Informatik Forsch. Entw. (2003) 18:♣–♣
Digital Object Identifier (DOI) 10.1007/s00450-003-0134-4

Informatik
Forschung und Entwicklung

Untersuchungen zur effizienten Kommunikation in EJB-Systemen

Klaus Beschorner, Wolfgang Rosenstiel, Wilhelm G. Spruth

Universität Tübingen, Arbeitsbereich Technische Informatik, Sand 13, 72762 Tübingen
(e-mail: {beschorner, rosenstiel, spruth}@informatik.uni-tuebingen.de)

Eingegangen am 7. Oktober 2002 / Angenommen am 27. Juni 2003

Zusammenfassung. Enterprise JavaBeans (EJB) ermöglichen die Erstellung von leistungsfähigen, mehrschichtigen Client/Server-Anwendungen auf Basis der Programmiersprache Java. Eine grundlegende und wichtige Entscheidung, die häufig unterschätzt wird, ist die Art und Weise, wie Daten zwischen komplexen Java-Clients (Fat-Clients) und Servern transportiert werden sollen. Während der Implementierung kann ein fehlendes Konzept zur Übertragung von Daten dazu führen, daß Entwickler unterschiedliche Verfahren wählen, die verschiedenen Anforderungen an die Anwendung, wie z.B. ein möglichst gutes Leistungsverhalten, widersprechen und evtl. unter hohem Aufwand rückgängig zu machen sind. Verschiedene Übertragungsverfahren erschweren außerdem die Erweiterung und Wartung des Systems, da zu analysieren ist, wie in verschiedenen Fällen die Datenübertragung gelöst wird. In diesem Beitrag werden deshalb neue Vorgehensweisen vorgestellt, um Daten zwischen einer objektorientierten Applikationsschicht, die sich auf dem Server befindet, und den Clients zu übertragen. Aktive Daten-Container (ADCs) stellen einen einheitlichen Transportmechanismus bereit, der zusätzliche Funktionalität besitzt, um die Datenübertragung zu optimieren. Dazu gehört z.B. der automatische Datenaustausch mit Geschäftsobjekten, um das manuelle Beschreiben und Auslesen des Daten-Containers durch den Anwendungsentwickler einzusparen. Zusätzlich wird die Optimierung des Datenübertragungsvorgangs im Sinne des Leistungsverhaltens angestrebt. ADCs sind sehr flexibel und können in der Anwendungsentwicklung als universelles und zentral zur Verfügung gestelltes Datenübertragungskonzept dienen. Eine Untersuchung des Leistungsverhaltens zeigt, daß mit den vorliegenden Konzepten der Transaktionsdurchsatz hinsichtlich der Kommunikation bis zum Faktor 8 gesteigert werden kann. Im Rahmen eines Industrieprojekts konnte mit Hilfe der Konzepte Entwicklungsaufwand im Umfang von ca. 18% der erforderlichen Codezeilen einer komplexen Anwendung eingespart werden.

Schlüsselwörter: Enterprise JavaBeans, Datenübertragungskonzepte, Kommunikationskosten, Entwicklungskosten

Abstract. Enterprise JavaBeans (EJB) are server-side components to build powerful multi-tier client/server applications with Java. A frequently underestimated but important question is the method used to transfer data between Java clients (fat clients) and the server. A missing concept to solve that problem will animate developers to find their own solutions depending on their special problems and their knowledge. This will lead to a system with many different data transfer solutions, which are not consistent with the requirements of the whole application, for example a high performance of the data transfer. As a consequence of that, changes and enhancements of the application can be difficult and expensive to accomplish. Adressing these problems, this article will present new approaches to transfer data between an object-oriented application tier on the server and the clients. Active Data Containers (ADC) constitute a uniform transport mechanism which uses additional functionality to optimize the data transfer. This includes, inter alia, an automatic data exchange with business objects to unburden the developer from manual read and write operations. Additionally, a performance optimization of the data transfer is intended. ADCs are very flexible and can be provided in application development centrally as a universal data transport mechanism. An analysis of the performance shows that the active concepts can improve the transaction throughput by up to a factor of 8. The application of the concept in an industrial project showed an overall reduction of the lines of code in a complex system by approx. 18%.

Keywords: Enterprise JavaBeans, data transfer concepts, communication costs, development costs

CR Subject Classification: C.2.4, D.2.11, D.1.5

1 Einführung

Enterprise JavaBeans [6] stellen einen Entwicklungsrahmen für mehrschichtige Client/Server-Anwendungen zur Verfügung, der die Entwickler von vielen Aufgaben entlastet. Jedoch müssen grundsätzliche Fragen, die für den kompletten Lebenszyklus einer Anwendung relevant sind, vom Entwickler bedacht werden. Hierzu gehört auch die Frage, wie Daten zwischen den Clients und der Applikationsschicht, die Geschäftsobjekte der jeweiligen Problemdomäne enthält,

transportiert werden sollen. An die Datenübertragungsart werden in der Praxis oft die folgenden Anforderungen geknüpft:

- *Wiederverwendung.* Ein allgemeingültiges Konzept kann in ein Framework zur Wiederverwendung integriert werden.
- *Einheitlichkeit.* Wenn immer dasselbe Konzept verwendet wird, führt dies zu einem leichter wartbaren und erweiterbaren System.
- *Benutzbarkeit.* Ein einfaches, möglichst transparentes Konzept spart Entwicklungszeit und kann auch von Entwicklern, die keine Client/Server-Spezialisten sind, eingesetzt werden.
- *Leistungsverhalten.* Die Lösung soll möglichst schnell sein und wenig Ressourcen verbrauchen.
- *Zusatznutzen.* Möglichst viele Probleme, die mit der Datenübertragung zusammenhängen bzw. direkt an diese anknüpfen, sollten berücksichtigt werden.

Typischerweise bergen die Anforderungen ein hohes Konfliktpotential. Eine Datenübertragungskonzept sollte früh in der Implementierung festgelegt werden, um eine Vielfalt von unterschiedlichen Übertragungsmechanismen zu verhindern, die bei späteren Anforderungsänderungen unter hohem Aufwand angepaßt werden müssen.

Im folgenden Abschnitt werden zunächst bestehende Konzepte zur Datenübertragung erläutert. Im Anschluß daran werden neue, flexible Konzepte vorgestellt, die im Rahmen eines Industrieprojekts eingesetzt wurden und neben einer Reduktion des Entwicklungsaufwands auch zu einer Verbesserung des Leistungsverhaltens der Anwendung führen können. Die Konzepte sind als mögliche Muster zur Erstellung von Daten-Containern zu verstehen. Die konkrete Implementierung der Container und zugehörigen Werkzeugen kann an die jeweilige Anwendungsdomäne und eingesetzten Entwicklungswerkzeuge angepaßt werden.

2 Stand der Technik

Im EJB-Umfeld basiert die Kommunikation auf dem Java-Serialisierungsmechanismus, der die Übertragung von Datenstrukturen zwischen Client und Server ermöglicht. Der Mechanismus wird z.B. in [15,8,16] beschrieben.

Bestehende Datenübertragungskonzepte lassen sich gemäß Abb. 1 in statische und dynamische Ansätze einteilen. Statische Ansätze beruhen auf der Verwendung von serialisierbaren Transportobjekten, die Daten in Form ihrer Attribute deklarieren. Zur Übersetzungszeit steht fest, welche Datentypen transportiert werden. Somit erfolgt eine strenge Typprüfung von Zugriffen auf solche Datenstrukturen durch den Java-Compiler. Dieser Ansatz entspricht der direkten Verwendung von serialisierbaren Geschäftsobjekten oder dem Einsatz des Entwurfsmusters *Value Objects*. Dynamische Ansätze verwenden serialisierbare Objekte, die beliebige Daten vom Basistyp `Object` zur Laufzeit aufnehmen. Diese Eigenschaften besitzen alle `Collection`-Objekte, die Bestandteil der Java-Bibliothek sind. Als Transportobjekt kommen sie in Frage, solange die übergebenen Objekte selbst serialisierbar sind. Aufgrund der beschriebenen Eigenschaften können Zugriffe auf diese Datenstruktur vom Compiler nicht zur Übersetzungszeit überprüft werden.

