3 Komponentenbasierte Softwareentwicklung mit Enterprise JavaBeans

In diesem Kapitel wird ein Überblick über die komponentenbasierte Softwareentwicklung gegeben. Die Grundidee der komponentenbasierten Entwicklung ist zwar weitgehend technologieunabhängig, aber dennoch kommt es bei konkreten Ausprägungen in Vorgehen und Basistechnologien zu konkreten Festlegungen, die substanziellen Einfluss auf konkrete komponentenbasierte Softwareprozesse haben. In diesem Kapitel werden zunächst die grundlegenden Ideen der komponentenbasierten Entwicklung erörtert (Abschnitt 3.1). Danach wird der Begriff der Softwarekomponente in Abschnitt 3.2 diskutiert. Abschnitt 3.3 gibt einen ganz kurzen Überblick über die Bedeutung des Komponentenmodells Enterprise JavaBeans. Weitere Details — insbesondere der Version 2.0 der EJB-Spezifikation — werden in Kapitel 4 erörtert. In Abschnitt 3.4 werden wesentliche Services von EJB-Applikationsservern vorgestellt.

3.1 Komponentenbasierte Softwareentwicklung

Der komponentenbasierten Softwareentwicklung liegt die Idee zugrunde, dass Anwendungen zukünftig aus Bausteinen (den so genannten Komponenten) zusammengesetzt werden können, die auf dem Abstraktionsniveau der Anwendungswelt angesiedelt sind. Mit anderen Worten: ein Anwendungsentwickler in der Versicherungswirtschaft bekommt solche Komponenten zur Verfügung gestellt, die im Kontext der Versicherungswelt eine fachliche Bedeutung haben, ein Anwendungsentwickler in der Logistik arbeitet mit ganz anderen Komponenten. Beide nutzen die Komponenten ihres Kontexts, um daraus Anwendungen aus ihrer Welt zusammenzusetzen.

Vereinfachend wird die komponentenbasierte Entwicklung häufig mit der Legostein-Metapher erörtert. Die Bausteine sollen so präpariert sein, dass sie einfach zusammengesetzt werden können. Sie sollen wiederverwendbar sein und es wird eine ganze Reihe von Bausteinen geben, die nur für bestimmte Anwendungsdomänen geeignet sind. Das Zusammensetzen der Komponenten erfolgt dadurch, dass Komponenten Services anbieten, die von anderen Komponenten benutzt werden. Ein typischer Grundsatz der komponentenbasierten Entwicklung lautet beispielsweise:

Anwendungen werden aus Komponenten zusammengebaut. Komponenten bieten als Service eine Menge von Funktionen an, die in einem bestimmten Gebiet logisch zusammengehören. Andere Komponenten können diesen Service einer Komponente nutzen. Als Vertragsgrundlage zwischen Komponenten dienen dabei Schnittstellen, die Teilmengen der angebotenen Methoden zusammenfassen.

Der Grundsatz besagt, dass es vorgefertigte Bausteine gibt, in denen auf einer höheren Abstraktionsstufe all die Programmteile zusammengefügt sind, die für die Erbringung eines Service in einem bestimmten Bereich zusammenarbeiten. Die Komponente kapselt die Implementierung des Service und stellt diesen Service extern durch eine Menge von Schnittstellen zur Verfügung, die von anderen Komponenten angesprochen werden können. Komponenten können Dienstleistungen aus der Anwendungswelt bereitstellen (Beispiel wäre hier eine Komponente »Partnerverwaltung«), aber auch Infrastrukturkomponenten (z.B. zentral zur Verfügung gestellte, allgemein genutzte Dienste wie Workflowmanagement und Dokumentenmanagement) sind möglich.

Die etwas zu starke Vereinfachung der genannten Metapher besteht darin,

- dass Komponenten nur ganz selten »zusammengestöpselt« werden können, ohne dass Anpassungen notwendig sind,
- dass von der internen Struktur einer Komponente in der Regel nicht so einfach abstrahiert werden kann wie bei einem Legostein,
- dass die Funktionalität der bereitgestellten Services beschrieben werden muss — und zwar möglichst eindeutig und unmissverständlich, bevor sie von anderen Komponenten sinnvoll benutzt werden können.

Die Metapher macht aber auch ein Kernproblem der komponentenbasierten Entwicklung deutlich: Genau wie bei Legosteinen ist es eine ganz wesentliche Entwurfsentscheidung, welche Komponenten bereitgestellt werden. Sind sie zu nah an einer konkreten Anwendung, können sie für diese Anwendung zwar wunderbar eingesetzt werden, die Chance auf ihre erneute Verwendung im Rahmen anderer Anwendungen ist aber eher gering. Sind sie zu allgemein, können sie prinzipiell sehr oft verwendet werden, allerdings muss in jedem Einzelfall mit Anpassungen gerechnet werden.

Die Nutzung von Services anderer Komponenten über Schnittstellen ist in Abbildung 3.1 erläutert. Die obere Komponente bietet einen Service an, der über drei Schnittstellen in Anspruch genommen werden kann (dargestellt am oberen Rand des grafischen Symbols). Jede dieser Schnittstellen realisiert einen Teil des Service. Die obere Komponente nimmt zwei Service-Funktionen anderer Komponenten in Anspruch. Über die entsprechenden Schnittstellen dieser Komponenten kann die obere Komponente bereitgestellte Service-Funktionen aufrufen (dargestellt am unteren Rand des grafischen Symbols für die obere Komponente). Eine Aufrufbeziehung wird mithilfe eines Pfeiles zwischen aufrufender und aufgerufener Komponente veranschaulicht.

