungsdomäne wie die Standardkomponente angesiedelt ist, benötigt werden. Das Entwurfsmuster *Value Objects* trägt nicht ausreichend zur Verbesserung dieser Situation bei, da es schwerpunktmäßig auf die Übertragung der Daten von Entity-Beans ausgelegt ist. Es geht nicht auf andere Anforderungen ein, die existieren, falls Architekturen ohne Entity-Beans verwendet werden oder Daten transportiert werden müssen, die aus anderen Quellen stammen und in Session-Beans anfallen. Die direkte Übertragung des Zustands von Geschäftsobjekten reicht ebenfalls nicht aus, da damit Fälle, in denen Daten von mehreren Geschäftsobjekten auf dem Server zusammengefaßt und zur

Darstellung auf dem Client aufbereitet werden sollen, in keiner Weise abgedeckt sind. Die Verwendung von dynamischen Collection-Klassen der Java-Bibliothek kann nur sehr umständlich erfolgen, da z.B. evtl. benötigte Zusatzfunktionalität (z.B. Datenkompression) als separater Mechanismus implementiert werden muß, der wiederum ein neues Übertragungsformat erforderlich macht.

- **Kopplungsproblematik.** Eines der Hauptziele beim Entwurf einer mehrschichtigen Architektur ist die Konzentration von Geschäftsobjekten, Geschäftsprozessen und dafür benötigten Algorithmen in einer dafür vorgesehenen Schicht. Durch die Trennung dieser Schicht von der Datenhaltungsschicht und Client-Schicht soll die unternehmensweite Wiederverwendbarkeit der Dienste gewährleistet und die zentrale Wartung, Änderung und Erweiterung gefördert werden. Die direkte Serialisierung von Geschäftsobjekten verletzt dieses Entwurfsziel grundsätzlich. Sind die Geschäftsobjekte als EJBs realisiert, wird zusätzlich die Grundidee der Etablierung von Komponenten, die sich ausschließlich unter der Kontrolle eines Containers auf dem Server befinden, unterlaufen. Insgesamt entsteht dabei eine Kopplung zwischen Server und Client über die Geschäftsobjekte. Bei der Verwendung von *Value Objects* wird faktisch eine zusätzliche Schicht von Objekten eingeführt, wodurch ebenfalls eine Kopplung zwischen Client und Server entsteht. Eine starke Kopplung führt bei jeder Änderung eines Geschäfts- oder Transportobjekts zu einer Neuübersetzung aller Clients, obwohl sie nicht direkt von der Änderung betroffen sind.

- **Kostenproblematik.** Wie aus den Aufgaben der Datenübertragung ersichtlich ist, besteht ein hoher Entwicklungsaufwand darin, die Daten aus den vorliegenden Objektstrukturen zu entnehmen und in ein Übertragungsformat zu überführen, oder umgekehrt aus einem Übertragungsformat in eine vorliegende Objektstruktur einzufügen. Dieses Problem existiert vor allem bei der Verwendung von *Value Objects* und dynamischen Collection-Klassen. Entsprechend dem Muster *Value Objects* wird für jede Entity-Bean eine weitere Klasse zum Datentransfer benötigt. Dies hat zur Folge, daß für jede Entity-Bean eine zusätzliche Klasse benötigt wird, die implementiert und gewartet werden muß. Je mehr Entity-Beans in einem System vorhanden sind, desto mehr Transportobjekte werden benötigt. Das Entwurfsmuster läßt sich ohne größere Schwierigkeiten auch auf persistente Objekte übertragen, die nicht als Entity-Beans realisiert sind (vgl. dazu Kapitel 3). Dabei existiert pro persistentem Objekt eine Datenübertragungsklasse und damit die um Faktor zwei vergrößerte Anzahl an zu implementierenden und zu wartenden Klassen. Dieses Problem wird signifikant verschärft, wenn weitere Transportobjekte entsprechend den fachlichen Anwendungsfällen erstellt werden, die nur die Attribute enthalten, die tatsächlich in einem Arbeitsschritt des Clients benötigt werden. Das erstellen weiterer Transportobjekte kann der Leistungsoptimierung dienen (vgl. Abschnitt 4.2.2). Zusätzlich kann dadurch aber eine Unübersichtlichkeit entstehen, die darauf beruht, daß Anwendungsentwickler viele Transportklassen schreiben, die sich nur marginal unterscheiden. Die Notwendigkeit der Übertragung vieler Transportobjekte (gleichen oder unterschiedlichen Typs), um einen geforderten Anwendungsfall zu implementieren, erfordert wiederum die Erstellung weiterer Transportklassen, die zur Sammlung der benötigten *ValueObjects* verwendet werden. Dies führt aber wiederum zu steigenden Kosten und einer mögli-

chen Unübersichtlichkeit, da viele weitere Klassen implementiert werden. Eine Alternative sind hier die Verwendung von *Collection*-Klassen, wie z.B. `Vector` oder `HashMap`. Der Einsatz der Klasse `Vector` führt jedoch zu Problemen, da auf die Daten mit einem Index zugegriffen werden muß. Um bestimmte Daten aus dem Vektor zu entnehmen, muß der betreffende Index bekannt sein. Ändert sich die Position der Daten im Vektor ist dies schwer zu verfolgen. Diese Situation ist bei der Verwendung eines `HashMap`-Objekts deutlich besser, da hier die Daten anhand von aussagekräftigen Schlüsseln identifiziert werden können. Es verbleibt jedoch das Problem, daß die Collection-Klassen nicht ausdrücklich für die Datenübertragung bestimmt sind und damit in Programmen auch nicht problemlos als Objekte mit diesem Zweck identifiziert werden können. Das Verhalten dieser Objekte kann nicht vollständig beeinflußt oder zentral geändert und erweitert werden.

- **Flexibilitätsproblematik.** Ändert ein Client seine Anforderungen dahingehend, daß er andere Daten oder eine größere Datenmenge anfordern will, muß eine Änderung der Transportobjekte erfolgen. Kommt ein neuer Client mit spezifischen Anforderungen hinzu muß mindestens ein neues Transportobjekt implementiert werden. Die Schnittstelle der Server-Komponente muß gemäß der neu hinzukommenden Transportobjekte um eine analoge Anzahl von Methoden erweitert werden. Stellt sich nachträglich heraus, daß eine Beeinflussung der Datenübertragung erfolgen muß, z.B. durch spezielle Kompressionsalgorithmen, muß dies nachträglich in alle Transportobjekte eingebaut werden.

- **Zeitproblematik.** Falls sich Transportobjekte ändern oder neu hinzukommen, weil neue Clients mit anderen Anforderungen existieren bzw. vorhandene Clients ihre Anforderungen ändern, muß die Applikation umfangreich geändert werden und dadurch neu übersetzt und auf dem Server installiert werden. Dabei ändern sich auch die Schnittstellen von Komponenten, was einen kompletten Generierungslauf (*Stubs*, *Skeletons* und *EJB-Object*) zur Folge hat. Dieser Vorgang kann sehr lange dauern. Dies behindert auch während der Entwicklung, wenn häufig Änderungen vorgenommen und neu auf dem Server propagiert werden müssen.

- **Ressourcenproblematik.** Falls ein komplexer Client realisiert werden muß, der in Form eines Applets auf dem Client gestartet wird, kann die Anzahl der vorhandenen Transportklassen problematisch sein, da diese alle zum Client hin übertragen werden müssen. Das selbe Problem existiert, wenn der Platzbedarf für den Java-Client möglichst gering sein soll und dennoch sehr viele Transportklassen verwendet werden sollen.

- **Sicherheitsproblematik.** Bei der kompletten Übertragung von Geschäftsobjektzuständen oder *Value Objects*, die den kompletten Zustand eines Geschäftsobjekts aufnehmen, kann der unerwünschte Effekt auftreten, daß sicherheitsbezogene Daten mit übertragen werden, die normalerweise auf dem Server verbleiben müssen. Um dies zu verhindern, müssen umfangreiche Vorkehrungen getroffen werden, um sensible Daten vor dem Versenden zum Client auszublenden. Ein weiterer Sicherheitsaspekt liegt in der Auslieferung der Geschäftsklassen selbst an den Client. Dies kann aus Gründen des Know-how-Schutzes unerwünscht sein, falls in den Geschäftsklassen Algorithmen implementiert sind, die ge-

heim gehalten werden sollen. Aufgrund der Möglichkeit übersetzte Java-Klassen zu dekompilieren ist die Anforderung der Geheimhaltung nicht gegeben.

- **Eindeutigkeitsproblematik.** Die Collection-Klassen sind nicht explizit für die Datenübertragung vorgesehen. Dies erschwert das Auffinden von Programmteilen, die sich mit der Datenübertragung befassen. Das selbe Problem existiert bei der direkten Übertragung von Geschäftsobjekten und der Übertragung von *Value Objects*, falls diese keiner Namenskonvention folgen.

- **Benutzbarkeitsproblematik.** Die Collection-Klassen besitzen nicht die notwendige Funktionalität, um die Anforderungen einer dynamischen Datenübertragung vollständig zu erfüllen. Dies führt zu einem höheren Entwicklungsaufwand, da die Funktionalität nachgebildet werden muß. Ein Beispiel hierfür ist die Unterscheidung, ob ein Attribut nicht in einer `HashMap` enthalten ist oder ob es enthalten ist, aber sein Wert `null` ist. In beiden Fällen ist das Ergebnis einer Abfrage des Attributs `null`. Das Fehlen des Attributs muß aber erkannt werden und zur Erzeugung einer Exception führen, wenn der aktuelle Geschäftsvorgang das Attribut voraussetzt.

Zusammenfassend läßt sich festhalten, daß bestehende Verfahren bisher viele Problematiken nicht oder nur unzureichend berücksichtigen. Insgesamt wird die Datenübertragung zu isoliert betrachtet, wodurch Einsparungspotentiale und ein hohes Maß an Flexibilität verlorengehen. Ebenso werden die Leistungsaspekte der Datenübertragung in modernen EJB-Systemen nur ungenügend beleuchtet und bei der Erstellung von Anwendungen zu stark vernachlässigt. Häufig kommt Leistungsaspekten aber in Anwendungen eine hohe Bedeutung zu. Aus diesem Grund wird im folgenden Abschnitt untersucht, welche zentralen Einflußfaktoren im Rahmen der Datenübertragung in EJB-Systemen eine Rolle spielen.

4.2.2 Leistungsaspekte

In der gängigen EJB-Literatur finden sich nur sehr wenig Hinweise, die nur sehr unzureichend Auskunft darüber geben, was wesentliche Einflußfaktoren bei der Kommunikation in EJB-Systemen sind und wie stark sie sich auswirken. Dabei kommt erschwerend hinzu, daß sehr viele kommerzielle und nichtkommerzielle Applikations-Server existieren, die der EJB-Spezifikation entsprechen. Die Implementierung dieser Server ist den Herstellern aber weitestgehend freigestellt. Detaillierte Kenntnisse über die Einflußfaktoren der Datenübertragung sind allerdings erforderlich, um negative Effekte auf das Leistungsverhalten der Anwendung zu verhindern oder darauf wirksam zu reagieren. Aus diesem Grund wurde im Rahmen dieser Arbeit eine umfangreiche Untersuchung durchgeführt, um diese Einflußfaktoren zu ermitteln. Zur Verallgemeinerung der Aussagen wurden gängige kommerzielle Applikations-Server, aber auch Produkte aus dem Open-Source-Bereich auf ihre Datenübertragungseigenschaften hin untersucht. Die Ergebnisse der Untersuchung dienen jedoch keinem Produktvergleich oder der Produktauswahl. Aus diesem Grund wurde jedem Applikations-Server eine eindeutige Nummer zugeordnet, ohne seinen Namen zu nennen.

Untersuschungsumgebung

Die Untersuchung wurde in der folgenden Umgebung durchgeführt:

- **Server:**
  - **Prozessor:** Pentium III, 800 MHz,
  - **Hauptspeicher:** 256 MB,
  - **Betriebssystem:** Windows 2000,
  - **Java-Version:** 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode),
  - **Applikations-Server:** Die Applikations-Server wurden standardmäßig installiert (*Out-of-Box-Installation*). Der Java-Heap wurde einheitlich auf 128 MByte gesetzt.
- **Client:**
  - **Prozessor:** Pentium II, 350 MHz,
  - **Hauptspeicher:** 256 MB,
  - **Betriebssystem:** Windows NT 4.0,
  - **Java-Version:** 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode).
- **Netzwerk:** 10 MBit/s Ethernet.

Sämtliche Messungen wurden mehrfach durchgeführt und Mittelwerte aus den erhaltenen Ergebnissen gebildet. Die Tests selbst wurden ebenfalls mehrfach durchgeführt und verifiziert, um eventuelle Unregelmäßigkeiten auszuschließen. Nachfolgend werden wesentliche Ergebnisse der Untersuchung dargestellt.

Datenmenge

Als signifikanter Einflußfaktor ist zunächst die zu übertragende Datenmenge zu sehen. In Abbildung 4.3 ist dargestellt, wie die Übertragungszeiten bei Steigerung der Datenmenge zunehmen. Übertragen wurde dabei eine Folge von Bytes (Typ: `byte[]`). Dabei kann festgestellt werden, daß sich die Applikations-Server bei kleineren Datenmengen ähnlich verhalten und auch annähernd gleiche Übertragungszeiten liefern. Auffällig ist der Server S2, der eine reproduzierbare Anomalie der Übertragungszeit bei Datenmengen um 1 KByte zeigt. Der Server S7 quittiert den vorliegenden Test bei 1000 KByte Daten mit einem Fehler. Aus dem selben Grund mußte in allen weiteren Untersuchungen auf den Test von S7 mit einer Datenmenge von 1000 KByte verzichtet werden.
Die Untersuchung macht deutlich, daß sich die gängigen Applikations-Server hinsichtlich ihrer Datenübertragungseigenschaften zunächst wenig unterscheiden. Die gemessenen Übertragungszeiten steigen annähernd linear mit der Datenmenge an. Qualitätsunterschiede in den Implementierungen lassen sich jedoch bereits erkennen, so besitzen einige Server konstant überdurchschnittlich gute Übertragungszeiten. Die festgestellten Anomalien deuten jedoch auf Probleme hin, die sich bei der Entwicklung einer Anwendung negativ auf die erzielten Antwortzeiten von entfernten Methodenaufrufen auswirken können. Die Fehlermeldung eines Servers

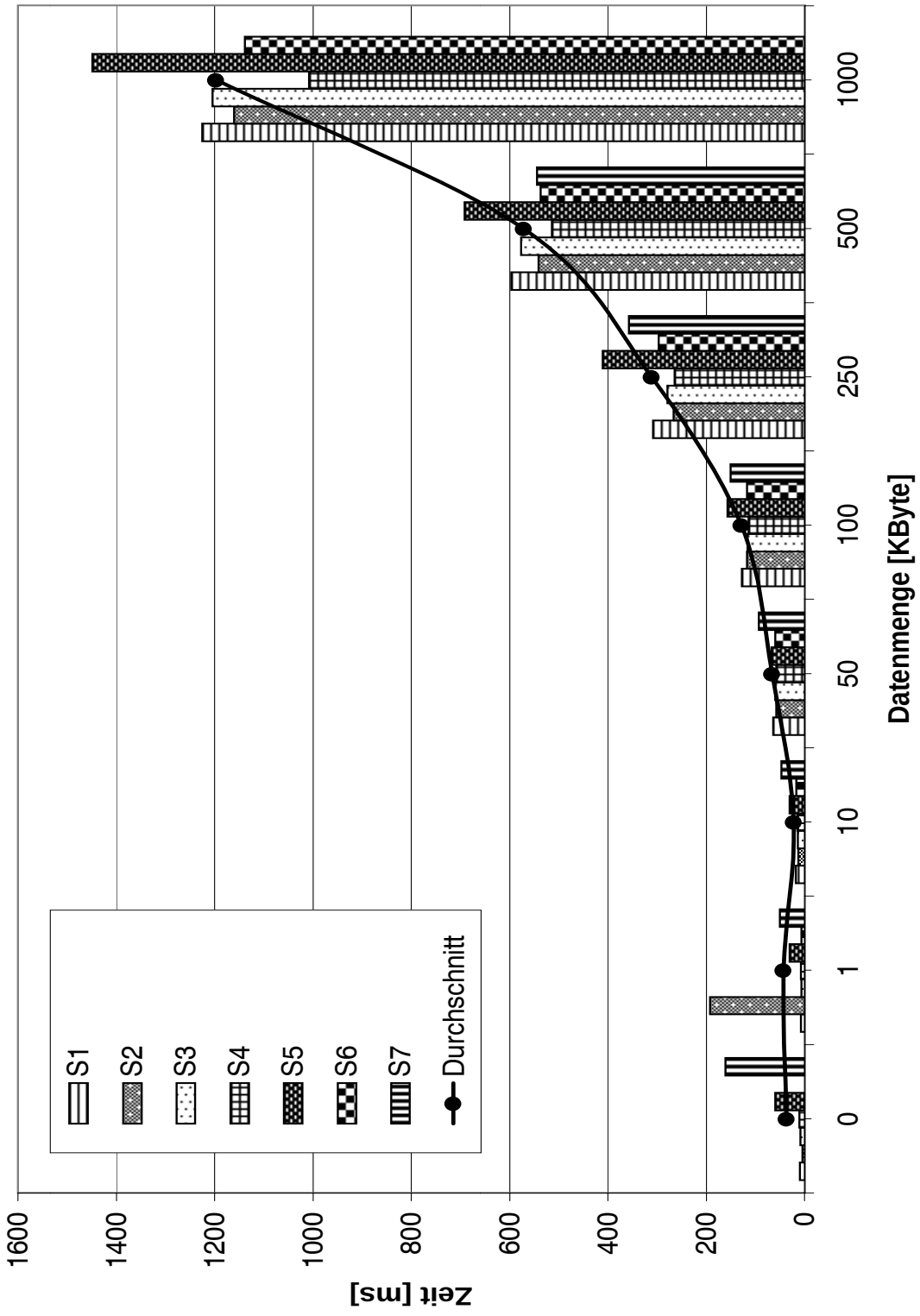


Abbildung 4.3: Übertragungszeit in Abhängigkeit der Datenmenge

macht deutlich, daß es bei der Übertragung großer Datenmengen zu unerwarteten Problemen kommen kann, die im schlimmsten Fall zur Unbrauchbarkeit einer Anwendung führen.
Aufgrund der Auswirkung der Datenmenge ist es als kritisch anzusehen, daß bei direkter Übertragung des Zustands von Geschäftsobjekten immer alle Attribute auf einmal übertragen werden, obwohl diese für den betreffenden Arbeitsschritt der Anwendung nicht erforderlich sind. Dies ist insbesondere dann der Fall, wenn Geschäftsobjekte sehr viele Attribute besitzen oder die Objektzustände von vielen Objekten übertragen werden müssen (Übertragung von Massendaten). Selbst wenn nicht alle Attribute tatsächlich mit Werten belegt sind, verursachen leere Datentypen entsprechend der Bitbreite des Datentyps bzw. der Kennzeichnung als „leer“ eine Grunddatenmenge, die transportiert werden muß. Die selbe Situation tritt auf, wenn pro Geschäftsobjekt, das z.B. als Entity-Bean implementiert ist, ein *Value Object* mit allen vorhandenen Attributen implementiert wird.
Bei der Verwendung von Collection-Klassen kann grundsätzlich die Datenmenge durch die gezielte Speicherung von zu übertragenden Attributen eingeschränkt werden. Um einen übersichtlichen Zugriff auf die Daten zu ermöglichen, müssen jedoch i.d.R. die Namen der Attribute mit übertragen werden (Verwendung `HashMap` oder `Hashtable`). Dies führt wiederum bei der Massendatenübertragung dazu, daß eine höhere Datenmenge anfällt, da die Attributnamen mehrfach redundant übertragen werden.
Allgemein kann eine Optimierung der Datenmenge zu einer Verbesserung des Leistungsverhaltens führen, da die Übertragungszeiten verkürzt werden. Dieser Faktor ist auch insbesondere dann wichtig, wenn die Bandbreite des Netzwerks eingeschränkt ist bzw. bei der gegebenen Systemlast nicht ausreicht. Zusätzlich kann der Applikations-Server aufgrund des geringeren Verarbeitungsaufwands mehr Anfragen abarbeiten.

Datenkomplexität

Bei den in Abbildung 4.3 übertragenen Daten handelt es sich um die Übertragung einer Folge von Bytes und damit um eine geringe Anforderung an die Übertragungsimplementierung der Server, da die Komplexität der Datenstruktur sehr gering ist. Unter Komplexität soll hier die Struktur der verwendeten Objekttypen und deren Schachtelungstiefe in einer zusammengesetzten Datenstruktur (Klasse) verstanden werden.
In Abbildung 4.4 sind die Übertragungszeiten und die anfallenden Datenmengen nach der Serialisierung für komplexere Datenstrukturen dargestellt. Die Daten niedriger Komplexität bestehen dabei wieder aus einem einfachen Array von Bytes. Die Daten mittlerer Komplexität bestehen aus einem Vektor, der mehrere Byte-Arrays enthält. Die Daten hoher Komplexität bestehen aus einem Vektor, der eine Folge von Vektoren enthält, die wiederum jeweils eine Folge von `String`-Objekten enthalten. Die Abbildung macht deutlich, daß die Datenmenge bei der Serialisierung größer wird, wenn bei gleicher Nutzdatenmenge die Datenstruktur variiert wird. Obwohl die Datenmenge bei hoher Komplexität im Vergleich zu niedriger Komplexität nur um 17,9% angewachsen ist, steigt die Übertragungszeit überproportional um durchschnittlich 129,2% an. Der Server S3 reagiert dabei mit einem Anstieg der Übertragungszeit um 82% am geringsten, der Server S5 mit 222,4% am stärksten.
Zur Isolierung des Einfluß der Datenkomplexität sind in den Abbildungen 4.5 und 4.6 die Übertragungszeiten von Daten höherer Komplexität in Form von ineinandergeschachtelten String-

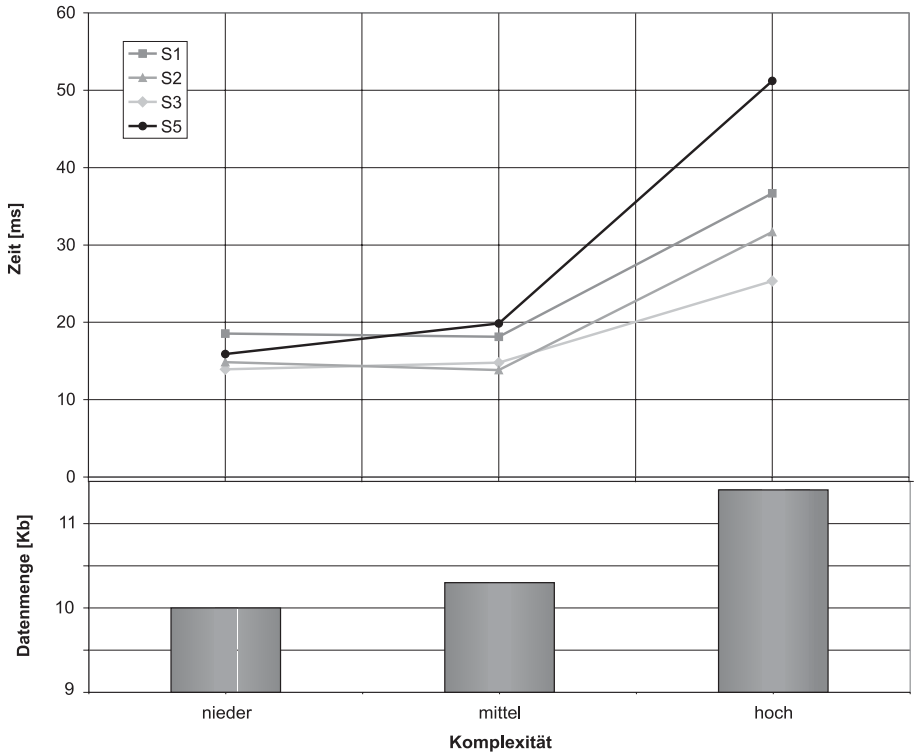


Abbildung 4.4: Übertragungszeit in Abhängigkeit von Datenkomplexität und Datenmenge

Folgen dargestellt. Die Aufteilung der Ergebnisse beruht auf den erhaltenen Übertragungszeiten, die sich vor allem bei hohen Datenmengen in unterschiedlichen Dimensionen bewegen. Während die in Abbildung 4.5 dargestellten Server (nachfolgend Gruppe 1 genannt) bei einer Datenmenge von 1000 KByte eine um ca. Faktor zwei schlechtere Übertragungszeit, als bei primitiven Daten liefern, zeigen die in Abbildung 4.6 dargestellten Applikations-Server (nachfolgend Gruppe 2 genannt) deutliche Effekte mit Übertragungszeiten, die sich im Minutenbereich bewegen. Bei den Applikations-Servern der Gruppe 2 handelt es sich um Produkte, die auf einer CORBA-Infrastruktur basieren und mittels dem Protokoll IIOP kommunizieren. Insgesamt wird deutlich, daß die Übertragung großer komplexer Datenmengen zu großen Problemen führen kann.
In Applikationen, die eine komplexe Anwendungsdomäne abdecken, können leicht Datenstrukturen entstehen, die ebenso umfangreich wie komplex sind. Die Erstellung solcher Datenstrukturen wird ausdrücklich durch den Java-Serialisierungsmechanismus, der die Basis von Übertragungsvorgängen in EJB-Systemen bildet, gefördert. Komplexe Objektgraphen werden ausdrücklich, ohne Implementierungsbelastung von Anwendungsentwicklern berücksichtigt (vgl. Kapitel 2, Abschnitt 2.3). Im Ergebnis kann dies jedoch zu einem signifikant schlechteren Leistungsverhalten bis hin zur Unbrauchbarkeit der Anwendung führen.
Der Einflußfaktor Datenkomplexität läßt sich in die folgenden Ursachen zerlegen:

- **Marshalling-Algorithmus.** Die Kommunikation-*Stubs* und -*Skeletons* müssen die Daten

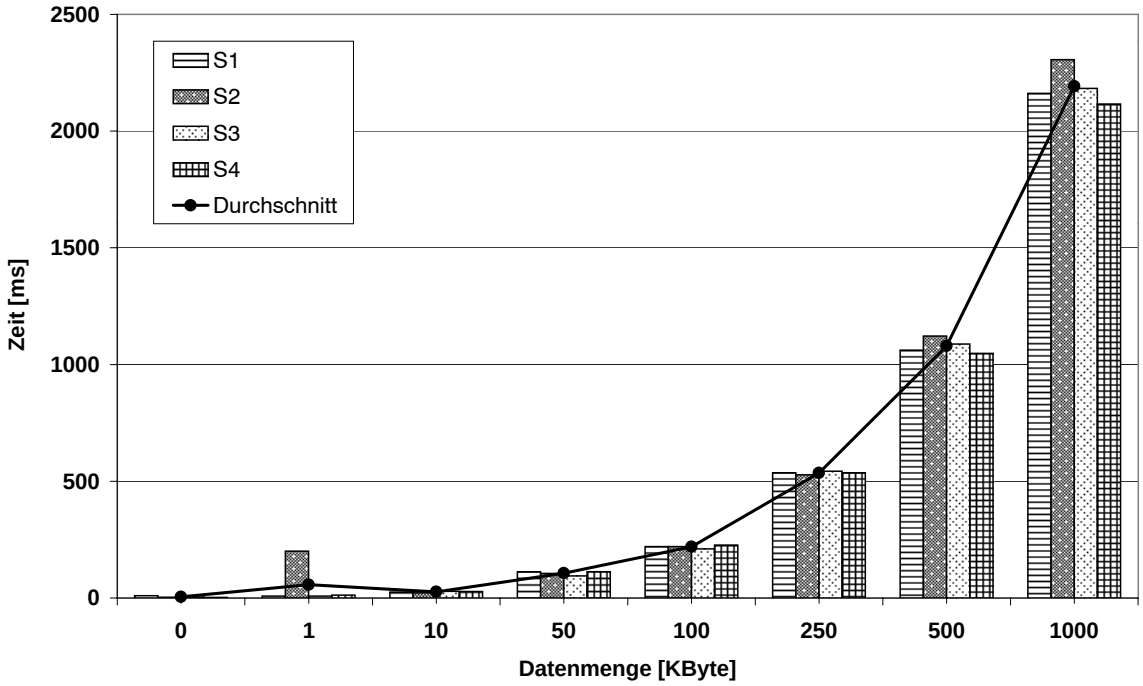


Abbildung 4.5: Übertragungszeit in Abhängigkeit der Datenmenge (hohe Datenkomplexität)

zur Übertragung in ein anderes Format überführen (vgl. dazu Kapitel 2, Abschnitte 2.3 und 2.2). Bei der Verwendung komplexer Datenstrukturen fällt dieser Vorgang umfangreicher aus, als bei weniger komplexen.

- **Datentypisierung.** Bei schwach typisierten Transportobjekten muß der genaue Typ der zu versendenden Daten ermittelt werden. Dies ist z.B. beim Typ *Object* und Sammlungen davon in Objekten vom Typ `java.util.Collection` notwendig.

- **Java-Objekterzeugung.** Das Erzeugen von Java-Objekten ist allgemein eine teure Operation [Sin97]. Komplexe Datenstrukturen sind häufig aus vielen Java-Objekten zusammengesetzt. Bei der Rekonstruktion der Datenstruktur nach einem Kommunikationsvorgang müssen diese Datenstrukturen in allen Einzelobjekten neu erzeugt werden.

- **Serialisierung.** Die Kommunikation mit RMI und RMI-IIOP basiert auf dem Java-Serialisierungsprotokoll (Kapitel 2, Abschnitt 2.3). Die Java-Serialisierung selbst ist nicht effizient [Nes99].

- **Datenmenge.** Häufig geht die Übertragung von komplexen Datenstrukturen mit einer größeren Datenmenge einher. Aufgrund der Zusammensetzung von geschachtelten Datenstrukturen aus vielen unterschiedlichen Objekten und Datentypen sind mehr Metainformationen über die transportierten Objekte erforderlich, um eine Rekonstruktion nach dem Kommunikationsvorgang leisten zu können. Die Datenmenge wird in dieser Arbeit als separater Einflußfaktor betrachtet.

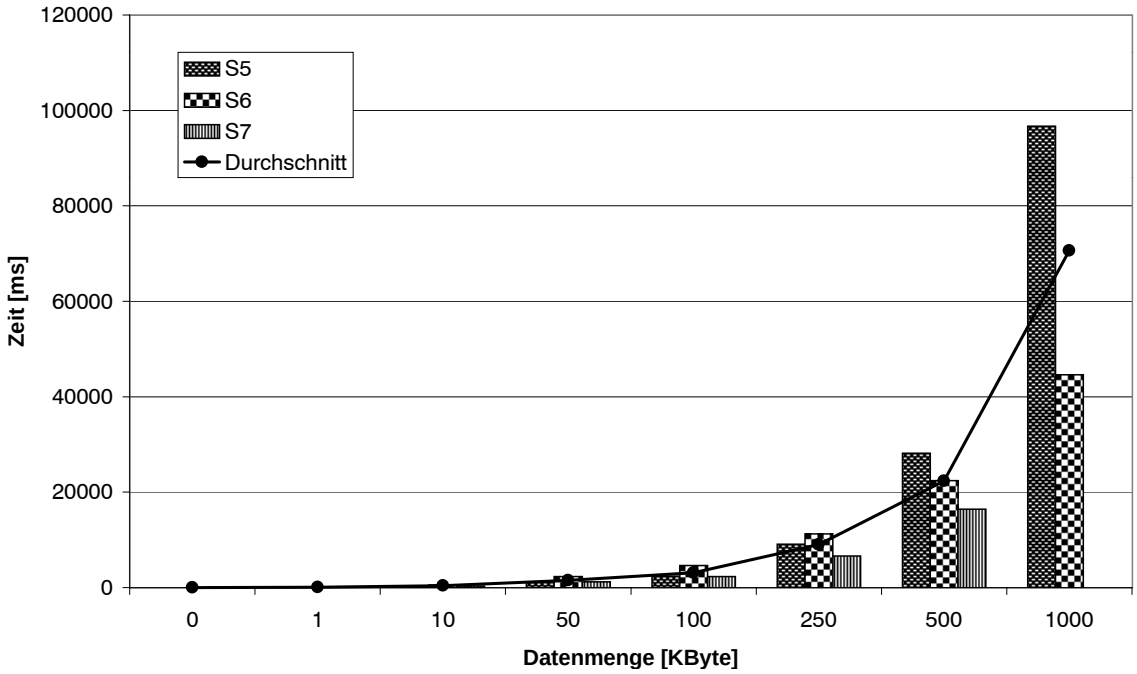


Abbildung 4.6: Übertragungszeit in Abhängigkeit der Datenmenge (hohe Datenkomplexität)

Der Faktor Datenkomplexität macht wiederum deutlich, daß es unbedingt erforderlich ist komplexe Attribute bei der Übertragung wegzulassen, falls diese beim vorliegenden Anwendungsfall nicht benötigt werden. Die Reduktion der Datenkomplexität stellt damit einen weiteren Ansatzpunkt zur Optimierung der Systemleistung dar. Bei einer Verringerung der Datenkomplexität ist wiederum zu erwarten, daß der Applikations-Server aufgrund der geringeren Verarbeitungsanforderungen mehr Anfragen abarbeiten kann. Häufig tritt dabei auch eine Reduktion der Datenmenge auf, die ebenfalls zu einer Optimierung der Systemleistung führt.

Protokolleinfluß

Bei näherer Untersuchung von Einflußfaktoren, die sich beim Aufruf von entfernten Methoden auswirken, kann festgestellt werden, daß sich das vom Applikations-Server verwendete Protokoll ebenfalls auf die Dauer von entfernten Aufrufen auswirkt. In Abbildung 4.7 ist der Einfluß des Protokolls dargestellt. Der Applikations-Server S3 kann mit unterschiedlichen Protokollen betrieben werden. Als Alternative zum gewöhnlichen RMI-Protokoll kann ein proprietäres Protokoll verwendet werden. Ein leerer Methodenaufruf ist bei dem Alternativprotokoll um durchschnittlich 56% schneller. Bei der Übertragung von 1000 KByte Daten niedriger und hoher Komplexität ist das optimierte Protokoll durchschnittlich um 34% schneller.
Es läßt sich also festhalten, daß die Verwendung eines proprietären Protokolls ebenfalls zur Optimierung der Leistung des Anwendungssystems dienen kann. Im Rahmen dieser Arbeit wurde die Erfahrung gemacht, daß die Verwendung eines proprietären Protokolls immer zu einem besseren Leistungsverhalten bei der Datenübertragung geführt hat.

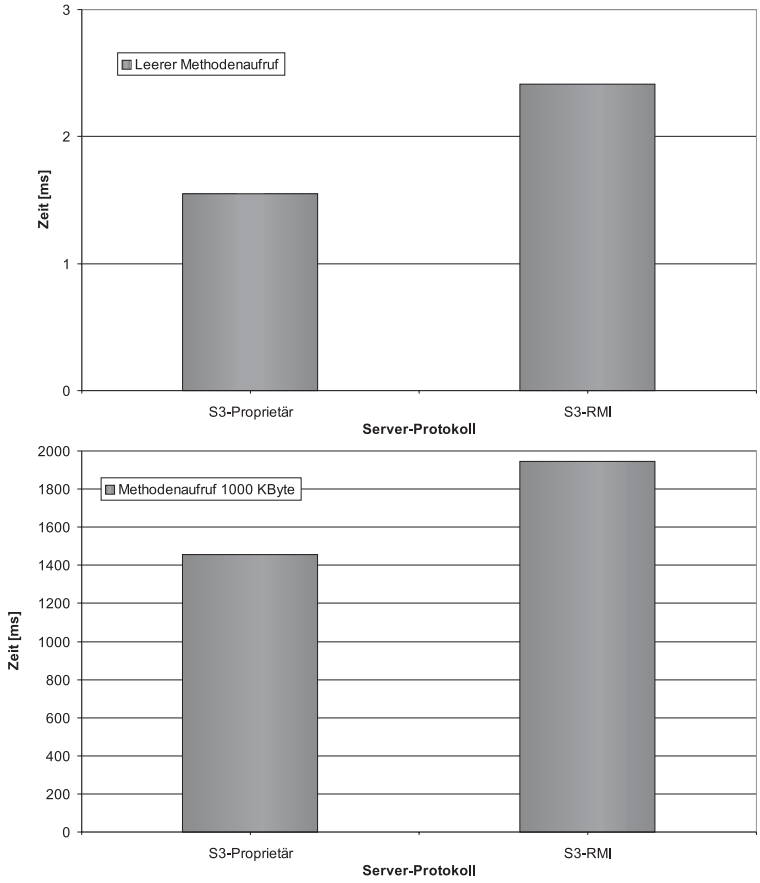


Abbildung 4.7: Protokolleinfluß

Lokale Kommunikation

Die Datenübertragung ist auch im Rahmen einer lokalen, serverinternen Kommunikation relevant, da Applikationen aus vielen EJB-Komponenten zusammengesetzt werden können, die sich innerhalb des selben Servers befinden und miteinander kommunizieren. In Abbildung 4.8 sind Applikations-Server dargestellt, die keine Optimierung bzw. eine nicht ausreichende Optimierung der Datenübertragung vornehmen. Dabei wurden wiederum zunächst Daten in Form einer Folge von Bytes und damit geringer Komplexität übertragen. Es ist eine Zunahme der Übertragungszeit mit steigender Datenmenge zu beobachten, die mit dem Anstieg bei echter Kommunikation über das Netzwerk vergleichbar ist. Aufgrund der lokalen Kommunikation fallen die Übertragungszeiten gegenüber der Kommunikation über ein Netzwerk (vgl. Abbildung 4.3) aber deutlich geringer aus.

Die in Abbildung 4.9 dargestellte Server-Gruppe nimmt eine wirksame Optimierung der serverinternen Kommunikation vor und liefert so weitgehend konstante Übertragungszeiten, die unabhängig von der Datenmenge sind und dabei deutlich geringer ausfallen als die der Abbildung 4.8. Teilweise konnte keine Übertragungszeit nachgewiesen werden (d.h. Meßwert 0). Dies ist auf die dafür zu geringe Auflösung der Methode `currentTimeMillis()` der Klasse

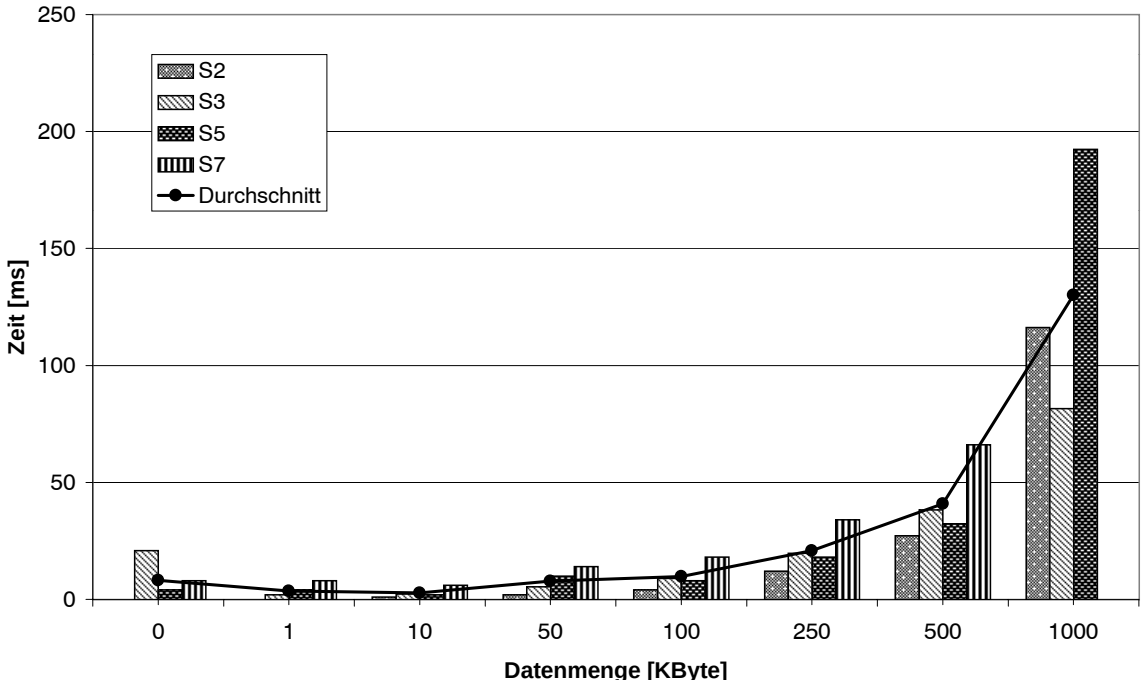


Abbildung 4.8: Lokale Übertragungszeit in Abhängigkeit der Datenmenge bei geringer Datenkomplexität

`java.lang.System`, die zur Zeitaufnahme verwendet wurde, zurückzuführen.

In Abbildung 4.10 ist die Gruppe der Server abgebildet, die keine oder eine nur unzureichende Optimierung der internen Kommunikation vornehmen. Aufgrund unterschiedlicher zeitlicher Dimensionen ist wiederum eine getrennte Darstellung vorgenommen worden. Insgesamt zeigt sich, daß der zeitliche Verlauf mit den Verläufen in den Abbildungen 4.5 und 4.6 prinzipiell vergleichbar ist, aber aufgrund der lokalen Kommunikation auf einem niedrigeren Niveau abläuft. Der Server S5 fällt hier jedoch bei 1000 KByte durch einen massiven Einbruch der Übertragungszeit auf über eine Minute auf. Der Server S7 ist auch lokal nicht in der Lage eine Datenmenge von 1000 KByte zu übertragen.

Dagegen sind in Abbildung 4.11 die Server mit effektiver Optimierung der internen Kommunikation dargestellt. Die Übergabe von Daten ist hier unabhängig von deren Menge und Komplexität.

Die interne Kommunikation ist beim Erzeugen von Aufrufketten unter EJBs zu berücksichtigen. Aufrufketten entstehen, wenn eine Methode implementiert wird, die zur Erbringung ihres Dienstes wiederum auf Methoden in anderen Komponenten zurückgreift, die ihrerseits ebenfalls Methoden in anderen Komponenten aufrufen, um ihre Dienste zu erbringen. Neben der Übermittlung von Daten zwischen den Aufrufen unterliegt jeder einzelne Aufruf gemäß dem EJB-Konzept der Kontrolle des EJB-Containers, um z.B. transaktionales Verhalten sicherzustellen. In Abbildung 4.12 sind die durchschnittlichen Übertragungszeiten von EJB-Ketten unterschiedlicher Länge dargestellt. Dabei wurden Ketten aus Entity-Beans und Session-Beans berücksichtigt. Zusätzlich ist der Einfluß der dabei übertragenen Datenmenge (0, 1, 10 und 50

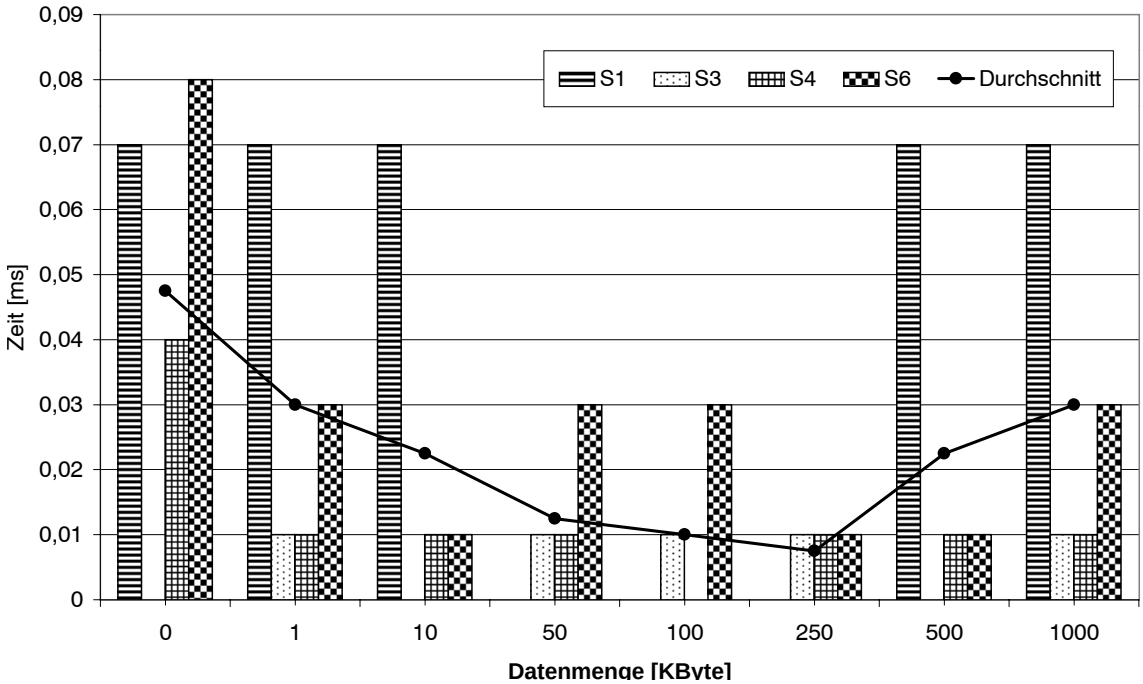


Abbildung 4.9: Lokale Übertragungszeit in Abhängigkeit der Datenmenge bei geringer Datenkomplexität

KByte) sichtbar.
Das Ergebnis zeigt, daß die Verwendung von EJB-Ketten bis zur Länge von 10 EJBs aus Sicht der Datenübertragung unproblematisch ist. Dies sollte für den praktischen Einsatz in Geschäftsanwendungen ausreichend sein. Server, die keine Optimierung der serverinternen Kommunikation vornehmen, zeigen einen Anstieg der Übertragungszeit mit steigender EJB-Anzahl und steigender Datenmenge. Dagegen bleibt die Übertragungszeit bei Servern mit optimiertem Protokoll nahezu konstant und ist damit unabhängig von der übertragenen Datenmenge.
Aus den Untersuchungsergebnissen lassen sich eine Reihe von Aussagen ableiten, die bei der Umsetzung eines Datenübertragungskonzepts berücksichtigt werden können und Hinweise auf die Implementierung konkreter Transportklassen und -datenstrukturen geben:

- Bei der Übertragung hoher Datenmengen müssen signifikante Probleme einkalkuliert werden, was das Leistungsverhalten und die Robustheit einer EJB-Anwendung anbelangt. Gegebenenfalls müssen auf mehreren Ebenen Gegenmaßnahmen ergriffen werden, um die Datenmenge pro Methodenaufruf auf ein für die verwendete Plattform vernünftiges Maß zu beschränken. Diese Anforderung steht häufig im Konflikt zu fachlichen Anforderungen, die z.B. die Übertragung von Massendaten vorsehen.

- Das beste Leistungsverhalten ist zu erwarten, wenn die Struktur der verwendeten Transportobjekte möglichst einfach ist und damit die Komplexität der zu übertragenden Daten möglichst gering ist. Aus diesem Grund sollten möglichst keine Objektgraphen übertragen werden und als Objektattribute sollten nur primitive Java-Typen verwendet werden.

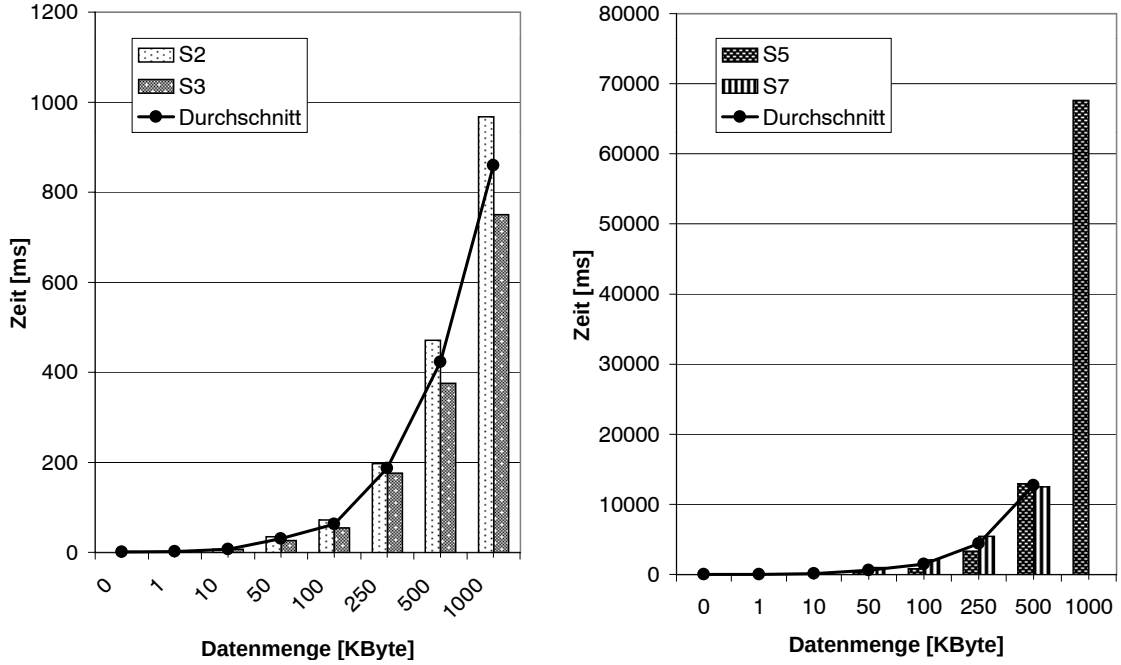


Abbildung 4.10: Lokale Übertragungszeit in Abhängigkeit der Datenmenge bei hoher Datenkomplexität

Somit kann erreicht werden, daß die Anzahl der zu behandelnden Objekte reduziert wird, was in den meisten Fällen neben der Reduktion der Komplexität auch automatisch eine Reduktion der Datenmenge, aufgrund der Einsparung von Objektmetadaten, zur Folge hat. Zusätzlich wird der zeitkritische Vorgang der Erzeugung und Vernichtung von Objekten im Rahmen der automatischen Speicherverwaltung von Java entschärft. Diese Anforderung steht häufig im Konflikt zu fachlich komplexen Anwendungen, die zu ebenso komplexen, untereinander verknüpften Datenstrukturen führen, die sich im Sinne der objektorientierten Anwendungsentwicklung auch auf Implementierungsebene widerspiegeln.

- Der Datentransport im Sinne der Menge und Komplexität ist nicht ausschließlich ein Problem bei echter verteilter Kommunikation über ein Netzwerk, sondern auch bei interner Kommunikation im Server. Im Sinne des Leistungsverhaltens ist es nicht ausreichend eine Optimierung erst dann vorzunehmen, wenn tatsächlich über ein Netzwerk kommuniziert wird. Viele Applikations-Server entschärfen das Problem durch optimierte interne Kommunikation. Falls dies nicht der Fall ist, sollten nach Möglichkeit die neuen lokalen Schnittstellen für EJBs bereitgestellt werden, die nur innerhalb eines Servers aufgerufen werden können und ausdrücklich eine Optimierung durch Übergabe per Referenz erlauben. Ist dies nicht möglich, muß so optimiert werden, als ob eine Kommunikation über das Netzwerk erfolgt.

Aufgrund der vorliegenden Ergebnisse werden im folgenden Abschnitt neue Datenübertra-

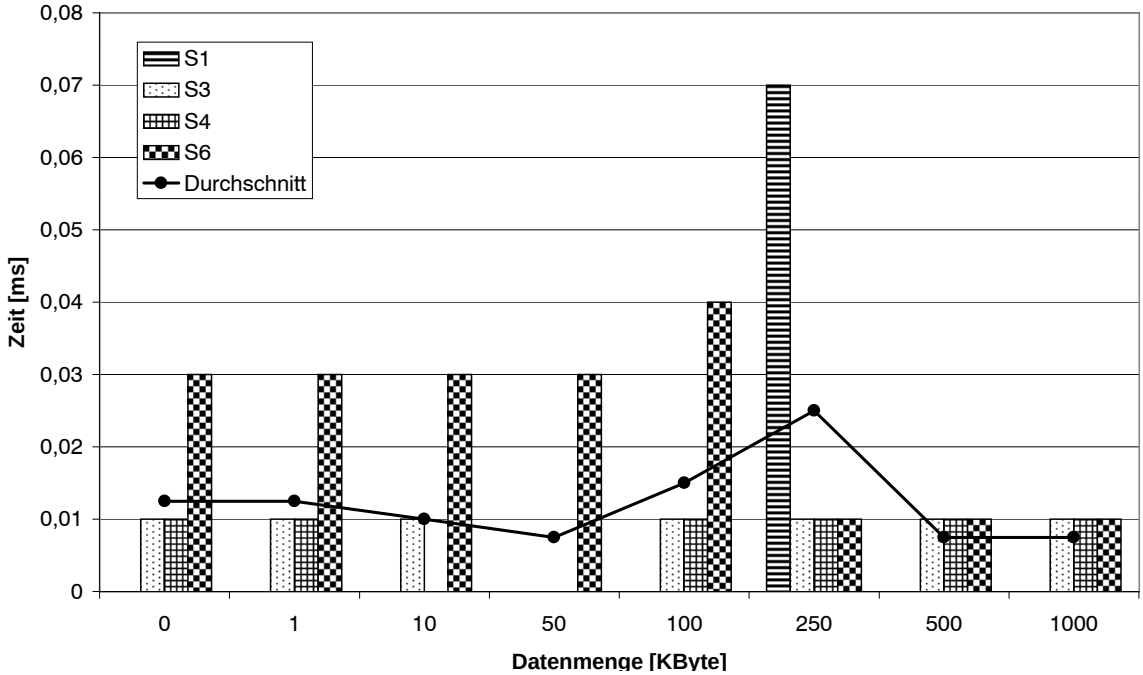


Abbildung 4.11: Lokale Übertragungszeit in Abhängigkeit der Datenmenge bei hoher Datenkomplexität

gungskonzepte entwickelt, um eine Verbesserung in Fragen der Implementierung und des Leistungsverhaltens zu erzielen.

4.3 Aktive Daten-Container

Ein *Aktiver Daten-Container (ADC)* ist ein Transportobjekt, das auf die Aufgaben der Datenübertragung spezialisiert ist [Bes00]. Die Objekte besitzen Schnittstellen, die vom Client und vom Server benutzt werden. Im Gegensatz zu passiven Daten-Containern kapselt ein ADC neben Daten auch Dienste, die mit der Datenübertragung zusammenhängen und erlaubt damit auch die gezielte Einflußnahme auf die transportierten Daten. Bei komplexen Funktionen kann ein ADC auch als Fassade (vgl. dazu *Facade* in [Gam95]) für Objekte dienen, die diese Dienste erbringen.

4.3.1 Eigenschaften

Das Konzept des Aktiven Daten-Containers verhindert die im vorigen Abschnitt erläuterten Probleme, die allgemein bei der Entwicklung von Datenübertragungsmechanismen auftreten dadurch, daß ein fertiger, universeller Mechanismus zentral bereitgestellt wird, der nachträglich transparent konfiguriert und geändert werden kann. Das Konzept besitzt die folgenden Eigenschaften:

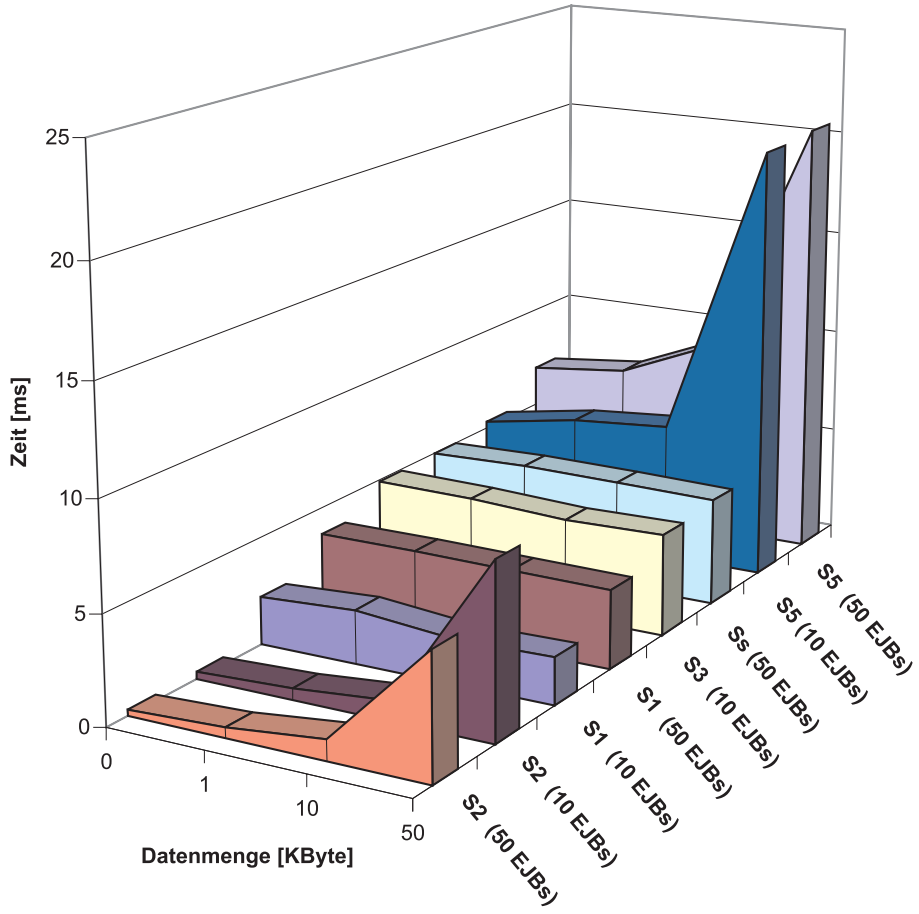


Abbildung 4.12: Lokale Übertragungszeit verschiedener Datenmengen in Abhängigkeit der EJB-Kettenlänge

- **Durchgängigkeit.** Die Daten-Container werden konsequent in allen Schichten der Anwendung eingesetzt. Durch die standardisierte Schnittstelle stellt sich die Anwendung der Konzepte immer gleich dar. Im Quellcode können Programmteile, die sich mit der Datenübertragung beschäftigen, identifiziert werden. Gleichzeitig wird die Benutzung der Konzepte durch Anwendungsentwickler festgelegt (Benutzungskonzept) und kann ggf. auch durch die Daten-Container selbst auf Korrektheit geprüft werden.

- **Kostenfaktor.** Die dynamische Fassung der Daten-Container wird zentral in einer Klasse bereitgestellt. Wartungen und Erweiterungen des Daten-Containers erfolgen zentral und transparent für die Anwendungsentwickler. Durch die Dynamik und Universalität der Klassen sind sie wiederverwendbar und können so z.B. Bestandteil eines Frameworks werden. Vorhandene Geschäftsklassen bleiben frei von technisch notwendiger Funktionalität. Diese Funktionalität ist in einer eigenen Klasse gekapselt. Eine Ersparnis von Entwicklungsaufwand wird erzielt, indem das Füllen und Auslesen des Daten-Containers auf dem Server automatisch erfolgt. Aufgrund der allgemeinen Schnittstelle des Daten-Containers können andere Anwendungsteile auf ihn abgestimmt werden. Dies trifft z.B.

auf Mechanismen der serverseitigen Präsentationsschicht zu, die aus den im Container enthaltenen Attributen eine andere Darstellung generieren oder auf dem Anwendungs-Client direkt den Datenaustausch zwischen Daten-Container und GUI-Elementen übernehmen.

Die statische Fassung der aktiven Konzepte beruht auf einer Weiterentwicklung des Entwurfsmusters *Value Objects* um die in diesem Abschnitt vorgestellten Mechanismen. Dabei wird auf die Einsparung von Transportklassen für jedes Geschäftsobjekt bzw. Datenobjekt, dessen Daten transportiert werden sollen, verzichtet.

- **Leistungsfaktor.** Die Attributmenge kann zur Übertragung auf die vom Anwendungsfall abhängige Menge begrenzt werden. Dies entspricht einer Umsetzung des Entwurfsmusters *Dynamische Attribute* [Mow97], das automatisch von jedem Entwickler verwendet wird.

  Der ADC fördert die strukturierte Übertragung von Daten mehrerer Datenquellen, wie z.B. Datenzugriffsobjekte und Geschäftsobjekte. Dies verhindert automatisch die Problematik, daß zu viele entfernte Methodenaufrufe getätigt werden, um verschiedenartige Daten zu transportieren. Gleichzeitig kann Einfluß auf die Datenmenge und -komplexität genommen werden, die über das Netzwerk transportiert werden muß, indem effiziente Speicherformen oder Kompressionsalgorithmen angewendet werden. Aufgrund des aktiven Verhaltens des Containers kann z.B. die Größe des Daten-Containers überprüft werden und bei Überschreitung einer kritischen Grenze können Maßnahmen ergriffen werden.

- **Kopplungsproblematik.** Das Objektmodell der Applikationsschicht wird vor dem Client verborgen. Vorhandenes Know-how wird geschützt, da Java-Klassen auf dem Server verbleiben und somit von einem externen Client nicht dekompiliert werden können. Zusätzlich wird keine weitere Abhängigkeit zu einer Reihe von *Value Objects* geschaffen.

- **Flexibilitätsproblematik.** Durch die dynamische Datenübertragung kann jederzeit die Übertragung zusätzlicher Daten erfolgen. Durch eine allgemeine Methode kann jeder Client Daten anfordern, die er benötigt. Neue oder geänderte Anforderungen haben keine neuen Daten-Container-Objekte oder Änderungen in vorhandenen Container-Objekten zur Folge. Daten werden unabhängig von der Form, in der sie vorliegen, im Daten-Container verpackt. Zusatzfunktionalität kann im Daten-Container zur Ausführung gebracht werden und auch systemweit transparent für die Anwendungsentwickler aktiviert werden. Im Daten-Container können Kontrollfunktionen realisiert werden, die z.B. überprüfen, ob Daten überhaupt zur Übertragung vorgesehen sind. Eine weitere Prüfung betrifft die Menge der übertragenen Daten. Bei Überschreitung einer konfigurierbaren Grenze kann ein Hinweis erfolgen, daß die Datenmenge zu groß wird und Maßnahmen ergriffen werden sollten, um das Leistungsverhalten des Gesamtsystems zu verbessern.

- **Zeitfaktor.** Die Verwendung von ADCs als Parameter in der Remote-Schnittstelle von EJBs trägt zur Stabilität dieser bei. Müssen Daten zusätzlich übertragen werden oder muß der Typ von übertragenen Daten geändert werden, erfordert dies keine Änderung der

Remote-Schnittstelle. Zusätzlich existiert eine Zeitersparnis, weil die Kommunikations-Stubs für Client und EJB-Container nicht neu erzeugt werden müssen.

- **Ressourcenfaktor.** Im dynamischen Fall sind nur zwei Transportklassen notwendig, die zum Client übertragen werden müssen. Falls es sich um ein Applet handelt, wird der Ladeprozeß entsprechend beschleunigt und der Speicherbedarf ist geringer gegenüber der Verwendung vieler Container-Klassen oder der direkten Verwendung von Geschäftsobjekten.

- **Sicherheitsproblematik.** Das Konzept schränkt die zu transportierenden Attribute auf die wirklich benötigten ein. Sicherheitsrelevante Daten können so auf dem Server verbleiben.

- **Benutzbarkeit.** Die Container-Klasse ist auf die Aufgabe der Datenübertragung spezialisiert. Dabei kann z.B. das Fehlen eines Attributs durch die Erzeugung einer Exception behandelt werden.

Aufgrund der Eigenschaften des Konzepts bietet es auch eine mögliche Lösung für Standardkomponenten, die nachträglich auf die unterschiedlichsten Projektanforderungen im Sinne der Datenübertragung angepaßt werden können, obwohl kein Quellcode für die Komponente vorliegt.

4.3.2 Struktur

Der ADC führt die Aufgaben der Datenübertragung, wie sie in Abbildung 4.2 dargestellt sind, in einer Klasse, bzw. in einer Schnittstelle zusammen und bietet zusätzlich noch die Möglichkeit an, weitere Operationen auf den transportierten Daten auszuführen. Dieses Grundprinzip ist in Abbildung 4.13 dargestellt. Durch diese Vorgehensweise erfolgt eine signifikante Entlastung des Entwicklers. Es verbleibt die Aufgabe, die Daten aus der Datenquelle zu beschaffen und sie dem ADC zu übergeben (1). Es ist nicht erforderlich, das Datenformat zu kennen (2), die Daten zu entnehmen (3) und ein Übertragungsformat festzulegen (4). Die Auseinandersetzung mit evtl. notwendiger Zusatzfunktionalität (5) ist nicht erforderlich, falls diese grundsätzlich eingesetzt werden muß und somit vor dem Entwickler weitgehend verborgen ist. Ist die Zusatzfunktionalität vom jeweiligen Anwendungsfall abhängig, beschränkt sich der Aufwand für den Entwickler darauf, zu entscheiden, ob die Funktionen eingesetzt werden sollen.
In Abbildung 4.14 werden unterschiedliche Objekttypen, deren Daten im Daten-Container transportiert werden können, dargestellt. Grundsätzlich können auch Richtlinien definiert werden, welche Objekte dem Container zum Datentransport übergeben werden dürfen. Aufgrund des aktiven Verhaltens des Daten-Containers kann zur Laufzeit überprüft werden, ob nur Objekte übergeben werden, die den Richtlinien entsprechen.
Bei den Objekttypen handelt es sich um:

- **Geschäftsobjekte:** Als Geschäftsobjekte sollen Objekte bezeichnet werden, die eine Entität in der fachlichen Problemdomäne modellieren. Dabei kann es sich bspw. um Objekte, wie „Kunde“, „Adresse“ und „Konto“ handeln. Es handelt sich dabei um Objekte, die das Ergebnis einer objektorientierten Analyse (OOA) sind. Hierbei muß beachtet werden,

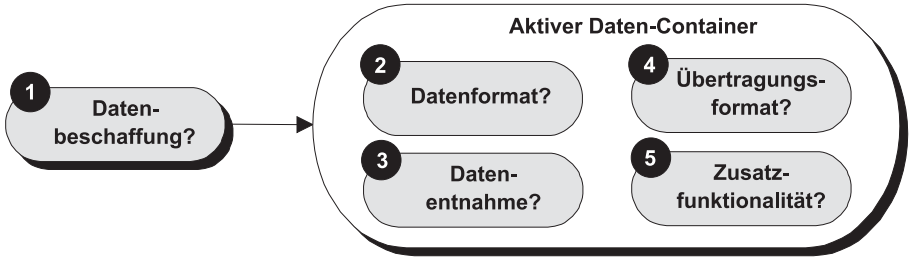


Abbildung 4.13: Aufgaben bei der Datenübertragung mit Aktiven Daten-Containern

daß die Form der Realisierung von der gewählten Architektur abhängig ist. Die Objekte können als „normale“ Java-Objekte vorliegen oder als EJBs realisiert sein (vgl. dazu 3).

- **Technische Objekte:** Als technische Objekte sollen Objekte bezeichnet werden, die nicht das Ergebnis einer OOA sind, sondern aufgrund der technischen Realisierung der Anwendung notwendig sind. Ein Beispiel hierfür sind Objekte, die zur Beschaffung von Daten benötigt werden. Dabei kann es sich z.B. um ein `ResultSet`-Objekt handeln, das die Ergebnismenge einer JDBC-Datenbankanfrage enthält. Ein weiteres Beispiel sind Objekte, die zu Zwecken der Darstellung im Client benötigt werden. Sie sind vor allem dann notwendig, wenn die Aufbereitung der Daten auf dem Server stattfindet. Ein Objekt dieser Art ist z.B. die Repräsentation der Zeile einer grafischen Tabelle. Bei technischen Objekten kann es sich aber auch um ADC-Objekte selbst handeln, da z.B. die Schachtelung von Daten-Containern zur Übertragung von Massendaten unterschiedlichen Typs erlaubt werden kann.

Gemäß Abbildung 4.15 können die Daten sämtlicher Objekte im Anwendungssystem als ADC-Objekte repräsentiert werden. Die Integrations- und Ressourcenschicht ist dabei aus Vereinfachungsgründen zusammengefaßt.
Darüber hinaus existieren Erweiterungen, deren Basis auf diesen Daten-Containern beruht. Die Daten-Container verhalten sich nicht passiv, sondern aktiv, indem sie Aufgaben übernehmen,

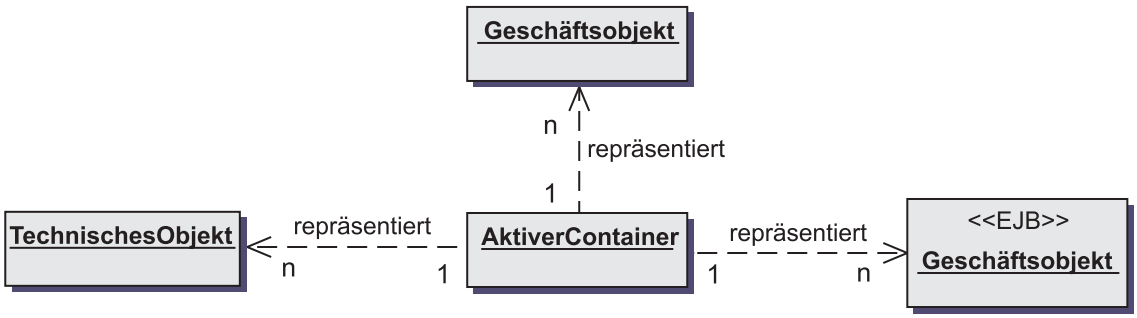


Abbildung 4.14: Objekttypen für den Transport im Aktiven Daten-Container

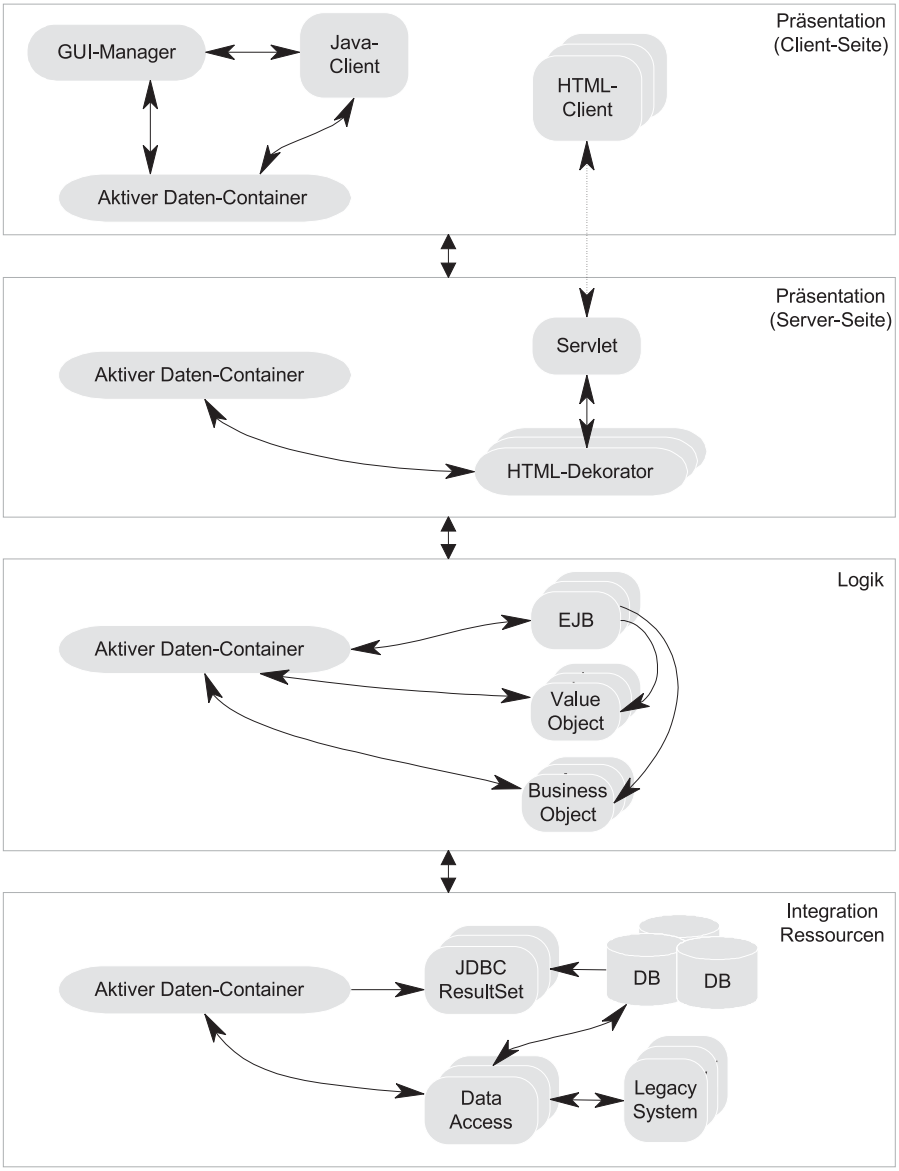


Abbildung 4.15: Einordnung der Konzepte

die direkt oder indirekt mit der Datenübertragung zusammenhängen. Dabei können verschiedene Ziele verfolgt werden. Diese bestehen z.B. darin, Entwicklungsaufwand einzusparen oder die Systemleistung zu optimieren. Zusätzlich wird die Aufgabe der Datenübertragung weitgehend vereinheitlicht und nachträglich konfigurierbar und erweiterbar gemacht. Dabei wird eine hohe Flexibilität bei gleichzeitiger Kontrolle über einen wichtigen Designfaktor innerhalb eines verteilten Systems erzielt. Nachfolgend werden die in Abbildung 4.15 dargestellten Konzepte kurz erläutert:

- **Clientseitige Präsentation.** Allgemein können die Daten-Container von Java-Clients di-

rekt verwendet werden, um Daten zu lesen und zu schreiben. Das Konzept des GUI[1]-Managers basiert auf den Daten-Containern und automatisiert den Datenaustausch zwischen diesen und den grafischen Elementen des Clients, um die Entwicklung von komplexen Benutzerschnittstellen zu vereinfachen und Entwicklungsaufwand zu sparen. Zusätzlich soll aufgezeigt werden, daß sich ein Datenübertragungskonzept auf alle Schichten einer Anwendung beziehen muß und auch auf Client-Seite Muster sinnvoll angewendet werden können. Im Gegensatz dazu finden sich in [Dee01] keine Muster für die Präsentation auf Seite des Clients. Die direkte Arbeit mit den Daten-Container wird in Abschnitt 4.3 beschrieben. Der GUI-Manager wird in Abschnitt 4.5.1 erläutert.

- **Serverseitige Präsentation.** Auf Seite des Servers können die Daten-Container als Transportobjekte für Daten verwendet werden, die speziell zur Darstellung auf dem Client aufbereitet werden. Dazu gehören insbesondere auch Dekoratoren für die Daten-Container, die Daten so aufbereiten, daß sie in ein anderes Format übertragen werden. Als Beispiel ist ein HTML[2]-Dekorator dargestellt, der von einem Servlet genutzt wird, um die in ADCs transportierten Daten für HTML-Clients, wie z.B. Internet-Browser darstellbar zu machen. Das selbe Konzept kann auch dazu verwendet werden, um die Daten z.B. in WML[3], XML[4] oder ein CORBA-konformes Datenformat umzuwandeln. Die Beschreibung des HTML-Dekorators erfolgt in Abschnitt 4.5.2.

- **Anwendungslogik.** In der Logikschicht, dienen die Daten-Container der Aufnahme von Zuständen der dort vorhandenen Geschäftsobjekte, die z.B. als EJBs oder Java-Objekte realisiert sind. Die ADCs können dabei in einer dynamischen Implementierung direkt mit diesen Objekten zusammenarbeiten und jegliche andere Transportobjekte einsparen. Damit existieren maximal zwei Typen von Transportobjekten und zwei Transportklassen im Anwendungssystem, die zentral bereitgestellt werden können. Wird auf diesen Effekt verzichtet, kann bei eingeschränkter Dynamik und Flexibilität der Konzepte eine Implementierung mit *Value Objects* gewählt werden. Dabei existiert eine von den Anwendungsfällen abhängige Anzahl von Transportobjekten und Transportklassen. Die dynamische Implementierung der Konzepte wird in Abschnitt 4.3 beschrieben. Die auf *Value Objects* basierende Implementierung befindet sich in Abschnitt 4.4. Die Integration der Daten-Container in EJBs und damit verbundene Architekturen, wird in Kapitel 5 beschrieben.

- **Integration/Ressourcen.** Die Daten-Container standardisieren den Zugriff und Transport auf Daten. Durch ihr aktives Verhalten können sie mit Datenzugriffsobjekten (vgl. dazu auch *Data Access Object* in [Dee01]) kommunizieren, die Informationen mit verschiedenen Datenquellen, wie z.B. Datenbanksystemen oder Fremdsystemen in Form von Legacy-Systemen, austauschen. Ein konkretes Beispiel hierfür ist der generische Zugriff auf `ResultSet`-Objekte, die Ergebnisse einer direkten SQL-Anfrage mittels JDBC an

[1] *Graphical User Interface (GUI),* grafische Benutzeroberfläche einer Anwendung
[2] *Hypertext Markup Language (HTML),* Seitenbeschreibungssprache zur Darstellung von Internetseiten in Web-Browsern
[3] *Wireless Markup Language (WML),* Seitenbeschreibungssprache für Web-Browser, die sich in Handys befinden
[4] *Extensible Markup Language (XML),* Metasprache zur Strukturierung von Dokumenten

eine relationale Datenbank liefern. Eine Umsetzung dieses Konzepts wird in Abschnitt 4.3.3 beschrieben.

Aufgrund der wohldefinierten Schnittstelle von ADCs finden Zugriffe immer auf die selbe Art und Weise statt, unabhängig davon, wo sie verwendet werden. Insbesondere ist die Handhabung für den Anwendungsentwickler immer gleich.

