

# **Vorlesung Software aus Komponenten**

## **3. Komponentenmodelle**

Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2004/05

## Java Komponentenmodelle

- Einleitung
- Java Servlets / Java Server Pages (JSP)
- Enterprise Java Beans
  - Einleitung
  - Aufbau
    - o Bean-Schnittstelle
    - o Container-Schnittstelle
    - o Bean-Klassen
    - o Deployment-Deskriptor
  - Architektur
    - o Container
    - o Arbeitsweise
    - o Kommunikation
    - o Persistenz
    - o Identifikation

## Java-Komponentenmodelle

- neben den Applets und JavaBeans (in J2SE) gibt es drei weitere Komponentenmodelle im Java-Umfeld (alle sind Teil der Java 2 Enterprise Edition)
  - **Servlets / Java Server Pages**
  - **Enterprise JavaBeans** und
  - **Application Client Components**
- Alle drei Modelle sind serverseitige Komponentenmodelle.
- Wichtige Gemeinsamkeit ist die **Absetzung der Entpackungsphase** (deployment): Gesteuert durch Deployment-Deskriptor im XML-Format
  - **Entpackung** = Prozess der Vorbereitung einer Komponente auf den Einsatz in einer speziellen **Umgebung** (context)
  - Herstellen/Prüfen der (komponentenspezifischen) Voraussetzungen, unter denen die Komponente arbeitsfähig ist.
    - Beispiel: Datenbank-Anbindung, Persistenzfragen
  - wird unterschieden von der **Installation**, die (plattformspezifisch) eine entpackte Komponente in einer speziellen Hardware-Konfiguration verfügbar macht.

## Java Komponentenmodelle im Überblick

- **Applets:** herunterladbare leichtgewichtige Komponenten für den Einsatz in Webbrowser-Clients
  - Einsatzgebiet rasch durch andere Technologien (GIF animated images, Shockwave, Flash, JavaScript) abgedeckt
- **JavaBeans:** unterstützt verbindungsorientiertes Programmieren
  - zentraler Ansatz: Beobachter und Ereignisse
  - wenig Gemeinsamkeit mit EJB (außer dem Namen)
- **Java Servlets** greifen den Gedanken von Applets auf, laufen jedoch auf dem Server ab und sind (meist) leichtgewichtige Komponenten
  - werden durch einen Web Server instanziiert
  - **Java Server Pages (JSP):** verwandte Technologie, mit der dynamische Webseiten deklarativ definiert werden können
  - Vergleichbar mit den Microsoft Active Server Pages .NET (ASP.NET)
- **Application Client Components**
  - unspezifizierte Java-Applikationen auf einem Client, die auf definierte Weise auf einen Server und dessen Ressourcen zugreifen

## Distribution und Packung von Java-Komponenten

- Geschäftsanwendungen (enterprise applications)
  - ⇒ \*.ear Dateien (enterprise archive)
- Servlets und JSP
  - ⇒ \*.war Dateien (web archive)
- Applets, JavaBeans, EJB
  - ⇒ \*.jar Dateien (Java archive)
- Entpackungs-Beschreibung
  - Formulierung von Anforderungen der Komponente durch Komponenten-Entwickler
  - Setzen offener Parameter (der „Blanks“) der Komponente durch Komponenten-Assembler
    - hart verdrahtet während der Entpackungsphase
    - Komponenten sind sonst zustandslos
  - Assembler ist eine andere Rolle als Komponentenentwickler, oft sogar in einer anderen Organisation
    - andere Ansätze trennen diese beiden Rollen deutlicher

## Java Server Pages und Servlets

- dynamische Webseiten = Kombination dreier grundlegender Funktionen
  - ankommende Anfragen akzeptieren, auf Gültigkeit und Autorisierung prüfen und an geeignete Komponente zur Weiterverarbeitung abgeben
  - relevante Information aus den Informationsquellen extrahieren und den angefragten Inhalt (content) zusammenstellen
  - Inhalt an den Anfrager übermitteln
- Prototypisches Modell: wird von einem **Webserver** abgehandelt
  - HTTP-Anfragen empfangen
  - URL und enthaltene Parameter auswerten
  - statische oder dynamische HTML-Seite (generiert mittels Aktivierung einer Komponente, z.B. über eine einfache Schnittstelle wie CGI) zurücksenden
- Modell ist nicht auf HTML-Anfragen beschränkt
  - Szenario liegt allen typischen Web-Diensten (Web Services) zu Grunde
  - Dienste-Komponenten müssen nur eine simple Server-Schnittstelle implementieren