Die letzten Jahrzehnte in der Softwareentwicklung waren von zunehmender Verteilung von Softwaresystemen geprägt. Aus monolithischen und intern nicht strukturierten Systemen wurden Two-tier- und Multi-tier-Systeme. Die Aufteilung in Schichten wurde weiter aufgeweicht, als Ergebnis entstanden Systeme, die aus einer Reihe von Bausteinen bestehen, die flexibel verteilt werden können und die über vordefinierte Arten von Beziehungen zusammengesetzt werden können. Abbildung 3.2 veranschaulicht diese Tendenz. Ausgehend von monolithisch strukturierter Software aus den Anfängen der Softwareentwicklung wurden die Systeme immer feingranularer strukturiert. Die Aufteilung

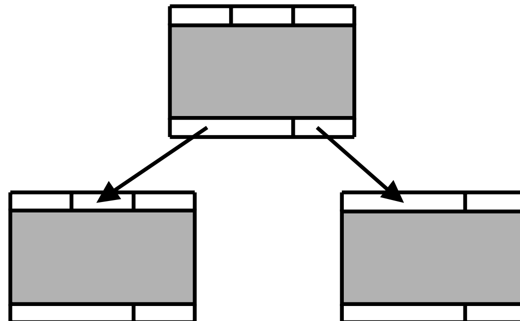


Abbildung 3.1: Nutzung von Services anderer Komponenten über Schnittstellen

in die drei klassischen Schichten Oberfläche, Anwendung/Applikation und Datenbank (DB) und der damit verbundene Übergang zu Client/Server-Systemen stellt einen ersten wichtigen Meilenstein dar. Die Aufteilung in weitere Schichten (oberflächennahe Anwendung, DB-Zugriffsschicht und ähnliche) und die horizontale Aufteilung (also die Aufteilung von Schichten in bestimmte logisch zusammengehörende Teile) setzen den Trend fort. Bei genügend kleinteiliger Strukturierung wird es dann fast unvermeidlich, die fixe Zuordnung von Bestandteilen zu Rechnern aufzugeben und damit bei Geflechten von verteilten und flexibel verteilbaren Komponenten anzukommen.

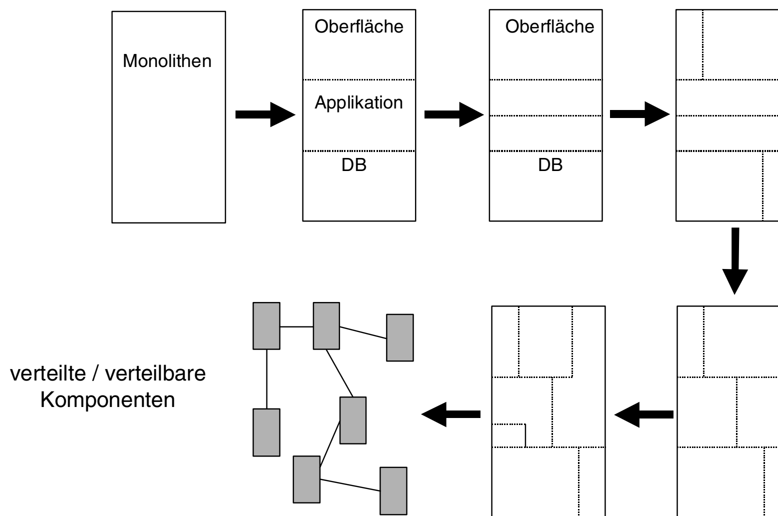


Abbildung 3.2: Verteilung und Strukturierung von Softwaresystemen

Vergleicht man diesen Aufbau mit herkömmlichen Architekturen, wie z.B. Client/Server-Architekturen, so kann man durch die flexible Kombination von Komponenten den Softwareaufbau nicht mehr einer der üblichen Softwarearchitekturen zuordnen. Es

ist somit eine neue Sicht und Beschreibung erforderlich. Abbildung 3.3 zeigt, dass in typischen komponentenbasierten Anwendungen die Anwendungslogik konzentriert in einer Schicht (zumindest aber in einer Reihe von Komponenten) angesiedelt wird, die von der Präsentationsschicht und von der Persistenzschicht getrennt ist. Diese Komponenten werden typischerweise von einem Applikationsserver unterstützt, der mittels der genannten Services eine Konzentration der Anwendungsentwickler auf die Anwendungslogik unterstützt. Auf diese Weise wird es überflüssig, die Anwendungslogik auf Client- und Persistenzschicht zu verteilen und so zu »fetten« Clients oder nicht mehr erkennbaren Persistenzmechanismen beizutragen.

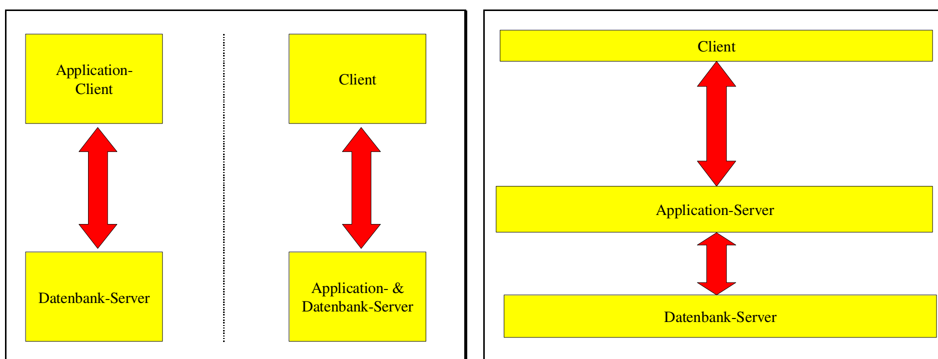


Abbildung 3.3: Zusammenfassung der Anwendungslogik in Komponenten/Schichten

3.2 Softwarekomponenten

Obwohl die grundlegenden Technologien rund um die komponentenbasierte Softwareentwicklung mittlerweile bewährt sind und obwohl die unterstützenden Werkzeuge in konsolidierter Version vorliegen, sind wichtige Begriffe nach wie vor nicht präzise definiert. Zwei der gängigen Erklärungen zum Begriff der Komponente lauten beispielsweise:

Eine Softwarekomponente ist ein Baustein mit vertraglich spezifizierten Schnittstellen und nur ausdrücklichen Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig verwendet werden und leicht mit anderen Komponenten integriert werden. [65]

Eine Komponente ist ein Stück Software, das klein genug ist, um es in einem Stück zu erzeugen und pflegen zu können, groß genug ist, um eine sinnvolle Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten. [29]

Aus Sicht einer komponentenbasierten Softwarelandschaft ist das wesentliche Merkmal einer Komponente, dass sie leicht mit anderen Komponenten gekoppelt werden kann. Darüber hinaus gibt es eine Reihe weiterer Eigenschaften, die Komponenten haben sollten:

- **Wohldefinierter Zweck:** Eine Komponente soll einem bestimmten Zweck dienen, der oftmals umfassender und abstrakter ist als der Dienst eines einzelnen Objektes. Andererseits stellt eine Komponente aber auch keine eigenständige Anwendung dar, vielmehr ist in der Regel die Einbettung in einem Anwendungskontext erforderlich. Eine Komponente ist also ein Spezialist für eine bestimmte Aufgabe. Zum Beispiel kann eine Komponente »Rechtschreibprüfung« in vielen Anwendungen sinnvoll eingesetzt werden. Es ist nicht nötig, eine komplette Textverarbeitung mit fest verdrahteter Rechtschreibkontrolle einzusetzen, wenn die übrige Funktionalität nicht benötigt wird. Die Komponente allein ist aber auch nicht »lebensfähig«.
- **Kontextunabhängigkeit:** Das Zusammenspiel von verteilten Komponenten sollte möglichst unabhängig von Rahmenbedingungen funktionieren, die durch die verwendeten Programmiersprachen, Betriebssysteme, Netzwerktechnologien und Entwicklungsumgebungen gesetzt werden. Alles was man über eine Komponente wissen muss, um über ihre Benutzung zu entscheiden, sollte in der Schnittstelle der Komponente beschrieben werden.
- **Portabilität und Programmiersprachenunabhängigkeit:** Eine Komponente soll in nahezu jeder Programmiersprache entwickelt werden können. Es kann sich dabei sowohl um objektorientierte, aber auch um prozedurale, funktionale oder logische Sprachen handeln. Entscheidend ist nur, dass die fertig übersetzte Komponente universell verwendbar ist. Komponenten sollen plattformunabhängig eingesetzt werden können. Das bedeutet, dass Komponenten ohne Portierung auf einer Vielzahl von Plattformen zum Einsatz kommen können.
- **Ortstransparenz:** Die Nutzbarkeit der Dienste einer Komponente sollte unabhängig vom Ausführungskontext der Komponente sein, d.h. es sollte für den Benutzer keine Rolle spielen, ob sich die Komponente auf dem lokalen Rechner, in einer anderen Prozessumgebung oder auf einem verteilten System in einem entfernten Knoten befindet. Das Verhalten der Komponente ist immer gleich.
- **Trennung von Schnittstelle und Implementierung:** Eine Softwarekomponente und ihre Schnittstelle(n) sollten vollständig unabhängig von der Implementierung spezifiert werden können. Die von ihr angebotenen Dienste sollten wohldefiniert und dem Benutzer bekannt sein. Die Art und Weise der Implementierung ist für den Benutzer transparent. Zur Kommunikation mit der Außenwelt sollten ausschließlich diese exakt spezifizierten Schnittstellen benutzt werden. Ein direkter Zugriff auf das Innenleben einer Komponente ist ausdrücklich nicht erwünscht und in der Regel auch nicht möglich.
- **Selbstbeschreibungsfähigkeit:** Eine Komponente sollte einen Mechanismus unterstützen, der eine Selbstbeschreibung der angebotenen Dienste ermöglicht. Dabei sollten mindestens die Signaturen der Methoden und Attribute abrufbar sein. Selbstbeschreibende Schnittstellen sind eine wichtige Voraussetzung für die Laufzeitkopplung (späte Bindung) von Komponenten und dienen einer besseren Wiederverwendbarkeit.
- **Sofortige Einsatzbereitschaft (plug&play):** Sofort nach der Verfügbarkeit sollte eine Komponente installierbar und einsatzfähig sein. Wünschenswert ist in diesem

Zusammenhang die Fähigkeit zur Selbstinstallation und -registrierung in der zugrunde liegenden Infrastruktur (Betriebssystem, Namens- und Verzeichnisdienste der Middleware).

- **Integrations- und Kompositionsfähigkeit:** Mehrere Komponenten sollten zu einer neuen Komponente zusammengesetzt werden können. Generell müssen Komponenten untereinander über ihre Schnittstellen interagieren können. Die Komposition und Interaktion sollte nicht nur statisch zur »Kompositionszeit«, sondern vor allem auch dynamisch zur Laufzeit möglich sein. Dabei kann eine Komponente sowohl Client einer anderen Komponente sein, als auch gleichzeitig als Server für viele weitere Komponenten fungieren. Die benötigten Basisdienste (Nachrichtenversand, Auffinden einer anderen Komponente usw.) müssen durch die Komponentenarchitektur als zugrunde liegende Infrastruktur und Middleware zur Verfügung gestellt werden.
- **Wiederverwendbarkeit:** Komponenten sollten so gestaltet sein, dass sie in ihrem Anwendungskontext möglichst einfach wieder verwendet werden können. Mit Komponenten als anwendungsnahen Bausteinen wird erstmals das Anwendungsniveau erreicht. Entsprechend kann Wiederverwendung auf Anwendungsniveau ermöglicht werden.
- **Konfigurierbarkeit, Anpassbarkeit:** Es ist bei der Entwicklung und Freigabe einer Komponente nicht vollständig vorhersehbar, in welchen Situationen sie später einmal zum Einsatz kommt. Daher sollten Komponenten über Parameter konfigurierbar sein, um sie einfach und schnell an neue Situationen anpassen zu können. Zum Beispiel sollte eine Komponente zur Rechtschreibkontrolle an ein neues Regelwerk anpassbar sein, ohne die Implementierung ändern zu müssen. Durch eine weitgehende Konfigurierbarkeit kann eine erneute Übersetzung und ein wiederholter Vertrieb der Komponente vermieden werden.
- **Bewährtheit:** Komponenten sollten ausgiebig erprobt und sorgfältig getestet werden, um eine hohe Zuverlässigkeit zu gewährleisten. Es kann daher nicht einfach jede Menge von Objekten als Komponente bezeichnet werden.
- **Binärcode-Verfügbarkeit:** Komponenten liegen in ausführbarer, auf verschiedenen Plattformen lauffähiger Form vor. Sie können ohne Quelltext ausgeliefert werden. Der Binärcode verhält sich auf allen unterstützen Plattformen gleich.

Für keines der gängigen Komponentenmodelle ist gewährleistet, dass die Komponenten, die den Vorgaben des Komponentenmodells folgen, die genannten Eigenschaften vollständig erfüllen. Während viele Eigenschaften von der Sorgfalt und der Arbeitsweise der Komponentenentwickler abhängen, sind andere Eigenschaften durch die Verwendung einzelner Komponentenmodelle geradezu ausgeschlossen. Dies gilt insbesondere für die Eigenschaften der Programmiersprachenunabhängigkeit, der Plattformunabhängigkeit und der einfachen Portierbarkeit. Komponenten, die dem Komponentenmodell Enterprise JavaBeans folgen, liegen in Java vor — nicht in irgendeiner anderen Programmiersprache. Und Komponenten, die Microsofts Komponentenmodell COM entsprechen, laufen auf Microsoft-Plattformen. Nichtsdestrotrotz sind die genannten Kriterien ein gu-

ter Maßstab dafür, inwiefern Softwareteile den Grundideen der komponentenbasierten Entwicklung entsprechen.

