LE 29
1

Software-Technik

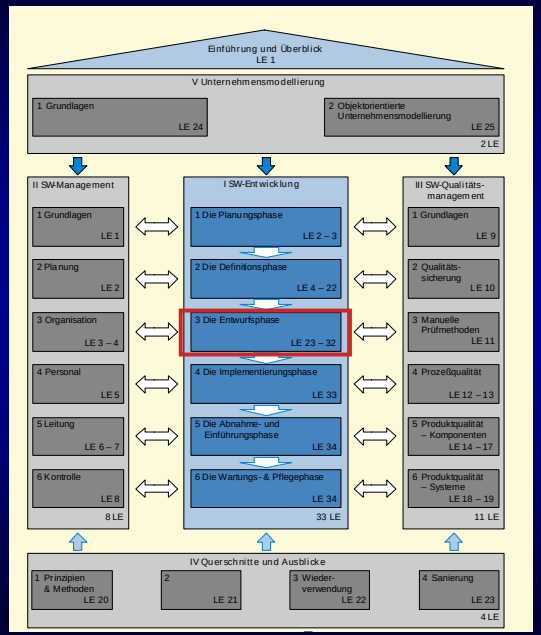
3 Die Entwurfsphase Verteilte objektorientierte Anwendungen

[Leicht gekürzt]

Prof. Dr. Helmut Balzert
Lehrstuhl für Software-Technik
Ruhr-Universität Bochum

© Helmut Balzert 2001

LE 29
2



LE 29
3

Inhalt

3.9 Verteilte objektorientierte Anwendungen
- 3.9.1 Software für Unternehmenslösungen
- 3.9.2 *Server*basierte Komponenten-Modelle
- 3.9.3 *Java-RMI*
- 3.9.4 *Enterprise JavaBeans*
- 3.9.5 CORBA
- 3.9.6 COM+
- 3.9.7 Die Modelle im Vergleich.

LE 29
4

3.9.1 Software für Unternehmenslösungen

- Unternehmenslösungen
  - Von vielen Benutzern gleichzeitig genutzt
  - Verteilt (oft über große Entfernungen)
  - Arbeiten mit unternehmenskritischen Daten
  - Anforderungen
    - Mehrbenutzerfähigkeit
    - Skalierbarkeit
    - Verfügbarkeit
    - Verbindung mit der Außenwelt
    - Schrittweise Migration
    - Alles nicht-funktionale Anforderungen, die für alle Unternehmenslösungen mehr oder weniger zutreffen.

LE 29
5

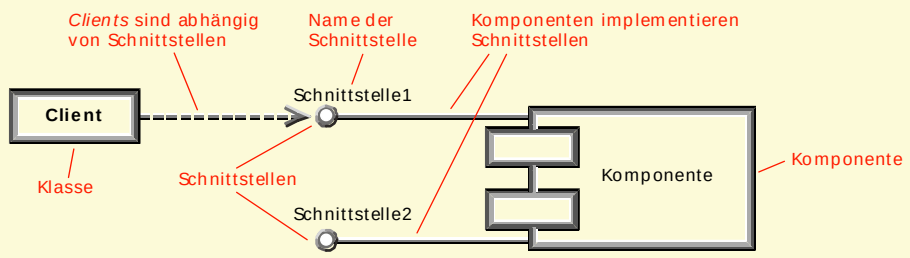
3.9.1 Software für Unternehmenslösungen

- Trennung Infrastruktur – Anwendungslogik
  - Infrastruktur wird von Entwerfern für verteilte Systeme entwickelt, die keine konkreten Geschäftsfelder kennen müssen
  - Anwendungslogik entwickeln Anwendungsspezialisten für den jeweiligen Anwendungsbereich
  - Diese Arbeitsteilung ermöglicht die Entwicklung *server*basierter Komponenten-Modelle, die unabhängig von einem konkreten Anwendungsbereich eine Infrastruktur spezifizieren
  - Komponenten-Plattformen realisieren die im Komponenten-Modell spezifizierte Infrastruktur
    - *Middleware*-Plattformen.

LE 29
6

Komponenten in der UML

- Komponente
  - *»A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs«*
- UML-Notation



LE 29
7

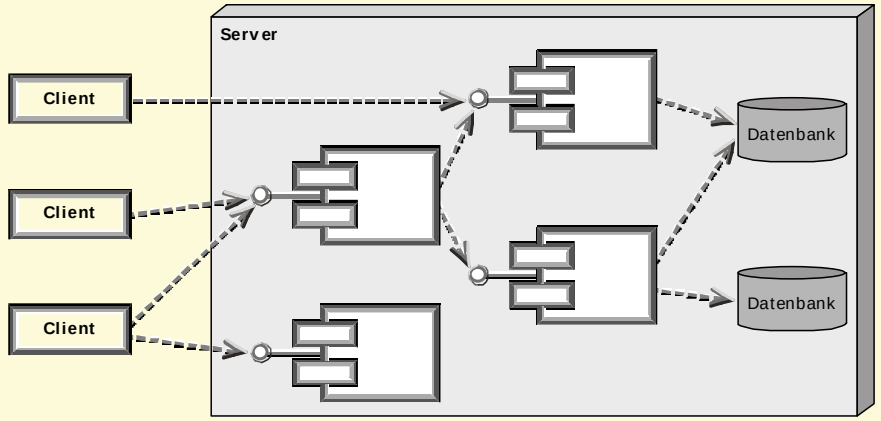
3.9.2 *Server*basierte Komponenten-Modelle

- Geschäftsobjekt
  - *»A business object is defined as a representation of a thing active in the business domain, including at least its business name and definition, attributes, behaviour, relationships, rules, policies and constraints. A business object may represent, for example, a person, place, event, business process or concept. Typical examples of business objects are: employee, product, invoice and payment.«*
  - Geschäftsobjekte werden von Komponenten-Modellen auf Komponenten abgebildet
    - Komponentenmodelle: keine Vererbung zwischen Komponenten
    - Von einer Komponente werden zur Laufzeit Objekte erzeugt (Geschäftsobjekte).

LE 29
8

3.9.2 *Server*basierte Komponenten-Modelle

- *Server*seitige Komponenten



LE 29
9

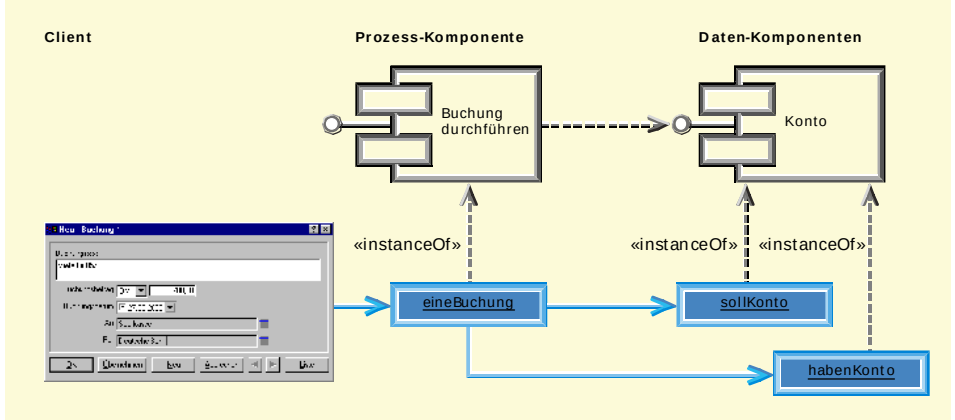
3.9.2 *Server*basierte Komponenten-Modelle

- 2 Arten von Komponenten
  - Daten-Komponenten *(entity components)*
    - OOA-Klassen: Durch Komponenten realisiert
    - Die Komponente stellt eine Schale für diese Daten dar
    - Dadurch wird auf der einen Seite eine OO-Sichtweise auf die Daten ermöglicht, auf der anderen Seite muss auf die Vorteile einer (relationalen) Datenbank nicht verzichtet werden
  - Prozess-Komponenten *(session components)*
    - Geschäftsprozesse: Durch Komponenten realisiert
    - Prozess-Objekte selbst enthalten meist keine Daten, die dauerhaft zu speichern sind.