- **Realisierung 1:**  
**Einbettung von Code direkt in das Markup einer HTML-Datei**
  - Realisierung durch Active Server Pages (ASP) und Java Server Pages (JSP)
  - Aus den Script-Teilen wird HTML-Code generiert
  - Webserver ersetzt den Script-Code durch den generierten Code
- Beispiel einer einfachen JSP-Seite:

```
<HTML><BODY>
<%
    java.util.Calendar calendar = new java.util.GregorianCalendar();
    int hour = calendar.get(currTime.HOUR);
    int minute = calendar.get(currTime.MINUTE);
%>
The time is: <%= hour %>: <%= minute %> - or it was when I looked.
</BODY></HTML>
```
- **Realisierung 2:** **Erzeugung von Markup durch Java Code**
  - Java Servlets: Interaktion mit dem Nutzer auf der Browser-Seite über den Webserver

- Das gleiche Beispiel als implementiertes Servlet:

```
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet
{
    protected void doGet (HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException
    {
        java.util.Calendar calendar = new java.util.GregorianCalendar();
        int hour = calendar.get(currTime.HOUR);
        int minute = calendar.get(currTime.MINUTE);
        ServletOutputStream out = resp.getWriter();
        out.print("<HTML><BODY>\r\n");
        out.print("The time is: " + hour + ":" + minute +
            " - or it was when I looked.\r\n");
        out.print("</BODY></HTML>\r\n");
    }
}
```



JSP versus Servlets:

- JSP erzeugen im Prinzip genau den Code des äquivalenten Servlets
- JSP existieren nur auf der Ebene von Java-Instanzen
  - keine Unterstützung der natürlichen Abstraktionsmechanismen von Java (Pakete, Klassen, Methoden)
- JSP können wie Servlets auf externen Java-Code zugreifen
  - Regel: Java Code in JSP auf absolut notwendigen „Klebstoff“ reduzieren, funktionalen Teil in separate Klassen auslagern
- JSP: Konfusion mit clientseitigem JavaScript-Code möglich
- Ausdrucksmächtigkeit von JSP kann durch JSP Standard Tag Library (JSTL) aufgebohrt werden.
  - Besonders in Richtung XML-Verarbeitung (SOAP, Web Services)
  - Erweiterung des Sprachumfangs muss durch den Webserver unterstützt werden

#### Vorteile des Servlet-Modells:

- Inhalts-Generierung kann über mehrere Servlets verteilt werden
- Trennung in Präsentationsgenerierung und Geschäftslogik
- weitergehende Faktorisierung von Servlets längs Aufgabengrenzen möglich
  - ⇒ Abstraktions- und Modellierungsprinzipien des klassischen Software-Entwurfs sind anwendbar
  - ⇒ Servlets als Komponentenmodell
- Servlets bieten sich auch als Einstiegspunkt in komplexere Geschäftsanwendungen an, die beispielsweise auf Enterprise JavaBeans aufsetzen
  - Probleme mit der Mischung unterschiedlicher Komponenten-Modelle:
    - mehrere Infrastrukturen müssen vorgehalten werden und interferieren
    - Kommunikation zwischen den Modellen muss entworfen werden

### Einleitung

- keine Gemeinsamkeiten mit JavaBeans
  - JavaBeans: Komposition durch **verbindungsorientierte** Programmierung
    - Beans können Ereignisquellen und –beobachter sein
    - Ereignisfluss wird festgelegt durch Anbinden von Quellen in einer Bean an Beobachter in anderen Beans
  - Könnte mit entsprechenden Konzepten (Beans-Container, Ereignisentkopplung durch InfoBus) auch **andere Kompositionskonzepte** (datenorientierte bzw. kontextorientierte Komposition) realisieren
  - spielt aber heute praktisch kaum eine Rolle, obwohl die Infrastruktur dafür vorhanden ist
- Das EJB-Konzept realisiert einen klassischen OO-Zugang
  - Kommunikation über Methodenaufrufe und Objektgenerierung
  - Komposition von e-beans nur über eigenes Design
  - generische verbindungsorientierte Kompositionskonzepte werden nicht unterstützt (erst mit CCM als Erweiterung von EJB)