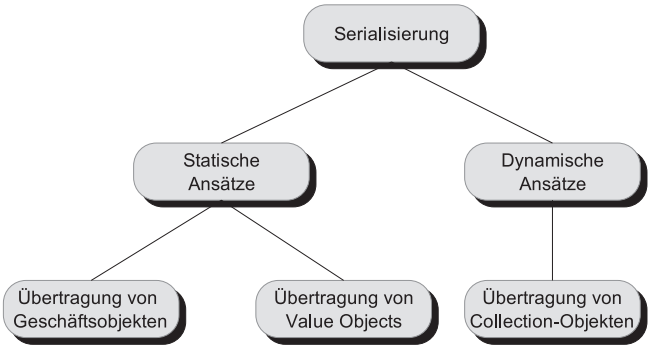


Abb. 1. Bestehende Datenübertragungskonzepte

Nachfolgend werden die gängigen statischen und dynamischen Datenübertragungsverfahren beschrieben. Dabei wird zunächst auf die Struktur einer EJB-Anwendung eingegangen, da diese bei der Konzeption eines Datenübertragungsmechanismus zu berücksichtigen ist.

2.1 Anwendungsarchitekturen

Bei der Erstellung eines EJB-Systems werden in der Analyse identifizierte Anwendungsfälle und Geschäftsobjekte in eine Struktur, die aus EJB-Komponenten und Java-Objekten besteht, umgesetzt. In der Literatur finden sich zahlreiche Entwurfs- und Implementierungshinweise. Empfehlenswert sind [10,19,11,23]. Das am häufigsten in der EJB-Literatur reflektierte Architekturkonzept beruht auf der Session-Fassade [5, 23], wobei Session-Beans auf Entity-Beans zugreifen, um Geschäftsprozesse auszuführen. Auf die Verwendung von Entity-Beans kann auch völlig verzichtet werden, indem sie durch persistente Java-Objekte ersetzt werden. Zur Realisierung der Persistenz können darauf spezialisierte Frameworks für relationale oder objektorientierte Datenbanken verwendet werden.

Trotz der Vielfalt möglicher Anwendungsstrukturen muß festgelegt werden, wie die verschiedenen Komponenten untereinander und mit ihren Anwendungs-Clients kommunizieren. Dabei müssen die Daten aus den Anwendungsstrukturen zwischen Server und Client transportiert werden.

2.2 Datenübertragungskonzepte

2.2.1 Serialisierung von Geschäftsobjekten

Die Java-Serialisierung überführt primitive Datentypen, Objekte und ganze Objektgraphen in eine Folge von Bytes, die über das Netzwerk verschickt werden können. Dieser Mechanismus kann zur Übertragung des Zustands von Geschäftsobjekten, die sich auf dem Server befinden, verwendet werden. Sämtliche Geschäftsklassen implementieren hierzu das Interface `java.io.Serializable`.

Dieses Konzept wird in der Literatur unterschiedlich zur Lösung der Datenübertragung vorgeschlagen. In den Publikationen [14,11,10] wird die Datenübertragung eng an Design- und Architekturfragen gekoppelt. Dabei sollen serialisierbare Geschäftsobjekte, die von der Existenz einer Entity-Bean

abhängig sind und selbst kaum Logik enthalten, direkt übertragen werden. In [25] wird darauf verwiesen, daß die Geschäftsobjekte selbst serialisiert werden sollten, da dies mit dem geringsten Implementierungsaufwand verbunden ist. In [22] wird die Serialisierung direkt auf Instanzen von Entity-Beans angewendet, um sie zum Client zu übertragen. Die serialisierten Instanzen werden als „Astrale Klone\ bezeichnet.

Der Hauptvorteil des vorliegenden Konzepts besteht im geringen Implementierungsaufwand, da es Bestandteil der Java-Bibliothek ist und auch Objektgraphen berücksichtigt. Dabei werden jedoch zu viele Daten übertragen, falls der Client pro Anwendungsfall nur eine Teilmenge der Attribute benötigt. Die Modifikation der Serialisierung kann dies verhindern, verursacht allerdings zusätzlichen Implementierungsaufwand. Der Hauptnachteil besteht jedoch in der Verletzung des Entwurfsziels mehrschichtiger Architekturen, Geschäftsobjekte und -prozesse sowie Algorithmen in einer Schicht zur besseren Wiederverwendung, Änderung und Erweiterung zu kapseln.

2.2.2 Value Objects

Das Entwurfsmuster (*Value Objects*) ist weit verbreitet und wird in den J2EE-Blueprints [10] und dem J2EE-Entwurfsmusterkatalog [5] beschrieben. Dabei werden für Geschäftsobjekte serialisierbare Daten-Container-Klassen implementiert, die alle Geschäftsattribute zum Transport enthalten. In der Literatur gibt es zahlreiche Konzepte, die weitgehend den *Value Objects* entsprechen. Dazu gehören *Detailobjekte* [19], *Properties-Objekte* [4] und *Domain Object State Holder* [25, 11].

Der Vorteil dieses Verfahrens liegt in der Verwendung spezieller Transportobjekte, die eine direkte Nutzung der Server-Objektstrukturen durch den Client verhindern. Ein signifikanter Nachteil ist jedoch der zusätzliche Implementierungs- und Wartungsaufwand für die Container-Klassen. Transportobjekte müssen manuell beschrieben und ausgelesen werden, wobei auch eine Überführung der zu transportierenden Daten zwischen den Datenstrukturen des Servers und des Transportobjekts erfolgen muß. Ändert ein Client seine Anforderungen an die vom Server bezogenen Daten, muß eine Änderung der Transportobjekte erfolgen. Kommt ein neuer Client mit spezifischen Anforderungen hinzu, muß mindestens ein neues Transportobjekt implementiert und in der Schnittstelle berücksichtigt werden. Falls ein Java-Applet implementiert wird, kann die Anzahl der vorhandenen Transportklassen problematisch sein, da diese alle zum Client übertragen werden müssen. Die Ressourcenproblematik besteht ebenfalls, wenn der Platzbedarf für eine Java-Client-Anwendung möglichst gering sein soll.

2.2.3 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 `Collection`- und `Map`-Klassen vorhanden, deren Objekte das beschriebene Verhalten ermöglichen. `HashSet`, `HashMap` und `Vector` sind typische Beispiele für die genannten Datenstrukturen [15]. Sie implementieren das Interface `Serializable` und können damit zur Kommunikation in EJB-Systemen verwendet werden [1,2].

Neben den `Collection`- und `Map`-Objekten existiert eine weitere Klasse der `ResultSet`-Objekte, die bei der direkten Verwendung von SQL-Anfragen mittels JDBC auftreten. Diese Objekte sind nicht serialisierbar und können nicht direkt zum Client übertragen werden. Der Typ `CachedRowSet` stellt eine serialisierbare Alternative dar, um tabellarische Daten direkt zum Client zu schicken [3].

Die dynamischen Konzepte sind nicht explizit für die Datenübertragung vorgesehen. Dies erschwert das Auffinden von Programmteilen, die sich mit der Datenübertragung befassen. Zusätzlich besitzen sie 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.

2.2.4 Verwandte Arbeiten

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 versucht, die Datenübertragung zu optimieren, indem z.B. ein clientseitiger Cache etabliert wird. Der Kommunikations-*Stub* wird durch ein weiteres Objekt gekapselt, wobei nicht jeder Aufruf einen Zugriff auf den Server bewirkt. Dieser Ansatz wird in CORBA- [26], RMI- [27] und EJB-Anwendungen [21] verwendet.
- *Intelligente Agenten:* Ein Ziel des Agenten-Paradigmas ist es, mobile Agenten, die sich reaktiv und proaktiv in ihrer Umgebung verhalten, Daten filtern zu lassen, 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 [9] befindet sich ein guter Gesamtüberblick des Forschungsgebiets. Internet-bezogene Anwendungen werden z.B. in [7] beschrieben.

3 Bewertung

3.1 Implementierungsaspekte

Grundsätzlich muß sich jeder Entwickler um die Form der Datenübertragung kümmern und für den Datenaustausch zwischen Anwendungsdatenstrukturen und den Daten-Containern sorgen. Daraus resultiert ein hoher Entwicklungsaufwand, der in einem großen System häufig zu vielen unterschiedlichen Lösungsansätzen führt, die auf den Konzepten aus Abschnitt 2.2 beruhen. Im Rahmen von nachträglichen Änderungen oder Optimierungen müssen die gewählten Konzepte umfangreich überarbeitet werden.