LE 29
10

3.9.2 *Server*basierte Komponenten-Modelle

- Beispiel für eine Prozess-Komponente



LE 29
11

3.9.2 *Server*basierte Komponenten-Modelle

- Prozess-Objekte
  - Entsprechen nicht der üblichen Sichtweise der Objektorientierung, wo Daten und Operationen auf diese Daten zu einer Einheit (eben einem Objekt) zusammengefasst werden
  - Komplexe Geschäftsprozesse arbeiten in der Praxis oft mit den Daten von vielen Objekten, d.h. ihre Funktionalität ist objektübergreifend
  - Es ist daher besser, die Funktionalität eines Geschäftsprozesses in einer Klasse (bzw. einer Komponente) zu kapseln, anstatt sie über viele kleine Operationen in unterschiedlichen Klassen zu verteilen.

LE 29
12

3.9.2 *Server*basierte Komponenten-Modelle

- Sicherheit
  - Benutzerverwaltung & Rollen
    - Bei der Entwicklung einer Komponente werden Rollen definiert
    - Diesen Rollen werden Ausführungsrechte für bestimmte Operationen eingeräumt
    - Beim Aufruf einer Operation identifiziert das Laufzeitsystem der Komponenten-Plattform den initiierenden Benutzer und prüft, ob er zu einer Rolle gehört, die den Aufruf durchführen darf
    - Die Zuordnung von Benutzern zu Rollen ist Aufgabe des System-Administrators.

LE 29
13

3.9.2 *Server*basierte Komponenten-Modelle

- Transaktion
  - Eine Folge von Aktionen zur kontrollierten Veränderung eines Datenbestandes
  - ACID: Eigenschaften einer Transaktion
    - unteilbar *(atomic)*
      - Entweder alle Aktionen oder keine Aktion
    - konsistent *(consistent)*
      - Vor Beginn und nach ihrer Beendigung befindet sich der Datenbestand in einem konsistenten Zustand
    - isoliert *(isolated)*
      - Inkonsistente Zustände während der Abarbeitung einer Transaktion sind für andere Transaktionen unsichtbar
    - dauerhaft *(durable)*
      - Die Änderungen, die eine Transaktion an einem Datenbestand vornimmt, werden persistent gespeichert.

LE 29
14

3.9.2 *Server*basierte Komponenten-Modelle

- Transaktions-Monitor (TAM)
  - Zur Abwicklung und Verwaltung von Transaktionen
  - Teil vieler Datenbank-Management-Systeme (DBMS)
- Verteilte Transaktionen
  - Der zu ändernde Datenbestand oder die Abarbeitung der einzelnen Aktionen erfolgt über mehrere CS verteilt
  - Wird auf einem CS initiiert *(master)* und von dort werden einzelne Teilaufgaben auf andere CS *(slaves)* verteilt, die dort als Teil-Transaktionen ablaufen
  - Zur Koordination: 2-Phasen-*Commit*-Protokoll
    - Alle Teil-Transaktionen müssen erfolgreich sein.

LE 29
15

3.9.2 *Server*basierte Komponenten-Modelle

- Komponenten-Modelle
  - Erlauben es, das Transaktionsverhalten unabhängig von der Implementierung einer Komponente zu spezifizieren
  - Der Komponenten-Entwickler implementiert die Operationen einer Komponente und braucht sich um Transaktionen nicht zu kümmern
  - Vereinfacht die Implementierung der Anwendung
- Installation
  - Erst bei der Installation der Komponente wird festgelegt, welche Operationen/Geschäftsprozesse durch Transaktionen zu sichern sind
  - Laufzeitsystem: Startet automatisch eine Transaktion.

LE 29
16

3.9.2 *Server*basierte Komponenten-Modelle

- Ortstransparenz
  - Der Zugriff auf Objekte der Anwendungsschicht ist ortstransparent
    - Ein *Client* muss den physischen Standort eines *Server*-Objekts *nicht* kennen
- Namensdienst
  - Eine Komponenten-Modell spezifiziert einen Namensdienst, der die Verbindung zu entfernten Objekten herstellt
  - Dieser Dienst liefert einem *Client* eine Referenz zu einem bestimmten Objekt oder legt ein Objekt einer bestimmten Komponente »irgendwo« an und liefert eine Referenz darauf.

LE 29
17

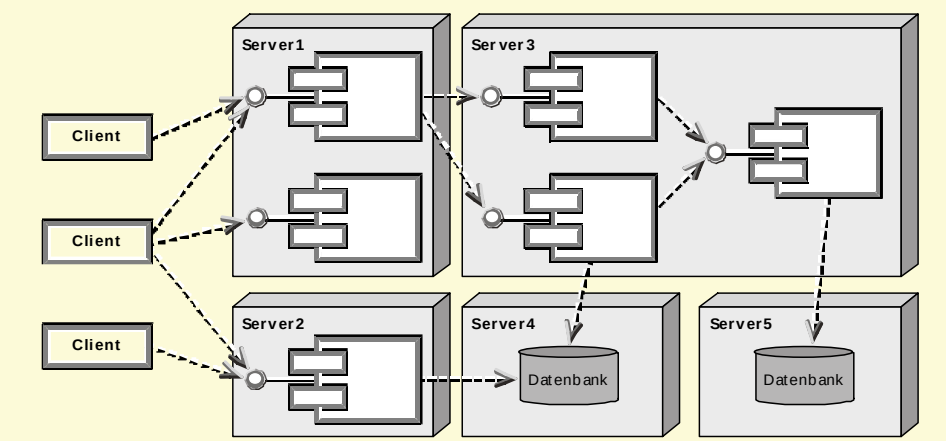
3.9.2 *Server*basierte Komponenten-Modelle

- Ortstransparenz: Vorteile auf 2 Ebenen
  - Implementierung
    - Entfernte Objekte können im Wesentlichen genau so behandelt werden wie lokale Objekte
    - Die Bibliotheken und das Laufzeitsystem sorgen dafür, dass Operationsaufrufe, die über ein Netzwerk übertragen werden müssen, auf der *Client*-Seite genau so durchgeführt werden können, wie Aufrufe eines lokalen Objekts
  - Administration
    - Da der Aufenthaltsort einer Komponente transparent ist, kann sie leicht von einem *Server* auf einen anderen verschoben werden, ohne dass Veränderungen an den *Clients* nötig sind.

LE 29
18

3.9.2 *Server*basierte Komponenten-Modelle

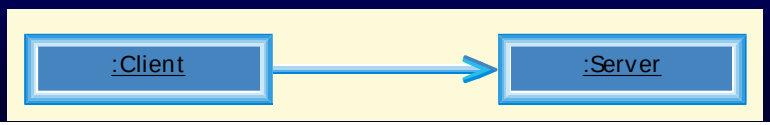
- Lastverteilung durch Verteilung von Anwendungs-Komponenten auf mehrere *Server* (→Ereignisse)



LE 29
19

Grundkonzept *server*basierter Komponenten

- Referenzen
  - Konzeptionell verfügt ein *Client*-Objekt über eine Referenz auf ein *Server*-Objekt, wenn es eine Dienstleistung in Anspruch nehmen möchte
  - Diese Referenz erhält es als Ergebnis eines Operationsaufrufes von einem anderen Objekt oder durch den Namensdienst der Komponenten-Plattform
  - Da *Client* und *Server* nicht auf demselben Computersystem liegen müssen, ist im Allgemeinen eine Abbildung von Referenzen auf Speicheradressen nicht möglich