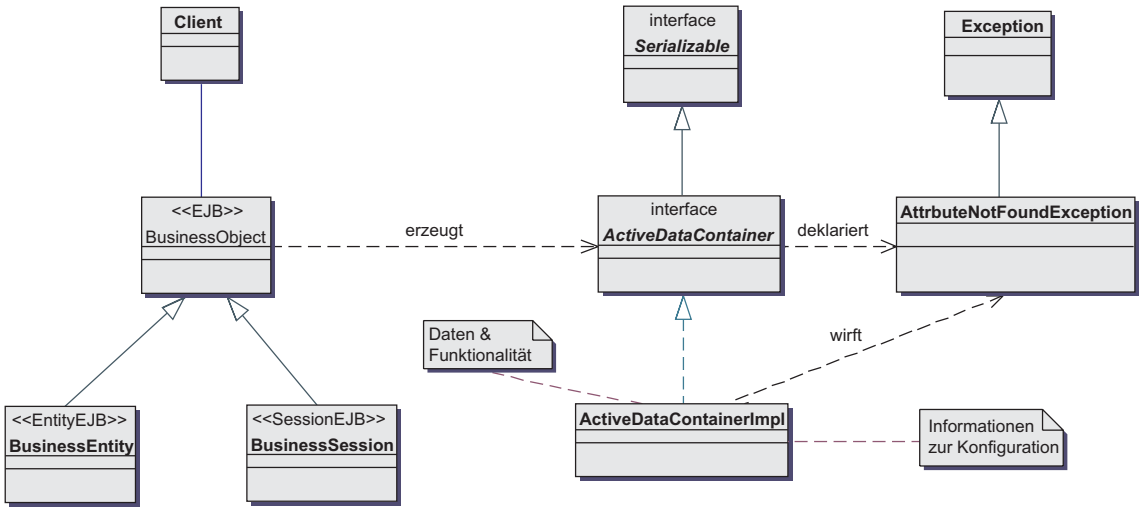


Abbildung 4.16: Klassendiagramm: Prinzip des Aktiven Daten-Containers

In Abbildung 4.16 ist die Grundstruktur des ADCs dargestellt. Im Zentrum des Konzepts steht die Schnittstelle `ActiveDataContainer`, die alle Methoden definiert, die ein Transportobjekt bereitstellen muß. Dabei sind die folgenden Arten von Methoden relevant:

- **Schreiben, Lesen, Löschen und Existenzprüfung von Attributen.** Serialisierbare Attribute müssen in den Container geschrieben, ausgelesen und auch gelöscht werden können. Mit der Existenzprüfung muß festgestellt werden können, ob bestimmte Attribute im Container enthalten sind. Zusätzlich ist eine Löschfunktion sinnvoll, die den gesamten Container-Inhalt löscht. Dies fördert die Wiederverwendung eines bestehenden Daten-Containers anstatt bei jedem Datenübertragungsvorgang ein neues Container-Objekt zu erzeugen.

- **Iteratoren für Attribute**. Mit Iteratoren können die im Container enthaltenen Attribute anhand ihrer Bezeichnung und ihrer Werte der Reihe nach durchlaufen werden. Dies kann z.B. von verallgemeinerten Anwendungsteilen, die unabhängig vom Container-Inhalt sind, genutzt werden.

- **Hinzufügen, Entnehmen und Synchronisieren von Objektzuständen.** Java-Objekte, deren Struktur dem Daten-Container bekannt sind, können ihm direkt zur automatischen Entnahme des Zustands übergeben werden. Umgekehrt kann der Daten-Container aus

den Attributen, die er transportiert, automatisch ein neues Java-Objekt erzeugen. Als weitere Möglichkeit können die Attribute eines vorhandenen Objekts mit den im Container vorhandenen Attributen synchronisiert werden.

- **Konfiguration und Ausführung von Funktionalität.** Falls der Anwendungsentwickler Einfluß auf die Eigenschaften des Daten-Containers besitzen soll und selbst entscheidet, ob bestimmte Funktionalitäten eingesetzt werden, müssen dafür entsprechende Methoden bereitgestellt werden.

Die Schnittstelle ist von `Serializable` abgeleitet, um implementierende Objekte zwischen Client und Server übertragen zu können. Zu dem Daten-Container-Konzept gehört auch die Exception `AttributeNotFoundException`, die von allen Lese-Methoden der Schnittstelle `ActiveDataContainer` deklariert wird, die zur Entnahme von Daten aus dem Container verwendet werden. Diese Ausnahme soll immer dann von der Implementierung `ActiveContainerImpl` geworfen werden, wenn ein angefordertes Attribut nicht im Daten-Container enthalten ist, um damit diese mögliche Fehlerquelle aufzudecken. Die Klasse `ActiveContainerImpl` stellt die Implementierung des Daten-Containers bereit, d.h. eine geeignete und ggf. optimierte Datenspeicherung und Zusatzfunktionalität, die auf die Daten im Container angewendet werden kann oder das Verhalten des Containers selbst bestimmt. Um die Universalität und Flexibilität des Containers sicherzustellen, sind für das Konzept die *Informationen zur Konfiguration* wichtig, die zur Übersetzungs- oder zur Laufzeit bestimmen, welche Funktionalität verwendet wird und wie sich das Verhalten des Containers darstellt. Diese Informationen sind entweder im Programmcode festgelegt, im Deployment-Deskriptor einer EJB abgelegt oder zentral in einem JNDI-Objekt enthalten. Abbildung 4.17 zeigt die grundsätzlichen Interaktionen der beteiligten Objekte und Komponenten. Wichtig dabei ist die unterschiedliche Verwendung des Containers auf Client- und Server-Seite. Während der Client hauptsächlich die `get()`- und `set()`–Methoden zum Lesen und Schreiben von Daten verwendet (2, 3), benutzt eine EJB vorrangig Methoden zur Übergabe oder Entnahme bzw. Synchronisation eines Objekts, dessen Daten zwischen Client und Server transportiert werden. Falls notwendig, kann der unterschiedlichen Verwendung auf Client und Server z.B. dadurch Rechnung getragen werden, daß zwei getrennte Schnittstellen für Client und Server bereitgestellt werden und vor oder während des Versands eine Typumwandlung in die Client-Schnittstelle erfolgt. Die Anwendung von Funktionalität kann dabei grundsätzlich bei jeder Interaktion mit dem Daten-Container erfolgen (1.1.1, 1.2.1, 2.1, 3.1 und 4.1.1). Dabei wird Funktionalität angewendet, die aufgrund der Datenübertragung notwendig ist. Es kann sich dabei z.B. um das Komprimieren oder Dekomprimieren von Daten handeln. Eine grundsätzliche Anwendung von Funktionalität kann auch global im Rahmen der Serialisierung/Deserialisierung stattfinden. Auf die Darstellung dieser Vorgänge wurde aus Übersichtlichkeitsgründen an dieser Stelle verzichtet. Dies wird in Kapitel 5, Abschnitt 5.2 nochmals aufgegriffen. In den folgenden Abschnitten werden Implementierungen des ADCs und ihrer Anwendung beschrieben.

4.3.3 Implementierungsstrategien

In diesem Abschnitt wird beschrieben, wie der ADC implementiert werden kann. Dabei wird die grundsätzliche Struktur aufgezeigt und die einzelnen Elemente werden beschrieben. Der

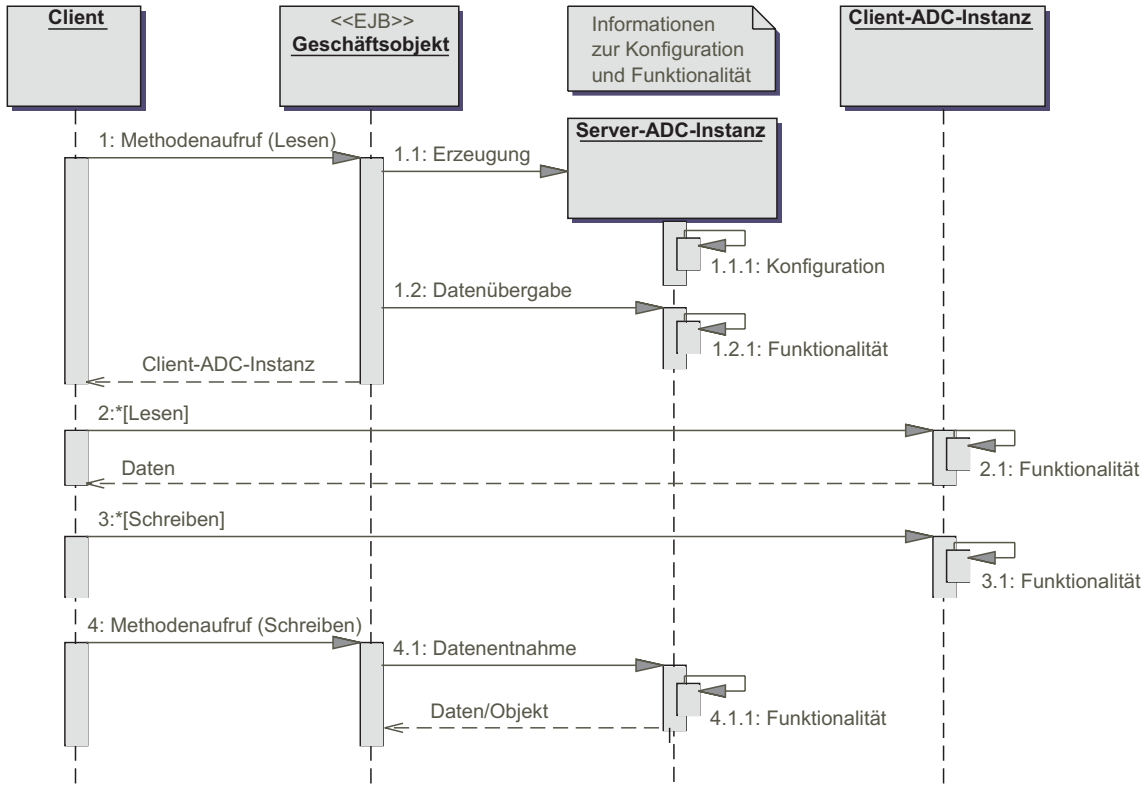


Abbildung 4.17: Sequenzdiagramm: Prinzip des Aktiven Daten-Containers

grundlegende Aufbau des ADC ist in Abbildung 4.18 schematisch dargestellt.

In der Praxis kann es sinnvoll sein, die Übertragung der Daten eines Objekts von der Datenübertragung vieler Objekte (Massendaten) zu unterscheiden. Dabei können zwei Typen von ADCs implementiert werden. Ein ADC-Typ transportiert dabei immer nur ausschließlich die Daten *eines* Objekts und kann daher zusätzlich immer als Stellvertreter (*Proxy*) [Gam95] für das betreffende Objekt angesehen werden. Dieser Umstand ist vor allem für Geschäftsobjekte interessant, die sich in der Applikationsschicht befinden und nicht auf Seite des Clients verwendet werden. Handelt es sich bei dem betreffenden Objekt um eine EJB, so kann der ADC durch den automatisierten Transport der EJB-Referenz auf Wunsch auch zusätzlich als intelligenter Stellvertreter (*Smart Proxy*) (vgl. Kapitel 3, Abschnitt 3.2.3) fungieren. Unabhängig davon wird ein zweiter ADC-Typ so konzipiert, daß er ausschließlich die Daten vieler Objekte aufnimmt. Dies hat den Vorteil, daß anhand des Programmcodes leichter unterschieden werden kann, ob die Daten eines Objekts oder die Daten vieler Objekte verarbeitet werden. Dieser Aspekt wird in Kapitel 5 ausführlicher erläutert. Im folgenden wird jedoch auf die Unterscheidung von zwei ADC-Typen verzichtet, da es sich um eine Implementierungsalternative handelt, die unabhängig von den hier dargestellten Konzepten vorgenommen werden kann.

Nachfolgend werden die einzelnen Elemente, ihr Zusammenspiel, ihr Verwendungszweck und Möglichkeiten ihrer Implementierung detailliert erläutert:

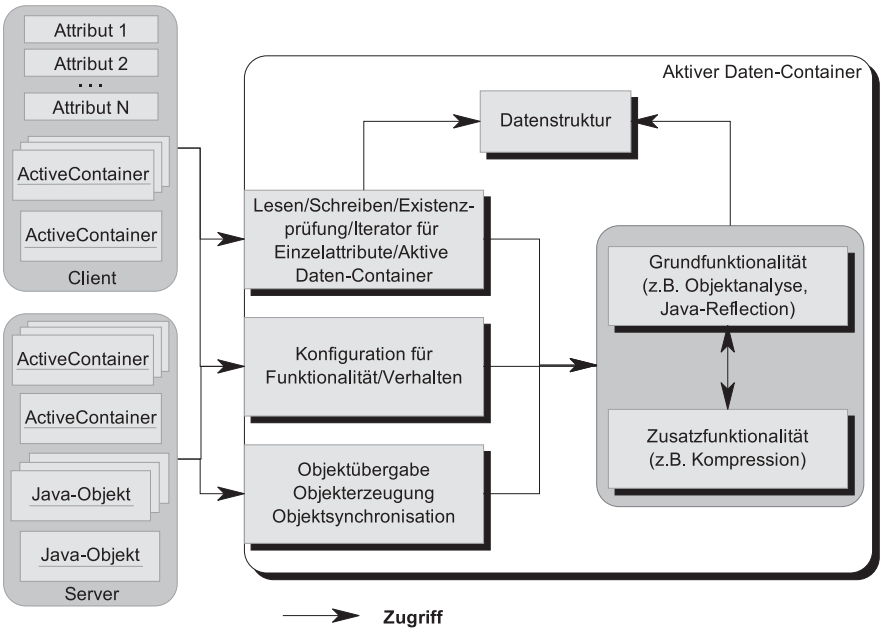


Abbildung 4.18: Schema des Aktiven Daten-Containers

Objektübergabe, -erzeugung und -synchronisation

Es handelt sich dabei um Methoden und Konstruktoren, die als Parameter Java-Objekte besitzen. Die Objektübergabe dient dem Hinzufügen von Objekten als Datenlieferant, d.h. sein Zustand oder die Daten, die es mittels seiner Schnittstelle liefert, werden in der internen Datenstruktur gespeichert. Grundsätzlich kann es sich dabei um beliebige Java-Objekte handeln, deren Struktur bzw. Schnittstelle dem Daten-Container bekannt ist. Wichtige Beispiele sind Datenzugriffsobjekte, deren Methoden genutzt werden, um Daten abzurufen und Geschäftsobjekte, deren Zustand (Attribute) ausgelesen und in der dafür vorgesehenen Datenstruktur abgespeichert wird. Durch Angabe einer Liste von gewünschten Attributnamen kann nur auf benötigte Attribute zugegriffen werden, um eine Reduktion der Datenmenge zu erreichen. Damit wird das Entwurfsmuster *Dynamische Attribute* umgesetzt, das ursprünglich aus der CORBA-Welt stammt und Attribute zur Laufzeit anhand ihres Namens überträgt [Mow97]. Dieses Muster wird dabei zum festen Bestandteil des Daten-Containers, der von jedem Entwickler genutzt werden kann, ohne einen Implementierungsaufwand zu erfordern. Methoden zur Objektsynchronisation dienen dazu, die im Container transportierten Attribute mit den Attributen eines Geschäftsobjekts abzugleichen. D.h. nachdem die Attribute auf dem Client im Daten-Container geändert wurden, können die Änderungen in einem auf dem Server existierenden Geschäftsobjekt übernommen werden. Mittels Methoden zur Objekterzeugung kann ein neues Geschäftsobjekt erzeugt werden, dessen Attributwerte den im Daten-Container transportierten Attributwerten entsprechen. In Abbildung 4.18 wird davon ausgegangen, daß diese Methoden nur auf dem Server verwendet werden, um zu gewährleisten, daß Clients nicht mit Objekten der Logik- oder Datenschicht arbeiten. Dies entspricht der Festlegung eines Benutzungskonzepts für Daten-Container. Falls erforderlich, kann der Zugriff auf solche Methoden auch im Client erfolgen.

Dies muß im Benutzungskonzept entsprechend verankert werden. Für die Implementierung der Methoden müssen die folgenden Aspekte in Betracht gezogen werden:

- **Streng typisierte Methoden:** Für jedes Objekt, das der Daten-Container verarbeiten kann, wird eine Methode bereitgestellt, die genau diesen Typ von Objekt entgegennimmt. Der Implementierungsaufwand ist dabei um so höher, je mehr Objekte übergeben werden können. Zusätzlich ist die Flexibilität des Containers eingeschränkt, da zur Laufzeit bereits festliegt, was übertragen werden kann. Nachfolgend ist ein Beispiel der Methoden für ein Objekt vom Typ `Fachobjekt` dargestellt:

```
// Objektuebergabe (Alle Attribute werden
// automatisch entnommen)
public void setFachObjekt(Fachobjekt obj)

// Objektuebergabe (Spezifizierte Attribute
// werden automatisch entnommen)
public void setFachObjekt(Fachobjekt obj, String[] attribute)

// Erzeuge neues Fachobjekt, das Attributwerte
// des Containers enthaelt
public Fachobjekt getFachObjekt()

// Synchronisiere Attribute des uebergebenen Objekts
// mit den im Container enthaltenen
public void syncFachObjekt(Fachobjekt fachObjekt)

// Objekt wird im Container gespeichert
public void putDatenObjekt(Datenobjekt datenObjekt)

// Objekt wird dem Container entnommen
public Datenobjekt getDatenObjekt()
```

- **Schwach typisierte Methode:** Es existieren nur wenige generische Methoden, die zur Übergabe von Objekten verwendet werden. Methoden, die Objekte zur Entnahme ihres Zustands entgegennehmen, besitzen dabei einen Parameter vom Java-Typ `Object`, der die Basis für alle Java-Objekte darstellt und somit die Übergabe jedes beliebigen Objekts erlaubt:

```
// Objektuebergabe (Alle Attribute werden
// automatisch entnommen)
public void setObject(Object object)

// Objektuebergabe (Spezifizierte Attribute
// werden automatisch entnommen)
public void setObject(Object obj, String[] attribute)

// Erzeuge neues Fachobjekt, das Attributwerte
```

```
// des Containers enthaelt
public Object getObject()

// Synchronisiere Attribute des uebergebenen Objekts
// mit den im Container enthaltenen
public void syncObject(Object object)
```

Methoden, die Objekte entgegennehmen, deren Zustand insgesamt übertragen werden soll, besitzen einen Parameter vom Typ `Serializable`, der die Übergabe von beliebigen serialisierbaren Objekten, die über das Netzwerk transportiert werden können, erlaubt.

```
// Objekt wird im Container gespeichert
public void putSerObject(Serializable object)

// Objekt dem Container entnehmen
public Serializable getSerObject()
```

Der Implementierungsaufwand ist dabei am geringsten und die Flexibilität am größten. Nach der Entnahme von Objekten muß hier eine Umwandlung in den ursprünglichen Typ durchgeführt werden. Eine potentielle Einschränkung dahingehend, welche Objekte vom Daten-Container akzeptiert werden, muß hier durch ein Benutzungskonzept abgedeckt werden, daß auf Wunsch auch zur Laufzeit im Rahmen der Zusatzfunktionalität kontrolliert werden kann. Dabei können im Daten-Container zur Laufzeit Prüfungen erfolgen, die eine Entgegennahme von Objekten ablehnen, die nicht im Container transportiert werden dürfen.

Zusätzlich zu den Methoden, die nur ein einzelnes Objekt übernehmen, können auch Folgen von Objekten übergeben werden, deren Daten im Container abgelegt werden. Im Gegensatz zur Übergabe eines einzelnen Objekts muß den Methoden nun zusätzlich eine Kennung übergeben werden, unter der die Daten von Objekten im Daten-Container gespeichert werden. Dies ermöglicht die strukturierte Speicherung der Daten vieler Folgen unterschiedlicher Objekte. Die Übergabe von Parametern kann wiederum schwach oder streng typisiert erfolgen. Um den Implementierungsumfang zu verringern und die Universalität des Daten-Containers zu gewährleisten, empfiehlt sich eine generische Methode, die eine Folge von Objekten in einem Objekt vom Typ `Collection` entgegennimmt. Dies entspricht auch der Rückgabe von Persistenz-Frameworks, die eine Treffermenge z.B. in Form eines `Vector`-Objekts zurückliefern, das die Schnittstelle `Collection` zur Verarbeitung der verschiedenen Objekte unterstützt. Nachfolgend ist ein möglicher Aufbau von generischen Methoden dargestellt:

```
// Fuege alle Attribute eines Objekts
// unter 'kennung' ein
public void addObject( String kennung,
                       Object object )

// Fuege spezifizierte Attribute eines Objekts
```

```
// unter 'kennung' ein
public void addObject(String kennung,
                      Object object,
                      String[] attribute )

// Fuege alle Attribute aller in dem Collection-
// Objekt enthaltenen Objekte unter 'kennung' hinzu
public void addObjects( String kennung,
                        Collection col )

// Fuege spezifizierte Attribute aller in dem
// Collection-Objekt enthaltener Objekte unter
// 'kennung' hinzu
public void addObjects( String kennung,
                        Collection col,
                        String[] attribute )

// Erzeuge eine Folge von Objekten, deren
// Attribute unter 'kennung' abgelegt sind
public Collection getObjects(String kennung)

// Synchronisiere alle in 'col' enthaltenen
// Objekte mit den Daten im Container
public void syncObjects( String kennung,
                         Collection col )
```

In der Realität müssen i.d.R. statische und dynamische Parameter verwendet werden, da z.B. eine hohe Anzahl von Geschäftsobjekten existiert, die mittels generischer Methoden abgedeckt werden. Sonderfälle werden in Form von speziellen Methoden realisiert. Die Implementierung dieser Methoden greift auf die im Daten-Container implementierte Funktionalität und Zusatzfunktionalität zu, die separat beschrieben wird.

Konfiguration für Funktionalität/Verhalten

Diese Methoden erlauben die Anpassung des Daten-Containers an bestimmte Systemanforderungen, die entweder bereits im Vorfeld eines Projekts bekannt sind, oder während der Implementierung des Anwendungssystems oder während des Betriebs einer Anwendung bekannt werden. Die Konfiguration kann dabei dynamisch zur Laufzeit oder statisch im Quellcode angegeben werden. Die dynamische Konfiguration kann vom jeweiligen Benutzer des Daten-Containers angewendet werden, um frei zu entscheiden, wann eine Funktionalität eingesetzt werden soll. Dies kann durch den Aufruf einer oder mehrerer dafür vorgesehener Methoden geschehen. Hierbei kann wieder ein generischer oder statischer Ansatz gewählt werden:

```
// Generisch
public void useFunction(String functionName)
public void useFunctions(String[] functionNames)
```

```
// Statisch
public void zip()
public void unzip()

public void crypt()
public void decrypt()
...
```

Die statische Konfiguration durch statische Konstanten auf Quellcodeebene kann vom Entwickler des Daten-Containers verwendet werden, um bestimmte Funktionalitäten transparent und systemweit einzuschalten:

```
public static final boolean zip=true;
public static final boolean crypt=false;
...
```

Beispiele für Zusatzfunktionalitäten sind die Kompression von Daten und das transparente „Durchschreiben“ von Attributen auf den Server bei clientseitigem Schreibzugriff auf den Daten-Container. Die unterschiedlichen Konfigurationsarten und -methoden haben unterschiedliche Folgen auf die Implementierung der Zusatzfunktionalität. Dies wird weiter unten ausführlich erläutert.

Lesen/Schreiben/Existenzprüfung/Iteratoren für Einzelattribute/Aktive Daten-Container

Nach dem Empfang eines Daten-Containers kann der Client mittels `get-` und `set`-Methoden auf die übertragenen Einzelattribute zugreifen. Die Attributnamen werden dabei als Zeichenkette übergeben. Falls pro Daten-Container nur ein Geschäftsobjekt übertragen wird, kann der Daten-Container praktisch als Stellvertreter dieses Objekts verstanden werden, wobei die zugehörige Geschäftsklasse auf dem Server verbleibt und dem Client nicht bekanntgegeben wird. Die Verwendung dieser Methoden kann auch zum Lesen und Schreiben auf dem Server verwendet werden, falls dies erforderlich ist. Aufgrund der Universalität des Daten-Containers handelt es sich hier um dynamische Datenspeicherung. Damit kann das Lesen und Schreiben mittels zweier generischer `get-` und `set`-Methoden der folgenden Form realisiert werden:

```
// Schreibe ein Schluessel/Wert-Paar, Schluessel
// kann ein beliebiges Objekt sein
public void set(Object key, Object value)

// Schreibe ein Schluessel/Wert-Paar, Schluessel
// kann nur ein Zeichenkette sein
public void set(String key, Object value)

// Lese einen Wert
public Object get(Object key)
```

Falls erforderlich, muß nach dem Auslesen eines Werts eine Typumwandlung erfolgen.
Alternativ können für alle möglichen Datentypen entsprechende `getXXX`- und `setXXX`-Methoden geschrieben werden, um die Typumwandlung zu verhindern und die Lesbarkeit des Quellcodes zu verbessern.

```
// Schreibe Ganzzahl int
public void setInt(String key, int intValue)

// Lese Ganzzahl int
public int getInt(String key)

// Schreibe Ganzzahlobjekt Integer
public void setInteger(String key, int intValue)

// Lese Ganzzahlobjekt Integer
public Integer getInteger(String key)


// Schreibe String
public void setString(String key, String value)

// Lese String
public String getString(String key)
...
```

Als weitere Besonderheit können zum Auslesen auch `getAsXXX`-Methoden implementiert werden, die eine Konvertierung der transportierten Daten in den jeweils vom Client geforderten Datentyp vornehmen. Dies sorgt für eine Entkopplung zwischen den auf dem Server und Client verwendeten Datentypen. Eine Entkopplung mit dieser Flexibilität ist bei bestehenden Datenübertragungskonzepten nicht vorgesehen und nur mit einem wesentlich erhöhten Implementierungsaufwand zu leisten. So müssen bei der Verwendung von *Value Objects* oder *Geschäftsobjekten* alle Objekte mit entsprechenden `getAsXXX`-Methoden ausgestattet werden. Bei der Verwendung von Collection-Klassen, wie z.B. `HashMap`, muß eine manuelle Konvertierung durch den Anwendungsentwickler beim Auslesen erfolgen. Der ADC kann diese Konvertierungsroutinen einmalig im Rahmen von verallgemeinerter Zusatzfunktionalität bereitstellen.

```
// Lese Wert als Integer
public Integer getAsInteger(String key)

// Lese Wert als String
public String getAsString(String key)
```

Zum Setzen eines Attributs mit `getAsXXX` sollte wiederum die generische `set`-Methode verwendet werden. Vor dem Überschreiben des aktuellen Attributwertes muß dann eine Rückkonvertierung vorgenommen werden, um zu verhindern, daß es auf dem Server zu Typkonflikten kommt, wenn die Attribute z.B. mit einem Geschäftsobjekt synchronisiert werden.
Zusätzlich ist das Durchlaufen aller Attribute und deren Werte sinnvoll, um allgemeine, vom Inhalt des Daten-Containers unabhängige Routinen zu implementieren. Beispiel für solche Methoden sind:

```
// Hole ein Objekt das die Iterator-Schnittstelle zum Durchlaufen
// aller Attributnamen implementiert
```

```
public Iterator attributeNames()

// Hole ein Objekt das die Iterator-Schnittstelle zum Durchlaufen
// aller Attributwerte implementiert
public Iterator attributeValues()

// Hole alle Attributnamen
public String[] getAttributnames()

// Hole alle Attributwerte
public Object[] getAttributvalues()
```

Auf dem Client können ADC-Objekte als Stellvertreter für Server-Objekte verwendet, manipuliert oder neu erzeugt werden und in den Daten-Container geschrieben werden. Auf dem Server können ebenso vom Client manipulierte ADC-Objekte entnommen werden und z.B. neue Geschäftsobjekte aus den im Container transportierten Attributen erzeugt werden oder vorhandene mit den Attributen synchronisiert werden. Durch Iterator-Methoden wird es möglich, alle ADC-Objekte, die unter einer bestimmten Kennung im Daten-Container gespeichert sind, zu durchlaufen. Zur Verallgemeinerung des Auslesens wird auch eine Iterator-Methode benötigt, die es erlaubt alle Kennungen zu durchlaufen, unter der Daten im Container abgelegt wurden. Nachfolgend sind Vorschläge für solche Methoden dargestellt. Jede Methode besitzt als Parameter eine Kennung, um zu entscheiden, auf welchen Daten der Daten-Container die Operation ausführen soll:

```
// Iterator-Objekt auf alle Kennungen
public Iterator identifiers()

// Iterator-Objekt auf alle Elemente von 'kennung'
public Iterator elements( String kennung )

// Hole alle Kennungen
public String[] getIdentifiers()

// Wieviele Elemente sind unter 'kennung' abgelegt?
public int size(String kennung)

// Hole Element an der Position 'pos' von 'kennung'
public ActiveContainer elementAt( String kennung,
                                  int pos )

// Hole alle unter einer Kennung gespeicherten Elemente
public ActiveContainer[] getElements( String kennung )

// Fuege Element unter 'kennung' hinzu
public void addDataContainer( String kennung,
                              ActiveContainer ac );
```

```
// Loesche Element an Position 'pos' unter
// Kennung 'kennung'
public void removeElementAt( String kennung,
                             int pos )
...
```

Bei der Übertragung von Objektfolgen kann auch auf die Repräsentation jedes einzelnen Objekts als ADC (der in einem ADC als Sammelbehälter geschachtelt ist) verzichtet werden. Statt dessen können die Attribute jedes einzelnen Objektzustands an der Schnittstelle z.B. als `HashMap` oder sonstiges Objekt, das zur Attributentnahme z.B. `getAsXXX`-Methoden anbietet, repräsentiert werden.

Datenstruktur

Die Datenstruktur innerhalb des Daten-Containers, die zur Speicherung des Zustands von übergebenen Objekten bzw. übergebenen Transportdaten herangezogen wird, richtet sich nach deren Beschaffenheit und muß je nach Präferenzen z.B. auf Platzbedarf und Zugriffsgeschwindigkeit optimiert werden. Im Rahmen dieser Arbeit wird als grundlegende, interne Speicherstruktur die Klasse `HashMap`, die in der Java-Bibliothek enthalten ist und ein Hash-Verfahren implementiert, bei dem die Adressen von Datensätzen aus den zugehörigen Schlüsseln berechnet werden, verwendet. Die Attributnamen des übergebenen Objekts werden dabei als Schlüssel verwendet und die Attributausprägungen als Wert. Gegenüber der vergleichbaren Datenstruktur `Hashtable` hat `HashMap` den Vorteil, daß die Attributausprägung `null` direkt gespeichert werden kann und nicht künstlich nachgeahmt werden muß. Der Wert `null` hat häufig eine fachliche Bedeutung und muß deshalb zwischen Client und Server übertragen werden. Zusätzlich ist ein Vorteil im Leistungsverhalten beim Zugriff vorhanden, da die Methoden der Klasse `HashMap` nicht synchronisiert sind. D.h. sie sind nicht gegen den Zugriff mehrerer Threads gesichert. Die Synchronisation beeinflußt das Leistungsverhalten eines Java-Programms negativ [Sin97]. Falls die Synchronisation benötigt wird, kann eine ebenfalls verfügbare synchronisierte Form von `HashMap` verwendet werden.
Nachfolgend wird darauf eingegangen, wie der Transport von Objektfolgen und Massendaten realisiert werden kann. Die Tatsache, daß der Daten-Container an seiner (Client-)Schnittstelle mit ADC-Objekten arbeitet, erfordert nicht unbedingt die Speicherung der Daten in genau dieser Form. Somit kann auch eine Optimierung der Datenmenge und Komplexität erfolgen, die jedoch auf Kosten der Flexibilität des Daten-Containers erfolgt. Die flexibelste Implementierung des Konzepts besteht darin, die ADC-Objekte direkt zu speichern. Dieses Konzept ist in Abbildung 4.19 dargestellt. Die Implementierung speichert dabei immer intern ADC-Objekte in einem `HashMap`-Objekt. Als Schlüssel wird die vom Anwendungsentwickler vergebene Kennung verwendet. Selbst bei direkter Übergabe von Objekten, deren Attribute transportiert werden sollen, werden diese jeweils in einem ADC-Objekt abgelegt und in dieser Form gespeichert. Weil zur Übersetzungszeit die Anzahl der übergebenen Objekte, deren Daten transportiert werden sollen, nicht feststeht, wird zur Speicherung ein Objekt vom Typ `Vector` verwendet, das dynamisch wachsen kann.
Diese Form des Daten-Containers erfordert einen geringen Implementierungsaufwand und bietet dabei eine sehr hohe Flexibilität, da die ADC-Objekte unterschiedlich konfiguriert und ggf.

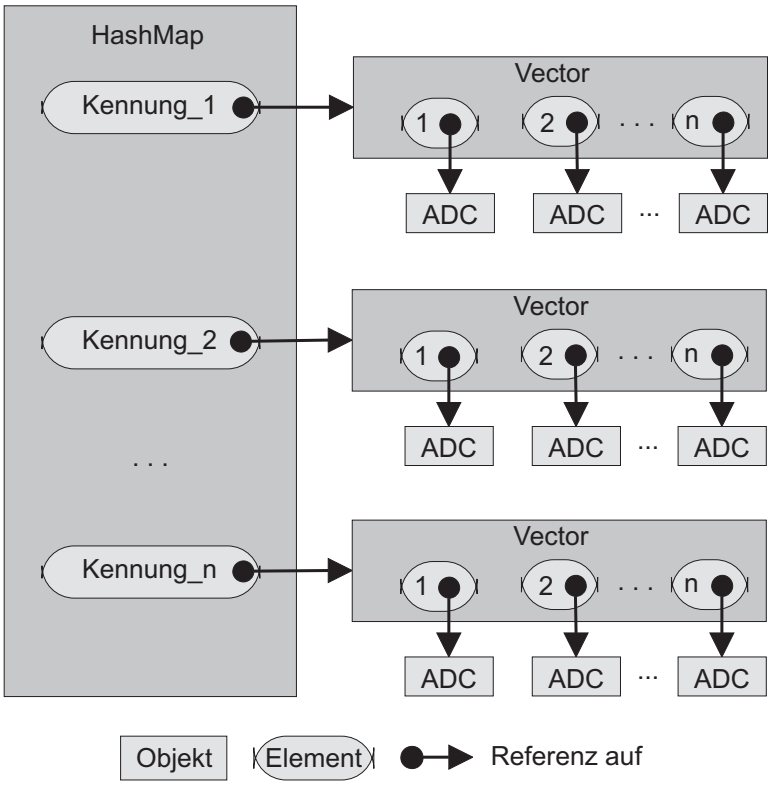


Abbildung 4.19: ADC-Datenspeicherung 1

auch unterschiedlich implementiert sein können. So können z.B. einige ADCs ihren Inhalt komprimieren, während andere einen transparenten Schreibzugriff für bestimmte Attribute, die sie transportieren, durchführen. Diese Flexibilität muß jedoch mit einer hohen Datenmenge, die zur Laufzeit anfällt, bezahlt werden. Die höhere Datenmenge fällt zum Teil aufgrund der entstehenden komplexen Datenstruktur an, die aus einem `HashMap`-Objekt besteht, das für jede Kennung ein `Vector`-Objekt enthält, worin wiederum viele ADC-Objekte enthalten sind, die ihrerseits ein `HashMap`-Objekt enthalten. Das Hauptproblem entsteht jedoch bei der Übertragung von Massendaten, da die Namen von Attributen und der Typ des Objekts redundant in jedem ADC-Objekt gespeichert sind. Eine Reduktion der Datenmenge und der Komplexität des Übertragungsformats kann dadurch erreicht werden, daß die Speicherung der Daten als ADC-Objekte aufgehoben wird und im wesentlichen nur noch die tatsächlichen Nutzdaten möglichst effizient gespeichert werden. Dieser Umstand ist in Abbildung 4.20 dargestellt. Zur späteren Rekonstruktion des eigentlichen Objekts wird dessen Typbezeichnung nur einmal abgespeichert. Die Redundanz der Attributnamen wird verhindert, indem sie für einen Objekttyp nur noch einmal in einem String-Array abgespeichert werden. Die Attributwerte selbst werden in einem `Object`-Array an der zugehörigen Indexposition ihres Attributnamens gespeichert, um sie später wieder zuordnen zu können. Falls Daten durch den Anwendungsentwickler abgerufen werden, wird aus diesen Daten ein ADC-Objekt erzeugt und gefüllt.

Eine weitere Reduktion der Datenmenge und Komplexität kann durch die in Abbildung 4.21 dargestellten Maßnahmen erreicht werden. Dabei werden die Daten in eine String-Repräsentati-

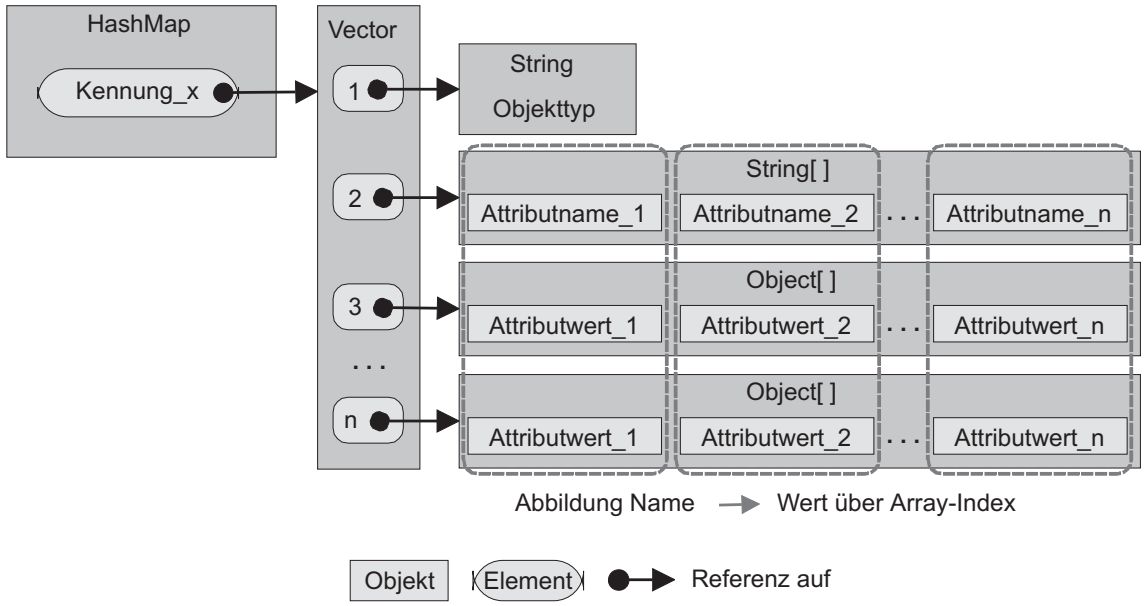


Abbildung 4.20: ADC-Datenspeicherung 2

on überführt und anhand der Indexposition des Attributnamens und des Datentyps rekonstruiert. Diese Form schränkt die Flexibilität des Containers allerdings ein, da nicht mehr automatisch beliebige Objekttypen unterstützt werden. Für jeden Objekttyp eines Attributs, der im Container gespeichert werden soll, muß eine String-Repräsentation möglich sein und eine Routine erstellt werden, die das Objekt aus dem erstellten String rekonstruiert. Bei primitiven Java-Datentypen sind diese Funktionen aber bereits vorhanden. Diese Vorgehensweise hat zwar einen höheren Implementierungsaufwand, kann allerdings die Datenmenge reduzieren, vor allem dann, wenn die String-Repräsentation eines Datentyps kleiner ist, als der Datentyp selbst.

Eine ähnliche Strategie zur Verkleinerung der Datenmenge besteht darin, die übertragenen Daten so zu kodieren, daß sie nur die wirklich benötigte Datenmenge in Anspruch nehmen. Ein Beispiel hierfür ist die Verwendung des Datentyps `int`, der in Java mit 4 Bytes repräsentiert wird. Unabhängig vom tatsächlichen Wert der Zahl, die als Integer repräsentiert wird, müssen diese 4 Bytes übertragen werden. Häufig werden jedoch kleinere Zahlen transportiert, die eigentlich nicht die volle Bitbreite benötigen. Eine Strategie im Daten-Container kann daher darin bestehen, Zahlen nur mit der tatsächlich benötigten Bitbreite zu repräsentieren und dadurch die Datenmenge zu reduzieren. Dies kann bei der Übertragung von Massendaten zu einer signifikanten Verkleinerung der Datenmenge führen. Falls erforderlich, kann auf die insgesamt zu übertragende Datenmenge zusätzlich ein universeller Kompressionsalgorithmus angewendet werden (siehe unten, Zusatzfunktionalität).

Die in dieser Arbeit vorgestellten Maßnahmen zur Veringerung der Datenmenge und -komplexität sind als Beispiele zur Illustration des vorgestellten Datenübertragungskonzepts zu verstehen. Um ein optimales Ergebnis zu erzielen, müssen in diese Überlegungen die Datenstrukturen und die Beschaffenheit der zu übertragenden Daten innerhalb einer Anwendung, in der das Konzept zum Einsatz kommen soll, mit einbezogen werden.

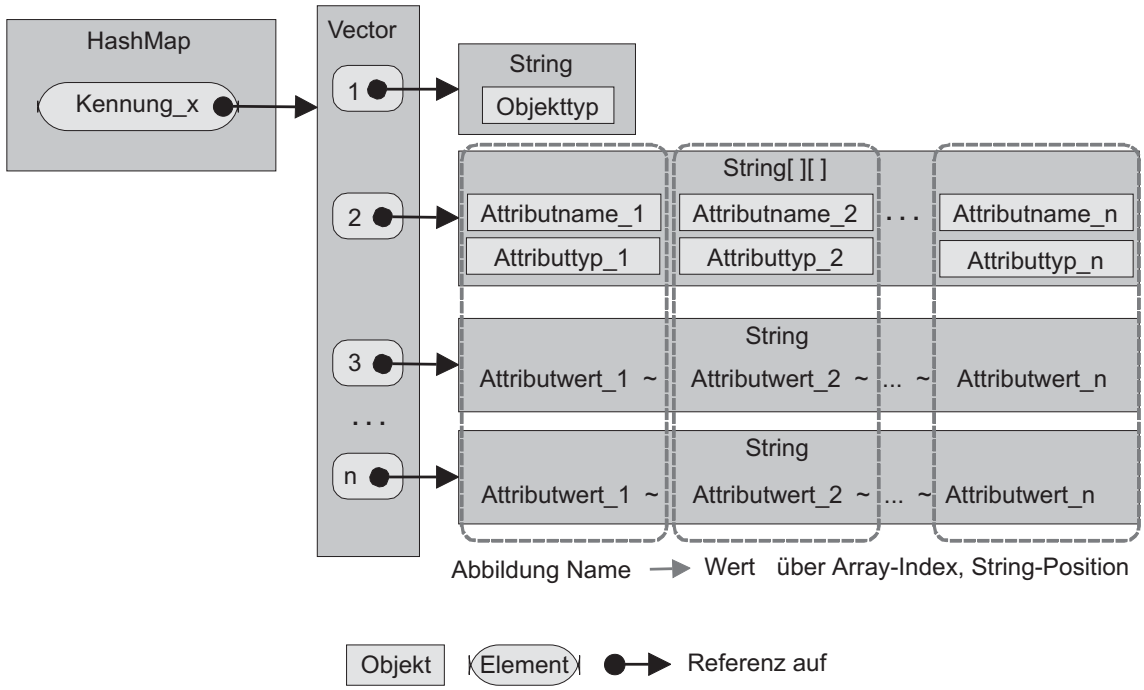


Abbildung 4.21: ADC-Datenspeicherung 3

Die Optimierung der Datenstruktur zur Verkleinerung der Datenmenge kann zu einem höheren Zeitaufwand zur Verpackung von Daten im Container und zur anschließenden Entnahme aus dem Container führen. Falls dieser Effekt zu groß wird, muß die Implementierung des Daten-Containers optimiert werden. Falls die Möglichkeiten der Optimierung erschöpft sind und das Ergebnis nicht als zufriedenstellend bewertet wird, kann das beste Ergebnis durch die Verwendung eines Generatoransatzes erzielt werden, der den Code von Objekten, deren Attribute transportiert werden sollen, erweitert. Die Erweiterung wird dabei so gestaltet, daß die Herstellung des effizienteren Formats zur Laufzeit entfällt. Das Format wird generiert und durch eine Methode zum Abruf bereitgestellt. Der Generatoransatz wird nachfolgend der Analyse von Objekten zur Laufzeit gegenübergestellt.

Grundfunktionalität

Um möglichst viele Objekte, deren Attribute im ADC transportiert werden, universell abzudecken, kann ein generischer Zugriff auf Attribute und Methoden mittels *Java-Reflection*[5] erfolgen. Die Implementierung der Datenentnahme mittels Java-Reflection kann auf zwei verschiedene Arten erfolgen. Auf die Attribute eines Objekts kann entweder direkt lesend und schreibend zugegriffen werden oder über zugehörige `get`- und `set`-Methoden. Am einfachsten läßt sich der Zugriff realisieren, wenn die Vereinbarung getroffen wird, daß die Attribute

[5]Es handelt sich dabei um einen Mechanismus, der es erlaubt Objekte und deren Klassen zur Laufzeit auf die verwendeten Attribute, Konstruktoren und Methoden hin zu analysieren sowie auf diese zuzugreifen. Java-Reflection wird z.B. ausführlich in [Mic01, McM97, McC98] beschrieben.

von Geschäftsobjekten öffentlich, d.h. `public` deklariert werden. Zugriffe im Programm erfolgen jedoch trotzdem über `get`- und `set`-Methoden, als ob die Attribute privat wären. Das folgende Codefragment skizziert den Auslesevorgang aller Attribute:

```
// Interne Datenstruktur des Containers
HashMap atts;
...
// Hole Klasse des Objekts
Class clazz=object.getClass();

// Hole alle Attribute des Objekts
Field attributes[]=clazz.getFields();

// Lese alle Attributwerte aus
for( int i=0 ; i < attributes.length ; i++)
{
  Object obj=attributes[ i ].get( object );

  // Schreibe Attributname und Wert
  atts.put( attributes[ i ].getName(), obj );
}
...
```

Falls eine Liste von Attributnamen übergeben wird, werden nur diese Attribute gezielt ausgelesen:

```
// Wird vom Benutzer des Containers uebergeben
String[] att={"name1", "name2", ..., "nameN"};

// Lese alle uebergebenen Attribute aus
// und speichere die Werte anhand ihres Namens
for(int i=0; i<att.length; i++)
{
   Field attribut=clazz.getField( att[i] );
   Object obj=attribut.get(object);
   atts.put(att[i], obj);
}
...
```

Bei der Verwendung von privaten Attributen muß alternativ die Methode `getDeclaredFields()` eingesetzt werden, um auf sie zuzugreifen. Die Verwendung dieser Methode muß allerdings häufig ausdrücklich in den Sicherheitseinstellungen von Java bzw. des Applikations-Servers erlaubt werden.
Die Verwendung von Methodenaufrufen ist zwar aus objektorientierter Sicht die bessere Alternative, erfordert allerdings aufgrund des Aufrufs von Methoden einen höheren Aufwand für die Verarbeitung von Attributen. Das nachfolgende Codefragment skizziert den lesenden Methodenzugriff. Dabei wird von der Konvention ausgegangen, daß die Methoden die Form

`get<Attributname>()` besitzen. Andere Methoden, wie z.B. die Standardmethode `getClass()`, die alle Java-Objekte besitzen und Methoden mit Parametern werden nicht aufgerufen.

```
// temporaeres Objekt zum Abspeichern eines Rueckgabeparameters
Object tmp=null;
String methodName=null;
Method method=null;
Class[] emptyParams={};

try
{
   Class c=o.getClass();
   ...
   // Hole alle Methoden der Klasse
   Method methods[]=c.getMethods();
   for(int i=0; i<methods.length; i++)
   {
      methodName=methods[i].getName();
      // Nur fachliche get-Methoden ohne Parameter holen
      if( methodName.startsWith("get")
          & !methodName.endsWith("Class")
          & !methodName.endsWith("Data")
          & methods[i].getParameterTypes().length==0)
      {
       // Methode aufrufen und Wert speichern
       tmp=methods[i].invoke(o, emptyParams);
       atts.put(methodName.substring(3), tmp);
      }
```

Falls eine Liste mit Attributnamen angegeben wird, werden die Methodennamen aus den Attributnamen erzeugt und aufgerufen:

```
for(int i=0; i<att.length; i++)
{
    method=c.getMethod("get"+att[i], null);
    tmp=method.invoke(o, emptyParams);
    atts.put(att[i], tmp);
}
```

Die Verwendung von Java-Reflection zum Auslesen des Zustands von Objekten hat den Vorteil, daß der Anwendungsentwickler keine manuelle Füllung vornehmen muß, sondern lediglich mit einem Aufruf das Objekt, dessen Attribute transportiert werden sollen, übergeben muß und somit sämtliche der sonst notwendigen `set`-Methoden eingespart werden können. Umgekehrt erfolgt der Vorgang, indem aus im Container enthaltenen Attributen ein neues Geschäftsobjekt (Java-Objekt) erzeugt und dessen Zustand mittels Java-Reflection wiederhergestellt wird. Das folgende Codefragment skizziert die Erzeugung eines neuen Objekts durch den Container:

```
String typinfo="klasse";
...
Class clazz=Class.forName(typinfo);
Object object=clazz.newInstance();
```

Anhand des Strings `typinfo` wird ein Objekt der gewünschten Klasse erzeugt (`forName()`) und anschließend ein neues Objekt dieser Klasse erzeugt. Der Inhalt des Strings `typinfo` kann z.B. bei der Übergabe eines Objekts, dessen Attribute zum Client geschickt werden sollen, ermittelt und im Container gespeichert werden:

```
Class clazz=object.getClass();
typeinfo=clazz.getName();
```

Dies hat den Nachteil, daß die Datenmenge vergrößert wird, die zwischen Client und Server hin und her geschickt wird, ist dafür aber sehr komfortabel. Als Alternative kann der Typ-String auch erst bei Anforderung eines neuen Objekts mit übergeben werden und muß damit auch nicht im Daten-Container gespeichert werden. Nachdem ein Objekt durch den Container erzeugt wurde, müssen noch die Attributwerte entsprechend den im Daten-Container transportierten Attributen gesetzt werden. Diese Operation entspricht der Funktion des Containers den Zustand eines bereits vorhandenen Geschäftsobjektes mit den Container-Attributen synchronisieren zu können. Das nachfolgende Codefragment wird somit bei der Erzeugung von neuen Objekten mittels der `getObject`-Methode und im Rahmen der `synchronize`-Methode ausgeführt:

```
// Attribut des zu synchronisierenden Objekts
Field f=null;

   ...
   // Durchlaufe alle im Container gespeicherten Attribute
   Iterator e=atts.keySet().iterator();
   Class c=o.getClass();
   while(e.hasNext())
   {
        // Hole im Container gespeicherte Attributnamen
        String s=(String) e.next();
        // Hole zugehoeriges Attribut des zu sync. Objekts
        f=c.getField(s);
        // Hole im Container gespeicherten Attributwert
        Object tmp=atts.get(s);
        // Setze Attributwert des zu sync. Objekts
        f.set(o, tmp);
      }
...
```

Dabei werden wiederum sämtliche `get`-Methoden eingespart, die sonst erforderlich wären, um alle transportierten Attribute aus dem Daten-Container zu entnehmen. Ebenso werden sämtliche `set`-Methoden des abzugleichenden Objekts eingespart. Trotz der hohen Ersparnis, die

eine Anwendung der Java-Reflection mit sich bringt, ist deren Einsatz nicht unproblematisch. Der Zugriff auf Objekte per Reflection erfordert einen höheren Zeitbedarf als der herkömmliche Aufruf von fest kodierten Zugriffsmethoden. Dieser durch Reflection entstehende Mehraufwand wirkt sich auf das Leistungsverhalten der gesamten Anwendung aus (vgl. dazu Kapitel 6). Darüber hinaus muß eine Konvention darüber erfolgen, wie die Benennung der ausgelesenen Attribute erfolgt. Es ist naheliegend, hierzu den Namen des Attributs (bei Direktzugriff) bzw. einen Teil des Methodennamens (bei indirektem Zugriff) zu verwenden. Hierbei entsteht aber eine Kopplung zwischen den persistenten Objekten des Servers und des Clients, die mittels Attributnamen auf die in dieser Schicht enthaltenen Daten zugreifen. Falls diese Form der Kopplung unerwünscht ist, muß dafür gesorgt werden, daß sich Attributnamen der Persistenzschicht und Attributnamen, die der Client verwendet, unterscheiden können. Die Unterscheidung kann dabei über den Namen hinausgehen und sich auch auf den Datentyp beziehen. So können z.B. auf Seite des Clients nur Strings verwendet werden, obwohl die Daten in den persistenten Objekten in unterschiedlicher Form vorliegen. Diese Entkopplung muß normalerweise durch den Anwendungsentwickler erfolgen und verursacht einen hohen Aufwand, da jedes Attribut manuell umbenannt werden muß und evtl. zusätzlich eine Typumwandlung erforderlich ist. Um diesen Vorgang ebenfalls zu automatisieren, kann eine Zuordnungsfunktion im Daten-Container implementiert werden, die weiter unten vorgestellt wird.
Falls auf die Verwendung von Reflection verzichtet werden soll, kann dennoch eine signifikante Einsparung von Entwicklungsaufwand erreicht werden, indem verhindert wird, daß jeder Anwendungsentwickler `get`- und `set`-Methoden implementiert, um Daten zu verpacken bzw. auszulesen. Hierzu werden Objekte, deren Daten in einem Daten-Container verschickt werden sollen, verpflichtet, eine Schnittstelle zu implementieren, die Daten-Container zum Auslesen und zum Beschreiben entgegennimmt. Der Anwendungsentwickler, der die Klasse des betreffenden Objekts entwickelt, implementiert dabei einmalig die erforderlichen `set`-Methoden, um die Attribute in den Container zu schreiben und die benötigten `get`-Methoden, um Attribute aus dem Container auszulesen und diese zurück in das Objekt zu schreiben. Dies hat den Vorteil, daß die Methoden nur einmal implementiert werden und für das erneute Versenden von Zuständen dieses Objekttyps wiederverwendet werden können. Als weitere Alternative kann der Anwendungsentwickler einmalig eine Methode implementieren, die es dem ADC erlaubt, die Daten in einer für ihn leicht zugänglichen Form abzufragen. Am Beispiel von Geschäftsobjekten sind diese Ansätze in dem nachfolgenden Codefragment skizziert:

```
public class Geschaeftsobjekt implements DataObject
{
    private int attribut_1;
    private String attribut_2;
    ...
    private float attribut_n;

 // Methode zum Fuellen von Daten-Containern
 public void setData(ActiveContainer ac)
 {
    ac.set("attribut_1", this.attribut_1);
    ac.set("attribut_2", this.attribut_2);
```

```
    ...
    ac.set("attribut_n", this.attribut_n);
 }

 // Methode zum Abrufen der Objektdaten
 public Object[ ] [ ] getData( )
 {
    Object [ ] [ ] attribute= new Object [ n ] [ 2 ];

    attribute[ 0 ]  [ 0 ] = "attribut_1";
    attribute[ 0 ]  [ 1 ] = new Integer ( this.attribut_1 );

    attribute[ 1 ]  [ 0 ] = "attribut_2";
    attribute[ 1 ]  [ 1 ] = this.attribut_2;
    ...
    attribute[ n ]  [ 0 ] = "attribut_3";
    attribute[ n ]  [ 1 ] = new Float ( this.attribut_n );

 }
...
}
```

Geschäftsobjekte können dabei normale Java-Objekte, aber auch EJBs sein. Die soeben vorgestellten Konzepte können um einen Generatoransatz erweitert werden, der die benötigten Methoden generiert. Hierzu wird das Interface `DataObject` während der Entwicklung als Markierungs-Interface ohne Methoden verwendet und fordert damit keine Implementierung durch den Entwickler. Durch einen Generatorlauf werden anschließend die Methoden zum Setzen und Lesen von Attributen generiert und die Schnittstelle `DataObject` wird durch die eigentliche Schnittstelle mit Methoden, wie sie in dem Codefragment skizziert sind, ersetzt. Durch diese Vorgehensweise ist es auch möglich, während der Entwicklung innerhalb des ADCs mit Reflection zu arbeiten. Erst bei der tatsächlichen Installation der Anwendung in einem Applikations-Server oder bei Überführung der fertigen Anwendung in die Produktivumgebung werden für alle Objekte, deren Attribute gelesen und geschrieben werden müssen, die entsprechenden Zugriffsroutinen generiert. Die Generatorimplementierung ist vergleichsweise einfach, wenn hierfür die Attribute eines Objekts mit den bereits vorgestellten Reflection-Mechanismen analysiert werden und dafür passende Codezeilen erzeugt werden.
Mit Hilfe des ADCs können auch Objektbäume[6] zur Reduktion des Implementierungsaufwands berücksichtigt werden. Dies ist vor allem bei Geschäftsobjekten interessant, die zueinander in Beziehung stehen (z.B. Kunden-Objekt hat Adress-Objekt). Somit werden die Daten mehrerer Objekte transportiert. Diese bilden aufgrund ihrer Beziehungsstruktur eine logische Einheit. Dadurch bildet der ADC ein ähnliches Verhalten ab, wie es beim Java-Serialisierungskonzept der Fall ist (vgl. Kapitel 2, Abschnitt 2.3 und Kapitel 3, Abschnitt 3.2.1). Es können die Daten ganzer Objektbäume übertragen werden. Abbildung 4.22 verdeutlicht diesen Umstand. Das Geschäftsobjekt `GO_1` besitzt Beziehungen zu den Geschäftsobjekten `GO_2` und `GO_3`. Darüber hinaus besteht noch eine Beziehung zwischen `GO_3` und `GO_4`. Bei Übergabe von `GO_1` an

[6] Falls Algorithmen zur Erkennung von Zyklen verwendet werden, können auch Graphen übertragen werden.

den ADC werden die Zustände aller beteiligten Objekte entnommen und gespeichert. Als Folge der Beziehung zwischen `GO_1` und `GO_2` werden ebenfalls mehrere Objektzustände im ADC für diesen Objekttyp abgelegt. In der Abbildung wurde eine logische Darstellung des ADC gewählt, d.h. die Daten der einzelnen beteiligten Objekte werden grundsätzlich als ADC repräsentiert, obwohl sie nicht unbedingt in diesem Format gespeichert werden (siehe oben). Statt dessen stellt der ADC an seiner Schnittstelle ADC-Objekte transparent für den Benutzer zur Verfügung. Durch die Implementierung des Entwurfsmusters *Dynamische Attribute* kann erreicht werden, daß nur bestimmte Attribute aus dem Objektbaum entnommen werden und die zu übertragende Datenmenge eingeschränkt wird. Durch die Berücksichtigung von Objektbeziehungen kann der Entwicklungsaufwand deutlich reduziert werden, da sich der Anwendungsentwickler nicht mehr um die Beschaffung von Attributen kümmern muß, die in einer tieferen Stufe der Objekthierarchie liegen. Gegenüber der Standardserialisierung von Objektbäumen können nun Attribute selektiv in Abhängigkeit des vorhandenen fachlichen Anwendungsfalls übertragen werden.

Der ablaufende Algorithmus für die in Abbildung 4.22 dargestellte Objektkonstellation ist anhand eines Sequenzdiagramms in Abbildung 4.23 skizziert.

Das übergebene Objekt (hier `GO_1`) wird als (potentielle) Wurzel eines Objektbaums angesehen, und in einem ersten Schritt werden alle Attribute samt Name und Wert entnommen (1). Bei den Attributen wird anschließend zwischen primitiven und komplexen Attributen unterschieden. Primitive Attribute sind dabei Java-Basistypen, wie z.B. `int`, `long` und `float`, aber auch Objekte, wie z.B. `Integer`, `Long`, `Float` und `String`. Diese werden unmittelbar im Daten-Container gespeichert, da sie nicht weiter aufgelöst werden können (2). Als komplexe Objekte werden alle anderen Objekte angesehen, dazu gehören insbesondere andere Geschäftsobjekte (hier `GO_2`, `GO_3`, `GO_4`). Für diese komplexen Objekte werden nun die beiden eben beschriebenen Schritte im Rahmen einer Rekursion wiederholt (3-8). Im Falle der Beziehung zwischen `GO_1` und `GO_2` werden die Schritte mehrfach durchgeführt, da ein Objekt `GO_1` mehrere `GO_2`-Objekte referenzieren kann.

Als Beispiel für die Entnahme von Daten aus Objekten, die nicht mit Java-Reflection analysiert werden können, sollen an dieser Stelle `ResultSet`-Objekte dienen. Ein `ResultSet`-Objekt ist das Ergebnis einer direkten JDBC-Datenbankanfrage, die zur Beschaffung von Daten, die Clients anfordern, dient. Um die Container-Funktionalität unabhängig vom Inhalt des `ResultSet`-Objekts zu implementieren, kann die in Abbildung 4.24 dargestellte Vorgehensweise implementiert werden.

Der Daten-Container analysiert dabei den Inhalt des Objekts `ResultSetMetaData`, das vom `ResultSet`-Objekt angefordert werden kann (1). Das Objekt beschreibt den Inhalt des `ResultSet`-Objekts. Der ADC ermittelt mittels der Methoden `getColumnCount()` und `getColumnLabel()` die Anzahl der Spalten und deren Bezeichnungen, die als Attributnamen im Daten-Container verwendet werden (2, 3). Pro Zeile des `ResultSet`-Objekts wird nun ein ADC-Objekt erzeugt (4) und mittels der Methode `getObject()` dem `ResultSet`-Objekt entnommen und mit der `put`-Methode in den Daten-Container geschrieben (5, 6). Am Ende fügt sich der ADC das neue Objekt selbst hinzu (7). Die dabei verwendete Kennung muß vom Benutzer bei Übergabe des `ResultSet`-Objekts mit übergeben werden. Diese Form der Implementierung kann mit wenig Aufwand realisiert werden und ist allgemeingültig, da sie sich den bereits vorhandenen ADC-Objekten und der Hinzufügeoperation des ADC bedient. Eine Alternative hierzu ist die direkte Speicherung der Daten in die Datenstruktur des ADCs.

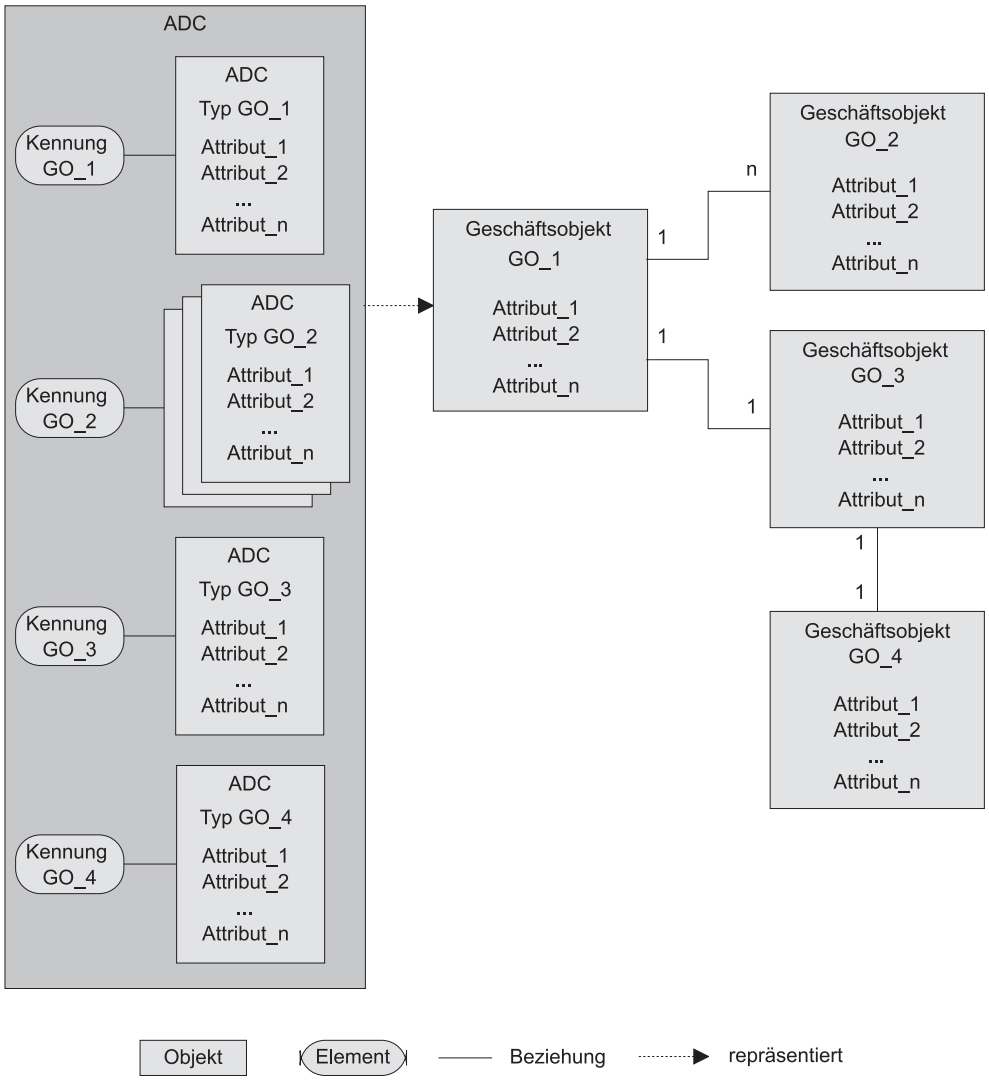


Abbildung 4.22: Berücksichtigung von Objektbäumen

Bei der direkten Übernahme von Daten aus einem `ResultSet-Objekt` kann es unerwünscht sein die Attributnamen der Tabelle zu übernehmen. Durch die optionale Übergabe von Zuordnungsinformationen kann zusätzlich erreicht werden, daß die Spaltennamen in eine andere Namensgebung überführt werden. Der zu diesem Zweck geänderte Datenverpackungsprozeß ist in Abbildung 4.25 dargestellt.

Zwischen dem Auslesen von Daten aus dem Datenobjekt (2) und dem Einfügen in den ADC (4) wird ein Objekt `MappingInfo` verwendet, das bei einer Übergabe eines Attributnamens der Relation den zugehörigen Attributnamen, der in der Anwendung verwendet werden soll, zurückgibt (3, `getMapping()`). Die Zuordnungsinformationen können hierzu z.B. tabellarisch in einer Datei oder einer Datenbanktabelle in der Form von Tabelle 4.1 abgelegt und gepflegt werden. Das Zuordnungsobjekt kann z.B. beim Systemstart intern eine Hash-Tabelle aufbauen, um die Informationen bereitzustellen. Im EJB-Umfeld gibt es die Möglichkeit das Objekt

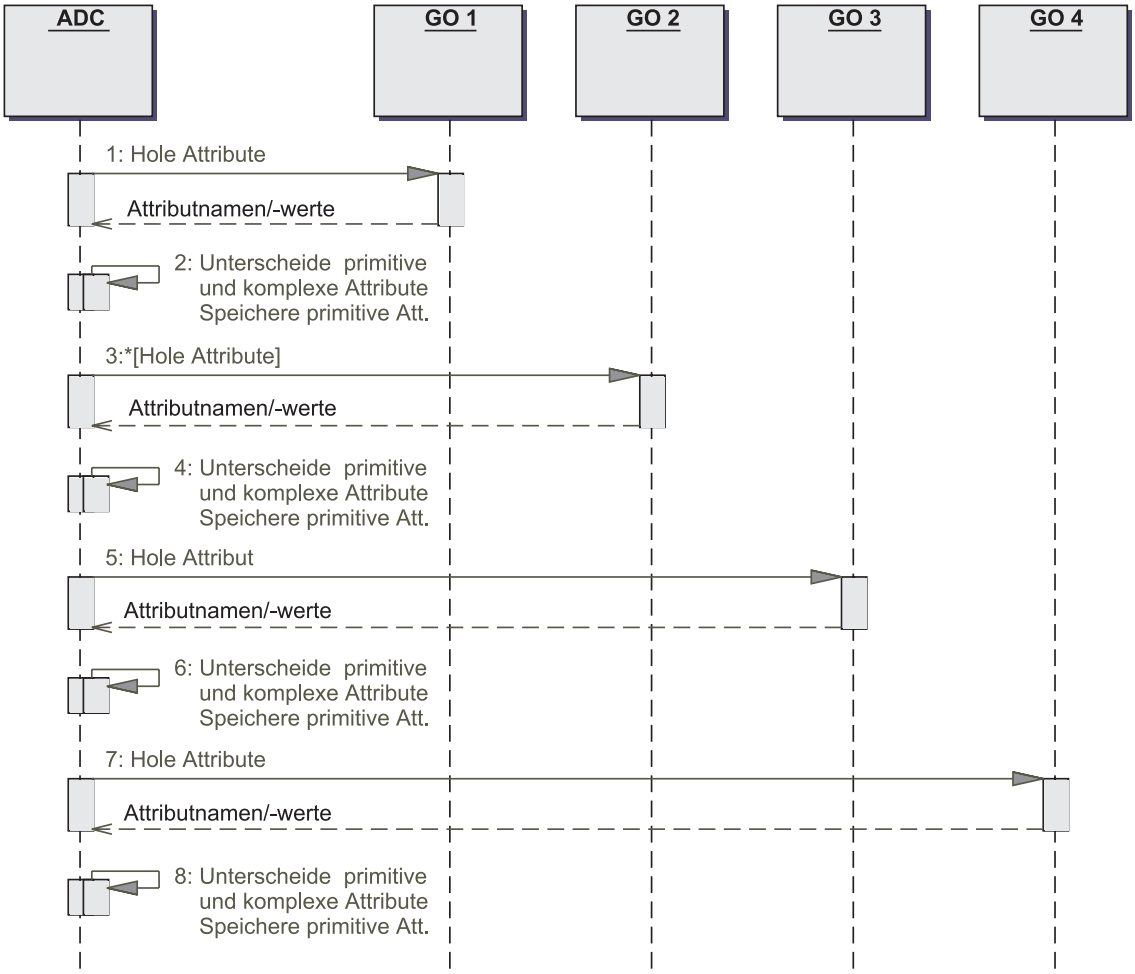


Abbildung 4.23: Ablauf einer Baumanalyse

zentral im JNDI bereitzustellen oder auch selbst als Session-Bean zu implementieren. Falls weitere Informationen, wie z.B. eine notwendige Datentypkonvertierung beim Zuordnungsvorgang berücksichtigt werden müssen, können diese z.B. durch weitere Spalten oder Zeilen angegeben und durch den Daten-Container berücksichtigt werden.

Attributname Relation	Attributname Anwendungssystem
Attributname_r1	Attributname_a1
Attributname_r2	Attributname_a2
. . .	. . .
Attributname_rN	Attributname_aN

Tabelle 4.1: Zuordnung neuer Attributnamen

Jeder Datensatz, der im `ResultSet`-Objekt vorhanden ist, wird somit später auf dem Client als ADC-Objekt repräsentiert und stellt sich damit gleich dar, als ob die Attribute eines Geschäftsobjekts ausgelesen wurden. Falls ein Bezug zwischen den Elementen des ADCs und tatsächli-

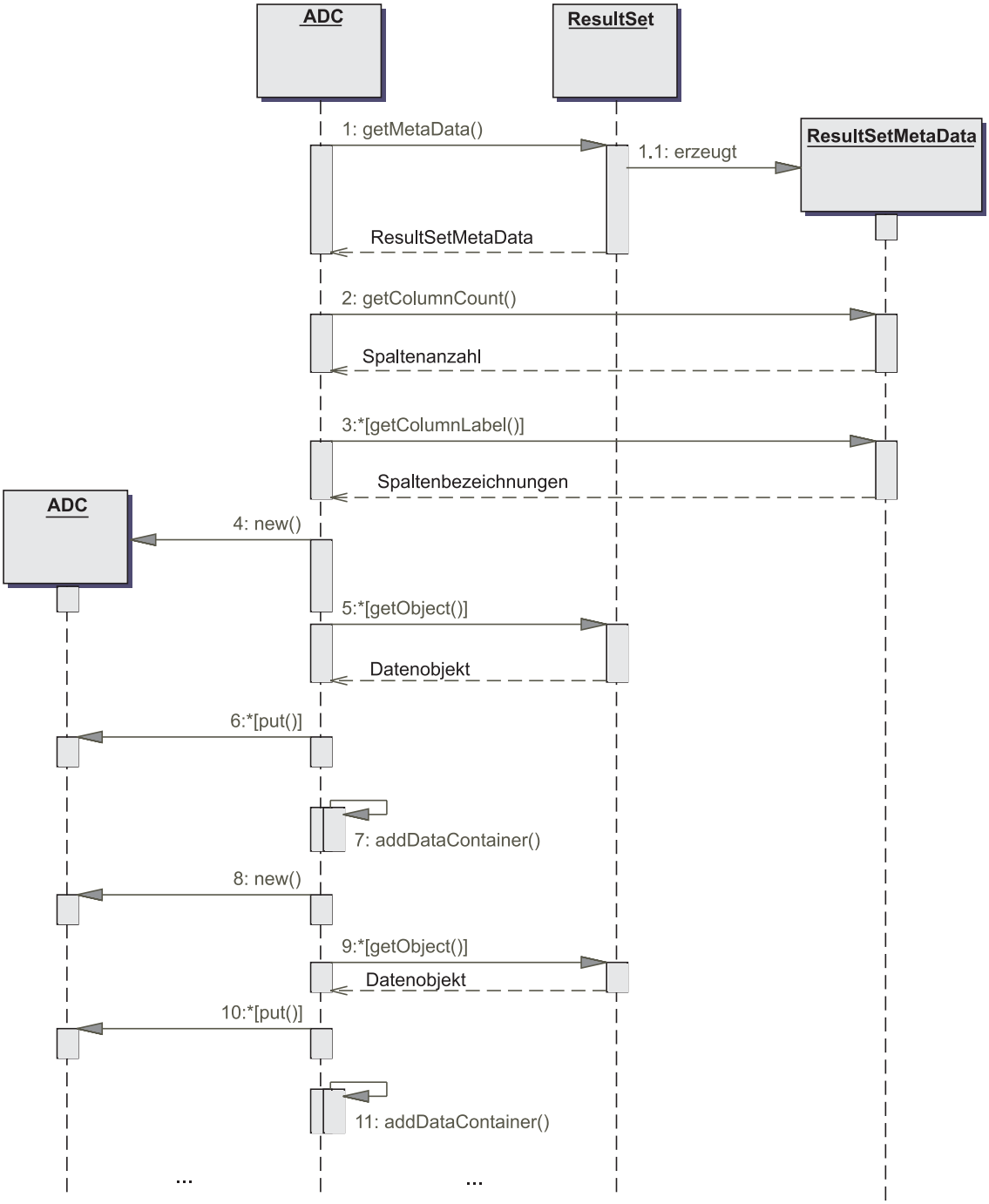


Abbildung 4.24: Allgemeine Analyse eines ResultSet-Objekts

chen Geschäftsobjekten besteht, ist es auch möglich bei Datenänderungen durch den Client auf dem Server wieder ein Geschäftsobjekt zu erzeugen, dessen Attribute mit den Container-Attributen synchronisiert sind. Dies kommt insbesondere dann in Betracht, wenn der direkte Datenbankzugriff durchgeführt wird, um aus Leistungsgründen objektorientierte Strukturen

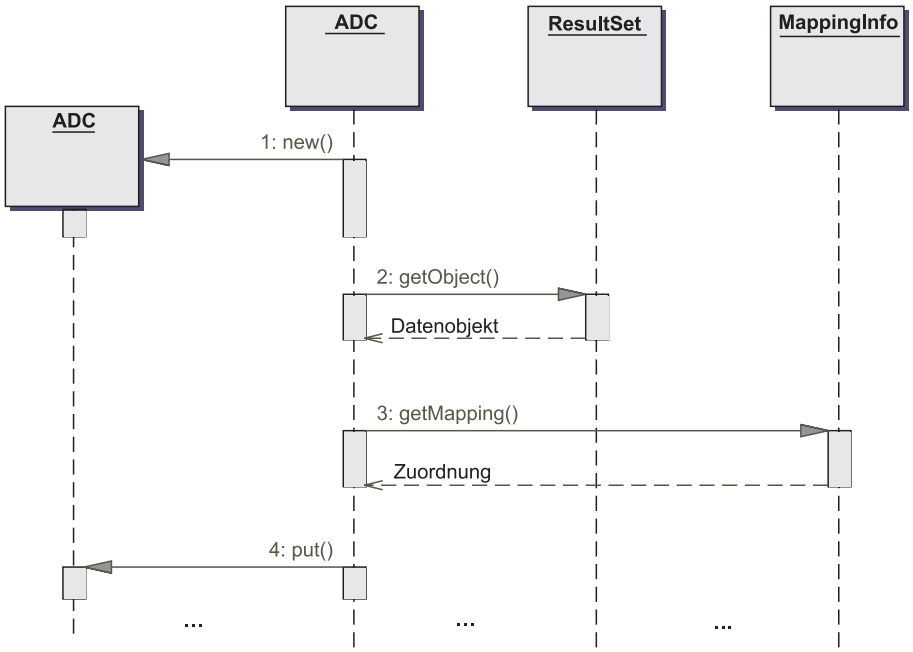


Abbildung 4.25: Ändern der Attributnamen beim Auslesen eines ResultSet-Objekts

und ein dafür verwendetes Persistenz-Framework zu umgehen. Um die Transaktionssicherheit bei Schreibzugriffen zu gewährleisten werden wieder Objekte verwendet, die der Kontrolle des Persistenz-Frameworks unterliegen. Dieser Aspekt betont das Ziel mittels ADC-Objekten Daten unabhängig von ihrer Quelle im System zu repräsentieren.

Diese Funktionalität kann auch dazu verwendet werden, um die bereits erwähnte Kopplungsproblematik unter Beibehaltung der automatischen Entnahme und Synchronisation zwischen Daten-Container und Geschäftsobjekten zu umgehen. Dadurch kann die Begrenzung der Verwendung von Zuordnungsinformationen im Zusammenhang mit `ResultSet`-Objekten aufgehoben und allgemein angewendet werden. In allen Fällen muß die Zuordnung nicht manuell von jedem Anwendungsentwickler implementiert werden, wie dies bei bestehenden Konzepten der Fall ist, sondern die Zuordnung kann einmalig zentral definiert und bereitgestellt werden.

Abbildung 4.26 faßt die soeben erläuterte Funktionalität im Überblick zusammen und verdeutlicht, wie das Konzept in einem Anwendungssystem verwendet werden kann.