Die genannten Kriterien machen keine konkrete Vorgabe bezüglich des Abstraktionsniveaus von Komponenten. Je nach Anwendungssituation kann es zu unterschiedlichen Komponenten kommen. Im Kontext der Entwicklung eines Bestandsführungssystems kann es sinnvoll sein, Komponenten »Partner« und »Vertragsbasisdaten« zu entwerfen. Im Kontext der Entwicklung eines Partnermanagement-Systems kann es zu kleinteiligeren Komponenten kommen. Generell gilt, dass eine Komponente ein Stück Software ist, das klein genug ist, um es erzeugen und verwalten zu können, groß genug ist, um es einsetzen und unterstützen zu können, und das festgelegte Interfaces zur Interoperabilität enthält.

Für die Beschreibung von Komponenten unterscheiden wir zwischen dem Aspekt der Komponentenspezifikation und dem Aspekt der Komponentenrealisierung (vgl. Abbildung 3.4). Die Komponentenspezifikation (oft auch als fachliche Komponente bezeichnet) beschreibt die Funktionalität einer Komponente und abstrahiert von den (technischen) Details der Realisierung dieser Funktionalität. Es werden der angebotene Service (die Dienstleistung) und die einzelnen (Service-)Funktionen benannt, aus denen der Service sich zusammensetzt. Jeder Service faßt mehrere Funktionen, die von der Komponente unterstützt werden, unter einer Sammelbezeichnung zusammen. Der Begriff des Service hilft, die möglicherweise große Menge der angebotenen Funktionen einer Komponente zu strukturieren und zu kleineren und überschaubaren Funktionen-Bündeln zusammenzufassen, die unmittelbar fachlich zusammengehören. Eine Komponente kann mehrere Services anbieten. Die Komponentenspezifikation ist außerdem nützlich, um in einer Komponentenbibliothek nach Komponenten zu suchen, die eine gewünschte Funktionalität bereitstellen können (mit anderen Worten: die eine bestimmte vorgegebene Spezifikation erfüllen). Es kann dabei mehrere Komponenten geben, die in einigen Service-Funktionen oder in der kompletten Spezifikation übereinstimmen.

Die folgenden Tabellen (3.1, 3.2) zeigen ein Beispiel für die Spezifikation einer Komponente, die die genannten Begriffe aufgreift und das Zusammenspiel zwischen Schnittstellen, Services und Service-Funktionen illustriert.

Die Komponentenrealisierung (oft auch als technische Komponente bezeichnet) beschreibt, auf welche Weise der von einer Komponente angebotene Service realisiert ist (das »Wie?«). Der gesamte Service wird durch eine Menge von Interfaces repräsentiert, deren Methoden auf Programmebene aufgerufen werden können. Die Komponentenrealisierung beschreibt eine Komponente durch die Interfaces und deren Methoden (inklusive ihrer Aufrufparameter). Das konkrete Aussehen einer realisierten Komponente ist abhängig vom gewählten Komponentenmodell.

Eine weitere Unterscheidung von technischen Komponenten ist die Unterscheidung in so genannte Client-Komponenten und Server-Komponenten. Client-Komponenten realisieren Benutzungsoberflächen und unmittelbar damit in Verbindung stehende Plausibilitätsprüfungen, jedoch keine weitergehenden Teile der Anwendungslogik. Server-Komponenten realisieren die Geschäftsobjekte mit ihren Daten und Methoden.

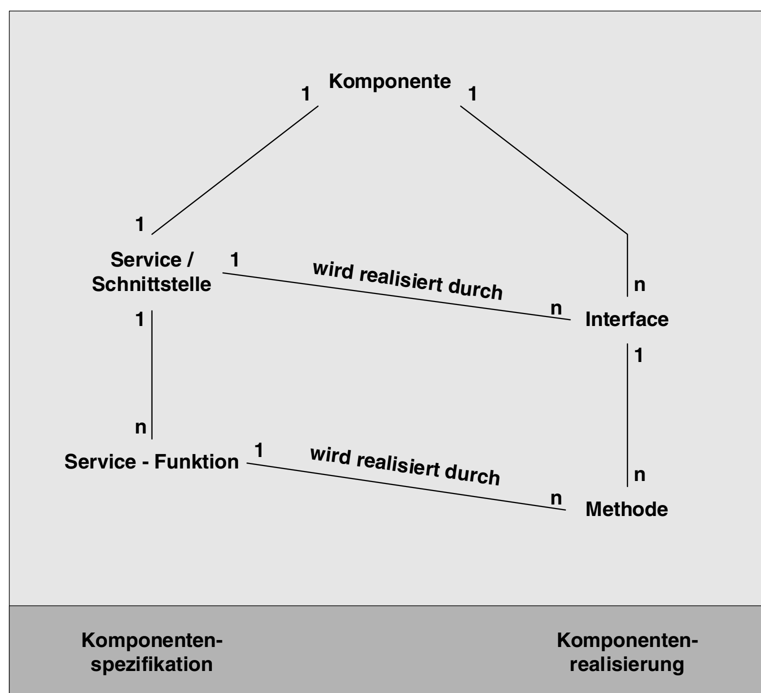


Abbildung 3.4: Komponentenspezifikation und Komponentenrealisierung

Sie repräsentieren die Anwendungslogik und sind von der Repräsentation auf bestimmten Endgeräten getrennt. Abbildung 3.5 verdeutlicht diesen Zusammenhang. Zu einem Geschäftsobjekt (zur Verfügung gestellt durch einen Applikationsserver) gibt es eine Reihe von präsentationskanalspezifischen Client-Komponenten, die sich an verschiedene Anwendergruppen wenden. Jede dieser Anwendergruppen sieht unterschiedliche Repräsentationen der Geschäftsobjekte. Die Darstellung ist auf die spezifischen Anforderungen der unterschiedlichen Anwendergruppen hin angepasst.