LE 29
20

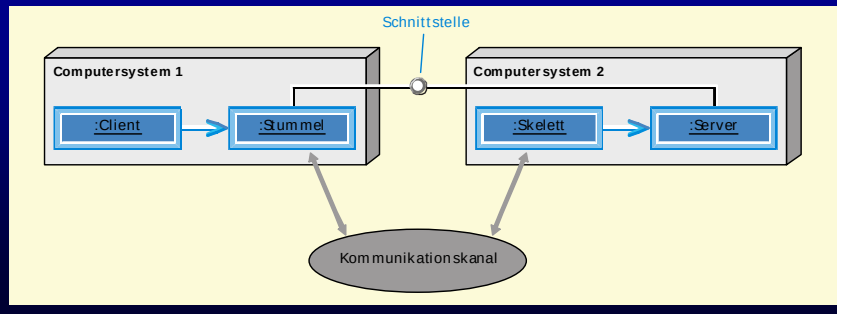
Grundkonzept *server*basierter Komponenten

- Stummel- und Skelett-Objekte (Animation)
  - Statt auf das entfernte Objekt verweist eine Referenz auf ein lokales Stummel-Objekt
  - Das Stummel-Objekt bietet die gleiche Schnittstelle wie das *Server*-Objekt
  - Ruft der *Client* eine Operation auf, verpackt das Stummel-Objekt die Daten des Aufrufs in ein Transportformat und schickt sie über einen Kommunikationsdienst an das CS, auf dem das *Server*-Objekt liegt
  - Dort werden die Daten von einem Skelett-Objekt empfangen und entpackt
  - Das Skelett-Objekt hat eine lokale Referenz auf das *Server*-Objekt und initiiert einen lokalen Aufruf.

LE 29
21

Grundkonzept *server*basierter Komponenten

- Ausgabe-Parameter werden analog übertragen
  - Stummel- und Skelett-Objekt tauschen beim Verpacken und Entpacken der Daten ihre Rollen
- Die Verwendung von Stummel- und Skelett-Objekten kostet natürlich Zeit
  - Jedoch meist gering im Vergleich zum Datentransport über ein Netzwerk



LE 29
22

3.9.2 *Server*basierte Komponenten-Modelle

- Ereignisse in einer verteilten Umgebung
  - Die Benachrichtigung über Ereignisse erfolgt oft über ein Netzwerk
  - Ein Komponenten-Modell spezifiziert Dienste, die die Verteilung von Ereignissen und das An- & Ab-melden von Ereignis-Empfängern zum Inhalt haben
  - Auch hier soll wieder ein möglichst hohes Abstraktionsniveau erreicht werden
  - Die konkrete Realisierung der Benachrichtigung braucht den Komponenten-Entwickler nicht zu interessieren, egal ob er eine Quelle oder einen Empfänger entwickelt
  - Die Komponenten-Plattform stellt die technische Infrastruktur bereit.

LE 29
23

3.9.2 *Server*basierte Komponenten-Modelle

- Asynchrone Botschaften
  - Synchrone Kommunikation als Standardlösung
  - Aber auch asynchrone Operationsaufrufe
    - Insbesondere in Verbindung mit Ereignissen
    - Ereignis-Empfänger sollen nur informiert werden, können die weitere Arbeit der Ereignis-Quelle jedoch *nicht* beeinflussen
- *Messaging*-Systeme
  - Für die Übermittlung asynchroner Botschaften
  - Bieten eine Reihe von Zusatzdiensten an
    - Zwischenspeicherung von Botschaften
    - Die Botschaft kann an einen anderen *Server* geleitet werden, der sie auch bearbeiten kann.

LE 29
24

3.9.3 *Java-RMI*

- *RMI*-Konzept *(remote method invocation)*
  - Einfache Infrastruktur zur Realisierung verteilter Anwendungen
  - Es handelt sich um mehrere Pakete des *Java Standard*-APIs
  - Dem Konzept liegt kein explizites Komponenten-Modell zu Grunde
  - Ermöglicht lediglich den Aufruf von Operationen für Objekte auf einem anderen Computersystem
    - Ereignisse, Transaktionen oder eine Benutzerverwaltung sind in RMI nicht vorhanden
  - Entfernte Objekte werden immer über Operationen der Schnittstellen angesprochen.

LE 29
25

3.9.3 *Java-RMI*

- **Beispiel: Klasse Seminartyp**
  - Zunächst wird eine Schnittstelle spezifiziert:

```
import java.rmi.* ;
interface Seminartyp extends Remote
{ String getKurztitel() throws RemoteException;}
```

  - Schnittstelle wird von einer Klasse implementiert:

```
import java.rmi.* ;
import java.rmi.server.* ;
public class SeminartypImpl extends UnicastRemoteObject
   implements Seminartyp
{ private String Kurztitel;
  public SeminartypImpl() throws RemoteException
  { //nichts zu tun   }
  public String getKurztitel() //Operation
  { return this.Kurztitel; }
}.
```


LE 29
26

3.9.3 *Java-RMI*

- Anschl. muss eine *Server*-Anwendung implementiert werden, die ein Objekt der Klasse `SeminartypImpl` erzeugt und anderen CS zur Verfügung stellen kann
- *Server*-Anwendung nutzt den RMI-Namensdienst um das erzeugte Objekt unter dem Namen »`einSeminartyp`« verfügbar zu machen

```
import java.rmi.* ;
public class SemOrgServer
{ public static void main( String[] args )
{System.setSecurityManager(new RMISecurityManager());
 try
 {SeminartypImpl einSeminartyp = new SeminartypImpl();
 Naming.rebind("einSeminartyp", einSeminartyp); }
 catch( Exception e ) {…} }}.
```


LE 29
27

3.9.3 *Java-RMI*

- Übersetzung der Quelltexte mit dem Java-Compiler
- Das Programm `rmic` generiert Stummel- und Skelett-Klassen für die Schnittstelle `Seminartyp`
- Aufruf: `rmic SeminartypImpl.class`
  - `rmic`: `Seminartyp_stub` & `Seminartyp_skel`
- Ein *Client* kann die Bezeichnung des Seminartyp-Objekts ermitteln:

```
import java.rmi.* ;
public class SeminartypClient
{ public static void main(String[] args)
  {Seminartyp einSeminartyp = (Seminartyp)
   Naming.lookup("einSeminartyp");
   String dieBezeichnung =
   einSeminartyp.getKurztitel();
   System.out.println(dieBezeichnung);} }.
```


LE 29
28

3.9.4 *Enterprise JavaBeans*

- ***Enterprise JavaBeans*** **(EJBs)**
  - Ein auf Java basierendes *server*seitiges Komponenten-Modell
  - Teil der *Java 2 Platform Enterprise Edition* (J2EE)
  - Es handelt sich bei EJBs um eine Spezifikation, nicht um ein konkretes Produkt
  - Inzwischen sind jedoch verschiedene Plattformen für EJBs erhältlich
  - Die EJB-Spezifikation unterscheidet
    - *Entity beans:* Enthalten persistente Daten
      - Entsprechen den beschriebenen Daten-Komponenten
    - *Session beans:* Meist transient
      - Sie entsprechen weitgehend den Prozess-Komponenten.

LE 29
29

3.9.4.1 Aufbau von EJBs

- EJB ist eine Komponente, die aus mehreren Teilen zusammengesetzt sein muss:
  - Aufruf-Schnittstelle *(remote interface)*
    - Schnittstelle, die die Dienstleistungen einer EJB für *Clients* beschreibt
    - Gewöhnliche Java-Schnittstelle, die von der Schnittstelle `javax.ejb.EJBObject` erbt
    - Alle in EJB-Schnittstellen deklarierten Operationen müssen die Ausnahme `java.rmi.RemoteException` auslösen können
    - Attribute werden durch ein Paar von `get-set`-Operationen dargestellt
    - Eine EJB kann auch mehrere Aufruf-Schnittstellen spezifizieren.

LE 29
30

3.9.4.1 Aufbau von EJBs

- Verwaltungs-Schnittstelle *(home interface)*
  - Bietet Operationen zum Erzeugen neuer EJB-Objekte und zum Auffinden vorhandener Objekte
  - Muss von `java.ejb.EJBHome` erben
  - Operationen `create` und `findByPrimaryKey`
- *Bean*-Klasse
  - Enthält Implementierung der Operationen
  - Implementiert Schnittstelle `javax.ejb.EntityBean` bzw. `javax.ejb.SessionBean`
- Auslieferungs-Beschreibung *(deployment descriptor)*
  - Eine EJB enthält zusätzlich eine Beschreibungs-Datei im XML-Format.