- Spezifikation, kein konkretes Produkt
- Im Mittelpunkt steht ein **kontextuelles Kompositionskonzept**
  - automatische Komposition von Komponenteninstanzen mit zugehörigen Ressourcen und Diensten
- das EJB-Konzept basiert auf **e-Beans** und **EJB Containern**
- In der Deployment-Phase erfolgt über den EJB Container eine kontextuelle Zusammenführung der Beans mit Diensten und Ressourcen
  - Container ist vergleichbar mit statischen Methoden, Beans mit Instanzmethoden einer Java-Klasse
  - Container ist der „Pate“ der Beans, über welchen die gesamte Kommunikation läuft
  - EJB Container werden von EJB-Servern bereitgestellt
    - J2EE Standard: J2EE application server
- Beschreibung (Inhalt, Relationen, Rollen-, Sicherheits- und Transaktionsverhalten) in einem speziellen **Deployment-Deskriptor**
  - da solche Deskriptoren umfangreich sein können, ist Werkzeugunterstützung sowohl zur Erstellung als auch zur Entpackung erforderlich

- kein direkter Zugriff auf Attribute und Methoden von Beans, auch nicht innerhalb desselben Containers
  - Verhältnis wie zwischen DB-Server und Datensätzen
  - Zugriff nur über zwei Schnittstellen, die **javax.ejb.EJBHome** (für Container) und **javax.ejb.EJBObject** (für Beans) erweitern
  - EJBHome: Methoden zum Management des Lebenszyklus der Beans
  - EJBObject: Methoden der Bean-Instanzen
- Zu beiden gibt es **Local**-Varianten für den Einsatz in einer lokalen Umgebung
- Implementation von:
  - Borland (Borland AppServer)
  - HP (HP Bluestone)
  - IBM (IBM WebSphere Application Server)
  - Oracle (Oracle 9iApplicationServer)
  - ❖ weitere: <http://java.sun.com/j2ee/compatibility.html>

### Bean-Schnittstelle EJBObject

- interface **MyEJBObject** extends **javax.ejb.EJBObject**
- Aufruf-Schnittstelle, über welche ein Client auf die Dienstleistung zugreifen kann
- **Beschreibt** Dienstleistung
  - Darstellung von Attributen über get/set-Methoden
- Nichtlokale Clients rufen Schnittstelle auf Stub-Objekt, welches über RMI oder RMI-über IIOP mit dem EJB Container kommuniziert
  - deklarierte Operationen müssen Exception **java.rmi.RemoteException** auslösen können
  - Abfangen von Kommunikationsproblemen im Netz
- Lokale Clients können über lokale Version der Schnittstelle (Erweiterung von **EJBLocalObject**) zugreifen, wenn von der e-bean bereitgestellt

### Container-Schnittstelle EJBHome

- interface **MyEJBHome** extends **java.ejb.EJBHome**
- Verwaltungs-Schnittstelle: verwaltet Bean-Instanzen im Container
  - Erzeugen neuer Instanzen
  - Auffinden vorhandener Instanzen
  - serialisiert alle Zugriffe auf seine Beans
- Operationen **create** und **findByPrimaryKey** (entity beans)
- optional weitere Methoden, die nicht mit einzelnen Beans zu assoziieren sind (analog zu statischen Methoden einer Klasse)
- begleitet Lebenszyklus seiner Beans
  - **setEntityContext** oder **setSessionContext** erzeugt Rückverweis auf den Container-Kontext
  - **ejbCreate** / **ejbRemove**
    - Ressourcenallokation bzw. -freigabe
  - **ejbPassivate** / **ejbActivate**
    - zeitweise Trennung von den Ressourcen und Speichern in serialisierter Form auf externem Medium