Der besondere Vorteil einer komponentenbasierten Software-Architektur liegt darin, dass Komponenten unterschiedlichen Ursprungs miteinander kombiniert werden können. Hierbei können existierende Softwareteile berücksichtigt werden, die als Komponenten verkleidet werden (wrapping). Genauso gut können neue, eigenentwickelte und zugekaufte Komponenten eingebunden werden. Auf diese Weise ist es möglich, dass eine komponentenbasierte Architektur als Zielarchitektur dient, der man sich schrittweise annähert. Hierbei können bestehende Komponenten mit neuen verbunden werden. Bestehende Komponenten können dann — je nach Bedarf — schrittweise gegen neue ausgetauscht werden. Solange die Interfaces unverändert bleiben, ist ein solcher Austausch transparent für die service-nutzenden Komponenten. Ein anderer Vorteil ist, dass Komponenten plattformübergreifend eingesetzt werden können. Da die gängigen Komponentenmodelle den Binärcode standardisieren, der die Implementierung von Komponenten darstellt, ist sichergestellt, dass Komponenten sich auf verschiedenen Plattformen

Spezifikation: Kundenverwaltung

Komponentenname	Kundenverwaltung
Kurzbeschreibung	Verwaltung von Kunden
Langbeschreibung	Mit dieser Komponente werden Kunden verwaltet, indem Adressen und Bankdaten in Beziehung gesetzt werden können. Der Adressenbestand und der Bankdatenbestand können getrennt gepflegt werden. Der Service »Adressdaten-Verwaltung« stellt die Funktionen »Adresse Anlegen«, »Adresse Löschen«,[. . .] zur Verfügung. Der Service »Bankdaten-Verwaltung« bietet die Funktionen [. . .]
Verantwortlicher	Herr Nehm, Abt. Kundenmanagement
Internes Objektmodell der Komponente	Nicht verfügbar
Rolle der Rolle	Server
Verwendete Komponenten	Keine
Aufrufende Komponenten	Lagerverwaltung, Finanzverwaltung
Service	
Service-Name	Adressdaten-Verwaltung
Kurzbeschreibung	Verwalten eines Adressbestandes
Langbeschreibung	Dieser Service bietet Funktionen an, um einen Adressenbestand zu bearbeiten. Die angebotenen Funktionen umfassen Einfügen, Löschen und Drucken von Adressdaten.
[Auflistung der Service-Funktionen, siehe Tabelle 3.2]	
Service	
Service-Name	Bankdaten-Verwaltung
Kurzbeschreibung	Verwalten eines Bestandes von Bankkonten und -bewegungsdaten
[...]	

Tabelle 3.1: Services einer Komponente

identisch verhalten und dass Portierungen und Mehrfachimplementierungen vermieden werden können.

Ein weiterer Vorteil komponentenbasierter Softwarearchitekturen liegt in der mehrfachen Wiederverwendung von Bausteinen und in der damit verbundenen Steigerung der Produktivität. Durch die Möglichkeit der räumlichen Verteilung von Komponenten können diese an einem einzigen Ort stationiert sein und die angebotenen Services von beliebig vielen anderen Komponenten auf beliebigen Plattformen in Anspruch genommen werden. Neben den unmittelbaren Einsparungen, die sich daraus ergeben, dass jeder Service nur genau einmal realisiert werden muss, ergeben sich qualitative Vorteile

Auflistung Service-Funktionen zu Adressdaten-Verwaltung

Service-Funktion	
Service-Funktionsname	Adresse_Anlegen
Kurzbeschreibung	Zu einem existierenden Kunden wird eine neue Adresse angelegt.
Langbeschreibung	Diese Service-Funktion legt eine neue Adresse an. Die Adresse kann von einem der folgenden Typen sein: Hauptadresse, Zusatzadresse, Sonderadresse oder Mitgliedsadresse. [...]
Parameterliste	
Attribut	IN
Parametername	Kundennummer
Beschreibung	Nummer des Kunden
Attribut	IN
Parametername	Adresse
Beschreibung	Neue Adresse
Vorbedingung	Es muss ein Kunde ohne Adresse bestehen.
Nachbedingung	Eine neue Adresse wurde erzeugt. Der Kunde ist mit der neuen Adresse verbunden.

Service-Funktion	
Service-Funktionsname	Adresse_Löschen
Kurzbeschreibung	Die Adresse eines Kunden wird gelöscht.
[...]	

Tabelle 3.2: Servicefunktionen verfeinert bis auf Parameterebene

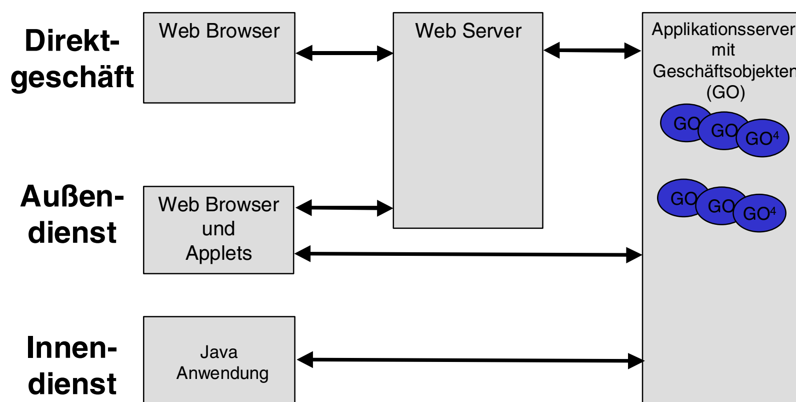


Abbildung 3.5: Client-Komponenten, Server-Komponenten und ihre Ausprägungen

dadurch, dass tatsächlich immer der gleiche Service benutzt werden kann und somit unterschiedliche Verhalten von Servicevarianten vermieden werden.

Ein Komponentenmodell legt fest, welche Anforderungen eine Komponente erfüllen muss und zwar sowohl in syntaktischer, als auch in semantischer Hinsicht. Ein Komponentenmodell macht damit Vorgaben bezüglich der Gestaltung von konkreten Komponenten. Alle Komponenten, die diesen Vorgaben folgen, können mit geringem Aufwand zusammengebaut werden. Um im Bild der Legosteine zu bleiben: Ein Komponentenmodell legt fest, wie die Steckverbindungen aussehen, über die die Komponenten zusammengebaut werden können. Die gängigen Komponentenmodelle sind JavaBeans, Enterprise JavaBeans und COM [30].