LE 29
31

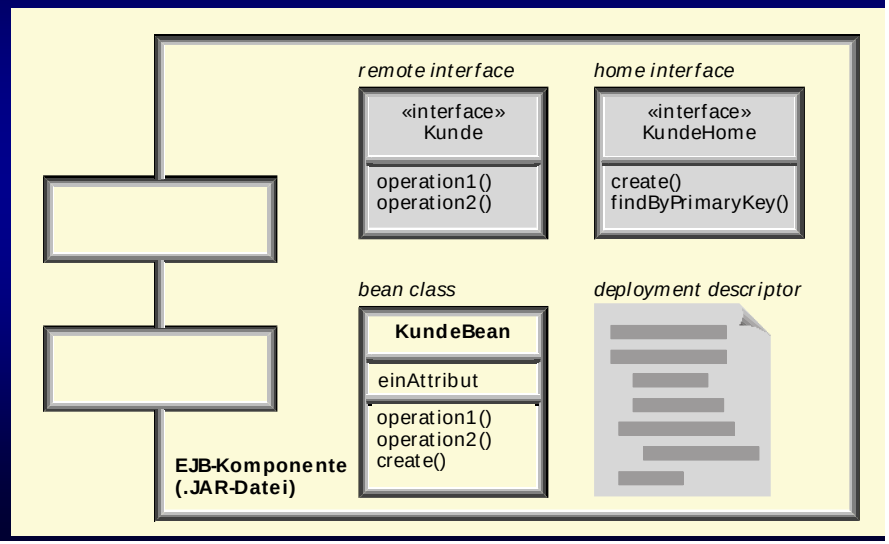
3.9.4.1 Aufbau von EJBs

- Dient zur Ermittlung benötigter Informationen bei der Installation einer EJB
- Enthält Informationen über die Schnittstellen, Attribute, Operationen, Rollen, Transaktionsverhalten
- Einfache EJB
  - Besteht aus 4 Dateien
  - Die beiden Schnittstellen und die *Bean*-Klasse werden vor der Auslieferung übersetzt
  - Die 3 `class`-Dateien und die XML-Datei werden in ein Java-Archiv (JAR-Datei) verpackt
  - Die JAR-Datei enthält alle Informationen, die zur Installation und zum Betrieb der EJB erforderlich sind.

LE 29
32

3.9.4.1 Aufbau von EJBs

- Aufbau einer EJB



LE 29
33

3.9.4.1 Aufbau von EJBs

- **Nutzung einer EJB**
  - Ein *Client* muss in 2 Schritten vorgehen, um die Dienste einer EJB nutzen zu können:
    1. Zunächst besorgt er sich vom Laufzeitsystem eine Referenz auf eine Verwaltungsschnittstelle
    2. Mit den Operationen dieser Schnittstelle kann er ein neues EJB-Objekt erzeugen oder ein bereits vorhandenes Objekt finden
  - Beispiel
    - Realisierung des Geschäftsprozess »Buchung« durch ein *session bean*
    - Im Paket `SemOrg.Schnittstellen` sind auch Schnittstellen für die Klassen Kunde und Seminartyp deklariert.

LE 29
34

3.9.4.1 Aufbau von EJBs

```
//Das Remote-Interface der EJB
package SemOrg.Schnittstellen
import java.rmi.*; import javax.ejb.* ;
public interface Buchung extends EJBObject
{ public void buchen(Kunde k, Seminartyp s)
        throws RemoteException
}
//Das Home-Interface der EJB
package SemOrg.Schnittstellen;
import java.rmi.* ; import javax.ejb.* ;

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


LE 29
35

3.9.4.1 Aufbau von EJBs

```
//Die Bean-Klasse
package SemOrg.Server;
import javax.ejb.* ; import javax.naming.* ;
import java.rmi.* ;
public class BuchungBean implements SessionBean
{ public void ejbCreate() throws RemoteException,
   CreateException
  { //Initialisierung des Objekts    }
  //Operationen der Schnittstelle SessionBean
  public void ejbRemove()
  { //Das Objekt wird gelöscht
    /*hier werden Aufräumarbeiten durchgeführt*/  }
  public void setSessionContext(SessionContext ctx)
  {//In diesem Beispiel wird der Session-Context
   /*nicht benötigt. Hier ist also nichts zu tun*/}.
```


LE 29
36

3.9.4.1 Aufbau von EJBs

```
  public void ejbActivate()
  { //für zustandslose session beans ist hier nichts
    //zu tun, da sie nicht aktiviert oder
    //passiviert werden
  }
  public void ejbPassivate()
  {//für zustandslose session beans ist hier nichts
   //zu tun,da sie nicht aktiviert oder
   //passiviert werden
  }
  //Operation der Remote-Schnittstelle (Buchung)
  public void buchen(Kunde k, Seminartyp s)
        throws RemoteException
  {//alle Schritte zur Durchführung einer Buchung
  }
}.
```

LE 29
37

3.9.4.1 Aufbau von EJBs

```
<!--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>
<assembly-description>
    <container-transaction>
        <method> <ejb-name>Buchung</ejb-name>
        <method-name>*</method-name> </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-description>.
```


LE 29
38

3.9.4.1 Aufbau von EJBs

```
//Ein einfacher Client
package SemOrg.Client;
import java.rmi.RemoteException;
import javax.naming.Context ;
import javax.naming.InitialContext ;
import SemOrg.Schnittstellen.*;
public class Client
{ public static void main(String[] args)
  { Client einClient = new Client();
    einClient.start();
  }
  public void start()
  { try
    { Kunde einKunde; Seminar einSeminar;
      //Referenzen für einKunde und einSeminartyp
      //initialisieren.
```


LE 29
39

3.9.4.1 Aufbau von EJBs