Ein Client erzeugt dabei eine Anfrage, die zum Server gesendet wird (1). Auf dem Server wird aus der Anfrage dynamisch ein SQL-Kommando erzeugt (2) und über JDBC ausgeführt (3, 4). Die Datenbank gibt die Treffer zurück (5), die von der JDBC-Schicht als `ResultSet`-Objekt zurückgegeben werden. Anschließend wird das `ResultSet`-Objekt dem ADC übergeben (6), der die enthaltenen Attribute entnimmt und dazu bei Bedarf Zuordnungsinformationen heranzieht (7). Der Container wird zum Client zurückgeschickt (8) und kann dort ausgewertet werden (9). Entsteht nach der Auswertung Bedarf einer Datenoperation, kann der Client die (geänderten) Daten im Container an den Server zurückschicken (10). Hier besteht nun die Möglichkeit aus dem Daten-Container ein Geschäftsobjekt zu erzeugen, das durch ein Persistenz-Framework

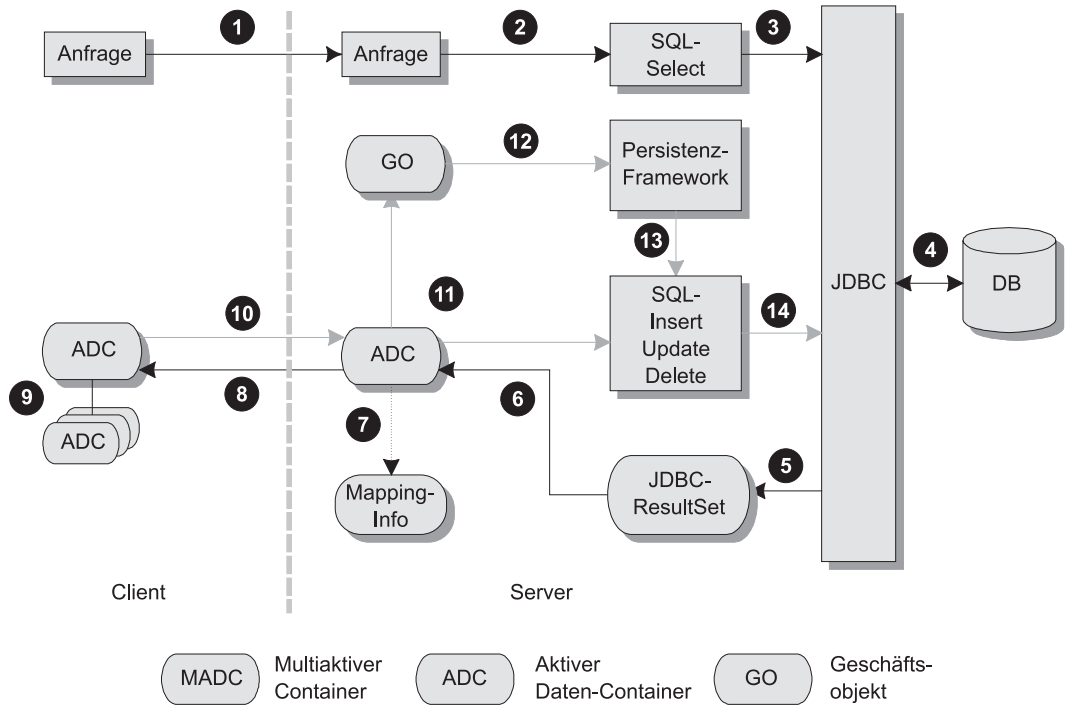


Abbildung 4.26: Verwendung ADC mit JDBC-ResultSet

in die Datenbank geschrieben wird (11, 12, 13, 14). Dies wird u.a. dadurch möglich, daß eine Zuordnung der Attribute bei Nichtübereinstimmung der Namen und Datentypen erfolgt. Als Alternative kann aus dem Daten-Container wieder eine SQL-Operation erzeugt werden, die mittels JDBC in der Datenbank ausgeführt wird (11, 14).

Zusatzfunktionalität

Die Implementierung der Zusatzfunktionalität kann auf unterschiedlichste Weise erfolgen. Dabei kommt zunächst das Kriterium der *Reichweite* in Betracht, die besagt, in welcher Granularität die Funktionalität angewendet wird:

- **Gesamter Daten-Container-Inhalt.** Funktionalität wird auf den gesamten Inhalt des Containers, unabhängig von den enthaltenen Attributen, angewendet. Am Beispiel einer Datenkompression bedeutet dies, daß die Datenstruktur, in der die Daten intern im Container gespeichert sind, komplett komprimiert wird.

- **Einzelattribute.** Funktionalität wird auf einzelne Attribute angewendet. Am Beispiel einer Datenkompression bedeutet dies, daß bestimmte Attribute, die im Container transportiert werden, komprimiert werden. Dadurch kann z.B. erreicht werden, daß eine spezielle Kompressionsroutine für bestimmte Attribute ausgeführt wird, die effizienter ist, als ein Kompressionslauf auf die komplette Datenstruktur des Containers, die evtl. Attribute enthält, die sich nicht komprimieren lassen, und damit eine unnötige Zeitdauer in Anspruch nehmen.

Insgesamt kann die Verwendung von Funktionalität sehr fein abgestimmt und optimiert werden. Durch die Bereitstellung im Daten-Container kann dies zentral erfolgen oder mit geringem Aufwand durch den Anwendungsentwickler, der die Funktionalität nur auslösen muß. Daraus ergibt sich eine weitere Frage nach dem *Zeitpunkt der Konfiguration* von Zusatzfunktionalität:

- **Konfiguration zur Übersetzungszeit.** Der ADC kann vollständig für einen fest vorgegebenen Fall implementiert werden. Dabei ist das Verhalten des Containers und die Funktionalität, die auf die Daten angewendet werden soll, nicht parametrisiert und damit bereits bei der Übersetzung fest vorgegeben. Falls mehrere Fälle vorhanden sind, können für diese Fälle weitere ADC-Objekte implementiert werden. Durch die Verpflichtung der ADC-Objekte, die Schnittstelle `ActiveContainer` zu unterstützen, können alle Objekte im System gleich behandelt werden, obwohl sie sich vollständig unterschiedlich verhalten können. In diesem Fall wird Polymorphie über die gemeinsame Schnittstelle ausgenutzt. Abbildung 4.27 stellt diese Form der Implementierung mit zwei unterschiedlichen Konzepten dar. Im linken Teil der Grafik werden unterschiedliche Container-Implementierungen erstellt, die von einem Basis-Container abgeleitet werden. Der Basis-Container kann z.B. als abstrakte Klasse implementiert werden, die eine Grundversion des Containers realisiert, die in den abgeleiteten Klassen entsprechend der benötigten Fälle konkretisiert wird. Im rechten Teil werden mehrere Container-Implementierungen erstellt, die das Interface `ActiveContainer` direkt implementieren. Dies ist insbesondere dann sinnvoll, wenn die Implementierungsklassen bereits von einer anderen Klasse abgeleitet werden. Eine grundsätzliche Frage dabei ist, ob die Container-Implementierungen von Anwendungsentwicklern nach Bedarf entwickelt werden. Dies birgt die Gefahr, daß zu viele, einzeln zu wartende Klassen erstellt werden. Die Entscheidung darüber muß durch das Benutzungskonzept der Container getroffen werden. Anstatt das Verhalten des Daten-Containers vollständig im Code zu verankern, können z.B. statische Konstanten (`static final`) im Code vorgesehen werden, die anhand ihres Wertes ein bestimmtes Verhalten im Container spezifizieren und somit vor der Übersetzung des Containers geändert werden können. Zur Änderung des Verhaltens müssen die statischen Konstanten geändert und der Container neu übersetzt werden.

- **Konfiguration zur Laufzeit.** Der ADC kann vollständig parametrisiert werden und sein Verhalten zur Laufzeit ändern. Hierzu werden dem ADC die Konfigurationsinformationen von außen während seines Lebenszyklus übergeben. Die dabei in Frage kommenden Verfahren werden nachfolgend im Zusammenhang mit der Form der Anwendung diskutiert.

Wie bereits angedeutet muß noch die *Form der Anwendung* festgelegt werden:

- **Manuelle Anwendung.** Der Anwendungsentwickler löst die Zusatzfunktionalität manuell mittels Methoden aus, die der Daten-Container zur Verfügung stellt. Am Beispiel der Datenkompression bedeutet dies, daß der Entwickler auf dem Server nach dem Füllen des Daten-Containers eine vorhandene `compress`-Methode und auf dem Client eine `decompress`-Methode aufruft, bevor er auf die Daten zugreift. Die beiden Methoden können noch um eine Methode `isCompressed()` mit einer booleschen Rückgabe ergänzt werden, die eine Abfrage ermöglicht, ob eine Kompression vorliegt. Soll die

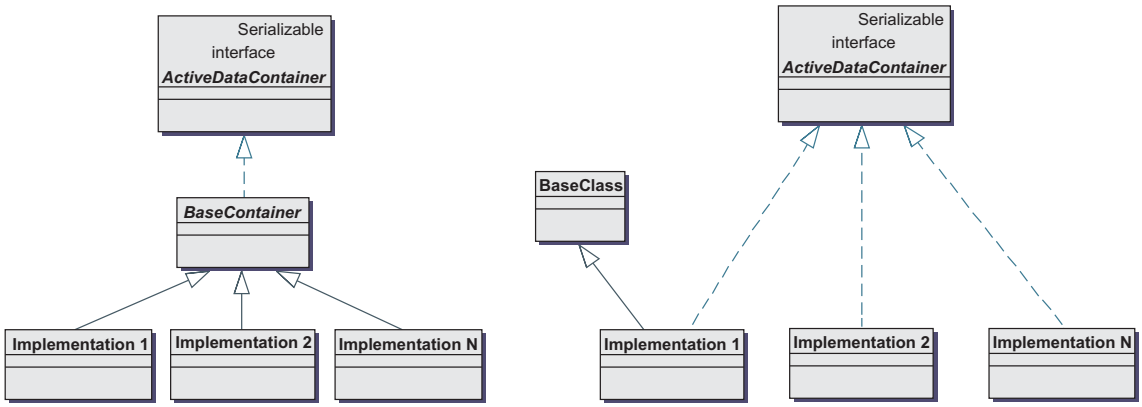


Abbildung 4.27: ADC-Polymorphie

Funktionalität auf einzelne Attribute angewendet werden, müssen die drei Methoden als Parameter den Namen des Attributs, auf das die Funktionalität angewendet werden soll, entgegennehmen.

- **Halbautomatische Anwendung.** Der Anwendungsentwickler konfiguriert den Container mit der anzuwendenden Funktionalität. Dies kann z.B. beim Erzeugen des Containers im Konstruktor oder mittels einer `addFunction`-Methode auf einen bereits vorhandenen Container geschehen. Die Container-Implementierung führt anschließend die Funktionalität entsprechend der Reichweite automatisch aus.

- **Vollautomatische Anwendung.** Bei dieser Alternative hat der Anwendungsentwickler keinen Einfluß auf die ausgeführte Funktionalität. Die Funktionalität wird dabei von einem Systementwickler zentral konfiguriert und in den Container-Objekten transparent für die Anwendungsentwickler zur Ausführung gebracht. Bei EJBs ist auch die Konfiguration der Daten-Container-Parameter im Deployment-Deskriptor interessant, um die Datenübertragung beim Deployment-Prozeß, je nach Bedarf, einzustellen. Dieses Konzept wird in Kapitel 5, Abschnitt 5.2.1 näher beschrieben.

Nachfolgend werden verschiedene technische Ansätze zur Implementierung von Zusatzfunktionalität vorgestellt, die sich im wesentlichen auf die soeben diskutierten Fragen beziehen. Zur Anwendung der Funktionalität kann der Container die `Externalizable`-Schnittstelle implementieren, um den Vorgang der Serialisierung und Deserialisierung zu beeinflussen. Die Serialisierung wird dabei in der `writeExternal`-Methode vorgenommen. Dabei können die jeweils konfigurierten Funktionen auf die im Container gespeicherten Daten angewendet werden. Die Funktionalität wird dabei zum Zeitpunkt des Versendens angewendet. Die Deserialisierung wird mit der Methode `readExternal()` kontrolliert. Dabei können die Funktionen, falls erforderlich rückgängig gemacht werden, um die Daten wieder in den Zustand zurückzubringen, in dem sie beim Versenden auf dem Server waren. Die umgekehrte Funktionalität wird damit zum Zeitpunkt des Datenempfangs auf dem Client ausgeführt. Ein alternativer Implementierungsansatz besteht darin, die Funktionalität unmittelbar nach der Übergabe eines Objekts, dessen Attribute transportiert werden sollen, auszuführen. Eine evtl. erforderliche Wiederauf-

bereitung auf dem Client kann z.B. beim ersten Zugriff mittels einer `get`-Methode erfolgen. Falls der Container komplett manuell mit Daten befüllt wird, kann es erforderlich sein, daß bei jeder Hinzufügeoperation (`set()`) und bei jeder Entnahmeoperation (`get()`) Funktionalität ausgeführt wird.

Nachfolgend soll exemplarisch beschrieben werden, wie Zusatzfunktionalität in den ADC-Transportobjekten implementiert werden kann. Es handelt sich dabei um für den Client transparente Schreibzugriffe auf EJBs und die Kompression von Daten mittels Zip-Mechanismen. Damit wird der Daten-Container zusätzlich zum universellen und intelligenten Stellvertreter für Geschäftsobjekte. Im Gegensatz zum herkömmlichen intelligenten Stellvertreter (Smart-Proxy) wird die Funktionalität hier mit einem Datentransportobjekt gekoppelt, nur einmal für alle Geschäftsobjekte implementiert und zusätzlich dynamisch konfigurierbar gemacht. Zur Realisierung eines transparenten Schreibzugriffs auf eine EJB muß der ADC die Referenz auf die entfernte Bean kapseln. Wie in Abbildung 4.28 dargestellt wird, kann der Daten-Container hierzu eine EJB-Referenz, die ihm auf dem Server übergeben wird in seiner internen Datenstruktur zusätzlich zu den angeforderten Attributen speichern. Bezüglich der Herkunft und Art der übergebenen EJB-Referenz besteht völlige Freiheit, solange gewährleistet ist, daß Schreibzugriffe auf dem Container durch die Reflection-Routine in einen entfernten Zugriff umgesetzt werden können. Die Referenz kann z.B. mittels der `getEJBObject`-Methode des Kontext-Objekts einer Session- oder Entity-Bean beschafft werden oder bei Entity-Beans das Ergebnis einer `finder`-Methode sein. Nachdem ein Daten-Container zum Client geschickt wurde, werden Aufrufe zum Setzen von Attributen (`set`-Methode) im Container mittels *Reflection*-Mechanismen in Methodenaufrufe der EJB-Referenz umgesetzt und aufgerufen, wodurch ein entfernter Methodenaufruf stattfindet. Das Durchschreiben kommt der Forderung nach Datenaktualität auf dem Server nach und betont die Tatsache, daß in einem Daten-Container nur die Kopie von Attributen eines Objekts enthalten ist, das sich in Wirklichkeit auf dem Server befindet. Die Verwendung von Java-Reflection macht dieses Vorgehen universell für jede EJB möglich. Dabei muß für die auf EJB-Seite vorhandenen Methoden ein Namensschema existieren, das sich aus den Attributnamen herleiten läßt. Das Namensschema kann z.B. durch ein konfigurierbares Zuordnungskonzept, wie es bereits dargestellt wurde, unterstützt werden. Eine Alternative ist der Aufruf einer generischen Schreibmethode der Bean, die den Namen und den Wert des zu schreibenden Attributs entgegennimmt, um das Schreiben durchzuführen. Die gesamte Vorgehensweise kann insoweit verfeinert werden, daß nur bestimmte Attribute transparent geschrieben werden, für die dieses Verhalten unbedingt erforderlich ist. Viele entfernte Lesezugriffe werden verhindert, indem mehrere Attribute gleichzeitig übertragen werden. Die Attribute werden auf dem Server mittels entsprechender Methoden übergeben und innerhalb eines entfernten Methodenaufrufs übergeben. Dies führt zu einer Optimierung des Kommunikationsaufwands. Der lesende Zugriff mittels `get`-Methode auf dem Client führt hier zu keinem entfernten Methodenaufruf, sondern zu einem Zugriff auf die im Daten-Container transportierten Attribute.

Diese Funktionalität kann Grundlage für die Verwendung weiterer Muster sein, wie z.B. eines Iterators (vgl. Iterator-Muster [Gam95]) der zum Einsatz kommt, wenn eine große Datenmenge nicht auf einmal zum Client übertragen werden kann. Beim Lesen wird dabei nur ein Teil der Daten übertragen. Beim Zugriff auf Daten, die nicht im Daten-Container enthalten sind, können diese transparent für den Client nachgefordert werden. Falls die Verwendung eines Iterators

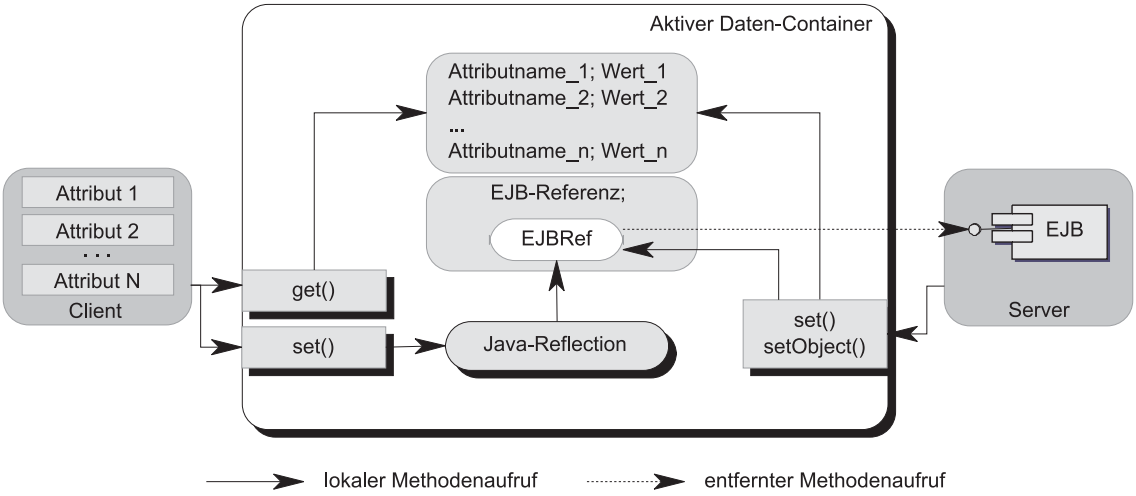


Abbildung 4.28: Transparenter Schreibzugriff auf Enterprise-Beans

nicht möglich ist, kann es sinnvoll sein hohe Datenmengen mittels eines leistungsfähigen Kompressionsalgorithmus zu komprimieren. Dabei können Kompressionsalgorithmen zum Einsatz kommen, die speziell auf die transportierten Daten abgestimmt sind. Die Java-Bibliothek unterstützt ebenfalls eine leistungsfähige Datenkompression mit dem Paket `java.util.zip`, das die Datenformate *ZIP* und *GZIP* unterstützt, die Daten mittels des *Deflate*-Algorithmus komprimieren [Mica]. Nähere Informationen zu den Datenformaten und dem realisierten Kompressionsalgorithmus können [Deu96a, Deu96b, Deu96c, PKW96] entnommen werden.

Aufgrund der Flexibilität des Containers können auch mehrere Algorithmen angeboten werden, die je nach transportierten Daten eingestellt werden. Die hier vorgestellte, in Abbildung 4.29 dargestellte Lösung, geht davon aus, daß die Kompression und Dekompression manuell mittels den Methoden `zip()` und `unzip()` durch den Anwendungsentwickler ausgelöst wird. Zusätzlich kann mittels der Methode `isZipped()` festgestellt werden, ob der Inhalt des Daten-Containers komprimiert ist oder nicht. Der Komprimierungsvorgang selbst und die Speicherung der komprimierten Daten in einer passenden Datenstruktur wird hier in einem darauf spezialisierten Objekt gekapselt. Das Kompressionsobjekt ist selbst serialisierbar und nimmt beliebige andere Java-Objekte entgegen, um diese intern mittels Deflate zu komprimieren und abzulegen. Beim Aufruf der `zip`-Methode des Containers wird ein Kompressionsobjekt erzeugt und die Datenstruktur des Daten-Containers übergeben. Anstatt die sonst übliche Datenstruktur zu übertragen, wird nun der Zustand des Kompressionsobjekts übertragen. Die normale Datenstruktur (hier: `HashMap`) wird auf `null` gesetzt. Bei Aufruf der Methode `unzip` wird die eigentliche Datenstruktur dem Kompressionsobjekt wieder entnommen. Bei der Entnahme dekomprimiert das Objekt die eigentliche Datenstruktur. Nun wird das Kompressionsobjekt nicht mehr benötigt und auf `null` gesetzt. Für eine konkrete Implementierung des Kompressionsobjekts sei an dieser Stelle z.B. auf [Sun98] verwiesen, worin die Anwendung der Klassen `ZipOutputStream` und `ZipInputStream` des Pakets `java.util.zip` beschrieben wird. Als Implementierungsalternative kommen ebenfalls die Klassen `GZIPOutputstream` und `GZIPInputStream` in Frage.

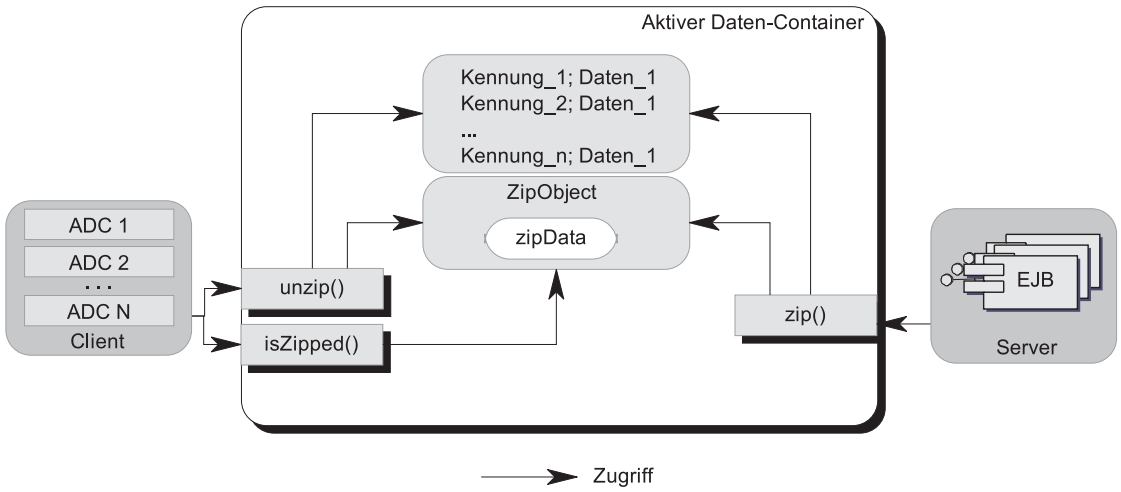


Abbildung 4.29: ADC-interne Kompression

Eine andere Implementierungsalternative bestünde darin, eigene Serialisierungsmethoden für den ADC bereitzustellen, die automatisch beim Vorgang der Serialisierung/Deserialisierung die Komprimierung/Entkomprimierung vornehmen. Dies ermöglicht die Automatisierung der Kompression ohne manuelle Eingriffe des Anwendungsentwicklers. Diese Implementierungsalternative kann auch herangezogen werden, falls das verwendete Kommunikationsprotokoll die Übergabe von Daten zwischen verschiedenen Komponenten dahingehend optimiert, daß Datenstrukturen per Referenz zwischen Client und Server übergeben werden, wenn sie sich im selben Prozeß befinden, also nicht über das Netzwerk transportiert werden müssen. Im Falle einer Übergabe per Referenz werden die Serialisierungsmethoden nicht aufgerufen und es findet keine unnötige Kompression statt. Falls der Daten-Container tatsächlich über ein Netzwerk übertragen wird, führt dies zum Aufruf der Serialisierungsmethoden und damit zu einer evtl. notwendigen Komprimierung der Daten.

4.4 Aktive Value Objects

In diesem Abschnitt soll erläutert werden, wie Konzepte des Aktiven Daten-Containers in Systeme eingebracht werden können, die auf statischer Datenübertragung basieren. Hierzu werden in diesem Abschnitt *Aktive Value Objects (AVOs)* entwickelt. Dabei wird ein neues Konzept zur Entwicklung von Anwendungen vorgeschlagen. Falls Geschäftsobjekte in Form von Java-Objekten direkt übertragen werden sollen, können die in diesem Abschnitt vorgestellten Konzepte analog angewendet werden.

4.4.1 Motivation

Die Motivation entspricht den bereits in Abschnitt 4.3 gemachten Ausführungen. Dabei soll jedoch eine Verbesserung von statischen Datenübertragungskonzepten erfolgen, die auf der Ver-

wendung von Daten-Containern basieren, die bereits zur Übersetzungszeit festlegen, welche Daten zu transportieren sind. Bei diesem Entwurfsansatz müssen viele Container-Klassen erstellt werden, deren Attribute die zu übertragenden Daten eines Anwendungsfalls abdecken. Somit müssen z.B. pro Geschäftsobjekt, pro Dialog oder pro Dialogelement Daten-Container erstellt werden. Der resultierende, höhere Entwicklungsaufwand führt zu einer strengeren Typisierung von Operationen, die zum Datenaustausch mit dem Container verwendet werden. Dies hat zur Folge, daß diese Operationen zur Übersetzungszeit der Typüberprüfung des Java-Compilers unterliegen. Dieser Entwurfsansatz kann mit den aktiven Konzepten dieser Arbeit dadurch verbessert werden, daß

- die Datenübertragung zentral beeinflußt werden kann,
- Zusatzfunktionalität in den Containern ausgeführt wird,
- auf deklarativer Ebene definiert wird, wie die Datenübertragung erfolgen soll,
- die Anwendungsimplementierung vereinfacht wird,
- die Anzahl der zu implementierenden Container-Klassen gesenkt wird,
- Systeme nachträglich im Hinblick auf die Datenübertragung erweitert oder z.B. durch Reduktion der Datenmenge, optimiert werden können,
- auf Wunsch eine Standardschnittstelle zur Verfügung steht, die z.B. von Framework-Komponenten genutzt werden kann,
- eine Zusammenfassung von Daten aus mehreren Quellen erfolgen kann, die in einem Aufruf zum Client übertragen werden können.

4.4.2 Struktur

Zur Durchsetzung der geforderten Flexibilität müssen alle im System vorhandenen Daten-Container eine Standardschnittstelle besitzen, die die folgenden Operationen erlaubt:

- **Objektübergabe/Objektentnahme**: Objekte, deren Daten transportiert werden sollen, müssen dem AVO übergeben bzw. entnommen werden können. Der Datenaustausch zwischen den beteiligten Objekten erfolgt dabei automatisch, um die Anwendungsentwicklung zu vereinfachen und zu beschleunigen. Falls es ermöglicht wird, nur die wirklich vom Client benötigten Attribute zur Übertragung anzugeben, genügt die Implementierung eines einzigen AVOs, statt vieler für unterschiedliche Anwendungsfälle. Somit tritt eine weitere Reduzierung des Entwicklungsaufwands ein, da weniger Transportklassen implementiert werden müssen. Gleichzeitig wird das Leistungsverhalten der Anwendung adressiert, da die unnötige Übertragung von Attributen, die eigentlich nicht benötigt werden, verhindert wird.

- **Generisches Lesen/Schreiben von Attributen**: Attribute müssen vom Typ des AVOs unabhängig gelesen und geschrieben werden können. Dazu gehört auch die Umsetzung von Iteratoren, die ein systematisches Durchlaufen der Attribute und ihrer Werte ermöglichen. Diese Funktionalität kann von verallgemeinerten Komponenten genutzt werden, um beliebige Objekte zu verarbeiten.

- **Konfiguration von Zusatzfunktionalität**: Erforderliche Konfigurationsmethoden für Zusatzfunktionalität, wie z.B. die Anwendung eines Kompressionsalgorithmus, müssen verfügbar sein.

Zur Übertragung mehrerer AVOs kann ein herkömmlicher ADC, wie in Abschnitt 4.3 verwendet werden. Dieser Container dient als Sammelbehälter zur strukturierten Ablage der Zustände mehrerer AVOs und übernimmt gleichzeitig die Aufgabe zur Anwendung von Funktionalität auf alle im Container gespeicherten Daten in ihrer Gesamtheit. Dies kann z.B. zur Optimierung genutzt werden, indem der gesamte Inhalt des ADCs zum Versand komprimiert und nach dem Empfang dekomprimiert wird. Alternativ kann auch die Verwendung von herkömmlichen Collection-Klassen, wie z.B. `java.util.Vector` erfolgen, um mehrere AVOs gleichzeitig zu übertragen. In Abbildung A.9 ist die Struktur des Konzepts anschaulich dargestellt.

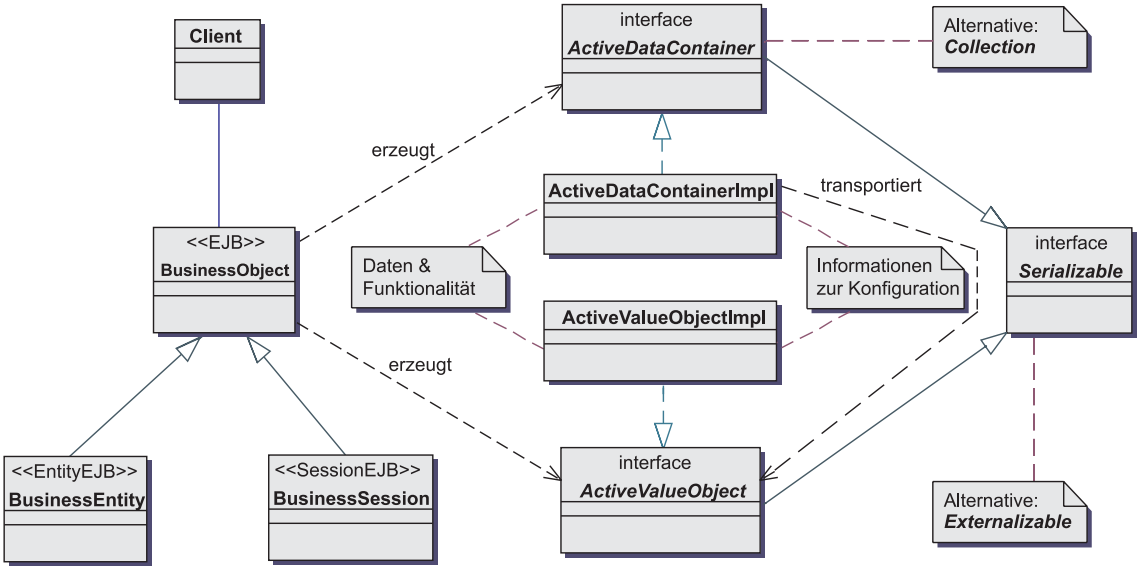


Abbildung 4.30: Prinzip von Aktiven *Value Objects*

4.4.3 Implementierungsstrategien

Im folgenden wird erläutert, wie AVOs implementiert werden können. Dabei wird jedoch auf die Diskussion, wie eine Standardschnittstelle in den einzelnen AVOs realisiert werden kann und wie ein automatischer Datenaustausch zwischen Geschäftsobjekt und Daten-Container erfolgt, weitestgehend verzichtet. In Abschnitt 4.3 wurde bereits ausführlich erläutert, welche

Methoden eine Standardschnittstelle benötigt und wie diese implementiert werden können. Dabei wurde auch diskutiert, wie mittels Java-Reflection auf Objekte zugegriffen werden kann und wie diese Routinen aus Laufzeitgründen mittels eines Generatoransatzes eliminiert werden können. Diese Ansätze können in AVOs nahezu unverändert übernommen werden. Der einzige Unterschied besteht darin, daß die benötigten Methoden in jedem AVO implementiert sein müssen. Dies kann durch einen Vererbungsansatz oder durch die Generierung der Methoden für jedes AVO erreicht werden. Der Schwerpunkt der Diskussion liegt in diesem Abschnitt auf neuen Ansätzen, wie AVOs zur Reduktion des Aufwands bereitgestellt werden können und wie gleichzeitig eine Reduktion der Datenmenge und -komplexität bei der Übertragung erfolgen kann. Dabei werden auch gezielt andere Implementierungen der neuen Konzepte dieser Arbeit gewählt, um deren Allgemeinheit und Anpaßbarkeit zu zeigen.

Ansatz mit Vererbung

Zunächst soll eine auf Vererbung basierende Implementierung vorgestellt werden. Dabei müssen alle im System vorhandenen Daten-Container von einer zentralen Klasse abgeleitet werden, die eine universelle Routine zur Serialisierung ihrer Kindklassen implementiert und eine Standardschnittstelle anbietet, mit der auf Wunsch alle im Container transportierten Attribute gelesen und geschrieben werden können. Dieses Konzept ist in Abbildung 4.31 dargestellt.

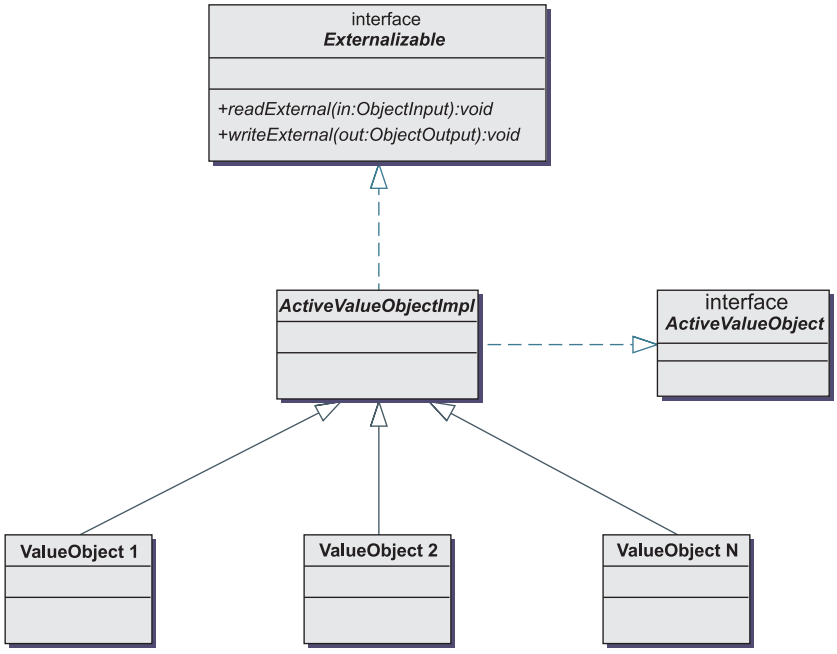


Abbildung 4.31: Aktive *Value Objects* mit Vererbung

Die abstrakte Basisklasse `ActiveValueObjectImpl` stellt das Kernelement dar und implementiert die Schnittstelle `Externalizable`, um die Serialisierung vollständig steuern zu können. Die Schnittstelle `ActiveValueObject` stellt die Standardschnittstelle bereit, auf die dynamisch zugegriffen werden kann, um Attribute im Container zu lesen und zu setzen. Da-

bei wird jede beliebige Klasse um aktive Konzepte erweitert. Durch die gemeinsame Schnittstelle ergibt sich auch die Möglichkeit EJBs nachträglich durch neue AVOs zu erweitern, ohne den Quellcode zu ändern. Dies kann z.B. bei Entity-Beans angewendet werden, die sehr viele Attribute besitzen und anwendungsunabhängig verwendet werden sollen. In solchen Fällen ist es wahrscheinlich, daß jede Anwendung, in deren Rahmen die Entity-Bean verwendet werden soll, unterschiedliche Anforderungen an die benötigten Attribute stellt. Durch die nachfolgend dargestellte Schnittstelle kann erreicht werden, daß mit dem Parameter `name` ein beliebiges AVO angefordert werden kann, das der übergebenen Bezeichnung entspricht. Das passende Objekt wird erzeugt, mit Daten belegt und als ADC zum Client übertragen. Um auf die Attribute und Methoden typsicher zugreifen zu können, muß nun eine Typumwandlung in den Typ des AVOs erfolgen.

```
ActiveValueObject getValueObject(String name)
       throws RemoteException;
```

Die in Abschnitt 4.3 erläuterten Konzepte und Vorgehensweisen werden hier von der Basisklasse zugesteuert und können weitestgehend übernommen werden. Durch die Implementierung der `Externalizable`-Schnittstelle ist die allgemeine Realisierung der Methoden `readExternal()` und `writeExternal()` möglich, die für alle AVOs gelten und Kontrolle darüber erlauben, welche Daten in welcher Form übertragen werden. Dabei können z.B. Optimierungen vorgenommen werden. Nachfolgend ist eine rein exemplarische Implementierung dargestellt, die auf der Verwendung des in Abschnitt 4.3 entwickelten Daten-Containers beruht. Der ADC wird hier als Transportobjekt mit Zip/Unzip-Funktion verwendet:

```
// In der Basisklasse
ActiveContainer ac=new ActiveContainerImpl();

// Serialisierung
public void writeExternal(ObjectOutput out)
{
  ...
  // Daten dieses Objekts im ADC speichern
  ac.setObject(this);
  // Daten komprimieren
  ac.zip();
  // Daten serialisieren
  out.writeObject(ac);
}

// Deserialisierung
public void readExternal(ObjectInput in)
{
  ...
  // Objekt deserialisieren
  ac=(ActiveContainer) in.readObject();
```

```
  // ggf. dekomprimieren und dieses Objekt
  // mit den Container-Attributen synchronisieren
  if(ac.isZipped())
  {
     ac.unzip();
     ac.synchronize(this);
  }
  else
  {
     ac.synchronize(this);
  }
}
```

Das Codefragment macht deutlich, daß durch diese Vorgehensweise kontrolliert werden kann, in welcher Form die Daten tatsächlich übertragen werden. Auf die Darstellung der nachträglichen Konfigurierbarkeit wird in dem Codefragment verzichtet. Die `zip`-Methode ist hier aus Vereinfachungsgründen fest implementiert. Die Konfiguration kann jedoch durch die Parametrisierung der Funktionalität auch aus einer externen Quelle geleistet werden, die dem Objekt z.B. bei dessen Erzeugung mit übergeben wird. Hierzu kann z.B. in der Basisklasse eine Suche nach dem EJB-Kontext oder nach einem Objekt im JNDI erfolgen, um diesem Informationen über die zu verwendenden Implementierungen und Funktionen zu entnehmen. Falls die Kontrolle über diese Größen für jede EJB individuell erfolgen soll, müssen allerdings bei der Erzeugung eines Daten-Containers Informationen über die EJB, in der er zur Verwendung kommt, übergeben werden. Alternativ hierzu kann die Angabe der Implementierung und Funktionen manuell durch den Anwendungsentwickler erfolgen, erschwert jedoch die nachträgliche, transparente Einstellung dieser Variablen. In Kapitel 5, Abschnitt 5.2 wird erläutert, wie der Bean-Kontext dazu verwendet wird, Parameter aus dem JNDI zu entnehmen, die im Deployment-Deskriptor definiert sind.

Generatorbasierende Ansätze ohne Vererbung

Im Rahmen dieser Arbeit werden neue, auf Generatoren basierende Ansätze zur Erzeugung von AVOs vorgeschlagen, die gleichzeitig eine Optimierung der übertragenen Daten vornehmen. Mit diesen Ansätzen ist es auch möglich, nachträglich die Datenrepräsentation zum Zwecke der Datenübertragung zu ändern, ohne ein Eingreifen durch die einzelnen Anwendungsentwickler an verschiedenen Systemstellen zu erfordern. Die Generierung von optimierten AVOs wird dabei fester Bestandteil des Entwicklungsprozesses einer Anwendung. Das Grundprinzip ist in Abbildung 4.32 dargestellt.
Anhand der Definition der Attributnamen und Attributtypen wird neben dem Rahmengerüst für eine Geschäftsklasse auch ein zugehöriges AVO generiert. Die Geschäftsklasse und das AVO enthalten bereits neben den üblichen `get-` und `set`-Methoden für den Datenzugriff alle weiteren Methoden, die zum automatischen Datenaustausch zwischen Geschäftsobjekt und AVO sowie zur Herstellung eines für die Übertragung günstigeren Formats benötigt werden. Durch den Generatoransatz ist es auch möglich auf die langsamere Java-Reflection beim Objektzugriff zu verzichten. Die Abbildung enthält zusätzlich einen Umsetzungsvorschlag des Konzepts, der

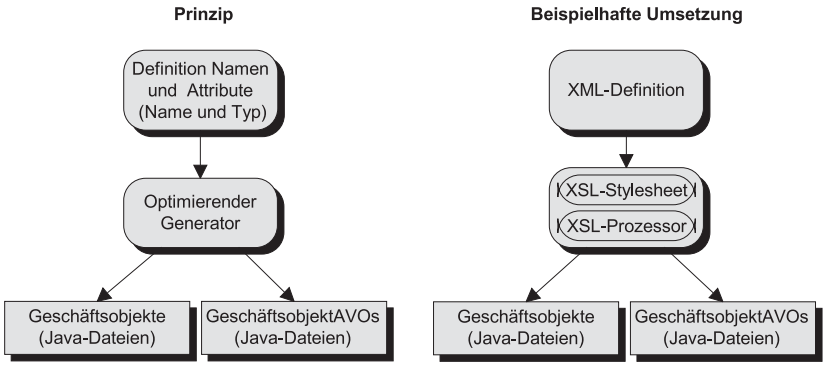


Abbildung 4.32: Generatorkonzept zur Erzeugung von AVOs

aus einer XML-Definition der Attribute besteht, die anschließend mittels XSL[7] -Stylesheet und XSL-Prozessor[8] in entsprechende Java-Quellen umgesetzt wird. Eine alternative Umsetzung des Konzepts besteht z.B. darin, ein UML-Entwicklungswerkzeug zu verwenden, das eine modifizierbare Codegenerierung aus dem Klassenmodell ermöglicht. Schließlich kann die Generierung von Optimierungsmethoden auch nachträglich erfolgen, indem bereits fertige Geschäfts- und normale AVO-Klassen durch einen Generator erweitert werden.
Eine einfache Implementierung des AVO-Konzepts mit Hilfe von Generatoren besteht zunächst darin, die AVOs so zu generieren, daß jedes einzelne die Schnittstelle `Externalizable` implementiert und in den zugehörigen Methoden `writeExternal()` bzw. `readExternal()` seine Attribute direkt serialisiert bzw. deserialisiert. Zum Transport einer Folge von AVOs kann z.B. ein Objekt vom Typ `Vector` verwendet werden, da sich jedes AVO selbst um seine Serialisierung kümmert. Dabei wird auf die Implementierung eines weiteren speziellen Sammel-Containers und damit auf eine Optimierung der entstehenden Datengesamtheit verzichtet. Dieser Implementierungsansatz ist in Abbildung 4.33 dargestellt.
Das nachfolgende Codefragment skizziert beispielhaft eine `writeExternal`-Methode. Die Serialisierung erfolgt dabei über eine Hilfsklasse, die für jeden Attributtyp eine spezialisierte Methode anbietet. In dieser Methode können z.B. Optimierungen vorgenommen werden, wie sie bereits in Abschnitt 4.3 genannt wurden. Im Rahmen dieser Arbeit wurde beispielhaft eine eigene Codierung von Ganzzahlen verwendet, die zu einer Datenreduktion bei Zahlenwerten führt, die nicht die volle Bitbreite ihres Datentyps benötigen. `<Type>` ist durch den konkreten Typ eines Attributs zu ersetzen. Durch den Generator kann der Inhalt der Methode jederzeit geändert werden. Auf die Verwendung einer Hilfsklasse kann auch verzichtet werden. Auf die Darstellung der `readExternal`-Methode wird an dieser Stelle verzichtet. Sie ist analog aufgebaut, verwendet jedoch die korrespondierenden `read<Type>`-Methoden der Hilfsklasse.

```
public void writeExternal(ObjectOutput out) throws IOException
{
  Helper.write<Type>(out, this.attribut1);
```

[7] *Extensible Stylesheet Language*. Die Sprache erlaubt die Festlegung von Regeln, die eine XML-Definition in ein beliebiges anderes Format umsetzen [Ahm01].
[8] Ein leistungsfähiger XSL-Prozessor für Java ist *Xalan-Java 2* der Apache Software Foundation [Apa].

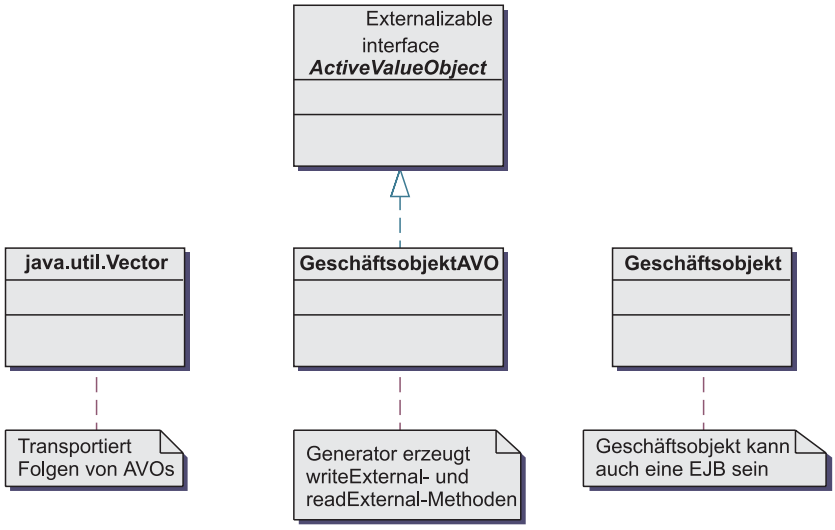


Abbildung 4.33: Einfache AVO-Implementierung

```
    Helper.write<Type>(out, this.attribut2);
    ...
    Helper.write<Type>(out, this.attributN);
  }
```

Das dargestellte Implementierungskonzept läßt sich zwar sehr schnell umsetzen verzichtet jedoch in dieser Form auf viele Konzepte, die bereits in dieser Arbeit vorgestellt wurden. Ein anspruchsvollerer Implementierungsansatz basiert auf einem ADC der als Sammelbehälter für eine Folge von AVOs dient, um ggf. noch weitere Optimierungen von Datenmenge und -komplexität zu ermöglichen. Prinzipiell kann hierfür bereits ein ADC aus Abschnitt 4.3 verwendet werden, der den Transport von beliebigen serialisierbaren Objekten mittels der Methoden `putSerObject()` und `getSerObject()` ermöglicht. An dieser Stelle soll jedoch ein neuer Daten-Container als Sammelbehälter verwendet werden, der seine Daten intern als Folge von Bytes (Typ `byte[]`) repräsentiert und diese Daten auf Wunsch zusätzlich mittels des Deflate-Algorithmus komprimieren kann. Die Idee basiert auf der Tatsache, daß die Komplexität dieser Datenstruktur am kleinsten ist und an die Kommunikationsschicht der verschiedenen Applikations-Server die geringsten Anforderungen stellt. Der Implementierungsansatz ist in Abbildung 4.34 skizziert.
Geschäftsobjekte, deren Daten transportiert werden sollen, müssen die Schnittstelle `BusinessObject` implementieren. Zu jedem Geschäftsobjekt existiert ein AVO, das die Schnittstelle `ActiveValueObject` implementiert. Die Implementierung des AVOs greift dabei auf die Klasse `Helper` zurück, die für das Lesen und Schreiben der Daten und eine evtl. mögliche Optimierung zuständig ist. Beide Klassen können mit dem bereits beschriebenen Generatoransatz erzeugt werden. Das AVO kann unverändert verwendet werden. Die generierte Klasse des Geschäftsobjekts kann als Grundgerüst für die weitere Implementierung dienen. Eine Folge von AVOs wird schließlich mittels der Klasse `AVOContainer` zur Übertragung zusammengefaßt. Die Methode `toValueObject()` in der Schnittstelle `BusinessObject` erlaubt dabei die

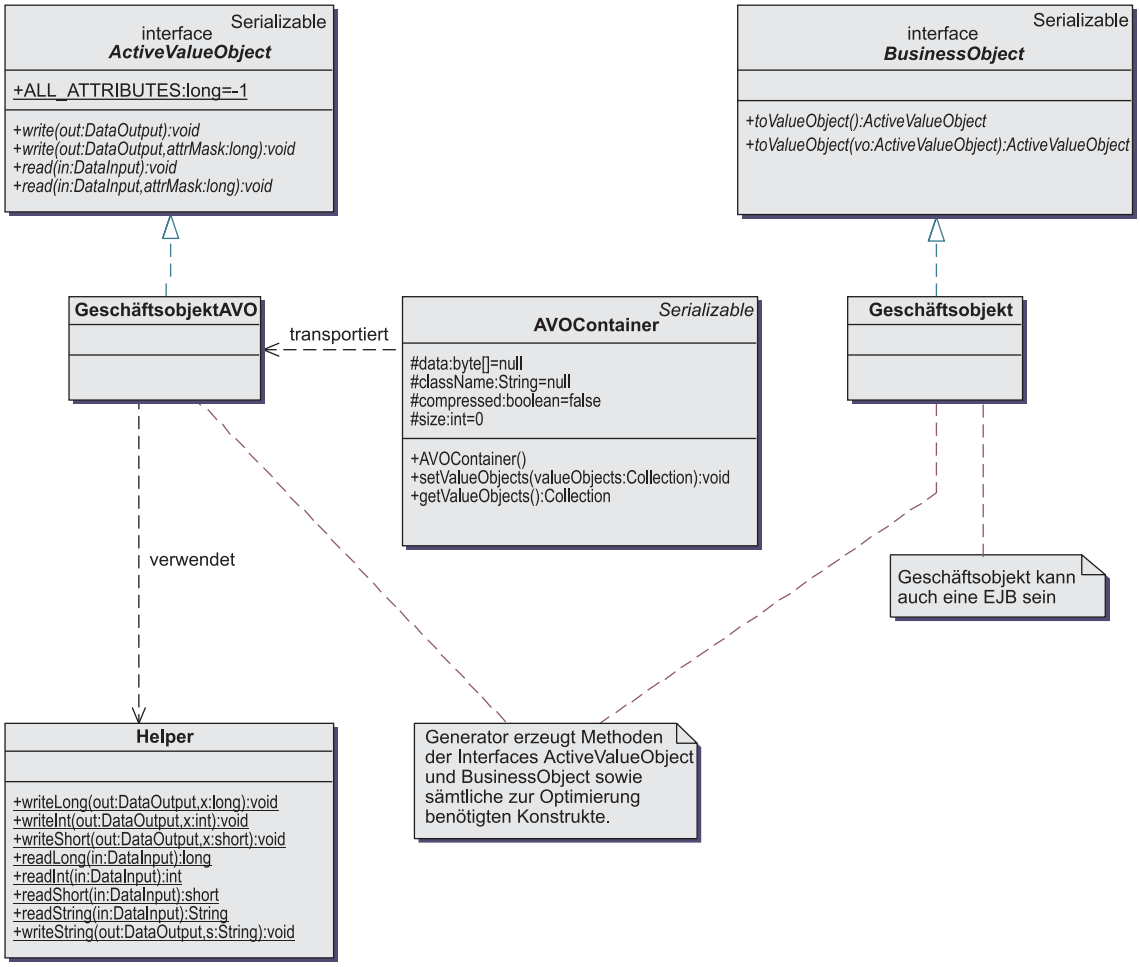


Abbildung 4.34: Umfassende AVO-Implementierung

Erzeugung von neuen AVOs, die mit den aktuellen Attributausprägungen des Geschäftsobjekts belegt sind, oder die Wiederverwendung von bestehenden AVOs, deren Attribute überschrieben werden. Der umgekehrte Weg kann z.B. mittels einer Methode `fromValueObject()` implementiert werden. Dabei ist sichergestellt, daß das AVO keine Informationen über das Geschäftsobjekt benötigt (das AVO wird auf dem Client verwendet). Das nachfolgende Codefragment skizziert den Aufbau der generierten Methode zur Wiederverwendung eines bestehenden AVOs:

```
public ActiveValueObject toValueObject(ActiveValueObject valueObject)
  {
    BussinessAVO vo = (BussinessAVO) valueObject;
    vo.setAttribut1 (this.attribut1);
    vo.setAttribut2 (this.attribut2);
    ...
    vo.setAttributN(this.attributN);
    return vo;
  }
```

In der Schnittstelle `ActiveValueObject` sind die Methoden zum Herstellen eines Datenformats zur Übertragung enthalten. Dabei wird auch berücksichtigt, daß gezielt Attribute zur Übertragung ausgewählt werden können. Im Gegensatz zu Abschnitt 4.3 werden hier die Attribute nicht als Folge von Strings angegeben, sondern zu jedem Attribut des AVOs wird eine `long`-Konstante der folgenden Form generiert:

```
public static final long _ATTRIBUT1 = 1;
public static final long _ATTRIBUT2 = 2;
...
public static final long _ATTRIBUTN = 2^N;
```

Diese Form erlaubt die Definition einer Attributliste durch eine ODER-Verknüpfung der korrespondierenden Attributkonstanten und ersetzt das sonst notwendige Vergleichen von Strings durch schnelle Bit-Operationen. Nachfolgend ist die `write`-Methode, die eine Attributliste entgegennimmt skizziert. Dabei wird davon ausgegangen, daß alle Attribute vom Typ String sind:

```
public void write(DataOutput out, long attrMask)
    throws IOException
{
  if ((attrMask & _ATTRIBUT1) != 0)
    Helpers.writeString(out, this.Attribut1);
  if ((attrMask & _ATTRIBUT2) != 0)
    Helpers.writeString(out, this.Attribut2);
  ...
  if ((attrMask & _ATTRIBUTN) != 0)
    Helpers.writeString(out, this.AttributN);
}
```

Die Attribute werden dabei durch die zur Optimierung vorgesehene Hilfsklasse `Helper` in einen für primitive Datentypen bereitgestellten binären Datenstrom vom Typ `DataOutput` (vgl. dazu [Mica]) geschrieben. Falls alle Attribute transportiert werden sollen existiert eine zweite Implementierung der Methode `write()`, die alle Attribute in den Datenstrom schreibt und auf die dann überflüssigen Vergleichsoperationen vollständig verzichtet. Die Übertragung aller Attribute wird mit der Konstante `ALL_ATTRIBUTES` im Interface `ActiveValueObject` kodiert. Der Datenstrom `out` wird vom `AVOContainer` bereitgestellt, der die gesamten Daten einer Folge von AVOs zur optimierten Übertragung aufnimmt. Der Container ruft hierbei bei Übergabe einer Folge von AVOs in Abhängigkeit davon ob alle Attribute oder nur ausgewählte übertragen werden sollen, die entsprechende `write`-Methode jedes Objekts auf. Nachfolgend ist die Methode zur Übergabe einer Folge von AVOs ohne Einschränkung skizziert:

```
public void setValueObjects(Collection valueObjects)
  {
    ...
    ByteArrayOutputStream out = ...
```

```
    DataOutputStream dataOut = null;
    DeflaterOutputStream zipOut = null;
    if (this.compressed)
    {
      zipOut = new DeflaterOutputStream(out,
          new Deflater(Deflater.BEST_SPEED));
      dataOut = new DataOutputStream(zipOut);
    }
    else
    {
      dataOut = new DataOutputStream(out);
    }
    ..
      while (valueObjects.hasNext())
      {
        ActiveValueObject vo =
           (ActiveValueObject) valueObjects.next();
        vo.write(dataOut, this.attrMask);
        ++this.size;
      }
      ...
     // Gesamtdaten als Byte-Array
     this.data = out.toByteArray();
      ...
  }
```

Auf die Darstellung der lesenden Zugriffe auf den AVO-Container und die Rekonstruierung der AVOs wird an dieser Stelle verzichtet. Die Implementierung ist analog zu den soeben erläuterten Konstrukten zu sehen. Statt Ausgabe-Streams werden Eingabe-Streams (`XInputStream`) verwendet und die `write`- Methoden werden durch `read`-Methoden ersetzt. In Abschnitt 4.5 werden Erweiterungen des ADC-Konzepts dargestellt, die auf diese Weise auch im Rahmen von AVOs zur Verfügung stehen. Die Verwendung mit EJBs und zusätzliche Erweiterungsmöglichkeiten werden in Kapitel 5 erläutert.

4.5 Erweiterungen

In diesem Abschnitt soll beschrieben werden, wie weitere Konzepte auf das universelle Datenübertragungskonzept aufgesetzt werden können. Diese Konzepte stellen Erweiterungen des Daten-Container-Konzepts dar, die nicht als Zusatzfunktionalität im Container selbst implementiert werden. Dies ist z.B. dann der Fall, wenn die Funktionalität nur auf dem Server angeboten werden soll oder so umfangreich ist, daß sie den Rahmen eines überschaubaren Daten-Containers sprengen. Im folgenden sollen zwei Beispiele für eine solche Erweiterung gegeben werden. In Abschnitt 4.5.1 wird ein Konzept für Java-Clients vorgestellt, die komplexe GUIs besitzen und die Aktive Daten-Container als Datenmodell verwenden. Abschnitt 4.5.2 dient als Beispiel dafür, wie die Daten-Container-Funktionalität auf dem Server erweitert wird. Als

Beispiel dient ein HTML-Dekorator, der die im Container transportierten Daten in eine HTML-Darstellung überführt. Es handelt sich dabei um die Umsetzung einer Form des Musters *Dekorator*, das die Funktionalität einer Klasse dynamisch erweitert. Dieses Muster wird auch als Umschlag (*Wrapper*) bezeichnet [Gam95].

4.5.1 GUI-Manager

In diesem Abschnitt wird beschrieben, wie die universellen Datenübertragungskonzepte auf Seite eines Java-Anwendungs-Client erweitert werden können, um die Programmierung zu vereinfachen und den Entwicklungsaufwand zu reduzieren. In [Bes98a, Bes98b] wird das Konzept eines Fenstermanagers vorgestellt, der die Verwaltung der Fenster und Dialoge einer Java-Anwendung oder eines Java-Applets übernimmt. Dieses Konzept wird nun entscheidend erweitert, indem der Manager zusätzlich Wissen über einzelne grafische Dialogelemente erhält und verwaltet.

Motivation

Die Entwicklung von anspruchsvollen Benutzeroberflächen stellt einen erheblichen Aufwand in der Entwicklung von modernen Client/Server-Anwendungen dar. Java stellt mit der Swing-Bibliothek [Mica] eine umfangreiche Sammlung von grafischen Elementen, die zur Anzeige und Eingabe von Anwendungsdaten verwendet werden können, zur Verfügung. Dabei müssen Daten, die zwischen Client und Server fließen sollen, ständig in die grafischen Elemente eingebracht oder entnommen werden. Die dazu notwendigen Routinen sind durch jeden Entwickler mühsam für seinen Bereich zu implementieren und zu pflegen. Dies stellt insbesondere dann eine Herausforderung dar, wenn es sich um komplexe grafische Elemente, wie z.B. eine Tabelle handelt. Das Entwurfsmuster GUI-Manager soll den Entwickler von diesen Aufgaben weitgehend entlasten und zu einer Einsparung von Codierungsaufwand führen, indem die Transportobjekte gleichzeitig als Datenmodell des Clients verwendet werden. Dies wird dadurch erreicht, daß der GUI-Manager Aktive Daten-Container entgegennimmt und die darin transportierten Daten automatisch in die dafür vorgesehenen GUI-Elemente einfügt. Umgekehrt können geänderte Daten wieder aus den GUI-Elementen entnommen werden und in einem Aktiven Daten-Container bereitgestellt werden.

Struktur

In Abbildung 4.35 ist die Grundstruktur des Konzepts dargestellt. Das Kernelement ist der GUI-Manager oder ein GUI-Builder. Der GUI-Manager wird z.B. zur Verwaltung von Anwendungsfenstern verwendet. Zu seinen Aufgaben gehört z.B.:

- Bereitstellung eines technischen Konzepts zur Darstellung der Fenster unabhängig von der Form, in der die Anwendung gestartet wurde: Eine Java-Anwendung kann z.B. wahlweise als Applikation von der Kommandozeile oder als Applet in einem Web-Browser gestartet werden. Beim Betrieb der Anwendung als Applet können z.B. alle Dialoge der Anwendung innerhalb des Browser-Fensters dargestellt werden, wogegen die Ausführung als Applikation zur Darstellung mehrerer Fenster auf dem Desktop führt. Eine andere

technische Umsetzung innerhalb der Applikation ist die Darstellung aller Dialoge innerhalb eines Hauptfensters. Die Client-Logik ist damit von den technischen Details einer Darstellung entkoppelt.

- Steuerung von Dialogfolgen: Um seine Aufgaben zu erledigen, muß der Benutzer oft verschiedene Dialoge durchlaufen, die seinen Arbeitsprozeß in der Anwendung abbilden. Anstatt die Dialoge untereinander zu verketten, können diese Informationen im Manager abgelegt und verarbeitet werden. Bei Änderungen der Reihenfolge oder Erweiterung der Arbeitsschritte, müssen diese Änderungen nur im Manager erfolgen.

- Weitere Dienste, die von vielen Dialogimplementierungen benötigt werden: Häufig wird weitere Funktionalität bereitgestellt, die durch die Dialoge genutzt werden kann. Ein Beispiel hierfür ist ein zentraler Mechanismus zum Austausch von Daten zwischen verschiedenen Dialogen.

Die Aufgaben eines nichtvisuellen GUI-Builders liegen in der Konstruktion von Dialogen und wird häufig in Frameworks zur Anwendungsentwicklung eingesetzt, um für eine bestimmte Anwendung spezialisierte GUI-Elemente, Fensterteile oder ganze Fenster per Programmanweisung zu erzeugen. Dabei wird versucht den Anwendungsentwickler von komplizierten Layoutfragen zu entlasten und keine Abhängigkeit zu Entwicklungswerkzeugen zu schaffen, die visuelle GUI-Builder besitzen und aus den konstruierten Dialogen Code erzeugen, der den Anforderungen der jeweiligen Anwendungsentwicklung nicht genügt. Die vom GUI-Builder erzeugten Fenster können mit Routinen, die den Aufgaben eines GUI-Managers gerecht werden, versehen werden.

Der GUI-Manager kann ADC-Objekte entgegennehmen und ordnet die enthaltenen Daten anhand eines Deskriptors einzelnen GUI-Elementen, oder Gruppen davon, zu. Bei der Manipulation von Daten in den GUI-Elementen werden diese als geändert gekennzeichnet und können vom GUI-Manager wiederum in einem ADC-Objekt zurückgegeben und zum Versand an den Server geschickt werden. Dabei kann eine Folge von Kommunikationsvorgängen, wie in Abbildung 4.35 entstehen. Eine erste Anfrage führt zur Lieferung eines ADC-Objekts, das die Daten zur Darstellung in das einzige GUI-Element von *Gruppe 1* enthält. Nachdem der Benutzer die Daten bearbeitet hat, wird vom GUI-Manager ein ADC mit den geänderten Daten bereitgestellt, der zum Server gesendet wird (2. Anfrage). Aus der zweiten Anfrage resultiert wiederum eine Reihe von Daten, die in die zwei GUI-Elemente der *Gruppe 2* gesetzt werden. Das GUI-Element von *Gruppe 3* repräsentiert Elemente, die prinzipiell auch dem GUI-Manager bekannt sind, hier aber manuell, durch eigene Routinen des Anwendungsentwickler mit Daten belegt werden. Bei Anwendung des GUI-Managers können nahezu sämtliche Codezeilen zur Darstellung von Daten in GUI-Elementen eingespart werden, da nur ein Deskriptor erstellt werden muß, der die Zuordnung zwischen Daten und GUI-Elementen definiert und ein Aufruf zur Belegung der GUI-Elemente mit Daten, bzw. Entnahme von Daten aus den GUI-Elementen notwendig ist. Insbesondere bei komplexen GUI-Elementen, wie z.B. Tabellen, tritt dadurch eine signifikante Entlastung des Entwicklers auf.

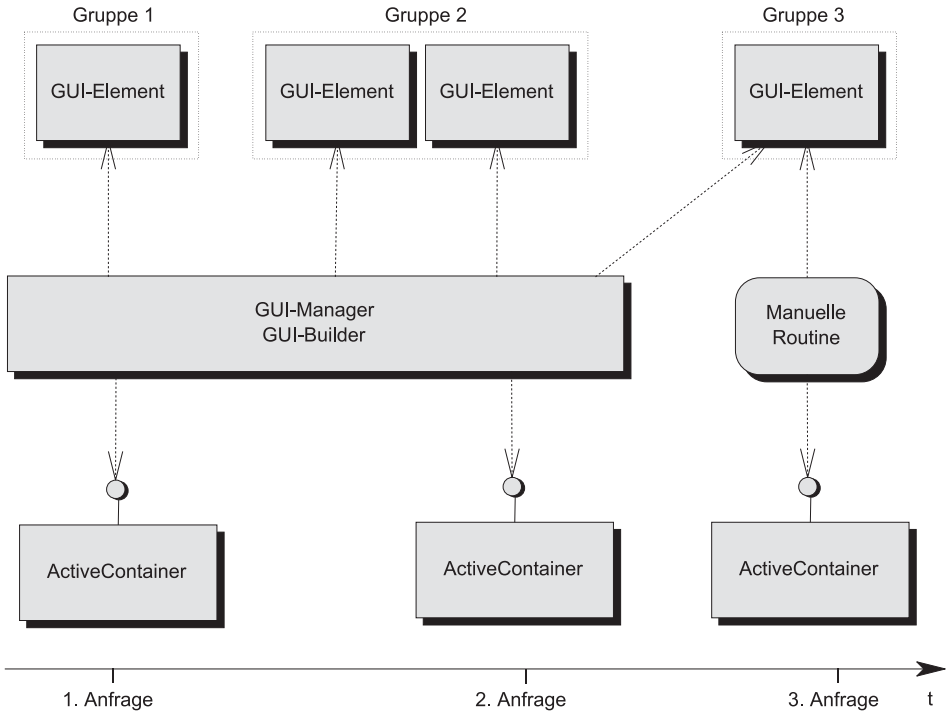


Abbildung 4.35: GUI-Manager

Implementierungsstrategien

Im Mittelpunkt der Implementierung des vorliegenden Konzepts steht die Entscheidung darüber, wie die Zuordnung der transportierten Daten auf die grafischen Elemente spezifiziert wird, wann diese Spezifikation zum Einsatz kommt und wo letztlich der Zugriff auf die GUI-Elemente erfolgt. In Abbildung 4.36 wird die Zuordnung grundsätzlich anhand eines Deskriptors aus logischer Sicht dargestellt.
Der Deskriptor definiert zunächst für jedes GUI-Element den Namen der Gruppe, zu dem es gehört. Durch mehrfache Vergabe des gleichen Namens entstehen so Gruppen, die aus mehreren potentiell unterschiedlichen Elementen bestehen können. Weiterhin wird spezifiziert unter welcher Kennung die Daten des GUI-Elements im ADC abgelegt sind, um eine Zuordnung beim Transport der Daten vieler GUI-Elemente vornehmen zu können. Logisch gesehen werden die Daten im ADC als Folge von ADC-Objekten abgelegt, die potentiell Server-Objekte repräsentieren. Im Anschluß daran wird definiert, welche Attribute zur Darstellung im GUI-Element verwendet werden sollen. Dies ermöglicht die Übertragung von Daten, die nicht dargestellt werden, aber trotzdem für weitere Bearbeitungsschritte benötigt werden. Dies ist z.B. für Primärschlüsselattribute der Fall, die für weitere Datenbankoperationen (Insert, Delete, Update) benötigt werden. Beim Schreiben der Daten durch einen GUI-Manager (`setValuesOf()`) werden durch Übergabe des Gruppennamens und des vom Server bezogenen ADC-Objekts die Deskriptoren herangezogen, um die zugehörigen Füllungsvorgänge anzustoßen. Beim Lesen von Daten findet der Vorgang umgekehrt statt. Anhand des übergebenen Gruppennamens wird der Deskriptor herangezogen, um aus GUI-Elementen entnommene Daten korrekt im verwen-

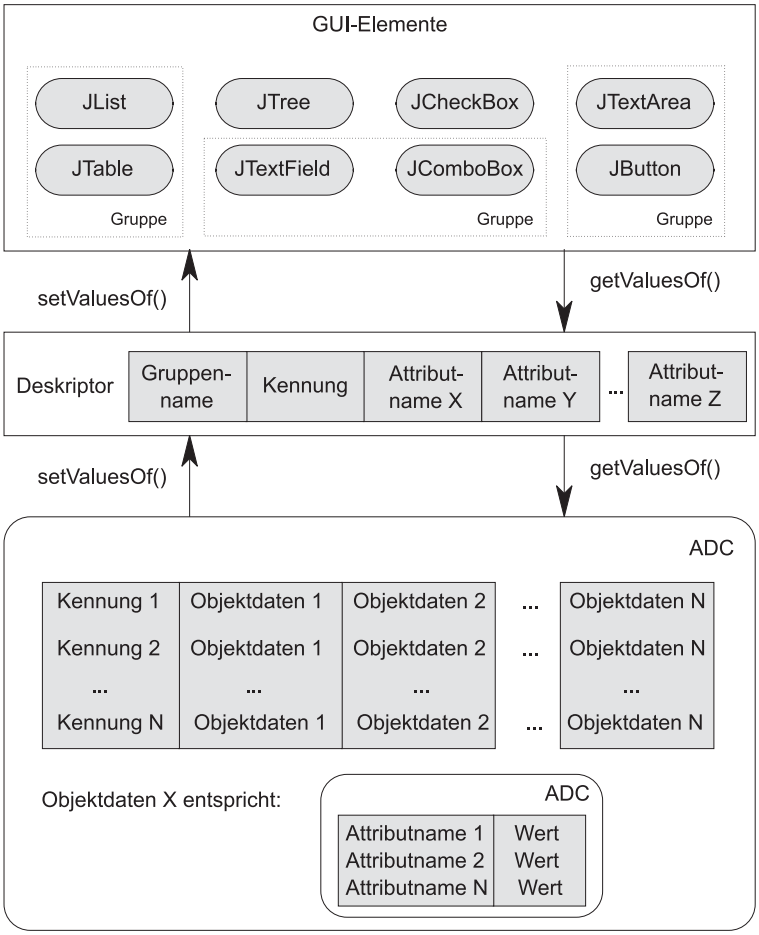


Abbildung 4.36: Zuordnung mittels Deskriptoren

deten ADC abzulegen und diesen anschließend zum Server zu schicken. Hierzu ist es sinnvoll, in die ADC-Objekte eine Kennung zu setzen, die darüber Auskunft gibt, ob es sich bei den Daten um neue, geänderte oder zu löschende Daten handelt. Dies ist u.a. bei komplexen GUI-Elementen, wie z.B. Tabellen sinnvoll, da hier Zeilen geändert, gelöscht und neu eingefügt werden können. Als Alternative können auch Routinen implementiert werden, die nur geänderte, gelöschte oder neue Daten zurückgeben. Das folgende Codefragment skizziert die soeben geschilderten Abläufe:

```
// Client
...
// Fordere Daten an
ADC daten=enterpriseBean.holeDaten();
guiManager.setValuesOf("meineGruppe", daten);

// Verarbeitung der Daten
...
// hole alle neu eingegebenen Daten
```

```
ADC daten2=guiManager.getValuesOf("meineGruppe", "insert");
enterpriseBean.uebergebeDaten(daten2);


// Server
ADC daten=new ADC();
...
// Suche Objekte
...
Vector treffer=session.executeQuery(...);

// Verpacke die Daten der gefundenen Objekte
daten.setObjects(treffer);
return daten;

// Erzeuge neue Objekte aus dem zurueckgegebenen ADC
Vector daten2.getObjects("kennung");

// Speichere die Objekte
...
```

Um das dargestellte Verhalten zu ermöglichen, muß der Inhalt des Deskriptors beim Aufrufen der `xxxValuesOf`-Methoden bekannt sein. Als Beispiel hierfür wird hier der Deskriptor direkt beim Erzeugen des GUI-Elements durch einen nichtvisuellen GUI-Builder übergeben, wie im folgenden Codefragment dargestellt:

```
// Client
String deskriptor="meineGruppe,
                   kennung,
                   attribut1,
                   attribut2,
                   ...,
                   attributN";

JComponent jc=guiBuilder.create("ComboBox", ..., deskriptor);

// Hinzufuegen des Elements zu einem Fenster (Panel)
panel.add(jc);
...
```

Häufig werden im Rahmen der Anwendungsentwicklung Java-Beans erzeugt, deren Verhalten parametrisiert ist und die von bestehenden GUI-Komponenten der Java-Bibliothek abgeleitet sind. Dabei kann z.B. mittels Parametern angegeben werden, welche Zeichen eingegeben werden dürfen und welcher Wertebereich für diese zulässig ist. Der Deskriptor kann dabei Bestandteil der übrigen Definitionen werden oder zusätzlich übergeben werden.
Falls ein visueller GUI-Builder mit projektspezifischen GUI-Elementen verwendet wird, kann der Deskriptor z.B. als *Property* einer Komponente definiert werden. Bei Verwendung von ein-

fachen Java-GUI-Elementen mit einem visuellen GUI-Builder muß dafür gesorgt werden, daß die Deskriptoren mit einer separaten Methode hinterlegt werden können:

```
// Client
myPanel.setDeskriptor("Name", deskriptor);
```

Dabei ist z.B. der Name des GUI-Elements mit dem entsprechenden Deskriptor zu übergeben. Die Funktionalität der Methode wird dabei durch eine Basisklasse zugesteuert, von der jedes Anwendungsfenster abgeleitet wird. Intern wird dabei eine Datenstruktur erzeugt, die alle GUI-Elemente und ihre Deskriptoren des Fensters enthält, um diese Informationen an den GUI-Manager bei Aufruf von `xxxValuesOf`-Methoden zur Verfügung zu stellen.

4.5.2 HTML-Dekorator

Der HTML-Dekorator dient als Beispiel dafür, wie auf Basis der Aktiven Daten-Container universell einsetzbare Erweiterungen auf dem Server erstellt werden können.

Motivation

Neben reinen Java-Anwendungs-Clients existieren noch eine Reihe anderer Clients, die auf eine EJB-Anwendung zugreifen müssen. Dazu gehören z.B. WWW-Browser, die auf eine HTML-Darstellung zurückgreifen, mobile Endgeräte, wie z.B. Handys, die mittels WAP-Browser WML-Seiten darstellen und Fremdsysteme, die ein XML-Format zum Datenaustausch benötigen. Um solche Systeme mit Daten zu versorgen, kann der Aktive Daten-Container gekapselt werden, um die Daten in der geforderten Form bereitzustellen. Die Kapsel kann aufgrund der Schnittstellen des ADC allgemein und konfigurierbar implementiert werden und so verschiedene Anforderungen hinsichtlich der Art und Menge der dargestellten Daten erfüllen.

Struktur

Die Struktur ist in Abbildung 4.37 dargestellt. HTML-Servlets verwenden den HTML-Dekorator, um eine HTML-Präsentation der Daten in einem Aktiven Daten-Container zu erhalten. Der HTML-Dekorator basiert auf der Schnittstelle des Aktiven Daten-Containers und erweitert dessen Funktionalität auf dem Server, um HTML-Clients den Zugriff auf die Daten zu ermöglichen.

Implementierungsstrategien

Zur universellen Implementierung des Dekorators kann er mit zwei Methoden versehen werden. Die `config`-Methode dient zur Konfiguration des Dekorators. Dabei können z.B. Darstellungsform und maximal dargestellte Datenmenge festgelegt werden. Diese Parameter können z.B. auch aus einem HTML-Formular entnommen werden, falls der Systembenutzer Einstellungen dieser Form vornehmen kann. Eine Konfiguration kann auch im Konstruktor der Dekorator-Klasse erfolgen. Die `config`-Methode ermöglicht jedoch die dynamische Konfiguration während des gesamten Lebenszyklus des Objekts und verhindert so, daß bei jeder Anfrage, die eine Parameteränderung nach sich zieht, ein neues Objekt erzeugt werden muß. In Abbildung 4.38 sind die beteiligten Objekte und ihre Interaktionen dargestellt.

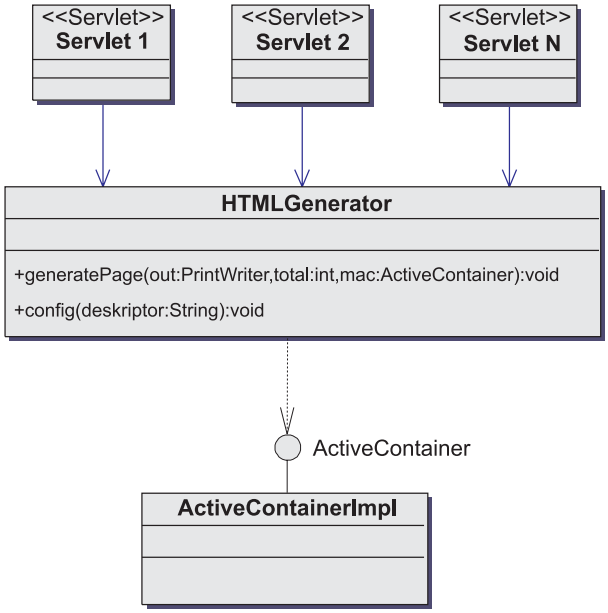


Abbildung 4.37: HTML-Dekorator

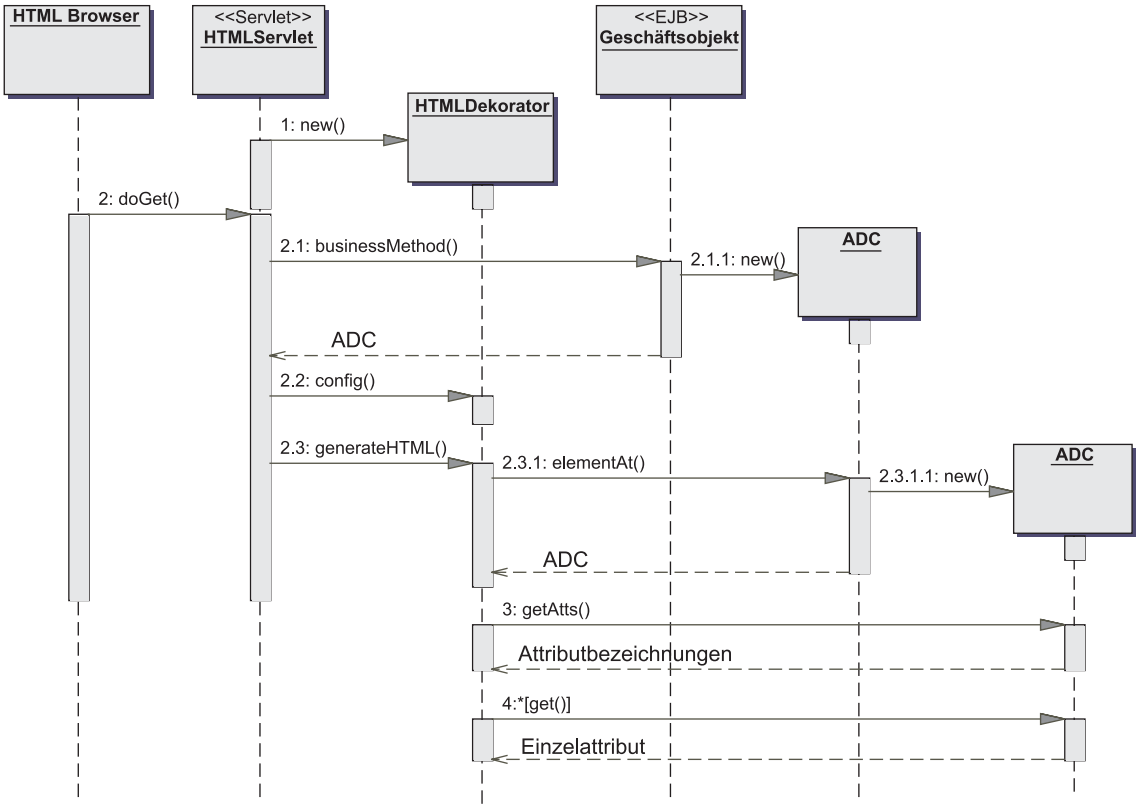


Abbildung 4.38: Interaktionen zwischen HTML-Dekorator und beteiligten Objekten

Um die Anfragen von Servlets entgegenzunehmen, erzeugt das HTML-Servlet zu Beginn seines Lebenszyklus ein HTML-Dekoratorobjekt (1). Die Anfrage eines HTML-Clients bewirkt den Aufruf der Methode `doGet()` des Servlets und übergibt dabei u.a. Parameter, die im Servlet zum Aufruf einer EJB-Geschäftsmethode führen (2.1). In der EJB wird ein ADC erzeugt, mit den angeforderten Daten gefüllt und an das Servlet zurückgeschickt (Rückgabe von 2.1). Entsprechend des Geschäftsprozesses wird das Dekorator-Objekt mittels der `config`-Methode konfiguriert, um die geforderte HTML-Darstellung zu liefern (2.2). Anschließend wird die `generateHTML`-Methode aufgerufen, um die HTML-Seite zu erzeugen (2.3). Innerhalb der Methode werden alle im ADC transportierten Daten in Form von ADC-Objekten durchlaufen (2.3.1, `elementAt`-Methode). Hierzu erzeugt der ADC entsprechende ADC-Objekte, falls die Daten zum Zwecke der Massendatenübertragung optimiert wurden (2.3.1.1) und gibt diese an das Servlet zurück (Rückgabe von 2.3.1). Das Servlet verwendet die ADC-Objekte als Stellvertreter für Geschäftsobjekte und fragt alle im ADC enthaltenen Attributbezeichnungen ab (3, `getAtts`-Methode), um diese anschließend mittels `get`-Methoden zu entnehmen (4). Der eigentliche Vorgang der HTML-Erzeugung ist in Abbildung 4.39 skizziert.