Komponenten tauschen Nachrichten miteinander aus (zum Zweck des Methodenaufrufes). Aufrufe über die Grenzen einzelner Anwendungen hinweg erfolgen gemäß eines Standards. Beispiele für solche Standards sind RPC, RMI, RMI über IIOP, und Corba Services. Mithilfe eines Vermittlers (Broker) können Komponenten miteinander kommunizieren, ohne dass sie wissen müssen, wo die aufzurufenden Objekte residieren. Unter Middleware versteht man Softwaresysteme, die benötigt werden, um verteilte Anwendungen oder Komponenten miteinander kommunizieren zu lassen. Middleware ist anwendungsunspezifisch, sie weiß nichts von dem einzelnen Anwendungskontext. In aller Regel werden Middleware-Systeme über eine Programmierschnittstelle aufgerufen. Zu den typischen Middleware-Systemen gehören Netzwerkbetriebssysteme, WWW-Browser, verteilte Datenbankmanagement-Systeme, Workflow-Managementsysteme, Groupware-Systeme, Broker und traditionelle Enterprise Application Integration-Werkzeuge. Zu den typischen Funktionalitäten von Middleware-Systemen sind die folgenden zu zählen: Datentransfer, Kommandoübermittlung (synchroner und asynchroner Aufruf), Empfang von Anfragen, Vermeidung von Verklemmungen, Etablierung und Beendigung von Kommunikationskanälen und spezifischere Dienste aus dem Workflow-/Groupware-Bereich.

3.3 Enterprise JavaBeans als Komponentenmodell

Enterprise JavaBeans (EJB) sind ein Komponentenmodell, das inbesondere die Realisierung von Komponenten unterstützt, die Anwendungslogik realisieren. Sun Microsystems definiert das Komponentenmodell EJB wie folgt:

The Enterprise JavaBeans architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and deployed on any server platform that supports the Enterprise JavaBeans specification. [63]

Ziel des Komponentenmodells EJB ist es, Entwickler bei der Erstellung von Server-Komponenten zu unterstützen. Entwickler sollen die Möglichkeit bekommen, sich auf die Realisierung der eigentlichen Anwendungslogik zu konzentrieren. Querschnittsaufgaben, die bei der Realisierung eines jeden Informationssystems anfallen, sollen den

Entwicklern weitgehend abgenommen werden, indem vorgefertigte Lösungen für solche Aufgaben (Transaktionsmanagement, Persistenz, Sicherheit usw.) in Form von Services angeboten werden. Die Realisierung der einzelnen Services ist Gegenstand von so genannten EJB-Applikationsservern. EJB-Applikationsserver sind Infrastruktursysteme, die Komponenten, die der EJB-Spezifikation folgen, unterstützen.

Der Versuch, möglichst viele und möglichst mächtige Services anzubieten, verdeutlicht, inwiefern sich ein Server-Komponentenmodell von einem Client-Komponentenmodell unterscheidet. Ein Client-Komponentenmodell verfolgt zwar auch das allgemeine Ziel, kompatible Komponenten zu ermöglichen, hat dabei aber eher die Erstellung grafischer Oberflächen im Fokus. JavaBeans sind ein typisches Beispiel für ein Client-Komponentenmodell. Das Komponentenmodell JavaBeans unterstützt den Prozess der Oberflächenerstellung aus vordefinierten Elementen und das Anbinden solcher Oberflächen an die Anwendungslogik. Die Erstellung von Komponenten, die die Anwendungslogik selbst realisieren, wird jedoch nicht ausdrücklich unterstützt. Diese Unterscheidung zwischen Enterprise JavaBeans und JavaBeans wird auch durch das folgende Zitat verdeutlicht:

Enterprise JavaBeans … cannot be manipulated by a visual Java IDE in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server. [20]

Die Idee von server-seitigen Komponentenmodellen im Allgemeinen und EJBs im Besonderen passt ideal zu einem Trend, der sich in der Softwaretechnologie seit Jahrzehnten beobachten lässt. Dieser Trend ist der zur Entlastung der Anwendungsentwickler von technischen Aufgaben, die von der Konzentration auf die Anwendungslogik ablenken. Anwendungsentwickler sollen sich darauf konzentrieren können, das knappe Fachwissen in Software umzusetzen. Sie sollen das tun können, ohne sich dabei über die immer wieder gleichen Transaktionsprobleme Gedanken machen zu müssen und ohne die Abbildung von Klassenstrukturen auf relationale Datenbank-Managementsysteme immer wieder neu erfinden zu müssen. Wie bei allen Lösungen, die den Anspruch erheben, allgemein gültig zu sein, sind auch die Services für EJB-Anwendungen selten ideal für die einzelne Anwendungssituation. In der Regel sind sie aber immer noch besser als die Lösungen, die Fachleute (die ja nicht notwendigerweise Experten in den eher technisch geprägten Fragestellungen sind) erfinden und vor allem reichen sie in der Regel aus.

Die Spezifikation von Enterprise JavaBeans ist von Sun Microsystems erarbeitet worden und liegt seit März 1998 in der Version 1.0 vor. In den letzten Jahren hat es eine Reihe von Folgeversionen dieser Spezifikation gegeben. Eine ganz wichtige Folgeversion ist EJB 2.0. In EJB 2.0 sind eine ganze Reihe von Ergänzungen realisiert, die insbesondere im Hinblick auf die Performanz von EJB-basierten Anwendungen grundlegend und teilweise unverzichtbar sind. An den grundlegenden Zielen von EJBs hat sich seit der Freigabe der Version 1.0 allerdings nichts Wesentliches geändert. Diese Ziele sind:

- Mit EJB soll ein Standard-Komponentenmodell für die Entwicklung von verteilten objektorientierten Anwendungen in Java vorgegeben werden.

- Dem Entwickler soll es ermöglicht werden, sich bei der Entwicklung von Anwendungen vollständig auf die Anwendungslogik zu konzentrieren.
- EJB-basierte Anwendungen folgen der »write-once, run anywhere«Philosophie von Java. Das bedeutet, ein Enterprise Bean kann einmalig entwickelt und ohne Modifikation des Programmcodes oder Rekompilation auf unterschiedlichen Plattformen eingesetzt werden.
- In der EJB-Architektur sind die Aufgaben und Verantwortlichkeiten von Client, Server und den individuellen Komponenten klar definiert.
- EJBs können flexibel zwischen verschiedenen Standorten ausgetauscht werden. Die Transformation in ein und aus einem Austauschformat erfolgt automatisch.