```
        Context ctx = new InitialContext();
        Object temp =
           ctx.lookup("java:comp/env/Buchung");
        BuchungHome home =(BuchungHome)
             javax.rmi.PortableRemoteObject.
                 narrow(temp, BuchungHome.class)
        Buchung bean = home.create();
        bean.buchen(einKunde, einSeminartyp);
        bean.remove();
    }
    catch(Exception e)
    { System.out.println(e);
    }
  }
}.
```


LE 29
40

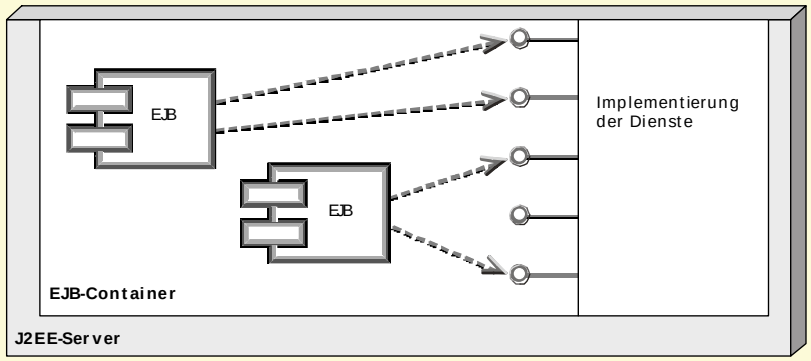
3.9.4.2 Architektur von EJB-Plattformen

- **EJB-Container**
  - **Grundlage ist ein J2EE-*Server***
  - **In einem EJB-Container können mehrere EJBs ablaufen**
  - **Der Container stellt den EJBs genau spezifizierte Dienste in Form von Schnittstellen zur Verfügung:**
    - **Namensdienst (JNDI = *Java Name and Directory Interface)***
    - **Transaktions-Monitor (JTA = *Java Transaction* API)**
    - **Schnittstellen für den Zugriff auf Datenbanken (JDBC = *Java DataBase Connectivity)***
    - ***Java Mail API* zum Versenden von *e-mail***
    - **Standard-Java-API.**

LE 29
41

3.9.4.2 Architektur von EJB-Plattformen

- Architektur einer EJB-Plattform



LE 29
42

3.9.4.2 Architektur von EJB-Plattformen

- Installation einer EJB in einem Container
  - Ein Werkzeug analysiert die EJB und generiert daraus 2 neue Klassen
    - die `EJBHome`-Klasse
      - implementiert die Verwaltungs-Schnittstelle *(home interface)*
    - die `EJBRemote`-Klasse
      - implementiert die Aufruf-Schnittstelle *(remote interface)*
    - Diese beiden Klassen sind praktisch Skelett-Klassen für die jeweilige Schnittstelle
    - Sie stellen die Verbindung zwischen den Schnittstellen der EJB und der *Bean*-Klasse her
    - Zur Laufzeit gehören zu jedem EJB-Objekt also 2 Skelett-Objekte.

LE 29
43

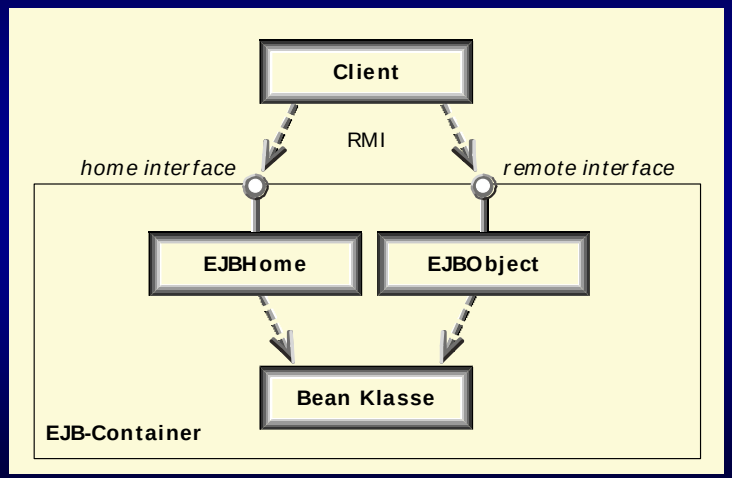
3.9.4.2 Architektur von EJB-Plattformen

- Ein *Client*, der ja nur die Schnittstellen kennt, ruft immer Operationen der Skelett-Objekte auf
- Diese leiten den Aufruf nach diversen Prüfungen und Verwaltungsarbeiten an die eigentliche Implementierung in der *Bean*-Klasse weiter
- Ein EJB ist also für *Clients* nie direkt zugreifbar
- Wie die Skelett-Klassen konkret aussehen, wird nicht in der EJB-Spezifikation festgelegt
- Sie arbeiten sehr eng mit dem EJB-Container zusammen, sodass es dem Container-Hersteller überlassen bleibt, wie diese Klassen aussehen
- Auf die Entwicklung von EJBs hat dies keinen Einfluss, da jeder EJB-Container-Hersteller ein Installations-Werkzeug zur Generierung der Skelett-Klassen liefern muss.

LE 29
44

3.9.4.2 Architektur von EJB-Plattformen

- EJB-Container



LE 29
45

3.9.4.2 Architektur von EJB-Plattformen

- EJBs kommunizieren immer über ihre Skelett-Objekte mit der Außenwelt, auch 2 EJB-Objekte im selben Container können nicht direkt aufeinander zugreifen
  - Dadurch wird eine hohe Ortstransparenz und somit Skalierbarkeit erreicht
- Die Kommunikation zwischen *Clients* und EJB-Objekten (ein *Client* kann natürlich ein anderes EJB-Objekt sein) wird über Java-RMI abgewickelt
- Alternativ ist für EJBs eine Kommunikation über das IIOP *(Internet Inter ORB Protocol)* der CORBA-Spezifikation vorgesehen.

LE 29
46

3.9.4.2 Architektur von EJB-Plattformen

- Persistente Speicherung von Daten
  - Für entity-EJB (Daten-Komponenten)
    - 2 Möglichkeiten, wie die Daten (Attribute der *Bean*-Klasse) dauerhaft gespeichert werden:
    - container managed persistence
      - EJB-Container ist dafür verantwortlich, die Daten zu speichern
      - Eine EJB wird auf eine Tabelle in einer RDB abgebildet
      - Der EJB-Container liest die Attribute eines *Bean*-Objekts aus und speichert sie als Datensatz in der Tabelle
      - Umgekehrt kann er ein neues EJB-Objekt erzeugen, die Attribute aus der Datenbank auslesen und setzen
      - EJB-Entwickler muss sich nicht um die dauerhafte Speicherung der Daten kümmern
      - Ein EJB-Objekt ist einfach persistent.

LE 29
47

3.9.4.2 Architektur von EJB-Plattformen

- bean managed persistence
  - Ein EJB-Objekt ist selbst dafür verantwortlich, seine Daten auf Anforderung dauerhaft zu speichern
- Ein entity-EJB gibt im deployment descriptor an, welche Art der Persistenz es unterstützt

- Auffinden von EJB-Objekten
  - Zum Auffinden von vorhandenen EJB-Objekten dienen Primärschlüssel
    - Setzt sich aus einer Teilmenge ihrer Attribute zusammen, die ein Objekt eindeutig identifizieren
    - Für den Primärschlüssel muss ein *entity*-EJB eine zusätzliche Klasse bereitstellen
    - Mit Hilfe eines Primärschlüssel-Objekts kann ein *Client* ein bestimmtes EJB-Objekt im Netz finden.

LE 29
48

3.9.4.3 Das EJB-Rollenkonzept

- Rollen
  - *Server*-Anbieter *(Server-Provider)*
    - Stellt die Plattform zur Verfügung
  - Container-Anbieter *(Container-Provider)*
    - Setzt auf der Plattform des *Server*-Anbieters auf
    - Benutzerverwaltung, Transaktions-Management, Persistenz von EJBs und Werkzeuge für die Installation von Komponenten
    - *Server* & Container oft vom selben Hersteller
  - Komponenten-Entwickler *(Bean-Provider)*
    - Realisiert Anwendungs- oder Geschäftslogik
    - Im einfachsten Fall entwickelt er eine transiente Anwendung und überlässt die Datenspeicherung komplett dem Container.

3.9.4.3 Das EJB-Rollenkonzept

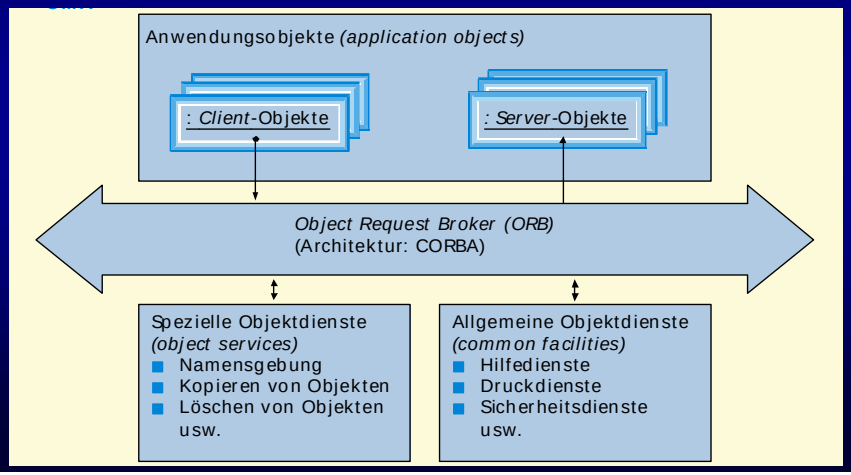
- **Monteur** *(Application-Assembler)*
  - Hat die Aufgabe, aus einzelnen Komponenten eine Anwendung zu machen
  - Komponenten-Entwickler und Monteur müssen eng zusammenarbeiten
- **Installations-Entwickler** *(Bean-Deployer)*
  - Muss die aus einzelnen Komponenten bestehende Anwendung auf dem Zielsystem installieren
- **Systemadministrator**
  - Überwacht das laufende System.

3.9.5 CORBA

- OMG *(Object Management Group)*
  - Entwickelt Standards und Spezifikationen für eine Infrastruktur, die für verteilte, objektorientierte Anwendungen erforderlich ist
  - Setzt den **de facto**-Standard für die Software-Industrie
  - Deckt heute einen großen Teil der objektorientierten Software-Entwicklung ab, z.B. ist die **UML** OMG-Standard
- CORBA
  - CORBA ist eine **Spezifikation** und **kein** Produkt
  - Es gibt jedoch eine Reihe von kommerziellen und frei verfügbaren Plattformen, die der CORBA-Spezifikation entsprechen.

3.9.5 CORBA

- OMA
  - Basis für die Standardisierung bildet die **Object Management Architecture** (OMA)



3.9.5 CORBA

- Hauptkomponenten von OMA
  - Anwendungsobjekte *(application objects)*
    - Können *Client*- als auch *Server*-Objekte sein
  - ***Object Request Broker*** (ORB)
    - Vermittelt die Kommunikation zwischen den Objekten
  - Spezielle Objektdienste *(object services)*
    - Benötigt der ORB zur Erfüllung seiner Aufgaben
    - Beispiel: Dienste für die Namensgebung, zum Kopieren und Löschen von Objekten
  - Allgemeine Objektdienste *(common facilities)*
    - Beispiel: Hilfe-, Druck-und Sicherheitsdienste.

LE 29
53

3.9.5 CORBA

- *Object Request Broker* (ORB)
  - Schlüsselkomponente der Architektur
  - Kommunikationszentrale
  - Übermittlung von Operationsaufrufen und Ergebnissen zwischen Anwender und Anbieter
  - Der ORB wurde als erste Komponente unter dem Namen CORBA *(Common Object Request Broker Architecture)* standardisiert
  - Der ORB benutzt die Schnittstellen der *Client*- und *Server*-Objekte, um Anforderungen weiterzuleiten
  - Als Schnittstellen-Definitionssprache wird CORBA-IDL *(interface definition language)* verwendet.

LE 29
54

3.9.5 CORBA

- CORBA & Komponenten
  - Auf CORBA-Objekte kann von außen nur über ihre Schnittstelle zugegriffen werden
  - Sie stellen jedoch keine Komponenten dar
  - CORBA ist bis zur Version 2 kein Komponenten-Modell
  - Da CORBA die Implementierung nicht berücksichtigt, werden z.B. keine Aussagen darüber gemacht, wie CORBA-Objekte auf binärer Ebene aussehen, wie sie zu installieren oder zu verteilen sind.

LE 29
55

3.9.5 CORBA

- *CORBA components* (CCM, *corba components model)*
  - Haben von der Architektur her große Ähnlichkeit mit *Enterprise JavaBeans*
  - Ziel war es, vorhandene EJBs als CORBA-Komponenten nutzen zu können