### Bean-Klassen

- `public class MyBeanClass implements javax.ejb.xxBean`
- es gibt **SessionBeans**, **EntityBeans** und **MessageDrivenBeans** als Subklassen von **EnterpriseBeans**
- **Implementierung** der zur Verfügung gestellten Operationen
  - also eigentlich auch ... **implements MyEJBObject**
  - wird aber nur informell überwacht und diese Verbindung erst bei der Installation hergestellt
  - Entwickler muss auf Übereinstimmung der Signaturen selbst achten



### Bean-Klassen

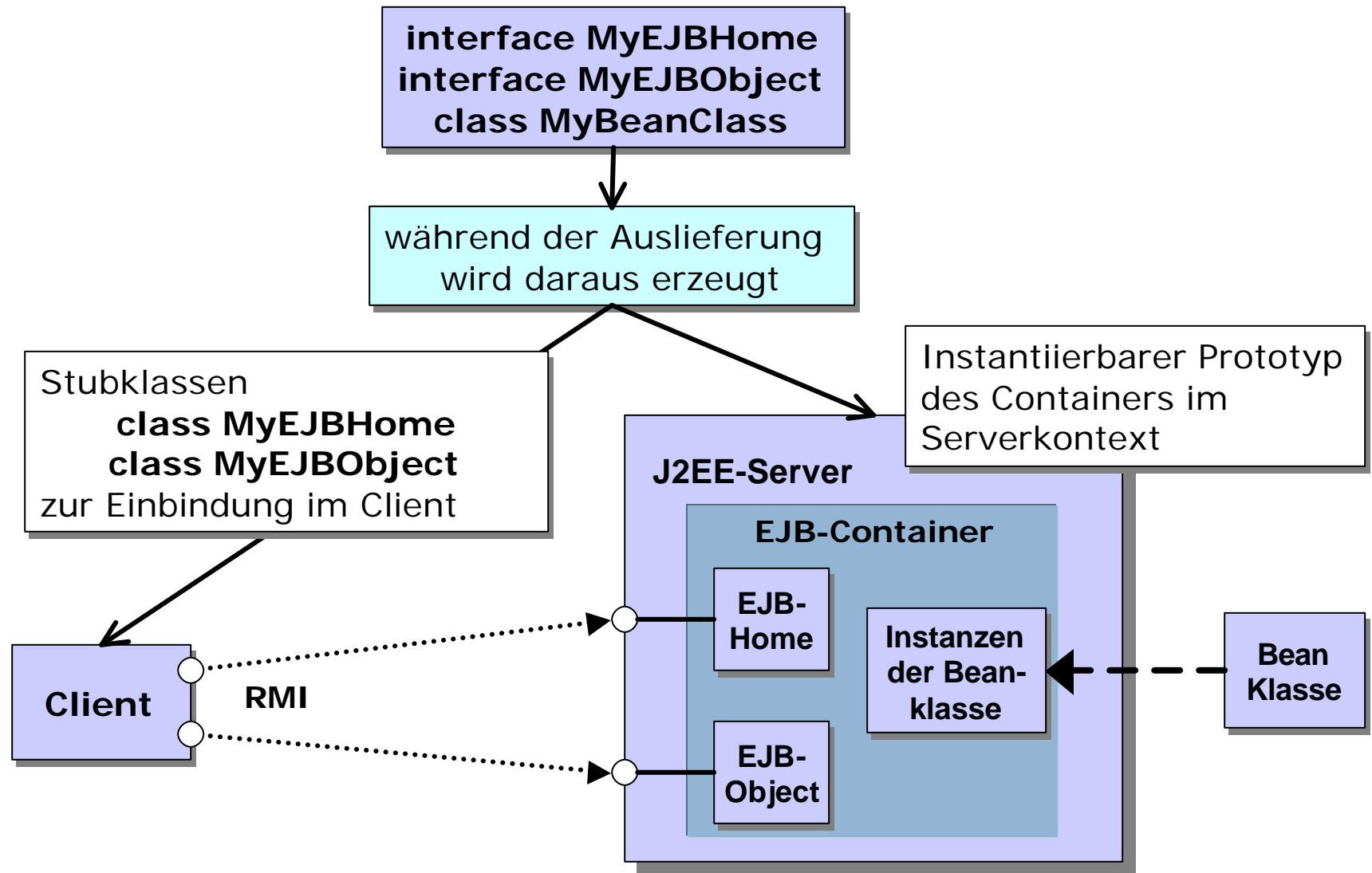
Es gibt vier Sorten von Enterprise JavaBeans