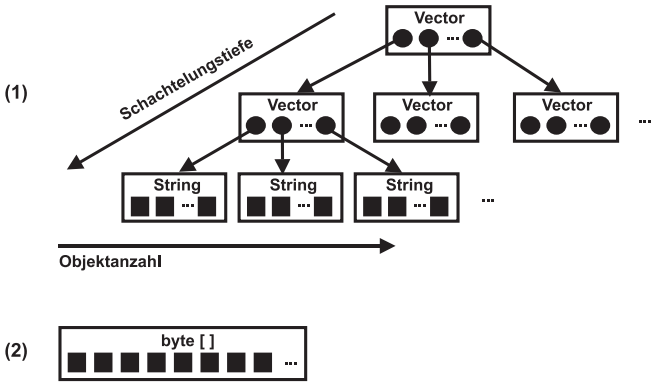


Abb. 2. Datenstrukturen

3.2 Leistungsaspekte

Bestehende Datenübertragungsmechanismen folgen ausschließlich objektorientierten Entwurfskonzepten und führen damit in komplexen Anwendungen zu aufwendigen, ineinander geschachtelten Datenstrukturen. Dies führt jedoch zu Leistungseinbußen bei der Datenübertragung, da die komplexeren Datenstrukturen hohe Ansprüche an die Kommunikationsschicht des verwendeten Applikations-Servers stellen. Abb. 2 verdeutlicht den Unterschied zwischen strukturierten und unstrukturierten Daten anhand eines Beispiels.

Die strukturierten Daten bestehen aus einem Vektor, der eine Folge von Vektoren enthält, die wiederum jeweils eine Folge von `String`-Objekten enthalten. Dagegen sind unstrukturierte Daten, hier in Form eines Byte-Arrays dargestellt, weitaus weniger komplex. In Abb. 3 sind die Übertragungszeiten der Datenstrukturen aus Abb. 2 mit verschiedenen Applikations-Servern dargestellt[1]

Beide Datenstrukturen wurden so belegt, daß die Datenmenge nach der Serialisierung gleich ist. Die Übertragung unstrukturierter Daten ist nahezu unabhängig vom verwendeten Applikations-Server . Die Übertragung unstrukturierter Daten stellt nur geringe Anforderungen an die Kommunikationsschicht der Server und liefert nahezu gleiche Übertragungszeiten, da nur geringe Anforderungen an die Kommunikationsschichten gestellt werden. Dagegen werden bei der Übertragung strukturierter Daten signifikante Unterschiede zwischen den Servern deutlich. So verzeichnen die Server S1, S4 und S7 massive Einbrüche der Übertragungszeit. Mit dem Server S1 war es aufgrund einer Fehlermeldung grundsätzlich nicht möglich Daten im Umfang von 1000 KByte zu übertragen.

Grundsätzlich ist bei allen Servern ersichtlich, daß die Übertragung strukturierter Daten zu höheren Übertragungszeiten führt, da die Kommunikationsschicht mehr leisten muß. Dies liegt in den folgenden Ursachen begründet:

- *Datentypisierung.* Schwach typisierte Daten erfordern vor dem Versand die Ermittlung des genauen Typs in der Kommunikationsschicht des Servers. Dies ist z.B. beim Typ *Object* und Objektsammlungen vom Typ `Collection` notwendig.
- *Java-Objekterzeugung.* Das Erzeugen von Java-Objekten ist allgemein eine teure Operation [24]. Komplexe Datenstrukturen sind aus vielen Java-Objekten zusammengesetzt, die nach einem Kommunikationsvorgang rekonstruiert, d.h. neu erzeugt, werden müssen.
- *Serialisierung.* Zur Kommunikation verwendete *Stubs* und *Skeletons* überführen Daten in ein zum Transport geeignetes Format. Komplexe Datenstrukturen nehmen dafür mehr Zeit in Anspruch als weniger komplexe. Komplexe Datenstrukturen verursachen i.d.R. eine größere Datenmenge, da Metainformationen für die Rekonstruktion nach dem Kommunikationsvorgang benötigt werden. Die Kommunikation mit RMI und RMI-IIOP basiert auf dem Java-Serialisierungsprotokoll, das selbst nicht effizient ist [20].
- *Kommunikationsprotokoll.* Optimierte Kommunikationsprotokolle können komplex strukturierte Daten i.d.R. besser übertragen. So kann z.B. der Applikations-Server S5 alternativ zum gewöhnlichen RMI-Protokoll mit einem proprietären Protokoll betrieben werden. Die Übertragung von 1000 KByte Daten niedriger und hoher Komplexität ist dabei um durchschnittlich 34% schneller.

4 Universelle Datenübertragungskonzepte

4.1 Überblick

Die in Abschnitt 2.2 beschriebenen Datenübertragungskonzepte werden nun gemäß Abb. 5 durch neue Konzepte ersetzt bzw. erweitert. Hierzu wird zusätzlich zwischen aktiven und passiven Datenübertragungskonzepten unterschieden [1,2]. Passive Konzepte beschränken sich auf die bloße Datenspeicherung, während aktive Konzepte eng mit der Datenübertragung verbundene Aufgaben mit berücksichtigen, auf die übertragenen Daten einwirken oder eigenständig Aktivitäten auslösen. Aktive Daten-Container (ADCs) werden insgesamt so gestaltet, daß sie zu aktiven Elementen einer Anwendung werden, die flexibel an wechselnde Anforderungen anpaßbar sind und zur Systemoptimierung beitragen.

4.2 Aktiver Daten-Container

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, das von den Anwendungsentwicklern eingehalten werden muß.

[1] Server: Pentium III, 800 MHz, 256 MB Hauptspeicher, Windows 2000, Java 1.3.1 HotSpotClient VM; Client: Pentium II/266 MHz, 256 MB Hauptspeicher, Windows NT 4.0, Java 1.3.1 HotSpotClient VM; Netzwerk: 10 MBit/s). Die Server wurden anonymisiert, da diese Arbeit keinen Vergleich von Applikations-Servern, sondern von Datenübertragungskonzepten anstrebt. Verwendet wurden Open-Source-Produkte, wie z.B. *JBoss* (www.jboss.org) und *JOnAS* (www.evidian.com) sowie kommerzielle Produkte, wie z.B. *Orion* (www. orionserver.com) und *iPortal* (www.iona.com).