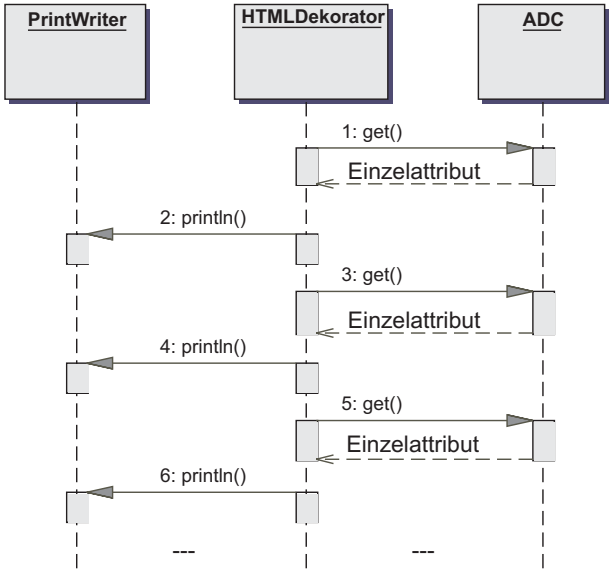


Abbildung 4.39: Generierung von HTML-Anweisungen im HTML-Dekorator

In der dargestellten Implementierung übernimmt die `generateHTML`-Methode auch den Versand der HTML-Seite zum Client. Hierzu besitzt die Methode den Daten-Container und das Stream-Objekt `PrintWriter` des Servlets, das die Verbindung zum Browser des Clients darstellt, als Parameter. Der Dekorator ruft anschließend die Daten mittels der Standardschnittstelle des Daten-Containers ab (1, 3, 5), fügt sie in das zugehörige HTML-Gerüst ein und schreibt die generierten HTML-Anweisungen in den HTML-Stream (2, 4, 6).
Falls mittels des HTML-Clients auch schreibende Zugriffe durch die Eingabe von Daten in entsprechende HTML-Formulare möglich sind und die Geschäfts-EJBs mit ADC-Objekten arbeiten, können aus dem HTML-Zugriff auch automatisch ADC-Objekte, die mit den Daten gefüllt

sind, erzeugt werden. Hierzu kann im ADC eine Routine zur generischen Analyse des vom Servlet bereitgestellten Anfrageobjekts `HttpServletRequest` verwendet werden, das die in der HTML-Seite übergebenen Parameter, enthält. Die Funktionsweise einer solchen Routine ist in Abbildung 4.40 exemplarisch dargestellt.

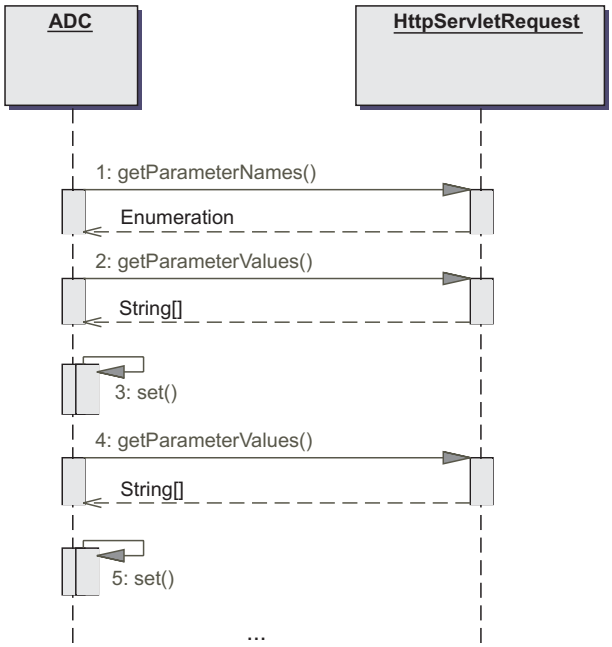


Abbildung 4.40: Erzeugung eines ADC aus einer HTML-Anfrage

Der ADC ruft mittels der Methode `getParameterNames()` die Namen aller übergebenen Parameter ab (1) und liest anschließend die Werte mittels der dafür vorgesehenen Methode `getParameterValues()` aus (2, 4), um sie anschließend in seiner internen Datenstruktur abzulegen (3, 5).

4.6 Zwischenergebnis

In diesem Kapitel wurden universelle Datenübertragungskonzepte in Form von Aktiven Daten-Containern entwickelt, die während des gesamten Lebenszyklus einer Anwendung an gegebene Anforderungen angepaßt werden können. Dieses Verhalten wird durch die Zerlegung der Datenübertragung in ihre verschiedenen Bestandteile (Aufgaben) und deren Parametrisierung erreicht. Wesentlich dabei ist die Trennung von Schnittstelle und Implementierung, die einen kompletten Austausch der Container-Implementierung im Laufe des Lebenszyklus einer Anwendung erlaubt, ohne deren Schnittstellen zu ändern. Insbesondere kann dabei die Datenstruktur geändert oder beeinflußt werden, um bei geänderten Anforderungen (z.B. geringere Bandbreite bei vergrößerter benötigter Datenmenge durch die Clients) reagieren zu können. Dynamische Datenübertragungskonzepte wurden dabei dadurch verbessert, daß eine einheitliche Schnittstelle existiert, die klar strukturierte Zugriffe auf transportierte Daten auch im Falle von Massendaten erlaubt. Zusätzlich wurden bei der Übertragung von Massendaten opti-

mierte Datenstrukturen verwendet, die sowohl die Menge, als auch die Komplexität der zu übertragenden Daten reduzieren. Bei beiden Faktoren handelt es sich um signifikante Einflußfaktoren auf das Leistungsverhalten einer EJB-Anwendung. Weiterhin wurde aufgezeigt, wie bestehende Entwurfsmuster mit den Daten-Containern verknüpft werden können. Dazu gehört z.B. das Entwurfsmuster *Dynamische Attribute*, das die zu übertragenden Attribute eines Objekts einschränkt und das Muster *Smart Proxy*, das hier beispielhaft bestimmte Attribute zum Server durchschreibt. Durch die Anwendung von Java-Reflection und die Nutzung anderer Objektschnittstellen wurden Wege aufgezeigt, wie im Rahmen der Datenübertragung Implementierungsaufwand eingespart werden kann. Dabei wurde auch ein Konzept zur Behandlung von Objektbäumen, die aus normalen Java-Objekten bestehen, vorgestellt. Als Alternative zum Auslesen und Beschreiben von Objekten mittels Reflection wurden auch neue Generatoransätze vorgestellt, die einer Programmbeschleunigung dienen. Durch die Umsetzung des ADC-Konzepts mit dem Entwurfsmuster *Aktive Value Objects* steht auch ein statisches Container-Konzept mit einem hohen Maß an Flexibilität sowie einer Einsparung von Implementierungsaufwand zur Verfügung, wenn die automatischen Lese-, Schreib- und Synchronisationsroutinen für Geschäftsobjekte verwendet werden. Statische Datenübertragungskonzepte wurden dabei ebenfalls durch eine festgelegte Schnittstelle erweitert, die eine allgemeine Parametrisierung und nachträgliche Konfiguration der Datenübertragung ermöglicht. Aufgrund der angestrebten Durchgängigkeit des Datenübertragungskonzepts auf alle Schichten einer Client/Server-Anwendung wurden in Form des GUI-Managers und des HTML-Dekorators Erweiterungen für grafische Anwendungs-Clients und HTML-Clients vorgestellt. Der GUI-Manager ermöglicht die Verwendung des ADCs als clientseitiges Datenmodell mit dem Ziel, Programmieraufwand zur Datenbelegung und Datenentnahme aus GUI-Elementen einzusparen und nachträglich abändern zu können. Darüber hinaus ermöglicht der HTML-Dekorator die Verwendung des ADCs mit Clients, die nicht auf Java basieren. Aufgrund der Standardschnittstelle kann diese Anbindung universell geschehen, was den Implementierungsaufwand gegenüber einer Lösung, die Daten individuell für jedes im System existierende Servlet aufbereitet, deutlich reduziert. Gleichzeitig wird wiederum erreicht, daß die Funktionalität zentral in der Dekorator-Klasse erweitert und gewartet werden kann.

Kapitel 5

Integration in Anwendungsarchitekturen

5.1 Zielsetzung

In dieser Arbeit besteht ein Hauptziel darin, eine möglichst hohe Flexibilität in Fragen der Datenübertragung zu erzielen. Um dies zu erreichen, werden in diesem Kapitel neue Vorschläge gemacht, wie Daten-Container in EJBs verwendet werden können. Die Vorschläge sind dabei komplementär zu den Eigenschaften der Datenübertragungskonzepte aus Kapitel 4 und stellen damit eine Vervollständigung zu einem Gesamtkonzept dar.

5.2 Verwendung in EJBs

In diesem Abschnitt wird erläutert, wie die universellen Datenübertragungskonzepte aus Kapitel 4 mit Entity-Beans, Session-Beans und Message-Driven-Beans verwendet werden können.

5.2.1 Bereitstellung

In dieser Arbeit wird als neue Vorgehensweise vorgeschlagen, benötigte Daten-Container einmalig bei der EJB-Initialisierung zu initialisieren, und die globalen Instanzen in allen Kommunikationsvorgängen nach Löschung des Inhalts wiederzuverwenden. Dabei wird die komplette Konfiguration der Datenübertragung mittels des Deployment-Deskriptors von EJBs ermöglicht und gleichzeitig der effiziente Umgang mit den Speicherressourcen des Servers gefördert. In [Sin97] wird die Erzeugung und automatische Speicherbereinigung von nicht mehr benötigten Objekten als signifikanter Einflußfaktor in einer Server-Anwendung identifiziert, die mit Java realisiert wird. Eine Verbesserung des Leistungsverhaltens kann danach durch die weitestgehende Vermeidung der Erzeugung neuer Objekte erzielt werden. In Abbildung 5.1 ist die Verwendung der Daten-Container in EJBs auf diese Weise skizziert.
Nachdem der Container eine Instanz einer EJB erzeugt hat (1), ruft er einmalig die Methode `setXXXContext()` auf, um der Bean den Kontext zu übergeben (2, vgl. auch Kapitel 2, Abschnitt 2.5). Die Methoden sind für den jeweiligen Bean-Typ in Tabelle 5.1 aufgelistet.
In diesen Methoden können nun ein oder mehrere Daten-Container erzeugt werden (2.1), die sich entsprechend den Anforderungen des Systems konfigurieren (2.2.2) und während des Lebenszyklus der EJB zur Kommunikation genutzt werden. Dieses Vorgehen stellt eine Optimie-

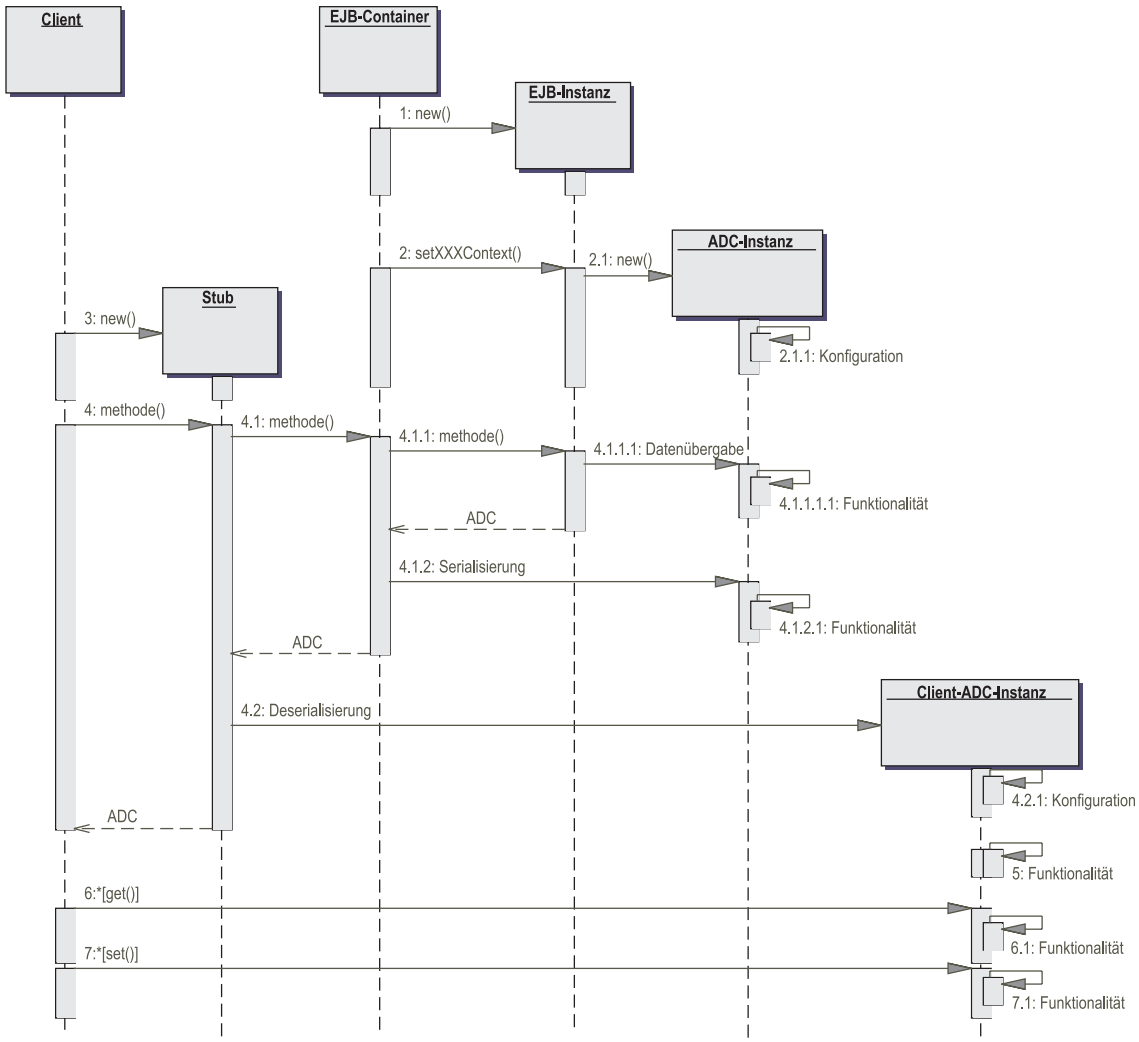


Abbildung 5.1: ADC-Verwendung in EJBs

Methodenname	EJB-Art
`setEntityContext()`	Entity-Bean
`setSessionContext()`	Session-Bean
`setMessageDrivenContext()`	Message-Driven-Bean

Tabelle 5.1: EJB-Methoden

rung dar, da die teure Objekterzeugung bei Kommunikationsvorgängen verhindert wird. Dies ist insbesondere dann der Fall, wenn der EJB-Container die EJB-Instanzen nach deren Erzeugung in einem Instanzen-Pool zur Wiederverwendung ablegt. Um eine Anfrage an eine EJB zu stellen, muß im Client ein *Stub*-Objekt erzeugt werden (3), um eine entfernte Methode aufrufen zu können (4). Der Methodenaufruf wird vom EJB-Container abgefangen (4.1) und unter dessen Kontrolle an eine dafür geeignete EJB-Instanz weitergeleitet (4.1.1). Im Rahmen der Methodenimplementierung, die in der Bean-Instanz realisiert ist, können Daten an die be-

reits erzeugt vorliegenden Daten-Container übergeben werden, die anschließend zum Client transportiert werden sollen (4.1.1.1). Dieser Vorgang kann eine Anwendung von Funktionalität, die im Daten-Container implementiert ist zur Folge haben (4.1.1.1.1). Die so entstandenen, gefüllten Daten-Container werden durch den EJB-Container im Rahmen des Kommunikationsvorgangs serialisiert (4.1.2), was bei entsprechender Implementierung der Daten-Container wiederum die Ausführung von Funktionalität (4.1.2.1) zur Folge haben kann. Clientseitig erfolgt im Rahmen der Deserialisierung (4.2) die Rekonstruktion der Daten-Container, wobei ebenso eine Konfiguration und Ausführung von Funktionalität möglich ist (4.2.1, 5). Am Ende dieser Vorgänge stehen die Daten-Container zur Benutzung im Client zur Verfügung und können ausgelesen und beschrieben werden (6, 7). Bei diesen Vorgängen kann bei entsprechender Implementierung der Daten-Container wiederum Funktionalität zur Anwendung kommen (6.1, 7.1). Um die Datenübertragung teilweise von der Anwendung abzukoppeln und auf Deployment-Deskriptor-Ebene zu konfigurieren, kann im Rahmen der `setXXXContext`-Methode die Bean-Umgebung ausgelesen werden, um näher zu spezifizieren, welche Daten-Container zur Kommunikation verwendet werden sollen und welche Funktionalität dabei zum Einsatz kommen soll. Somit kann z.B. nachträglich für bestimmte Beans entschieden werden, daß eine Kompression der Daten erfolgen oder eine andere Container-Implementierung verwendet werden soll. Das nachfolgende Codefragment deutet diese Vorgehensweise an:

```
public class MyBean implements XXXBean {
   private ActiveContainer ac;
   ...
   public void setXXXContext(EntityContext context)
   {
      this.context = context;
      ...
      // Hole die Bean-Umgebung
      Context initial = new InitialContext();
      Context environment =
         (Context) initial.lookup("java:comp/env");

      // Hole Container-Klasse und Funktion(en)
      String acClassName=
         (String) environment.lookup("ADC_Class");
      ...
      // Lade Container-Klasse und erzeuge eine Instanz
      Class contClass=Class.forName(acClassName);
      ac=(ActiveContainer) contClass.newInstance();

      // Konfiguriere Container und Funktion(en)
      ac.addFunction(function);
      ...
}
```

Alternativ können diese Parameter auch global für alle Beans konfigurierbar gemacht werden, indem ein Parameter-Objekt im JNDI des Applikations-Servers abgelegt wird, welches

die entsprechenden Werte enthält. Die Beans können auf dieses Objekt zugreifen und die Parameter entsprechend setzen. Eine weitere Möglichkeit ist das Setzen von Umgebungsvariablen beim Starten der virtuellen Maschine (vgl. dazu [Mica]). Der Austausch der Daten-Container-Implementierung ist aufgrund der Trennung zwischen Schnittstelle und Implementierung möglich, die Bestandteil des ADC-Konzepts ist. Der folgende Auszug aus dem Deployment-Deskriptor der Bean zeigt beispielhaft die Konfiguration der Container-Klasse und der zu verwendenden Funktionalität. I.d.R. besitzen die Applikations-Server grafische Tools, um die Deskriptoren zu bearbeiten. Damit wird der Konfigurationsvorgang visuell ermöglicht und stark vereinfacht. Das *Tag* <XXX> steht hier für die in Tabelle 5.2 aufgelisteten Bean-Arten.

Tag-Name	EJB-Art
`entity`	Entity-Bean
`session`	Session-Bean
`message-driven`	Message-Driven-Bean

Tabelle 5.2: EJB-Tags

```
<enterprise-beans>
   <XXX>
     ...
     <ejb-name>MyBean</ejb-name>
     ...
      <env-entry>
         <env-entry-name>ADC_Function</env-entry-name>
         <env-entry-type>java.lang.String</env-entry-type>
         <env-entry-value>zip</env-entry-value>
      </env-entry>
      <env-entry>
         <env-entry-name>ADC_Class</env-entry-name>
         <env-entry-type>java.lang.String</env-entry-type>
         <env-entry-value>ActiveContainer2</env-entry-value>
      </env-entry>
      ...
    </XXX>
    ...
```

Vererbungskonzept

Die in diesem Abschnitt erläuterten Konzepte können, wie in Abbildung 5.2 am Beispiel einer Entity-Bean dargestellt, durch Vererbung einer Basisklasse an die EJBs weitergegeben werden. Die abstrakte Basisklasse (`BaseEntityEJB`) implementiert dabei die EJB-Callback-Methoden so weit wie möglich und schirmt damit gleichzeitig Anwendungsentwickler vor der Auseinandersetzung mit diesen Methoden ab, sofern sie nicht ausdrücklich in den Lebenszyklus der betreffenden Bean eingreifen müssen. Methoden zum Lesen und Setzen von Daten-Containern sind hier in der Form von `getData()`, `setData()`, `getTreeData()`

und `setTreeData()` vorhanden. Sämtliche Bean-Methoden sind dabei in der Schnittstelle `BaseEntity` deklariert und in der Klasse `BaseEntityEJB` implementiert. Die Daten-Container werden dabei in der Methode `setEntityContext()` entsprechend initialisiert. In der abgeleiteten Entity-Bean `MyEntityBean` steht dem Entwickler dann das gesamte Konzept zur Verfügung. Um das Konzept von außen zugänglich zu machen, muß die Entity-Bean die Methoden der Schnittstelle `BaseEntity` ebenfalls anbieten. In der Abbildung wird dies durch die Schnittstellen `MyEntityBusiness` und `MyEntity` realisiert. Die `BaseEntity`-Schnittstelle vererbt ihre Methoden an die Schnittstelle `MyEntityBusiness`. Diese Schnittstelle deklariert zusätzlich die Geschäftsmethoden der Bean `MyEntity` und vererbt ihre gesamten Methoden schließlich an die eigentliche Remote-Schnittstelle `MyEntityBean`. Die Implementierungsklasse von `MyEntityBean` implementiert alle Operationen der Schnittstelle `MyEntityBusiness`, ohne gegen die Regel zu verstoßen, daß eine Bean-Implementierung nicht ihre Remote-Schnittstelle implementieren soll [Rom99]. Damit ist das Muster *Business-Interface* realisiert [Dee01]. Falls Entity-Beans nur lokal verwendet werden sollen, kann statt der Remote-Schnittstelle auch eine lokale Schnittstelle verwendet werden. Dabei ist statt `EJBObject` die Schnittstelle `EJBLocalObject` abzuleiten.

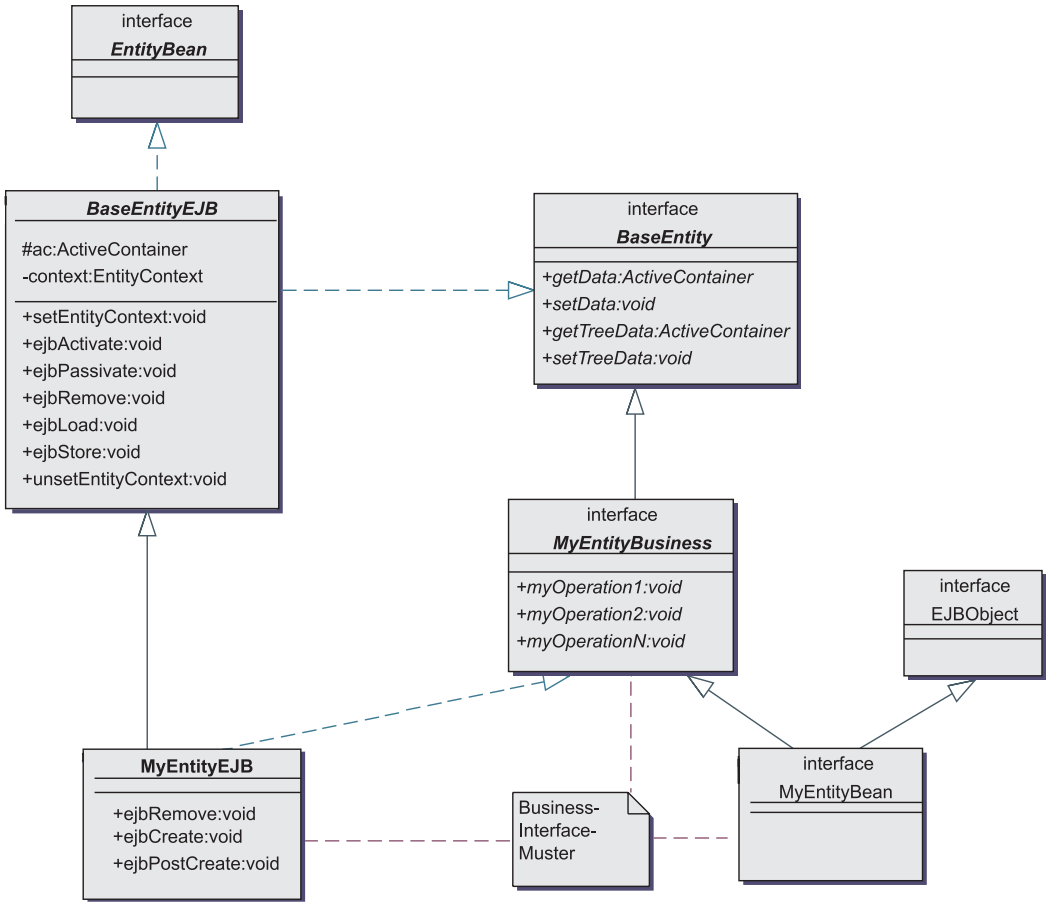


Abbildung 5.2: Verwendung ADC-Konzept durch Vererbung

Besonderheiten bei Entity-Beans

Beim Erzeugen von Entity-Beans durch den EJB-Container können mit Hilfe der Callback-Methode `ejbLoad()` direkt nach der Belegung der Attribute mit Daten aus dem Datenspeicher fertig befüllte Daten-Container bereitgestellt werden. Hierbei muß jedoch auf die dynamische Angabe der zu übertragenden Attribute beim Methodenaufruf verzichtet werden. Innerhalb der `ejbLoad`-Methode ruft die Entity-Bean die `setObject`-Methode des global erzeugten Daten-Containers mit einer Referenz auf sich selbst auf (`this`). Damit wird der Daten-Container mit allen Attributen der Entity-Bean-Instanz gefüllt. Bei Aufruf einer `getData`-Methode kann der gefüllte Daten-Container zurückgegeben werden.
Innerhalb einer `setData`-Methode kann die Entity-Bean-Instanz die `synchronize`-Methode des ADC mit einer Referenz auf sich selbst aufrufen. Dies führt zu einem automatischen Abgleich der im ADC transportierten Attributwerte mit den Attributwerten der Bean-Instanz. In der Literatur wird häufig gefordert, daß Transportobjekte nur zum Auslesen von Attributwerten auf dem Client verwendet werden sollen, da es sich im Grunde um die Kopie eines serverseitigen Objekts handelt, die mehrfach existieren und somit beim Schreiben zu Konflikten führen kann (vgl. dazu z.B. [Mye00, MH99]). Um trotzdem schreibende Zugriffe auf Daten-Container zu erlauben, müssen Konflikte dieser Art mittels Zeitstempeln oder Zählern [Dav99, Tea00] aufgelöst werden. Dabei wird ein Zeitstempel oder Zähler in der Datenbank und in den persistenten Objekten mitgeführt, der im Daten-Container gespeichert wird. Vor dem Schreiben wird der aktuelle Zeitstempel oder Zähler mit dem im Daten-Container vorhandenen verglichen. Bei Gleichheit ist sichergestellt, daß kein anderer Client zwischenzeitlich Änderungen vorgenommen hat. Sind die Kennzeichnungen unterschiedlich, liegt ein Konflikt vor, der z.B. dadurch gelöst werden kann, daß eine Ausnahme geworfen wird, die dem Client anzeigt, daß er den aktuellen Arbeitsschritt nochmals mit aktuellen Daten durchführen muß. Die Integration eines solchen Konzepts zur Konfliktauflösung kann problemlos mittels ADC erfolgen, da evtl. im Objekt vorhandene Zeitstempel oder Zähler automatisch erfaßt werden. Aufgrund des Konzepts des ADCs, selbst Aktivitäten auf den transportierten Daten durchzuführen, kann der Vergleich von Zeitstempeln oder Zählern im Daten-Container erfolgen und eine Synchronisation mit den Bean-Attributen zurückgewiesen werden (z.B. durch Erzeugung einer entsprechenden Exception). Dieser Mechanismus steht nach einmaliger Implementierung automatisch bereit und wird systemweit durchgesetzt.

Besonderheiten bei Session-Beans

Bei der Verwendung von zustandsbehafteten Session-Beans muß entschieden werden, ob global definierte Daten-Container bei einer Passivierung durch den EJB-Container (vgl. dazu Kapitel 2, Abschnitt 2.5) im Sekundärspeicher abgelegt werden müssen. Eine Ablage muß erfolgen, falls die Daten-Container Zustandsinformationen des Clients enthalten. Ist dies nicht der Fall, kann eine unnötige Sicherung der ADCs durch Angabe von `transient` verhindert werden. Es muß dann allerdings dafür gesorgt werden, daß sie bei einer erneuten Aktivierung wieder erzeugt werden. Zu diesem Zweck kann die dafür in Session-Beans vorgesehene Methode `ejbActivate()` verwendet werden, die vom EJB-Container bei jedem Aktivierungsvorgang aufgerufen wird. Eine mögliche Alternative zu diesem Vorgehen ist es, auf `transient` zu verzichten und bei der Passivierung durch den EJB-Container in der `ejbPassivate()`-

Methode der Session-Bean den Inhalt der Daten-Container mittels deren `clear`-Methode zu löschen. Das Halten von Zustandsinformationen in einem ADC kann sinnvoll sein, falls eine Anfrage eines Clients eine umfangreiche Treffermenge verursacht hat, die nach Bedarf stückweise zum Client übertragen wird (Iterator-Muster [Gam95]). Das Belegen des Containers mit Daten und die Datenentnahme wird in den jeweiligen Geschäftsmethoden realisiert. Dabei werden die dafür vorgesehenen Hinzufügeoperationen des ADC verwendet.

Besonderheiten bei Message-Driven-Beans

Message-Driven-Beans ermöglichen die asynchrone Kommunikation durch Entkopplung mittels JMS-Mechanismen (vgl. dazu Kapitel 2, Abschnitt 2.5). Ein JMS-Client stellt dabei Nachrichten in eine Warteschlange, die von MDBs konsumiert werden. Es liegt in der Natur der asynchronen Kommunikation, daß keine unmittelbare Rückgabe erfolgt. Zudem besitzen EJBs von dieser Art nur eine Geschäftsmethode `onMessage()`, die serialisierbare Objekte vom Typ `Message` entgegennimmt. Die Kommunikation vom JMS-Client hin zum Applikations-Server erfolgt damit durch das Versenden von solchen Nachrichten. ADC-Objekte, die manuell oder im Rahmen der `setMessageDrivenContext`-Methode erzeugt werden, können damit nur zur Weiterverarbeitung von Nachrichten verwendet werden, in deren Folge Session- oder Entity-Beans kontaktiert werden müssen, die solche Objekte als Übergabeparameter entgegennehmen. D.h. die Message-Driven-Bean tritt in diesem Fall selbst als Client anderer EJBs auf.
ADC-Transportobjekte können jedoch zur Kommunikation zwischen JMS-Client und JMS-Dienst auf dem Applikations-Server genutzt werden. Hierzu kann das nachfolgend beschriebene Integrationskonzept verwendet werden. Mit Hilfe der JMS-Dienste kann ein Message-Objekt vom Typ `ObjectMessage` erzeugt werden, das ein serialisierbares Anwendungsobjekt enthalten kann. Das folgende Code-Fragment skizziert das Konzept für ADC-Objekte:

```
...
// Client-seitig ist ein gefuelltes ADC-Objekt enthalten
ActiveContainer ac=new ActiveContainerImpl();
ac.set(Schluessel, Wert);
...

// Erzeuge eine Objektnachricht mit der JMS-Session
// und fuege das ADC-Objekt ein.
ObjectMessage message=session.createObjectMessage(ac);
```

Beim Empfang der Nachricht durch eine Message-Driven-Bean auf dem Server wird das ADC-Objekt wieder extrahiert und verarbeitet:

```
// Server-seitige onMessage-Methode
//
ObjectMessage om=(ObjectMessage) message;
ActiveContainer ac=(ActiveContainer) om.getObject();

// verarbeite den Inhalt
```

```
Typ name=ac.get(Schluessel);
...
```

Aufgrund der Flexibilität des Daten-Containers können neben den eigentlichen Anwendungsdaten sämtliche Steuerungsinformationen, die festlegen, was für Aktionen durch die Nachricht ausgelöst werden müssen, strukturiert in ihm abgelegt werden. Zusätzlich können Informationen für ein Berechtigungskonzept, das momentan nicht von Message-Driven-Beans angeboten wird, im Daten-Container transportiert werden. Dazu gehört z.B. der Benutzername und ein Paßwort, das z.B. mittels Zusatzfunktionalität verschlüsselt werden kann. Durch die Verwendung des ADCs als Nachrichtenformat können z.B. die Daten für einen größeren Vorgang strukturiert gebündelt werden und in einem Aufruf zum Server geschickt werden. Dabei besteht zusätzlich wieder das Potential, eine allgemeine Message-Driven-Bean zu implementieren, die eine Auswertung der Daten vornimmt (entsprechend dem Dekorator-Prinzip, das in Kapitel 4, Abschnitt 4.5 vorgestellt wurde).

5.2.2 Remote-Schnittstellen

Zur Verwendung der Datenübertragungskonzepte aus Kapitel 4 können Session- und Entity-Beans grundsätzlich mit der in Abbildung 5.3 dargestellten Schnittstelle versehen werden.

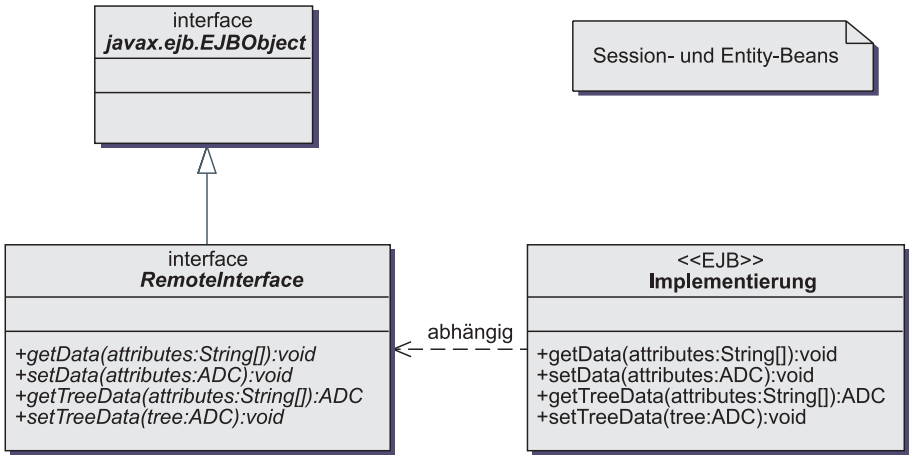


Abbildung 5.3: Aktive Daten-Container in Session- und Entity-Beans

Die abgebildeten Standardmethoden dienen dabei den folgenden Zwecken:

- **Methoden** `xxxData()`: Die Methoden eignen sich zur Übertragung des Zustands von EJBs. Die `getData()`-Methode erlaubt die Übergabe einer Folge von Attributnamen, deren Werte anschließend in einem ADC zurückgegeben werden. Umgekehrt erlaubt die `setData()`-Methode die Übergabe eines ADCs mit geänderten Attributen zum Schreiben. Dieses Vorgehen ist vor allem für Entity-Beans, aber auch für Session-Beans, die mit Attributen ausgestattet werden, geeignet. Eine Anpassung dieses Vorgehens besteht bei Session-Beans darin, neben einer Attributliste auch eine Kommandoliste zu übergeben. Die Session-Bean führt die den Kommandos entsprechenden Aktionen durch und

füllt den ADC mit den Ergebnissen. Gegenüber bestehenden Verfahren kann mit einer Methode jede beliebige Attributkombination übertragen werden, ohne dafür eine eigene Übertragungsklasse zu implementieren. Dabei wird die übertragene Datenmenge insbesondere bei sehr vielen Attributen stark eingeschränkt, falls im jeweiligen Anwendungskontext nur eine Teilmenge benötigt wird. Die Datenmenge und -komplexität kann im Daten-Container auch nachträglich noch reduziert werden. Zusätzlich wird das Füllen des Daten-Containers mit einem Methodenaufruf, unabhängig von der Anzahl der zu transportierenden Attribute geleistet. Die Implementierung der Methode besteht dabei lediglich darin der ADC-Instanz mittels der `setObject`-Methode eine `this`-Referenz, also die Bean-Instanz selbst, zu übergeben:

```
public ActiveContainer getData(String[] attributes)
{
       return activeContainer.setObject(this, attributes);
}
```

Das Zurückschreiben eines geänderten Daten-Containers kann mittels einer Methode `setData()` erfolgen, die dafür sorgt, daß die im Daten-Container transportierten Attribute mit den Attributen der Bean-Instanz abgeglichen werden. Die Implementierung der Methode enthält das nachfolgende Code-Fragment:

```
public void setData(ActiveContainer attributes)
{
       attributes.synchronize(this);
}
```

- **Methoden** `xxxTreeData()`: Falls die EJBs Java-Objekte referenzieren, kann der in Kapitel 4 beschriebene Algorithmus zum Durchlaufen von Objektbäumen zum Einsatz kommen. Gegenüber bestehenden Verfahren ist es möglich durch einen Methodenaufruf die Daten eines kompletten Objektbaumes oder einzelne Attribute daraus zu verarbeiten. Die Objekte benötigen dafür keine speziellen Methoden oder zusätzliche Container-Klassen. Weiterhin kann wiederum eine Reduktion der Datenmenge und -komplexität erfolgen. Die Implementierung der Methoden ist sehr einfach, da die Komplexität des Algorithmus im ADC gekapselt ist:

  ```
  public ActiveContainer getTreeData(String[] attributes)
  {
        activeContainer.setTreeData(this, attributes);
  }
  ```

  Die Bean übergibt der `setObject`-Methode des ADCs eine Referenz auf sich selbst und die vom Client angeforderten Attribute. In der Methode wird dann der Objektbaum rekursiv durchlaufen, um die geforderten Attribute zu beschaffen. Im Falle des Schreibens erfolgt die Synchronisation der Attribute mit der `synchronize`-Methode des ADC:

```
    public void setTreeData(ActiveContainer adc)
    {
          adc.synchronize(this);
    }
```

Unabhängig von diesen Methoden, kann der ADC in beliebigen Geschäftsmethoden der Remote-Schnittstellen als Übergabe- und Rückgabeparameter verwendet werden. Dabei ist zu beachten, daß die verallgemeinerten Schnittstellen der Daten-Container eine schwächere Typisierung nach sich ziehen, die auch die Aussagekraft von Remote-Schnittstellen reduziert. Eine Schnittstelle der Form:

```
ActiveContainer eineMethode(ActiveContainer data)
   throws RemoteException;
```

ist sehr flexibel und erlaubt potentiell die Übertragung von Daten verschiedenster Art und Quelle. Sie hat dafür aber weniger Aussagekraft, als die Verwendung von primitiven und zusammengesetzten Datentypen. Dieser Nachteil wird durch eine sorgfältige Dokumentation ausgeglichen, die z.B. die folgende Form besitzt:

```
/*
 * Suchfunktion
 * Methode nimmt Suchparameter im ADC entgegen:
 * parameter1 ( Typ1 ),   Beschreibung
 * parameter2 ( Typ2 ),   Beschreibung
 * ...
 * parameterN ( TypN ),   Beschreibung
 *
 * Suchergebnisse werden im ADC zurueckgegeben:
 * kennung1: attribut1 ( Typ1 ),   Beschreibung
 *           attribut2 ( Typ2 ),   Beschreibung
 *           ...,
 *           attributN ( typN ),   Beschreibung
 *
 * kennung2: attribut1 ( Typ1 ),   Beschreibung
 *           attribut2 ( Typ2 ),   Beschreibung
 *           ...,
 *           attributN ( typN )    Beschreibung
 *
 * kennungN: attribut1 ( Typ1 ),   Beschreibung
 *           attribut2 ( Typ2 ),   Beschreibung
 *           ...,
 *           attributN ( typN )    Beschreibung
 */

ActiveContainer suchMethode(ActiveContainer data)
   throws RemoteException;
```

Die Dokumentation stellt keinen Mehraufwand dar, da in jedem Fall im Rahmen der Systementwicklung eine ausführliche Dokumentation der Remote-Schnittstelle, ihrer Methoden und den darin verwendeten Parametern erfolgen sollte. Darüber hinaus sind beim Aufbau weiterhin Überlegungen zum Schnittstellenentwurf angebracht. So kann es am Beispiel der oben dargestellten Suchmethode sinnvoller sein, die Suchparameter streng typisiert zu übergeben, falls das Hauptproblem bei der Übertragung der Treffermenge liegt:

```
ActiveContainer suchMethode(Typ1 parameter1,
                            Typ2 parameter2,
                             ...,
                            TypN parameterN)
   throws RemoteException;
```

D.h. der ADC wird als Übergabeparameter nur dann verwendet, wenn eine größere Datenmenge vom Client zum Server transportiert werden muß. Dies ist z.B. dann der Fall, wenn Daten in einer Tabelle bearbeitet und anschließend gesammelt zum Server übertragen werden sollen.
Eine weitere Überlegung beim Design der Remote-Schnittstelle besteht darin, den Client selbst entscheiden zu lassen, unter welcher Kennung er Daten im ADC übertragen möchte. Hierzu werden die Kennungen als Übergabeparameter verwendet:

```
ActiveContainer suchMethode(String kennung1,
                            String kennung2,
                            ...,
                            String kennungN,
                            Typ1 parameter1,
                            Typ2 parameter2,
                            ...,
                            TypN parameterN)
   throws RemoteException;
```

Dies führt in erster Linie zu einer Entkopplung zwischen Client und Server, erfordert aber die Aufnahme von technisch motivierten Parametern in eine fachliche Methode. Der Vorteil der Entkopplung ist dabei allerdings höher zu bewerten, da die Clients unabhängiger von den Server-Komponenten werden und der Umgang mit dem ADC verbessert wird. Die Lesbarkeit der Schnittstellen kann auch erhöht werden, wenn zur Übertragung von Massendaten ein spezieller Container implementiert wird. Der Aktive Daten-Container wird dann nur als Stellvertreter für ein Objekt verwendet. Das folgende Code-Fragment veranschaulicht dies:

```
// Uebertragung mehrerer Objektzustaende
MultiActiveContainer suchMethode( Typ1 parameter1,
                                  Typ2 parameter2,
                                  ...,
                                  TypN parameterN)
   throws RemoteException;

// Uebertragung eines Objektzustands
ActiveContainer getData( Typ1 parameter1,
```

```
                        Typ2 parameter2,
                         ...,
                        TypN parameterN)
   throws RemoteException;
```

Der Typ `MultiActiveContainer` ist der Übertragung von Daten mehrerer Objektzustände vorbehalten. An seiner Schnittstelle können die einzelnen Objektzustände jeweils als ADC repräsentiert werden.

5.3 Verwendung in EJB-Architekturen

Nachfolgend wird erläutert, wie durch die Berücksichtigung von EJB-Architekturansätzen mit Aktiven Daten-Containern eine weitere Vereinfachung der Implementierung erreicht werden kann.

5.3.1 Architekturansätze

In Abbildung 5.4 sind verschiedene Architekturansätze, wie sie bereits in Abschnitt 3 ausführlich erläutert wurden, zusammengefaßt.

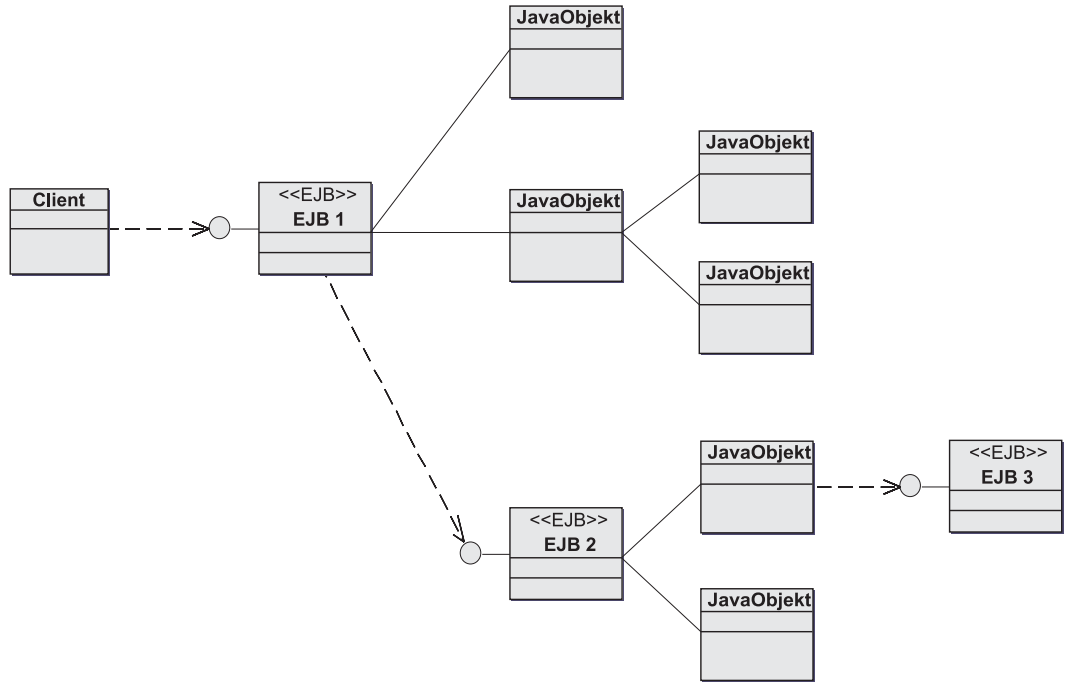


Abbildung 5.4: Architekturansätze

Wie bereits in Abschnitt 5.2 erläutert, können Session- oder Entity-Beans mit einer `getTree`-Methode ausgestattet werden, die das durchlaufen eines Objektbaums gemäß des in Kapitel 4, Abschnitt 4.3 erläuterten Algorithmus erlauben. Damit können bereits Entwurfsansätze, die auf

abhängigen Objekten beruhen abgedeckt werden. In der Abbildung besitzt z.B. die `EJB 1` eine Reihe abhängiger Java-Objekte, die wiederum auch andere Java-Objekte referenzieren können. Diese Struktur existiert real z.B. in Form des umgesetzten Musters *Aggregate Entity* (vgl. Kapitel 3). In diesem Fall können die Daten des kompletten Objektbaums oder Teile davon mit einem einzigen Methodenaufruf entnommen bzw. hinzugefügt werden. Dabei muß der jeweiligen Methode des Daten-Containers nur das Wurzelobjekt des Objektbaums (hier EJB 1) übergeben werden. Nun wird die Funktionsweise des Java-Serialisierungsmechanismus nachgebildet. Die Server-Objekte werden dabei jedoch vor Clients verborgen und es kann zur Laufzeit entschieden werden, welche Attribute der einzelnen Objekte im Baum übertragen werden sollen. Die abhängigen Objekte benötigen keinerlei zusätzliche Methoden um den Datenaustausch für bestimmte Anwendungsfunktionen zu ermöglichen. Ebenso wird keine zusätzliche Objektschicht aus *Value Objects* benötigt, die den Entwicklungsaufwand ebenfalls vergrößern würden. Die Reichweite dieses Einsparungseffekts kann noch weiter ausgedehnt werden, indem weitere typische Architekturkonzepte von EJB-Anwendungen berücksichtigt werden. Weitere Konzepte sind in der Beziehung zwischen `EJB 1` und `EJB 2` zu sehen. Sie sind für die Realität typisch, wenn z.B. Entity-Beans so modelliert werden, daß sie zueinander in Beziehung stehen (z.B. „Kunde“ hat „Adresse“) oder das Muster *Fassade* umgesetzt wird. Ebenso ist es auch möglich, daß ein abhängiges Java-Objekt mit einer weiteren EJB (hier `EJB 3`) in Beziehung steht. Im folgenden Abschnitt wird erläutert, wie der Algorithmus zur Baumanalyse erweitert wird, damit auch Referenzen zwischen EJBs und zwischen Java-Objekten und EJBs berücksichtigt werden. Damit können sämtliche Methoden und *Value Objects* eingespart werden, die für einen Datenaustausch mit den einzelnen Baumelementen erforderlich wären. Sie werden durch den ADC und die bereits vorhandenen `xxxTree`-Methoden überflüssig.

5.3.2 Erweiterte Aktive Container-Konzepte

Zur Berücksichtigung von Objektbäumen, die auch EJBs enthalten, wird der Algorithmus im ADC so erweitert, daß entschieden werden kann, ob es sich bei einem analysierten Attribut um eine Referenz auf eine EJB handelt. Um dies festzustellen, muß bekannt sein, wie der verwendete Applikations-Server diese Referenzen repräsentiert. Es kann sich dabei z.B. um die Implementierung der Remote-Schnittstelle durch ein *Stub*-Objekt handeln. Auf dieser Basis können die `xxxTree`-Methoden selbständig vom Daten-Container aufgerufen werden, was die Grundvoraussetzung zum automatischen Durchlaufen des Baums darstellt. Als Alternative zu diesem Vorgehen kann eine ohnehin sinnvolle Namenskonvention eingeführt werden, die besagt, daß Attribute, die Referenzen auf EJBs repräsentieren bei der Namensgebung die Endung `EJB` bekommen. Somit können solche Attribute bei der Implementierung und Wartung des Programms von herkömmlichen Attributen besser unterschieden werden. Zusätzlich kann diese Namenskonvention durch den Algorithmus zur Unterscheidung zwischen EJB-Referenzen und herkömmlichen Referenzen genutzt werden. Anhand des in Abbildung 5.5 dargestellten Beispiels soll die Funktionsweise des auf EJBs erweiterten Algorithmus erläutert werden.
Dargestellt sind vier Geschäftsobjekte `GO_X`, die als EJBs implementiert sind. Jede EJB besitzt eine `getTree`-Methode, die den Teilbaum dieser Bean in Form eines ADC liefert. Der ADC selbst wird nicht nur zur Kommunikation zwischen den Beans verwendet, sondern kapselt auch mittels der Methode `setTreeData()` den Algorithmus zur Baumanalyse. Die EJB `GO_1`

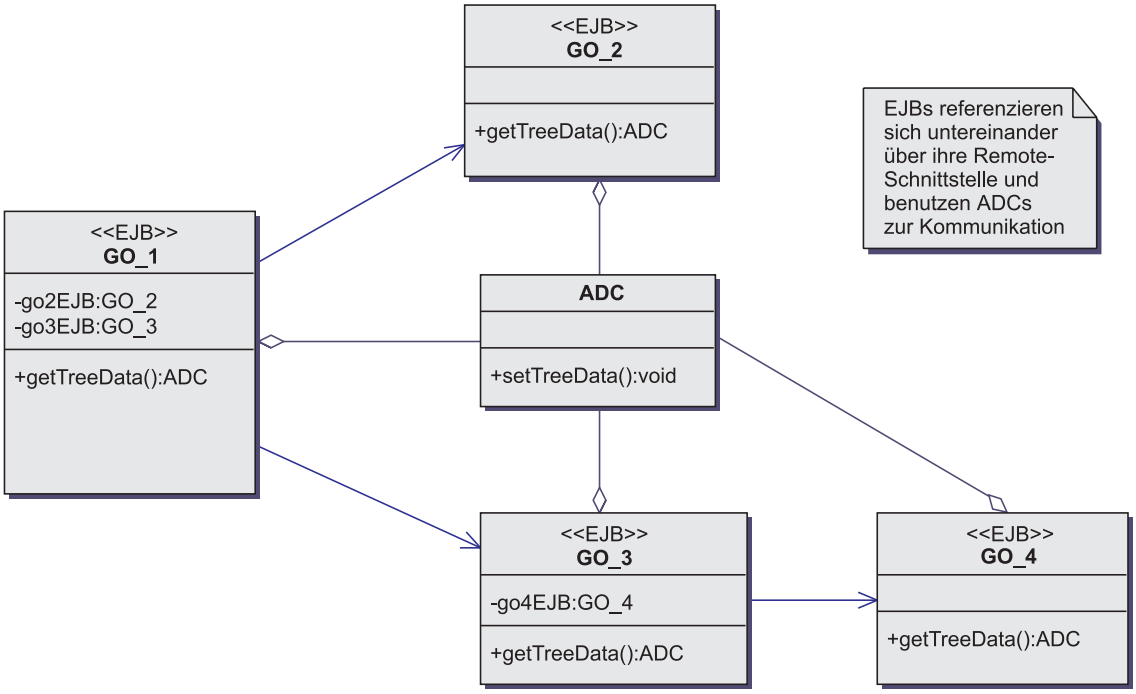


Abbildung 5.5: Berücksichtigung von EJB-Bäumen

referenziert die EJBs `GO_2` und `GO_3`. Die EJB `GO_3` referenziert ihrerseits die EJB `GO_4`. Ein Aufruf der `getTree`-Methode von `GO_1` führt durch die in Abbildung 5.6 dargestellten Interaktionen zum Auslesen aller EJB-Daten.

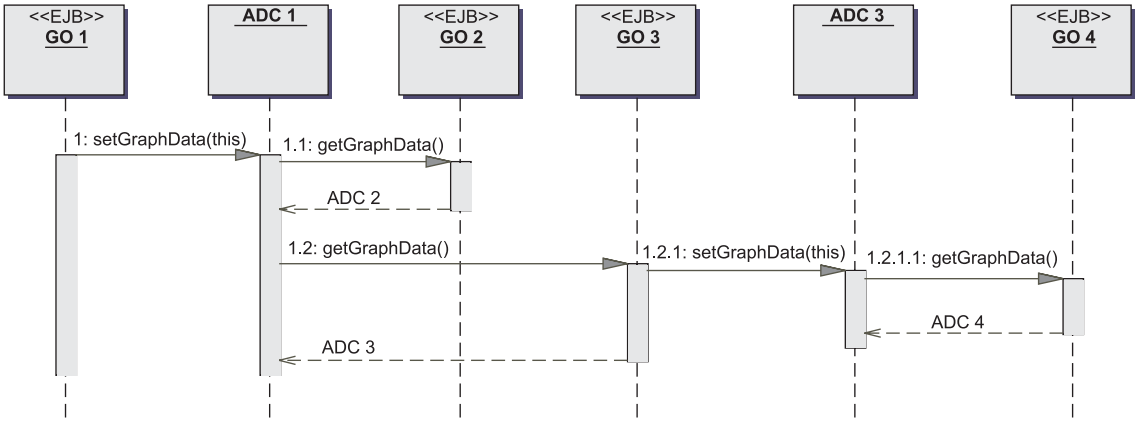


Abbildung 5.6: Interaktionen zwischen EJBs

Die EJB `GO_1` ruft die `setTreeData` mit einer Referenz auf sich selbst auf (1). Daraufhin werden die Attribute der EJB durch den ADC analysiert und komplexe Objekte rekursiv durchlaufen. Der Algorithmus findet dabei zwei EJB-Referenzen auf `GO_2` und `GO_3`, was zum Aufruf deren `getTreeData`-Methode führt (1.1 und 1.2). Innerhalb dieser Methoden rufen

die betreffenden EJBs wiederum auf ihrem ADC die `setTreeData` mit einer Referenz auf sich selbst auf (hier nur für `GO_3` dargestellt, 1.2.1). Dies führt während der Attributanalyse wiederum zu einem Aufruf der `getTreeData`-Methode von `GO_4`. Die ADC-Objekte sorgen im Rahmen des Analyse-Algorithmus dafür, daß die `getTree`-Methode von referenzierten Beans aufgerufen werden. Die EJBs können hierzu z.B. eine Schnittstelle implementieren, zu deren Operationen die `getTree`-Methode gehört. Dies ermöglicht dann eine Typumwandlung der EJB-Referenz auf diese Schnittstelle durch den ADC und den Aufruf der `getTreeData`-Methode. Als Alternative zu diesem Vorgehen kann der ADC auch Java-Reflection verwenden, um festzustellen, ob eine EJB eine `getTreeData`-Methode besitzt und diese ggf. aufrufen. Dieses Vorgehen vereinfacht den Algorithmus, falls es EJBs gibt, die keine `getTreeData`-Methode besitzen sollen und damit übergangen werden. Damit in der auslösenden `getTree`-Methode am Ende die Daten des gesamten Baums zur Verfügung stehen, werden die Daten in jeder `getTreeData`-Methode ADC-Intern zusammengeführt. Dieser Vorgang ist für das vorliegende Beispiel in Abbildung 5.7 dargestellt.

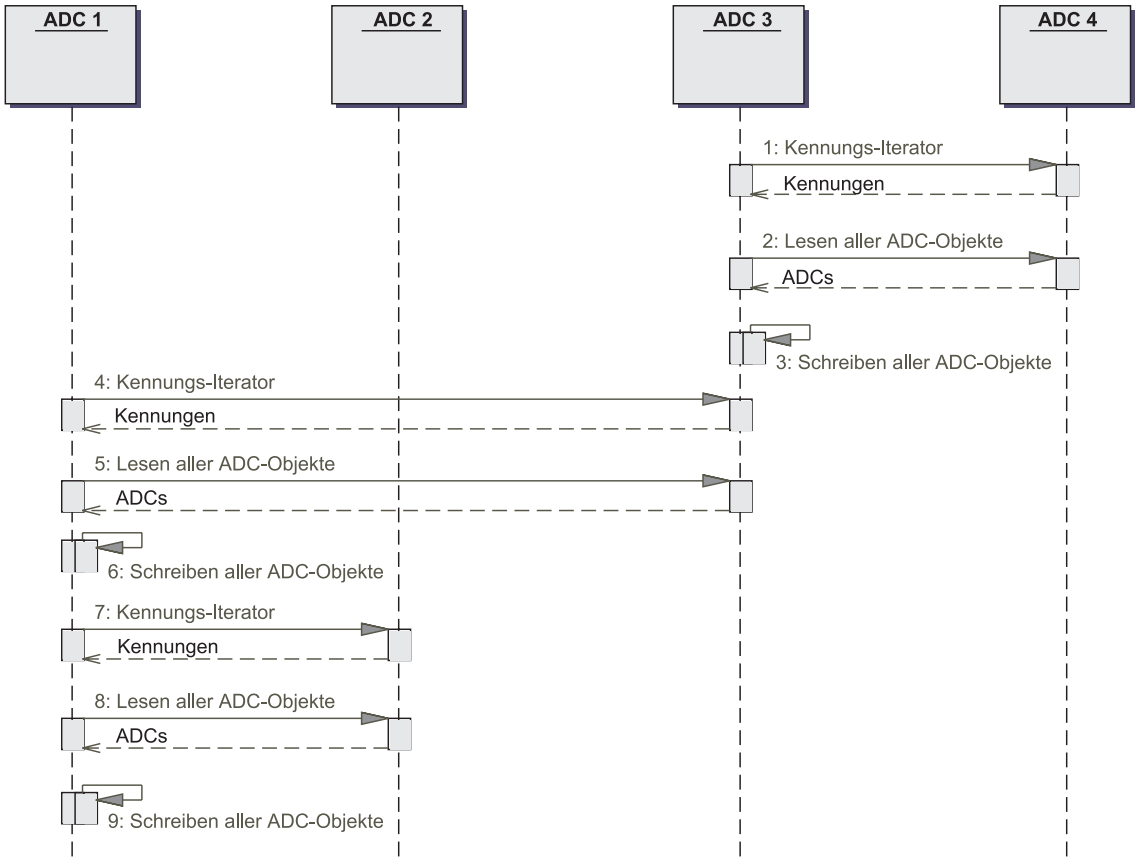


Abbildung 5.7: Interaktionen zwischen ADCs

Der Daten-Container ADC 3 erhält beim Aufruf der `getTreeData`-Methode von `GO_4` eine Kopie des Daten-Containers ADC 4 (1.2.1.1 in Abbildung 5.6). Von diesem Daten-Container werden alle Kennungen ausgelesen, unter denen Daten abgelegt sind (1). Die Daten werden für alle Kennungen daraufhin entnommen (2) und in der Datenstruktur von ADC 3 selbst ab-

gelegt (3). ADC 1 erhält beim Aufruf der `getTreeData`-Methode von `GO_3` eine Kopie von diesem Daten-Container (1.2 in Abbildung 5.6), liest die Daten aller Kennungen aus und speichert sie wiederum in seiner eigenen Datenstruktur (4, 5, 6). Der selbe Ablauf findet ebenfalls bei der Kopie des Daten-Containers ADC 2 von `GO_2` statt (7, 8, 9), der im Rahmen der `getTreeData`-Methode verschickt wird (1.1 in Abbildung 5.6). Damit stehen am Ende die Daten des Baums in ADC 1 zu Verfügung und können zum Anwendungs-Client geschickt werden. Beim Schreiben findet der selbe Vorgang umgekehrt statt. Insgesamt soll mit diesem neuen Ansatz gezeigt werden, wie stark die Entwicklung im Umfeld der Datenübertragung vereinfacht werden kann. Durch die Implementierung des Algorithmus im Daten-Container sind keine zusätzlichen Methoden in den EJBs erforderlich, die sonst zur Beschaffung bestimmter Daten pro Geschäftsvorfall erforderlich wären. Der Datenaustausch zwischen Objektbaum und Daten-Container selbst wird auf einen Methodenaufruf beschränkt.

5.4 Zwischenergebnis

In diesem Kapitel wurde gezeigt wie die universellen Datenübertragungskonzepte in die verschiedenen EJB-Arten integriert und in verschiedenen Architekturansätzen verwendet werden können. Dabei wurden neue Vorschläge gemacht, wie die Bereitstellung von Daten-Containern erfolgen soll, damit gleichzeitig eine Ressourcenschonung geleistet wird. Außerdem wurde gezeigt, daß die Konzepte in allen Bean-Arten auf die selbe Art und Weise zum Einsatz kommen können und ebenso unabhängig von der verwendeten Architektur sind. Demnach steht in allen Architekturen eine neue Form der Flexibilität und das Optimierungspotential der universellen Übertragungskonzepte zur Verfügung. Die Datenübertragung kann durch die Definition von Parametern im Deployment-Deskriptor und/oder durch eine global verfügbare Definition (z.B. im JNDI zugängliches Parameterobjekt) gelöst werden. Dabei kann durch dynamische Erzeugung der Daten-Container-Objekte anhand ihres Namens die zu verwendende Implementierung zur Laufzeit erzeugt werden. Diese Vorgehensweise kann z.B. auch zur dynamischen Anforderung unterschiedlicher *Aktiver Value Objects*, die zusätzlich mit universellen Konzepten ausgestattet sind, verwendet werden. Diese können während des Lebenszyklus einer EJB hinzugefügt werden, ohne den EJB-Code zu ändern. Die Anwendung der Konzepte führt in allen Architekturen auch zu einer Vereinfachung, da nur ein Methodenaufruf des Daten-Containers genügt, um die Daten von Objektbäumen, die aus Java-Objekten und EJBs bestehen können, zu übertragen. Damit wurde ein weiteres, bisher nicht berücksichtigtes Einsparungspotential aufgedeckt. Im folgenden Kapitel werden die Gesamtergebnisse dieser Arbeit zusammengefaßt und Ergebnisse ihrer praktischen Anwendung dargestellt.

Kapitel 6

Ergebnisse und Anwendungen

Im Rahmen dieser Arbeit wurde ein universelles Datenübertragungskonzept entwickelt, das auf *Aktiven Daten-Containern (ADC)* beruht. Die Container können bestehende Konzepte komplett ersetzen (dynamische Konzepte) oder erweitern (statische Konzepte, *Aktive Value Objects (AVO)*). Zur Erläuterung der Konzepte wurden im Rahmen dieser Arbeit beispielhafte Implementierungen vorgenommen.

6.1 Implementierungsaspekte

6.1.1 Allgemeines

Ein Ziel der Arbeit bestand darin, den Aufwand für die Erstellung von Anwendungen im Umfeld der Datenübertragung zu reduzieren und dabei möglichst gleichzeitig ein Höchstmaß an Flexibilität zu erreichen. In Folge davon kann auf geänderte Anforderungen, die sich auf die Datenübertragung auswirken, reagiert werden, ohne den Code der Anwendung anzutasten. Weiterhin sollte eine klare Strategie zur Übertragung von Daten gewährleistet werden, die zu einem besseren Verständnis des Programmcodes führt. Diese Ziele wurden durch das neue Konzept der Aktiven Daten-Container erreicht, die im Gegensatz zu bestehenden Konzepten die folgenden Merkmale besitzen:

- Es werden spezialisierte Transportobjekte zur Datenübertragung verwendet, die über eine an die Anforderungen der Anwendung angepaßte, standardisierte Schnittstelle verfügen. Die Konzeption der Schnittstelle definiert dabei eine Reihe von allgemeingültigen Operationen, die im Rahmen des Hinzufügens und des Entnehmens von Daten sowie zur Anwendung von Funktionalität, die an die Datenübertragung angelehnt ist, zum Einsatz kommen.

- Die Realisierung der Daten-Container-Objekte kann während des Lebenszyklus einer Anwendung konfiguriert, erweitert oder gar komplett ausgetauscht werden, ohne die Implementierung der Codeteile, die auf diese Transportobjekte zurückgreifen, zu beeinflussen. Diese Flexibilität durch Konfigurierbarkeit wird statisch durch Codeänderungen der Daten-Container-Klassen (Trennung von Schnittstelle und Implementierung) und dynamisch durch den Aufruf von Konfigurationsmethoden sowie der Hinterlegung von Konfi-

gurationsinformationen im Deployment-Deskriptor von EJBs oder durch globale Bereitstellung (z.B. im JNDI-Namensdienst) erreicht.

- Die Daten-Container besitzen Funktionalität, die zur Datenbefüllung und Entnahme der Daten von Objekten in der Logik- und Datenschicht vorgesehen ist. Dies führt zu einer Senkung der Herstellungskosten einer EJB-Anwendung, da der Entwickler keine manuellen Datenoperationen mehr durchführen muß. Dabei wurde zusätzlich ein neues Verfahren entwickelt, das in der Lage ist, Objektbäume auch über EJBs hinweg automatisch zu durchlaufen. Aufgrund des Schnittstellenkonzepts können weitere Einsparungen erzielt werden, indem serverseitig weitere allgemeine Komponenten implementiert werden, die z.B. andere Sichten auf die Daten generieren. Ein weiterer neuer Ansatz besteht darin, die Daten-Container selbst als clientseitiges Datenmodell zu verwenden, das seine Daten automatisch mit den GUI-Elementen austauscht. Dies führt neben der Entlastung jedes einzelnen Entwicklers zusätzlich zu einer einfacheren Struktur der Anwendung, da weniger Klassen benötigt werden. Bei Verwendung von dynamischen Aktiven Daten-Containern genügt ein Transportobjekt anstatt vieler für jedes Server-Objekt, dessen Daten transportiert werden müssen. Die statische Variante erlaubt die Begrenzung der Transportobjekte auf eines pro Objekt, dessen Daten übertragen werden sollen. Dabei können die vollen Vorteile der aktiven Konzepte genutzt werden, insbesondere kann dadurch auch eine Leistungssteigerung erzielt werden (vgl. dazu Abschnitt 6.2).

- Es kann eine Abbildung (*Mapping*) zwischen Attributnamen von Objekten, deren Zustand transportiert werden soll und Attributnamen, die im Daten-Container (und damit im Client) verwendet werden, erfolgen. Dies führt zu einer weitergehenden Entkopplung zwischen Client und Server. Selbst wenn sich Attributnamen von Objekten auf dem Server ändern, können diese weiterhin unter dem gleichen Namen an den Client geliefert werden. Hierzu ist lediglich die Zuordnungstabelle der Attributnamen zu ändern. Es muß keine Neuerstellung des Clients erfolgen.

- Die hier vorgestellten neuen Konzepte erlauben eine feingranularere Kontrolle über die Anwendung von Funktionalität. Sie kann sowohl auf den kompletten Inhalt eines Daten-Containers angewendet werden, als auch auf einzelne Attribute. Zusätzlich kann der Zeitpunkt der Funktionalitätsanwendung gesteuert werden. Somit ist es möglich, die Funktionalität vor bzw. nach dem Datenübertragungsvorgang auszuführen, oder erst beim tatsächlichen Zugriff auf die Daten. Somit können z.B. große Datenmengen clientseitig solange komprimiert bleiben bis ein tatsächlicher Zugriff stattfindet.

- Durch die Konzepte dieser Arbeit kann Expertenwissen in zentrale Datenübertragungsklassen fließen und anderen Entwicklern, die sich ausschließlich mit fachlicher Logik der Anwendung auseinandersetzen, zur Verfügung gestellt werden. Dazu gehört z.B. auch die Anwendung von Entwurfsmustern, wie z.B. *Dynamic Attributes* und *Smart Proxies*.

Dynamische Datenübertragungskonzepte werden durch die Konzepte dieser Arbeit insbesondere dahingehend verbessert, daß Klassen und Objekte verwendet werden, die ausschließlich für die Datenübertragung bestimmt sind. Dies führt zu einer besseren Lesbarkeit des Programm-Codes und zu einem besseren Verständnis, da die Methoden eine feste semantische Bedeutung

im Zusammenhang mit der Datenübertragung besitzen. Zusätzlich werden Belange abgedeckt, die mit herkömmlichen dynamischen Konzepten nicht oder nur umständlich erreicht werden können. Dazu gehört z.B. die Erzeugung einer Ausnahme, falls vom Daten-Container ein Attribut angefordert wird, das dieser nicht enthält. Dieses Verhalten dient der Fehlervermeidung. Darüber hinaus werden Zusatzabfragen verhindert, die sonst notwendig sind, um festzustellen, ob der Wert eines Attributs `null` ist oder ob das Attribut nicht im Container enthalten ist.
Statische Datenübertragungskonzepte werden durch die Konzepte dieser Arbeit insbesondere dahingehend verbessert, daß Objekte die Eigenschaften eines universellen Daten-Containers durch eine Basisklasse erben können. Somit können die Objekte entweder als universeller oder als statischer Daten-Container verwendet werden. Dies eröffnet neben den bereits erläuterten Optimierungspotentialen die Möglichkeit, statische Daten-Container mit einer Methode unter ihrem Namen als universelle Daten-Container anzufordern. Durch eine Typumwandlung können die Container anschließend im Client wieder als statische Container verwendet werden. Auf diese Art und Weise können einer EJB nachträglich statische Daten-Container hinzugefügt werden, ohne den Quellcode ändern zu müssen. Dies kann z.B. für Standardkomponenten eingesetzt werden, die nachträglich an die Anforderungen eines Projekts angepaßt werden. Weiterhin können mittels eines Generatoransatzes Aktive *Value Objects (AVO)* generiert werden, die ihre Daten optimieren bzw. mit Methoden ausgestattet sind, die den Transport ihrer Daten optimiert übernehmen. Dabei wird pro Geschäftsobjekt nur ein AVO benötigt, das zur Laufzeit nur die vom Anwendungsentwickler benötigten Attribute überträgt und dabei die anfallende Datenmenge grundsätzlich einschränkt, ohne für diesen Fall eine neue Transportklasse, die nur die zu übertragenden Attribute definiert, schreiben zu müssen. Somit entsteht eine Einsparung von Aufwand im mehrfachen Sinne. Der Applikations-Server benötigt weniger Zeit zur Verarbeitung des Daten-Containers in seiner Kommunikationsschicht, die Netzwerkbandbreite wird weniger in Anspruch genommen, die Implementierungsbelastung des Anwendungsentwicklers sinkt und es kommt zu keiner schwer überschaubaren, schlecht wartbaren Menge an Transportklassen.

6.1.2 Industrieller Einsatz

Die Konzepte Aktiver Daten-Container und GUI-Manager wurden bei der *iT media Consult GmbH* im Rahmen eines großen Industrieprojekts eingesetzt. Projektziel war die Erstellung einer Applikation zur umfassenden, komplexen Datenerfassung und -auswertung auf Basis einer mehrschichtigen EJB-Anwendung. Die Komplexität der Anwendung kommt u.a. in einer relationalen Datenbank mit über 400 Tabellen zur Datenhaltung und weit über 100 komplexen Bearbeitungsdialogen, die hauptsächlich GUI-Elemente in Form von Tabellen zur Ein- und Ausgabe von Daten verwenden, zum Ausdruck. Als Basisarchitektur kam eine auf zustandslosen Session-Beans beruhende Struktur, die auf persistente Java-Geschäftsobjekte zugreifen, zum Einsatz (vgl. dazu Kapitel 3, Abschnitt 3.1). Die Java-Clients ermöglichen eine Datenverarbeitung mittels aufwendiger grafischer Swing-Benutzeroberfläche.
Mit den nachfolgend genannten Prämissen wurden die Aktiven Daten-Container-Konzepte in das Projekt eingeführt:

- **Einheitlichkeit.** Alle im System anfallenden Daten sollen mit dem selben Konzept übertragen werden. Dies soll zum besseren Verständnis des Programmcodes beitragen.

- **Dynamische Datenübertragung.** Aufgrund der vielfältigen Anforderungen an die zu übertragenden Daten und die hohe Anzahl von Geschäftsobjekten (400 Datenbanktabellen), soll ein dynamischer Ansatz zur Übertragung von Daten gewählt werden.
- **Flexibilität.** Neben der Flexibilität, die durch die dynamische Datenübertragung erzielt wird, soll es zusätzlich möglich sein, die Datenübertragung nachträglich zu beeinflussen, um auf unvorhergesehene Anforderungen reagieren zu können.
- **Leichte Benutzbarkeit.** Das Konzept soll möglichst einfach benutzbar sein und zu einer Einsparung von Entwicklungsaufwand führen.
- **Keine Geschäftsklassen auf dem Client.** Der Client soll keine Geschäftsobjekte des Servers verwenden und damit keine Geschäftsklassen benötigen.

Im Vordergrund stand die Optimierung des Entwicklungsaufwands. Durch den Einsatz der Konzepte dieser Arbeit konnten die in Tabelle 6.1 dargestellten Einsparungen erzielt werden.

Nicht benötigte Daten-Container	17028 LOC (244 Klassen)
Automatischer Datenaustausch zwischen ADC und Server-Objekten	1600 LOC
Automatischer Datenaustausch zwischen ADC und GUI-Elementen	5050 LOC

Tabelle 6.1: Mit Aktiven Daten-Containern und zugehörigen Erweiterungen erzielte Einsparungen

Durch das Entwurfsmuster *Aktiver Daten-Container* in seiner dynamischen Fassung konnte auf die Einführung von *Value Objects* verzichtet werden. Bei 134 persistenten Geschäftsobjekten entspricht dies einer Einsparung von 244 Klassen mit ca. 17028 Programmzeilen (*Lines of Code (LOC)*). Aufgrund der Existenz von Geschäftsobjekten mit sehr vielen Attributen (zwischen 20 und 183 Attributen) würden 134 Daten-Container nicht ausreichen. In 112 Fällen erfolgt eine Einschränkung der zu Übertragenden Attributmenge, was wiederum die Erstellung von 112 speziellen Daten-Containern mit ca. 4214 Programmzeilen zwingend erforderlich machen würde. Statt 246 verschiedene Daten-Container einzusetzen, erfolgt die Verwendung von 2 Aktiven Daten-Containern mit insgesamt 438 Programmzeilen. Die Existenz von 2 verschiedenen Daten-Containern ermöglicht die Unterscheidung zwischen Einzelobjekten und Objektfolgen (Massendaten).
Die neuen Konzepte dieser Arbeit erfordern keinen manuellen Datenaustausch zwischen Daten-Containern und Objektstrukturen auf der Server-Seite und aufgrund des Entwurfsmusters *GUI-Manager* keinen manuellen Datenaustausch zwischen Daten-Containern und GUI-Elementen auf der Client-Seite. Unter der Prämisse, daß pro Client-Anfrage immer alle Attribute der darin benötigten Geschäftsobjekte übertragen werden müssen, führt die Anwendung der Aktiven Daten-Container zu einer Codeeinsparung von 3848 Programmzeilen. Dieses Potential wird jedoch nicht vollständig ausgeschöpft, da die unnötige Übertragung von Attributen, die im aktuellen Arbeitsschritt des Clients nicht benötigt werden, zugunsten des Leistungsverhaltens vermieden wird. Bei 219 Verwendungen eines ADCs zum Datentransport verbleibt jedoch eine

tatsächliche Einsparung von 1600 Codezeilen. Dies entspricht einer Reduktion der Datenaustauschoperationen um 82,9%. Die Verwendung des ADCs als Client-Datenmodell im Rahmen eines GUI-Managers führt in der vorliegenden Anwendung zu einer Einsparung von mindestens 5050 Codezeilen. Dies entspricht einer Reduktion der Datenaustauschoperationen um 91,5%. Die Ersparnis beruht im wesentlichen auf der Vermeidung von komplexen Datenaustauschoperationen mit Swing-Tabellen, die in 121 Fällen Bestandteil von aufwendigen Dialogen sind. In den Angaben ist der Entwicklungsaufwand der dafür notwendigen Standardkomponenten auf Client-Seite berücksichtigt. Insgesamt konnte in der vorliegenden Applikation Entwicklungsaufwand im Umfang von ca. 23678 Codezeilen eingespart werden[1]. Dies entspricht einer Reduktion des Gesamtumfangs der Anwendung um ca. 18%[2].

Neben der direkten Einsparung von Codezeilen existiert auch insgesamt eine Zeiteinsparung, da kaum Überlegungen stattfinden müssen, wie Daten zwischen Containern und Server-Objektstrukturen bzw. zwischen Containern und GUI-Elementen ausgetauscht werden müssen. Zusätzlich können Überlegungen jedes einzelnen Entwicklers entfallen, die sich mit einer Optimierung der Datenübertragung befassen. Diese Überlegungen können zentralisiert von Experten vorgenommen und realisiert werden.

Bei Anwendung des Konzepts *Aktive Value Objects* würde die in Tabelle 6.2 dargestellte Einsparung möglich werden.