  - Von diesem Standpunkt aus können CORBA-Komponenten als sprachunabhängige Obermenge von EJBs betrachtet werden.
- CORBA vs. COM
  - Die Spezifikation sieht eine Abbildung vor, die das Zusammenarbeiten zwischen COM und CORBA-Objekten grundsätzlich ermöglicht.

LE 29
56

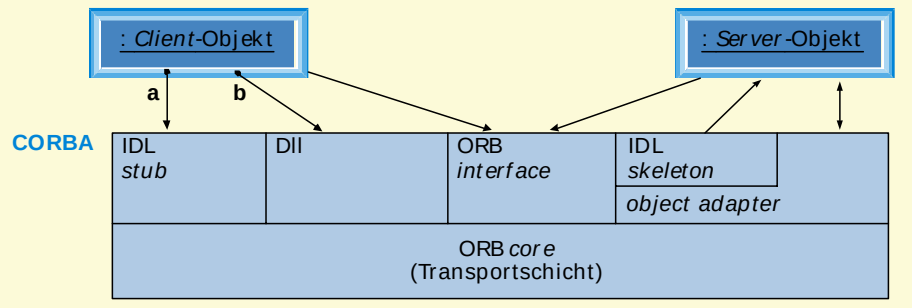
3.9.5 CORBA

- Quality of Services (QoS)
  - CORBA-Einsatz in speziellen Umgebungen:
  - Minimum-CORBA
    - Variante, die für Systeme mit begrenzten Hardware-Ressourcen (Rechenleistung, Hauptspeicher) entwickelt wurde
    - Zielplattformen sind z.B. Handys oder PDAs
  - Echtzeit-CORBA
    - Definiert, wie Echtzeitanforderungen zu handhaben sind
  - Fehlertolerantes CORBA
    - Aussagen zu fehlertoleranten Systemen.

LE 29
57

3.9.5.1 Die Architektur von CORBA

- ORB-Kern: Transportschicht
- *Server*
  - Operationen müssen in einer Schnittstelle, hier IDL, beschrieben werden



LE 29
58

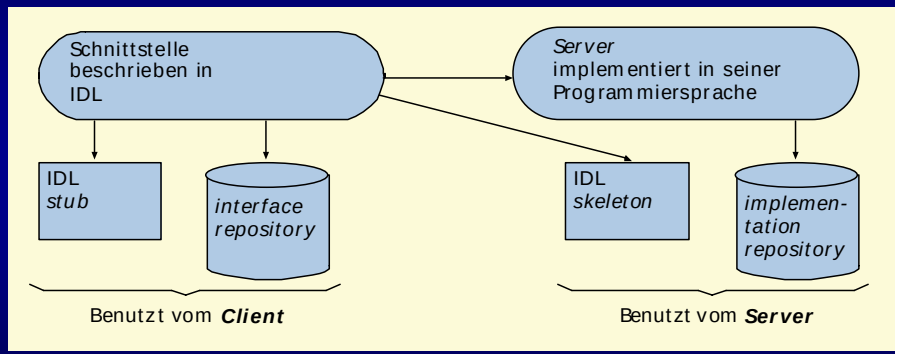
3.9.5.1 Die Architektur von CORBA

- Aus einer IDL-Beschreibung werden generiert:
  - IDL-Stummel *(stub)*:
    - Wird vom *Client* benutzt, um Dienstleistungen des *Servers* anzufordern
  - *Interface Repository:*
    - Speichert Informationen über die Schnittstelle, die zur Laufzeit von den *Clients* abgefragt werden
      - Dazu wird das DII *(dynamic invocation interface)* benutzt
  - IDL-Skelett *(skeleton)*:
    - Code-Rahmen
      - Wird vom Programmierer mit der Implementierung der Operationen gefüllt
  - *Implementation Repository:*
    - Informationen, um *Server* zu lokalisieren.

LE 29
59

3.9.5.1 Die Architektur von CORBA

- Aus der IDL-Schnittstelle abgeleitete Informationen



LE 29
60

3.9.5.1 Die Architektur von CORBA

- *Clients*
  - Interagieren mit dem ORB über IDL-Stummel und DII jeweils in der eigenen Programmiersprache
  - Können auf 2 Arten Dienstleistungen anfordern:
    - Ist die Schnittstelle in IDL definiert und kennt der *Client* die Typ- bzw. Klassendefinition des *Servers*, dann kann eine statische Anforderung erfolgen
    - Ist die Schnittstelle des *Servers* nicht bekannt, dann kann die DII verwendet werden
      - Die Anforderung wird von der DII-Komponente durchgeführt, die dazu Informationen aus dem *interface repository verwendet*.

LE 29
61

3.9.5.1 Die Architektur von CORBA

- *Server*
  - Der *Server* weiß nicht, ob eine Anforderung statisch oder dynamisch erstellt wurde
  - Anforderungen passieren den IDL-Stummel, um zum *Server* zu gelangen
  - Ein IDL-Skelett hängt i.d.R. vom Objekt-Adapter ab
  - Es kann mehrere Skelette für dieselbe Schnittstelle und dieselbe Programmiersprache mit unterschiedlichen Objekt-Adaptern geben
  - Ein Objekt-Adapter stellt eine Schnittstelle zu den ORB-Dienstleistungen zur Verfügung, die die *Server* nutzen können.

LE 29
62

3.9.5.1 Die Architektur von CORBA

- ORB-*Interface*
  - Andere Dienstleistungen, die von den *Servern* benutzt werden, werden direkt von der Komponente ORB-*Interface* zur Verfügung gestellt
  - Diese Schnittstelle ist für alle ORB-Implementierungen identisch und hängt nicht vom Objekt-Adapter ab
  - Das ORB-*Interface* wird auch von *Clients* genutzt.

LE 29
63

3.9.5.1 Die Architektur von CORBA

- *Interface Repository* (IR)
  - Verwaltet die IDL-Definitionen
    - Diese sind zur Laufzeit verfügbar
  - Die DII benutzt diese Informationen, um *Clients* Anforderungen an *Server* zu erlauben, deren Schnittstelle zur Übersetzungszeit des *Clients* noch unbekannt war
  - Die IR-Dienstleistung wird auch vom ORB selbst benutzt, um Anforderungen weiterzuleiten
- *Implementation Repository*
  - Verwaltet Informationen, die der ORB benötigt, um die *Server* zu lokalisieren und zu starten
  - Es ist spezifisch für eine Betriebssystem-Umgebung.

LE 29
64

3.9.5.2 Die Schnittstellendefinitionssprache IDL

- Schnittstellendefinitionssprache IDL
  - Dient zur Beschreibung der Schnittstellen von Klassen, die als *Server* Dienstleistungen anbieten
  - Nur Spezifikationen beschreibbar, kein Code
  - Die Spezifikationen werden verwendet, um vom *Client* aus Operationen des *Servers* aufzurufen
    - Aufruf in der *Client*-Programmiersprache
  - Die Implementierung der Operationen auf der *Server*-Seite in der *Server*-Programmiersprache
  - IDL-*Precompiler* transformieren die IDL-Spezifikationen in die jeweiligen Zielsprachen
    - Syntax ist an C++ angelehnt
    - Grammatik ist Teilmenge der C++-Grammatik mit Erweiterungen für verteilte Aufrufe.

LE 29
65

3.9.5.2 Die Schnittstellendefinitionssprache IDL

- Vergleich IDL vs. ODL
  - ODL wurde auf der Grundlage von IDL entwickelt
  - IDL = ODL ohne *extents*, *keys* und *relationships*
- CORBA und Java
  - Es gibt Spezifikation für Abb. von IDL auf Java
  - Java-Schnittstellen (Schlüsselwort: `interface`) werden direkt genutzt
  - Die CORBA-Spezifikation sieht eine Umkehrabbildung *(reverse mapping)* vor
    - Damit lässt sich aus Java-Schnittstellen IDL-Code generieren
    - Somit ist es leicht möglich, über CORBA EJBs von *Clients* anzusprechen, die in anderen Programmiersprachen entwickelt wurden.

LE 29
66

3.9.5.3 Standardisierte CORBA-Dienste

- CORBA definiert 2 Kategorien von Diensten
  - Objektdienste *(object services)*
    - Gegenstück zum API eines Betriebssystem
    - In IDL spezifiziert und werden vom Plattform-Entwickler implementiert
  - Allgemeine Dienste *(common facilities)*
    - Höherwertige Dienste, die für viele Anwendungsbereiche relevant sind
    - Vergleichbar mit grundlegenden Klassenbibliotheken, die allgemein benötigte Funktionalitäten modular zur Verfügung stellen.

LE 29
67

3.9.5.3 Standardisierte CORBA-Dienste

- Standardisierte Objektdienste
  - Namensdienst *(name service)*
    - Kreierung von Namensräumen und die Abbildung von benutzerdefinierten Objektnamen auf CORBA-Objektreferenzen
  - Lebenszyklusdienst *(lifecycle service)*
    - Verwaltung von Objekten (Erzeugen, Kopieren, Löschen, Abfragen auf Äquivalenz)
  - Ereignismeldedienst *(event notification)*
    - Erkennen von Ereignissen und Benachrichtigung von Objekten über Ereignisse
  - Persistenzdienst *(persistence service)*
    - Dauerhaftes Speichern und Laden von Objekten auf / von (externen) Speichern.

LE 29
68

3.9.5.3 Standardisierte CORBA-Dienste

- Nebenläufiger Dienst *(concurrency service)*
  - Synchronisierung konkurrierender Zugriffe auf Objekte
- Auslagerungsdienst *(externalization service)*
  - Datenexport von Objekten in Dateien außerhalb des Objektsystems
- Beziehungsdienst *(relationship service)*
  - Erzeugen, Löschen und Verwalten von Beziehungen zwischen Objekten, Navigation über Beziehungen
- Transaktionsdienst *(transaction service)*
  - Techniken zur Durchführung transaktionsgesteuerter Abläufe mit einem zweistufigen *commit*.

LE 29
69

3.9.5.3 Standardisierte CORBA-Dienste

- **Zeitdienst** *(time service)*
  - Synchronisierung der Uhren von Komponentensystemen in verteilter Umgebung
- **Sicherheitsdienst** *(security service)*
  - Autorisierungs- und Überwachungsfunktionen auf Objektebene
- **Lizenzdienst** *(licensing service)*
  - Verwaltung von Objektlizenzen, Abrechnung von Gebrauchsgebühren für Objekte
- **Eigenschaftsdienst** *(properties service)*
  - Dynamisches Erzeugen, Löschen und Verwenden von Attributen
- **Anfragedienst** *(query service)*
  - SQL-artige Anfrageoperationen für Objekte.

LE 29
70

3.9.5.3 Standardisierte CORBA-Dienste

- Allgemeine Objektdienste *(common facilities)*
  - **Vertikale Marktdienste**
    - Basisfunktionalität für unterschiedliche Marktsegmente
  - **Horizontale allgemeine Dienste**
    - Dienste über mehrere Anwendungsbereiche
      - *user interface* definiert den Zugriff auf ein grafisch orientiertes Informationssystem
      - *information management* legt Modellierung, Definition, Speicherung, Wiedergewinnung, Verwaltung und Austausch von Informationen fest
      - *systems management* unterstützt die Verwaltung komplexer Informationssysteme
      - *task management* erlaubt die Automatisierung von Aufgaben.

LE 29
71

3.9.5.4 Entw. von CORBA-Anwendungen in Java

1 Spezifikation der Schnittstelle in CORBA-IDL

- Beispiel
  - Ausschnitte der Klasse `Firma` aus der Fallstudie »Seminarorganisation«