- **zustandslose** (stateless) und **zustandsbehaftete** (stateful) **Session Beans**
  - implementieren **javax.ejb.SessionBean**
  - entsprechen einer Session in der Datenbankterminologie
  - beide Arten sind transient, also nur innerhalb einer Session gültig
  - Zustand wird zwischen verschiedenen Methodenaufrufen gespeichert/nicht gespeichert
- **Entity Beans**
  - implementiert **javax.ejb.EntityBean**
  - enthalten persistente Daten, entsprechen dem Zugriff auf einen konkreten Datensatz in einer Datenbank
    - bzw. eher einem Datensatz in einem Join
    - Persistenz kann Bean-gesteuert (etwa JDBC-Implementierung) oder Container-gesteuert sein

- Zugriff wie bei Datensätzen über Primärschlüssel
  - ggf. muss eine Bean erzeugt und an den Datensatz mit diesem Schlüssel gebunden werden
  - Wert aus validem Java-Typ (auch komplexer Natur)
- Anfragesprache EJB QL ähnlich SQL
- **nachrichtengesteuerte** (message-driven) **Beans** (neu in EJB 2.0)
  - gebunden an Container, transient, aber ohne Schnittstelle
  - Bean wird einer Nachrichtenschlange zugeordnet und, wenn sie „dran“ ist, ihre zentrale Methode onMessage abgearbeitet.
  - Schlange muss dem Java Messaging Standard (JMS) genügen
  - damit kann voll asynchrones Kompositionsmodell entworfen werden
    - sinnvoll etwa für workflow-orientierte automatische Systeme

### 3.3. Sun und Java

## Enterprise JavaBeans (EJB)



### Auslieferungs-Beschreibung (Deployment-Deskriptor)

- Beschreibungsdatei im XML-Format

Beispiel Session-Bean

```
<session>
  <ejb-name> Name der Session-Bean </ejb-name>
  <home> Name der EJBHome Schnittstelle </home>
  <remote> Name der EJBObject Schnittstelle </remote>
  <local-home> Name der EJBLocalHome Schnittstelle </local-home>
  <local> Name der EJBLocalObject Schnittstelle </local>
  <ejb-class> Name der Bean-Klasse </ejb-class>
  <session-type> stateless | stateful </session-type>
  <transaction-type> Container (default) | Bean </transaction-type>
  <ejb-ref> Importdeklaration anderer Beans </ejb-ref>
  <security-identity> eigene oder die des Aufrufers </security-identity>
</session>
```

- enthält Informationen zur Installation:
  - Schnittstellen, Attribute, Operationen
  - Rollen für Benutzer
  - Rechte dieser Rollen
  - Transaktionsverhalten

### Anforderungen an den EJB–Container-Kontext

- JVM wird benötigt, ist aber allein nicht ausreichend
- auf einem J2EE–Server können mehrere EJB–Container gleichzeitig laufen
- in einem Container können wiederum mehrere (Arten von) Beans gekapselt sein
- Schnittstelle und Verhalten von Container und J2EE sind spezifiziert
- Container stellt Dienste zur Verfügung
  - Verwaltung seiner Beans
    - Namensdienst (Java Name and Directory Interface)
    - Transaktionsmonitor (Java Transaction API)
    - Datenbankzugriff (Java Data Base Connectivity)
    - eMail (Java Mail API)
    - Standard Java API
- optional: eine Bean kann eine Schnittstelle implementieren, über die sie vom Container über Ereignisse informiert wird