Nicht benötigte Daten-Container	4033 LOC (112 Klassen)
Automatischer Datenaustausch zwischen AVO und Server-Objekten	1600 LOC
Automatischer Datenaustausch zwischen AVO und GUI-Elementen	5050 LOC

Tabelle 6.2: Mit Aktiven Value Objects und zugehörigen Erweiterungen mögliche Einsparungen

Mit AVOs ist eine Beschränkung auf 134 benötigte Daten-Container-Klassen (ca. 13252 LOC), bei gleichzeitiger Möglichkeit der Übertragung von Attributteilmengen großer Objekte, möglich. Auf die Implementierung von 112 Daten-Containern (ca. 4214 LOC) zur Reduktion der übertragenen Attribute könnte verzichtet werden. Neben 134 AVOs würde lediglich ein weiterer Daten-Container (ca. 181 LOC) als Sammelbehälter für Objektfolgen benötigt. Damit würde sich eine Gesamteinsparung von 10683 LOC ergeben. Dies entspricht einer Reduktion des Gesamtumfangs der Anwendung um ca. 9%.

Die erzielten Ergebnisse sind in Tabelle 6.3 insgesamt zusammengefaßt.

[1] Stand März 2002. Die Implementierung ist noch nicht abgeschlossen.

[2] Laut einer in [Bal00] angegebenen Faustregel, kann ein Entwickler im Jahr ca. 3500 Codezeilen erstellen. Demnach konnte mit den Konzepten in dieser Arbeit eine Einsparung von ca. 6,8 Personenjahren erzielt werden.

Anforderung	Ergebnis
Einheitlichkeit der Daten-übertragung	Zur Übertragung von Objektzuständen werden konsequent ADC-Objekte verwendet.
Dynamische Datenübertragung	ADC-Objekte können beliebige serialisierbare Daten transportieren. Es werden keine weiteren Daten-Container-Objekte benötigt.
Flexibilität	ADC-Objekte können nachträglich angepaßt bzw. zur Laufzeit (zentral) konfiguriert werden.
Zusatzfunktionalität	ADC-Objekte können bei hohen Datenmengen komprimiert werden.
Leichte Benutzbarkeit und Reduktion des Entwicklungsaufwands	ADC-Objekte besitzen eine einfache Schnittstelle. Die Objektzustände werden automatisch zwischen Daten-Containern u. Geschäftsobjekten transferiert. Eingespart werden ca. 23678 Codezeilen und mindestens 244 Daten-Container
Keine Geschäftsklassen auf dem Anwendungs-Client	Der Client benötigt nur die ADC-Klassen. Die ADC-Objekte können als Stellvertreter für Geschäftsobjekte angesehen werden.

Tabelle 6.3: Zusammenfassung der Ergebnisse

6.2 Leistungsaspekte

Die in dieser Arbeit neu entwickelten Konzepte ermöglichen auch die Optimierung der zu übertragenden Daten hinsichtlich Menge und Komplexität. Die Optimierung kann dabei mit allgemeinen Maßnahmen erfolgen, die weitgehend unabhängig von den zu transportierenden Daten sind. Dazu gehört z.B. die Anwendung von universell einsetzbaren Kompressionsalgorithmen. Im Rahmen dieser Arbeit wird jedoch vorgeschlagen, auch spezielle Überlegungen durchzuführen, die sich auf die Struktur der Daten und die Kommunikationsschicht des zugrundeliegenden Applikations-Servers beziehen, um daraus Maßnahmen abzuleiten, die ein besseres Leistungsverhalten zur Folge haben. Die Implementierung der Ergebnisse von solchen Überlegungen kann aufgrund der Konzeption von ADCs während des gesamten Lebenszyklus einer Anwendung zentral stattfinden und steht damit als weitere Möglichkeit zur Reaktion auf geänderte Systemanforderungen und evtl. auftretende Probleme, die mit der Datenübertragung zusammenhängen, zur Verfügung.

6.2.1 Testumgebung

Zur Bestimmung des Leistungsverhaltens wurde die nachfolgende Umgebung verwendet.

- **Server:** Pentium III, 800 MHz, 256 MB Hauptspeicher, Windows 2000, Java 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode). Die Applikations-Server wurden standardmäßig installiert (*Out-of-Box-Installation*). Der Java-Heap wurde einheitlich auf 128 MByte gesetzt. Die Server waren sowohl kommerzielle als auch frei verfügbare

Open-Source-Implementierungen des J2EE-Standards. Die Produkte sind anonymisiert, da mit dieser Arbeit nicht das Ziel verfolgt wird, einen Vergleich zwischen Applikations-Servern durchzuführen.

- **Clients:**
  - **Tests mit einem Client:** Pentium II, 350 MHz, 256 MB Hauptspeicher, Windows NT 4.0, Java 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode).
  - **Tests mit mehreren Clients:** 10 SPARC-Workstations, Java 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode). Die Daten der Workstations sind in Tabelle 6.4 zusammengefaßt. Pro Maschine wurden 2 virtuelle Clients verwendet, die mittels einem im Rahmen dieser Arbeit erstellten Werkzeugs gesteuert wurden. Das Werkzeug wird in Anhang B beschrieben.
- **Netzwerk:** 100 MBit/s Ethernet.

Anzahl	Modell	Prozessor	Hauptspeicher	Betriebssystem
1	Sun Ultra 2 UPA/SBus (2x)	UltraSPARC-II 296 MHz	640 MB	SunOS 5.8
1	Sun Ultra 5/10 UPA/PCI	UltraSPARC- IIi 300 MHz	640 MB	SunOS 5.7
1	Sun Ultra 5/10 UPA/PCI	UltraSPARC- IIi 360 MHz	384 MB	SunOS 5.7
2	Sun Blade 100	UltraSPARC- IIe 502 MHz	512 MB	SunOS 5.8
5	Sun Ultra 5/10 UPA/PCI	UltraSPARC-IIi 333MHz	384 MB	SunOS 5.7

Tabelle 6.4: Verwendete SPARC-Workstations

Testfälle

Zum Vergleich der unterschiedlichen Datenübertragungskonzepte wurden 4 repräsentative Testobjekte in Anlehnung an das in Abschnitt 6.1.2 vorgestellte Industrieprojekt zusammengestellt, deren Attributtypen und -belegungen typischen Anwendungssituationen entsprechen. In Tabelle 6.5 ist die Zusammensetzung der Objekte aus den verschiedenen Attributtypen beschrieben.
Die Belegungen der Attribute mit Werten erfolgte zufällig. Dabei wurden jedoch die typischen Wertebereiche des jeweiligen Anwendungshintergrunds beibehalten. Im folgenden werden die Objekte kurz erläutert:

- **Objekt 1:** Die Ganzzahlen sind nur mit Zahlen von 0 bis 9 belegt. Dieses Objekt spiegelt die sehr häufige Verwendung von Integer-Zahlen für kurze Schlüssel und Kennzeichen wider, die den Datenbereich der gewählten Datentypen nicht völlig ausschöpfen. Dies

Testobjekt	Attributanzahl	int	String	long	float	double	char	short	boolean
1	11	9	1	0	1	0	0	0	0
2	4	0	4	0	0	0	0	0	0
3	16	7	4	2	1	1	1	0	0
4	61	20	29	2	0	2	0	2	6

Tabelle 6.5: Verwendete Testobjekte

kommt in der Praxis sehr häufig vor. Dieses Vorgehen scheint zunächst unproblematisch zu sein, dabei wird jedoch die Auswirkung auf die Datenübertragung (höhere Datenmenge) nicht berücksichtigt. Diese Ineffizienz wurde im Rahmen dieser Arbeit als Beispiel ausgewählt, um Überlegungen zu demonstrieren, die bei der Datenreduktion an der im jeweiligen Anwendungsgebiet typischen Beschaffenheit der übertragenen Daten selbst ansetzt. Die im Rahmen dieser Arbeit entwickelten Daten-Container sind in der Lage, solche Daten zu optimieren, d.h. zu reduzieren.

- **Objekt 2:** Die Strings sind mit kurzen, 15-stelligen Zeichenketten belegt. Die Attributzusammensetzung repräsentiert Objekte, die auf textueller Repräsentation basierende Daten besitzen, die z.B. zum Aufbau von Listen verwendet werden. Hier kann keine zusätzliche Optimierung, wie bei Testobjekt 1 vorgenommen werden.

- **Objekt 3:** Das Objekt ist ein typisches Anwendungsobjekt, das Beschreibungen, Nummern, Preis- und Größenangaben enthält. Die Belegung ist anwendungsorientiert vorgenommen. Die Integer-Werte können hier im Vergleich zu Testobjekt 1 nicht signifikant reduziert werden.

- **Objekt 4:** Das Objekt repräsentiert die Daten eines typischen Geschäftsobjekts „Kunde“. Die Attributbelegungen sind anwendungsspezifisch vorgenommen. Im Vergleich zu Testobjekt 1 kann nur eine geringe anwendungsspezifische Optimierung vorgenommen werden.

Die Größe der verwendeten Testobjekte (ermittelt durch Serialisierung) inklusive Datenbelegung ist in Abbildung 6.1 dargestellt.
Die Daten der Testobjekte wurden zum Vergleich mit den bestehenden und den neuen Datenübertragungskonzepten übertragen. Als Vertreter für bestehende Datenübertragungsverfahren wurden verwendet:

Dynamische Konzepte:

Vektor aus HashMaps (VHM): Ein Objekt vom Typ `HashMap` erlaubt die Speicherung von Schlüssel/Wert-Paaren und speichert jeden Attributwert des Testobjekts unter dem zugehörigen Attributnamen. Zur Übertragung der Daten mehrerer Transportobjekte wird eine Folge von `HashMap`-Objekten innerhalb eines Objekts vom Typ `Vector` übertragen.

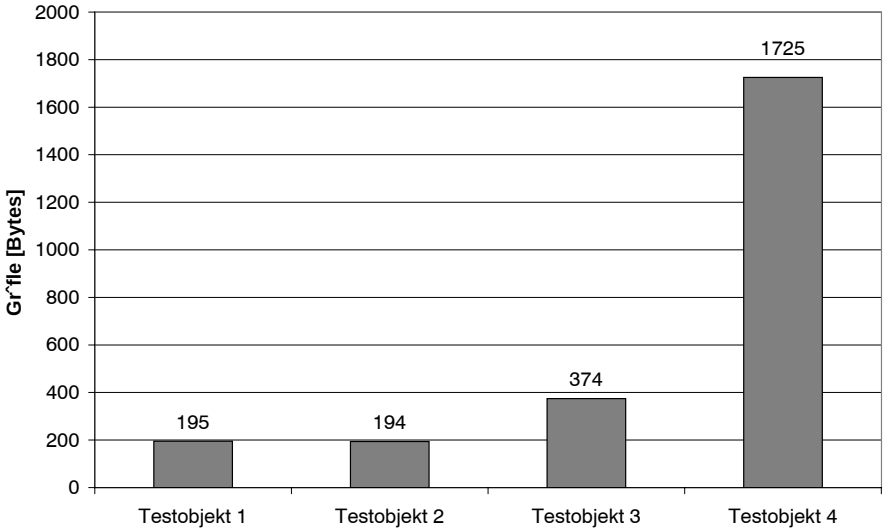


Abbildung 6.1: Größe der Testobjekte

Statische Konzepte:

Vektor aus Value Objects (VVO): Die Daten jedes Testobjekts werden in ein Transportobjekt übertragen, das die selben Attribute wie das Testobjekt besitzt. Es handelt sich dabei um die Umsetzung des Entwurfsmusters *Value Objects*. Um die Daten mehrerer Testobjekte auf einmal zu übertragen, werden die korrespondierenden *Value Objects* als Folge innerhalb eines Objekts vom Typ `Vector` übertragen. Dieses Konzept ist im Leistungsverhalten mit der direkten Übertragung der Testobjekte (Geschäftsobjekte) gleichzusetzen, da alle Attribute transportiert werden und der Zustand beider Objekte exakt gleich ist.
Als Beispielimplementierungen der in dieser Arbeit vorgestellten neuen Datenübertragungskonzepte, wurden die in Kapitel 4 vorgestellten Kernkonzepte prototypisch umgesetzt. Nachfolgend sind die wesentlichen Eigenschaften der Daten-Container verkürzt zusammengefaßt:

- **Dynamische Konzepte:**
  - **Aktiver Daten-Container, Implementierung 1 (ADC 1)**: Der Container kann beliebige serialisierbare Daten transportieren. Die Datenentnahme erfolgt automatisch mittels Java-Reflection. Um die Zustände vieler Objekte zu übertragen, werden Container-Objekte ineinander geschachtelt. Es handelt sich also um eine universelle Implementierung, die in der vorliegenden Form keinerlei Optimierung vornimmt. Somit kann der ADC 1 auch als Beispiel für eine komplexe Java-Datenstruktur aufgefaßt werden, die sehr leicht bei der Erstellung komplexer Anwendungen entsteht.
  - **Aktiver Daten-Container, Implementierung 2 (ADC 2)**: Der Container ist genauso universell einsetzbar wie der ADC 1 und basiert ebenfalls auf der Datenentnahme mittels Java-Reflection. Es wird jedoch eine Optimierung dahingehend vorgenommen, daß beim Speichern vieler Objektzustände die redundante Speicherung von

Daten verhindert wird und somit zu einer Reduktion der Datenmenge führt. Die Datenkomplexität wird durch diese Maßnahme ebenfalls geringfügig gesenkt.

- **Aktiver Daten-Container, Implementierung 3 (ADC 3)**: Der Container nimmt eine signifikante Reduktion der Datenkomplexität durch eine interne String-Repräsentation vor. Dies ist direkt auch mit einer Reduktion der Datenmenge verbunden. Eine zusätzliche Datenreduktion tritt in der vorliegenden Implementierung zusätzlich ein, wenn die im Container gewählte Datenrepräsentation kleiner ist, als die des ursprünglich verwendeten Datentyps. Objekte werden mittels Java-Reflection ausgelesen. Die ursprünglichen Daten werden ebenso beim Auslesen mittels Java-Reflection rekonstruiert. Die Datenverpackungs und -entpackungsprozesse des Daten-Containers sind hier, bei gleichzeitiger Optimierung der Datenstruktur, selbst nicht optimiert worden. Der Container dient also gleichzeitig zur Verdeutlichung des Einfluß von Verpackungs- und Entpackungsvorgängen.
- **Aktiver Daten-Container, Implementierung 4 (ADC 4)**: Der Container verwendet die selbe interne Datenrepräsentation wie ADC 3. Die Datenverpackungs- und entpackungsprozesse werden in dieser Implementierung jedoch ebenfalls optimiert. Auf die Verwendung von Java-Reflection wird vollständig verzichtet. Um die Einsparung von Entwicklungsaufwand bei der Verwendung des Containers zu sichern, muß der Einsatz eines einfachen Code-Generators erfolgen. Der Container ist damit als Beispiel für einen vollständig optimierten Container zu verstehen.

- **Statische Konzepte:** Alle statischen Konzepte setzen die Verwendung eines Generators voraus, um den Implementierungsaufwand zu senken. Außerdem erlaubt der Generator zusätzlich die zentrale Beeinflussung des Datenformats, das tatsächlich bei Kommunikationsvorgängen übertragen wird. Der Generator wird damit zum Bestandteil des Deployment-Prozesses, der für die Installation von EJBs im Applikations-Server durchgeführt werden muß.

  - **Vektor aus Aktiven Value Objects (VAVO)**: Ein Objekt vom Typ `Vector` dient als Sammelbehälter für AVOs, die ihre Serialisierung vollständig mittels `Externalizable` steuern. Dabei werden Ganzzahlen optimiert, falls ihr Wert nicht die volle Bitbreite benötigt. Jedes AVO nimmt gemäß den statischen Ansätzen die Daten eines Testobjekts zum Transport auf.
  - **Aktiver Container für Aktive Value Objects (AVOC)**: Der Container stellt eine spezielle Implementierung eines Sammelbehälters für AVOs dar, dessen interne Datenstruktur aus einer Folge von Bytes (`byte[]`) besteht. AVOs, deren Daten in dem Container gespeichert werden sollen, müssen `read/write`-Methoden besitzen, um ihre Daten in einen Byte-Strom zu schreiben, der vom AVOC bereitgestellt wird. Im Gegensatz zu VAVO wird hier neben einer Datenoptimierung durch die AVOs selbst (hier z.B. Optimierung von Ganzzahlen) auch eine Gesamtoptimierung der Daten durch den AVOC vorgenommen.
  - **Aktiver Container für Aktive Value Objects mit Kompression (AVOCC)**: Es handelt sich um den Daten-Container AVOC mit aktivierter Kompression und Dekompression. Der zur Kompression verwendete Deflate-Algorithmus wird im Mo-

dus BEST_SPEED verwendet. Dabei wird zugunsten einer besseren Laufzeit auf eine höhere Kompression verzichtet.

Die Tests wurden mit einem Client (Einzeltest) und 20 Clients (Lasttest) durchgeführt. In den nachfolgenden Abschnitten werden die Ergebnisse beider Tests dargestellt.

6.2.2 Einzeltest

Mit Hilfe des Einzeltests sollen grundsätzliche Aussagen über das Verhalten der unterschiedlichen Datenübertragungskonzepte im Umfeld der unterschiedlichen Applikations-Server gewonnen werden. Die Ergebnisse müssen jedoch durch einen Lasttest ergänzt werden, da die Existenz eines völlig unbelasteten Systems in der Praxis selten vorzufinden ist. Für den Test wurde in jedem Applikations-Server eine zustandsbehaftete Session-Bean, im folgenden *Test-Bean* genannt, installiert[3], die einen Abruf der Daten einer beliebigen Anzahl von Testobjekten mit den unterschiedlichen Übertragungskonzepten ermöglicht. Die Testobjekte werden dabei von der Session-Bean vor dem Test erzeugt und mit (zufälligen) Daten belegt. Dieses Vorgehen wurde gewählt, um einen reinen Test der Kommunikationsvorgänge vornehmen zu können, der nicht durch weitere Vorgänge, wie z.B. der Durchlauf von Persistenzschichten und Datenbankabfragen verfälscht wird.
Um die Eigenschaften der unterschiedlichen Konzepte klar zu erkennen, wurden zum Vergleich der unterschiedlichen Datenübertragungskonzepte jeweils die Daten von 500 Testobjekten übertragen. Gemessen wurden die folgenden Größen:

- **Datenmenge**: Wie bereits im Rahmen dieser Arbeit festgestellt wurde, ist die Datenmenge ein wesentlicher Einflußfaktor, der sich auf das Leistungsverhalten einer EJB-Anwendung auswirkt. Die Größe erlaubt eine grobe Einschätzung des Leistungsverhaltens, kann aber nicht isoliert betrachtet werden, da weitere Einflußfaktoren, wie Komplexität der vorliegenden Datenstruktur, Implementierungsqualität der Kommunikationsschicht des verwendeten Applikations-Servers bis hin zum verwendeten Kommunikationsprotokoll berücksichtigt werden müssen (vgl. dazu Kapitel 4, Abschnitt 4.2.2). Die Größe wurde mittels der Serialisierung des jeweiligen Testobjekts in ein Byte-Array bestimmt. Die Datenmenge der einzelnen Objekte war über alle Tests konstant.

- **Übertragungszeit**: Diese Größe gibt an, wie lange der Applikations-Server zur Übertragung eines Daten-Containers benötigt. In der Größe sind keine Zeiten zur Erstellung des Daten-Containers auf dem Server sowie zum Auslesen des Container-Inhalts im Client enthalten. Die Übertragungszeit hat eine hohe Bedeutung bei Engpässen im Anwendungssystem, die z.B. aufgrund einer nicht ausreichenden Netzwerkbandbreite oder einer stark belasteten Server-Maschine entstehen. Zusätzlich ist diese Größe interessant für Anwendungsfälle, die nur den Abruf bereits vorbereiteter oder statischer Daten zum Ziel haben. Die verschiedenen Daten-Container wurden zu diesem Zweck vor dem Test erzeugt, mit den Daten der Testobjekte belegt und in der Test-Bean zum Abruf bereitgestellt. Die

[3] Eine Untersuchung hat ergeben, daß die Verwendung einer zustandsbehafteten Session-Bean gegenüber einer zustandslosen Session-Bean im Einzeltest keine Auswirkungen auf die Übertragungszeiten hat.

Übertragungszeit ergibt sich dann aus der Dauer des entfernten Methodenaufrufs eines Clients.

- **Bereitstellungszeit**: Diese Größe gibt an, wie lange der Applikations-Server zur Erstellung und Übertragung eines Daten-Containers benötigt. Es werden keine Zeiten zum Auslesen des Containers auf dem Client berücksichtigt. Die Bereitstellungszeit erlaubt damit einen Vergleich der unterschiedlichen Datenübertragungskonzepte hinsichtlich der Zeitaufwände, die zum Erstellen und Befüllen von Daten-Containern erforderlich sind. Zusätzlich hat diese Zeit eine hohe Bedeutung, wenn der Client den Inhalt des Containers nicht sofort komplett ausliest. Zur Bestimmung der Bereitstellungszeit wurden die Daten bei Anforderung durch den Client gemäß den unterschiedlichen Vorgehensweisen in den Datenübertragungskonzepten aus den Testobjekten ausgelesen und in den jeweiligen neu erzeugten Daten-Container geschrieben. Die Bereitstellungszeit ergibt sich damit aus der Dauer des entfernten Methodenaufrufs eines Clients.

- **Antwortzeit**: Diese Größe gibt an, wie lange ein Kommunikationsvorgang inklusive Erstellung des Daten-Containers auf dem Server, Übertragung zum Client und das anschließende Auslesen, benötigt. Diese Größe berücksichtigt also den kompletten Verarbeitungsprozeß, der für eine Datenübertragung zwischen Server und Client erforderlich ist. Die Antwortzeit rückt bei schnellen Netzwerken in den Mittelpunkt, da sie dabei wesentlich von den Verpackungs- und Entpackungsvorgängen mitbestimmt wird. Die Bestimmung dieser Zeitgröße entspricht dem Vorgehen bei der Bereitstellungszeit. Allerdings ergibt sich die Antwortzeit erst, wenn der Client nach dem entfernten Methodenaufruf alle Daten im erhaltenen Transport-Container ausgelesen hat.

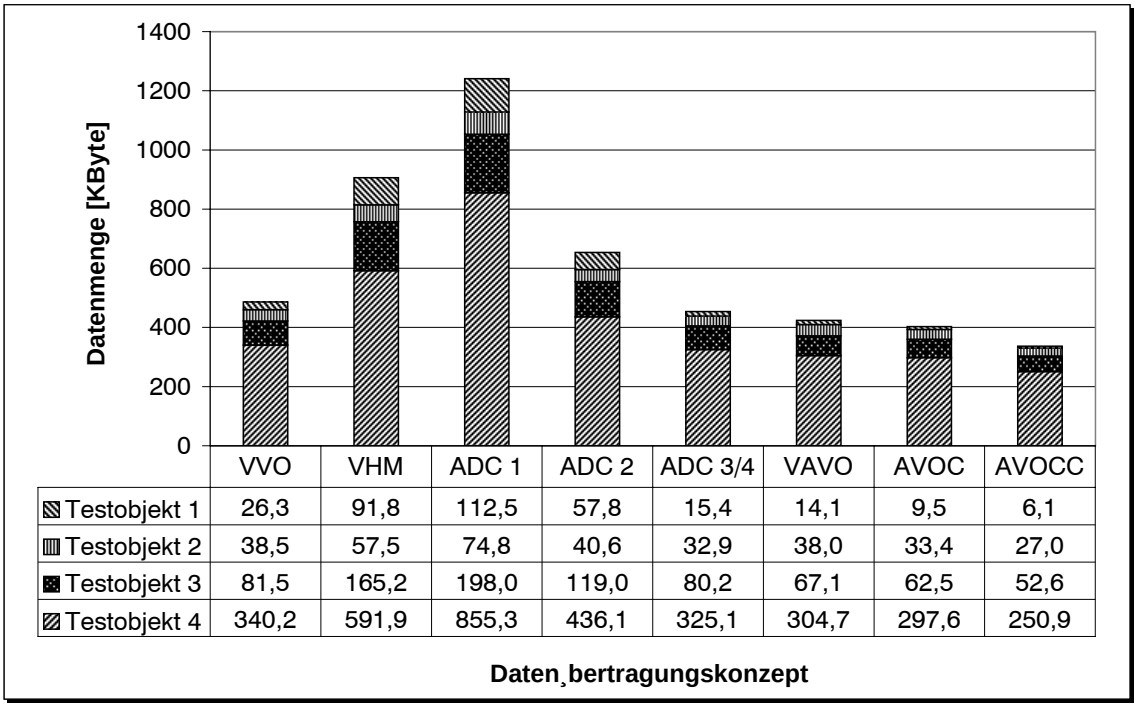
Zum Vergleich der unterschiedlichen Datenübertragungskonzepte werden nachfolgend mit Ausnahme der Datenmenge relative Werte in Prozent dargestellt. Dabei erfolgt gemäß der Klassifikation aus Kapitel 4 eine Aufteilung in dynamische und statische Datenübertragungskonzepte. Daten-Container, die gemäß der neuen Konzepte dieser Arbeit entwickelt wurden, werden dabei mit bestehenden Ansätzen verglichen. In der dynamischen Gruppe wird der Ansatz VHM als Basis (100%) verwendet. In der statischen Gruppe wird der Ansatz VVO als Basis (100%) verwendet. Die Vergleiche wurden auf acht verschiedenen Applikations-Servern (S0 bis S7) durchgeführt. Aufgrund der im Rahmen dieser Arbeit gewonnenen Erkenntnis, daß sich Applikations-Server, die auf RMI oder proprietären Protokollen basieren, unterschiedlich zu Servern, die auf CORBA/IIOP basieren, verhalten (vgl. dazu Kapitel 4), erfolgt zusätzlich eine entsprechende Aufteilung in zwei Gruppen:

- **Gruppe 1**: Auf RMI bzw. proprietären Protokollen basierende Kommunikationsschicht (S0 bis S4).

- **Gruppe 2**: Auf CORBA/IIOP basierende Kommunikationsschicht (S5 bis S7).

Datenmenge

In Abbildung 6.2 sind die anfallenden Datenmengen für jedes Datenübertragungskonzept nach der Serialisierung dargestellt.



	VVO	VHM	ADC 1	ADC 2	ADC 3/4	VAVO	AVOC	AVOCC
Testobjekt 1	26,3	91,8	112,5	57,8	15,4	14,1	9,5	6,1
Testobjekt 2	38,5	57,5	74,8	40,6	32,9	38,0	33,4	27,0
Testobjekt 3	81,5	165,2	198,0	119,0	80,2	67,1	62,5	52,6
Testobjekt 4	340,2	591,9	855,3	436,1	325,1	304,7	297,6	250,9

Abbildung 6.2: Einzeltest: Datenmenge der Container nach der Serialisierung

Es ist ersichtlich, daß der Daten-Container ADC 1 die größte Datenmenge besitzt. In diesem Sinne ist von ihm ein schlechteres Verhalten zu erwarten. Der Container ADC 2 stellt eine Verbesserung von ADC 1 dar, die redundante Daten eliminiert und damit im Vergleich zum VHM-Ansatz zu einer Reduktion der Datenmenge führt. ADC 3 und ADC 4 stellen eine stark optimierte Datenstruktur dar, die sich vor allem bei Testobjekt 1 am günstigsten auswirkt. Insgesamt können die Daten-Container bei jedem Testobjekt die Datenmenge gegenüber VVO unterschreiten. Dies ist bemerkenswert, da bei den Testobjekten 2 bis 4 die Reduktion vor allem aus einer Verkleinerung der Datenstruktur (Komplexität) und nicht der Anwendungsdaten selbst besteht. Insbesondere Testobjekt 2, dessen Attribute Zeichenketten mit fester Länge sind, macht dies deutlich. Das selbe gilt für die Daten-Container AVO und AVOC.
Die eingeschaltete Kompression von AVOC (AVOCC) reduziert die Datenmenge nochmals deutlich gegenüber allen Daten-Containern. Das Einschalten der Kompression bewirkt jedoch **keine** weitere Reduktion der Datenkomplexität.

Übertragungszeit

In Abbildung 6.3 sind die Übertragungszeiten für die Server der Gruppe 1 dargestellt. Die Implementierung ADC 1 schneidet bei allen Testobjekten mit einer um durchschnittlich 64,6% höheren Übertragungszeit am schlechtesten ab. Dies ist auf die hohe Datenmenge und die hohe Komplexität des Daten-Containers zurückzuführen. ADC 2 schneidet aufgrund der stark gesunkenen Datenmenge um durchschnittlich 31,5% besser ab. ADC 3 bzw. ADC 4 senken die

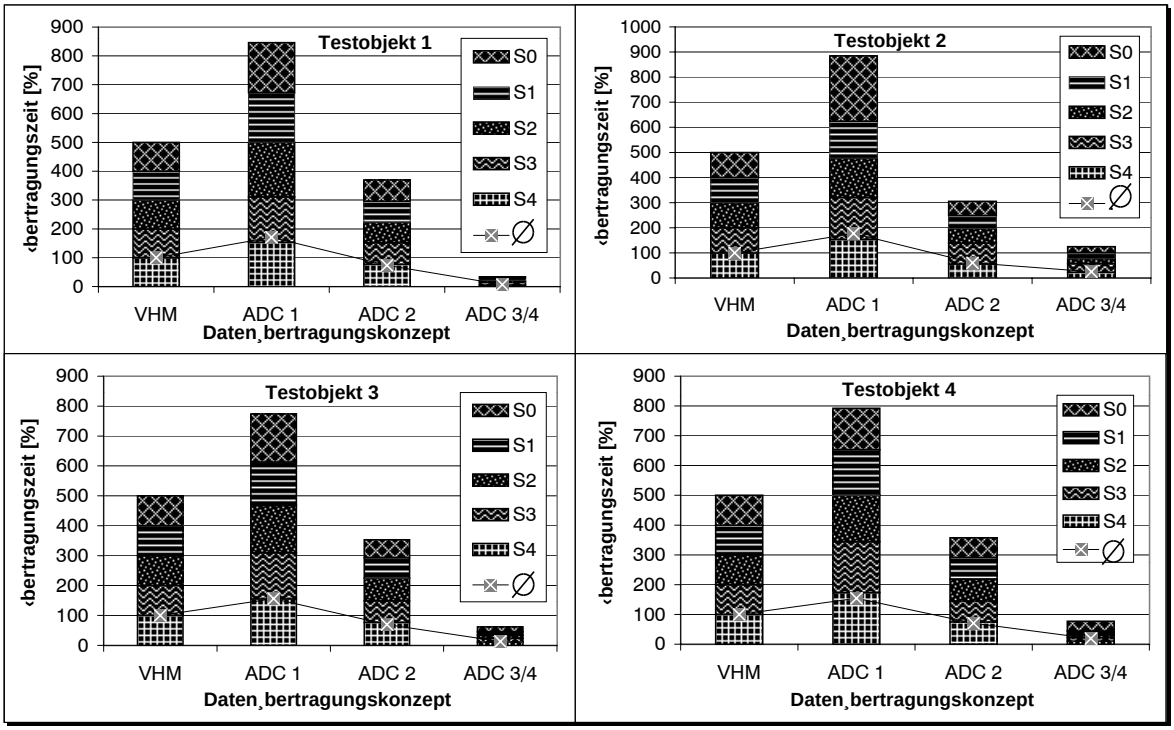


Abbildung 6.3: Übertragungszeitvergleich der dynamischen Daten-Container (Gruppe 1)

Übertragungszeit im Durchschnitt um 84,3%. Dabei fällt die Einsparung bei Testobjekt 1 mit durchschnittlich 93,5% am höchsten aus. Bei Testobjekt 2 fällt die Einsparung mit 75,6% am geringsten aus. Die Einsparung tritt neben der reduzierten Datenmenge bei allen Objekten vor allem auch wegen der signifikant gesenkten Komplexität der zu transportierenden Datenstruktur auf.

Abbildung 6.4 enthält die erhaltenen Übertragungszeiten für die Server der Gruppe 2.

ADC 1 schneidet wiederum mit einer um 162,2 % höheren Übertragungszeit am schlechtesten ab. Bemerkenswert ist das signifikant schlechtere Verhalten der Server gegenüber der Gruppe 1. Dieses Verhalten zeigt sich trotz der gesunkenen Datenmenge ebenfalls bei der Implementierung MAC 2 mit einer durchschnittlich 26,1% schlechteren Übertragungszeit. Die Kommunikationsschichten der Server von Gruppe 2 kommen wesentlich schlechter mit komplexen Datenstrukturen zurecht als die Server der Gruppe 1. Dies belegt auch die in der vorliegenden Testumgebung erhaltene Übertragungszeit von ADC 2, die sich bei Servern der Gruppe 2 im Durchschnitt bei 11 **Sekunden** bewegt. Im Gegensatz dazu beträgt die Übertragungszeit bei Servern der Gruppe 1 422,5 **Millisekunden**. Eine Reduktion der Datenkomplexität mit ADC 3/4 sorgt für eine um durchschnittlich 90,5% bessere Übertragungszeit. Dabei fällt die Reduktion bei Testobjekt 4 mit 97,9% am höchsten und bei Testobjekt 2 mit 78,7% am geringsten aus. Die Auswirkung der Datenkomplexität läßt sich am Vergleich der Datenmenge zur eingesparten Übertragungszeit erkennen. Die Implementierungen ADC 3 und ADC 4 reduzieren die Datenmenge von Testobjekt 2 nur um 42,8% bzw. 45% bei Testobjekt 4.

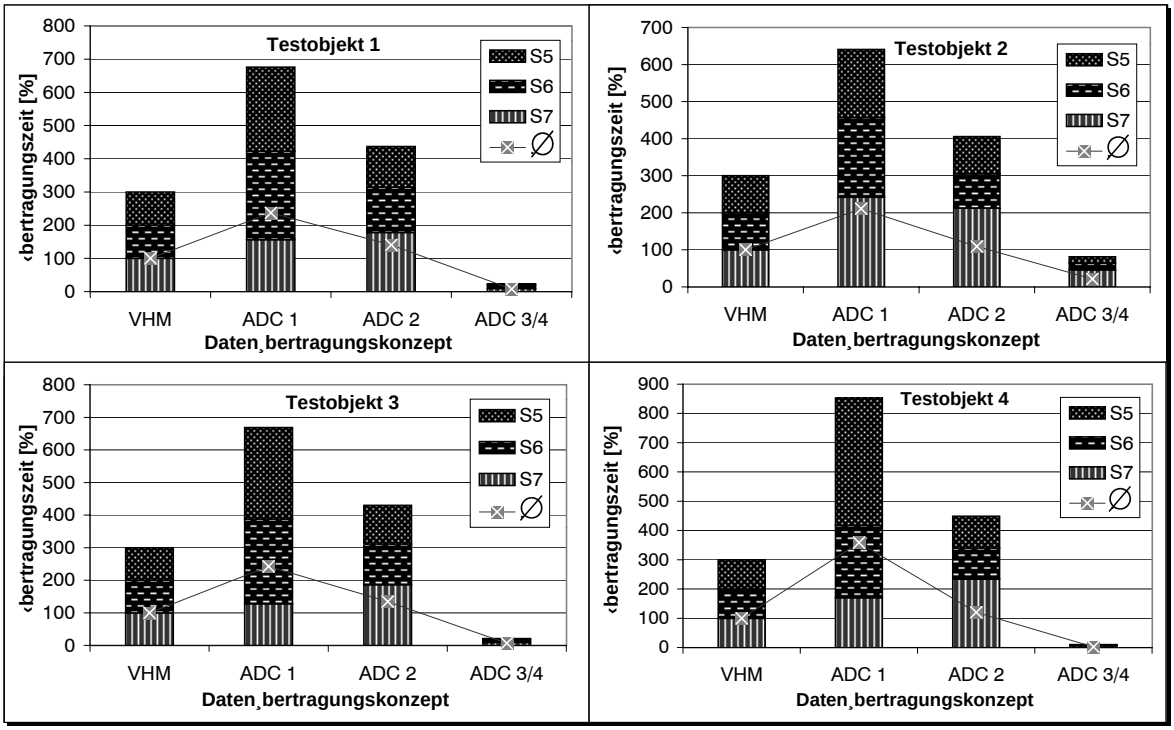


Abbildung 6.4: Übertragungszeitvergleich der dynamischen Daten-Container (Gruppe 2)

In Abbildung 6.5 sind die Übertragungszeiten der statischen Konzepte für Gruppe 1 dargestellt. Alle neuen Ansätze verbessern die Übertragungszeit. Dabei erfolgt durch den Ansatz VAVO eine Senkung um durchschnittlich 27,8%. Am schwächsten fällt das Ergebnis mit 18,7% bei Testobjekt 1 aus. Das beste Ergebnis wird bei den Daten von Testobjekt 2 mit 38,8% erzielt. Der Container AVOC führt zu einer Reduktion um durchschnittlich 79,4% unkomprimiert bzw. 82,9% komprimiert. Das beste Ergebnis liefert die Übertragung der Daten von Testobjekt 3 mit einer Reduktion um 84,7%. Die geringste Reduktion wird mit 67,5% bei den Daten von Testobjekt 4 erzielt. Das interne Datenformat von AVOC erfordert nur noch eine sehr geringe Verarbeitung in der Kommunikationsschicht der verschiedenen Server und führt so zu einer schnelleren Übertragung. Die erhaltenen Übertragungszeiten lassen sich nochmals durch Aktivierung der Kompression um durchschnittlich 82,9% senken.

Der Aufbau der neuen statischen Daten-Container wirkt sich besonders positiv bei den Servern der Gruppe 2 aus, die in Abbildung 6.6 dargestellt sind. Dies zeigt die deutliche Senkung der Übertragungszeit um durchschnittlich 51,1% (VAVO), 98,3% (AVOC) und 98,6% (AVOCC). Die Verwendung des Ansatzes VAVO war dabei mit dem Server S6 nicht möglich. Übertragungsvorgänge brachen mit einer Fehlermeldung ab. Dies ist auf einen Fehler in der Implementierung von S6 zurückzuführen. Auf die Verwendung von VAVO wurde daher bei S6 komplett verzichtet. Vor allem die interne Datenstruktur von AVOC verhindert die Probleme der Server mit komplexen Datenstrukturen. Wie bereits in Kapitel 4 gezeigt wurde, verhalten sich alle Server bei Daten geringer Komplexität ähnlich.

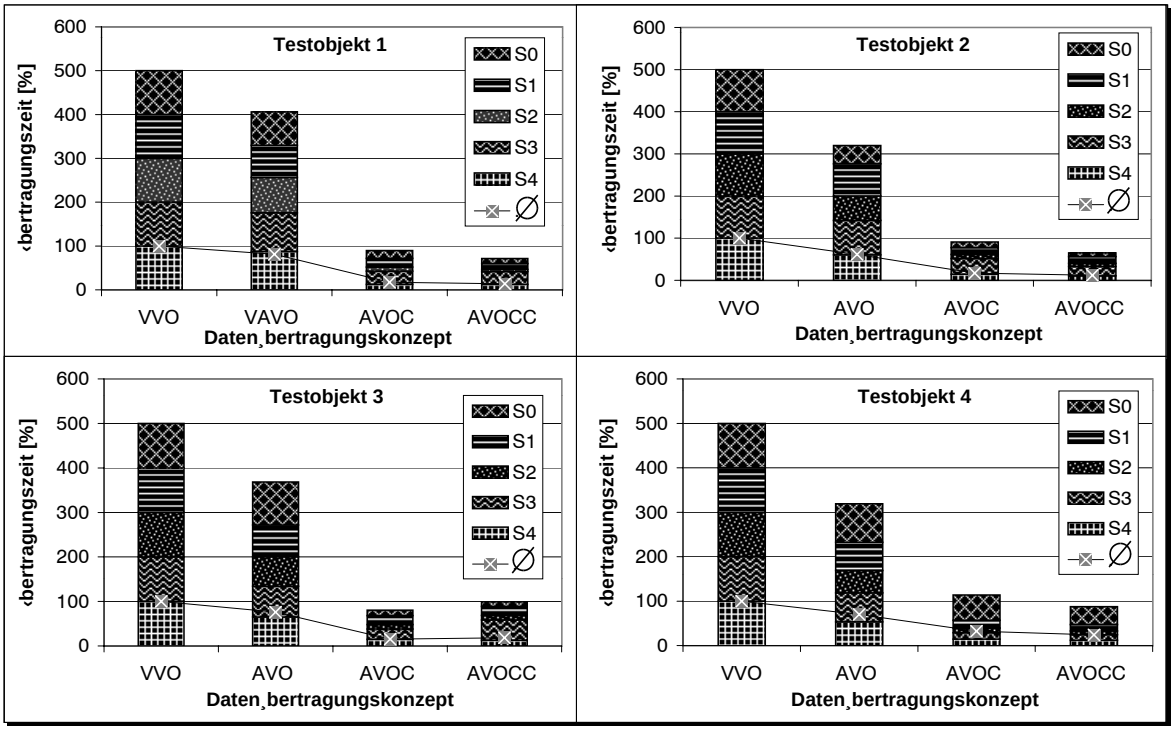


Abbildung 6.5: Übertragungszeitvergleich der statischen Daten-Container (Gruppe 1)

Bereitstellungszeit

Die Bereitstellungszeit bezieht das Erzeugen und das Befüllen der Container mit Daten ein. In Abbildung 6.7 sind die Ergebnisse für die Server der Gruppe 1 dargestellt. Hierbei zeigt sich nun die Verwendung von Java-Reflection zum Auslesen der Testobjekte, deren Daten transportiert werden sollen. So sind die erhaltenen Bereitstellungszeiten von ADC 1, ADC 2 und ADC 3 im Vergleich zu den Übertragungszeiten durchweg schlechter. Die Ersparnis von ADC 2 sinkt auf durchschnittlich 16,6% und von ADC 3 auf 65,6%. Der zusätzlich hinsichtlich Datenbefüllung und Datenentnahme optimierte Container ADC 4 besitzt dagegen eine Ersparnis von 78,8% im Durchschnitt.
Bei den in Abbildung 6.8 dargestellten Servern der Gruppe 2 ergibt sich bei ADC 3 ebenfalls eine geringere Zeiteinsparung, die sich auf durchschnittlich 87,9% beläuft. Dagegen schneidet der ADC 4 mit einer 88,8% geringeren Bereitstellungszeit ab.
Die statischen Daten-Container liefern in Gruppe 1 (Abbildung 6.9) mit durchschnittlich 32,5% (VAVO) und 70,9 % (AVOC) eine deutlich geringere Bereitstellungszeit. Mit eingeschalteter Kompression sinkt die Ersparnis von AVOC jedoch auf 41,5%. Dies macht deutlich, wie teuer die Anwendung eines Standard-Kompressionsalgorithmus sein kann, obwohl er nochmals zu einer Reduktion der Datenmenge führt. Im Gegensatz dazu entsteht durch die speziell an die Datenübertragung angepaßten Konzepte eine signifikant höhere Ersparnis.
Abbildung 6.10 zeigt die erhaltenen Bereitstellungszeiten der statischen Container für die Server der Gruppe 2. Wie erwartet, stellt sich eine höhere Ersparnis gegenüber den Servern der

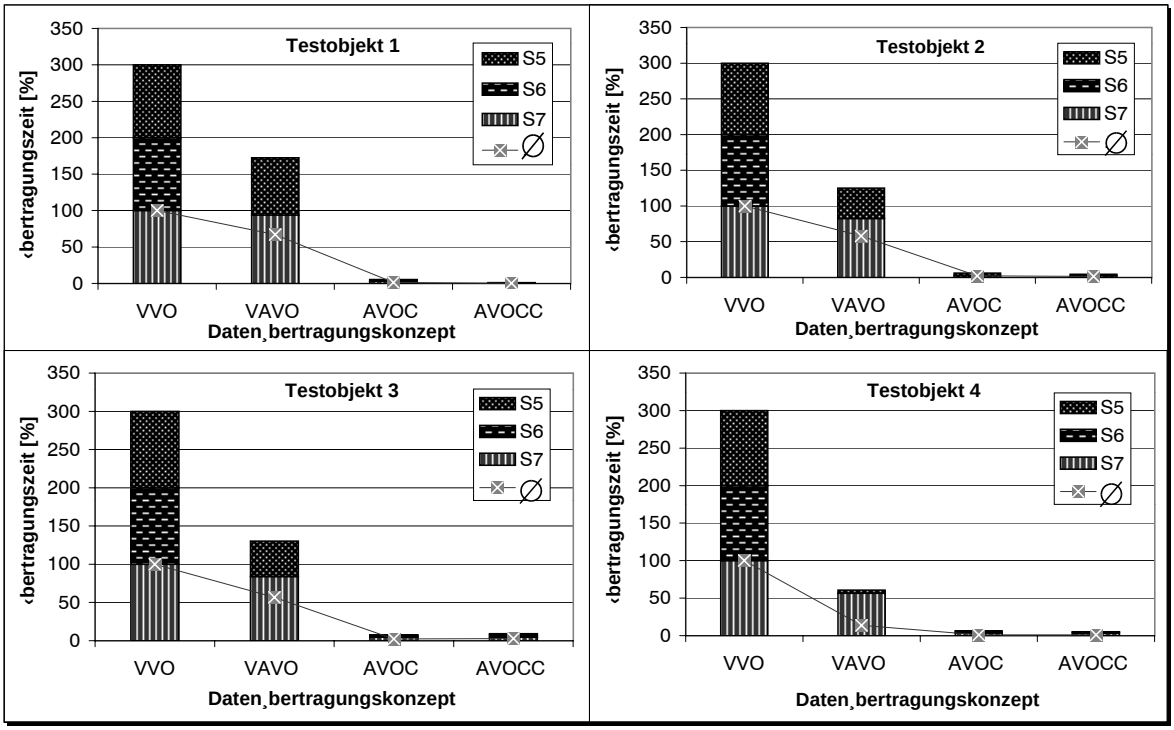


Abbildung 6.6: Übertragungszeitvergleich der statischen Daten-Container (Gruppe 2)

Gruppe 1 ein, die mit 50% (VAVO), 97,2% (AVOC) und 91,6% (AVOCC) im Durchschnitt ausfällt. Wiederum wird deutlich, daß die Kompression bereits bei der Anwendung auf dem Server zu einem schlechteren Leistungsverhalten führt.

Antwortzeit

Die Antwortzeit bezieht neben der Bereitstellungszeit zusätzlich die Zeit zur Datenentnahme auf dem Client mit ein. Die erhaltenen Antwortzeiten für Server der Gruppe 1 mit dynamischen Konzepten ist in Abbildung 6.11 veranschaulicht. Während sich die Ersparnis von ADC 2 auf dem Niveau der Bereitstellungszeit bewegt, existiert bei ADC 3 keine Ersparnis mehr, sondern eine deutliche Verschlechterung um durchschnittlich 152,2%. Dies zeigt wiederum die hohen Kosten der Java-Reflection, die bei der Datenentnahme im Client zur Ermittlung und zum Aufruf von Konstruktoren mittels Reflection zum Einsatz kommt. ADC 3 schneidet dadurch trotz seiner effizienten Datenstruktur schlechter als der Ansatz ADC 1 ab, der eine um durchschnittlich 61,1% höhere Antwortzeit liefert. Dieser Container benötigt beim Auslesen auf Seite des Clients keine Reflection. Bei den sehr schnellen Servern der Gruppe 1 fällt die Zeit zum Auslesen eines Daten-Containers besonders stark ins Gewicht, da die Übertragungszeiten vergleichsweise klein sind. Der Verzicht auf Java-Reflection bei der Datenentnahme führt bei ADC 4 immer noch zu einer Ersparnis von 53,7% im Durchschnitt. Im schlechtesten Fall existiert eine Verbesserung um 43,5% (Testobjekt 4). Im besten Fall sinkt die Antwortzeit um 81,5%

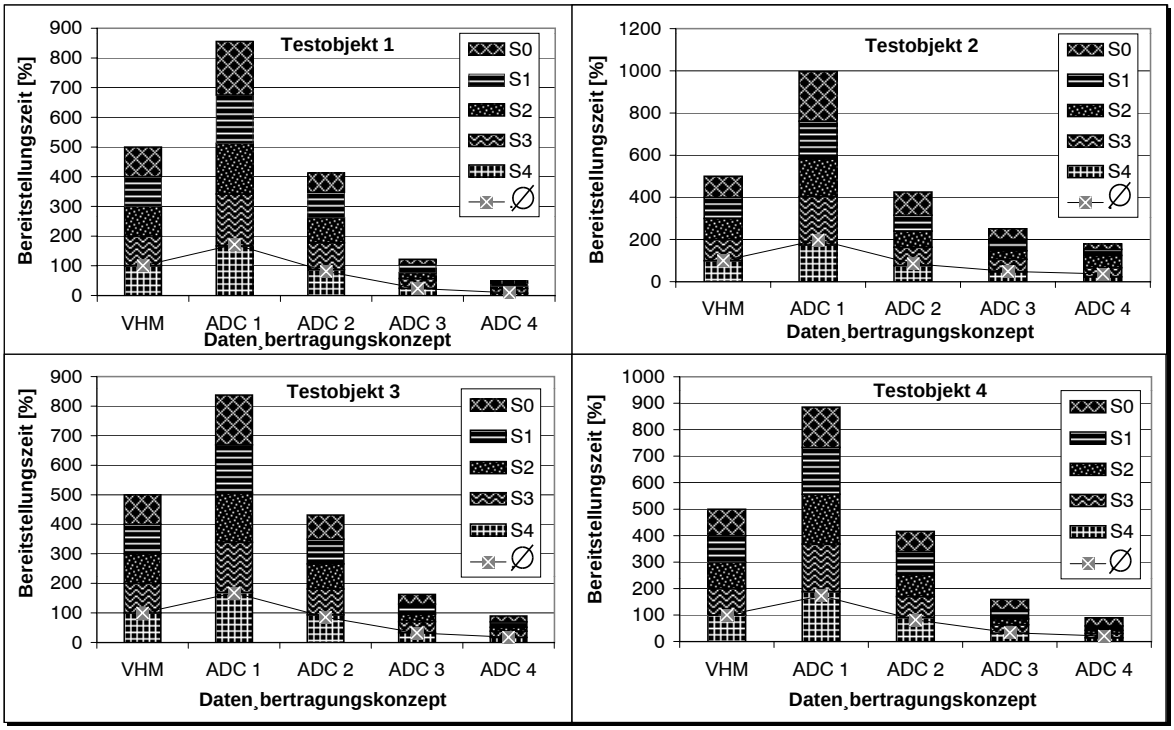


Abbildung 6.7: Bereitstellungszeitvergleich der dynamischen Daten-Container (Gruppe 1)

(Testobjekt 1). Insgesamt wird verdeutlicht, wie wichtig die Optimierung von Verpackungs- und Entnahmeoperationen bei der Entwicklung von optimierten Daten-Containern ist. In bestimmten Situationen kann dies dazu führen, daß zu Gunsten einer schnelleren Verpackungs- bzw. Entpackungszeit eine höhere Datenmenge in Kauf genommen wird.

Bei den Servern der Gruppe 2 (Abbildung 6.12) hingegen verbleibt bei der Antwortzeit der dynamischen Konzepte immer noch eine Ersparnis von durchschnittlich 63% bei ADC 3. Im Maximalfall beträgt die Reduktion 81,3% (Testobjekt 4). Im ungünstigsten Fall beträgt die Einsparung 39,3% (Testobjekt 2). In Folge davon fällt die Reduktion der Antwortzeit durch die Implementierung ADC 4 mit durchschnittlich 86% noch höher aus. Bei Testobjekt 4 zeigt sich die maximale Ersparnis von 94,8%. Testobjekt 2 liefert die schlechteste Ersparnis mit 69,4%. Die Server der Gruppe 2 profitieren im Vergleich zu Gruppe 1 überproportional stark von einer reduzierten Datenkomplexität. Dies hat aber auch zur Folge, daß die komplexere Struktur von ADC 2 keinen Gewinn in dieser Gruppe liefert. ADC 2 verlängert die Antwortzeit um durchschnittlich 34,4%. ADC 1 ist ebenfalls mit einer um durchschnittlich 170,8% gesteigerten Antwortzeit vergleichsweise schlecht.

Die in Abbildung 6.13 dargestellte Gruppe 1 liefert mit bei den statischen Konzepten durchschnittlich 31,5% (VAVO) und 55,7% (AVOC) bessere Antwortzeiten als das bestehende Verfahren VVO. Die beste Antwortzeit liefert AVOC bei Testobjekt 1 mit einer um 62% reduzierten Antwortzeit. Im Falle von Testobjekt 4 existiert mit diesem Ansatz eine um 44,6% kleinere Antwortzeit. Die eingeschaltete Kompression führt jedoch bei AVOCC insgesamt zu einer im

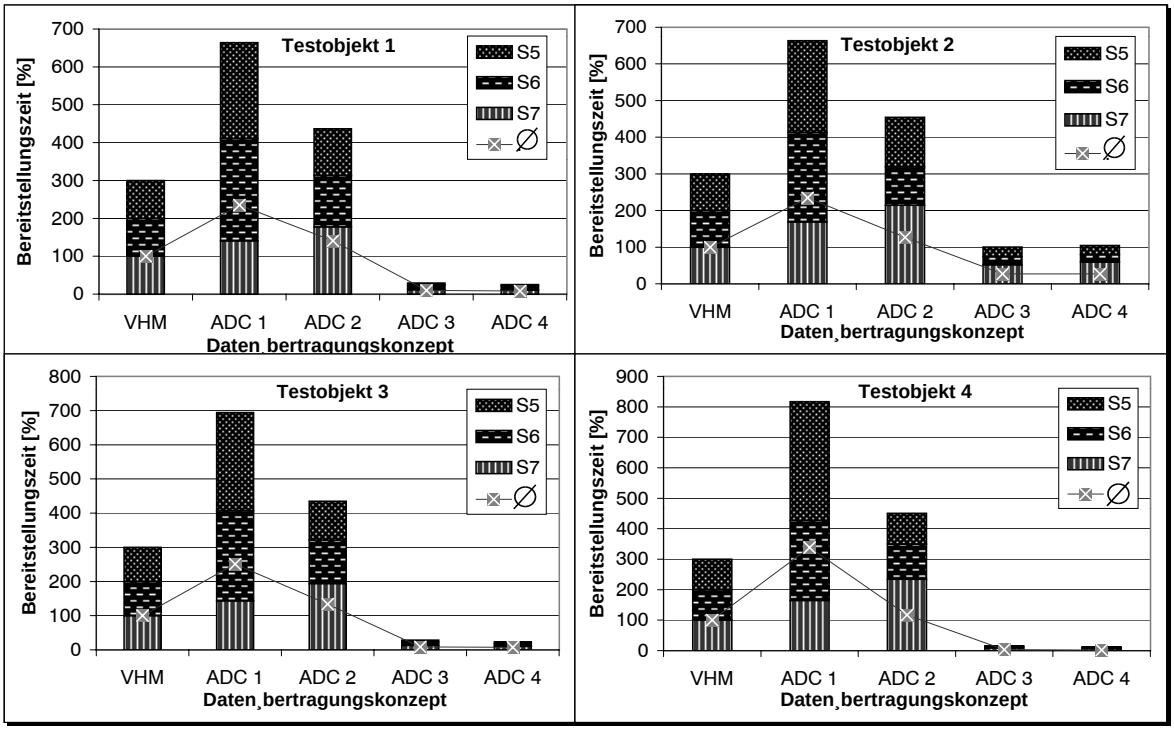


Abbildung 6.8: Bereitstellungszeitvergleich der dynamischen Daten-Container (Gruppe 2)

Schnitt 54,8% schlechteren Antwortzeit. Im Gegensatz zur Bereitstellungszeit verursacht die Dekompression einen vollständigen Verlust des Gewinns, der durch die Reduktion der Datenmenge bei der Übertragungszeit entsteht.

Die Server der Gruppe 2 (Abbildung 6.14) zeigen bei den statischen Daten-Containern selbst mit eingeschalteter Kompression eine Einsparung von Antwortzeit im Umfang von 79,2%. Dieses Ergebnis war zu erwarten, da die Server der Gruppe 2 auch bei der Anwendung von Reflection in dynamischen Konzepten noch einen deutlichen Leistungsgewinn zeigen. Die Übertragungszeit dominiert bei diesen Servern und wird maßgeblich durch die Komplexität der zu transportierenden Daten bestimmt. Bei VAVO beläuft sich die Ersparnis auf durchschnittlich 52,6% und bei AVOC auf sehr gute 95,1%. Diese Werte zeigen erneut, wie sinnvoll es ist, die Container-interne Datenstruktur für die Datenübertragung zu optimieren. Dies ist insbesondere von Vorteil, wenn die Implementierung der Kommunikationsschicht eines Applikations-Servers mit komplexen Datenstrukturen und hohen Datenmengen Probleme verursacht.

Bei Servern der Gruppe 2 zeigt sogar die dynamische Implementierung ADC 4 signifikante Vorteile gegenüber dem statischen Ansatz VVO. Im Vergleich existiert ein durchschnittlicher Gewinn von 41,1%. Tabelle 6.6 zeigt die erhaltenen Antwortzeiten.

Aufgrund der dargestellten Ergebnisse kann festgehalten werden, daß mit den Konzepten dieser Arbeit signifikante Verbesserungen der Übertragungs-, Bereitstellungs- und Antwortzeit erzielt werden können. Bei Applikations-Servern mit optimierter Kommunikationsschicht, die in einem schnellen Netzwerk betrieben werden, rückt die Zeit zum Verpacken und Entpacken der

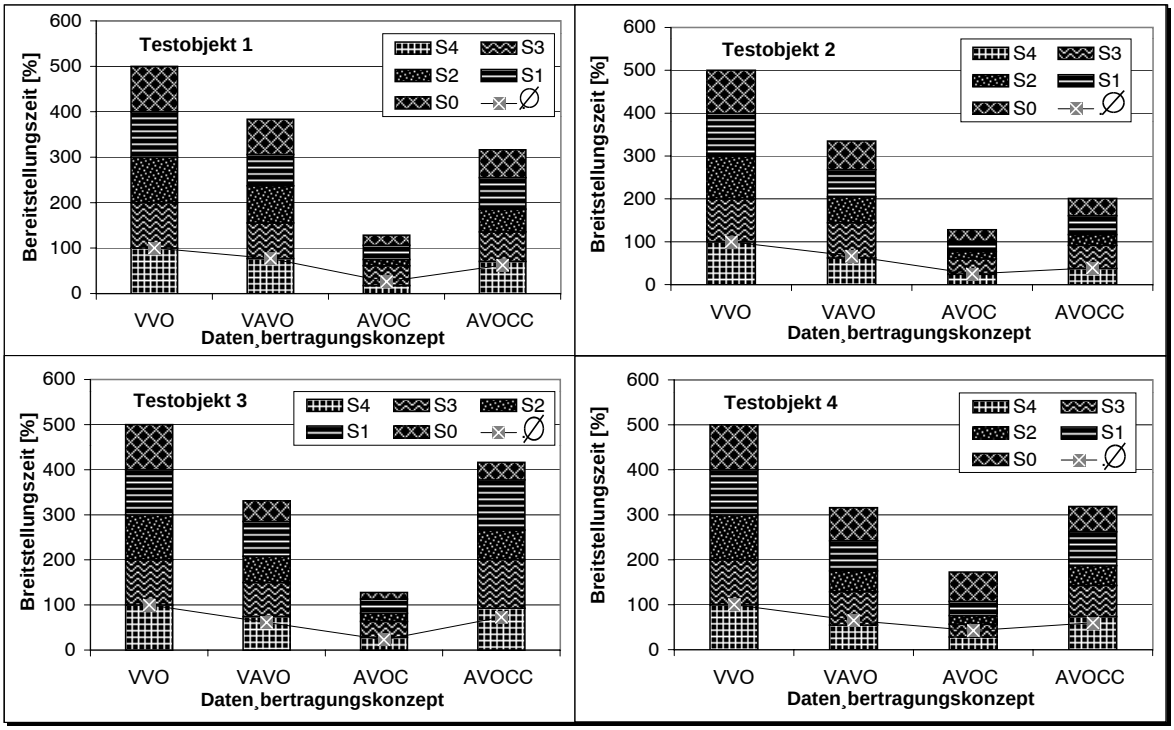


Abbildung 6.9: Bereitstellungszeitvergleich der statischen Daten-Container (Gruppe 1)

Daten in einen Daten-Container in den Mittelpunkt, da diese vergleichsweise stark ins Gewicht fällt. Bei Servern mit einer nicht ausreichend optimierten Kommunikationsschicht steht eine Optimierung der Übertragungszeit im Vordergrund, da sich diese bei Übertragungsvorgängen sehr stark auswirkt.

Eine endgültige Beurteilung der verschiedenen Datenübertragungskonzepte wird jedoch erst möglich, wenn ein Test mit mehreren Clients durchgeführt wird. In diesem Fall konkurrieren die Clients um die Ressourcen des Servers und es zeigt sich, ob sich die erzielten Einsparungseffekte in dieser Situation positiv auf das Verhalten des Gesamtsystems auswirken. Im folgenden Abschnitt sind die Ergebnisse eines Lasttests zusammengefaßt.

Konzept	Testobjekt 1	Testobjekt 2	Testobjekt 3	Testobjekt 4
VVO	100	100	100	100
ADC 4	69,1	51,6	84,0	30,8

Tabelle 6.6: Einzeltest: Vergleich zwischen statischem und dynamischem Verfahren (Angaben in Prozent)

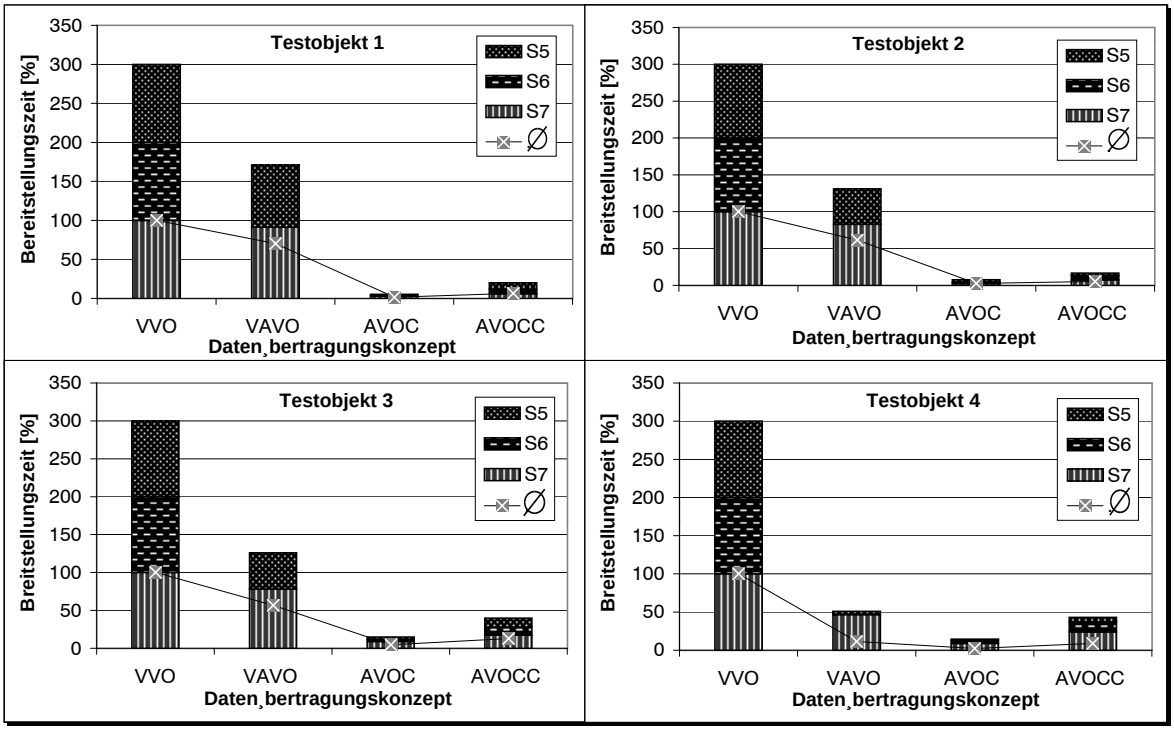


Abbildung 6.10: Bereitstellungszeitvergleich der statischen Daten-Container (Gruppe 2)

6.2.3 Lasttest

Der Lasttest soll zeigen, wie sich die unterschiedlichen Konzepte im Umfeld stark belasteter Applikations-Server verhalten. Dabei konkurrieren viele Clients um die im System vorhandenen Ressourcen. Durch die Reduktion der Datenmenge und -komplexität der zu übertragenden Daten wird die Kommunikationsschicht des Applikations-Servers und das Netzwerk entlastet. Somit kann mit den im Rahmen dieser Arbeit erstellten optimierten Daten-Containern ein höherer Transaktionsdurchsatz in einem EJB-System erzielt werden. Zur Durchführung eines Lasttests wurde in jedem getesteten Server eine zustandslose Session-Bean, im folgenden *Test-Bean* genannt, installiert. Die Test-Bean verhält sich analog zu der Session-Bean für Einzeltests, die bereits in Abschnitt 6.2.2 beschrieben wurde. Die am Lasttest teilnehmenden virtuellen Clients wurden auf unterschiedlichen Rechnern mittels des in Anhang B beschriebenen Testwerkzeugs gesteuert. Jeder virtuelle Client führte dabei ununterbrochen Anfragen an die Test-Bean durch. D.h. es wurde keine Pause zwischen zwei Anfragen eingefügt. Zur sinnvollen Durchführung des Lasttests in der vorhandenen Testumgebung, wurde die Anzahl der Testobjekte, deren Daten übertragen werden sollen, bei den Testobjekten 2 bis 4 herabgesetzt. Dies war Aufgrund der hohen Datenmengen und den daraus resultierenden langen Testlaufzeiten erforderlich. Die für den Lasttest verwendeten Objektanzahlen sind in Tabelle 6.7 dargestellt.
Beim Lasttest wurde der Transaktionsdurchsatz bei einer festen Anzahl von 20 Clients für jedes Datenübertragungskonzept bestimmt. In Anlehnung an die Tests mit einem Client wurden die folgenden Größen bestimmt:

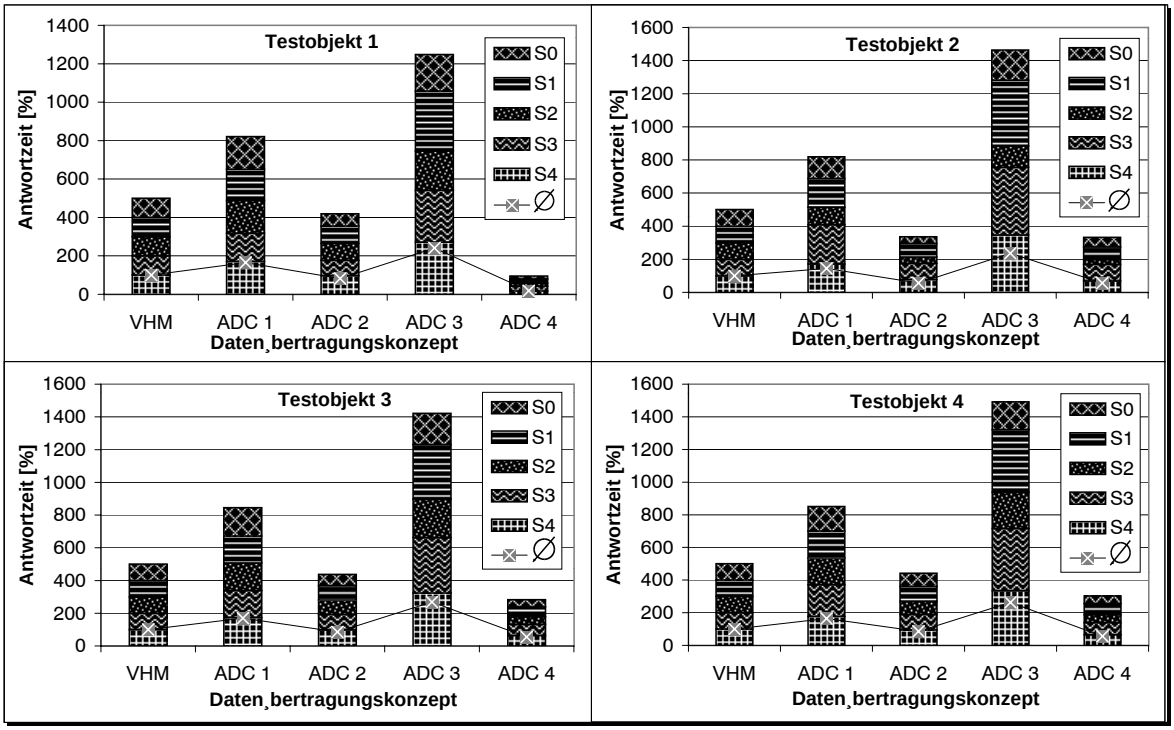


Abbildung 6.11: Antwortzeitvergleich der dynamischen Daten-Container (Gruppe 1)

- **Datenmenge**
- **Transaktionen/Sekunde (Kommunikation)**: Diese Größe gibt die Anzahl der Transaktionen pro Sekunde bei reinen Kommunikationsvorgängen der verschiedenen Daten-Container an. Dabei werden keine Erstellungsvorgänge auf dem Server und keine Auslesevorgänge im Client berücksichtigt. Ein hoher Transaktionsdurchsatz kennzeichnet Konzepte die schnell übertragen werden können und damit implizit bessere Übertragungszeiten besitzen.
- **Transaktionen/Sekunde (Erstellung)**: Diese Größe berücksichtigt neben dem erforderlichen Übertragungsvorgang zusätzlich die Vorgänge, die erforderlich sind, um die Daten-Container zu erstellen und mit Daten zu belegen. Ein hoher Transaktionsdurchsatz kennzeichnet Konzepte, die Daten-Container schnell bereitstellen und übertragen können und

Testobjekt	Anzahl
1	500
2	200
3	100
4	50

Tabelle 6.7: Testobjektanzahl im Lasttest

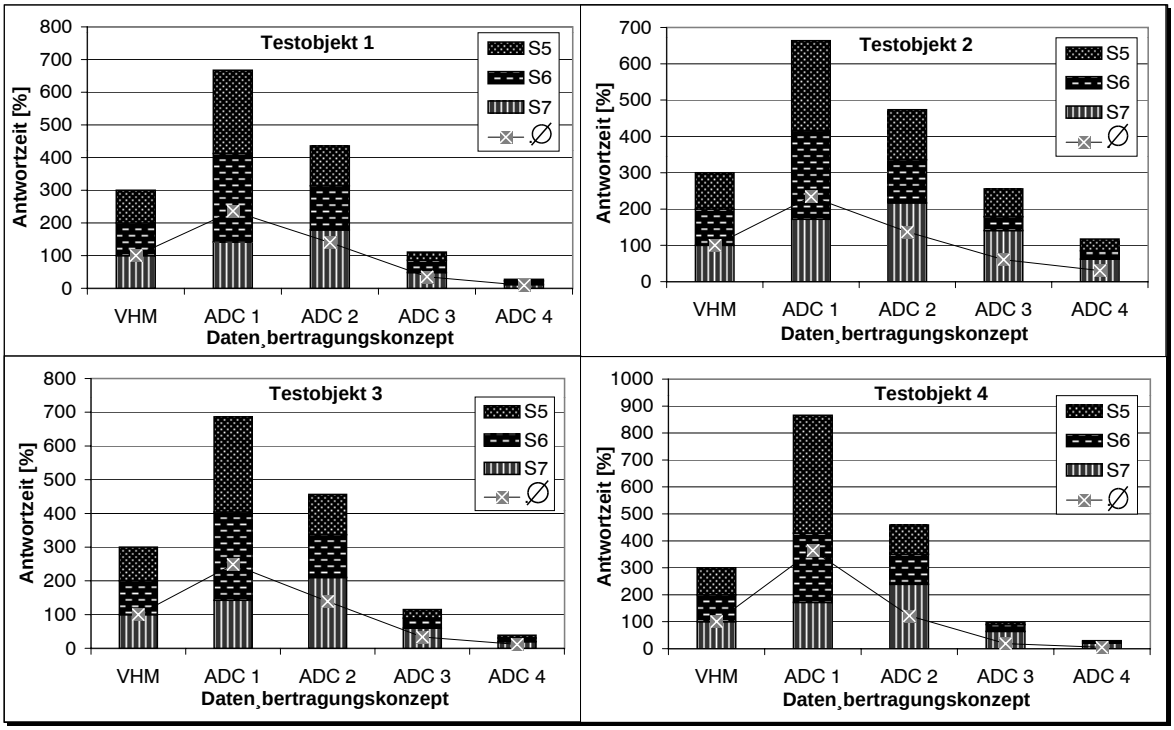


Abbildung 6.12: Antwortzeitvergleich der dynamischen Daten-Container (Gruppe 2)

damit implizit bessere Erstellungszeiten besitzen.

- **Transaktionen/Sekunde (Auslesen)**: Diese Größe berücksichtigt zusätzlich zur Erzeugung und Übertragung von Daten-Containern den Auslesevorgang im Client. Ein hoher Transaktionsdurchsatz kennzeichnet Daten-Container die vom Server schnell bereitgestellt und deren Inhalt schnell auf dem Client ausgelesen werden können. Damit besitzen diese Daten-Container implizit eine bessere Erstellungszeit.

Datenmenge

In Abbildung 6.15 sind die anfallenden Datenmengen für jedes Datenübertragungskonzept nach der Serialisierung dargestellt.
Die Änderung der Testobjektanzahl führt bis auf Testobjekt 1 zu einer geringeren Datenmenge. Es gelten weiterhin die Aussagen, die bereits bei der Diskussion der Datenmenge in den Einzeltests (vgl. Abbildung 6.2, Abschnitt 6.2.2) vorgenommen wurden.

Transaktionsdurchsatz (Kommunikation)

In Abbildung 6.16 sind die Ergebnisse des Lasttests bei reinen Datenübertragungsvorgängen dynamischer Konzepte auf Servern der Gruppe 1 dargestellt. Wie zu erwarten, schneidet der

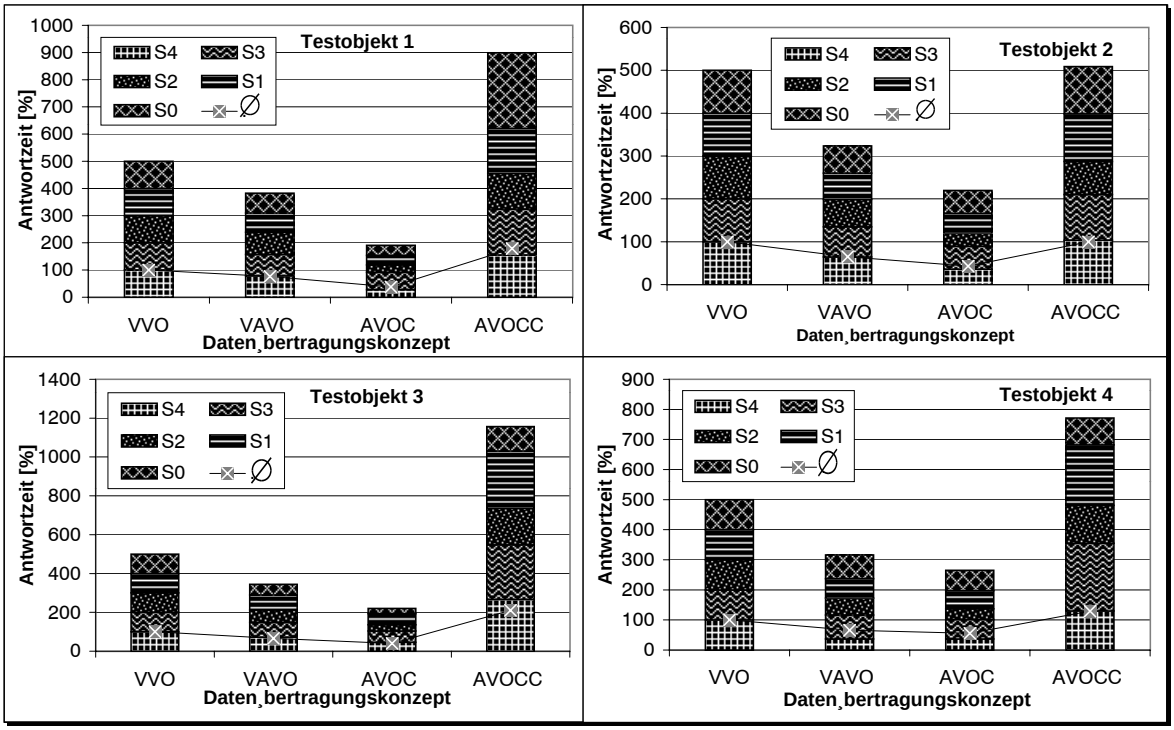


Abbildung 6.13: Antwortzeitvergleich der statischen Daten-Container (Gruppe 1)

am wenigsten optimierte Daten-Container ADC 1 mit durchschnittlich 36,8% weniger Transaktionen am schlechtesten ab. ADC 2 erhöht den Transaktionsdurchsatz gegenüber VHM um durchschnittlich 38,2%. Die optimierten ADC 3 bzw. ADC 4 erhöhen den Transaktionsdurchsatz um durchschnittlich 596,1%. Im besten Fall beträgt die Steigerung bei Testobjekt 1, das sich am besten optimieren läßt, 1342,9%. Im schlechtesten Fall beträgt die Steigerung 187,4% (Testobjekt 2).

Die in Abbildung 6.17 dargestellte Gruppe 2 der Applikations-Server zeigt bei den dynamischen Datenübertragungsverfahren bei ADC 1 durchschnittlich 38,7% weniger Transaktionen und ebenso bei ADC 2 durchschnittlich 59,0% weniger Transaktionen. Wie bereits bei den Einzeltests festgestellt, wirkt sich die Komplexität der Datenstruktur von ADC 2 bei Servern dieser Gruppe ebenfalls negativ aus. Der grundsätzliche Unterschied zu Servern der Gruppe 1 drückt sich wiederum auch deutlich in den absoluten Werten aus. So beträgt der Transaktionsdurchsatz bei Servern der Gruppe 1 durchschnittlich 110,4 (ADC 3/4) gegenüber 22,6 bei Servern der Gruppe 2.