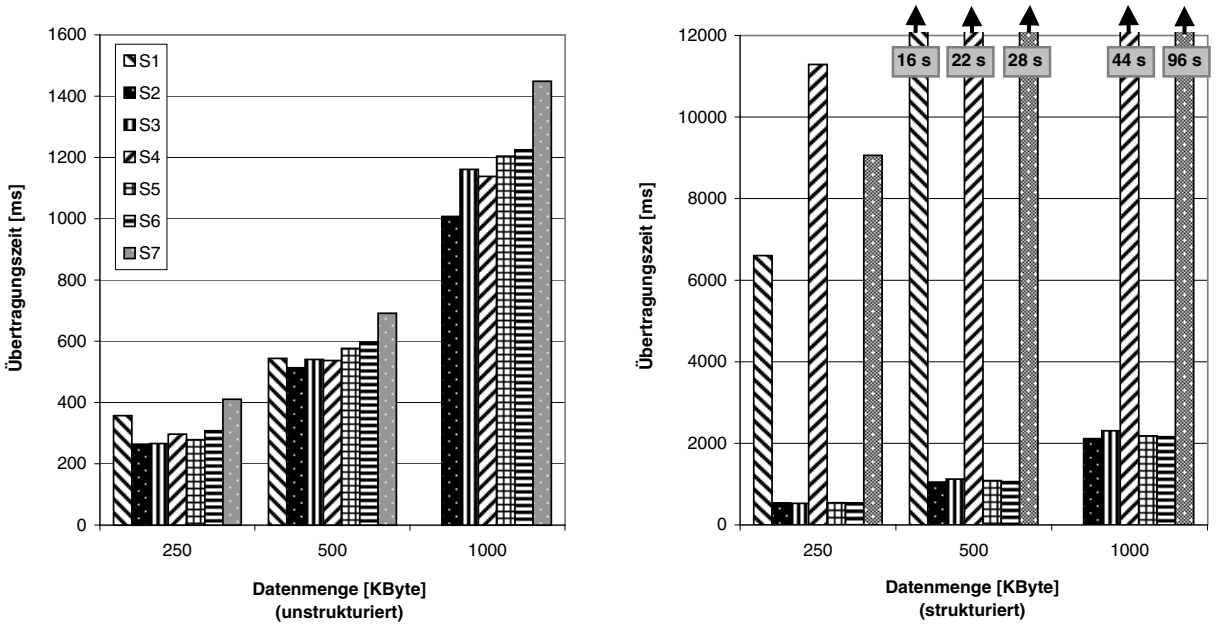


Abb. 3. Datenübertragung mit unterschiedlichen Applikations-Servern

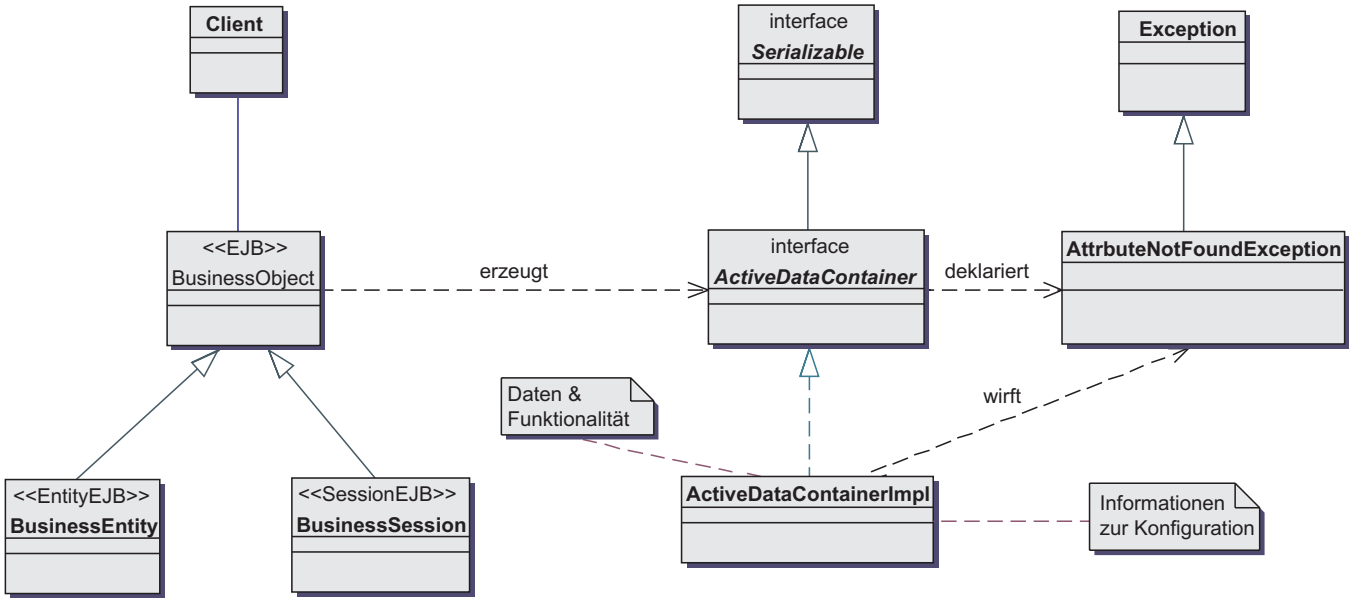


Abb. 4. Prinzip des Aktiven Daten-Containers

- *Kosten.* Die Container-Funktionalität wird zentral bereitgestellt, um die Wartung und Erweiterungen des Daten-Containers transparent für die Anwendungsentwickler durchzuführen. Datenaustauschoperationen zwischen Datenquellen und Daten-Containern 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.
- *Leistung.* Die Attributmenge kann zur Übertragung auf die vom Anwendungsfall abhängige Menge begrenzt werden. Dies entspricht einer Umsetzung des Entwurfsmusters *Dynamische Attribute* [18], das automatisch von jedem Entwickler verwendet wird.
  Die strukturierte Datenübertragung aus mehreren Datenquellen wird ermöglicht. Dies verhindert, daß mehrere entfernte Methodenaufrufe getätigt werden, um verschiedenartige Daten zu transportieren.
  Es kann Einfluß auf die Datenstruktur genommen werden, um einen schnelleren Transport durch die Kommunikationsschicht des verwendeten Applikations-Servers zu erzielen.
- *Kopplung.* 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.
- *Flexibilität.* Dynamische Datenübertragung ermöglicht jederzeit die Übertragung zusätzlicher Daten. Daten wer-

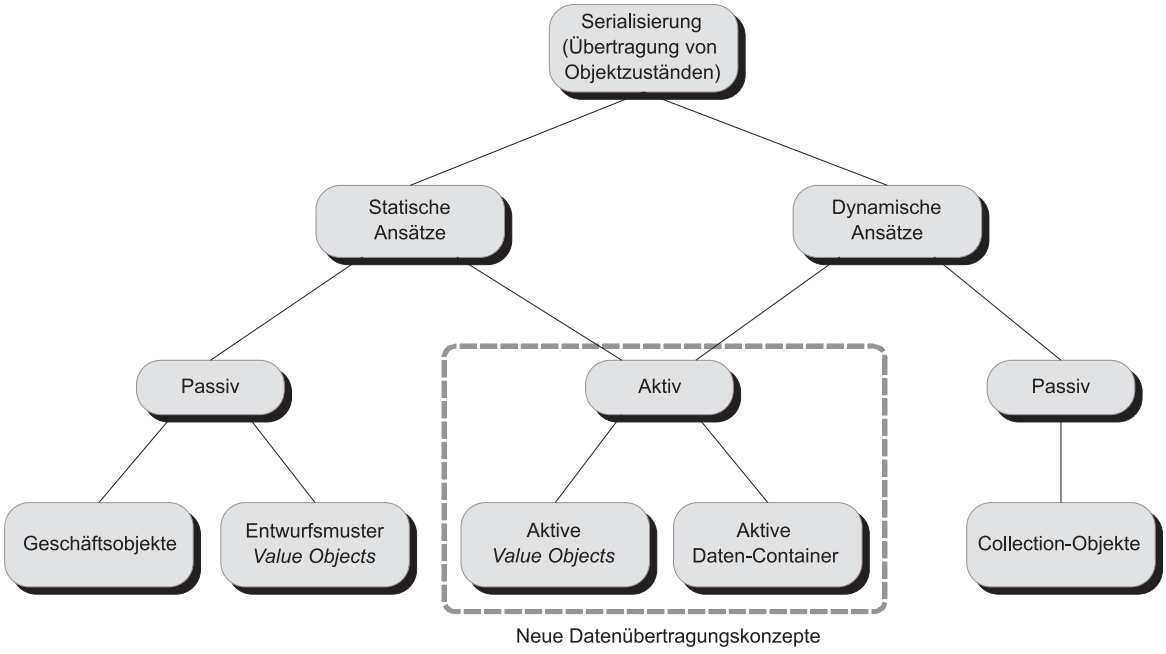


Abb. 5. Neue Datenübertragungskonzepte

den unabhängig von der Form im Daten-Container verpackt. Zusatzfunktionalität kommt im Daten-Container zur Ausführung und kann auch systemweit transparent für die Anwendungsentwickler aktiviert und angepaßt werden.
- *Ressourcen.* Im Minimalfall ist nur eine Transportklasse notwendig, die zum Client übertragen werden muß.
- *Sicherheit.* 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.

4.2.1 Struktur

Der ADC führt die Aufgaben der Datenübertragung in einer Klasse bzw. in einer Schnittstelle zusammen und bietet zusätzlich die Möglichkeit, weitere Operationen auf den transportierten Daten auszuführen. Eine signifikante Entlastung des Entwicklers erfolgt dadurch, daß dem ADC lediglich die zu übertragenden Daten übergeben werden müssen. Der Container kann die Daten sämtlicher Objekte aufnehmen, deren Struktur ihm bekannt sind. In Abb. 4 ist die Grundstruktur eines ADCs dargestellt. Im Zentrum des Konzepts steht die Schnittstelle `ActiveDataContainer`, die alle Methoden des Daten-Containers festlegt. Sie ist von `Serializable` abgeleitet, um zu implementierende Objekte zwischen Client und Server übertragen zu können. Die Ausnahme `AttributeNotFoundException` wird erzeugt, falls ein angefordertes Attribut nicht im Daten-Container enthalten ist. `ActiveContainerImpl` stellt die Implementierung des Daten-Containers bereit. Dazu gehören eine geeignete Datenspeicherung und Zusatzfunktionalität, die auf die Daten im Container angewendet werden oder das Verhalten des Containers selbst bestimmen. Die *Informationen zur Konfiguration* bestimmen zur Übersetzungs- oder Laufzeit, welche Funktionalität verwendet wird. 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.

Der Daten-Container besitzt Methoden zur Übergabe von Java-Objekten, deren Daten in der internen Datenstruktur gespeichert werden. Durch Angabe einer Liste von gewünschten Attributnamen durch den Anwendungsentwickler kann eine Einschränkung auf wirklich benötigte Attribute im jeweiligen Anwendungsfall erfolgen. 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. Die Methoden zur Konfiguration von Funktionalität und Verhalten legen die Konfiguration dynamisch zur Laufzeit oder statisch im Quellcode fest. Ein Beispiel für Zusatzfunktionalität ist die Datenkompression. Zur Verarbeitung der im Container transportierten Daten existieren Methoden zum Lesen, Schreiben und zur Existenzprüfung bestimmter Attribute. Zusätzlich sind Iteratoren vorhanden, um alle transportierten Attribute zu durchlaufen.