### Bean-Schnittstelle

```
package SemOrg.Schnittstellen;  
import java.rmi.*;  
import javax.ejb.*;  
  
public interface Buchung extends EJBObject {  
    public void buchen(Kunde k, Seminartyp s) throws RemoteException;  
}
```

### Container-Schnittstelle

```
package SemOrg.Schnittstellen;  
import java.rmi.*;  
import javax.ejb.*;
```

```
public interface BuchungHome extends EJBHome {  
    public Buchung create() throws RemoteException, CreateException;  
}
```

### Bean-Klasse

```
package SemOrg.Server;
import SemOrg.Schnittstellen.*;
import java.rmi.*;
import javax.ejb.*;

public class BuchungBean implements SessionBean{

    // Konstruktor
    public BuchungBean() {}

    //Operationen der Remote-Schnittstelle Buchung
    public void buchen(Kunde k, Seminartyp s) throws RemoteException
    { //Code zur Ausführung der Buchung }

    // Implementierung der Schnittstelle SessionBean
    public void ejbRemove() {}
    public void ejbActivate() {} // etc.
}
```



### Deployment-Deskriptor

```
<!-- Deployment descriptor -->
<enterprise-beans>
  <session>
    <ejb-name> SemOrg/Buchung </ejb-name>
    <home> SemOrg.Schnittstellen.BuchungHome </home>
    <remote> SemOrg.Schnittstellen.Buchung </remote>
    <ejb-class> SemOrg.Server.BuchungBean </ejb-class>
    <session-type> Stateless </session-type>
    <transaction-type> Container </transaction-type>
  </session>
</enterprise-beans>
```

### Ein einfacher Client

```
package SemOrg.Client;
import java.rmi.*;
import javax.naming.*;
import SemOrg.Schnittstellen.*;

public class Client {
    public static void main(String[] args) {
        Client einClient=new Client();
        einClient.run();
    }

    public void run() {
        Kunde einKunde=getKunde();
        Seminar einSeminar=getSeminar();
    }
}
```

```
// Fortsetzung Methode run
try {
    // Namenskontext des Servers finden
    Context ctx = new InitialContext();
    Object temp = ctx.lookup("java:comp/env/Buchung");

    // EJB-Container-Objekt vom Typ BuchungHome auf dem Server
    // finden und instantiieren
    BuchungHome home=
        (BuchungHome) PortableRemoteObject.narrow(
            temp, BuchungHome.class);

    // Bean anfordern und Buchung auslösen
    Buchung bean = home.create();
    bean.buchen(einKunde, einSeminartyp);
}
catch(Exception e) { }
```

### Server-Provider (Server-Anbieter)

- stellt Plattform zur Verfügung
- Netzwerkanbindung
- Skalierungsfunktion
- Prozess- und Ressourcenmanagement

### Container-Provider (Container-Anbieter)

- setzt auf Plattform des Server-Providers auf
- Benutzerverwaltung
- Transaktionsmanagement
- Persistenz
- Installationswerkzeuge
- Implementation der in der EJB-Spezifikation definierten Schnittstellen
- Container- und Server-Provider oft identisch
  - bessere Performance und Wartbarkeit

### Bean-Provider (Komponenten-Entwickler)

- realisiert geforderte Anwendungslogik (Geschäftslogik)
- keine grundlegenden Funktionalitäten (Persistenz, Netzwerk, etc.)
- benötigt Fachwissen über Anwendungsbereich
- Konzentration auf inhaltliche Problemstellung

### Application-Assembler (Monteur)

- verbindet Komponenten zu einer Anwendung
- Clients
- Verwendung von Basiskomponenten
- Nutzung wiederverwendbarer Komponenten von Drittanbietern
- oftmals Bean-Provider und Application-Assembler in einem Haus

### Bean-Deployer (Installation)

- installiert Komponenten der Anwendung auf Zielsystem
- verwendet Werkzeuge des Container-Providers
- nutzt Deployment-Deskriptor
- konfiguriert EJBs
- generiert Stummel- und Skelettklassen
- legt Benutzerdatenbank an
- benötigt detailliertes Wissen über Server und Container

### Systemadministrator

- verantwortlich für reibungslosen Ablauf
- Benutzerverwaltung