```
//SeminarOrganisation.idl
module SeminarOrganisation
{   //Schnittstelle der Klasse Firma
    interface Firma
    {
        attribute string Name;
        attribute float Umsatz;
        //Operationssignatur
        float berechneGewinn(in float Kosten);
    };
};.
```


LE 29
72

3.9.5.4 Entw. von CORBA-Anwendungen in Java

2 Übersetzung der IDL-Spezifikation

- Die Datei `SeminarOrganisation.idl` kann mit dem IDL-Compiler übersetzt werden
- Der IDL-Compiler erzeugt eine Reihe von Dateien
  - Aufruf: `idl SeminarOrganisation.idl`
- Erzeugt ein Java-Paket `SeminarOrganisation` mit
  - `Firma`: Die aus der CORBA-IDL erzeugte Java-Schnittstelle
  - `FirmaOperations`: eine Hilfsklasse
  - `_FirmaStub`: die Stummel-Klasse
  - `FirmaPOA`: die Skelett-Klasse
    - Bildet eine Hülle für die eigentliche Implementierung und implementiert die Schnittstelle `Firma`
  - `FirmaPOATie`: eine Hilfsklasse.

3.9.5.4 Entw. von CORBA-Anwendungen in Java

3 Implementierung der Operationen der Schnittstelle

- Implementierung erfolgt in Klasse `FirmaImpl`
- Sie muss von der Klasse `FirmaPOA` erben
- Die Implementierung könnte z.B. so aussehen:

```
package SeminarOrganisation;
public class FirmaImpl extends FirmaPOA
{   private String derName; private float derUmsatz;
    public FirmaImpl()
    { //Attribute initialisieren
    }
    public float berechneGewinn(float Kosten)
    { return this.Umsatz – Kosten;
    }.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

```
    //get- und set-Oerationen für Attribut Name
    public String Name()
    {   return this.derName;
    }
    public void Name(String neuerName)
    {   this.derName = neuerName;
    }
    //analog werden die get- und set-Operationen
    //für Umsatz implementiert
}.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

4 Entwicklung einer Rahmen-Anwendung, in die die entwickelte CORBA-Klasse eingebettet ist

- Die Rahmen-Anwendung erzeugt ein Objekt der Klasse und macht es für *Clients* auf anderen Computersystemen zugreifbar

```
package SeminarOrganisation;
import java.io.*;
import org.omg.CosNaming.*;
public class SeminarOrganisationServer
{ public static void main( String[] args )
  {//den Object-Request-Broker initialisieren
    org.omg.CORBA.ORB orb =
          org.omg.CORBA.ORB.init(args, null);.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

```
try {
      //vom ORB eine Referenz auf einen Objekt-
      //Adapter holen, RootPOA ist ein fest
      //definierter Name, den der ORB immer kennt
      org.omg.CORBA.Object tempPOA =
        orb.resolve_initial_references("RootPOA");
      //da der ORB Referenzen vom Basistyp
      //CORBA.Object liefert, muss sie anschl. in
      //Referenz vom Typ POA umgewandelt werden
      org.omg.PortableServer.POA poa =
   org.omg.PortableServer.POAHelper.narrow(tempPOA);
      //den Objekt-Adapter aktivieren, damit er
      //request entgegennehmen und z.B. Objekte
      //erzeugen kann
      poa.the_POAManager().activate();.
```

3.9.5.4 Entw. von CORBA-Anwendungen in Java

```
//das Firmen-Objekt erzeugen
FirmaImpl tempFirma = new FirmaImpl();
//Erzeugung einer CORBA-Referenz auf das
//Java-Objekt
//CORBA-Referenzen auf Java-Objekte
//werden servants genannt
org.omg.CORBA.Object dieFirma =
      poa.servant_to_reference(tempFirma);
//vom ORB eine Referenz auf den Namensdienst-
//Server besorgen und von CORBA.Object in
//den gewünschten Typ umwandeln
org.omg.CORBA.Object tempNameServer =
orb.resolve_initial_reference("NameService");
NamingContextExt nameServer =
NamingContextExtHelper.narrow(tempNameServer);.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

```
//das Firmen-Objekt unter dem Namen "dieFirma"
//bekannt machen, wobei der Name
//(ein String) zunächst in ein Namens-Objekt
//konvertiert werden muss
nameServer.bind(
nameServer.to_name("eineFirma"), dieFirma );
}
catch(Exception e) { … }
}
}.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

5 Entwicklung der *Clients*

- In diesem einfachen Beispiel ist der *Client* eine einfache Anwendung
- In realen Projekten sind *Clients* zumeist selbst wieder CORBA-Klassen, die eigene Dienstleistungen anbieten, sodass ein Netz von miteinander kommunizierenden Objekten entsteht

```
package SeminarOrganisation;
import org.omg.CosNaming.*;
public class Client
{ public static void main(String[] args);
    { try
        { Firma eineFirma;.
```


3.9.5.4 Entw. von CORBA-Anwendungen in Java

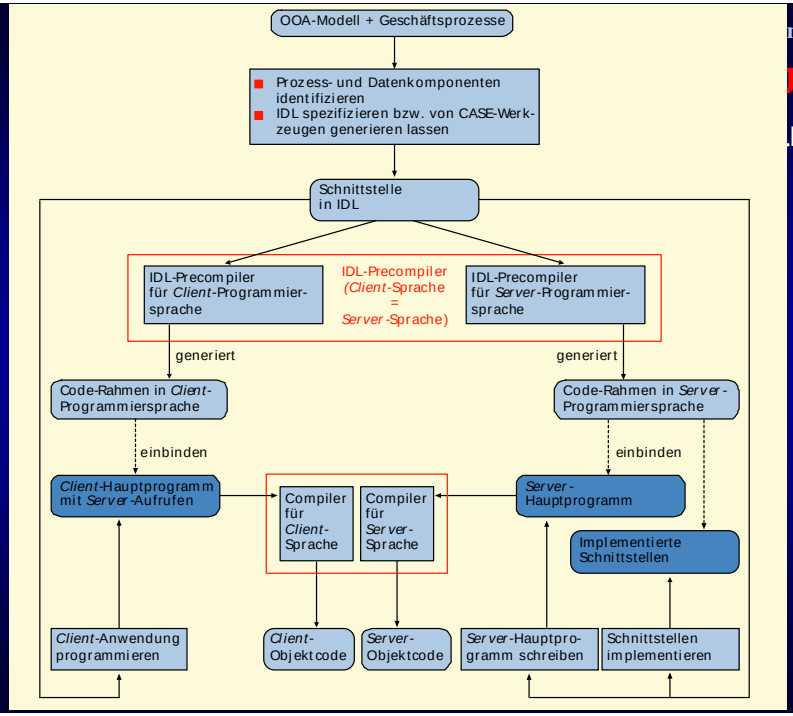
```
//den Object-Request-Broker initialisieren
Org.omg.CORBA.ORB orb =
              org.omg.CORBA.ORB.init(args, null);
//Namensdienst initialisieren
org.omg.CORBA.Object tempNameServer =
   orb.resolve_initial_reference("NameService");
NamingContextExt nameServer =
   NamingContextExtHelper.narrow(tempNameServer);
//den Namensdienst nach einer Referenz fragen
Org.omg.CORBA Object tempFirma =
nameServer.resolve(nameServer.to_name("eineFirma"));
eineFirma = FirmaHelper.narrow(tempFirma);//Aufrufe
System.out.println( eineFirma.Name() );
eineFirma.Umsatz(100.0f);
System.out.println(eineFirma.berechneGewinn(20.0f));
} catch(Exception e) {…} } }.
```

LE 29
81

3.9.5.4 Entw. von CORBA-Anwendungen in Java

- Nun können alle `.java`-Dateien mit einem Java-Compiler übersetzt werden
- Dann kann das System gestartet werden:
  - Zunächst wird der Namensdienst mit `ns C:\temp\ns_ref` gestartet
  - Der Parameter gibt eine Datei an, in die der Namensdienst Ausgaben machen kann
  - Mit `jaco SeminarOrganisation.SeminarOrganisationServer` wird die Rahmen-Anwendung gestartet und das Firmen-Objekt verfügbar gemacht (wenn JavaOrb als CORBA-Plattform verwendet wird)
  - Die *Client*-Anwendung wird mit `jaco SeminarOrganisation.Client` gestartet.

LE 29
82



LE 29
83

3.9.6 COM+

- COM+
  - Verschmelzung von COM/DCOM mit dem MTS *(Microsoft Transaction Server)*
  - In Verbindung mit Microsofts *Windows* DNA-Architektur *(Distributed interNet Application Architecture)* Konkurrenz zu CORBA
  - Im Gegensatz zu CORBA und J2EE ist COM+ nicht nur eine Spezifikation, sondern auch ein Produkt
  - Es wird kostenlos mit *Windows* 2000 ausgeliefert
  - Microsoft bietet im Rahmen seiner *Windows*-DNA-Schiene weitere Produkte (z.B. MS SQL-*Server* als Datenbanksystem) an, die nicht kostenlos sind.

LE 29
84

3.9.7 Die Modelle im Vergleich

- Vergleich der Modelle

Kriterium	EJB	CORBA *Com.*	COM+
Plattform-unabhängigkeit	gegeben	gegeben	auf *Windows* beschränkt
Sprach-unabhängigkeit	Java	gegeben	gegeben
Hersteller-unabhängigkeit	gegeben (Sun)	gegeben	einziger Hersteller ist Microsoft.