Die Abbildungen 6.18 und 6.19 zeigen die Ergebnisse der statischen Konzepte für die Server der Gruppe 1 und die Server der Gruppe 2. In Gruppe 1 existiert eine durchschnittliche Erhöhung des Transaktionsdurchsatzes um 91,6% (VAVO) und 407,9% (AVOC). Auffällig ist die überdurchschnittlich große Reaktion des Servers S2, der die am schlechtesten optimierte Kommunikationsschicht der Gruppe 1 besitzt und damit von der Vereinfachung der Datenstruktur am meisten profitiert. Die Komprimierung von AVOC wurde nur noch bei Testobjekt 1 untersucht,

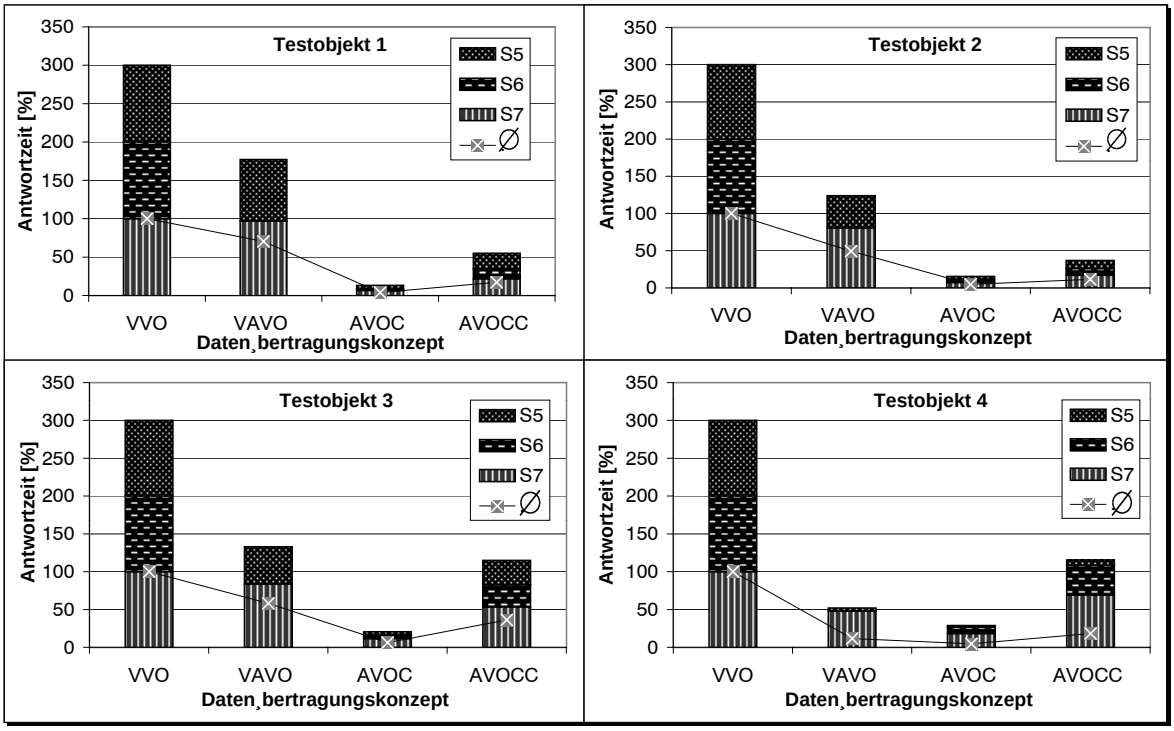
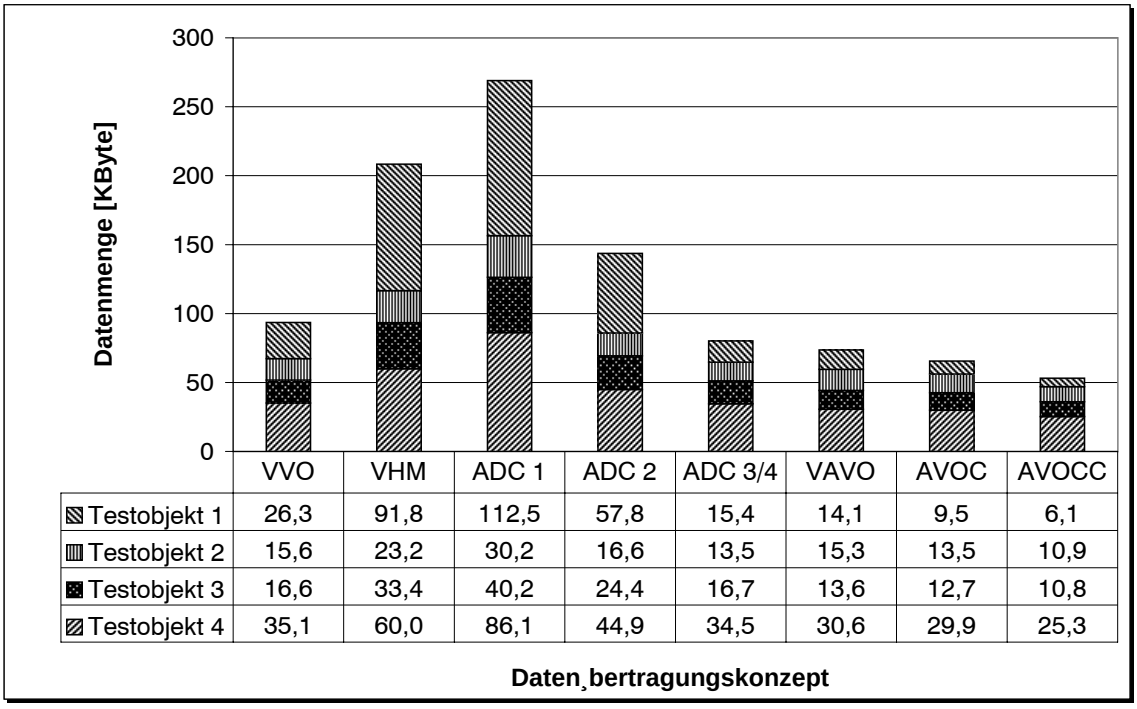


Abbildung 6.14: Antwortzeitvergleich der statischen Daten-Container (Gruppe 2)

da hier der Kompressionsgrad am höchsten ausfällt. Es ergibt sich ein um 687,1% gesteigerter Transaktionsdurchsatz gegenüber 550,4% ohne Kompression. Die Server der Gruppe 2 erfahren beim Ansatz VAVO nur eine Steigerung des Transaktionsdurchsatzes um durchschnittlich 21,8% (die VAVO Implementierung war auf Server S6 nicht lauffähig). Dagegen erfährt der Transaktionsdurchsatz beim Ansatz AVOC eine signifikante Steigerung um durchschnittlich 2642,6%. Mit eingeschalteter Kompression erreicht der Transaktionsdurchsatz bei Testobjekt 1 eine Steigerung von 6702,5% gegenüber 5501,0% ohne Kompression. Die starken Steigerungen bestätigen erneut die geringe Komplexität des internen Datenformats im Container. Die Server der Gruppe 2 verhalten sich bei der Übertragung von Daten geringer Komplexität (`byte[]`) nahezu gleich wie Server der Gruppe 1. Dies wurde bereits in Kapitel 4 erläutert.
Zusammenfassend kann festgehalten werden, daß der Transaktionsdurchsatz mit den Konzepten in dieser Arbeit bei Kommunikationsvorgängen signifikant gesteigert werden kann.

Transaktionsdurchsatz (Erstellung)

Aus den Abbildungen 6.20 und 6.21 ist für die dynamischen Konzepte ersichtlich, wie der Transaktionsdurchsatz sinkt, wenn die Erzeugung der Daten-Container mit hinzugenommen wird. Bei den Servern der Gruppe 1 verbleibt beim ADC 2 ein um durchschnittlich 3,5% geringfügig höherer Transaktionsdurchsatz. Die Steigerung bei ADC 3 beträgt durchschnittlich 68,1%. Die Leistungseinbuße ist hier wiederum auf die Verwendung von Java-Reflection



	VVO	VHM	ADC 1	ADC 2	ADC 3/4	VAVO	AVOC	AVOCC
Testobjekt 1	26,3	91,8	112,5	57,8	15,4	14,1	9,5	6,1
Testobjekt 2	15,6	23,2	30,2	16,6	13,5	15,3	13,5	10,9
Testobjekt 3	16,6	33,4	40,2	24,4	16,7	13,6	12,7	10,8
Testobjekt 4	35,1	60,0	86,1	44,9	34,5	30,6	29,9	25,3

Abbildung 6.15: Lasttest: Datenmenge der Container nach der Serialisierung

zurückzuführen, die zur Entnahme der Daten aus den Testobjekten in den Containern verwendet wird. Die Implementierung ohne Reflection in Form von ADC 4 zeigt im Gegensatz dazu eine sehr gute Steigerung von durchschnittlich 332,7%. Bei Testobjekt 1 fällt die Steigerung mit 677,4% am höchsten aus. Bei Testobjekt 3 stellt sich die geringste Steigerung mit 196,5% ein. Bei den Servern der Gruppe 2 existiert selbst noch bei ADC 3 eine Steigerung des Transaktionsdurchsatzes. Die Steigerung beträgt durchschnittlich 109,9% bei ADC 3 und 188,5% bei ADC 4. Die Konzepte profitieren hier weiterhin überdurchschnittlich von der stark vereinfachten Übertragung der Datenstrukturen.

Die Werte für die statischen Konzepte zeigen in den Abbildungen 6.22 und 6.23, daß der Ansatz VAVO bei Servern der Gruppe 1 mit einer durchschnittlichen Steigerung des Transaktionsdurchsatzes um 95% nahezu die selbe Steigerungsrate besitzt, wie bei der bloßen Übertragung der Daten-Container. Beim Ansatz AVOC existiert noch eine durchschnittliche Steigerung von 188,8%. Im Gegensatz zu VAVO ist das Hinzufügen von Daten bei AVOC aufwendiger, da hier eine Umwandlung in ein anderes internes Datenformat vollzogen wird. Die eingeschaltete Kompression im Falle von Testobjekt 1 führt bei AVOC zu einem um insgesamt 17% gesunkenen Transaktionsdurchsatz. Dies zeigt, daß der Aufwand für die Kompression gegenüber dem erzielten Gewinn, der durch eine verringerte Datenmenge existiert, zu groß ist. Erneut wird deutlich, daß bereits durchgeführte Optimierungen, die auf die Datenübertragung abgestimmt sind, deutlich wirksamer sind, als die Anwendung eines normalen Kompressionsalgorithmus.

Bei den Servern der Gruppe 2 bleibt die Steigerung bei VAVO mit durchschnittlich 25,1% ebenfalls nahezu identisch zur reinen Übertragung der Container. Der Ansatz AVOC zeigt zwar eine

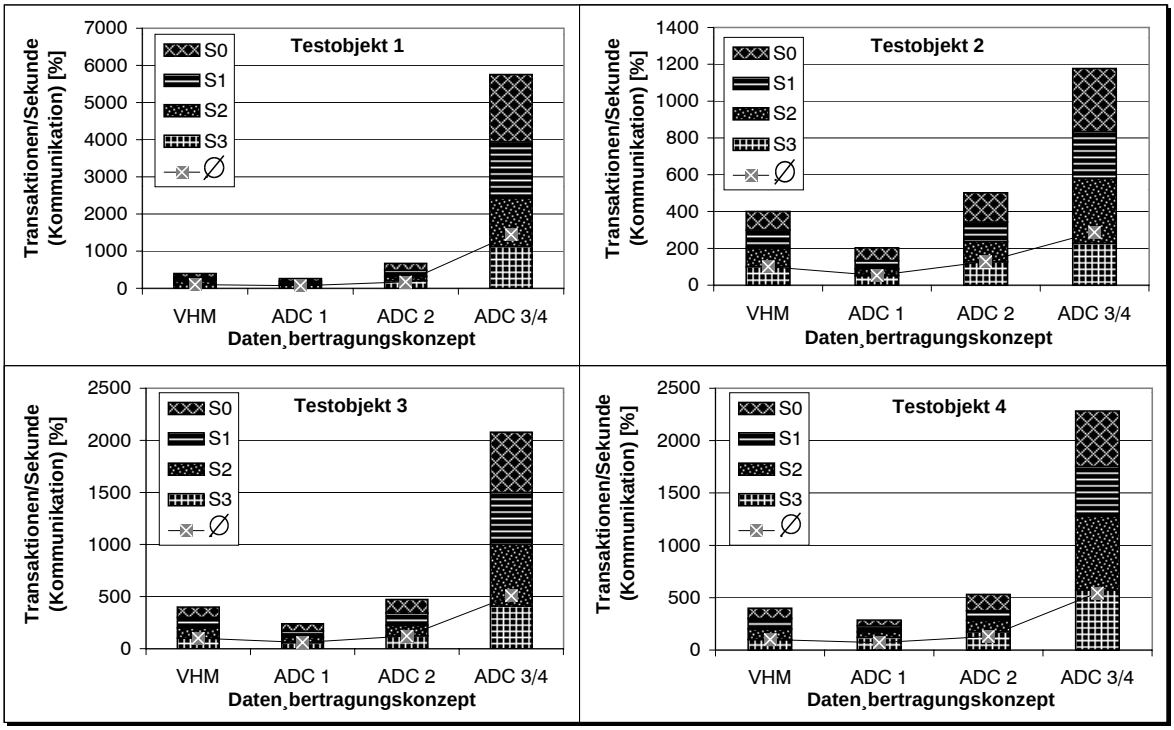


Abbildung 6.16: Transaktionsdurchsatz (Kommunikation) der dynamischen Daten-Container (Gruppe 1)

gesunkene aber dennoch deutliche durchschnittliche Steigerung des Transaktionsdurchsatzes um 1162%. Im Gegensatz zu den Servern der Gruppe 1 findet im Falle von Testobjekt 1 bei eingeschalteter Kompression immer noch eine Erhöhung des Transaktionsdurchsatzes um 411,5% statt. Diese ist jedoch gegenüber einer Steigerung von 2193,1% ohne Kompression sehr gering. Zusammenfassend kann festgehalten werden, daß mit den Konzepten dieser Arbeit eine deutliche Steigerung des Transaktionsdurchsatzes bei der Bereitstellung von Daten-Containern erzielt werden kann.

Transaktionsdurchsatz (Auslesen)

Für die dynamischen Konzepte in Servern der Gruppe 1 (Abbildung 6.24) existiert beim Ansatz ADC 3 trotz der Anwendung von Java-Reflection eine Steigerung des Transaktionsdurchsatzes um durchschnittlich 25,5%. Diesem Faktor steht eine Steigerung um durchschnittlich 315,5% bei ADC 4 gegenüber, der ohne Reflection auskommt. Die übrigen Ansätze ADC 1 und ADC 2 verringern den Transaktionsdurchsatz um durchschnittlich 42% bzw. 4%. Dies ist grundsätzlich auf die Verwendung von Java-Reflection zum Befüllen der Daten-Container zurückzuführen. Im Falle von ADC 2 reichen die eingesparte Datenmenge und die Komplexitätsreduktion nicht aus, um den Verlust durch das Befüllen des Containers mit Reflection aufzufangen. Im Falle von ADC 1 kommt die höhere Komplexität und Datenmenge als negativ beeinflussender Faktor hinzu. An dieser Stelle muß allerdings auch erwähnt werden, daß die Entnahme der Daten

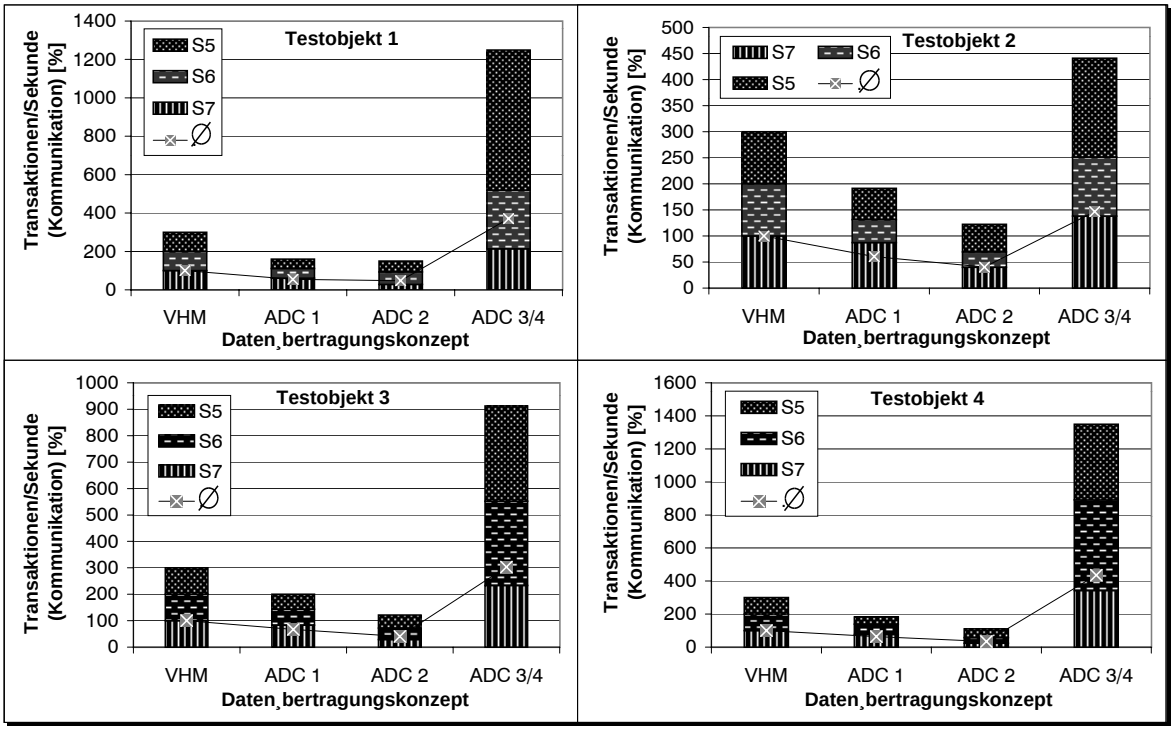


Abbildung 6.17: Transaktionsdurchsatz (Kommunikation) der dynamischen Daten-Container (Gruppe 2)

mittels Reflection in den verwendeten Containern nicht optimiert ist, d.h. jedes Objekt wird neu analysiert auch wenn dessen Typ schon einmal untersucht wurde.

Die Server der Gruppe 2 (Abbildung 6.25) zeigen bei dem dynamischen Ansatz ADC 3 eine Steigerung des Transaktionsdurchsatzes um durchschnittlich 92,5%. Der Ansatz ADC 4 führt dagegen zu einem um durchschnittlich 181,1% gesteigerten Transaktionsdurchsatz. Die Ansätze ADC 1 und ADC 2 führen zu einem um durchschnittlich 40,5% bzw. 57,9% geringeren Transaktionsdurchsatz. Dieses Verhalten ist neben der Verwendung von Java-Reflection auf die höhere Datenmenge bei ADC 1 und eine für die Kommunikationsschicht der Gruppe 2 nicht ausreichend optimierte interne Datenstruktur von ADC 2 zurückzuführen.

Die statischen Ansätze (Abbildung 6.26) führen insgesamt zu deutlich gesteigerten Transaktionsdurchsätzen. Während die Server der Gruppe 1 mit dem Ansatz VAVO eine Steigerung des Transaktionsdurchsatzes um 96,1% erfahren, führt das selbe Konzept bei Servern der Gruppe 2 (Abbildung 6.27) zu einer Steigerung von durchschnittlich 23,7% (auf dem Server S6 war die Implementierung VAVO nicht lauffähig). Die Implementierung AVOC führt bei Servern der Gruppe 1 zu einer durchschnittlichen Erhöhung des Transaktionsdurchsatzes um 192,9%. Bei Servern der Gruppe 2 führt die Implementierung AVOC zu einer durchschnittlichen Steigerung um 1169,7%. Die Verwendung der Kompression (AVOCC) führt bei Servern der Gruppe 1 zu einem um 12,2% gesunkenen Transaktionsdurchsatz. Dagegen führt die Kompression bei Servern der Gruppe 2 immer noch zu einer Steigerung um 454,5%. Die Server der Gruppe 2 profitieren überproportional von der gesunkenen Datenmenge und -komplexität. Im Vergleich

zu AVOC ist jedoch ebenfalls eine deutliche Senkung des Transaktionsdurchsatzes sichtbar, der auf den Aufwand des Kompressionsalgorithmus zurückzuführen ist.
Zusammenfassend kann festgehalten werden, daß der Transaktionsdurchsatz mit den Konzepten dieser Arbeit signifikant gesteigert werden kann. Die Steigerung beruht auf der Reduktion der Datenkomplexität und der Datenmenge. Bei der Implementierung von effizienten Daten-Containern müssen die Eigenschaften der Kommunikationsschicht des zugrundeliegenden Applikations-Servers berücksichtigt werden. Insgesamt ist auffällig, daß die Verarbeitung von komplexen Java-Datenstrukturen für Server, die auf CORBA/IIOP basieren, wesentlich aufwendiger ist, als für Server die auf RMI oder hochoptimierten proprietären Protokollen basieren. Mit den Konzepten in dieser Arbeit können diese Probleme jedoch insgesamt umgangen werden, indem eine interne Datenstruktur der Transport-Container mit geringer Komplexität gewählt wird.

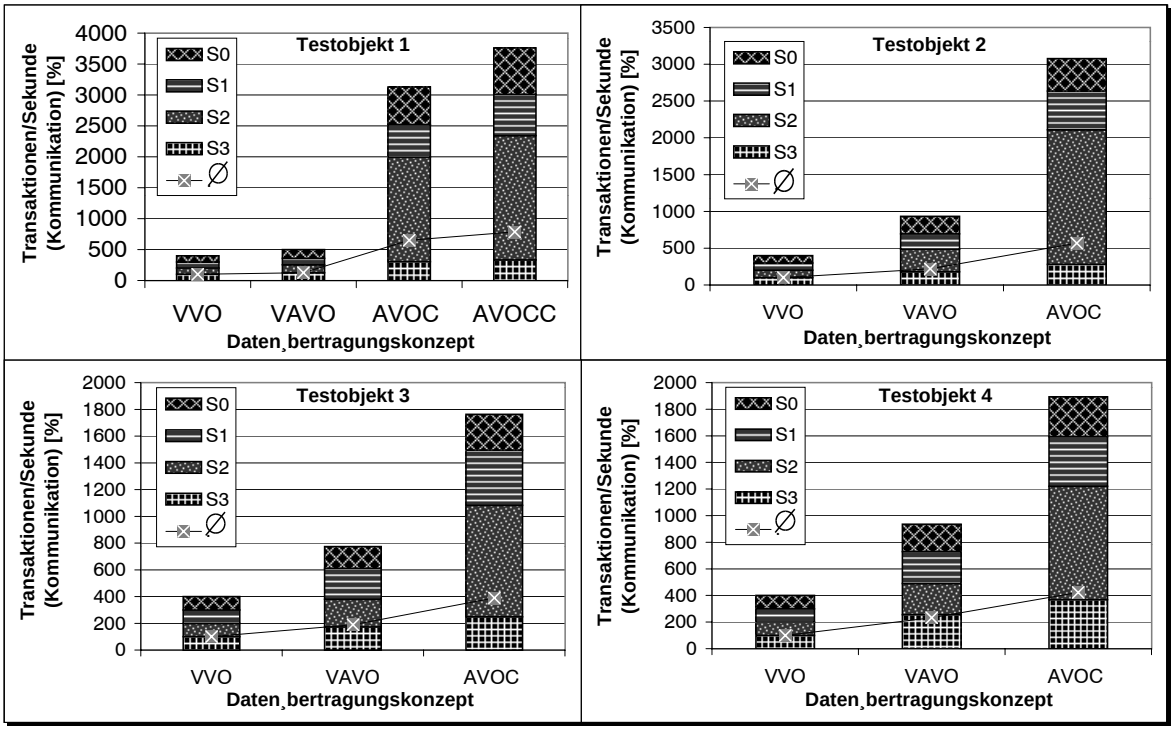


Abbildung 6.18: Transaktionsdurchsatz (Kommunikation) der statischen Daten-Container (Gruppe 1)

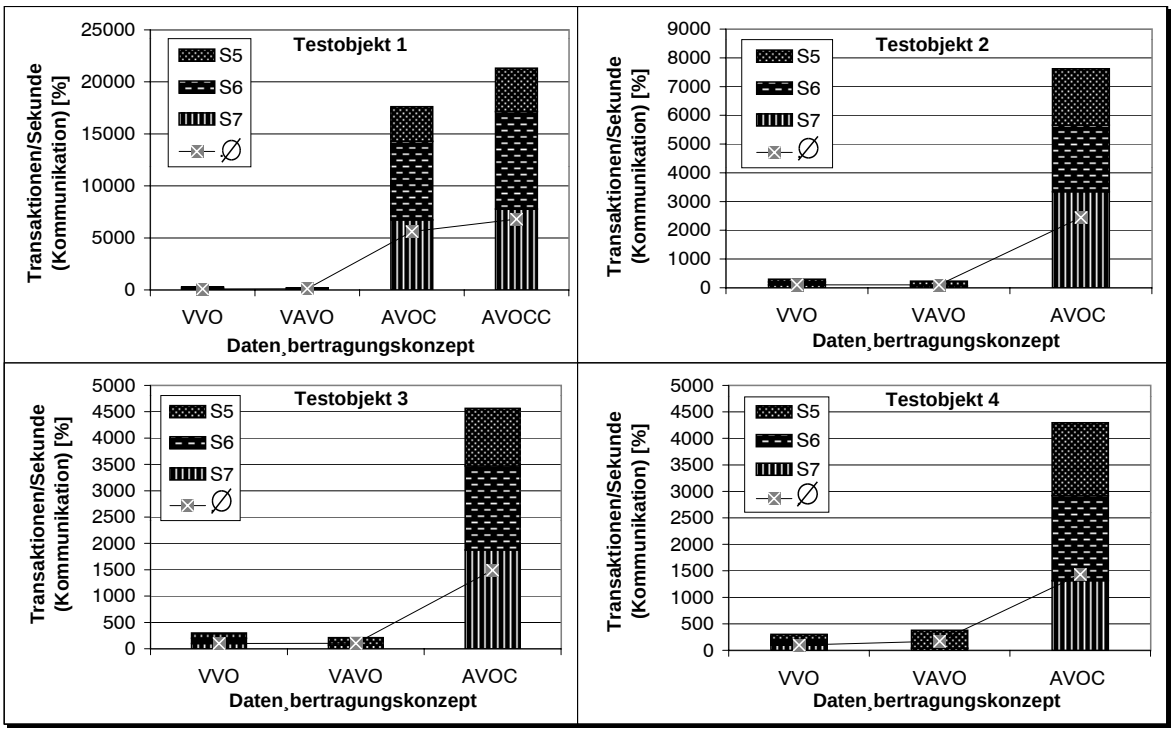


Abbildung 6.19: Transaktionsdurchsatz (Kommunikation) der statischen Daten-Container (Gruppe 2)

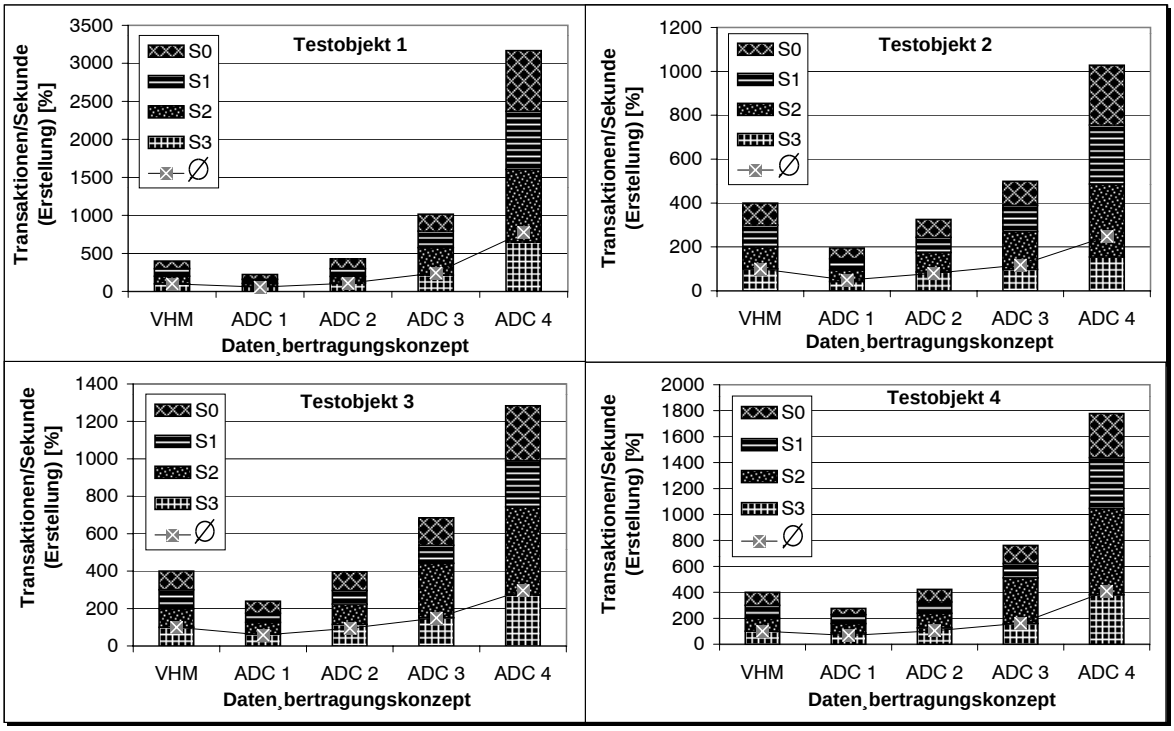


Abbildung 6.20: Transaktionsdurchsatz (Erstellung) der dynamischen Daten-Container (Gruppe 1)

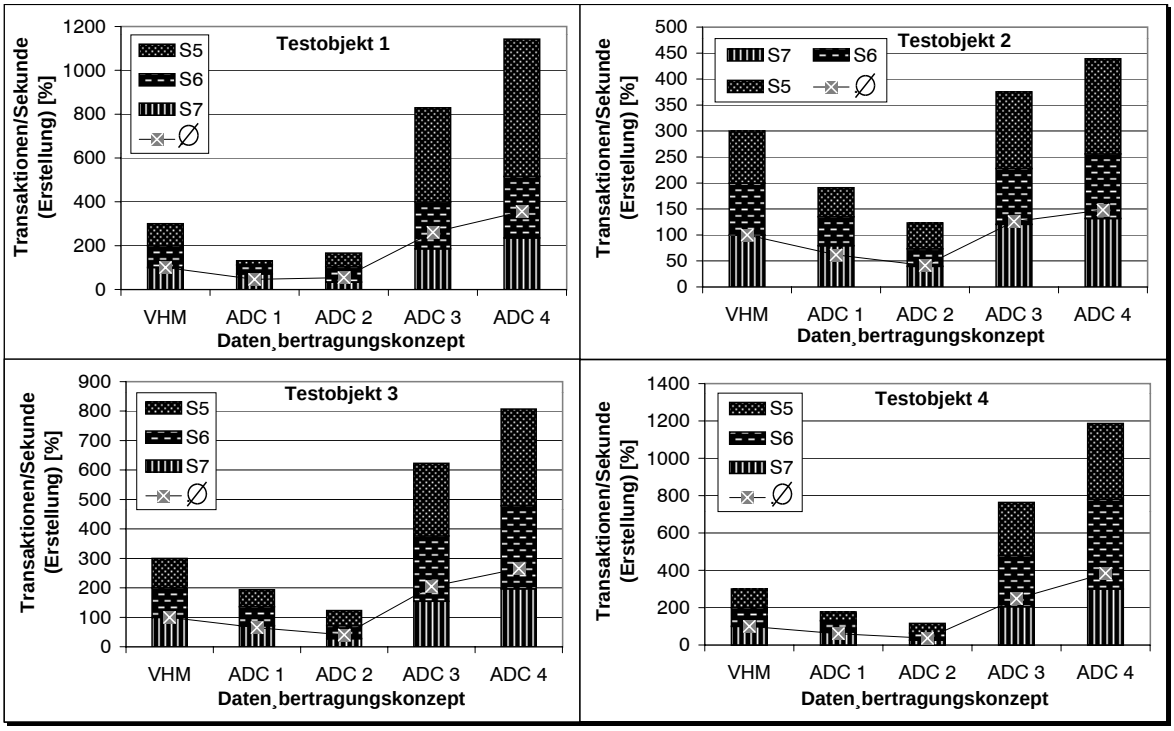


Abbildung 6.21: Transaktionsdurchsatz (Erstellung) der dynamischen Daten-Container (Gruppe 2)

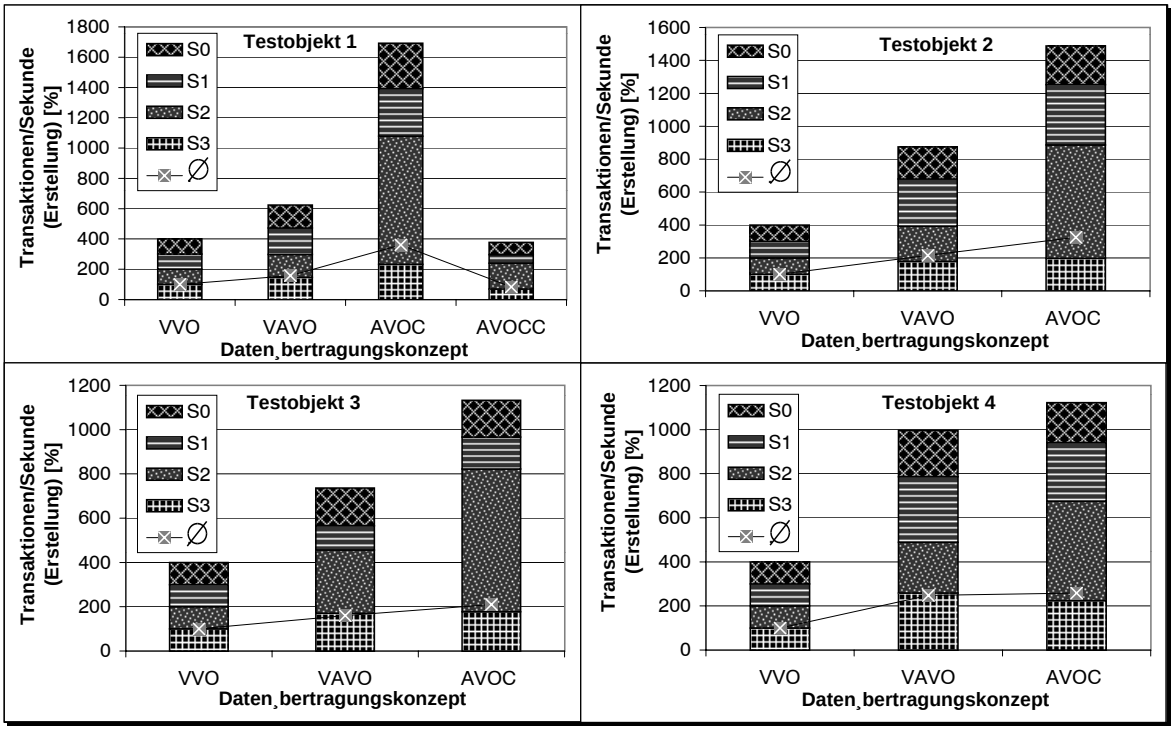


Abbildung 6.22: Transaktionsdurchsatz (Erstellung) der statischen Daten-Container (Gruppe 1)

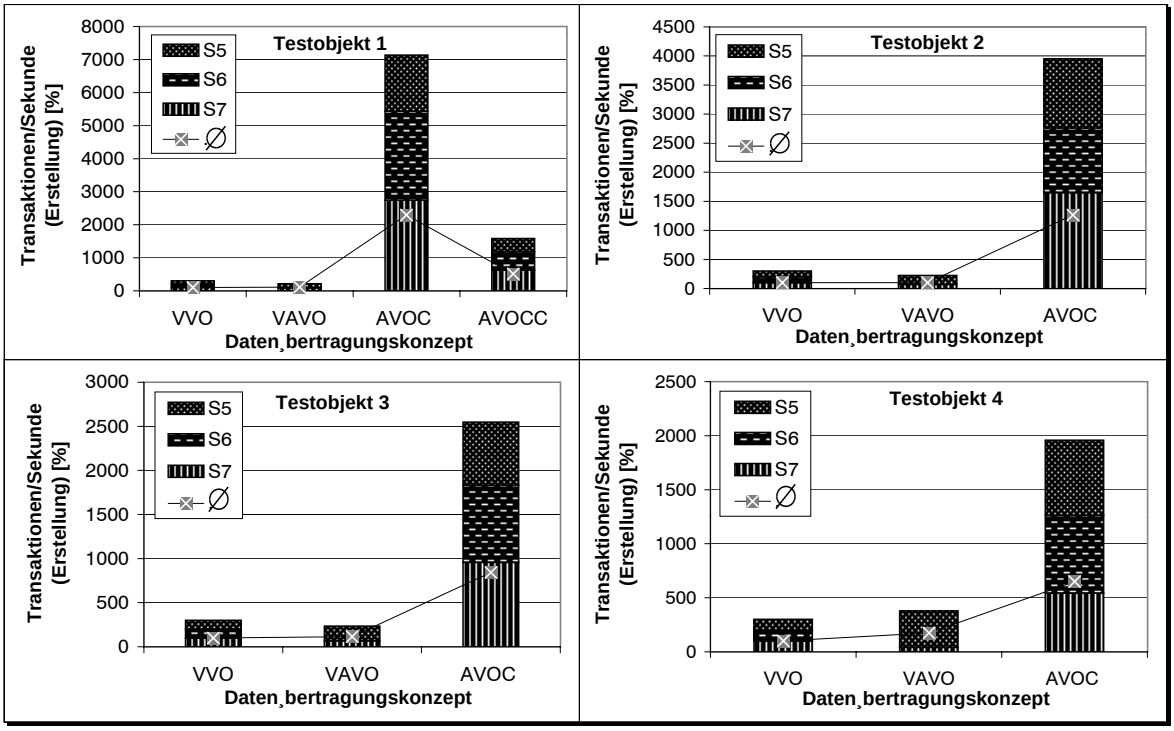


Abbildung 6.23: Transaktionsdurchsatz (Erstellung) der statischen Daten-Container (Gruppe 2)

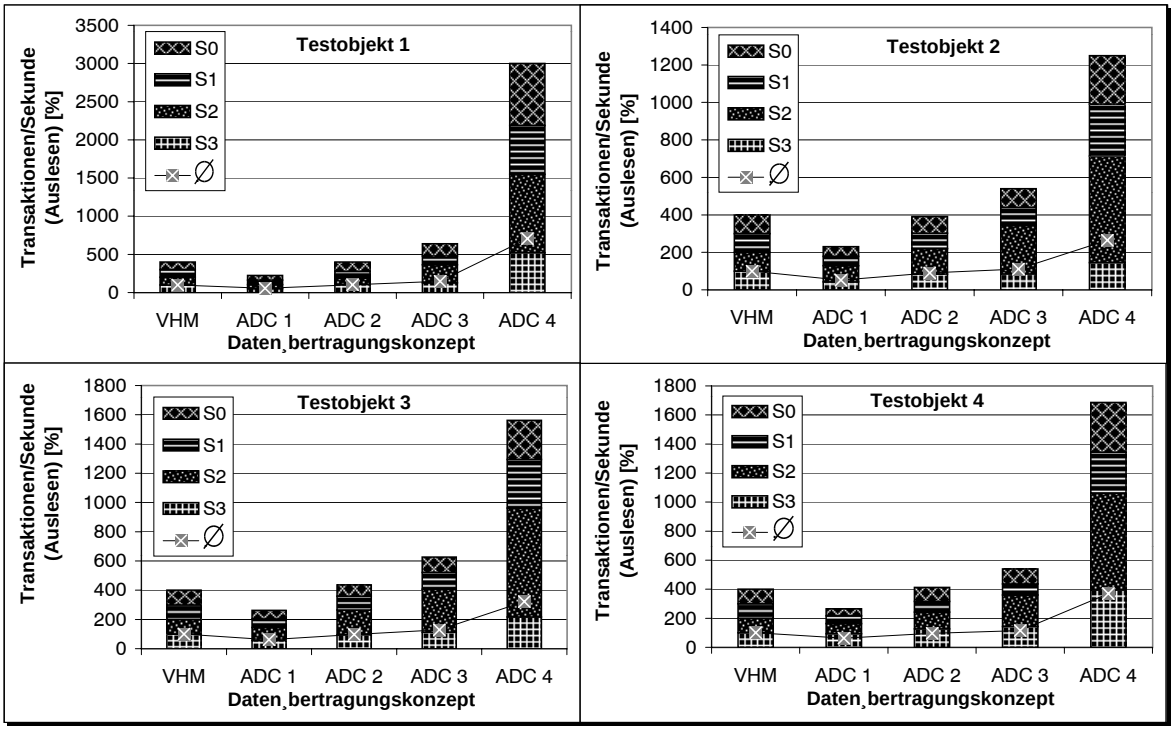


Abbildung 6.24: Transaktionsdurchsatz (Auslesen) der dynamischen Daten-Container (Gruppe 1)

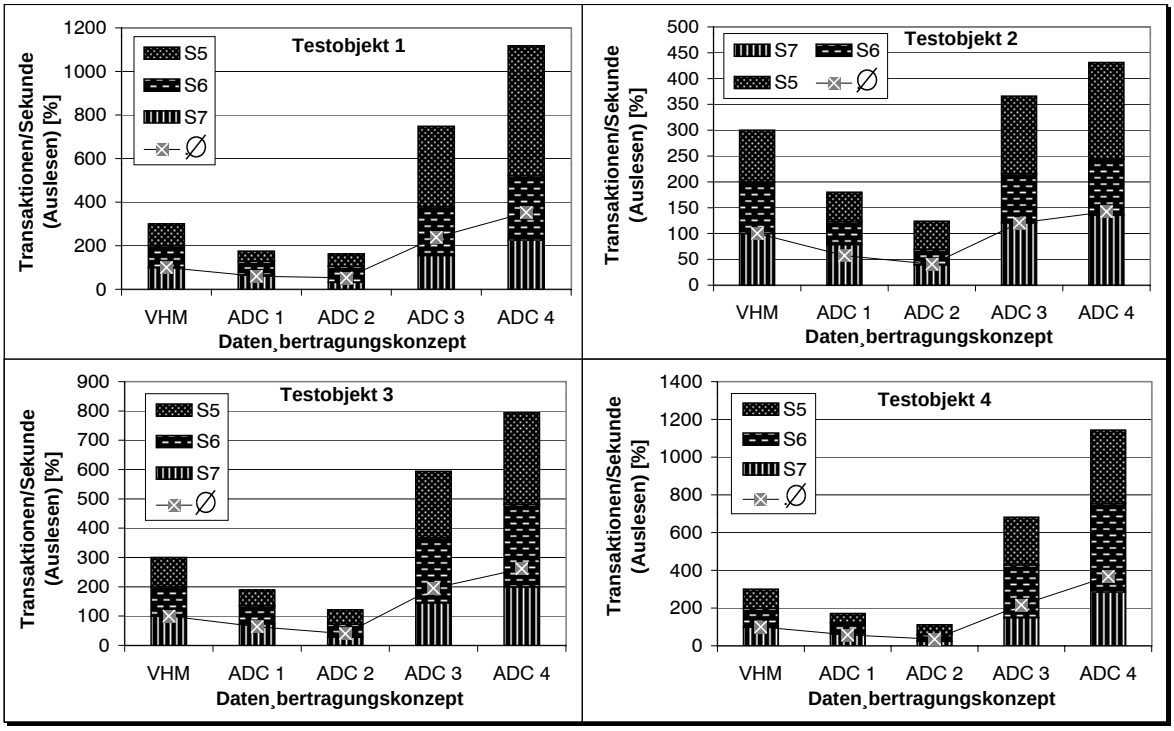


Abbildung 6.25: Transaktionsdurchsatz (Auslesen) der dynamischen Daten-Container (Gruppe 2)

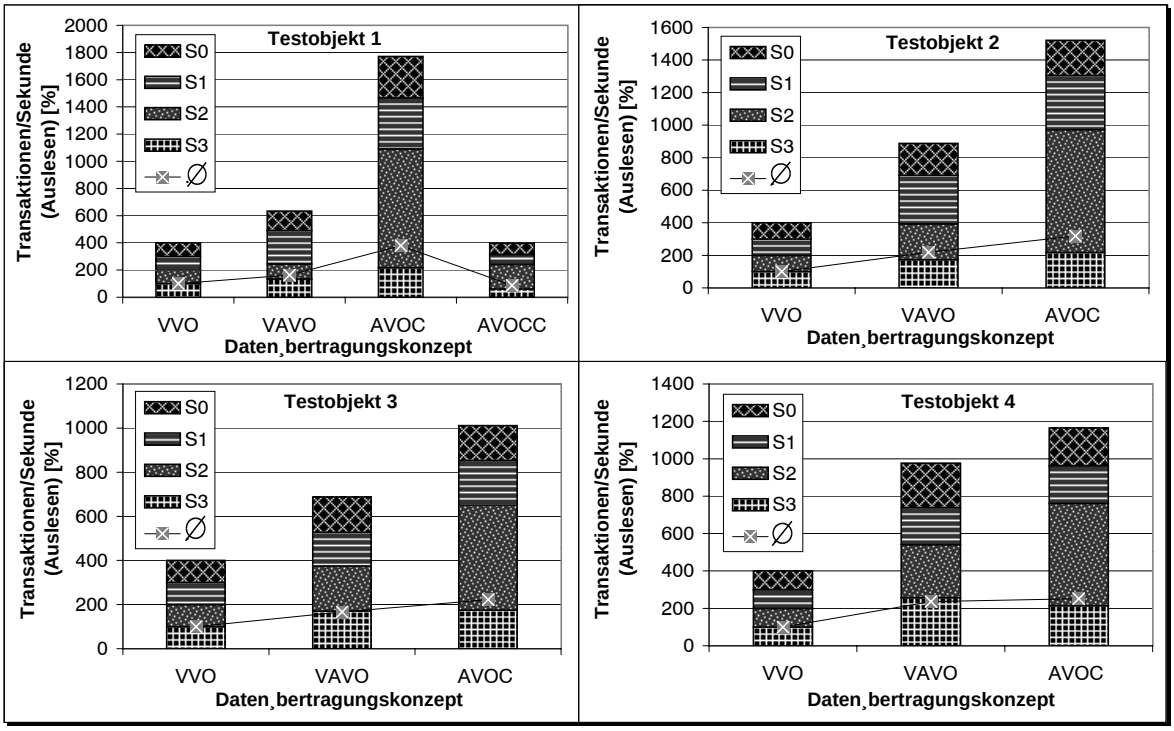


Abbildung 6.26: Transaktionsdurchsatz (Auslesen) der statischen Daten-Container (Gruppe 1)

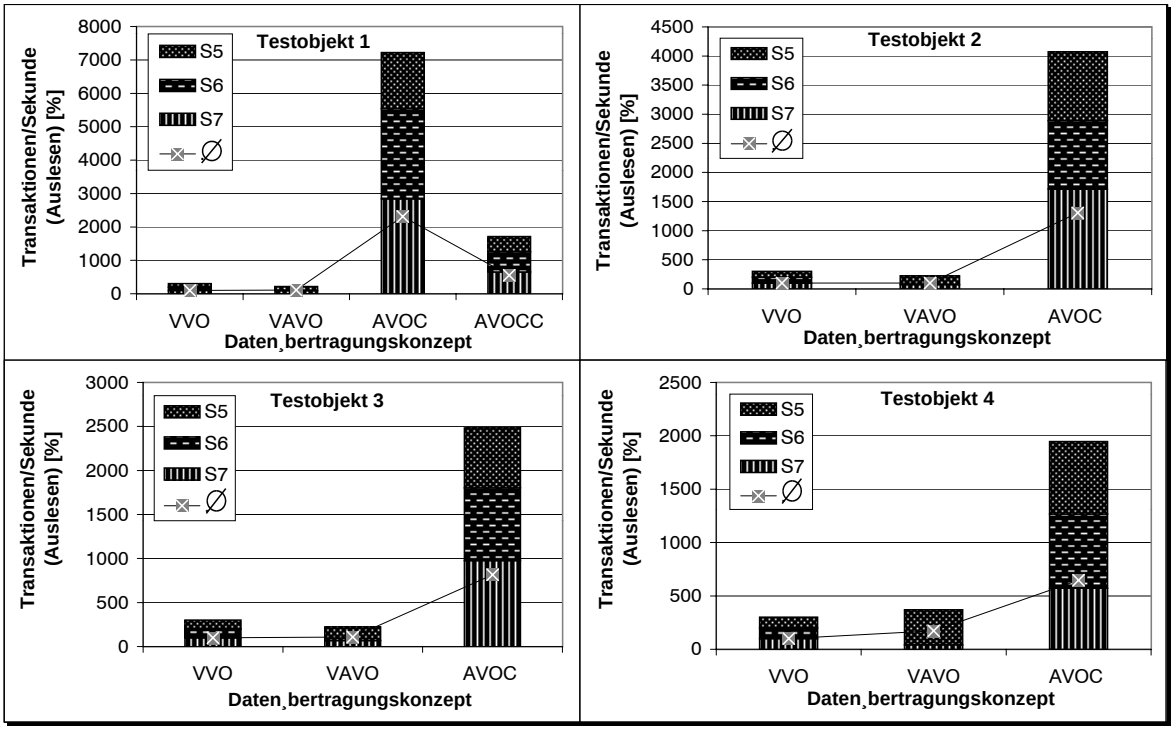


Abbildung 6.27: Transaktionsdurchsatz (Auslesen) der statischen Daten-Container (Gruppe 2)

Kapitel 7

Zusammenfassung

In dieser Arbeit wurde ein universelles Datenübertragungskonzept in Form des Aktiven Daten-Containers entwickelt. Das Ziel dieses Konzepts ist die Entwicklung eines anwendungs- und architekturunabhängigen Datenübertragungsmechanismus, der möglichst flexibel ist, sowohl was die Möglichkeit der Übertragung von Daten aus unterschiedlichen Quellen mit unterschiedlicher Struktur und in unterschiedlicher Menge als auch die Anpassung an die jeweiligen Projektanforderungen und vorhandene Systemumgebung anbelangt. Aufgrund der Tatsache, daß sich Projektanforderungen und die vorhandene Systemumgebung während der Entwicklung und in der Betriebs- und Wartungsphase ändern können, sind die universellen Daten-Container so konzipiert und in EJB-Systeme eingebettet, daß zu jedem Zeitpunkt durch zentrale Eingriffe auf diese Anforderungen reagiert werden kann, ohne Änderungen an existierenden Systemstrukturen vornehmen zu müssen oder diese minimal zu halten. Diese Flexibilität wird durch die Definition von Schnittstellen und deren zentrale Implementierung erreicht. Dabei beschränkt sich die Implementierung nicht auf eine Datenstruktur zur Aufnahme von Daten, sondern bezieht sich auch auf Funktionalität, die konfigurierbar nutzbar gemacht wird und eng mit der Datenübertragung verbunden ist. Die Implementierung kann zentral erweitert, gewartet und konfiguriert werden. Ein vorgestelltes Konfigurationskonzept bezieht insbesondere den Deployment-Deskriptor von Enterprise-JavaBeans mit ein und erlaubt damit eine feingranulare Kontrolle über die Datenübertragung dieser Komponenten. Dies stellt auch eine mögliche Lösung für Standardkomponenten dar, die in verschiedenen Anwendungsszenarien Verwendung finden sollen und damit vielfältigen Anforderungen gegenüberstehen, wie bereitgestellte Daten geliefert werden sollen. Die konfigurierbare Funktionalität, die über die Eigenschaften eines rein passiven Daten-Containers hinausgeht, ist ein weiteres allgemeines Ziel der hier vorgestellten Konzepte, nämlich das der Optimierung der Kommunikation selbst und des Entwicklungsaufwands, der bei der Nutzung des Kommunikationsmechanismus entsteht. In einem Industrieprojekt konnte dabei der Codeumfang einer komplexen Anwendung insgesamt um ca. 18% gesenkt werden. Diese Reduktion resultiert aus der Einsparung von ca. 99% Daten-Container-Klassen, ca. 83% Datenaustauschoperationen zwischen Daten-Containern und Objektstrukturen auf dem Server, sowie ca. 92% zwischen Daten-Containern und GUI-Elementen auf dem Client. Eine zusätzliche Zeitersparnis bei der Entwicklung entsteht dadurch, daß einzelne Anwendungsentwickler keine Überlegungen mehr anstellen müssen, wie Daten idealerweise zu übertragen sind. Anhand von Beispielen wurde das erhebliche Optimierungspotential der Kommunikation in EJB-Systemen dargestellt, das sich eröffnet, wenn die Form und Menge der

zu transportierenden Daten optimiert wird. Dabei konnten Reduktionen der Antwortzeit bis zu 95% und Steigerungen des Transaktionsdurchsatzes bis zu 1170% erzielt werden.

Anhang A

Entwurfsmuster

Im folgenden werden wesentliche Konzepte dieser Arbeit als Entwurfsmuster formuliert. Die Beschreibung orientiert sich dabei an der in [Dee01] verwendeten Entwurfsmusterschablone.

A.1 Aktive Daten-Container

A.1.1 Zusammenhang

Bei der Entwicklung von Client/Server-Anwendungen müssen fachlich relevante Daten zwischen Client und Server übertragen werden. Bei einer mehrschichtigen Architektur können sich die zu übertragenden Daten in unterschiedlichster Form in den verschiedenen Schichten der Anwendung befinden. Nachfolgend sind einige Beispiele genannt:

- **Präsentationsschicht:** Die Daten werden in eine Grafische Benutzerschnittstelle (GUI) eingegeben und befinden sich in den einzelnen GUI-Elementen.
- **Logikschicht:** Die Daten befinden sich in Attributen von Geschäftsobjekten, die gemäß des fachlichen Hintergrunds der Anwendung modelliert sind.
- **Datenschicht:** Die Daten befinden sich in Datenbankmanagementsystemen oder werden von Fremd- und Altsystemen (*Legacy-Systeme*) bezogen. Ein direkter Datenzugriff kann über entsprechende Systemschnittstellen erreicht werden.

Mit diesem Entwurfsmuster können universelle Daten-Container erstellt werden, die Daten unabhängig von der Form in der sie vorliegen übertragen. Dabei wird der Entwicklungsaufwand verringert, indem manuelle Datenaustauschoperationen zwischen Server-Objekten und Daten-Containern vermieden werden. Zusätzlich wird eine Optimierung der Datenmenge und -komplexität vorgenommen, die zu einem signifikant besseren Leistungsverhalten der Anwendung im Sinne der Kommunikation führt.

A.1.2 Probleme

Die Probleme beziehen sich auf *Implementierungsaspekte*, die sich mit dem Aufwand zur Verwendung eines Datenübertragungskonzepts befassen und *Leistungsaspekte*, die sich mit Leistungsverhalten eines Datenübertragungskonzepts befassen.

Implementierungsaspekte

Zur Datenübertragung muß sich jeder Entwickler um die *Beschaffung* der Daten aus der jeweiligen Quelle kümmern, das *Datenformat* oder die Schnittstelle, um auf die Daten zuzugreifen verstehen, die *Entnahme* der benötigten Daten durchführen und ein *Übertragungsformat* in Form eines Daten-Containers festlegen und mit den Daten belegen. Durch diese Anforderungen ergeben sich eine Reihe von Problemen:

- **Durchgängigkeitsproblematik.** Der gewählte Übertragungsmechanismus wird in Abhängigkeit des Problemfelds und den Kenntnissen des Entwicklers gewählt. Dies führt in einem großen System zu vielen unterschiedlichen Lösungsansätzen. Es kann keine allgemeine Optimierung erfolgen und Änderungen müssen an unterschiedlichen Stellen auf unterschiedliche Weise durchgeführt werden.

- **Kopplungsproblematik.** Die direkte Serialisierung von Geschäftsobjekten verletzt das Entwurfsziel mehrschichtiger Architekturen, Geschäftsobjekte, Geschäftsprozesse und Algorithmen in einer Schicht zur besseren Wiederverwendung, Änderung und Erweiterung zu kapseln. Bei der Verwendung von *Value Objects* wird eine zusätzliche Schicht von Objekten eingeführt, durch die ebenfalls eine Kopplung zwischen Client und Server entsteht.

- **Kostenproblematik.** Ein hoher Entwicklungsaufwand besteht darin, Daten aus den vorliegenden Datenstrukturen zu entnehmen und in ein Übertragungsformat zu überführen, oder umgekehrt aus einem Übertragungsformat in eine vorliegende Datenstruktur einzufügen. Zusätzlich müssen die vorliegenden Datenstrukturen häufig mit Schnittstellen versehen werden, die Datenaustauschoperationen erleichtern. Entsprechend dem Muster *Value Objects* muß für jedes Geschäftsobjekt eine weitere Datenübertragungsklasse implementiert werden. Häufig werden weitere Transportobjekte entsprechend den fachlichen Anwendungsfällen benötigt, um die Anzahl der zu übertragenden Attribute einzuschränken.

- **Flexibilitätsproblematik.** Ändert ein Client seine Anforderungen dahingehend, daß er andere Daten oder eine größere Datenmenge anfordern will, muß eine Änderung der Transportobjekte erfolgen. Kommt ein neuer Client mit spezifischen Anforderungen hinzu, muß mindestens ein neues Transportobjekt implementiert werden. Die Schnittstelle der Server-Komponente muß gemäß der neu hinzukommenden Transportobjekte um eine analoge Anzahl von Methoden erweitert werden. Stellt sich nachträglich heraus, daß eine Beeinflussung der Datenübertragung erfolgen muß, z.B. durch spezielle Kompressionsalgorithmen, muß dies nachträglich in alle Transportobjekte eingebaut werden.

- **Zeitproblematik.** Falls sich Transportobjekte ändern oder neu hinzukommen, weil neue Clients mit anderen Anforderungen existieren bzw. vorhandene Clients ihre Anforderungen ändern, muß die Applikation umfangreich geändert werden und dadurch neu übersetzt und auf dem Server installiert werden. Dabei ändern sich auch die Schnittstellen von Komponenten, was einen kompletten Generierungslauf zur Folge hat.

- **Ressourcenproblematik.** Falls ein komplexer Client realisiert werden muß, der in Form eines Applets auf dem Client gestartet wird, kann die Anzahl der vorhandenen Transportklassen problematisch sein, da diese alle zum Client hin übertragen werden müssen. Das selbe Problem existiert, wenn der Platzbedarf für den Java-Client möglichst gering sein soll und dennoch sehr viele Transportklassen verwendet werden sollen.

- **Sicherheitsproblematik.** Bei der kompletten Übertragung von Geschäftsobjektzuständen oder *Value Objects*, die den kompletten Zustand eines Geschäftsobjekts aufnehmen, kann der unerwünschte Effekt auftreten, daß sicherheitsbezogene Daten mit übertragen werden. Um dies zu verhindern, müßten umfangreiche Vorkehrungen getroffen werden, um sensible Daten vor dem Versenden zum Client auszublenden. Die Auslieferung der Geschäftsklassen selbst kann aus Gründen des Know-how-Schutzes unerwünscht sein, da übersetzte Java-Klassen dekompiliert werden können.

- **Eindeutigkeitsproblematik.** Die Java-Collection-Klassen sind nicht explizit für die Datenübertragung vorgesehen. Dies erschwert das Auffinden von Programmteilen, die sich mit der Datenübertragung befassen. Das selbe Problem existiert bei der direkten Übertragung von Geschäftsobjekten und der Übertragung von *Value Objects*, falls diese keiner Namenskonvention folgen.

- **Benutzbarkeitsproblematik.** Die Collection-Klassen besitzen nicht die notwendige Funktionalität, um die Anforderungen einer dynamischen Datenübertragung vollständig zu erfüllen. Dies führt zu einem höheren Entwicklungsaufwand, da die Funktionalität nachgebildet werden muß.

Leistungsaspekte

Bei der Implementierung von Daten-Containern müssen die folgenden leistungsbeeinflussenden Faktoren berücksichtigt werden:

- **Datenmenge:** Die Datenmenge ist ein grundlegender Einflußfaktor. Die Übertragungszeiten nehmen proportional zur Steigerung der Datenmenge zu. Applikations-Server können jedoch Anomalien aufweisen, die sich in hohen Übertragungszeiten bei bestimmten Datenmengen oder massiven Einbrüchen bei hohen Datenmengen niederschlagen. Darüber hinaus kann die Übertragung hoher Datenmengen mit einem Fehler abgebrochen werden.

- **Datenkomplexität:** In komplexen Anwendungsdomänen entstehen leicht ineinander geschachtelte Datenstrukturen, die sich in einer massiven Steigerung von Antwortzeiten niederschlagen können. Zusätzlich sind die Kommunikationsschichten von Applikations-Servern unterschiedlich realisiert und reagieren zum Teil sehr unterschiedlich auf komplexe Datenstrukturen.

- **Protokolleinfluß:** Das von Applikations-Servern implementierte Protokoll wirkt sich ebenfalls auf die Dauer von entfernten Aufrufen aus. Proprietäre, optimierte Protokolle bieten häufig bessere Antwortzeiten.

- **Lokale Kommunikation:** Die Datenübertragung ist auch im Rahmen einer lokalen, serverinternen Kommunikation relevant, da Applikationen aus vielen EJB-Komponenten zusammengesetzt werden können, die sich innerhalb des selben Servers befinden und miteinander kommunizieren. Es existieren Server, die keine Optimierung der internen Kommunikation vornehmen. Dies kann sich ebenfalls negativ auf das Leistungsverhalten auswirken.

A.1.3 Gründe

Das Konzept des ADCs verhindert Probleme, die allgemein bei der Entwicklung von Datenübertragungsmechanismen auftreten dadurch, daß ein fertiger, universeller Mechanismus zentral bereitgestellt wird, der nachträglich transparent konfiguriert und geändert werden kann. Das Konzept besitzt die folgenden Eigenschaften:

- **Durchgängigkeit.** Die Daten-Container werden konsequent in allen Schichten der Anwendung mit einer standardisierten Schnittstelle eingesetzt. Außerdem ist ein Benutzungskonzept mit den Containern verbunden.

- **Kostenfaktor.** Die Container-Funktionalität wird zentral in einer Klasse bereitgestellt. Wartung und Erweiterungen des Daten-Containers erfolgen zentral und transparent für die Anwendungsentwickler. Datenaustauschoperationen zwischen Datenquelle und Daten-Container erfolgen automatisch. Die feste Schnittstelle erlaubt die Implementierung weiterer universeller Komponenten, wie z.B. Mechanismen, die einen automatischen Datenaustausch zwischen Daten-Container und GUI-Elementen übernehmen.

- **Leistungsfaktor.** Die Attributmenge kann zur Übertragung auf die vom Anwendungsfall abhängige Menge begrenzt werden. Dies entspricht einer Umsetzung des Entwurfsmusters *Dynamische Attribute* [Mow97], das automatisch von jedem Entwickler verwendet wird.

  Der ADC fördert die strukturierte Übertragung von Daten aus mehreren Datenquellen, wie z.B. Datenzugriffsobjekte und Geschäftsobjekte. Dies verhindert automatisch die Problematik, daß zu viele entfernte Methodenaufrufe getätigt werden, um verschiedenartige Daten zu transportieren.

  Es kann Einfluß auf die Datenmenge und -komplexität genommen werden, die über das Netzwerk transportiert werden muß, indem effiziente Speicherformen oder Kompressionsalgorithmen angewendet werden.

- **Kopplungsproblematik.** Das Objektmodell der Applikationsschicht wird vor dem Client verborgen. Vorhandenes Know-how wird geschützt. Zusätzlich wird keine weitere Abhängigkeit zu einer Reihe von *Value Objects* geschaffen.

- **Flexibilitätsproblematik.** Durch dynamische Datenübertragung kann jederzeit die Übertragung zusätzlicher Daten erfolgen. Die Daten werden unabhängig von der Form, in

der sie vorliegen, im Daten-Container verpackt. Zusatzfunktionalität kann im Daten-Container zur Ausführung gebracht werden und auch systemweit transparent für die Anwendungsentwickler aktiviert werden. Dies ermöglicht eine ständige Anpassung des Containers.

- **Zeitfaktor.** Die Verwendung von ADCs als Parameter in der Remote-Schnittstelle von Enterprise JavaBeans trägt zur Stabilität dieser bei. Müssen Daten zusätzlich übertragen werden oder muß der Typ von übertragenen Daten geändert werden, erfordert dies keine Änderung der Remote-Schnittstelle.

- **Ressourcenfaktor.** Im Minimalfall ist nur eine Transportklasse notwendig, die zum Client übertragen werden muß.

- **Sicherheitsproblematik.** Das Konzept schränkt die zu transportierenden Attribute auf die wirklich benötigten ein.

- **Benutzbarkeit.** Die Container-Klasse ist auf die Aufgabe der Datenübertragung spezialisiert und deckt alle Belange der Datenübertragung ab.

Aufgrund der Eigenschaften des Konzepts, bietet es auch eine mögliche Lösung für Standardkomponenten, die nachträglich an die unterschiedlichsten Anwendungsanforderungen im Sinne der Datenübertragung angepaßt werden können, obwohl kein Quellcode für die Komponente vorliegt.

A.1.4 Lösung

Struktur

Der ADC führt die Aufgaben der Datenübertragung in einer Klasse bzw. in einer Schnittstelle zusammen und bietet zusätzlich noch die Möglichkeit, weitere Operationen auf den transportierten Daten auszuführen. Eine signifikante Entlastung des Entwicklers erfolgt dadurch, daß dem ADC lediglich die zu übertragenden Daten unabhängig von der vorliegenden Form übergeben werden müssen. Der Container kann die Daten sämtlicher Objekte aufnehmen, deren Struktur ihm bekannt sind.
In Abbildung A.1 ist die Grundstruktur des ADCs dargestellt. Im Zentrum des Konzepts steht die Schnittstelle `ActiveDataContainer`, die alle Methoden definiert, die ein Transportobjekt bereitstellen muß. Dabei sind die folgenden Arten von Methoden relevant:

- **Schreiben, Lesen, Löschen und Existenzprüfung von Attributen.**

- **Iteratoren für Attribute und Attributwerte**.

- **Hinzufügen, Entnehmen und Synchronisieren von Objektzuständen.**

- **Konfiguration und Ausführung von Funktionalität.**

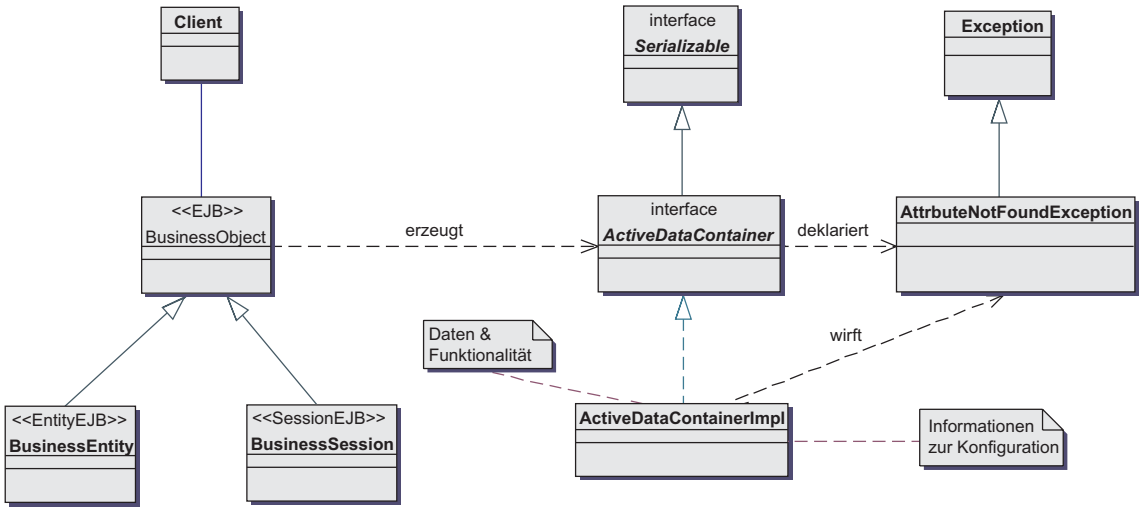


Abbildung A.1: Klassendiagramm: Prinzip des Aktiven Daten-Containers

Die Schnittstelle ist von `Serializable` abgeleitet, um implementierende Objekte zwischen Client und Server übertragen zu können. Eine mögliche Alternative hierzu ist die Schnittstelle `Externalizable`, die eine weitergehende Beeinflussung der Kommunikation erlaubt. Die Exception `AttributeNotFoundException` wird erzeugt, wenn ein angefordertes Attribut nicht im Daten-Container enthalten ist, um damit diese mögliche Fehlerquelle aufzudecken. Die Klasse `ActiveContainerImpl` stellt die Implementierung des Daten-Containers bereit. Dazu gehört eine geeignete und ggf. optimierte Datenspeicherung und Zusatzfunktionalität, die auf die Daten im Container angewendet werden kann, oder das Verhalten des Containers selbst bestimmt. Um die Universalität und Flexibilität des Containers sicherzustellen, sind für das Konzept die *Informationen zur Konfiguration* wichtig, die zur Übersetzungs- oder zur Laufzeit bestimmen, welche Funktionalität verwendet wird und wie sich das Verhalten des Containers darstellt. Diese Informationen sind entweder im Programmcode festgelegt, werden als Parameter beim Start des Java-Interpreters übergeben, im Deployment-Deskriptor einer EJB abgelegt oder zentral in einem JNDI-Objekt gehalten. Es empfiehlt sich die Container global in EJBs zu erzeugen, um Konfigurationsinformationen zu übergeben und die Container für wiederholte Kommunikationsvorgänge wiederzuverwenden.

Abbildung A.2 zeigt die grundsätzlichen Interaktionen der beteiligten Objekte und Komponenten. Wichtig dabei ist die unterschiedliche Verwendung des Containers auf Client- und Server-Seite. Während der Client hauptsächlich die `get-` und `set-`Methoden zum Lesen und Schreiben von Daten verwendet (2, 3), benutzt eine EJB vorrangig Methoden zur Übergabe oder Entnahme bzw. Synchronisation eines Objekts, dessen Daten zwischen Client und Server transportiert werden. Falls notwendig, kann der unterschiedlichen Verwendung auf Client und Server z.B. dadurch Rechnung getragen werden, daß zwei getrennte Schnittstellen für Client und Server bereitgestellt werden und vor oder während des Versands eine Typumwandlung in die Client-Schnittstelle erfolgt. Die Anwendung von Funktionalität kann dabei grundsätzlich bei jeder Interaktion mit dem Daten-Container erfolgen (1.1.1, 1.2.1, 2.1, 3.1 und 4.1.1). Dabei

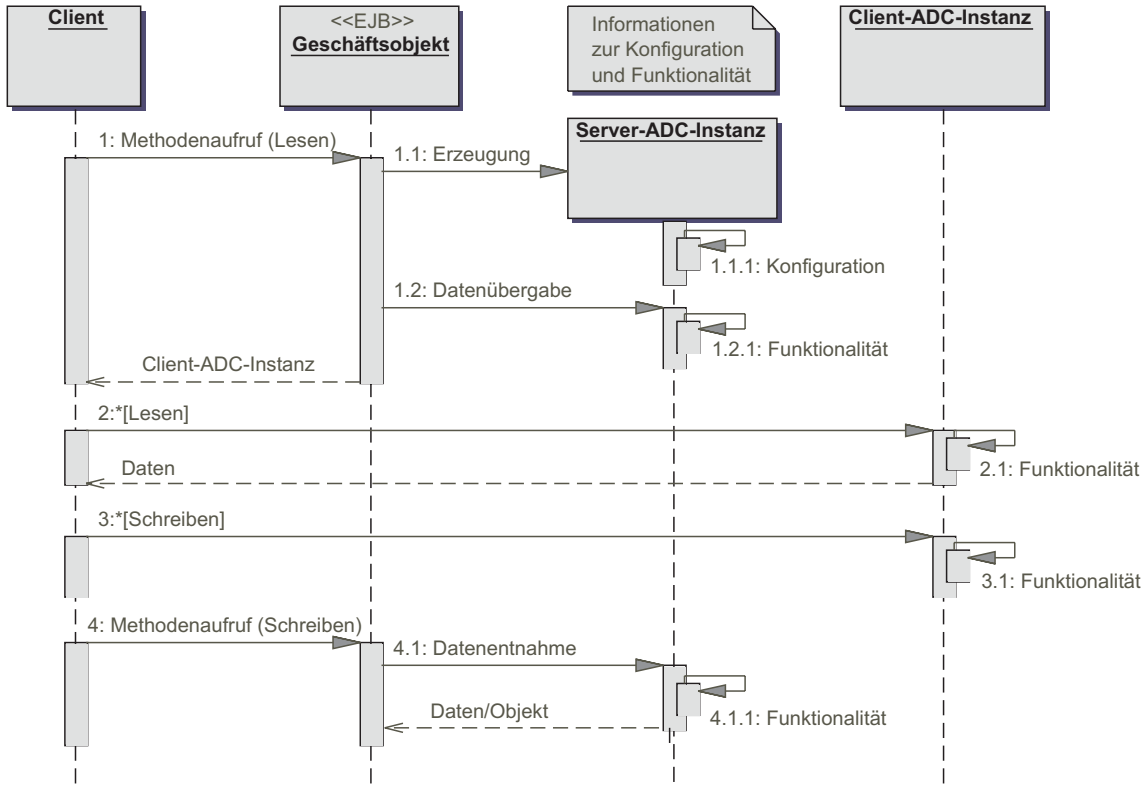


Abbildung A.2: Sequenzdiagramm: Prinzip des Aktiven Daten-Containers

wird Funktionalität angewendet, die aufgrund der Datenübertragung notwendig ist. Es kann sich dabei z.B. um das Komprimieren oder Dekomprimieren von Daten handeln. Eine grundsätzliche Anwendung von Funktionalität kann auch global im Rahmen der Serialisierung/Deserialisierung stattfinden.

Vorgehensweisen

Der grundlegende Aufbau einer ADC-Implementierung ist in Abbildung A.3 schematisch dargestellt.
In der Praxis kann es sinnvoll sein, die Übertragung der Daten eines Objekts von der Datenübertragung vieler Objekte (Massendaten) zu unterscheiden. Dabei können zwei Typen von ADCs implementiert werden. Ein ADC-Typ transportiert dabei immer nur ausschließlich die Daten *eines* Objekts und wird damit Stellvertreter (*Proxy*) [Gam95] für das betreffende Objekt auf dem Applikations-Server. Handelt es sich bei dem Objekt um eine EJB, so kann der ADC durch den automatisierten Transport der EJB-Referenz auch als intelligenter Stellvertreter (*Smart Proxy*) fungieren. Ein zweiter ADC-Typ wird auf die Datenübertragung vieler Objekte ausgelegt. Dies hat den Vorteil, daß anhand des Programmcodes leichter unterschieden werden kann, ob die Daten eines Objekts oder die Daten vieler Objekte transportiert werden.
Nachfolgend werden die einzelnen Elemente erläutert:

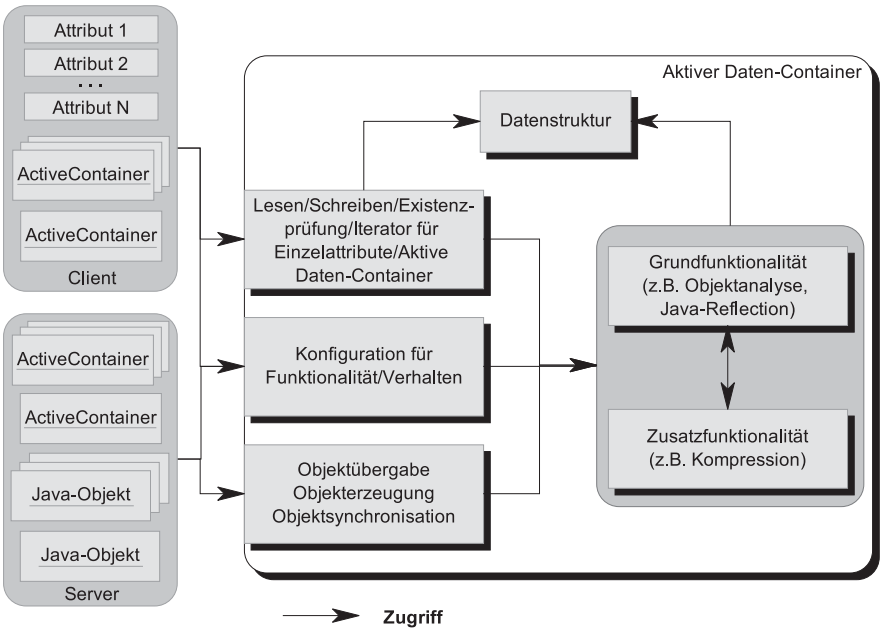


Abbildung A.3: Schema des Aktiven Daten-Containers

- **Objektübergabe, -erzeugung und -synchronisation:** Es handelt sich dabei um Methoden und Konstruktoren, die als Parameter Java-Objekte besitzen. Die Objektübergabe dient dem Hinzufügen von Objekten als Datenlieferant, d.h. die Daten der Objekte werden in der internen Datenstruktur gespeichert. Durch Angabe einer Liste von gewünschten Attributnamen kann nur auf benötigte Attribute zugegriffen werden, um eine Reduktion der Datenmenge zu erreichen (*Dynamische Attribute* [Mow97]).

  Die Objekterzeugung erstellt neue Objekte, deren Attributwerte mit den im Daten-Container transportierten Attributwerten synchronisiert werden.

  Die Objektsynchronisation gleicht im Container transportierte Daten mit den Attributen der übergebenen Objekte ab, d.h. evtl. im Client vorgenommene Änderungen werden auf dem Server übernommen.

  Die Methoden können als **streng typisierte Methoden** für jeden akzeptierten Objekttyp Methoden bereitstellen, oder als **schwach typisierte Methoden** nur jeweils eine generische Methode mit Übergabe bzw. Rückgabeparametern vom Typ `Object` besitzen. Zur Übergabe von Objektfolgen kann der Typ `Collection` verwendet werden. Der Implementierungsaufwand ist bei strenger Typisierung um so höher, je mehr Objekttypen übergeben werden können. Zusätzlich ist die Flexibilität des Containers eingeschränkt, da zur Laufzeit bereits feststeht, was übertragen werden kann.

- **Konfiguration für Funktionalität/Verhalten:** Zur Anpassung des Daten-Containers an verschiedene Systemanforderungen, kann die Konfiguration dynamisch zur Laufzeit oder statisch im Quellcode angegeben werden. Beispiele für Zusatzfunktionalitäten sind die Kompression von Daten und das transparente „Durchschreiben“ von Attributen auf den

Server bei clientseitigem Schreibzugriff (Smart-Proxy).

- **Lesen/Schreiben/Existenzprüfung/Iteratoren für Einzelattribute/ADCs:** Mittels `get`- und `set`-Methoden kann auf die Einzelattribute zugegriffen werden. Attributnamen werden dabei als Zeichenkette übergeben. Aufgrund der dynamischen Datenspeicherung kann das Lesen und Schreiben mittels zweier generischer `get`- und `set`-Methoden, die als Rückgabe- bzw. Übergabeparameter Objekte vom Typ `Object` besitzen, erfolgen. Alternativ können für alle möglichen Datentypen `getAsXXX`-Methoden implementiert werden, die eine Konvertierung der Daten vornehmen (z.B. `getAsString()`). Dies führt zu einer Entkopplung zwischen Server und Client hinsichtlich der verwendeten Datentypen. Das Durchlaufen aller Attribute und deren Werte durch die Implementierung der Schnittstelle `java.util.Iterator` auf der ADC-internen Datenstruktur ermöglicht die Programmierung allgemeiner, vom Inhalt des Daten-Containers unabhängiger Routinen. ADC-Objekte können als Stellvertreter für Server-Objekte verwendet, manipuliert oder neu erzeugt werden und in den Daten-Container geschrieben werden. ADC-Objekte können entnommen werden und neue Geschäftsobjekte aus den im Container transportierten Attributen erzeugt werden oder vorhandene mit den Attributen synchronisiert werden. Bei der Übertragung von Objektfolgen kann auch auf die Repräsentation jedes einzelnen Objekts als ADC (der in einem ADC als Sammelbehälter geschachtelt ist) verzichtet werden. Statt dessen können die Attribute jedes einzelnen Objektzustands an der Schnittstelle z.B. als `HashMap` repräsentiert werden.