Das große Interesse an EJB-basierten Anwendungen basiert auch darauf, dass das Zusammenwachsen von klassischen Informationssystemen und e-Business/e-Commerce-Lösungen zu flexibel verteilbaren Softwaresystemen führt. Die folgende Einschätzung veranschaulicht diesen Mix von Gründen für die Auseinandersetzung mit EJBs:

During the early 90s, traditional enterprise information system providers began responding to customer needs by shifting from the two-tier, client-server application model to more flexible three-tier and multi-tier application models. The new models separated business logic from system services and the user interface, placing it in a middle tier between the two. The evolution of new middleware services — transaction monitors, message-oriented middleware, object request brokers, and others — gave additional impetus to this new architecture. And the growing use of the internet and intranets for enterprise applications contributed to a greater emphasis on lightweight, easy to deploy clients. [20]

Eine weitere für das Verständnis von EJBs unverzichtbare Unterscheidung ist die zwischen Entity Beans und Session Beans [46]. Entity Beans realisieren die statischen Anteile von Anwendungen, sie repräsentieren die Informationen, um die es letztlich geht. Eine Entity Bean kann von mehreren Clients genutzt werden und wird durch einen Primary Key identifziert. Entity Beans sind transaktionsorientiert und recovery-fähig. Eine Entity Bean kann selbst die Verantwortung für ihre persistenten Daten übernehmen oder die Aufgabe an den Container delegieren.

Session Beans verkapseln Teile von Geschäftsprozessen, sie fassen interaktive Teile zusammen, die in einem engen logischen Zusammenhang stehen. Session Beans repräsentieren somit eher die Dynamik einer Anwendung. Eine Session Bean repräsentiert den Zustand der Kommunikation mit einem Client. Eine Session Bean wird durch einen Client erzeugt, ist exakt nur diesem Client zugeordnet und existiert in der Regel nur während der Zeitdauer einer einzelnen Client/Server-Session. Eine Session Bean führt die Operationen für den Client aus; das können zum Beispiel Datenbank-Operationen oder Berechnungen sein. Eine Session Bean kann transaktionsorientiert sein, ist aber nicht recovery-fähig. Persistente Daten einer Session Bean sind von ihr selbst zu verwalten.

Abbildung 3.6 veranschaulicht das Zusammenspiel zwischen den verschiedenen Arten von Beans. Sie zeigt, wie auf EJBs von außen zugegriffen werden kann und welche Arten von Schnittstellen dazu benutzt werden.

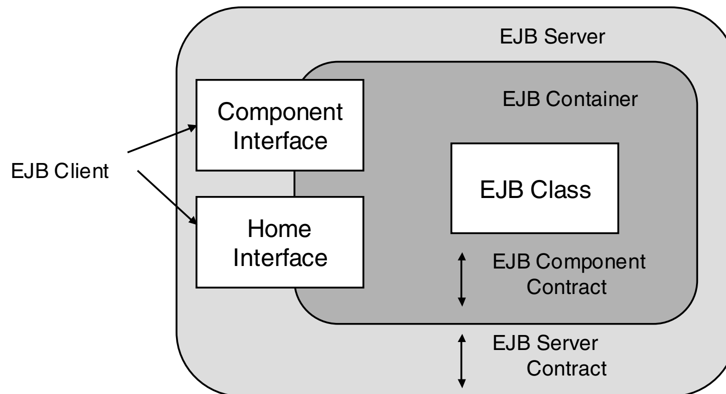


Abbildung 3.6: Schema einer EJB-basierten Architektur

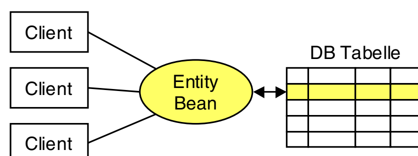
Die in Abbildung 3.6 dargestellten Bestandteile einer EJB-basierten Architektur haben folgende Bedeutung:

- **Schnittstellen zu EJBs:** Ein EJB-Container stellt zwei Schnittstellentypen zur Verfügung, über die der Client mit den EJB kommunizieren kann. Über das Home Interface werden Funktionen zur Verwaltung einer EJB bereitgestellt. Dazu gehören das Erzeugen und Initialisieren einer EJB, das Löschen einer EJB, das Abfragen von Metadaten zur EJB und bei Entity Beans eine Funktion zur Suche nach EJBs. Über das Component Interface (oder in der Diktion des 1.1 Standard: Remote Interface) wird die eigentliche Anwendungslogik eines EJB implementiert.

- **EJB Clients:** Der EJB-Client nutzt die Operationen, die ein EJB anbietet. Der Client findet den EJB-Container durch das Java Naming and Directory Interface (JNDI). Der Client greift niemals direkt auf die Bean-Methoden zu, sondern immer über container-generierte Methoden, die ihrerseits die eigentlichen Bean-Methoden initiieren.

Die Prinzipien der Differenzierung zwischen Entity und Session Beans wird in Abbildung 3.7 zusammenfassend dargestellt. Es ist angedeutet, dass Entity Beans ihren direkten Niederschlag in einer persistenten Datenhaltung finden (oft in einer relationalen Datenbank) und dass Session Beans den koordinierten Zugriff auf eine Menge von Entity Beans ermöglichen, die in einem konkreten Kontext gemeinsam bearbeitet werden müssen.

- Entity Beans
  - Einfache Objekte
  - Komplexe Objekte durch Aggregate
  - Kapselung von Host-Datenbanken



- Session Beans
  - Berechnungen als Dienste
  - Kapselung von Host-Diensten
  - evtl. „lange“ Transaktionen

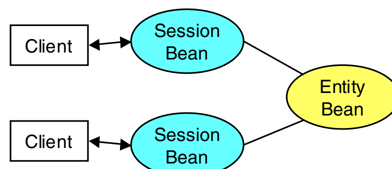


Abbildung 3.7: Abbildung Entity Beans versus Session Beans

3.4 Services der EJB-Applikationsserver

Ein EJB-Applikationsserver hat zweierlei Grundaufgaben. Zum einen soll er den neuen Anwendungen möglichst komfortable Services anbieten und zum zweiten soll er die Integration mit existierenden Anwendungen und Datenbeständen ermöglichen. In der Praxis zeigt sich immer wieder, dass die Realisierung des Applikationsserver-Zugriffs auf existierende Anwendungen besonders aufwändig und fehleranfällig ist.