Aufgrund der Standardschnittstelle wird es möglich, allgemeingültige Komponenten zu implementieren, die sämtliche als ADCs repräsentierte Daten verarbeiten können. Dies eröffnet z.B. die Möglichkeit, eine Komponente auf der Seite des Clients zu implementieren, die automatisch Daten zwischen ADCs und GUI-Elementen austauscht. Dabei wird der Entwicklungsaufwand reduziert, da die ADCs zusätzlich als Datenmodell auf dem Client verwendet und Datenaustauschoperationen automatisiert werden.

4.3 Optimierung

4.3.1 Datenstruktur

Die Datenstruktur zur Speicherung des Zustands von übergebenen Objekten bzw. Transportdaten richtet sich nach der Beschaffenheit der Daten selbst, aber auch nach den Eigenschaften der Kommunikationsschicht des verwendeten Applikations-Servers. Zusätzlich erfolgt eine Optimierung auf Platzbedarf und Zugriffsgeschwindigkeit. Als Basisspeicherstruktur bietet sich die Klasse `HashMap` [15] an. Die Attributnamen der übergebenen Objekte werden dabei als Schlüssel und die Attributausprägungen als Werte verwendet.

Mögliche Implementierungen des Konzepts werden nachfolgend kurz erläutert:

1. Die Daten von Objekten, deren Attribute transportiert werden sollen, werden jeweils in einem ADC-Objekt abgelegt und in einem Objekt vom Typ Vektor gespeichert. Diese Datenstruktur erfordert einen geringen Implementierungsaufwand und bietet die höchste Flexibilität, da die ADC-Objekte unterschiedlich konfiguriert sein können. Dies führt jedoch wegen der ineinandergeschachtelten Objekte und der redundanten Speicherung von Attributen zu einer hohen Datenmenge und einer hohen Komplexität der Datenstruktur.
2. Die Speicherung der Daten als Objekte wird aufgehoben, um nur noch die tatsächlichen Nutzdaten effizient in Arrays vom Typ `Object` zu speichern. Zur späteren Rekonstruktion des eigentlichen Objekts werden dessen Typbezeichnung und seine Attributnamen einmal gespeichert. Falls Daten abgerufen werden, wird aus diesen Daten ein ADC-Objekt (oder ein beliebiges anderes Objekt) rekonstruiert.
3. Eine weitere Reduktion der Datenmenge und Komplexität kann durch die Verwendung einer reinen String-Repräsentation erreicht werden. Diese Form schränkt die Flexibilität des Containers allerdings ein, da nicht mehr automatisch beliebige Objekttypen unterstützt werden. Es müssen Methoden vorhanden sein, die zwischen Objekt- und String-Repräsentation konvertieren können. Bei primitiven Java-Datentypen sind diese Funktionen bereits vorhanden. Die Datenmenge wird zusätzlich gesenkt, wenn die String-Repräsentation eines Datentyps kleiner ist als der Datentyp selbst.

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.

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.

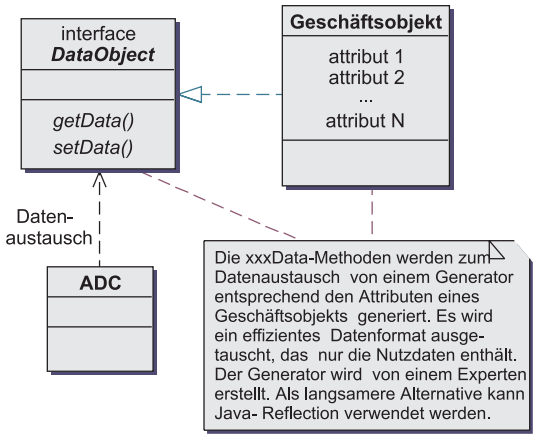


Abb. 6. Optimierung des Datenaustauschs

4.3.2 Datenaustausch

Der automatische Datenaustausch mit den Objekten, deren Daten im Container transportiert werden sollen, stellt eine Grundfunktionalität dar. 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 Reflection. 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 [17, 13, 12] beschrieben. Dies erfordert jedoch einen höheren Zeitbedarf als der herkömmliche Aufruf von fest kodierten Zugriffsmethoden. Darüber hinaus ist eine Konvention darüber erforderlich, 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, die direkt im Daten-Container gespeichert wird. Dieser Ansatz ist in Abb. 6 skizziert. Gegenüber der Verwendung von Java-Reflection bietet dieser Ansatz ein wesentlich besseres Leistungsverhalten (vgl. Abschnitt 5.2).

4.3.3 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. 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.

4.4 Aktive Value Objects

Aktive Value Objects (AVOs) stellen eine statische Umsetzung der Konzepte aus Abschnitt 4.2 dar. Es werden nun Transportobjekte verwendet, die zu übertragende Datentypen zur Übersetzungszeit festlegen. Dies hat zur Folge, daß für jeden Datenübertragungsfall, der in einer Anwendung existiert, eine passende Daten-Container-Klasse implementiert werden muß. Außerdem wird nun ein Sammel-Container zur Übertragung von mehreren Transportobjekten vom selben Typ oder unterschiedlichen Typen benötigt.

Mit dem AVO-Konzept wird eine Beschränkung auf ein Transportobjekt pro Geschäftsobjekt möglich, da die Attributmenge individuell pro Anwendungsfall zur Laufzeit festgelegt werden kann. Durch eine Standardschnittstelle wird generisches Lesen, Schreiben und das Iterieren über Attribute möglich, was von verallgemeinerten Komponenten genutzt werden kann. Zusätzlich wird eine (nachträgliche) Konfiguration von Zusatzfunktionalität, wie z.B. Kompressionsalgorithmen, verfügbar, die auf Transportobjektebene eingesetzt werden kann, aber insbesondere auch bei der Übertragung von mehreren Transportobjekten in einem Sammelbehälter. Ebenso kann sich die Optimierung der Datenübertragung auf einzelne Attribute eines Transportobjekts, auf das gesamte Transportobjekt oder auf mehrere Transportobjekte durch den Sammel-Container beziehen. Eine mögliche Umsetzung des Konzepts ist in Abb. 7 skizziert. Geschäftsobjekte, deren Daten transportiert werden sollen, besitzen eine Schnittstelle zum automatischen Datenaustausch mit dem zugehörigen AVO-Typ, der alle Attribute des Geschäftsobjekts definiert. Jedes AVO besitzt Methoden zum Lesen und Schreiben seiner Attribute in einen Java-Stream zur Ein- und Ausgabe von Daten.

Dabei kann ein selektives Lesen und Schreiben von Attributen erfolgen, um eine Einschränkung der Daten auf den vorliegenden Anwendungsfall zu ermöglichen. Dies ist vor allem bei Geschäftsobjekten mit sehr vielen Attributen notwendig. Zusätzlich können in diesen Methoden weitere Optimierungsmaßnahmen, die z.B. zu einer Verkleinerung der Datenmenge führen, auf Attributebene durchgeführt werden. Die Methoden sind Bestandteil der Standardschnittstelle, die jedes AVO besitzen muß. Zur Übertragung einer Folge von AVOs wird ein AVO-Container (AVOC) verwendet, der die zu übertragenden Daten direkt in sein primitives, internes Datenformat (Byte-Array) übernimmt. Dies stellt eine Optimierung dar, die verhindert, daß die Kommunikationsschicht des vorliegenden Applikations-Servers die zu übertragende Datenstruktur suboptimal verarbeitet. Bei primitiven Datenstrukturen verhalten sich die verschiedenen Applikations-Server nahezu gleich (vgl. Abschnitt 3.2). Falls erforderlich, kann

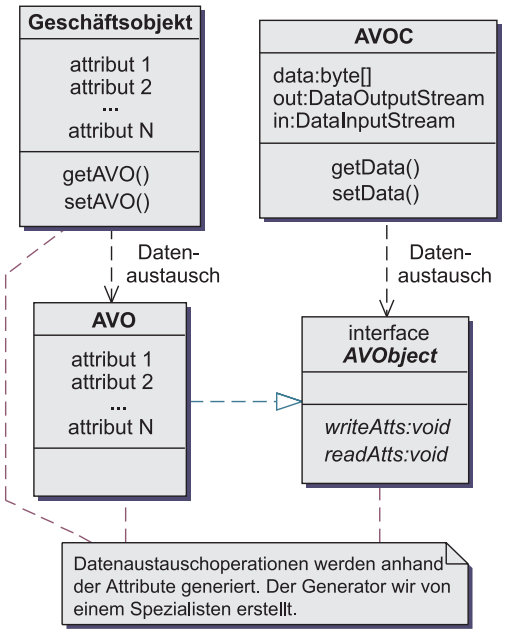


Abb. 7. Umsetzung von Aktiven Value Objects

der AVOC eine weitere Optimierung über die Gesamtheit der vorliegenden Daten hinweg vornehmen. Die Umsetzung des automatischen Datenaustauschs und der verschiedenen Optimierungsmaßnahmen wird durch einen Spezialisten gelöst. Um das beste Leistungsverhalten zu gewährleisten und den Entwicklungsaufwand zu reduzieren, wird auf einen Generatoransatz zurückgegriffen, der die passenden Methoden anhand der Definition der Attributnamen und Attributtypen generiert. Das Generatorkonzept kann entweder auf bestehende Klassen angewendet werden oder Bestandteil eines Entwicklungsprozesses sein, der Java-Rahmengerüste für Geschäftsklassen und zugehörige AVO-Klassen generiert. Hierzu können z.B. aus einer XML-Definition der Attribute (Name und Datentyp) eines Geschäftsobjekts mittels XSL-Stylesheet und XSL-Prozessor in Java-Quellen umgesetzt werden. Alternativ kann ein UML-Entwicklungswerkzeug verwendet werden, das eine modifizierbare Codegenerierung aus dem Klassenmodell ermöglicht.