- **Datenstruktur:** Die Datenstruktur innerhalb des Daten-Containers, die zur Speicherung des Zustands von übergebenen Objekten bzw. übergebenen Transportdaten herangezogen wird, richtet sich nach der Beschaffenheit der Daten selbst, aber auch nach den Eigenschaften der Kommunikationsschicht des verwendeten Applikations-Servers. Zusätzlich muß eine Optimierung auf Platzbedarf und Zugriffsgeschwindigkeit erfolgen. Als Basisspeicherstruktur bietet sich die Klasse `HashMap` an. Die Attributnamen der übergebenen Objekte werden dabei als Schlüssel verwendet und die Attributausprägungen als Werte. Gegenüber der vergleichbaren Datenstruktur `Hashtable` hat `HashMap` den Vorteil, daß die Attributausprägung `null` direkt gespeichert werden kann. Zusätzlich ist ein Vorteil im Leistungsverhalten vorhanden, da die Methoden der Klasse `HashMap` nicht synchronisiert sind. Die Synchronisation beeinflußt das Leistungsverhalten eines Java-Programms negativ [Sin97]. Falls die Synchronisation benötigt wird, kann eine synchronisierte Form von `HashMap` verwendet werden.

  Nachfolgend sind drei unterschiedliche Formen zur Speicherung der zu transportierenden Daten aufgeführt:

  1. Die flexibelste Implementierung des Konzepts ist in Abbildung A.4 dargestellt und besteht darin, intern ADC-Objekte zu speichern. Als Schlüssel wird die vom Anwendungsentwickler vergebene Kennung verwendet. Selbst bei direkter Übergabe von Objekten, deren Attribute transportiert werden sollen, werden diese jeweils in einem ADC-Objekt abgelegt und in dieser Form gespeichert. Weil zur Übersetzungszeit die Anzahl der übergebenen Objekte, deren Daten transportiert werden

sollen, nicht feststeht, wird zur Speicherung ein Objekt vom Typ `Vector` verwendet, das dynamisch wachsen kann. Diese Form des Daten-Containers erfordert einen geringen Implementierungsaufwand und bietet dabei eine sehr hohe Flexibilität, da die ADC-Objekte unterschiedlich konfiguriert und ggf. auch unterschiedlich implementiert sein können. Diese Flexibilität führt jedoch aufgrund der ineinandergeschachtelten Objekte und der redundanten Speicherung von Attributen zu einer hohen Datenmenge und einer hohen Datenkomplexität.

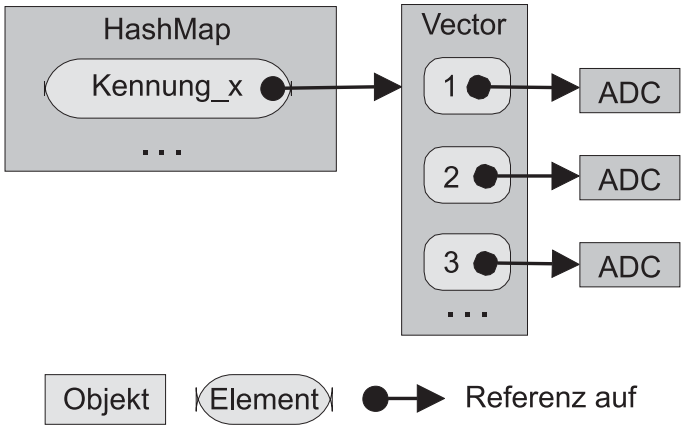


Abbildung A.4: Aus ADC-Objekten bestehende Datenstruktur

2. Eine Reduktion der Datenmenge und der Komplexität des Übertragungsformats kann erreicht werden, indem die Speicherung der Daten als ADC-Objekte aufgehoben wird und im wesentlichen nur noch die tatsächlichen Nutzdaten möglichst effizient gespeichert werden. Zur späteren Rekonstruktion des eigentlichen Objekts wird dessen Typbezeichnung nur einmal abgespeichert. Die Redundanz der Attributnamen wird verhindert, indem sie für einen Objekttyp nur noch einmal in einem String-Array abgespeichert werden. Die Attributwerte selbst werden in einem `Object`-Array an der zugehörigen Indexposition ihres Attributnamens gespeichert, um sie später wieder zuordnen zu können. Falls Daten durch den Anwendungsentwickler abgerufen werden, wird aus diesen Daten ein ADC-Objekt (oder ein beliebiges anderes Objekt) erzeugt und gefüllt. Dieses Konzept ist in Abbildung A.5 dargestellt.
3. Eine weitere Reduktion der Datenmenge und Komplexität kann durch die in Abbildung A.6 dargestellte Verwendung einer reinen String-Repräsentation erreicht werden. Daten werden anhand der Indexposition des Attributnamens und des Datentyps rekonstruiert. Diese Form schränkt die Flexibilität des Containers allerdings ein, da nicht mehr automatisch beliebige Objekttypen unterstützt werden. Für jeden Objekttyp eines Attributs, der im Container gespeichert werden soll, muß eine String-Repräsentation möglich sein und eine Routine erstellt werden, die das Objekt aus dem erstellten String rekonstruiert. Bei primitiven Java-Datentypen sind diese Funktionen aber bereits vorhanden. Diese Vorgehensweise hat zwar einen höheren Implementierungsaufwand zur Folge, kann allerdings die Datenmenge reduzieren. Das ist

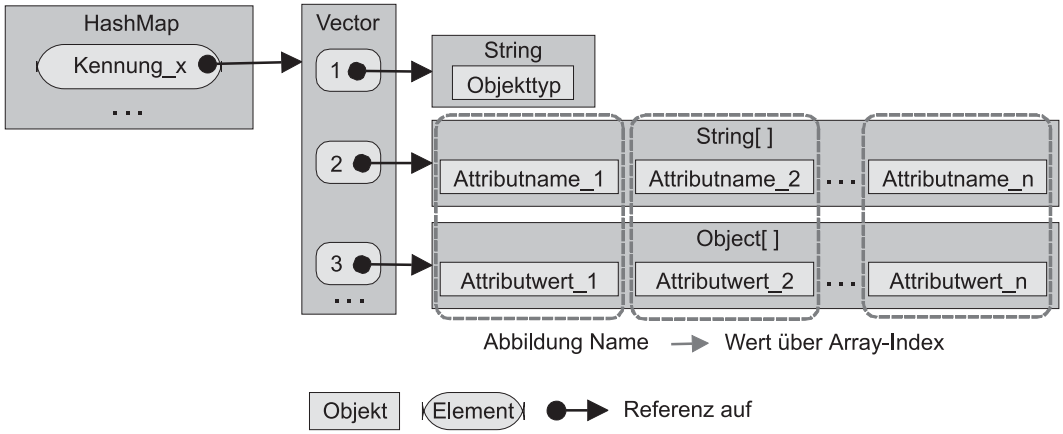


Abbildung A.5: Auf Arrays basierende Datenstruktur

vor allem dann der Fall, wenn die String-Repräsentation eines Datentyps kleiner ist, als der Datentyp selbst. Eine ähnliche Strategie zur Verkleinerung der Datenmenge besteht darin, die übertragenen Daten so zu kodieren, daß sie nur die wirklich benötigte Datenmenge in Anspruch nehmen. Ein Beispiel hierfür ist die Verwendung des Datentyps `int`, der in Java mit 4 Bytes repräsentiert wird. Unabhängig vom tatsächlichen Wert der Zahl, die als Integer repräsentiert wird, müssen diese 4 Bytes übertragen werden. Häufig werden jedoch kleinere Zahlen transportiert, die eigentlich nicht die volle Bitbreite benötigen. Eine Strategie im Daten-Container kann daher darin bestehen, Zahlen nur mit der tatsächlich benötigten Bitbreite zu repräsentieren und dadurch die Datenmenge zu reduzieren.

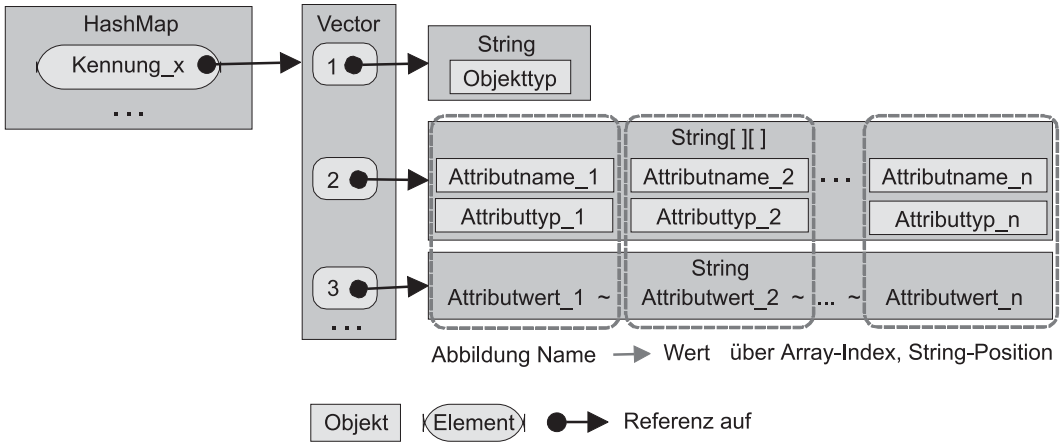


Abbildung A.6: Auf Strings basierende Datenstruktur

Die hier vorgestellten Maßnahmen zur Verringerung der Datenmenge und -komplexität sind als Beispiele zur Illustration des vorgestellten Datenübertragungskonzepts zu verstehen. Um ein optimales Ergebnis zu erzielen, müssen in diese Überlegungen immer die Beschaffenheit der zu übertragenden Daten innerhalb einer Anwendung,

das Verhalten der Kommunikationsschicht des verwendeten Applikations-Servers und die grundlegenden Anforderungen an die Eigenschaften des Übertragungskonzepts berücksichtigt werden. So kann z.B. die Komplexität der Daten nochmals reduziert werden, falls auf die Möglichkeit der Speicherung mehrerer unterschiedlicher Objekttypen unter einer frei vergebenen Kennung verzichtet wird. Dies stellt eine geringere Anforderung an das Übertragungskonzept dar und führt zur Eliminierung des `HashMap`-Objekts.

Die Optimierung der Datenstruktur kann zu einem höheren Zeitaufwand beim Beschreiben bzw. Auslesen des Daten-Containers führen. Es ist darauf zu achten, daß dieser Zeitaufwand in der vorhandenen Systemumgebung nicht die erzielte Einsparung bei der Übertragungszeit des Containers aufhebt. Aus diesem Grund müssen häufig Datenaustauschoperationen ebenfalls optimiert werden.

- **Grundfunktionalität:** Schwerpunkt der Grundfunktionalität ist der automatische Datenaustausch mit den Objekten, deren Daten im Container transportiert werden sollen. Daraus resultiert eine Entlastung des Anwendungsentwicklers, der diesen Vorgang nicht manuell durchführen muß. Beispiele für die Implementierung dieser Funktionalität sind:

  – **Verwendung von Java-Reflection:** Um möglichst viele Objekte abzudecken, erfolgt ein generischer Zugriff auf Attribute und Methoden mittels Java-Reflection. Die Anwendung von Reflection erfordert einen höheren Zeitbedarf als der herkömmliche Aufruf von fest kodierten Zugriffsmethoden. Darüber hinaus muß eine Konvention darüber erfolgen, wie die Benennung der ausgelesenen Attribute erfolgt. Die Attributnamen können übernommen werden oder mittels einer Zuordnungsfunktion in andere Namen überführt werden.

  – **Verwendung eines Generators:** Objekte deren Daten im Daten-Container gespeichert werden sollen, müssen das Interface `DataObject` implementieren, dessen Methoden generiert werden und den Datenaustausch mit dem Objekt erlauben. Der Daten-Container greift auf diese Schnittstelle zu und tauscht darüber Attribute mit seiner internen Datenstruktur aus. Für optimale Geschwindigkeit kann vom Generator eine effiziente Datenstruktur zur Übersetzungszeit generiert werden. Eine Generatorimplementierung ist vergleichsweise einfach, wenn hierfür die Attribute eines Objekts mit Reflection-Mechanismen analysiert werden, um dafür passende Codezeilen zu erzeugen.

  Mit Hilfe des ADCs können auch Objektbäume[1] berücksichtigt werden. Dies ist z.B. bei Geschäftsobjekten möglich, die zueinander in Beziehung stehen (z.B. Kunden-Objekt hat Adress-Objekt). Hierzu ist bei der Entnahme von Attributen aus einem Objekt ein rekursiver Algorithmus zu implementieren, der jedes erhaltene Attribut erneut analysiert, falls es sich nicht um einen primitiven Datentyp handelt. Der Algorithmus kann so erweitert werden, daß auch Geschäftsobjekte berücksichtigt werden, die als EJBs realisiert sind. Dabei muß vorausgesetzt werden, daß die EJBs eine standardisierte Methode zum Abruf ihrer Daten mittels eines ADCs besitzen. Innerhalb des Algorithmus wird dann

[1] Falls Algorithmen zur Erkennung von Zyklen verwendet werden, können auch Graphen übertragen werden.

zusätzlich geprüft, ob ein erhaltenes Attribut eine EJB-Referenz ist und ggf. die standardisierte Methode aufgerufen, um die Daten dieser EJB zu beschaffen. Im schreibenden Fall kann der Algorithmus analog angewendet werden. Insgesamt entsteht ein ähnliches Verhalten, wie es beim Java-Serialisierungskonzept der Fall ist. Durch die Implementierung des Entwurfsmusters *Dynamische Attribute* wird jedoch erreicht, daß nur bestimmte Attribute aus dem Objektbaum entnommen werden und die zu übertragende Datenmenge eingeschränkt wird.

Als Beispiel für die Entnahme von Daten aus Objekten, die nicht mit Java-Reflection analysiert werden können, sollen an dieser Stelle `ResultSet`-Objekte, die das Ergebnis einer JDBC-Datenbankanfrage sind, dienen. Um die Container-Funktionalität unabhängig vom Inhalt des `ResultSet`-Objekts zu implementieren, kann die in Abbildung A.7 dargestellte Vorgehensweise gewählt werden.

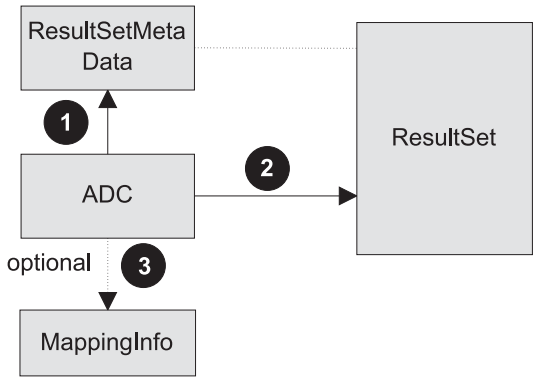


Abbildung A.7: Automatischer Zugriff auf ein beliebiges ResultSet-Objekt

Der ADC analysiert den Inhalt des Objekts `ResultSetMetaData`, das vom `ResultSet`-Objekt angefordert werden kann und Auskunft über seine Struktur gibt (1). Aufgrund der Analyse liest der ADC die Daten aus dem `ResultSet`-Objekt und legt sie in seiner internen Datenstruktur ab (2). Das `ResultSet`-Objekt enthält die Attributnamen der zugrundeliegenden Datenbanktabellen. Falls diese Namen unerwünscht sind, kann durch die optionale Auswertung von Zuordnungsinformationen erreicht werden, daß eine andere Namensgebung verwendet wird. Zwischen dem Auslesen von Daten aus dem Datenobjekt und dem Einfügen in den ADC kann ein Objekt `MappingInfo` verwendet werden, daß bei einer Übergabe eines Attributnamens der Relation den zugehörigen Attributnamen, der in der Anwendung verwendet werden soll, zurückgibt (3). Die Zuordnungsinformationen können hierzu z.B. tabellarisch in einer Datei oder in der Datenbanktabelle abgelegt und gepflegt werden. Das Zuordnungsobjekt kann z.B. beim Systemstart intern eine Hash-Tabelle aufbauen, um die Informationen bereitzustellen. Das Objekt kann z.B. zentral im JNDI bereitgestellt oder auch selbst als Session-Bean implementiert werden. Weitere Informationen, wie z.B. eine notwendige Datentypkonvertierung, die beim Zuordnungsvorgang berücksichtigt werden müssen, können mit diesem Konzept ebenfalls abgedeckt werden.

- **Zusatzfunktionalität:** Funktionalität kann auf den **gesamten Daten-Container-Inhalt** angewendet werden oder auf **Einzelattribute.** Dadurch kann die Verwendung von Funktionalität sehr fein abgestimmt und optimiert werden. Die Konfiguration kann zur **Übersetzungszeit** oder zur **Laufzeit** erfolgen. Dabei kann die Funktionalität **manuell** durch den Anwendungsentwickler ausgelöst werden oder nur spezifiziert werden, um später im Container ausgelöst zu werden (**halbautomatisch**). Die **vollautomatische Anwendung** erlaubt keinen Einfluß auf ausgeführte Funktionalität und wird einmalig zentral konfiguriert. Zur Anwendung der Funktionalität werden im Container die Methoden der Schnittstelle `Serializable` überschrieben bzw. die `Externalizable`-Schnittstelle implementiert, um die Serialisierung/Deserialisierung zu beeinflussen. Die konfigurierte Funktionalität wird dabei zum Zeitpunkt des Versendens/Empfangens auf die im Container gespeicherten Daten angewendet. Ein alternativer Implementierungsansatz besteht darin, die Funktionalität unmittelbar nach der Übergabe eines Objekts, dessen Attribute transportiert werden sollen, auszuführen. Eine evtl. erforderliche Wiederaufbereitung auf dem Client kann z.B. beim ersten Zugriff mittels einer `get`-Methode erfolgen. Falls der Container komplett manuell mit Daten befüllt wird, kann es erforderlich sein, daß bei jeder Hinzufügeoperation (`set()`) und bei jeder Entnahmeoperation (`get()`) Funktionalität ausgeführt wird. Die unterschiedlichen Implementierungsansätze der Zusatzfunktionalität ermöglichen eine hohe Flexibilität bei der Optimierung der Daten-Container-Struktur, die sich an geänderte Anforderungen leicht anpassen läßt.

  Abbildung A.8 stellt ein Beispiel für Zusatzfunktionalität in Form der Datenkompression/Datendekompression des ADC-Inhalts mit dem Paket `java.util.zip` dar.

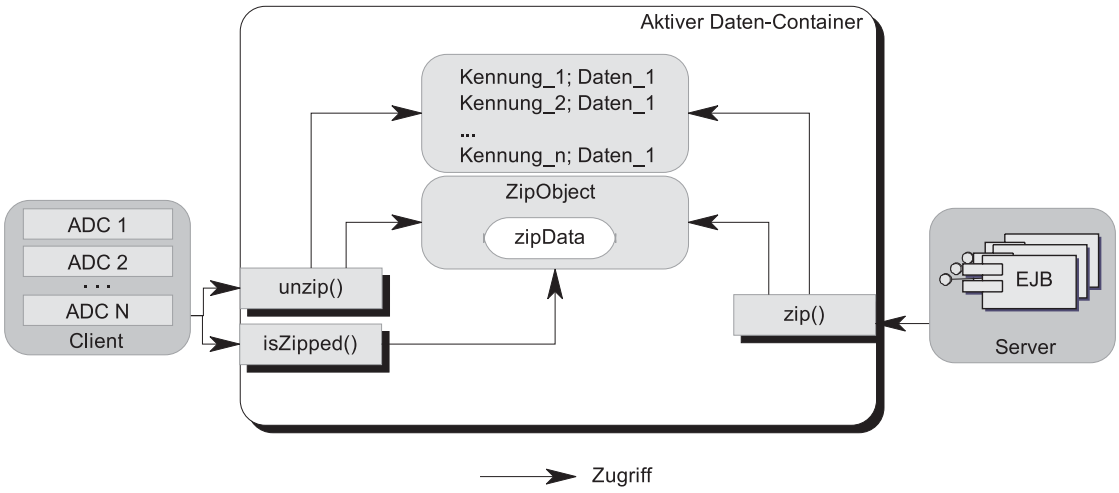


Abbildung A.8: ADC-interne Kompression

Die Kompression und Dekompression erfolgt mit den Methoden `zip()` und `unzip()`. Die Methode `isZipped()` dient der Feststellung, ob der Inhalt eines ADCs komprimiert ist. Der Komprimierungsvorgang selbst und die Speicherung der komprimierten Daten in einer passenden Datenstruktur wird hier in einem darauf spezialisierten Kompressionsobjekt gekapselt. Beim Aufruf der `zip`-Methode des Containers wird ein Kom-

pressionsobjekt erzeugt und die interne Datenstruktur des Daten-Containers übergeben. Anstatt die sonst übliche Datenstruktur zu übertragen, wird nun das Kompressionsobjekt übertragen. Die normale Datenstruktur (hier: `HashMap`) wird auf `null` gesetzt. Bei Aufruf der Methode `unzip` wird die eigentliche Datenstruktur dem Kompressionsobjekt wieder entnommen und dabei dekomprimiert. Nun wird das Kompressionsobjekt nicht mehr benötigt und auf `null` gesetzt.

Eine andere Implementierungsalternative bestünde darin, eigene Serialisierungsmethoden für den ADC bereitzustellen, die automatisch beim Vorgang der Serialisierung/Deserialisierung die Komprimierung/Entkomprimierung vornehmen. Dies ermöglicht die Automatisierung der Kompression, ohne manuelle Eingriffe des Anwendungsentwicklers. Aufgrund der Flexibilität des Containers können auch mehrere Algorithmen angeboten werden, die je nach transportierten Daten eingestellt werden.

A.1.5 Folgen

Der Einsatz von Aktiven Daten-Containern bietet die folgenden Vorteile:

- Daten werden unabhängig von ihrer Quelle immer gleich übertragen und repräsentiert.
- Die Datenübertragung kann in Abhängigkeit der zu transportierenden Daten und den Rahmenbedingungen, wie verwendeter Applikations-Server, zur Verfügung stehende Netzwerkbandbreite, usw. optimiert werden.
- Die Struktur des Daten-Containers erlaubt einen nachträglichen Austausch der Implementierung, ohne bestehenden Code zu tangieren. Der Austausch der Implementierung kann aufgrund geänderter oder neuer Rahmenbedingungen und Anforderungen erforderlich sein.
- Der Daten-Container kann konfigurierbar gestaltetet werden. Damit kann eine feingranulare Anwendung von Zusatzfunktionalität, die zur Optimierung des Leistungsverhaltens benötigt wird, den jeweiligen Rahmenbedingungen angepaßt werden. Die Konfiguration kann z.B. durch den Anwendungsentwickler, im Deployment-Deskriptor der EJB, durch ein Konfigurationsobjekt im JNDI oder durch die Übergabe von JVM-Startparametern erfolgen. Durch dieses Verhalten können wiederum verschiedene Rahmenbedingungen ohne nachträglichen Implementierungsaufwand berücksichtigt werden.
- Der Daten-Container spart Entwicklungsaufwand, da keine manuellen Datenaustauschoperationen zwischen Container und Datenquelle erfolgen müssen. Zusätzlich kann auf die Implementierung vieler Daten-Container für jeden Anwendungsfall verzichtet werden.
- Durch die festgelegte Standardschnittstelle des Containers können weitere Anwendungsteile implementiert werden, die z.B. einen Datenaustausch zwischen GUI-Elementen und dem Container automatisieren und so zu einem reduzierten Implementierungsaufwand führen.

Der Einsatz des Aktiven Daten-Containers besitzt die folgenden Nachteile:

- Die Schnittstelle von EJBs kann an Aussagekraft verlieren, wenn sie nicht ausreichend kommentiert wird.
- Es ist nur eine eingeschränkte Typüberprüfung der Datenaustauschoperationen durch den Compiler möglich.

A.1.6 Beziehungen zu anderen Mustern

Der ADC verwendet das Muster *Dynamische Attribute* [Mow97], um bei großen Geschäftsobjekten nur die für einen Anwendungsfall benötigten Attribute zu reduzieren. Der ADC kann gleichzeitig als *Proxy* bzw. *Smart-Proxy* [Gam95, Vog98, Wil00, Neu99] für Geschäftsobjekte, die sich auf dem Server befinden fungieren und z.B. mittels des Musters *Iterator* große Datenmengen aufteilen, um sie bei Bedarf nachzufordern.
Das Muster *Aktive Value Objects* stellt eine statische Implementierung der Konzepte des ADCs dar. Das Muster *GUI-Manager* und *HTML-Dekorator* stellen speziell auf den ADC abgestimmte Muster dar, die zur Implementierung von Standardkomponenten verwendet werden können, um eine weitere Reduktion des Implementierungsaufwands zu erzielen.

A.2 Aktive Value Objects

A.2.1 Zusammenhang

In einem mehrschichtigen EJB-System müssen Daten aus verschiedenen Quellen und in unterschiedlicher Form zwischen Client und Server übertragen werden. Dieses Muster basiert auf dem Muster *Aktive Daten-Container*, benutzt jedoch statische Transportklassen zur Übertragung, die mit aktiven Konzepten ausgestattet sind.

A.2.2 Probleme

Zur Datenübertragung muß sich jeder Entwickler um die *Beschaffung* der Daten aus der jeweiligen Quelle kümmern, das *Datenformat* oder die Schnittstelle, um auf die Daten zuzugreifen verstehen, die *Entnahme* der benötigten Daten durchführen und ein *Übertragungsformat* in Form eines Daten-Containers festlegen und mit den Daten belegen. Durch diese Anforderungen entstehen eine Reihe von Problemen:

- **Kostenproblematik.** Ein hoher Entwicklungsaufwand besteht darin, Daten aus den vorliegenden Datenstrukturen zu entnehmen und in ein Übertragungsformat zu überführen, oder umgekehrt aus einem Übertragungsformat in eine vorliegende Datenstruktur einzufügen. Zusätzlich müssen die vorliegenden Datenstrukturen häufig mit Schnittstellen versehen werden, die Datenaustauschoperationen erleichtern. Bei Verwendung des Musters *Value Objects* müssen in komplexen Anwendungen für jedes Geschäftsobjekt mehrere Transportobjekte implementiert werden, um die Datenmenge zu reduzieren.

- **Flexibilitätsproblematik.** Ändert ein Client seine Anforderungen dahingehend, daß er andere Daten oder eine größere Datenmenge anfordern will, muß eine Änderung der Transportobjekte erfolgen. Kommt ein neuer Client mit spezifischen Anforderungen hinzu muß mindestens ein neues Transportobjekt implementiert werden. Die Schnittstelle der Server-Komponente muß gemäß der neu hinzukommenden Transportobjekte um eine analoge Anzahl von Methoden erweitert werden. Stellt sich nachträglich heraus, daß eine Beeinflussung der Datenübertragung erfolgen muß, z.B. durch spezielle Kompressionsalgorithmen, muß dies nachträglich in alle Transportobjekte eingebaut werden.

- **Ressourcenproblematik.** Falls ein komplexer Client realisiert werden muß, der in Form eines Applets auf dem Client gestartet wird, kann die Anzahl der vorhandenen Transportklassen problematisch sein, da diese alle zum Client hin übertragen werden müssen. Das selbe Problem existiert, wenn der Platzbedarf für den Java-Client möglichst gering sein soll und dennoch sehr viele Transportklassen verwendet werden sollen.

- **Sicherheitsproblematik.** Bei der kompletten Übertragung von Geschäftsobjektzuständen oder *Value Objects*, die den kompletten Zustand eines Geschäftsobjekts aufnehmen, kann der unerwünschte Effekt auftreten, daß sicherheitsbezogene Daten mit übertragen werden. Um dies zu verhindern, müßten umfangreiche Vorkehrungen getroffen werden, um sensible Daten vor dem Versenden zum Client auszublenden. Die Auslieferung der Geschäftsklassen selbst kann aus Gründen des Know-how-Schutzes unerwünscht sein, da übersetzte Java-Klassen dekompiliert werden können.

- **Leistungsverhalten.** Bei der Implementierung von effizienten Daten-Containern muß das Umfeld z.B. in Form des vorliegenden Applikations-Servers und der vorliegenden Geschäftsdaten berücksichtigt werden. Die Datenmenge und -komplexität muß verringert werden, um ein optimales Leistungsverhalten zu erzielen.

A.2.3 Gründe

Mit *Aktiven Value Objects* kann erreicht werden, daß

- die Datenübertragung zentral beeinflußt werden kann,
- Zusatzfunktionalität zur Optimierung in den Containern ausgeführt wird,
- auf deklarativer Ebene definiert wird, wie die Datenübertragung erfolgen soll,
- die Anwendungsimplementierung vereinfacht wird,
- die Anzahl der zu implementierenden Container-Klassen gesenkt wird,
- Systeme nachträglich im Hinblick auf die Datenübertragung erweitert oder z.B. durch Reduktion der Datenmenge, optimiert werden können,
- auf Wunsch eine Standardschnittstelle zur Verfügung steht, die z.B. von Framework-Komponenten genutzt werden kann,

- eine Zusammenfassung von Daten aus mehreren Quellen erfolgen kann, die in einem Aufruf zum Client übertragen werden können.

A.2.4 Lösung

Struktur

Zur Durchsetzung der geforderten Flexibilität müssen alle im System vorhandenen Daten-Container eine Standardschnittstelle besitzen, die die folgenden Operationen erlaubt:

- **Objektübergabe/Objektentnahme**: Objekte, deren Daten transportiert werden sollen, müssen dem AVO übergeben bzw. entnommen werden können. Der Datenaustausch zwischen den beteiligten Objekten erfolgt dabei automatisch, um die Anwendungsentwicklung zu vereinfachen und zu beschleunigen. Durch Angabe der wirklich vom Client benötigten Attribute wird die Datenmenge reduziert und es genügt die Implementierung eines einzigen AVOs.

- **Generisches Lesen/Schreiben von Attributen**: Attribute müssen vom Typ des AVOs unabhängig gelesen und geschrieben werden können. Dazu gehört auch die Umsetzung von Iteratoren, die ein systematisches Durchlaufen der Attribute und ihrer Werte ermöglichen. Diese Funktionalität kann von verallgemeinerten Komponenten genutzt werden, um beliebige Objekte zu verarbeiten.

- **Konfiguration von Zusatzfunktionalität**: Erforderliche Konfigurationsmethoden für Zusatzfunktionalität, wie z.B. die Anwendung eines Kompressionsalgorithmus, müssen verfügbar sein.

Zur Übertragung mehrerer AVOs kann ein herkömmlicher ADC, wie in Abschnitt A.1 zur Optimierung genutzt werden. Alternativ können Collection-Klassen, wie z.B. `Vector` verwendet werden, um mehrere AVOs gleichzeitig zu übertragen. In Abbildung A.9 ist die Struktur des Konzepts anschaulich dargestellt.
Zur Gewährleistung der Flexibilität können die Container (nachträglich) von außen konfiguriert werden (Informationen zur Konfiguration). Hierzu empfiehlt es sich die Daten-Container zentral in EJBs zu erzeugen und z.B. mittels Deployment-Deskriptor und JNDI zu konfigurieren.

Vorgehensweisen

Im folgenden wird auf die Diskussion, wie eine Standardschnittstelle in den einzelnen AVOs realisiert werden kann und wie ein automatischer Datenaustausch zwischen Geschäftsobjekt und Daten-Container erfolgt, weitestgehend verzichtet. In Abschnitt A.1 wurde erläutert, welche Methoden eine Standardschnittstelle benötigt und wie mittels Java-Reflection oder eines Generatoransatzes auf Objekte zugegriffen werden kann. Diese Ansätze können in AVOs nahezu unverändert übernommen werden. Der einzige Unterschied besteht darin, daß die benötigten Methoden in jedem AVO implementiert sein müssen. Dies kann durch einen Vererbungsansatz

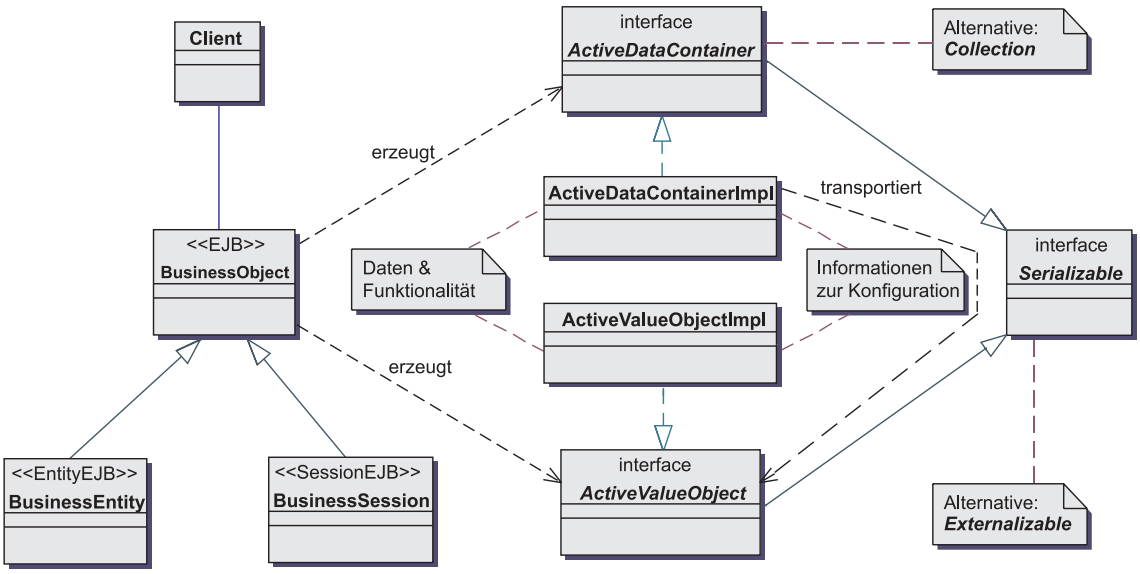


Abbildung A.9: Prinzip von Aktiven *Value Objects*

oder durch die Generierung der Methoden für jedes AVO sichergestellt werden. Der Schwerpunkt der Diskussion liegt in diesem Abschnitt darauf, wie AVOs zur Reduktion des Implementierungsaufwands bei gleichzeitiger Reduktion der Datenmenge und -komplexität bereitgestellt werden können. Dabei werden gezielt andere Implementierungen der Konzepte aus A.1 gewählt, um die Flexibilität und Anpaßbarkeit der vorliegenden Konzepte zu demonstrieren.

- **Vererbungsbasierende Ansätze:** Bei dieser Implementierung müssen alle Daten-Container von einer zentralen Klasse abgeleitet werden, die eine universelle Routine zur Serialisierung ihrer Kindklassen implementiert und eine Standardschnittstelle anbietet, mit der auf Wunsch generisch auf Attribute zugegriffen werden kann. Dieses Konzept ist in Abbildung A.10 dargestellt.

  Die abstrakte Basisklasse `ActiveValueObjectImpl` implementiert die Schnittstelle `Externalizable`, um die Serialisierung vollständig zu steuern. So wird es möglich, die Datenübertragung zu beeinflussen, um Optimierungen vornehmen zu können. Die Schnittstelle `ActiveValueObject` stellt die Standardschnittstelle bereit, auf die dynamisch zugegriffen werden kann, um Attribute im Container zu lesen und zu setzen. Durch die gemeinsame Schnittstelle ergibt sich auch die Möglichkeit EJBs nachträglich durch neue AVOs zu erweitern, ohne den Quellcode zu ändern. Dies kann z.B. bei Entity-Beans angewendet werden, die sehr viele Attribute besitzen. In solchen Fällen ist es wahrscheinlich, daß jede Anwendung, in deren Rahmen die Entity-Bean verwendet werden soll, unterschiedliche Anforderungen an die benötigten Attribute stellt. Durch die nachfolgend dargestellte Schnittstelle kann erreicht werden, daß mit dem Parameter `name` ein beliebiges AVO angefordert werden kann, das der übergebenen Bezeichnung entspricht. Das passende Objekt wird erzeugt, mit Daten belegt und als ADC zum Client übertragen.

  ```
  ActiveValueObject getValueObject(String name)
  ```

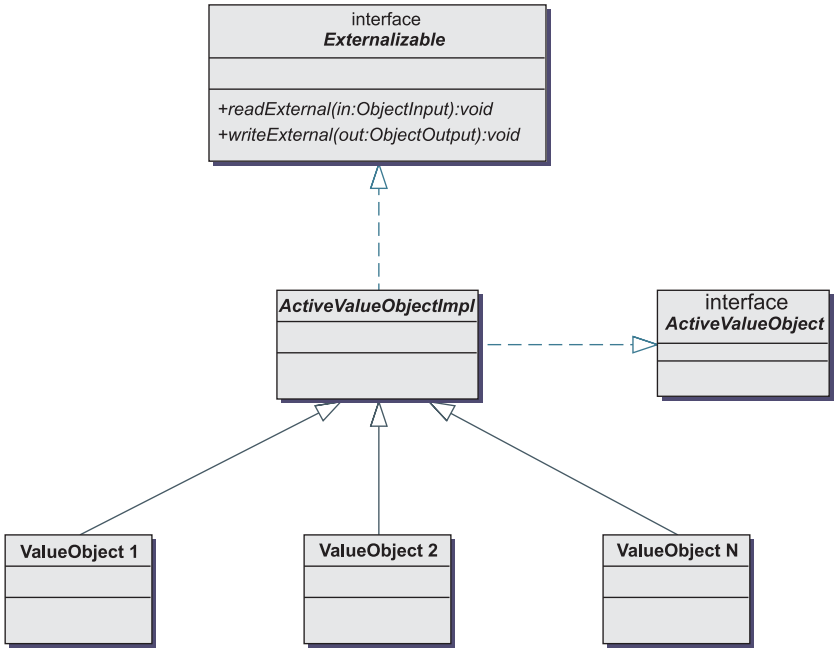


Abbildung A.10: Aktive *Value Objects* mit Vererbung

```
throws RemoteException;
```

Um auf die Attribute und Methoden typsicher zugreifen zu können, muß im Client eine Typumwandlung in den Typ des AVOs erfolgen.

- **Generatorbasierende Ansätze:** Mit Hilfe des in Abbildung A.11 dargestellten Generatorkonzepts kann die automatische Erzeugung von optimierten AVOs erfolgen.

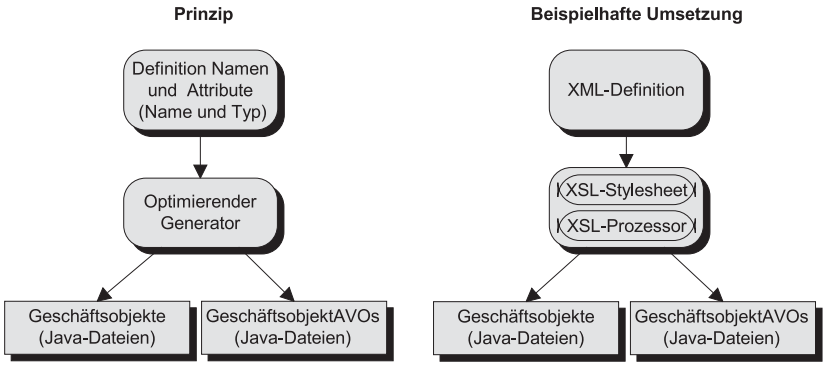


Abbildung A.11: Generatorkonzept zur Erzeugung von AVOs

Anhand der Definition der Attributnamen und Attributtypen wird neben dem Rahmengerüst für eine Geschäftsklasse auch eine zugehörige AVO-Klasse generiert. Die Geschäftsklasse und die AVO-Klasse enthalten bereits neben den üblichen `get`- und `set`-Methoden

für den Datenzugriff alle weiteren Methoden, die zum automatischen Datenaustausch zwischen Geschäftsobjekt und AVO sowie zur Herstellung eines für die Übertragung günstigeren Formats benötigt werden. Die Abbildung enthält zusätzlich einen Umsetzungsvorschlag des Konzepts, der aus einer XML-Definition der Attribute besteht, die anschließend mittels XSL-Stylesheet und XSL-Prozessor in Java-Quellen umgesetzt werden. Eine alternative Implementierung besteht z.B. darin, ein UML-Entwicklungswerkzeug zu verwenden, das eine modifizierbare Codegenerierung aus dem Klassenmodell ermöglicht. Schließlich kann die Generierung von Optimierungsmethoden auch nachträglich erfolgen, indem bereits fertige Geschäfts- und normale AVO-Klassen durch einen Generator erweitert werden.

- Eine einfache Implementierung des AVO-Konzepts mit Hilfe von Generatoren besteht darin, die AVOs so zu generieren, daß jedes einzelne die Schnittstelle `Externalizable` implementiert und in den zugehörigen Methoden `writeExternal()` bzw. `readExternal()` seine Attribute direkt serialisiert bzw. deserialisiert. Dabei wird eine Optimierung der Datenmenge und -komplexität vorgenommen, die auch bei der Einschränkung der zu übertragenden Attribute durch die Übergabe einer Attributliste ansetzt. Zum Transport einer AVO-Folge kann z.B. ein Objekt vom Typ `Vector` verwendet werden, da sich jedes AVO selbst um seine Serialisierung kümmert.
- Ein anspruchsvollerer Implementierungsansatz basiert auf einem ADC, der als Sammelbehälter für eine Folge von AVOs dient, um ggf. noch weitere Optimierungen von Datenmenge und -komplexität zu ermöglichen. Prinzipiell kann hierfür bereits ein ADC aus Abschnitt A.1 verwendet werden, der den Transport von beliebigen serialisierbaren Objekten ermöglicht. Hier wird jedoch ein neuer Daten-Container als Sammelbehälter verwendet, der seine Daten im Sinne einer Komplexitätsreduktion intern als Folge von Bytes (Typ `byte[]`) repräsentiert und diese Daten auf Wunsch zusätzlich mittels des Deflate-Algorithmus komprimieren kann. Der Implementierungsansatz ist in Abbildung A.12 skizziert.

  Geschäftsobjekte deren Daten transportiert werden sollen, implementieren die Schnittstelle `BusinessObject`. Zu jedem Geschäftsobjekt existiert ein AVO, das die Schnittstelle `ActiveValueObject` implementiert. Die Implementierung des AVOs greift dabei auf die Klasse `Helper` zurück, die für das Lesen und Schreiben der Daten und eine evtl. mögliche Optimierung zuständig ist. Eine Folge von AVOs wird mit der Klasse `AVOContainer` zur Übertragung zusammengefaßt. Die Methode `toValueObject()` in der Schnittstelle `BusinessObject` erlaubt die Erzeugung von neuen AVOs, die mit den aktuellen Attributausprägungen des Geschäftsobjekts belegt sind, oder die Wiederverwendung von bestehenden AVOs, deren Attribute überschrieben werden. Der umgekehrte Weg kann z.B. mittels einer Methode `fromValueObject()` implementiert werden. In der Schnittstelle `ActiveValueObject` sind die Methoden zum Herstellen eines Datenformats zur Übertragung enthalten. Durch Übergabe einer Liste können gezielt Attribute zur Übertragung ausgewählt werden. Attribute werden durch die zur Optimierung vorgesehene Hilfsklasse `Helper` in einen für primitive Datentypen bereitgestell-

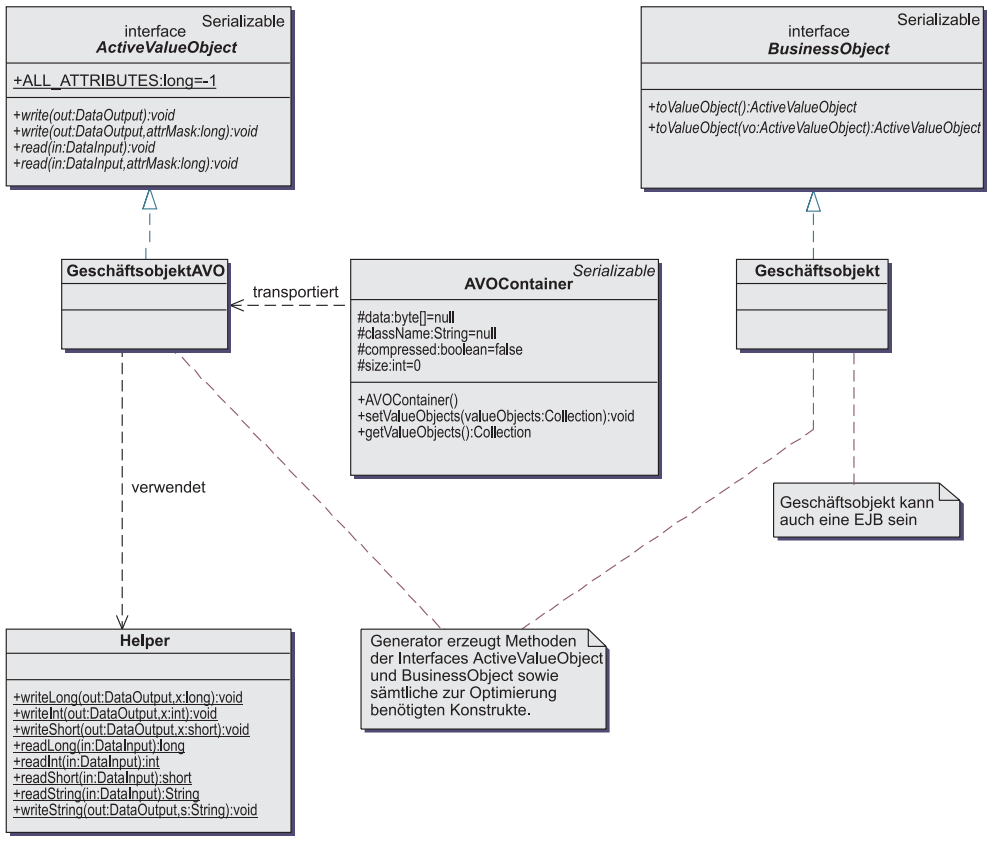


Abbildung A.12: Umfassende AVO-Implementierung

ten binären Datenstrom vom Typ `DataOutput` (vgl. dazu [Mica]) geschrieben. Falls alle Attribute transportiert werden sollen existiert eine zweite Implementierung der Methode `write()`, die alle Attribute in den Datenstrom schreibt und auf die dann überflüssigen Vergleichsoperationen vollständig verzichtet. Der Datenstrom `out` wird vom `AVOContainer` bereitgestellt, der die gesamten Daten einer Folge von AVOs zur optimierten Übertragung aufnimmt. Der Container ruft hierbei bei Übergabe einer Folge von AVOs in Abhängigkeit davon, ob alle Attribute oder nur ausgewählte übertragen werden sollen, die entsprechende `write`-Methode jedes Objekts auf. Bei der Serialisierung von `AVOContainer` wird damit die interne Byte-Struktur mit sehr niedriger Komplexität übertragen.

A.2.5 Folgen

Der Einsatz von *Aktiven Value Objects (AVOs)* hat die folgenden Vorteile:

- Es existiert nur ein Daten-Container pro Geschäftsobjekt bei gleichzeitig möglicher Einschränkung der Attributmenge.
- Es erfolgt eine Trennung von Schnittstelle und Implementierung. Die interne Datenstruktur kann für die vorliegende Umgebung optimiert und jederzeit angepaßt werden.

- Die Daten-Container besitzen eine Standardschnittstelle, die es ermöglicht weitere allgemeingültige Komponenten zu implementieren. Über die Schnittstelle kann auf Daten immer auf die selbe Art und Weise zugegriffen werden.
- Die Container können zur Laufzeit konfiguriert werden und sich flexibel an ihre Systemumgebung anpassen.
- Der Entwicklungsaufwand wird reduziert, da automatische Datenaustauschoperationen stattfinden.
- Auf die Attribute kann auch typsicher zugegriffen werden.

Der Einsatz von AVOs ist mit den folgenden Nachteilen verbunden:

- Zur signifikanten Reduktion des Entwicklungsaufwands wird ein Generator benötigt, der zusätzlich implementiert werden muß.
- Pro Geschäftsobjekt muß ein AVO implementiert werden.

A.2.6 Beziehungen zu anderen Mustern

Im wesentlichen basieren AVOs auf den Eigenschaften von *Aktiven Daten-Containern*, die hier statisch umgesetzt werden. Als statisches Datenübertragungskonzept, das nicht auf der Übertragung von Geschäftsobjekten beruht, ist auch eine Beziehung zum Muster *Value Objects* vorhanden. Die Beschränkung auf die Übertragung der wirklich notwendigen Attribute entspricht der Umsetzung des Musters *Dynamische Attribute* [Mow97]. Durch Kapselung einer Referenz auf eine entfernte EJB kann ein AVO ebenfalls zum *Proxy* bzw. *Smart Proxy* [Gam95, Vog98, Wil00, Neu99] werden.

A.3 GUI-Manager

A.3.1 Zusammenhang

Der GUI-Manager verwaltet die Benutzerschnittstelle einer Anwendung und verhindert den manuellen Datenaustausch zwischen universellen Daten-Containern und den einzelnen GUI-Elementen.

A.3.2 Probleme

Die Entwicklung von anspruchsvollen Benutzeroberflächen stellt einen erheblichen Aufwand in der Entwicklung von modernen Client/Server-Anwendungen dar. Java stellt mit der Swing-Bibliothek [Mica] eine umfangreiche Sammlung von grafischen Elementen zur Verfügung, die zur Anzeige und Eingabe von Anwendungsdaten verwendet werden können. Dabei müssen Daten, die zwischen Client und Server fließen sollen, ständig in die grafischen Elemente eingebracht, bzw. entnommen werden. Die dazu notwendigen Routinen sind durch jeden Entwickler mühsam für seinen Bereich zu implementieren und zu pflegen. Dies stellt insbesondere dann

eine Herausforderung dar, wenn es sich um komplexe grafische Elemente, wie z.B. eine Tabelle handelt. Das Entwurfsmuster GUI-Manager soll den Entwickler von diesen Aufgaben weitgehend entlasten und zu einer Einsparung von Codierungsaufwand führen, indem die Transportobjekte gleichzeitig als Datenmodell des Clients verwendet werden. Dies wird dadurch erreicht, daß der GUI-Manager ADC-Objekte bzw. AVOs entgegennimmt und die darin transportierten Daten automatisch in die dafür vorgesehenen GUI-Elemente einfügt. Umgekehrt können geänderte Daten wieder aus den GUI-Elementen entnommen werden und in Daten-Containern bereitgestellt werden.

A.3.3 Gründe

Das vorliegende Entwurfsmuster führt zur Reduktion des Entwicklungsaufwands, indem der Datenaustausch zwischen GUI-Elementen und standardisierten Daten-Containern automatisch erfolgt. Zusätzlich werden auf dem Client weniger Klassen benötigt, da Daten-Container als Datenmodell verwendet werden.

A.3.4 Lösung

Struktur

In Abbildung 4.35 ist die Grundstruktur des Konzepts dargestellt. Das Kernelement ist der GUI-Manager oder ein GUI-Builder. Ziel des GUI-Managers ist es, Funktionalität zur Darstellung von Anwendungsfenstern und zur Kommunikation von Fenstern untereinander zu verallgemeinern, um eine höhere Flexibilität und leichtere Wartbarkeit der Anwendung zu gewährleisten. Zu den Aufgaben gehört z.B.:

- Die Präsentation von Anwendungsfenstern unabhängig davon, ob die Anwendung von der Kommandozeile oder als Applet innerhalb eines Web-Browsers gestartet wurde.
- Die Bereitstellung eines zentralen Mechanismus zum Austausch von Daten zwischen verschiedenen Dialogen.
- Die Steuerung von Dialogfolgen.

Die Aufgaben eines nichtvisuellen GUI-Builders liegen in der Konstruktion von Dialogen. Er wird häufig in Frameworks zur Anwendungsentwicklung eingesetzt, um für eine bestimmte Anwendung spezialisierte GUI-Elemente, Fensterteile oder ganze Fenster per Programmanweisung zu erzeugen. Dabei wird versucht den Anwendungsentwickler von komplizierten Layoutfragen zu entlasten und keine Abhängigkeit zu Entwicklungswerkzeugen zu schaffen, die visuelle GUI-Builder besitzen und aus den konstruierten Dialogen Code erzeugen, der den Anforderungen einer Anwendungsentwicklung nicht genügt. Bei Verwendung eines GUI-Builders können gleichzeitig Routinen realisiert werden, die den Aufgaben eines GUI-Managers gerecht werden.
Der GUI-Manager kann ADC-Objekte entgegennehmen und ordnet die enthaltenen Daten anhand eines Deskriptors einzelnen GUI-Elementen, oder Gruppen davon, zu. Bei der Manipulation von Daten in den GUI-Elementen werden diese als geändert gekennzeichnet und können

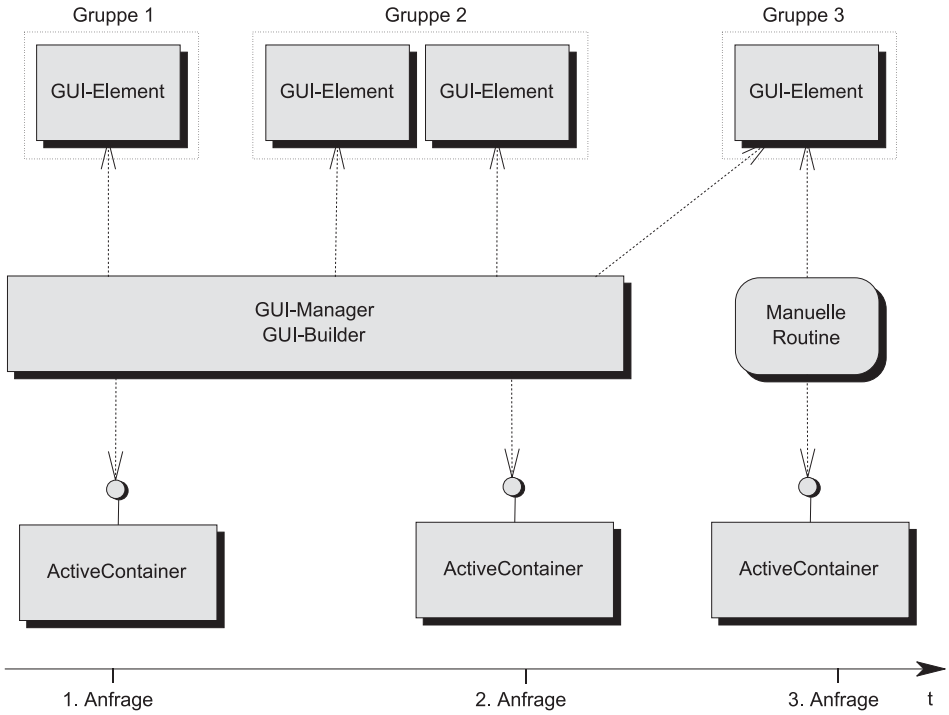


Abbildung A.13: GUI-Manager

vom GUI-Manager wiederum in einem ADC-Objekt zurückgegeben und zum Versand an den Server geschickt werden. Dabei kann eine Folge von Kommunikationsvorgängen, wie in Abbildung A.13 gezeigt, entstehen. Eine erste Anfrage führt zur Lieferung eines ADC-Objekts, das die Daten zur Darstellung in dem einzigen GUI-Element von *Gruppe 1* enthält. Nachdem der Benutzer die Daten bearbeitet hat, wird vom GUI-Manager ein ADC mit den geänderten Daten bereitgestellt, der zum Server gesendet wird (2. Anfrage). Aus der zweiten Anfrage resultiert wiederum eine Reihe von Daten, die in die zwei GUI-Elemente der *Gruppe 2* gesetzt werden. Das GUI-Element von *Gruppe 3* repräsentiert Elemente, die prinzipiell auch dem GUI-Manager bekannt sind, hier aber manuell, durch eigene Routinen des Anwendungsentwicklers mit Daten belegt werden. Bei Anwendung des GUI-Managers können nahezu sämtliche Codezeilen zur Darstellung von Daten in GUI-Elementen eingespart werden, da nur ein Deskriptor erstellt werden muß, der die Zuordnung zwischen Daten und GUI-Elementen definiert und ein Aufruf zur Belegung der GUI-Elemente mit Daten, bzw. zur Entnahme von Daten aus den GUI-Elementen notwendig ist. Insbesondere bei komplexen GUI-Elementen, wie z.B. Tabellen tritt dadurch eine signifikante Entlastung des Entwicklers auf.

Vorgehensweisen

Im Mittelpunkt der Implementierung des vorliegenden Konzepts steht die Entscheidung darüber, wie die Zuordnung der transportierten Daten auf die grafischen Elemente spezifiziert wird, wann diese Spezifikation zum Einsatz kommt und wo letztlich der Zugriff auf die GUI-Elemente erfolgt. In Abbildung A.14 wird die Zuordnung grundsätzlich anhand eines Deskriptors aus logi-

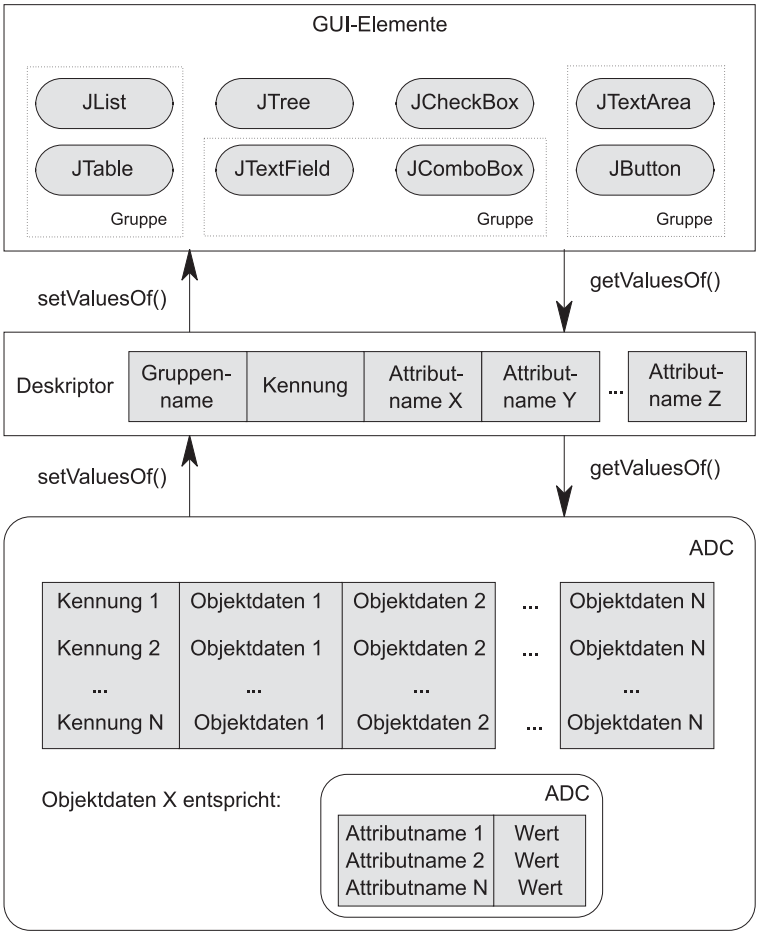


Abbildung A.14: Zuordnung mittels Deskriptoren

scher Sicht dargestellt.

Der Deskriptor definiert für jedes GUI-Element den Namen der Gruppe, zu dem es gehört. Durch mehrfache Vergabe des gleichen Namens entstehen Gruppen, die aus mehreren unterschiedlichen Elementen bestehen können. Weiterhin wird spezifiziert unter welcher Kennung die Daten des GUI-Elements im ADC abgelegt sind, um eine Zuordnung beim Transport der Daten vieler GUI-Elemente vornehmen zu können. Im Anschluß daran wird definiert, welche Attribute zur Darstellung im GUI-Element verwendet werden sollen. Dies ermöglicht die Übertragung von Daten, die nicht dargestellt werden, aber trotzdem für weitere Bearbeitungsschritte benötigt werden. Dies ist z.B. für Primärschlüsselattribute der Fall, die für weitere Datenbankoperationen (Insert, Delete, Update) benötigt werden. Beim Schreiben der Daten durch einen GUI-Manager (`setValuesOf()`) werden durch Übergabe des Gruppennamens und des vom Server bezogenen ADC-Objekts die Deskriptoren herangezogen, um die zugehörigen Füllungsvorgänge anzustoßen. Beim Lesen von Daten findet der Vorgang umgekehrt statt. Anhand des übergebenen Gruppennamens wird der Deskriptor herangezogen, um aus GUI-Elementen entnommene Daten korrekt im verwendeten ADC abzulegen und diesen anschließend zum Server zu schicken. Hierzu ist es sinnvoll in die ADC-Objekte eine Kennung zu set-

zen, die darüber Auskunft gibt, ob es sich bei den Daten, um neue, geänderte oder zu löschende Daten handelt. Dies ist u.a. bei komplexen GUI-Elementen, wie z.B. Tabellen sinnvoll, da hier Zeilen geändert, gelöscht und neu eingefügt werden können. Als Alternative können auch Routinen implementiert werden, die nur geänderte, gelöschte oder neue Daten zurückgeben. Das folgende Codefragment skizziert die soeben geschilderten Abläufe:

```
// Client
...
// Fordere Daten an
ADC daten=enterpriseBean.holeDaten();
guiManager.setValuesOf("meineGruppe", daten);

// Verarbeitung der Daten
...
// hole alle neu eingegebenen Daten
ADC daten2=guiManager.getValuesOf("meineGruppe", "insert");
enterpriseBean.uebergebeDaten(daten2);


// Server
ADC daten=new ADC();
...
// Suche Objekte
...
Vector treffer=session.executeQuery(...);

// Verpacke die Daten der gefundenen Objekte
daten.setObjects(treffer);
return daten;

// Erzeuge neue Objekte aus dem zurueckgegebenen ADC
Vector daten2.getObjects("kennung");

// Speichere die Objekte
...
```

Um das dargestellte Verhalten zu ermöglichen, muß der Inhalt des Deskriptors beim Aufrufen der `xxxValuesOf`-Methoden bekannt sein. Als Beispiel hierfür wird hier der Deskriptor direkt beim Erzeugen des GUI-Elements durch einen nichtvisuellen GUI-Builder übergeben, wie im folgenden Codefragment dargestellt:

```
// Client
String deskriptor="meineGruppe,
                   kennung,
                   attribut1,
                   attribut2,
                   ...,
```

```
                    attributN";

JComponent jc=guiBuilder.create("ComboBox", ..., deskriptor);

// Hinzufuegen des Elements zu einem Fenster (Panel)
panel.add(jc);
...
```

Häufig werden im Rahmen der Anwendungsentwicklung Java-Beans erzeugt, deren Verhalten parametrisiert ist und die von bestehenden GUI-Komponenten der Java-Bibliothek abgeleitet sind. Dabei kann z.B. mittels Parametern angegeben werden, welche Zeichen eingegeben werden dürfen und welcher Wertebereich für diese zulässig ist. Der Deskriptor kann dabei Bestandteil der übrigen Definitionen sein oder zusätzlich übergeben werden.
Falls ein visueller GUI-Builder mit projektspezifischen GUI-Elementen verwendet wird, kann der Deskriptor z.B. als *Property* einer Komponente definiert werden. Bei Verwendung von einfachen Java-GUI-Elementen mit einem visuellen GUI-Builder muß dafür gesorgt werden, daß die Deskriptoren mit einer separaten Methode hinterlegt werden können:

```
// Client
myPanel.setDeskriptor("Name", deskriptor);
```

Dabei ist z.B. der Name des GUI-Elements mit dem entsprechenden Deskriptor zu übergeben. Die Funktionalität der Methode wird dabei durch eine Basisklasse zugesteuert, von der jedes Anwendungsfenster abgeleitet wird. Intern wird dabei eine Datenstruktur erzeugt, die alle GUI-Elemente und ihre Deskriptoren des Fensters enthält, um diese Informationen an den GUI-Manager bei Aufruf von `xxxValuesOf`-Methoden zur Verfügung zu stellen.

A.3.5 Folgen

Mit Hilfe des GUI-Managers können Daten automatisch zwischen GUI-Elementen und Aktiven Daten-Containern (bzw. Aktiven Value Objects) realisiert werden. Dabei werden Daten-Container direkt als Datenmodell auf dem Client verwendet. Neben den Austauschoperationen werden somit auch weitere Klassen eingespart, die sonst zur Datenhaltung erforderlich wären.
In besonders komplizierten Fällen können manuelle Eingriffe notwendig werden und die Einsparung von Implementierungsaufwand vereiteln. Dies muß hingenommen werden, um die Komplexität der notwendigen Deskriptoren und Klassen nicht zu erhöhen.

A.3.6 Beziehungen zu anderen Mustern

Das Muster setzt Daten-Container voraus, die nach aktiven Konzepten gestaltet sind und eine Standardschnittstelle zum Datenaustausch anbieten. Somit besteht die Beziehung zu *Aktiven Daten-Containern* und *Aktiven Value Objects*.

A.4 HTML-Dekorator

A.4.1 Zusammenhang

Der HTML-Dekorator erzeugt aus Daten-Containern, die nach aktiven Konzepten gestaltet sind, eine HTML-Darstellung. Er dient als Beispiel, wie allgemeine Komponenten zur Umformung der in den Container transportierten Daten verwendet werden können.

A.4.2 Probleme

Neben reinen Java-Anwendungs-Clients existieren noch eine Reihe anderer Clients, die auf eine EJB-Anwendung zugreifen müssen. Dazu gehören z.B. WWW-Browser, die auf eine HTML-Darstellung zurückgreifen, mobile Endgeräte, wie z.B. Handys, die mittels WAP-Browser WML-Seiten darstellen und Fremdsysteme, die ein XML-Format zum Datenaustausch benötigen. Um solche Systeme mit Daten zu versorgen, kann eine allgemeine Komponente erstellt werden, um die Daten in der geforderten Form bereitzustellen.

A.4.3 Gründe

Es wird eine verallgemeinerte Komponente erstellt, die eine Umformung der Daten vornimmt. Dadurch erfolgt eine Zentralisierung dieser Aufgaben, die mit einer Reduktion des Entwicklungsaufwands verbunden ist. Zusätzlich können Daten unabhängig von ihrer Quelle umgeformt werden, da sie in Form von standardisierten Daten-Containern vorliegen.

A.4.4 Lösung

Struktur

Die Struktur ist in Abbildung A.15 dargestellt. HTML-Servlets verwenden den HTML-Dekorator, um eine HTML-Präsentation der Daten in einem Aktiven Daten-Container (ADC) zu erhalten. Der HTML-Dekorator basiert auf der Schnittstelle des ADCs und erweitert dessen Funktionalität auf dem Server, um HTML-Clients den Zugriff auf die Daten zu ermöglichen.

Vorgehensweisen

Zur universellen Implementierung des Dekorators kann er mit zwei Methoden versehen werden. Die `config`-Methode dient zur Konfiguration des Dekorators. Dabei können z.B. Darstellungsform und maximal dargestellte Datenmenge festgelegt werden. Diese Parameter können auch aus einem HTML-Formular entnommen werden, falls der Systembenutzer Einstellungen dieser Form vornehmen kann. Eine Konfiguration kann auch im Konstruktor der Dekorator-Klasse erfolgen. Die `config`-Methode ermöglicht jedoch die dynamische Konfiguration während des gesamten Lebenszyklus des Objekts und verhindert so, daß bei jeder Anfrage, die eine Parameteränderung nach sich zieht, ein neues Objekt erzeugt werden muß.
Um die Anfragen von Servlets entgegenzunehmen, erzeugt das HTML-Servlet zu Beginn seines Lebenszyklus ein HTML-Dekoratorobjekt (1). Die Anfrage eines HTML-Clients bewirkt

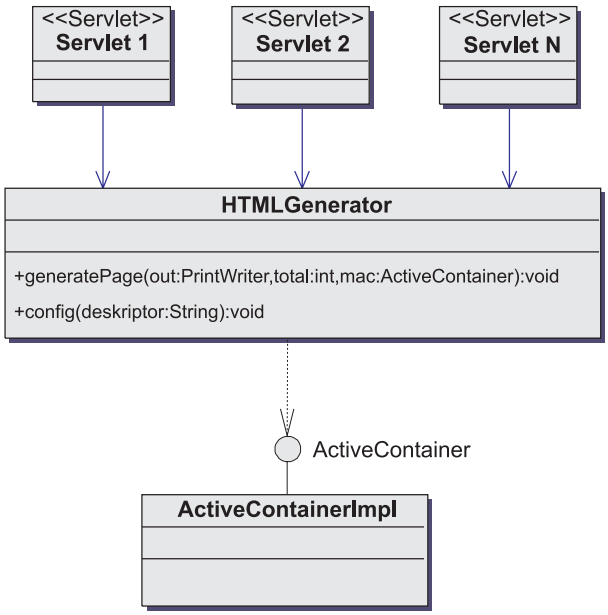


Abbildung A.15: HTML-Dekorator

den Aufruf der Methode `doGet()` des Servlets und übergibt dabei u.a. Parameter, die im Servlet zum Aufruf einer EJB-Geschäftsmethode führen (2.1). In der EJB wird ein ADC erzeugt, mit den angeforderten Daten gefüllt und an das Servlet zurückgeschickt (Rückgabe von 2.1). Entsprechend des Geschäftsprozesses wird das Dekorator-Objekt mittels der `config`-Methode konfiguriert, um die geforderte HTML-Darstellung zu liefern (2.2). Anschließend wird die `generateHTML`-Methode aufgerufen, um die HTML-Seite zu erzeugen (2.3). Innerhalb der Methode werden alle im ADC transportierten Daten in Form von ADC-Objekten durchlaufen (2.3.1, `elementAt`-Methode). Hierzu erzeugt der ADC entsprechende ADC-Objekte, falls die Daten zum Zwecke der Massendatenübertragung optimiert wurden (2.3.1.1) und gibt diese an das Servlet zurück (Rückgabe von 2.3.1). Das Servlet verwendet die ADC-Objekte als Stellvertreter für Geschäftsobjekte und fragt alle im ADC enthaltenen Attributbezeichnungen ab (3, `getAtts`-Methode), um diese anschließend mittels `get`-Methoden zu entnehmen (4). Der eigentliche Vorgang der HTML-Erzeugung ist in Abbildung 4.39 skizziert.

A.4.5 Folgen

Mit Hilfe der Implementierung eines HTML-Dekorators können beliebige, im gekapselten Daten-Container enthaltenen Daten, in eine HTML-Darstellung überführt werden. Im Idealfall genügt ein Dekorator, um alle notwendigen Darstellungen zu generieren. Der Dekorator kann aufgrund seines dynamischen Aufbaus sehr komplex werden, falls viele unterschiedliche und anspruchsvolle Darstellungsformen vorhanden sind. In solchen Fällen sind evtl. mehrere Dekorator-Implementierungen erforderlich.

A.4.6 Beziehungen zu anderen Mustern

In der vorliegenden Form bezieht sich das Muster auf Daten-Container, die nach aktiven Konzepten gestaltet sind (*Aktive Daten-Container* und *Aktive Value Objects*).

Anhang B

Test- und Untersuchungskonzepte

B.1 Zielsetzung

Das Ziel dieses Kapitels besteht darin, einige Anforderungen an ein Werkzeug zu formulieren, das zur Durchführung von Tests hinsichtlich des Leistungsverhaltens und der Korrektheit eines J2EE-Systems durchzuführen. Im Rahmen dieser Arbeit wurde eine solches Werkzeug prototypisch entwickelt, da kommerzielle Werkzeuge die bestehenden Anforderungen hinsichtlich Flexibilität, Erweiterbarkeit und Plattformunabhängigkeit nicht erfüllen konnten. Das Testwerkzeug wurde dabei zur Untersuchung der im Rahmen dieser Arbeit entwickelten Datenübertragungskonzepte herangezogen. Darüber hinaus kam das Werkzeug in einem Industrieprojekt der *iT media Consult GmbH* zum Einsatz. In diesem Kapitel werden die realisierten Testkonzepte und deren prototypische Implementierung beschrieben. Eine ausführliche Auseinandersetzung mit Testverfahren der Software-Technik würde den Rahmen dieser Arbeit sprengen. Deshalb wird in diesem Abschnitt eine pragmatische Beschreibung von Testaufgaben und Testdurchführungen im Rahmen der Entwicklung einer EJB-Anwendung gewählt, die sich auf Komponenten- und Lasttests beschränken. Für eine Ausführliche Abhandlung von Testverfahren sei hier z.B. auf [Bal98] verwiesen.

B.2 Testaufgaben

In diesem Abschnitt werden die verschiedenen Aufgaben beim Testen einer EJB-Anwendung beschrieben, die das universelle Datenübertragungskonzept einsetzt.

- **Testkonfiguration:** Es ist erforderlich festzulegen, welche EJBs getestet werden sollen und auf welchem Applikations-Server sich diese befinden. Entsprechend dieser Auswahl muß ein entsprechender Client bereitgestellt und konfiguriert werden, der mit dem ausgewählten Server kommunizieren kann. Dabei ist zu berücksichtigen, daß notwendige Bibliotheken, evtl. zu generierende *Stubs* und *Skeletons* zur Verfügung stehen und daß der JNDI-Kontext des Servers kontaktiert werden kann. Soll ein Lasttest erfolgen, muß festgelegt werden, wieviele Clients an dem Test teilnehmen sollen und ob diese auf mehrere Maschinen verteilt werden.

- **Testfalldefinition.** Nachdem die entsprechende Testkonfiguration hergestellt ist, muß entschieden werden, welche Methoden der EJBs getestet werden sollen. Diese Methoden müssen vom Client ausgeführt werden. Bei einem Lasttest kann es ausreichend sein, die Aufrufe nur durchzuführen, ohne eine Kontrolle der Ergebnisse vorzunehmen. Bei einem Korrektheitstest muß die Kontrolle der erhaltenen Ergebnisse erfolgen, um festzustellen, ob die Methode fehlerfrei ausgeführt wurde. Bei einem solchen Test muß der Inhalt eines zurückgelieferten Daten-Containers automatisch oder manuell untersucht werden.

- **Testfallerstellung.** Nach der Erstellung von Testfalldefinitionen müssen diese mit Hilfe des entsprechenden Clients umgesetzt werden. Idealerweise wird ein parametrisierter Client erstellt, der auf den gewünschten Testfall konfiguriert wird. Bei einer Unterstützung durch Werkzeuge muß somit eine möglichst flexible Definition von Parametern möglich sein, die es auch ermöglicht verschiedene Clients verschieden zu konfigurieren, um die Benutzung eines Systems durch reale Benutzer zu simulieren, die quasi gleichzeitig unterschiedliche Anfragen an das System stellen.

- **Testdurchführung.** Liegen ein oder mehrere Test-Clients vor, müssen diese zur Ausführung gebracht werden. Bei einem Lasttest muß evtl. eine Verteilung auf mehrere Maschinen erfolgen. Häufig stellt sich heraus, daß die Testfälle angepaßt und erweitert werden müssen. D.h. die Implementierung der Clients muß mehrfach geändert werden und erneut zur Ausführung kommen.

- **Testauswertung.** Nachdem eine Testkonfiguration abgelaufen ist, muß eine Auswertung der Ergebnisse erfolgen. Bei einem Unit-Test ist es erforderlich die Korrektheit aller Ergebnisse in den einzelnen Testfällen sicherzustellen. Bei einem Lasttest kann eine Begutachtung des Systemdurchsatzes und der erhaltenen Antwortzeit erfolgen. Oftmals genügt die Prüfung, ob ein bestimmter Durchsatz erreicht wurde und eine bestimmte durchschnittliche Antwortzeit im System erreicht werden konnte.

B.3 Entwicklungsprozeßintegration

Für den effektiven Einsatz von Testkonzepten ist es notwendig, daß eine Integration in den Entwicklungsprozeß eines Softwaresystems erfolgt. Dabei muß festgelegt werden, in welchen Phasen des Entwicklungsprozesses das Werkzeug zum Einsatz kommen kann und wie die bereitgestellte Funktionalität angewendet wird.

- **Produkttest.** Zur Bestimmung der Entwicklungs- und Produktivumgebung muß die Auswahl von Applikations-Servern, Datenbanken und anderen Entwicklungswerkzeugen erfolgen. Dabei wird dem Leistungsverhalten der einzelnen Produkte oft ein hoher Stellenwert beigemessen. Um die Produkte unter einer höheren Belastung testen zu können, muß ein Werkzeug verwendet werden, das viele Clients simulieren kann.

- **Design.** Verschiedene Entwurfsansätze und Anwendungsstrukturen müssen ebenso wie Produkte auf ihr Leistungsverhalten geprüft werden. Dies ist insbesondere auch für Prototypen relevant, die zur Evaluierung von technischen Aspekten erstellt werden. Zur Be-

lastung des erstellten Prototypen muß wiederum ein Werkzeug verwendet werden, daß viele Clients simulieren kann.

- **Implementierung.** Während der Implementierung sollten Anwendungsentwickler, die EJBs erstellen die Möglichkeit erhalten, diese auf ihr Lastverhalten hin zu testen und offensichtliche Schwächen in der Implementierung zu korrigieren. Zusätzlich sollte der erstellte Code auf seine Korrektheit geprüft werden. Dies kann z.B. in Form von Unit-Tests erfolgen, die mit Hilfe eines Test-Frameworks erstellt werden. Mit JUnit [JT] steht ein solches Framework frei erhältlich zur Verfügung.

- **Test.** In dieser Phase wird ein Test eines größeren Anwendungsteils oder der ganzen Anwendung durchgeführt. Im Rahmen von J2EE-Anwendungen spielt hier vor allem die Integration und das Zusammenspiel zwischen verschiedenen Komponenten eine Rolle.

- **Wartung.** In der Wartung müssen hauptsächlich die Tests der Implementierungs- und Testphase erneut durchlaufen werden, um die Auswirkungen von Änderungen, Erweiterungen und Fehlerkorrekturen zu überprüfen.

B.4 Testwerkzeugarchitektur

In den nachfolgenden Abschnitten wird die Architektur des prototypisch implementierten Testwerkzeugs näher beschrieben. Es unterstützt den Entwickler dabei bei den in Abschnitt B.2 dargestellten Testaufgaben und kann, wie in Abschnitt B.3 beschrieben, während des kompletten Lebenszyklus einer Anwendung zum Einsatz kommen. Das Werkzeug besteht aus mehreren Komponenten, die mittels CORBA kommunizieren. Das System kann vom Benutzer selbst flexibel angepaßt werden und stellt eine Reihe von Werkzeugen bereit, die zur Untersuchung von EJB-Systemen herangezogen werden können.