Ein Applikationsserver stellt die Laufzeitumgebung für Server-Komponenten dar. Er empfängt Nachrichten von den Clients und ruft die dazu gehörenden Methoden der Server-Komponenten auf. Dazu benötigte Objekte werden vor einem Methodenaufruf instanziiert und mit ihren persistenten Daten initialisiert. Während ihrer Lebenszeit werden die Objekte in einem Cache gehalten, was wiederholte Zugriffe auf diese Objekte beschleunigt. Werden Objekte verändert, so synchronisiert der Applikationsserver die dazugehörenden Einträge in dem Persistenzspeicher. Dies geschieht in der Regel transaktionsgesichert.

Im Folgenden wird ein kurzer Überblick über die Services gegeben, die von den gängigen EJB-Applikationsservern angeboten werden. Nicht alle diese Services werden von allen EJB-Applikationsservern angeboten. Oft ist es sogar so, dass die Verfügbarkeit der Services ein Auswahlkriterium für einen konkreten Applikationsserver sein kann. Angesichts der großen preislichen Bandbreite zwischen High-End-Produkten (hier werden je nach Nutzerzahl 6-stellige Euro-Lizenzbeträge fällig) und kostenfrei einsetzbaren Produkten, differieren die Mengen der jeweils angebotenen Services erheblich. Bei einer konkreten Produktbewertung ist jedoch nicht nur die Verfügbarkeit eines Services, sondern auch dessen Qualität und seine genaue Ausgestaltung zu berücksichtigen. An dieser Stelle entsteht dann zuweilen durchaus die Situation, dass kostenfreie Produkte in einzelnen Aspekten den teuren Applikationsservern überlegen sind.

- ▶ **Naming-Service:** Der Naming-Service stellt sicher, dass aufrufende Komponenten nicht über den Aufenthaltsort von aufzurufenden Komponenten und Komponenten-

instanzen informiert sein wissen. Sie müssen nicht wissen, ob ein Aufruf innerhalb des eigenen Adressraums erfolgen muss oder ob es sich um einen entfernten Aufruf handelt. Der Naming-Service ist damit die wesentliche Unterstützung für die Gewährleistung der Ortstransparenz.

- **Lebenszyklus-Management:** Einzelne EJBs brauchen sich nicht explizit um Prozessallokation, Thread-Management, Objekterzeugung oder -vernichtung zu kümmern. Der Lebenszyklus-Service bietet die entsprechende Unterstützung.
- **Zustandsmanagement:** Wenn EJBs im Rahmen eines Geschäftsprozesses verschiedene Zustände annehmen, dann ist der EJB-Zustand zu verwalten. Dies trifft insbesondere dann zu, wenn mehrere Methoden des EJB nacheinander aufgerufen werden (und auch nur in dieser Reihenfolge aufgerufen werden sollen).
- **Sicherheit:** Einzelne EJBs müssen sich nicht um die Authentifizierung oder Autorisierung des Clients kümmern. Zur Prüfung wird der Service benutzt.
- **Persistenz:** Einzelne EJBs müssen sich nicht explizit darum kümmern, wie Daten in der DB abgelegt oder aus ihr herausgeholt werden. Stattdessen kann festgelegt werden, dass EJBs per Container-Managed Persistence persistent gemacht werden. Die Abbildung auf eine — in der Regel — relationale Datenbank erfolgt dann automatisch.
- **Transaktionsmanagement:** Einzelne EJBs müssen sich nicht um die Definition von Transaktionsgrenzen oder die Implementierung von Transaktionsmodellen kümmern. Stattdessen bietet der Service verschiedene Transaktionsmodelle an.
- **Ressourcen-Pooling:** Knappe Ressourcen wie Threads, Client-Connections, DBMS-Connections und Caches werden vom Applikationsserver gehalten. Dies geschieht so, dass diese Ressourcen einer anfordernden Komponente sehr schnell und ohne weiteren Administrations-Overhead zur Verfügung gestellt werden können.
- **Load-Balancing und Failover bei Applikationsserver-Clustern:** Applikationsserver sind skalierbar, weil sie mit geringem Aufwand zu Clustern zusammengeschaltet werden können. Im Falle eines solchen Clusters ist es wichtig, die Last so zu verteilen, dass Engpässe weitgehend vermieden werden. Im Fehlerfall muss dafür gesorgt werden, dass die Aufgaben eines ausgefallenen Applikationsservers dynamisch von einem anderen übernommen werden. Die Load-Balancing- und Failover-Services bieten genau diese Unterstützung.

Ein EJB-Server sorgt dafür, dass dieses Services von den einzelnen EJBs genutzt werden können. Er bietet eine Umgebung, in der aus EJB bestehende Anwendungen ausgeführt werden können. Er managed Mengen von EJBs. Diese werden in Containern verwaltet. Ein EJB-Container (manchmal nur als Container bezeichnet) umfasst eine Menge logisch zusammengehörender EJBs. Eine EJB-Klasse ist genau einem EJB-Container zugeordnet und ein EJB-Container verwaltet genau eine EJB-Klasse. Der Container-Managed den Lebenszyklus der Objekte, implementiert Sicherheitsmechanismen für die Objekte und koordiniert verteilte Transaktionen und die Persistenz von Objekten. Dieser Zusammenhang wird in Abbildung 3.8 veranschaulicht:

– Client
  • beliebig, muss nicht Java sein

– EJB Server
  • kann mehrere Protokolle unterstützen (RMI, IIOP, DCOM)

– Container
  • transparent für Client
  • verwaltet Beans (Lebenszyklus, Zustand, ...)

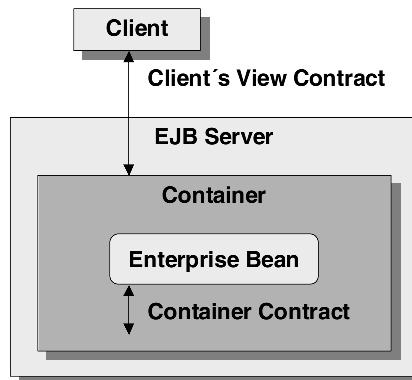


Abbildung 3.8: EJB Server, Container, EJBs und ihr Zusammenspiel

Durch den Einsatz eines Applikationsservers müssen viele Infrastrukturaufgaben nicht mehr programmiert werden. Der Applikationsserver übernimmt sie automatisch bzw. wird dazu konfiguriert (deskriptive statt programmiertechnische Realisierung).