5 Ergebnisse

5.1 Implementierungsaspekte

5.1.1 Allgemeines

Ein Ziel der Arbeit bestand darin, den Aufwand für die Erstellung von Anwendungen im Umfeld der Datenübertragung zu reduzieren und dabei möglichst flexibel auf geänderte Anforderungen, die sich auf die Datenübertragung auswirken, zu reagieren, 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 Konzept der Aktiven Daten-Container bzw. Aktiven Value Objects erreicht. Dabei werden spezialisierte Transportobjekte

zur Datenübertragung verwendet, die über eine standardisierte Schnittstelle verfügen. Die Implementierung der Daten-Container-Objekte kann dadurch während des Lebenszyklus einer Anwendung konfiguriert, erweitert oder gar komplett ausgetauscht werden, ohne die Implementierung der Codeteile, die auf diese Transportobjekte zurückgreifen, zu beeinflussen. Durch den automatischen Datenaustausch zwischen Objektstrukturen, deren Daten transportiert werden sollen (serverseitig) sowie zwischen GUI-Elementen und Daten-Container (clientseitig, GUI-Manager) kann der Entwicklungsaufwand reduziert werden. Die vorgesehene Funktionalität kann sowohl auf den kompletten Inhalt eines Daten-Containers angewendet werden als auch auf einzelne Attribute. Dies ermöglicht eine gezielte Optimierung der transportierten Daten, um das Leistungsverhalten der Anwendung zu verbessern. Insgesamt kann durch die Konzepte dieser Arbeit Expertenwissen in zentrale Datenübertragungsklassen fließen und anderen Entwicklern, die sich ausschließlich mit fachlicher Logik der Anwendung auseinandersetzen, zur Verfügung gestellt werden.

5.1.2 Industrieller Einsatz

Die Konzepte Aktiver Daten-Container und GUI-Manager wurden bei der *iT media Consult GmbH*, Stuttgart, im Rahmen eines Industrieprojekts eingesetzt. Projektziel war die Erstellung einer Applikation zur umfassenden, komplexen Datenerfassung und -auswertung auf Basis einer mehrschichtigen EJB-Anwendung. Als Basisarchitektur kam eine auf zustandslosen Session-Beans, die auf persistente Java-Geschäftsobjekte zugreifen, beruhende Struktur zum Einsatz. 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* des Konzepts zur Übertragung aller im System anfallenden Daten.
- *Dynamische Datenübertragung* zur Berücksichtigung der vielfältigen Anforderungen und zur Reduktion der benötigten Daten-Container.
- *Flexibilität* zur nachträglichen Anpassung der Datenübertragung, um auf unvorhergesehene Anforderungen reagieren zu können.
- *Leichte Benutzbarkeit* zur Einsparung von Entwicklungsaufwand.
- *Keine Geschäftsklassen auf dem Client*.

Im Vordergrund stand die Optimierung des Entwicklungsaufwands. Durch den Einsatz der Konzepte dieser Arbeit konnten die in Tabelle 1 dargestellten Einsparungen erzielt werden.

Durch den Einsatz von dynamischen ADCs konnte in der vorliegenden Applikation insgesamt Entwicklungsaufwand im Umfang von ca. 23678 Codezeilen eingespart werden, was einer Reduktion des Gesamtumfangs um ca. 18% entspricht. In dieser Angabe sind die notwendigen Entwicklungsaufwände zur Umsetzung der Konzepte dieses Beitrags bereits berücksichtigt. Im einzelnen konnte auf die Einführung von 244 *Value Objects* im Umfang von ca. 17028 Programmzeilen (*Lines of Code (LOC)*) verzichtet werden. Diese wurden durch zwei ADCs zum Transport der Daten von Einzelobjekten und Objektfolgen ersetzt. Durch den automatischen Datenaustausch zwischen Daten-Containern und Objektstrukturen auf der Server-Seite konnten 1600 Programmzeilen gegenüber manuell implementierten Austauschoperationen eingespart werden. Dies entspricht einer Reduktion der Datenaustauschoperationen um ca. 83%. Dabei werden nur Attribute übertragen, die im aktuellen Arbeitsschritt des Clients benötigt werden. Unter der Prämisse, daß pro Client-Anfrage immer alle Attribute der darin benötigten Geschäftsobjekte übertragen werden müssen, hätte die Codeeinsparung 3848 Programmzeilen betragen. Auf der Client-Seite konnten durch den automatischen Datenaustausch zwischen Daten-Containern und GUI-Elementen mindestens 5050 Codezeilen eingespart werden, was einer Reduktion um ca. 92% entspricht. Dabei wurden die Daten-Container als clientseitiges Datenmodell verwendet. Aufgrund der Standardschnittstelle wurde eine Komponente (*GUI-Manager*) für den automatischen Datenaustausch zwischen Daten-Containern und GUI-Elementen (im wesentlichen komplexe Swing-Tabellen) automatisiert. Neben der direkten Einsparung von Codezeilen existiert eine zusätzliche Zeiteinsparung, da kaum Überlegungen stattfinden müssen, wie Daten zwischen Containern und Server-Objektstrukturen bzw. zwischen Containern und GUI-Elementen ausgetauscht werden müssen. Zusätzlich entfallen Überlegungen jedes einzelnen Entwicklers, die sich mit einer Optimierung der Datenübertragung befassen.

Tabelle 1. Erzielte Einsparungen (LOC)

Nicht benötigte Daten-Container	17028 (244 Klassen)
Automatischer Datenaustausch zwischen ADC und Server-Objekten	1600
Automatischer Datenaustausch zwischen ADC und GUI-Elementen	5050
Gesamteinsparung	23678

5.2 Leistungsaspekte

Die Optimierung der Datenübertragung kann mit allgemeinen Maßnahmen, die weitgehend unabhängig von den zu transportierenden Daten sind, erfolgen. Dazu gehört z.B. die Anwendung von universell einsetzbaren Kompressionsalgorithmen. Es müssen jedoch auch spezielle Überlegungen durchgeführt werden, die sich auf die Struktur der Daten und die Kommunikationsschicht des zugrundeliegenden Applikations-Servers beziehen. Nachfolgend werden die Ergebnisse einer Untersuchung des Leistungsverhaltens der vorliegenden Konzepte dargestellt.

5.2.1 Testfälle

Zum Vergleich der unterschiedlichen Datenübertragungskonzepte wurden 4 repräsentative Testobjekte in Anlehnung an das in Abschnitt 5.1.2 vorgestellte Industrieprojekt zusammengestellt, deren Attributtypen und -belegungen typischen

Tabelle 2. Untersuchte Implementierungen

	Übertragungs-konzept	Daten-austausch	Daten-format
ADC 1	dynamisch	Reflection	HashMaps
ADC 2	dynamisch	Reflection	Objects
ADC 3	dynamisch	Reflection	Strings
ADC 4	dynamisch	generiert	Strings
AVOC	statisch	generiert	Bytes

Anwendungssituationen entsprechen. Die Belegungen der Attribute mit Werten erfolgte zufällig. Dabei wurden jedoch die typischen Wertebereiche des jeweiligen Anwendungshintergrunds beibehalten und sichergestellt, daß die zu übertragende Datenmenge für alle Tests gleich war. Die Daten der Testobjekte wurden zum Vergleich mit bestehenden und den im Rahmen dieses Beitrags vorgestellten Datenübertragungskonzepten übertragen. Als Beispielimplementierungen wurden die in Abschnitt 4.2 und 4.4 vorgestellten Kernkonzepte prototypisch in Sammel-Container für viele Objekte umgesetzt. In Tabelle 2 sind die wesentlichen Eigenschaften der Daten-Container verkürzt zusammengefaßt. Dabei wird zwischen dynamischen und statischen Konzepten unterschieden. Die Datenaustauschoperationen werden danach unterschieden, ob sie zur Laufzeit mittels Java-Reflection durchgeführt oder mittels generierter Methoden bereits zur Übersetzungszeit bereitgestellt werden. Das interne Datenformat reicht von einer komplexen Struktur aus `HashMap`-Objekten über eine optimierte Struktur aus Objekten (Typ `Object`) und Strings (Typ `String`) bis hin zu einem optimierten, unstrukturierten Byte-Format (Typ: `byte`).