B.4.1 Überblick

In Abbildung B.1 ist der schematische Aufbau des Testwerkzeugs enthalten. Kernelement des Testwerkzeugs ist der Koordinator, der die Objektfabriken steuert und mittels einer grafischen Oberfläche benutzt werden kann. Es handelt sich dabei um eine Umsetzung des Entwurfsmusters *Mediator* [Gam95]. Die Objektfabriken sind in der Lage beliebige Testobjekte zu erzeugen und ihren Lebenszyklus zu kontrollieren (vgl. dazu auch die Entwurfsmuster *Abstract Factory* und *Factory Method* in [Gam95]). Diese Objekte rufen z.B. entfernte Methoden von Server-Komponenten auf oder fordern HTML-Seiten eines Web-Servers an. Sämtliche Systemkomponenten sind als CORBA-Objekte realisiert und kommunizieren mittels eines *Object Request Brokers (ORB)* (vgl. dazu Kapitel 2, Abschnitt 2.2). Damit können die Systemkomponenten beliebig in einem Rechnernetzwerk verteilt werden. Das Auffinden der Objekte untereinander wird durch den CORBA-Namensdienst, der sich ebenfalls auf einem Rechnerknoten befinden muß, gewährleistet. Nachfolgend werden die Komponenten im einzelnen erläutert.

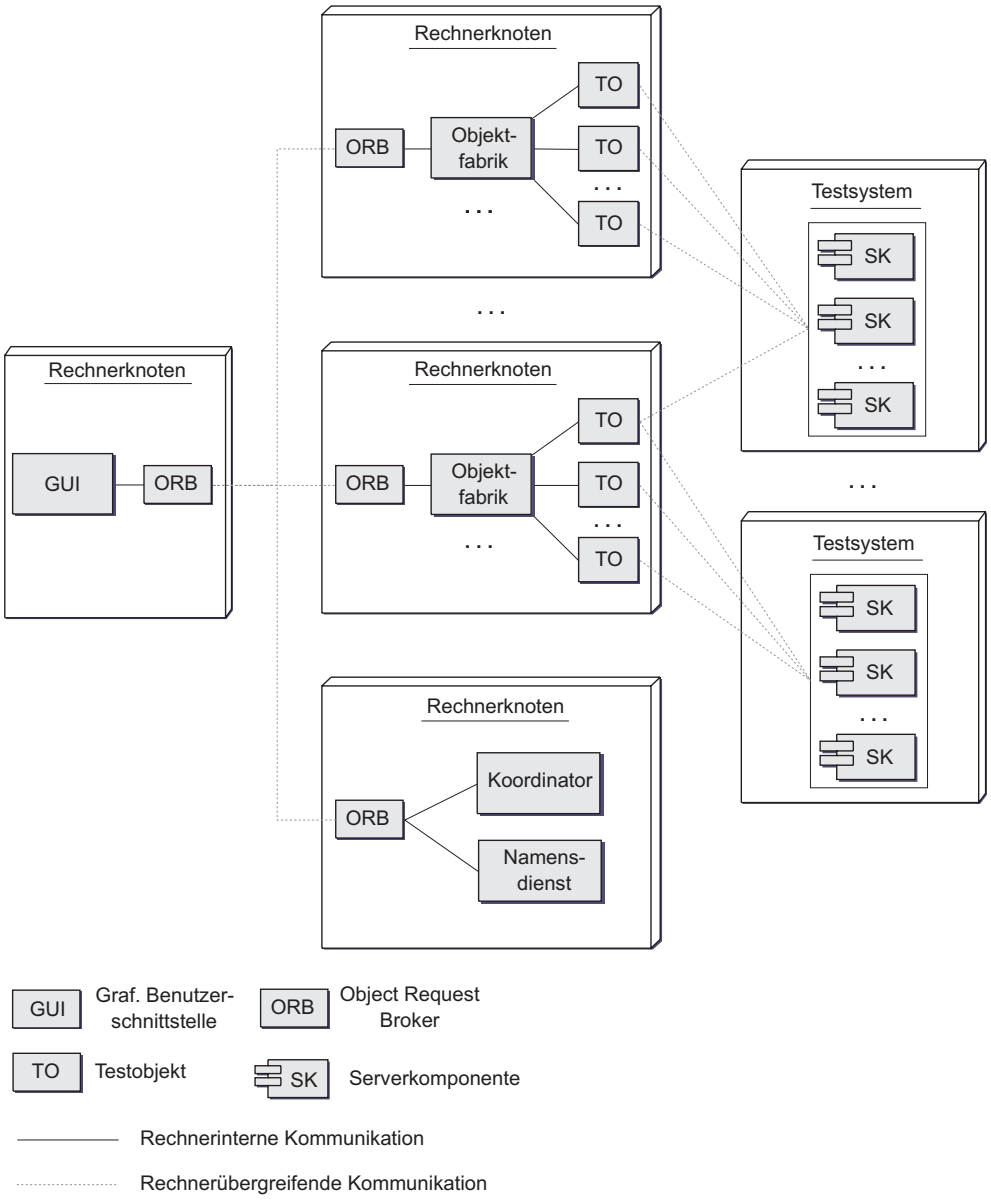


Abbildung B.1: Architekturüberblick

B.4.2 Koordinator

Neben der Kommunikation mit den Objektfabriken übernimmt der Koordinator die Aufgabe, sämtliche Informationen im System zu verwalten und zentral bereitzustellen. Aufgrund dieser Systemarchitektur befindet sich das Datenmodell im Koordinator. Zu den verwalteten Informationen gehören:

- Im System angemeldete Objektfabriken.
- Objekte, die benutzerabhängig von den Objektfabriken erzeugt wurden.

- Parameter, die für das Gesamtsystem, einzelne Objektfabriken oder einzelne Objekte angegeben wurden.

- Ergebnisse von abgelaufenen Tests.

Sämtliche Daten werden in einer Baumstruktur zusammengefaßt und verwaltet. Das Prinzip ist als Übersicht in Abbildung B.2 abgebildet. Das Gesamtsystem wird durch die Wurzel repräsentiert und besteht aus den Rechnerknoten, die im System angemeldet sind. Jeder Rechnerknoten verweist auf seine Objektfabriken, die zur Verwaltung von Testobjekten notwendig sind. Jede Objektfabrik verweist wiederum auf die von ihr verwalteten Testobjekte. Jeder einzelne Baumknoten speichert dabei die für ihn relevanten Daten, die sich wie folgt zusammensetzen:

- **Wurzel.** Repräsentiert das Gesamtsystem und speichert alle angemeldeten Rechnerknoten. Zusätzlich sind alle Testparameter hinterlegt, die für alle Kindknoten im Baum gelten sollen.

- **Rechnerknoten.** Repräsentiert einen im System angemeldeten Rechner und speichert Informationen, wie dessen IP-Adresse, Name und Informationen über seine Hard- und Software (Prozessortyp, Betriebssystem, usw.). Zusätzlich können Testparameter definiert sein, die nur für Kinder dieses Knotens gelten sollen. In Folge einer solchen Definition werden evtl. in der Wurzel definierte Testparameter überschrieben. Als wichtigste Information speichert er die Referenzen auf die Objektfabriken, die auf dem Rechnerknoten gestartet sind. In der vorliegenden prototypischen Implementierungen sind dies CORBA-Objektreferenzen.

- **Objektfabrik.** Repräsentiert eine auf einem Rechnerknoten gestartete Objektfabrik. Im wesentlichen speichert sie Namen, Typ und Eigentümer von Testobjekten, die sie verwaltet. Zusätzlich können wiederum Testparameter gespeichert werden, die evtl. bereits auf Ebene der Systemwurzel oder des Rechnerknotens definierte Testparameter überschreiben und für alle Kinder des Knotens gelten.

- **Testobjekt.** Repräsentiert ein im System erzeugtes Testobjekt. Es können Testparameter gespeichert werden, die ausschließlich für das jeweilige Testobjekt gelten. Evtl. in den Elternknoten definierte Testparameter werden dadurch überschrieben.

Der Aufbau einer Baumstruktur erfolgt im Koordinator durch korrespondierende Java-Klassen, die in Abbildung B.3 dargestellt sind. Sämtliche gehaltene Daten werden als Attribute in Form von entsprechenden Datenstrukturen in den Klassen definiert. Die Basisklasse `TreeObject` vererbt dabei ihre Eigenschaften an die Klasse `NodeObject` für Rechnerknoten, `FactoryObject` für Objektfabriken und `ControllableObject` für steuerbare Testobjekte. In den Basisklassen sind Funktionen, wie z.B. das Hinzufügen von Baumknoten und ein Iterator zum Durchlaufen aller Kinder eines Baumknotens implementiert. Die Basisklasse implementiert das Interface `Serializable`, um die Datenstruktur abspeichern zu können. Dies kann dazu genutzt werden, um nach einem Ausfall des Koordinators eine Wiederherstellung vorzunehmen.

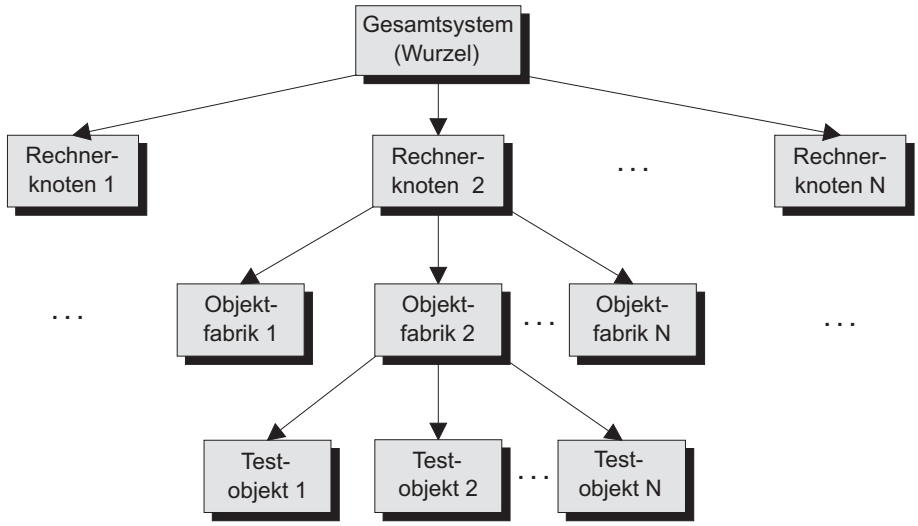


Abbildung B.2: Datenmodell

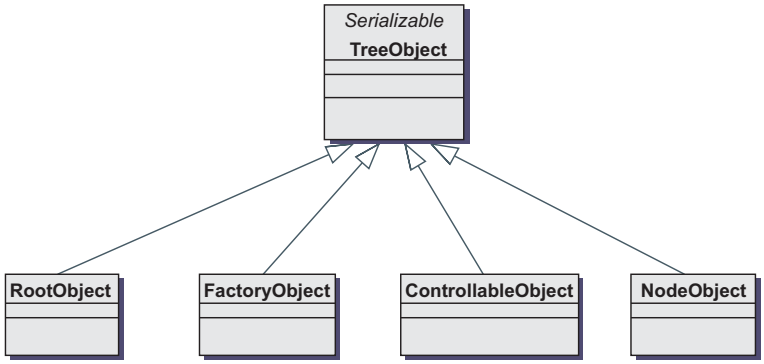


Abbildung B.3: Zentrale Klassen des Datenmodells

B.4.3 Testobjekte

Als Testobjekt sollen hier alle Objekte bezeichnet werden, die Bestandteil einer Testkonfiguration sind und durch das Testwerkzeug gesteuert werden können. Zur Gewährleistung der Steuerbarkeit muß ein Testobjekt die Java-Schnittstelle `Controllable` implementieren. Alternativ kann die abstrakte Klasse `ControllableBase` abgeleitet werden. Dies ist insbesondere für einfache Testobjekte gedacht, die nicht alle Methoden der Schnittstelle `Controllable` benötigen. Abbildung B.4 stellt die beiden Möglichkeiten zur Erstellung von Testobjekten anschaulich dar.
Aufgrund der fehlenden Mehrfachvererbung in Java, ist die Implementierung der Schnittstelle `Controllable` gut geeignet, um bestehende Objekte, die bereits Bestandteil einer Vererbungshierarchie sind, für das Testsystem verfügbar zu machen. Gemäß der Schnittstelle `Controllable` muß ein Testobjekt die folgenden Methoden besitzen:

- **init():** Einmalige Initialisierung des Testobjekts. Wird zu Beginn des Lebenszyklus ei-

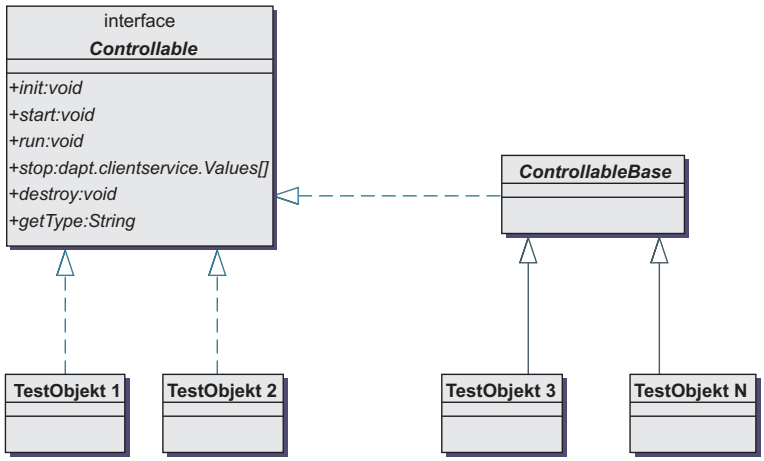


Abbildung B.4: Lebenszyklus eines Testobjekts

nes Testobjekts aufgerufen und kann in einer Client/Server-Anwendung z.B. zum Aufbau einer Server-Verbindung verwendet werden. Im Falle von EJBs entspricht dies der Kontaktaufnahme mit dem Namensdienst des Applikations-Servers (JNDI) zur Beschaffung der Home-Objekte und Erzeugung von Remote-Objekten, der zu kontaktierenden EJBs.

- **run():** Testroutinen, die das Objekt ausführen soll. Diese Methode kann beliebig oft im Lebenszyklus eines Testobjekts aufgerufen werden. Im Falle von EJBs entspricht dies dem Aufruf von Geschäftsmethoden von den in `init()` initialisierten Beans.

- **start():** Startet das Objekt und bringt die in der `run`-Methode enthaltenen Routinen zur Ausführung. Die `start`-Methode kann Parameter entgegennehmen, die zur Testdurchführung notwendig sind.

- **stop():** Stoppt die Testausführung des Objekts. In den Testroutinen aufgenommene Testwerte können zurückgegeben werden. Analog zur `start`-Methode kann die `stop`-Methode ebenfalls mehrfach während des Lebenszyklus eines Testobjekts aufgerufen werden.

- **destroy():** Vernichtung eines Objekts. Wird am Ende des Lebenszyklus eines Testobjekts einmalig aufgerufen und kann in einer Client/Server-Anwendung z.B. zum Abbau einer Verbindung zwischen Client und Server genutzt werden. Im Falle von EJBs wird dem EJB-Container mitgeteilt, daß die vor Beginn des Tests erzeugten Beans nicht mehr benötigt werden.

- **getType():** Erlaubt dem Entwickler eines Testobjekts die Rückgabe einer Typbezeichnung für das Testobjekt, die in der Benutzerschnittstelle als Bezeichnung für das Objekt verwendet wird. Bei der Rückgabe von `null` zeigt das Werkzeug den tatsächlichen, voll qualifizierten Namen der Testklasse an.

Der Lebenszyklus eines Testobjekts ist in Abbildung B.5 anhand eines Zustandsdiagramms anschaulich dargestellt. Nachdem ein Objekt vom Typ `Controllable` mittels der Java-Operati-

on `new()` erzeugt wurde, wird die `init`-Methode zur Initialisierung aufgerufen. Im initialisierten Zustand kann ein Testlauf durch die Methode `start()` ausgelöst werden. Im testenden Zustand wird die `run`-Methode solange ausgeführt, bis die `stop`-Methode den Testlauf beendet und wieder zurück in den Ausgangszustand überführt. Ein Objekt, das sich im initialisierten Zustand befindet, kann durch die Methode `destroy` zerstört werden und kann daraufhin von der automatischen Speicherbereinigung entfernt werden (hier durch eine fiktive `delete()` Methode repräsentiert).

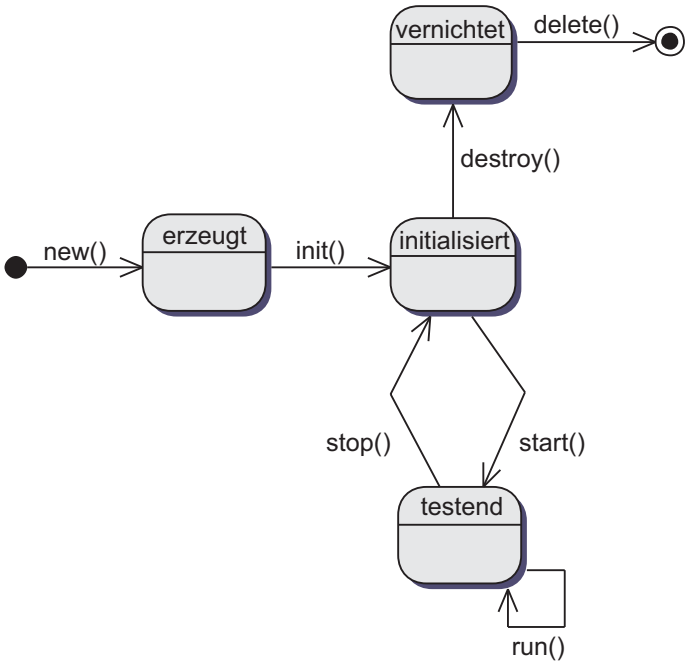


Abbildung B.5: Lebenszyklus eines Testobjekts

B.4.4 Objektfabrik

Die Objektfabrik dient der Erzeugung von Objekten, die im Testsystem zur Ausführung von Tests herangezogen werden können. Die erzeugten Objekte müssen die Schnittstelle `Controllable`, wie im vorigen Abschnitt erläutert, implementieren. Eine wesentliche Eigenschaft der Objektfabrik ist die Fähigkeit, Klassen dynamisch zur Laufzeit zu laden und auch zu entladen. Dies ist erforderlich, um dem Benutzer das Einladen und Ausführen beliebiger Testobjekte zur Laufzeit zu ermöglichen. Dabei kann die Implementierung des Testobjekts geändert werden und erneut zur Ausführung gebracht werden, obwohl die Klasse bereits in die Java-Laufzeitumgebung eingeladen wurde. Weil die virtuelle Maschine keinen Einfluß auf das Entladen von Klassen erlaubt, muß dieses Verhalten durch mehrere Klassenladerobjekte nachgebildet werden. Dabei stellt jedes Klassenladerobjekt einen separaten Namensraum dar und erlaubt das Einladen von beliebigen Klassen, selbst wenn diese Klassen bereits durch einen anderen Klassenlader eingeladen wurden. Dieses Prinzip ist in Abbildung B.6 dargestellt. Die Klasse `TestClass` kann über verschiedene Klassenlader mehrmals in die JVM eingeladen werden.

Es kann sich dabei auch um völlig unterschiedliche Klassen handeln, die nur den gleichen Namen besitzen. Die korrekte Funktionsweise dieser Vorgehensweise kann nur gewährleistet werden, wenn sich die einzuladenden Klassen nicht im Klassenpfad der virtuellen Maschine befinden. So kann verhindert werden, daß der System-Klassenlader die Klassen dauerhaft einlädt. Klassenladerobjekte, die nicht mehr benötigt werden und somit im Code nicht mehr referenziert werden, können durch die automatische Speicherbereinigung (*Garbage Collector*) entfernt werden. Dies hat automatisch die Entfernung, der zuvor in dieses Objekt eingeladenen Klassen zur Folge. Die Klassenladerproblematik wird technisch fundiert in [Hal00a] beschrieben.

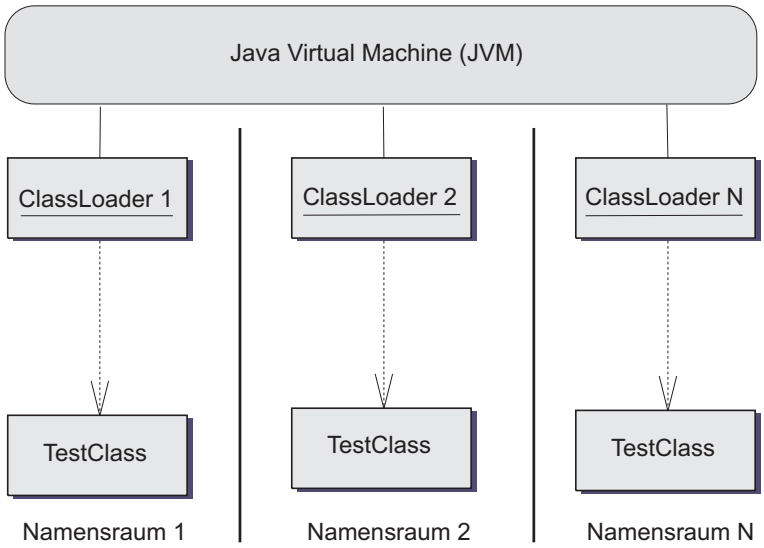


Abbildung B.6: Klassenlader-Prinzip

Durch die Anwendung des soeben beschriebenen Mechanismus in der Objektfabrik ist es möglich den in Abbildung B.7 dargestellten Entwicklungszyklus zu realisieren.

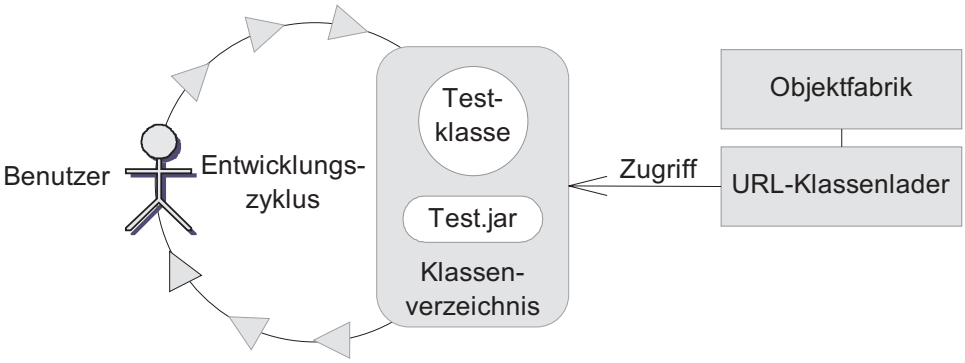


Abbildung B.7: Entwicklungszyklus

Der Benutzer kann eine Testklasse iterativ entwickeln und kann sie beliebig oft im Testsystem verwenden, ohne die Objektfabriken beenden zu müssen. Die Objektfabrik lädt immer

die aktuelle Version der Klasse ein und verwirft die vorhergehende. Dies stellt eine erhebliche Erleichterung des Entwicklungsprozesses von Testobjekten dar. Zur Implementierung des Klassenladerkonzepts kann die Klasse `java.net.URLClassLoader` verwendet werden, die in der Java-Bibliothek vorhanden ist [Mica]. Der URL-Klassenlader ist in der Lage Klassen von mehreren Quellen einzuladen, die mittels *Uniform Resource Locator (URL)* spezifiziert werden können. Dabei können auch Java-Archivdateien als Quelle spezifiziert werden. Somit wird es möglich die Testklasse selbst und alle davon abhängigen Klassen in einem Archiv zusammenzufassen und als Einheit in das Klassenverzeichnis der Objektfabrik zu kopieren. Zu den abhängigen Klassen gehören ebenfalls die Bibliotheken des jeweils verwendeten Applikations-Servers, die notwendig sind, um Test-Clients zu starten. Falls das verwendete Rechnernetz ein gemeinsames Dateisystem verwendet, kann das Klassenverzeichnis zentral bereitgestellt und von allen Objektfabriken im System benutzt werden. Die benötigten Klassen, um einen Test durchzuführen, können dann in dieses Verzeichnis kopiert und bei Bedarf wieder überschrieben werden. Falls kein gemeinsames Dateisystem existiert, besitzt die Objektfabrik eine Methode, die entfernt aufgerufen werden kann, um Dateien im jeweiligen Klassenverzeichnis abzulegen. Somit wird verhindert, daß die benötigten Klassen manuell auf jeden Rechner gebracht werden müssen.
Um zu gewährleisten, daß viele Testobjekte innerhalb einer Factory ausgeführt werden können, wird jedes Objekt innerhalb eines eigenen Threads ausgeführt. Hierzu wird die dafür vorgesehene Klasse `ControllableThread`, die als Umschlag für Testobjekte verwendet wird und die Methoden der Schnittstelle `Controllable` an die eigentliche Implementierung des Testobjekts delegiert. Somit ist die Objektfabrik als Prozeß mit dem Haupt-Thread in der Lage, die Kontrolle über den Lebenszyklus des Objekts zu übernehmen. In Abbildung B.8 ist der Zusammenhang zwischen Prozessen und Threads dargestellt. Das Starten mehrerer Prozesse für Objektfabriken kann auch auf einem Rechnerknoten erfolgen. Dies ermöglicht die bessere Ausnutzung von Ressourcen durch das Betriebssystem und erlaubt z.B. die Berücksichtigung von Mehrprozessorsystemen. Das Starten mehrerer Prozesse kann aber auch aufgrund des verwendeten Applikations-Servers erforderlich sein, wenn ein möglichst realistischer Lasttest erfolgen soll. So verwendet z.B. der Applikations-Server *Bea Weblogic* [Bea] eine Optimierung in seinem proprietären Kommunikationsprotokoll *t3*, die darin besteht, daß Java-Clients, die sich innerhalb der selben JVM befinden, eine Kommunikationsverbindung (*Socket*) teilen [Gom00].

B.4.5 Benutzerschnittstelle

Die Benutzerschnittstelle ist als eigenständige Komponente realisiert und kommuniziert mit dem Koordinator. Dabei werden die im Koordinator gespeicherten Informationen visuell angezeigt. Die realisierte Schnittstelle trägt wesentlich zur Erfüllung der Anforderungen bei, die beim Testen von EJB-Anwendungen erforderlich sind. Dem Benutzer stehen dabei die folgenden Informationen visuell zur Verfügung:

- **Testkonfiguration:** Die teilnehmenden Rechnerknoten, Fabriken und Testobjekte in Form einer Baumstruktur.
- **Hardware/Software-Informationen:** Z.B. Prozessor, Hauptspeicher, Betriebssystem und Java-Version der am Test teilnehmenden Rechnerknoten in Form einer Tabelle.

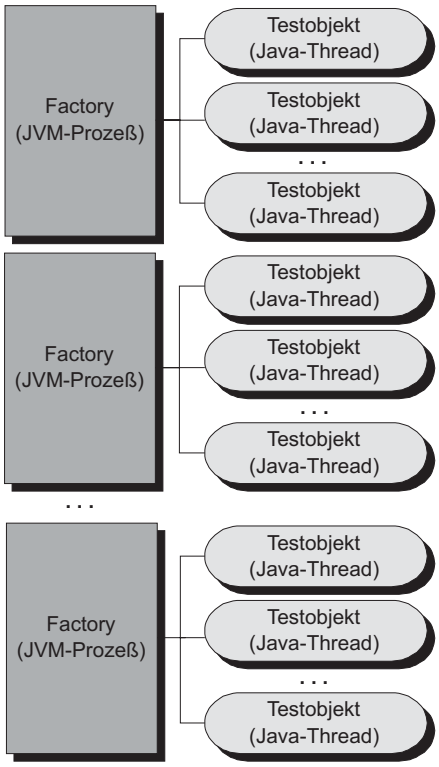


Abbildung B.8: Prozesse und Threads

- **Funktionalität:** In Form von Knöpfen und Kontextmenüs kann der Benutzer des Werkzeugs die angebotene Funktionalität nutzen.

- **Testergebnisse:** In Form einer Tabelle oder eines Schaubilds. Zusätzlich wird eine Zusammenfassung des letzten Testlaufs, die aus der Gesamtzahl der durchgeführten Anfragen, der durchschnittlichen Antwortzeit aller Clients und die durchgeführte Anzahl der Operationen pro Sekunde des getesteten Systems, dargestellt. Somit können wesentliche Aussagen über das Systemverhalten schnell erfaßt werden, ohne eine umfangreiche Testauswertung vorzunehmen. Dies ist insbesondere dann hilfreich, wenn Einstellungen des Applikations-Servers optimiert werden sollen, oder verschiedene Designfragen während der Anwendungsentwicklung oder des Prototypings überprüft werden.

- **Erweiterungen:** Darstellungsfläche für durch den Benutzer realisierte Testsystemerweiterungen des Systems. Die Erweiterungen kann der Benutzer frei gestalten (vgl. dazu Abschnitt B.6). Somit werden alle relevanten Testerfordernisse unter der Oberfläche eines Werkzeugs zentral bereitgestellt, die allen Anwendungsentwicklern zugänglich sind, um sie in ihrer täglichen Arbeit zu unterstützen.

Abbildung B.10 zeigt das Hauptfenster des Werkzeugs. Vom Hauptfenster aus können die in Abbildung B.9 dargestellten Dialoge erreicht werden. Die Dialoge werden nachfolgend kurz erläutert.

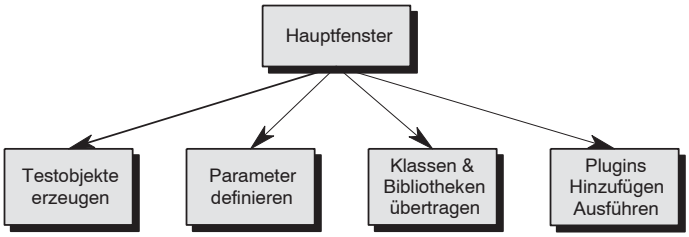


Abbildung B.9: Erreichbare Dialoge

Testobjekte können mittels des in Abbildung B.11 dargestellten Dialogs, der durch das Kontextmenü des Systembaums geöffnet wird, erzeugt werden. Dabei können zu bestehenden Objekten neue hinzugefügt werden oder vorhanden Objekte ersetzt werden. Durch das Hinzufügen von Testobjekten können mehrere unterschiedliche Testobjekte (benutzerabhängig) erzeugt werden. Dies trägt zur Flexibilität der Testfälle bei, da so auch Testfälle zusammengefaßt werden können, die zwar mit unterschiedlichen Testobjekten realisiert sind, aber im Sinne einer realistischeren Testumgebung gemeinsam ausgeführt werden müssen. Im Gegensatz zum Hinzufügen als Thread führt das Hinzufügen als Prozeß zur Erzeugung einer neuen Objektfabrik, die automatisch im System angemeldet wird und die erzeugten Testobjekte innerhalb von Threads ablaufen läßt (vgl. Abschnitt B.4.4). Diese Funktion ist vorgesehen, um Threads auf mehrere Prozesse verteilen zu können und so potentiell auch eine Unterstützung von Mehrprozessorsystemen gewährleisten zu können. Eine Definition von Testobjekten auf der Systemwurzel erzeugt die Objekte systemweit.

Ein zentraler Aspekt bei der Durchführung von Systemtests ist die flexible Parameterübergabe an Testobjekte, die sich im System befinden. In Abbildung B.12 ist der Dialog zur Definition beliebig vieler Parameter für Testobjekte dargestellt. Er kann durch das Kontextmenü des Systembaums erreicht werden und erlaubt die Definition von Parametern in Form von Java-Basistypen, wie `String`, `int` und `float`. Die Granularität der Parametervergabe kann durch ein Vererbungskonzept der Baumelemente frei bestimmt werden und ermöglicht die in Abschnitt B.2 geforderte Flexibilität. Ein Elternknoten vererbt dabei die für ihn deklarierten Parameter an alle seine Kinder, d.h. die Definition von Parametern auf der Systemwurzel sorgt dafür, daß alle angemeldeten und aktiven Knoten die selben Parameter erhalten und diese Parameter an ihre Kinder, also die Objektfabriken, vererben. Die Objektfabriken vererben die Parameter letztlich an die im System erzeugten Testobjekte. In den Kindelementen können vererbte Parameter durch eine Neudefinition von Parametern einfach überschrieben werden. Somit wird es möglich einzelne Knoten, Objektfabriken und Testobjekte mit individuellen Parametern zu versehen und damit eine Simulation der Systemlast durch unterschiedliche Benutzer, die unterschiedliche Anfragen stellen, zu erzeugen.

Falls im verwendeten Rechnernetz kein gemeinsames Dateisystem vorhanden ist, können benötigte Dateien und Bibliotheken mittels des in Abbildung B.13 dargestellten Dialogs zu jeder Objektfabrik übertragen werden. Dies verhindert die mit hohem Aufwand verbundene manuelle Ausstattung jedes Rechnerknotens mit den benötigten Dateien. Bei Verwendung eines gemeinsamen Dateisystems empfiehlt es sich das Testsystem in einem global verfügbaren Verzeichnis zu installieren und die benötigten Dateien einmalig in das dafür vorgesehene Verzeichnis hin-

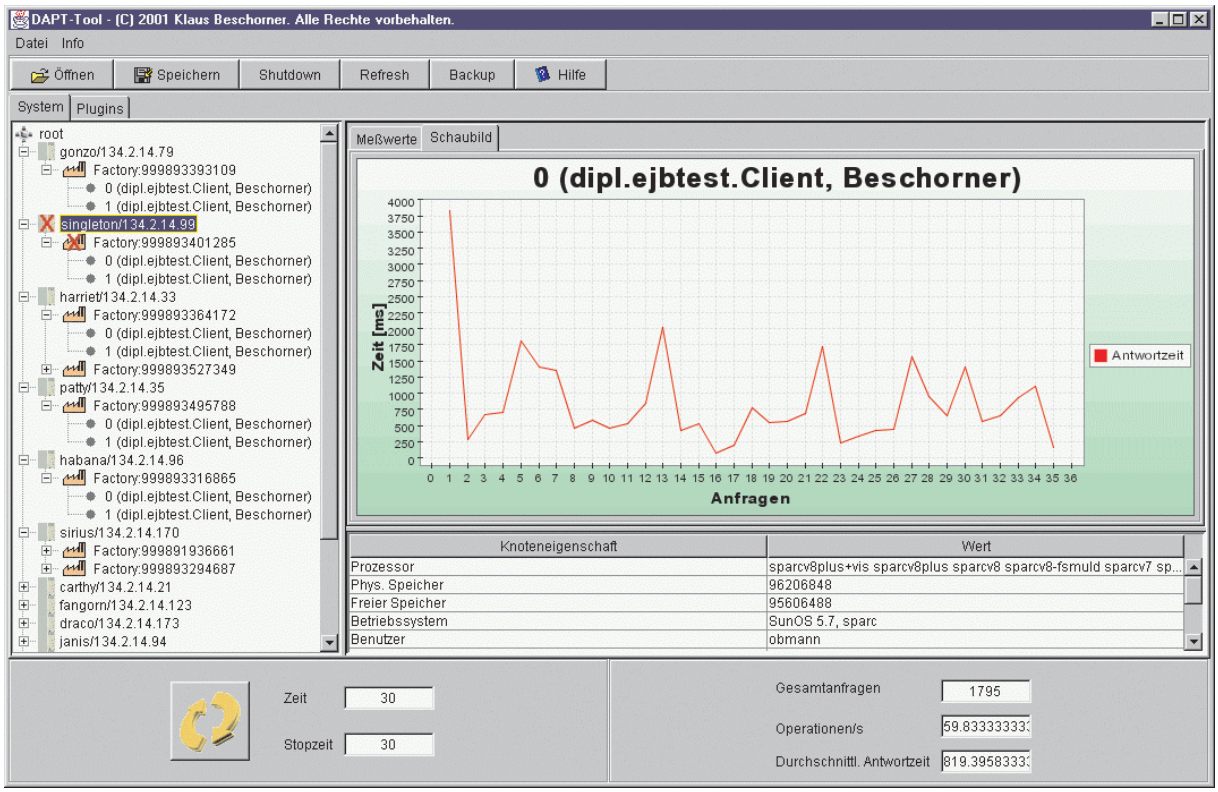


Abbildung B.10: Hauptfenster des Testwerkzeugs

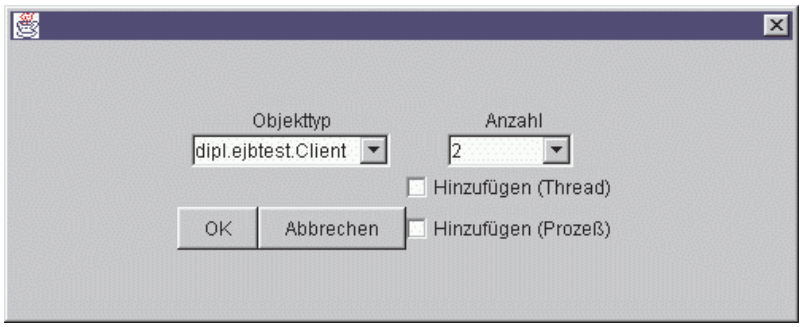


Abbildung B.11: Testobjekterzeugung

einzukopieren (Verzeichnis `classes` im Installationspfad des Testwekzeugs).

Neben der Visualisierung von Systeminformationen und -vorgängen besitzt die Benutzerschnittstelle zusätzlich eine Funktion, um Testergebnisse abzuspeichern. Dabei erfolgt die Abspeicherung in einer strukturierten Form, die anschließend in gängige Tabellenkalkulationsprogramme importiert und weiterverarbeitet werden kann. Weitere Komponenten können der Benutzerschnittstelle durch die in Abschnitt B.6 beschriebene Erweiterungsschnittstelle hinzugefügt werden.

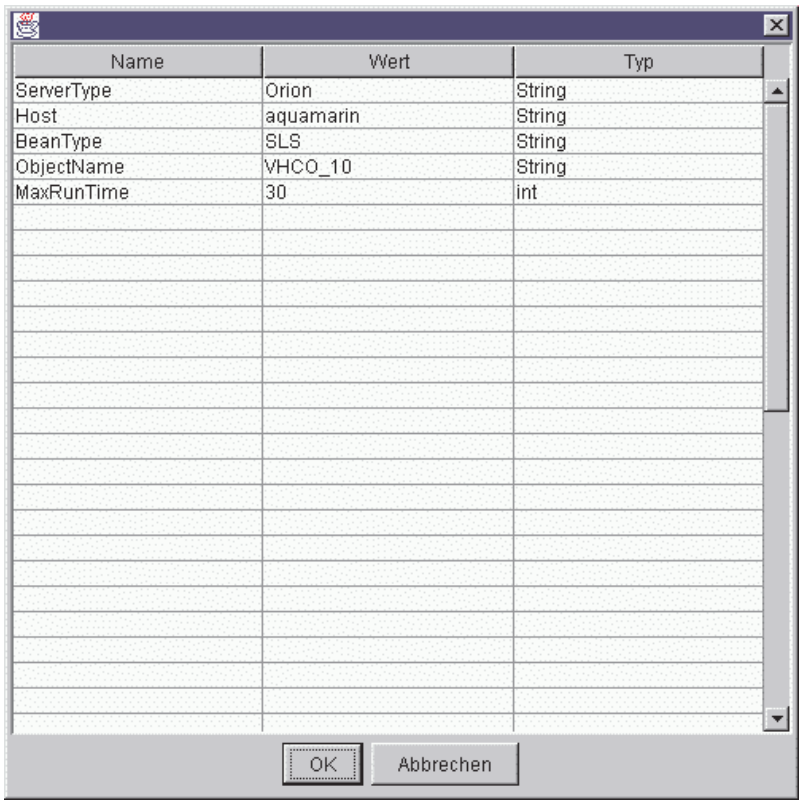


Abbildung B.12: Parameterdefinition

B.5 Kommunikation

Die Kommunikation im gesamten System wird mittels CORBA gelöst. Als ORB wird die im JDK enthaltene Implementierung verwendet. Die Verwendung von CORBA hat gegenüber Java-RMI den Vorteil, daß zusätzlich eine potentielle Programmiersprachenunabhängigkeit vorliegt. Dabei können einzelne Systemteile in anderen Sprachen implementiert werden, wenn gewährleistet ist, daß die zugehörige IDL-Definition der jeweiligen Komponente zugrunde liegt. Dieser Umstand ist besonders für die Factory-Komponente interessant, die für eine andere Plattform implementiert werden kann und Testobjekte starten kann, die in einer anderen Programmiersprache implementiert sind. Die Abbildungen B.14 und B.15 geben einen Überblick über die wichtigsten Datenstrukturen, die zur Kommunikation zwischen den CORBA-Objekten verwendet werden.
Nachfolgend erfolgt eine kurze Erläuterung der verwendeten Strukturen:

- **NodeFactories.** Diese Struktur wird vom Koordinator bereitgestellt, um Informationen über die auf einem Rechnerknoten vorhandenen Objektfabriken zu liefern. Die Struktur speichert dabei eine Folge von **FactoryInfo**-Strukturen, die Informationen über den Namen der Objektfabrik und der verfügbaren Testobjekte gibt. Zur Beschreibung der Testobjekte wird die IDL-Struktur **DObject** verwendet.

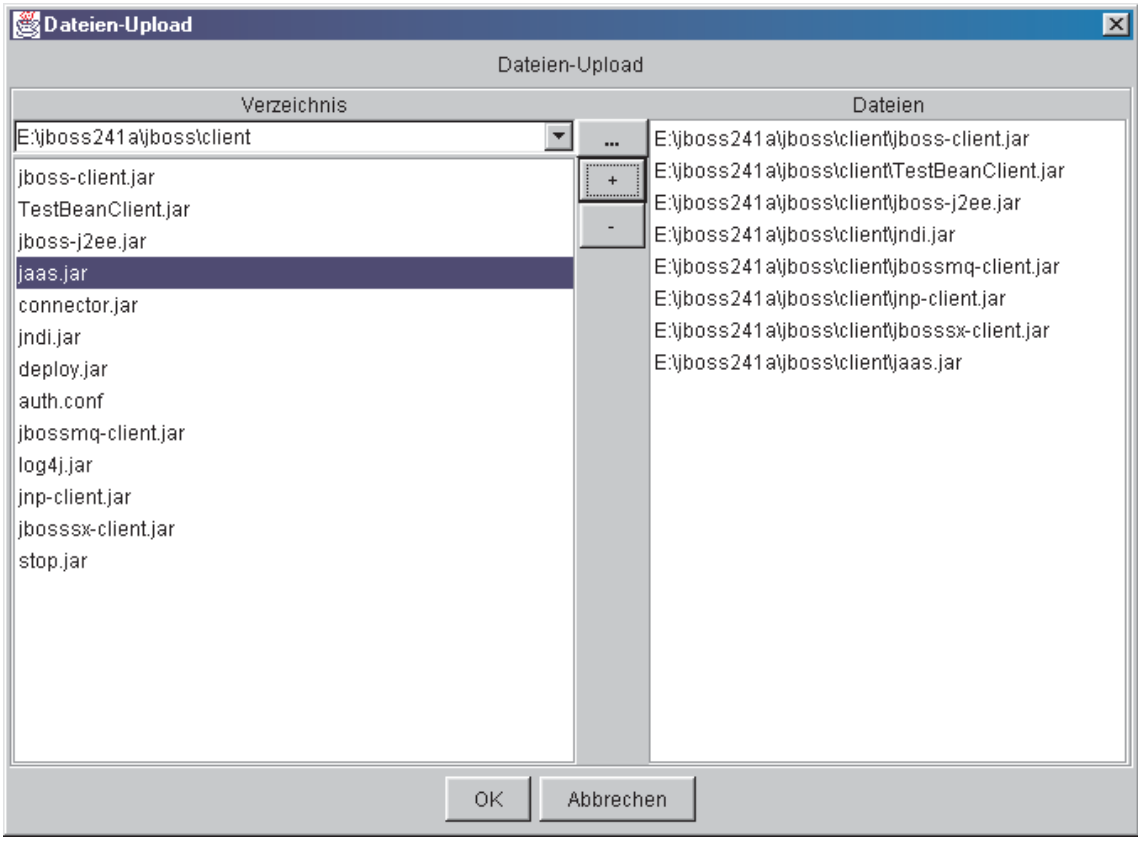


Abbildung B.13: Übertragung von Dateien

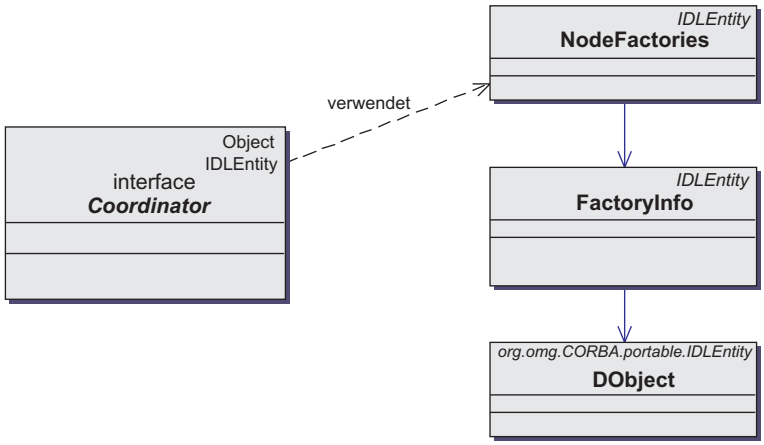


Abbildung B.14: IDL-Datenstrukturen des Coordinators

- **CompilationUnit.** Diese IDL-Struktur dient dem Transport von Testklassen, Testbibliotheken und Bibliotheken des verwendeten Applikations-Servers zu einer Objektfabrik. Falls das verwendete Rechnernetz kein gemeinsames Dateisystem besitzt, können so für

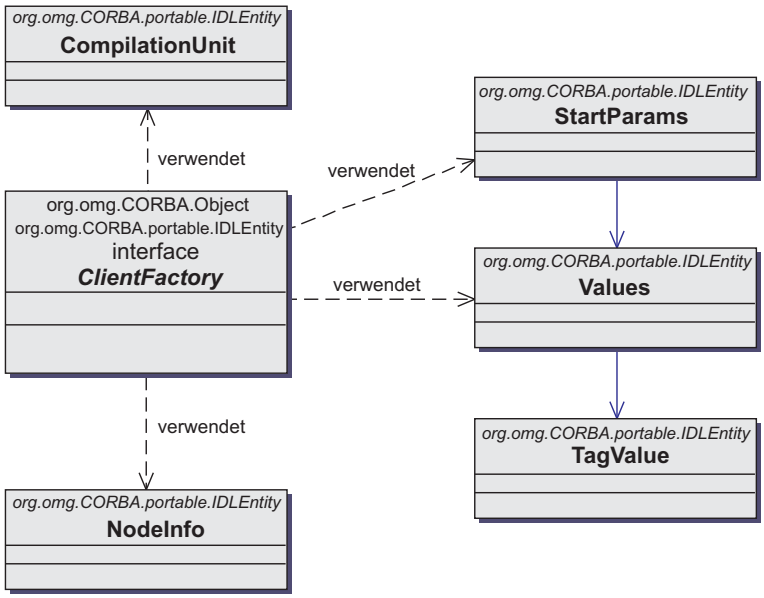


Abbildung B.15: IDL-Datenstrukturen der Objektfabrik

Tests benötigte Klassen zu den Objektfabriken übertragen werden. Hierzu wird eine Folge von Bytes (IDL-Typ `octet`) verwendet, die zur Übertragung auch komprimiert werden kann. Zusätzlich ist die Struktur darauf vorbereitet eine Folge von Kommandos aufzunehmen, die auf dem entfernten Rechnerknoten ausgeführt werden kann. Dies ermöglicht z.B. den Versand von Quellcode, der auf dem entfernten Rechner mit den übergebenen Kommandos zunächst kompiliert wird. Auf die Implementierung dieses Verhaltens wurde jedoch beim vorliegenden Prototyp verzichtet.

- **NodeInfo.** Die Struktur dient zur Sammlung von Daten über die zugrundeliegende Hardware- und Software-Plattform, auf der eine Objektfabrik gestartet ist. Zu den Daten gehört z.B. der vorhandene Prozessor, das verwendete Betriebssystem, der verfügbare Speicher und die verwendete Java-Laufzeitumgebung. Diese Daten werden im Hauptfenster tabellarisch dargestellt (vgl. Abbildung B.10).

- **StartParams.** Die IDL-Struktur dient zur Übergabe von benutzerdefinierten Parametern, die bei der Initialisierung und beim Start von Testobjekten übergeben werden, um das Verhalten von universellen Test-Clients zu modifizieren. Dabei faßt sie Strukturen vom Typ **Values** zusammen, die auch zur Rückgabe von Meßwerten benutzt wird. Kernstück davon ist eine Folge von **TagValue**-Strukturen, die eine Folge beliebiger Schlüssel/Wert-Paare darstellt.

Die Struktur `TagValue` ist besonders hervorzuheben, da sie es ermöglicht beliebige Daten in Form von Schlüssel/Wert-Paaren im Testsystem zu transportieren. Dies erlaubt dem Benutzer des Systems zur Laufzeit beliebige Parameter zu definieren, die an Testobjekte in den Methoden `start()` und `init()` übergeben werden können. Mit der gleichen Struktur werden auch beliebige Meßwerte, die von den Testobjekten aufgenommen werden an die Benutzerschnitt-

stelle zurückgeschickt und dort in Tabellenform dargestellt oder als Textdatei exportiert. Zur Umsetzung dieser Eigenschaft wird eine dynamische CORBA-Datenstruktur verwendet, die nachfolgend skizziert ist:

```
struct  TagValue {
                string tagName;
                any value;
         };

typedef sequence<TagValue> tagValueSeq;
```

Es handelt sich dabei um eine unbeschränkte Folge von Schlüssel/Wert-Paaren (`tagValueSeq`). Jedes Schlüssel/Wert-Paar besteht dabei aus einem Schlüssel, der aus einer Zeichenkette besteht (`string tagName`) und dem eigentlichen Wert, der jedem beliebigen, in IDL darstellbaren Typ, entsprechen kann (`any`). Die Schlüssel/Wert-Paare können vom Benutzer in Tabellenform eingegeben werden. Jede Zeile der Tabelle in Abbildung B.12, Abschnitt B.4.5 ergibt ein Element der Sequenz `tagValueSeq`. Gemäß der Java-Sprachanbindung wird aus der IDL-Struktur eine Java-Klasse `TagValue` generiert die zwei Attribute `tagName` und `value` enthält. Während der IDL-Typ `string` in Java als String-Objekt repräsentiert wird, bleibt der IDL-Typ `any` CORBA-spezifisch und wird als Objekt vom Typ `org.omg.CORBA.Any` repräsentiert. Der Umgang mit diesem Datentyp ist sehr umständlich und würde vom Anwendungsentwickler, der Testobjekte schreibt, CORBA-Kenntnisse erfordern. Um dies zu verhindern verwendet der Anwendungsentwickler eine andere Klasse, die als Dekorator für die Klasse `TagValue` verwendet wird und ihm die Arbeit mit der komplexen CORBA-Datenstruktur abnimmt. Das soeben erläuterte Konzept ist in Abbildung B.16 dargestellt. Die Klasse `Values` bietet dem Benutzer Methoden zum Schreiben (`setXXX`-Methoden) und Lesen (`getXXX`-Methoden) von Schlüssel/Wert-Paaren an. Die Klasse versteckt dabei sämtliche CORBA-Spezifika, indem sie mit dem Datentyp `any` und allen vom IDL-Compiler generierten Klassen für die Struktur `TagValue` arbeitet (unterer Teil der Abbildung). Aus Gründen der Übersichtlichkeit sind die Klassen `Values` und `Any` nicht vollständig dargestellt.

B.6 Erweiterungsschnittstelle

Die Erweiterungsschnittstelle (Abbildung B.17) verfolgt das Ziel, eine Erweiterung des Werkzeugs zuzulassen, um sämtliche Testprozesse in einem Werkzeug zu vereinen und einheitlich verfügbar zu machen. Gleichzeitig können projektspezifische Testverfahren, die normalerweise manuell durch einen Entwickler durchgeführt werden müssen, durch eine automatisierte Erweiterung abgedeckt werden. Für die Realisierung von Erweiterungen ist kein Erlernen einer proprietären Skriptsprache erforderlich. Die Umsetzung erfolgt mittels Java.
Die Erweiterungsschnittstelle ermöglicht die Integration eigener grafischer Benutzeroberflächen und der dazugehörigen Funktionalität in das Testwerkzeug. Dabei kann auch ein Zugriff auf die Funktionalität des Testwerkzeugs selbst erfolgen. Somit wird es der Erweiterung z.B. ermöglicht Testobjekte zu erzeugen, Testvorgänge zu starten und zu beenden. In Abbildung B.17 ist die Umsetzung der Erweiterungsschnittstelle skizziert. Um eine Erweiterung zu implementieren,

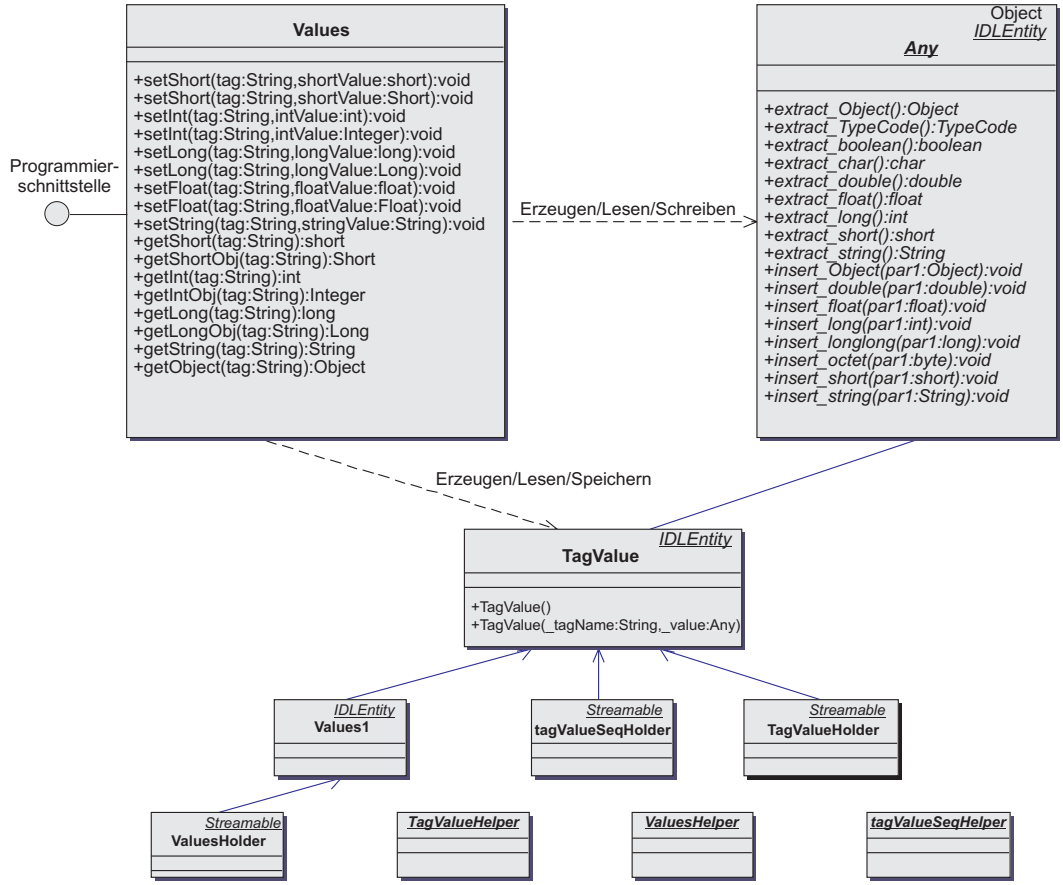


Abbildung B.16: CORBA-Dekorator

muß eine Erweiterungsklasse (hier: `Plugin`) implementiert werden, die von `JPanel`[1] abgeleitet werden muß, um grafisch innerhalb des Werkzeugs dargestellt werden zu können. Als zweite Bedingung muß die Schnittstelle `DaptPlugin` implementiert werden, die mittels der nachfolgend erläuterten Methoden versehen ist:

- **init().** Diese Methode wird beim Einladen der Erweiterung in das Werkzeug einmalig aufgerufen und übergibt dabei ein Objekt vom Typ `Functions`, das die Schnittstelle zu den bereits im Werkzeug implementierten Funktionen bildet. Mit Hilfe dieses Objekts können z.B. Testobjekte im System erzeugt werden und Testprozesse mit den im System aktiven Rechnerknoten durchgeführt werden.

- **start().** Diese Methode wird zum Starten der Erweiterung ausgeführt. Sie kommt bei der manuellen Aktivierung der Erweiterung durch den Benutzer zur Ausführung. Je nach Implementierung der Erweiterung kann diese Methode auch zum Ablauf von implementierter Funktionalität führen, wenn sonst keine anderen grafischen Steuerungsmöglichkeiten

[1] `javax.swingJPanel` dient als Zeichenfläche zur Aufnahme von GUI-Elementen (vgl. dazu [Mica])

auf der Erweiterungs-GUI vorhanden sind. Die `start`-Methode kann beliebig oft aufgerufen werden.

- **stop().** Diese Methode führt zur Deaktivierung der Erweiterung. Sie kommt bei der manuellen Deaktivierung durch den Anwendungsentwickler zur Ausführung. Je nach Implementierung der Erweiterung kann dabei das Beenden von implementierter Funktionalität stattfinden, wenn sonst keine anderen grafischen Steuerungsmöglichkeiten auf der Erweiterungs-GUI vorhanden sind. Die `stop`-Methode kann beliebig oft aufgerufen werden.

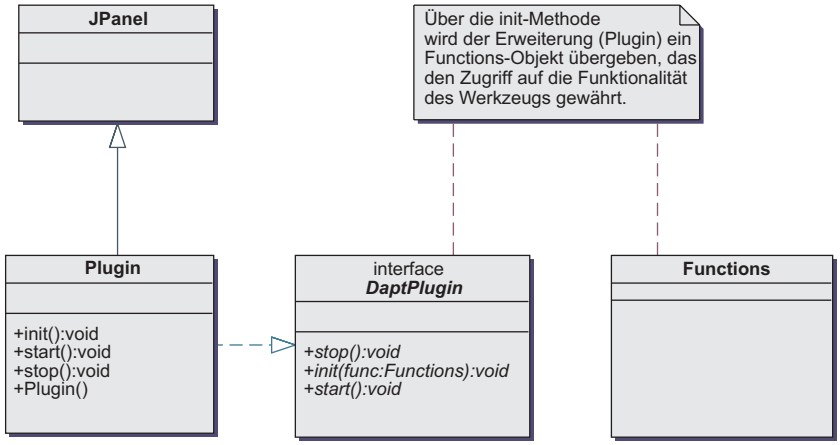


Abbildung B.17: Erweiterungsschnittstelle

B.7 Zusammenfassung

Mit Hilfe des erstellten Testwerkzeugs können Lasttests von Client/Server-Systemen vorgenommen werden. Als Zielplattformen kommen dabei zunächst sämtliche Systeme, die Java unterstützen, in Frage. Durch die CORBA-Konzeption des Testwerkzeugs wird es aber prinzipiell auch möglich das Testsystem im Hinblick auf andere Systeme zu erweitern, für die eine CORBA-Infrastruktur vorhanden ist. Das prototypisch implementierte Testwerkzeug wurde für den Benutzer möglichst einfach gehalten und ermöglicht durch das dynamische Einladen von Testklassen einen iterativen Testzyklus. Die aufgenommenen Testdaten werden durch das Werkzeug auf entscheidungsrelevante Aussagen reduziert und grafisch dargestellt. Zusätzlich können die Daten insgesamt zur weiteren Auswertung exportiert werden. Um das Testwerkzeug mit speziellen Testverfahren zu versehen bzw. dessen Funktionalität zu erweitern, existiert eine Erweiterungsschnittstelle.

Anhang C

CORBA-Datenstrukturen

In Tabelle C.1 sind die im Rahmen der *Interface Definition Language (IDL)* vorhandenen Grunddatentypen [Gro97c] dargestellt.

Typ	IDL-Bezeichner	Beschreibung
Integer	short	$-2^{15}..2^{15}-1$
	long	$-2^{31}..2^{31}-1$
	long long	$-2^{63}..2^{63}-1$
	unsigned short	$0..2^{16}-1$
	unsigned long	$0..2^{32}-1$
	unsigned long long	$0..2^{64}-1$
Floating Point	float	IEEE 754 Single Precision
	double	IEEE 754 Double Precision
	long double	IEEE 754 Double-Extended
Fixed Point	fixed	31 Significant Digits
Character	char	8-Bit Wert
	wchar	Wide Character (implementierungsabhängig)
Boolean	boolean	True oder False
Octet	octet	8-Bit Wert, wird zur Übertragung nicht konvertiert.
Any	any	Enthält Paar aus Beschreibung eines willkürlichen IDL-Typen (TypeCode) und die zugehörigen Daten.

Tabelle C.1: IDL-Grunddatentypen

Zusammengesetzte Typen erlauben die Zusammenfassung von mehreren Grunddatentypen zu einer größeren Datenstruktur. Die dafür zur Verfügung stehenden IDL-Konstrukte sind in Tabelle C.2 dargestellt.
Tabelle C.3 enthält einen Auszug aus der Java-Sprachanbindung. Im Rahmen der Sprachanbindung wird aus IDL-Konstrukten mittels eines IDL-Compilers Java-Code erzeugt. Dabei erfolgt eine Umsetzung der IDL-Konstrukte in äquivalente Java-Konstrukte.

Datenstruktur	Beschreibung
Struktur (`struct`)	Faßt Elemente, bestehend aus Name und Typ, zu einer Einheit zusammen.
Aufzählung (`enum`)	Besteht aus einer geordneten Liste von Bezeichnern.
Union (`union`)	Ähnlich zur Struktur mit dem Unterschied, daß zu einem Zeitpunkt immer nur eines der definierten Elemente enthalten sein kann.
Array (`[ ]`)	Ein- oder mehrdimensionale Folgen fester Länge mit Elementen des selben Typs.
Werttyp (`valuetype`)	Objekt, das per Wert übergeben wird.
Template-Typen	
Sequenz (`sequence`)	Folgen von Elementen, begrenzt oder unbegrenzt.
Strings (`string`)	Alphanumerische Zeichenkette, begrenzt oder unbegrenzt.
Konstanten (`const`)	Konstanter Wert, der nicht geändert werden kann.
Typdefinitionen (`typedef`)	Definition neuer Namen für Grundtypen und zusammengesetzte Typen.

Tabelle C.2: Zusammengesetzte Typen

IDL	Java
boolean	boolean
float	float
sequence, array [ ]	Array des entsprechenden Java-Typs.
string, wstring	java.lang.String
octet	byte
interface	Java-Interface, Helper-Klasse, Holder-Klasse
const	public static final
enum, struct, union, exception	final public class

Tabelle C.3: Java-Sprachanbindung (Auszug)

Literaturverzeichnis

[Ahm01] K. et. al. Ahmed. *Professional Java XML*. Wrox Press, Birmingham, 2001.

[Ale01] S. Alexander. Praxis und Trends bei EJB-Servern. *COMPUTERWOCHE Infonet News*, 23.4., 2001.

[Amb00] S.W. Ambler. *Mapping Objects to Relational Databases*, 2000. http://www.ambysoft.com/mappingObjects.pdf.

[And99] A. Andoh und S. Nash. RMI over IIOP: The new RMI-IIOP standard features easy programming combined with CORBA connections. *JavaWorld*, 12, 1999.

[Apa] Apache Software Foundation. http://www.apache.org/.

[Ava] Avantis GmbH. http://www.avantis.de/.

[Bal98] H. Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum, Akad. Verl., Heidelberg, 1998.

[Bal00] H. Balzert. *Lehrbuch der Software-Technik*. Spektrum, Akad. Verl., Heidelberg, Berlin, 2. Auflage, 2000.

[Bau00] N. Bauer, J. Hauptmann, P. Mandl, und T. Weise. Schaltzentrale: Verteilte Transaktionen auf Basis von CORBA OTS. *iX*, 1, 2000.

[Bea] Bea Systems Inc. http://www.bea.com.

[Bes98a] K. Beschorner. *Realisierung einer Client/Server-Anwendung mit CORBA und Java unter Berücksichtigung bestehender C++-Komponenten*. Diplomarbeit, Universität Tübingen, 1998.

[Bes98b] K. Beschorner und W. Rosenstiel. Realisierung einer Client/Server-Anwendung mit CORBA und Java unter Berücksichtigung bestehender C++-Komponenten. In *JIT'98 Java-Informationstage 1998*, Berlin, 1998. Springer.

[Bes00] K. Beschorner und W. Rosenstiel. Effiziente Datenübertragung in EJB-Systemen. In *Net.ObjectDays2000*, Ilmenau, 2000. Net.ObjectDays-Forum, c/o tranSIT GmbH.

[Bir95] A. Birrer, W.R. Bischofberger, und T. Eggenschwiler. Wiederverwendung durch Framework-Technik - vom Mythos zur Realität. *OBJEKTspektrum*, 5, 1995. Auch: http://www.ubilab.org/publications/print_versions/pdf/objektspektrum95.pdf.

[Bog99] M. Boger. *Java in verteilten Systemen: Nebenläufigkeit, Verteilung, Persistenz*. dpunkt.verlag, Heidelberg, 1999.

[Bor] Borland Software Corp. http://www.borland.com.

[Che99] S. Cheung. *Java Transaction Service (JTS) 1.0*, 1999, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Che01] S. Cheung und V. Matena. *Java Transaction API (JTA) 1.0.1*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Cor98] INPRISE Corporation. *INPRISE Application Server: An Integrated Solution for Developing, Deploying, and Managing Distributed Multi-tier Applications*, 1998. http://www.borland.com.

[Cow01a] T. G. Cowan. Get disconnected with CachedRowSet: The new J2EE RowSet implementation provides updateable disconnected ResultSets in your JSPs. *JavaWorld*, 2, 2001.

[Cow01b] D. Coward. *Java Servlet Specification Version 2.3*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Cow01c] D. Coward. *JavaServer Specification Version 1.2*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Dav99] T. Davis. Direct network traffic of EJBs: Avoid bottlenecking by encapsulating bean properties into a single object. *JavaWorld*, 11, 1999.

[Dee01] A. Deepak, J. Crupi, und D. Malks. *Sun Java Center J2EE Patters. First Public Release: Version 1.0 Beta*, 2001, Sun Java Center. http://developer.java.sun.com.

[DeM01] L.G. et al. DeMichiel. *Enterprise JavaBeans Specification, Version 2.0*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Den00] S. Denninger und I. Peters. *Enterprise JavaBeans*. Addison-Wesley, 2000.

[Deu96a] P. Deutsch. *RFC 1951: DEFLATE Compressed Data Format Specication version 1.3*, 1996, NetworkWorking Group. http://www.faqs.org/rfcs/index.html.

[Deu96b] P. Deutsch. *RFC 1952: GZIP File format specication version 4.3*, 1996, NetworkWorking Group. http://www.faqs.org/rfcs/index.html.

[Deu96c] P. Deutsch und J-L. Gailly. *RFC 1950: ZLIB Compressed Data Format Specification version 3.3*, 1996, NetworkWorking Group. http://www.faqs.org/rfcs/index.html.

[Dow98] B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc, Foster City, 1998.

[Ell01] Ellis, J. et. al. *JDBC 3.0 Specification*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Emm00] W. Emmerich. Software Engineering and Middleware: A Roadmap. In *International Conference on Software Engineering*, New York, 2000. ACM Press. http://www.cs.ucl.ac.uk/staff/W.Emmerich/publications/ICSE2000/SOTAR/.

[Etz95] O. Etzioni und D. S. Weld. *Intelligent Agents on the Internet: Fact, Fiction, and Forecast*, 1995, University of Washington, Seattle, Department of Computer Science and Engineering.

[Evi] Evidian. http://www.evidian.com/jonas/.

[Fis99] J. Fischer. Frage und Antwort: Dynamische Methodenaufrufe mit CORBA. *iX*, 8, 1999.

[Fla98] D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, 2. Auflage, 1998.

[Fle] K. Fleming, J. Aslam-Mir, M. Damstra, und M. Vilicich. Distributed Transactions using CORBA. http://www.expersoft.com.

[Frö01] D. Fröhlich, M. Hubert, und C. Frey. Java Data Objects (JDO) im Überblick. *JAVA SPEKTRUM*, 4, 2001.

[Fus97] M. Fussell. *Foundations of Object Relational Mapping*, 1997, Sunnyvale. http://www.chimu.com.

[Gam95] E. Gamma, R. Helm, R. Johnson, und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gil00] M. Gilpin und C. Zetie. Planning Assumption: 2000 Forecast for the EJB Application Server Market. 2000. http://www.gigaweb.com.

[Göb00] W. Göbl. Der Komponenten-Hype. *OBJEKTspektrum*, 3, 2000.

[Gom00] P. Gomez und Zadrozny. *Professional Java 2 Enterprise Edition with BEA WebLogic Server*. Wrox Press, 2000.

[Gra93] J. Gray und A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Gro97a] Object Management Group. *CORBAservices: Common Object Services Specification*, 1997, Object Management Group. Auch: http://www.omg.org/corba/.

[Gro97b] Object Management Group. *A Discussion of the Object Management Architecture*, 1997, Object Management Group. http://www.omg.org/corba/.

[Gro97c] Object Management Group. *Object Management Group: The Common Object Request Broker: Architecture and Specification, 2.1 ed.*, 1997, Object Management Group. Auch: http://www.omg.org/corba/.

[Gro98] Object Management Group. *Objects By Value*, 1998, Object Management Group. OMG TC Document orbos/98-01-18.

[Gro99] Object Management Group. *Java Language to IDL Mapping Specification*, 1999, Object Management Group. http://www.omg.org/technology/documents/vault.htm.

[Gro00] Object Management Group. *OMG Unified Modeling Language Specification Version 1.3*, 200, Object Management Group. http://www.omg.org.

[Gro01a] S. Große. EJB und Geschäftsanwendungen - Mythos und Realität. *JAVA SPEKTRUM*, 2, 2001.

[Gro01b] Java Data Objects Group. *Java Data Objects Version 1.0*, 2001, Sun Microsystems. http://access1.sun.com/jdo.

[Hal00a] S. Halloway. Tech Tips: Classloaders. *Java Developer Connection*, 31.10., 2000. http://developer.java.sun.com/TechTips.

[Hal00b] S. Halloway. Tech Tips: Serialization in the real world. *Java Developer Connection*, 29.2., 2000. http://developer.java.sun.com/developer/TechTips.

[Hap01] Hapner, M. et al. *Java Message Service*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[IBMa] IBM. http://www-4.ibm.com/software/ts/cics/.

[IBMb] IBM. http://www.transarc.ibm.com/Product/Txseries/Encina/.

[Inc99] Sun Microsystems Inc. und International Business Machines Corporation. *RMI-IIOP Programmer's Guide*, 1999. http://java.sun.com/products/rmi-iiop/index.html.

[Ion] Iona. http://www.iona.com.

[JBo] JBoss Group. http://www.jboss.org.

[Jen98] N. R. Jennings, K. Sycara, und M. Wooldridge. A Roadmap of Agent Research and Development. In *Autonomous Agents and Multi-Agent Systems*, Boston, 1998. Kluwer Academic Publishers.

[JT] JUnit-Testframework. http://www.junit.org.

[Kas00] N. Kassem. *Designing Enterprise Applications*. Sun Microsystems, Inc., Palo Alto, 2000.

[Küh01] M. Kühn und C. Och. Über Objekte und Relationen. *JAVA SPEKTRUM*, 6, 2001.

[Lar00] C. Larman. Enterprise JavaBeans 201: The Aggregate Entity Pattern. *Software Development Magazine*, 4, 2000.

[Man98] C. Mangione. Performance test show Java as fast as C++. *JavaWorld*, 2, 1998.

[McC98] G. McCluskey. Using Java Reflection. *Java Developer Connection*, 1998.

[McM97] C. McManis. Take an in-depth look at the Java Reflection API: Learn about the new Java 1.1 tools for finding out information about classes. *JavaWorld*, 9, 1997.

[MH99] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates Inc., 1999.

[Mica] Sun Microsystems. http://java.sun.com/docs/index.html.

[Micb] Sun Microsystems. http://java.sun.com/products/jdk/idl/index.html.

[Mic99a] Sun Microsystems. *Java Naming and Directory Interface: Application Programming Interface (JNDI API) 1.2*, 1999, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Mic99b] Sun Microsystems. *Java Naming and Directory Interface: Service Provider Interface (JNDI SPI) 1.2*, 1999, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Mic99c] Sun Microsystems. *Java Object Serialization Specification*, 1999, Sun Microsystems, Inc.

[Mic99d] Sun Microsystems. *Java Remote Method Invocation Specification*, 1999, Sun Microsystems, Inc.

[Mic00] Sun Microsystems. *JavaMail API Design Specification Version 1.2*, 2000, Sun Microsystems. http://java.sun.com/products/javamail/index.html.

[Mic01] Sun Microsystems. *The Java Tutorial: A practical guide for programmers*. Sun Microsystems, 2001.

[Mor01] R. et. al. Mordani. *Java API for XML Processing*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Mow97] T. J. Mowbray und R. Malveau. *CORBA Design Patterns*. Wiley Computer Publishing, 1997.

[Mye00] T. et. al. Myers. *Professional Java Server Programming J2EE Edition*. Wrox Press, 2000.

[Nes99] C. Nester, M. Philippsen, und B. Haumacher. Effizientes RMI für Java. In *JIT'99 Java-Informationstage 1999*, 1999.

[Neu99] O. Neumann, C. Pohl, und K. Franze. Caching in Stubs und Events mit Enterprise Java Beans bei Einsatz einer objektorientierten Datenbank. In *JIT'99 Java-Informationstage 1999*, 1999.

[Nol97] G. Nolan. Decompile Once, Run Anywhere: Protecting your Java source. *Web Techniques Magazine*, 2(9), 1997. http://www.webtechniques.com.

[Oes98] B. Oestereich. *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified medeling language*. Oldenbourg Verlag, 4. Auflage, 1998.

[Orf98] R. Orfali und D. Harkey. *Client/Server Programming with JAVA and CORBA*. Wiley Computer Publishing, 2. Auflage, 1998.

[PKW96] PKWARE. *Info-ZIP Application Note 970311*, 1996, PKWARE.

[Red96] J.-P. Redlich. *Corba 2.0, Praktische Einführung für C++ und Java*. Addison-Wesley, 1996.

[Res00] M. Res. Reduce EJB network traffic with astral clones. *JavaWorld*, 12, 2000.

[Rom99] E. Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. Wiley Computer Publishing, 1999.

[Rom00] J. Rommel. Java is here to stay: Why Java is ready for enterprise applications. *JavaWorld*, 1, 2000.

[Rom02] E. Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. Wiley Computer Publishing, 2. Auflage, 2002.

[Ruh00] W. Ruh, T. Herron, und P. Klinker. *IIOP Complete: Understanding CORBA and Middleware Interoperability*. Addison Wesley, 2000.

[Sch97] D. C. Schmidt und S. Vinoski. Object Interconnections, Object Adapters: Concept and Terminology (Column 11). *SIGS C++ Report*, 9(11), 1997. Auch: http://www.cs.wustl.edu/ schmidt/report-doc.html.

[Ses99] G. Seshadri und G. S. Raj. *Enterprise Java Computing: Applications and Architectures*. Cambridge University Press, 1999.

[Ses00] G. Seshadri. *Fundamentals of RMI: Short Course*, 2000, Java Developer Connection. http://developer.java.sun.com/developer/onlineTraining/rmi/index.html.

[Sha98] Y.-P. Shan und R. H. Earle. *Enterprise Computing with objects*. Wiley Computer Publishing, 1998.

[Sha01a] B. Shannon. *Java 2 Platform Enterprise Edition Specification, v1.3*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Sha01b] R. Sharma. *Java 2 Enterprise Edition: J2EE Connector Architecture Specification 1.0*, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.

[Sin97] S.K. Singhal, B. Q. Nguyen, R. Redpath, M. Fraenkel, und J. Nguyen. *Building High-Performance Applications and Servers in Java: An Experimental Study*, 1997, IBM. http://www-106.ibm.com/developerworks/library/javahipr/javahipr.html.

[Sta00a] M. Stal. Episode 3: CORBA 3, Teil 1: Komponenten. *iX*, 4, 2000.

[Sta00b] M. Stal. Reich der Mitte. Die Komponententechnologien COM+, EJB und „CORBA Components“. *OBJEKTspektrum*, 3, 2000.

[Sta01] B. Staudacher und R. Pichler. CORBA3.0-Komponentenmodell: Anwendung und Besonderheiten. *OBJEKTspektrum*, 1, 2001.

[Suc01] A. Sucharitakul. Seven Rules for Optimizing Entity Beans. *JAVA DEVELOPER CONNECTION*, 5, 2001.

[Sun98] T. Sundsted. Zip your data and improve the performance of your network-based applications. *JavaWorld*, 11, 1998.

[Tea00] The Advanced Application Architecture Team. *iCommerce Design Issues and Solutions*, 2000, Beaverton, Oregon. http://www.javasuccess.com.

[The95] The Standish Group. *Chaos*, 1995. http://standishgroup.com/visitor/chaos.htm.

[Tho] Thought Inc. http://www.thoughtinc.com/index.html.

[Til99] S. Tilkov. Ganz oder gar nicht: JTS: der Java Transaction Service. *iX*, 8, 1999.

[Ver] Versant Corp. http://www.versant.com.

[Vin97] S. Vinoski. CORBA: Integrating Diverse Applications Within Heterogenous Environments. *IEEE Communications Magazine*, 14(2), 1997. Auch: http://www.cs.wustl.edu/ schmidt/corba-papers.html.

[Vog] A. Vogel. *The Inprise Application Server: Building Enterprise Applications for the Net with EJB, CORBA, and XML*. http://www.borland.com.

[Vog97] A. Vogel und K. Duddy. *Java Programming with CORBA*. Wiley Computer Publishing, New York, 1997.

[Vog98] A. Vogel und K. Duddy. *Java programming with CORBA: Advanced Techniques for Building Distributed Applications*. Addison Wesley, 2. Auflage, 1998.

[Vog99] A. Vogel und R. Madhavan. *Programming with Enterprise JavaBeans, JTS, and OTS: Building Distributed Transactions with Java and C++*. Wiley Computer Publishing, 1999.

[Web] WebGain Inc. http://www.webgain.com/products/toplink/.

[Wey99] C. Weyer. Microsoft COM+: in Zukunft alles anders? *OBJEKTspektrum*, 3, 1999.

[Wil00] J. Wilson. Get smart with proxies and RMI. *JavaWorld*, 11, 2000.

[Wol01] V. Wolf. Java: Vom Daumenkino zur Unternehmensanwendung - eine Bestandsaufnahme. *IBM Software eNews-Magazin*, 7, 2001. www.ibm.com/software/de/eNews.