Die Daten-Container-Implementierungen wurden mit den folgenden bestehenden dynamischen und statischen Übertragungskonzepten verglichen:

- *Vektor aus HashMaps (VHM)*: Ein Objekt vom Typ `HashMap` speichert Schlüssel/Wert-Paare, die aus Attributname und Attributwert des Testobjekts bestehen. Zur Übertragung der Daten mehrerer Testobjekte dient ein `Vector` aus `HashMap`-Objekten.
- *Vektor aus Value Objects (VVO)*: Die Daten jedes Testobjekts werden in einem *Value Object* mit denselben Attributen übertragen. Zur Übertragung der Daten mehrerer Testobjekte dient ein `Vector` aus *Value Objects*. Dieses Konzept ist im Leistungsverhalten mit der direkten Übertragung der Testobjekte (Geschäftsobjekte) gleichzusetzen, da alle Attribute transportiert werden und damit der Zustand beider Objekte exakt gleich ist.

Die Tests wurden mit 20 Clients, die den jeweiligen Applikations-Server belasten, durchgeführt. Die Clients konkurrieren dabei um die im System vorhandenen Ressourcen. Zur Durchführung eines Lasttests wurde in jedem getesteten Server eine zustandslose Session-Bean (Test-Bean) installiert. Jeder Client führte dabei ununterbrochen Anfragen durch. Die für den Lasttest verwendeten Objektanzahlen, deren Daten pro Anfrage transportiert werden mußten, sind in Tabelle 3 dargestellt.

Für jedes Datenübertragungskonzept wurde der Transaktionsdurchsatz bestimmt. Eine Transaktion besteht dabei aus der Erzeugung der Daten-Container, den automatisierten Datenaustauschoperationen, der Übertragung und dem Auslesevorgang im Client.

Tabelle 3. Testobjektanzahl pro Anfrage im Lasttest

Testobjekt	Anzahl
1	500
2	200
3	100
4	50

5.2.2 Testergebnisse

In Abb. 8 ist ein Teil der Untersuchungsergebnisse kompakt zusammengefaßt[2]. Die Werte sind dabei auf die bestehenden Verfahren (VHM und VVO) normiert. Um einen Gesamtüberblick zu ermöglichen, wurden die erhaltenen Faktoren aller verwendeten Applikations-Server aufeinandergestapelt. Die ausführlichen Untersuchungsergebnisse befinden sich in [2].

Die beispielhaft umgesetzten dynamischen Konzepte liefern die folgenden Ergebnisse:

- Die Umsetzung ADC 1 liefert mit einem um durchschnittlich ca. Faktor 0,4 gesunkenen Transaktionsdurchsatz gegenüber VHM das schlechteste Leistungsverhalten. Dies ist auf die nicht optimierte Datenstruktur und die Verwendung von Java-Reflection für den automatischen Datenaustausch zurückzuführen. Der Daten-Container dient gleichzeitig als Beispiel für ein Datenübertragungskonzept, das nicht optimiert wurde.
- Die Umsetzung ADC 2 liefert im Vergleich zu VHM einen um durchschnittlich ca. Faktor 0,2 gesunkenen Transaktionsdurchsatz. Die Optimierung der Datenstruktur führt gegenüber ADC 1 zu einem besseren Leistungsverhalten. Die aus Daten vom Typ `Object` zusammengesetzte Datenstruktur wirkt sich auf die Kommunikationsschicht der Server S1, S4 und S7 negativ aus. Es handelt sich dabei um die Gruppe der Server aus Abb. 3, Abschnitt 3.2, die Probleme mit der Übertragung komplex strukturierter Daten haben. Applikations-Server mit einer optimierten Kommunikationsschicht liefern durchschnittlich denselben Transaktionsdurchsatz wie VHM.

[2] Verwendete Umgebung:

- *Server:* Pentium III, 800 MHz, 256 MB Hauptspeicher, Windows 2000, Java 1.3.1 mit aktivierter HotSpotClient VM (build 1.3.1, mixed mode). Der Java-Heap wurde einheitlich 128 MByte gesetzt. Die Applikations-Server waren sowohl kommerzielle als auch Open-Source-Implementierungen. Die Server wurden anonymisiert, da diese Arbeit keinen Vergleich von Applikations-Servern, sondern von Datenübertragungskonzepten anstrebt.
- *Clients:* 10 SPARC-Workstations (UltraSPARC 296-502 MHz, 384-640 MB Hauptspeicher), Java 1.3.1 mit aktivierter HotSpot-Client VM (build 1.3.1, mixed mode). Pro Maschine wurden 2 virtuelle Clients verwendet, die mittels eines im Rahmen dieser Arbeit erstellten Werkzeugs gesteuert wurden.
- *Netzwerk:* 100 MBit/s Ethernet.

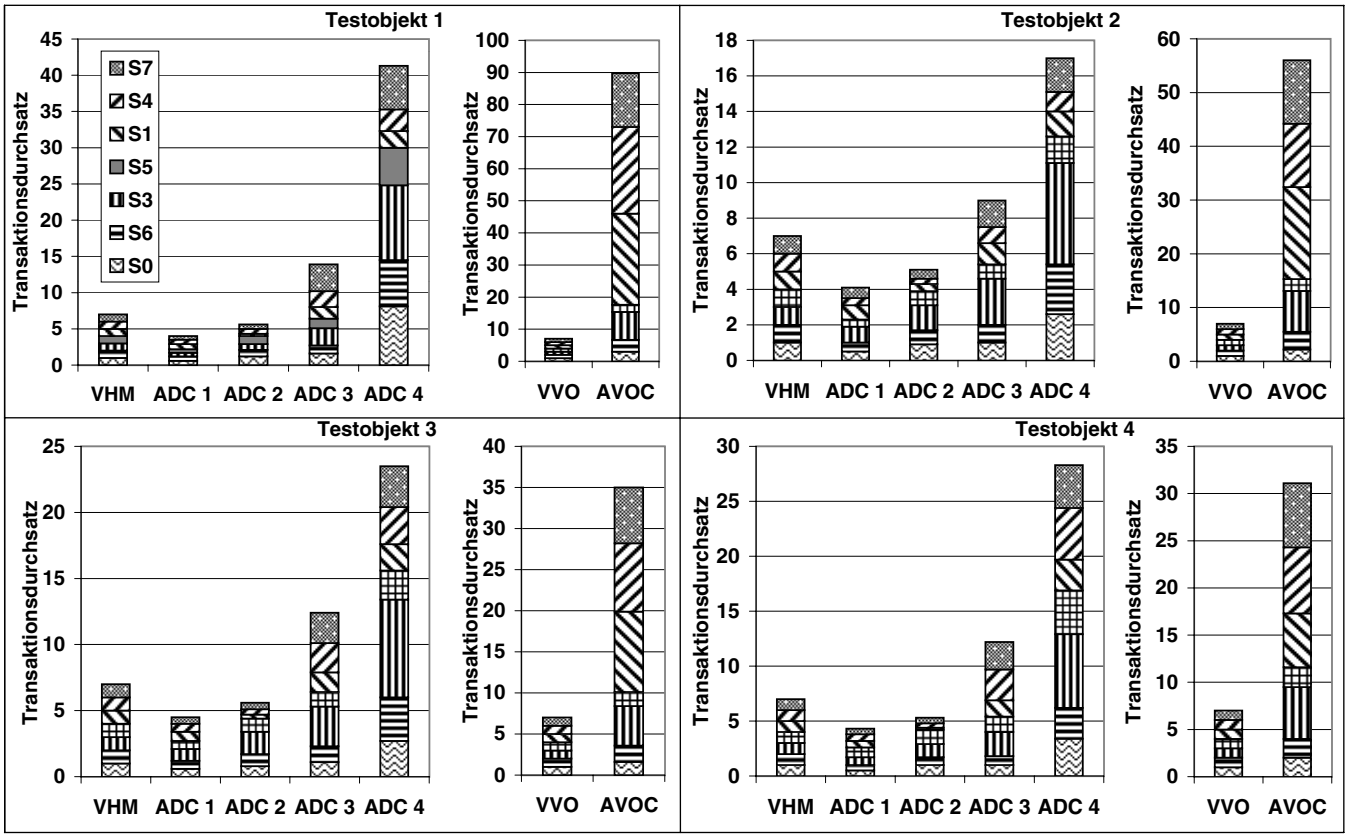


Abb. 8. Transaktionsdurchsatz der Datenübertragungskonzepte

- Die Umsetzung ADC 3 führt zu einem um durchschnittlich ca. Faktor 2 gesteigerten Transaktionsdurchsatz gegenüber VHM. Dies ist auf die signifikant reduzierte Komplexität der verwendeten Datenstruktur zurückzuführen.
- Die Umsetzung ADC 4 führt mit einem um durchschnittlich ca. Faktor 4 gesteigerten Transaktionsdurchsatz gegenüber VHM zum besten Ergebnis. Dies wird aufgrund der Eliminierung der Reflection-Zugriffe beim automatischen Datenaustausch möglich und macht deutlich, wie wichtig die Optimierung aller Teilaspekte eines Datenübertragungskonzepts für das Leistungsverhalten ist. Ebenso werden die hohen Kosten von Java-Reflection in einer Server-Anwendung deutlich.

Das beispielhaft statisch umgesetzte Konzept liefert einen um durchschnittlich ca. Faktor 8 gesteigerten Transaktionsdurchsatz gegenüber VVO. Dabei wirkt sich das Konzept am positivsten auf die Applikations-Server S1, S4 und S7 aus. Dies ist auf die stark vereinfachte interne Datenstruktur des Daten-Containers (AVOC) zurückzuführen. Sie stellt gegenüber komplex strukturierten Daten nur geringe Anforderungen an die Kommunikationsschichten der Applikations-Server.

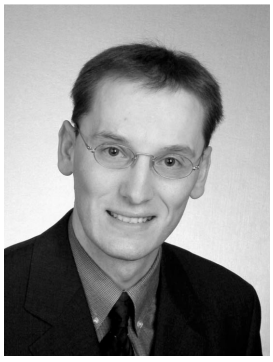
Zusammenfassend kann festgehalten werden, daß mit den in dieser Arbeit vorgestellten Konzepten das Leistungsverhalten einer EJB-Anwendung im Rahmen der Datenübertragung signifikant gesteigert werden kann.

6 Zusammenfassung

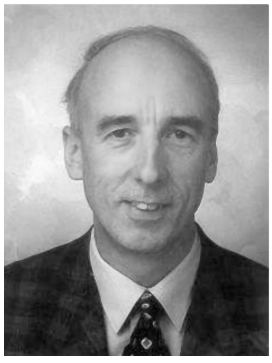
Im vorliegenden Beitrag wurden Datenübertragungskonzepte für EJB-Anwendungen, die mit Java-Clients kommunizieren, vorgestellt. Bestehende Konzepte wurden dabei durch neue Konzepte erweitert bzw. ersetzt. Aktive Konzepte adressieren die Probleme bestehender Datenübertragungskonzepte und Applikations-Server, sind universell einsetzbar und führen zu einem besseren Leistungsverhalten der Anwendung bei gleichzeitiger Reduktion des Entwicklungsaufwands. Dabei sind sie so flexibel und konfigurierbar gestaltet, daß sie während des gesamten Lebenszyklus einer Anwendung mit geringem Aufwand an wechselnde Anforderungen angepaßt werden können. In einem Industrieprojekt konnte der Codeumfang einer komplexen Anwendung insgesamt um ca. 18% gesenkt werden. Diese Reduktion resultiert aus der Einsparung von ca. 99% der Daten-Container-Klassen, ca. 83% der Datenaustauschoperationen zwischen Daten-Containern und Objektstrukturen auf dem Server, sowie ca. 92% der Austauschoperationen 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 konnte eine Steigerung des Transaktionsdurchsatzes bis zum Faktor 8 erzielt werden.

Literatur

1. K. Beschorner, W. Rosenstiel. Effiziente Datenübertragung in EJB-Systemen. In: Net.ObjectDays2000, Ilmenau, 2000. Net.ObjectDays-Forum, c/o tranSIT GmbH.
2. K. Beschorner. Untersuchungen zur effizienten Kommunikation in komponentenbasierten Client/Server-Systemen. Logos-Verlag, Berlin, 2002.
3. T. G. Cowan. Get disconnected with CachedRowSet: The new J2EE RowSet implementation provides updateable disconnected ResultSets in your JSPs. JavaWorld, 2, 2001.
4. T. Davis. Direct network traffic of EJBs: Avoid bottlenecking by encapsulating bean properties into a single object. JavaWorld, 11, 1999.
5. A. Deepak, D. Crupi, J., Malks. Sun Java Center J2EE Patterns. First Public Release: Version 1.0 Beta, 2001, Sun Java Center. http://developer.java.sun.com.
6. L.G. DeMichiel et al. Enterprise JavaBeans Specification, Version 2.0, 2001, Sun Microsystems. http://www.java.sun.com/products/j2ee.
7. D. S. Etzioni, O, Weld. Intelligent Agents on the Internet: Fact, Fiction, and Forecast, 1995, University of Washington, Seattle, Department of Computer Science and Engineering.
8. S. Halloway. Tech Tips: Serialization in the real world. Java Developer Connection, 29.2., 2000. http://developer.java.sun.com/developer/TechTips.
9. N. R. Jennings, K. Sycara, M. Wooldridge. A Roadmap of Agent Research and Development. In Autonomous Agents and Multi-Agent Systems, Boston, 1998. Kluwer Academic Publishers.
10. N. Kassem. Designing Enterprise Applications. Sun Microsystems, Inc., Palo Alto, 2000.
11. C. Larman. Enterprise JavaBeans 201: The Aggregate Entity Pattern. Software Development Magazine, 4, 2000.
12. G. McCluskey. Using Java Reflection. Java Developer Connection, 1998.
13. 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.
14. R. Monson-Haefel. Enterprise JavaBeans. O'Reilly & Associates Inc., 1999.
15. Sun Microsystems. http://java.sun.com/docs/index.html.
16. Sun Microsystems. Java Object Serialization Specification, 1999, Sun Microsystems, Inc.
17. Sun Microsystems. The Java Tutorial: A practical guide for programmers. Sun Microsystems, 2001.
18. T. J. Mowbray, R. Malveau. CORBA Design Patterns. Wiley Computer Publishing, 1997.
19. T. Myers et al. Professional Java Server Programming J2EE Edition. Wrox Press, 2000.
20. C. Nester, M. Philippsen, B. Haumacher. Effizientes RMI f ür Java. In JIT'99 Java-Informationstage 1999, 1999.
21. O. Neumann, C. Pohl, K. Franze. Caching in Stubs und Events mit Enterprise Java Beans bei Einsatz einer objektorientierten Datenbank. In JIT'99 Java-Informationstage 1999, 1999.
22. M. Res. Reduce EJB network traffic with astral clones. JavaWorld, 12, 2000.
23. E. Roman. Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition. Wiley Computer Publishing, 2002.
24. S.K. Singhal, B. Q. Nguyen, R. Redpath, M. Fraenkel, 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.
25. The Advanced Application Architecture Team. iCommerce Design Issues and Solutions, 2000, Beaverton, Oregon. http://www.javasuccess.com.
26. A. Vogel, K. Duddy. Java programming with CORBA: Advanced Techniques for Building Distributed Applications. Addison Wesley, 1998.
27. J. Wilson. Get smart with proxies and RMI. JavaWorld, 11, 2000.



Klaus Beschorner studierte Informatik in Tübingen. Nach Abschluss seines Diploms im April 1998 war er wissenschaftlicher Mitarbeiter am Lehrstuhl Technische Informatik an der Universität Tübingen, wo er im Juli 2002 seine Promotion abschloss. Seine Interessen liegen in den Bereichen objektorientierte Software-Entwicklung und verteilte Anwendungsarchitekturen.



Wolfgang Rosenstiel ist seit 1990 Leiter des Arbeitsbereichs Technische Informatik des Wilhelm-Schickard-Instituts für Informatik der Universität Tübingen und Direktor des Forschungsbereichs Systementwurf in der Mikroelektronik des Forschungszentrums Informatik. Seine Forschungsgebiete umfassen vor allem Eingebettete Systeme, Parallelrechner, Neuronale Netze, Multimedia-Anwendungen und Client/Server-Systeme.

Wilhelm G. Spruth ist als Honorarprofessor an den Universitäten Leipzig und Tübingen tätig. Sein Forschungsgebiet ist die kommerzielle Großrechnertechnologie mit dem Schwerpunkt der Einbindung von z/OS und z/Linux-Systemen in moderne, Internet-orientierte Client/Server-Umgebungen.