Kapitel 5
Verteilte Objekte und Komponenten
- Grundlagen

Inhalt:

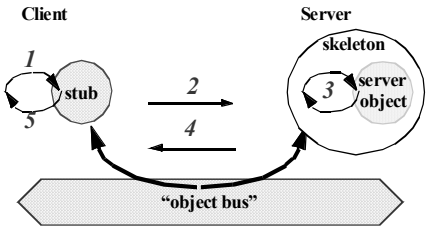
- Remote Method Invocation
- Java RMI
- CORBA
- COM+
- Enterprise JavaBeans
- Zusammenfassung und Ausblick

Verteilte Objekte

- Grundidee: Weiterentwicklung des RPC-Konzepts für Objekte
  - Anwendung besteht aus interoperablen Objektkomponenten, potentiell über Netzwerk verteilt
  - Verteilungsaspekte sind für die Anwendung nicht sichtbar
  - Objektdienste sind über Remote Method Invocation (RMI) nutzbar
- Vorteile der Objektorientierung nutzbar
  - Objektidentität
  - Verkapselung: Dienste/Zustandänderung nur über Methodenaufruf
  - Vererbung, Polymorphismus
  - Trennung von Schnittstelle und Implementierung
  - Wiederverwendbarkeit von Objekten
- Objekte können (für den Programmierer transparent) interagieren
  - über Rechnerknoten mit Hilfe verschiedener Netzwerkprotokolle
  - über verschiedene Programmiersprachen hinweg
  - zwischen unterschiedlichen Betriebssystem- und Hardwareplattformen

Grundprinzip *Remote Method Invocation*

- *server object*: Objektinstanz, die den Zustand eines "business objects" einkapselt, und sein Verhalten implementiert (durch *public interface* beschrieben)
- *stub* und *skeleton* realisieren Ortstransparenz; *stub* implementiert das gleiche interface wie das *server object*, agiert and seiner Stelle
- Verarbeitungsprinzip
  - Client-Objekt ruft Objektmethode des Stub-Objekts (***1***)
  - *stub* kodiert Aufruf(-parameter) als Botschaft („marshalling") und schickt sie zum Server (***2***)
  - *skeleton* dekodiert die Botschaft ("unmarshalling") und ruft die enstpr. Serverobjektmethode (***3***)



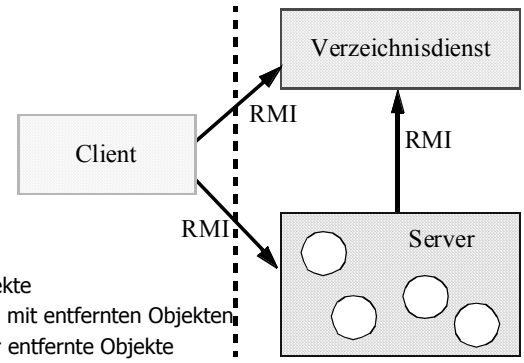
  - das Resultat des Aufrufs wird vom *skeleton* an den Client zurückgeschickt (***4***)
  - *stub* liefert das Resultat an Clientanwendung ab (***5***)
- „Client" und „Server" sind nur Rollen

Grundlegender Dienst: Namensdienst

- Fragestellungen
  - Wie findet ein Client seine Server?
  - Wie kann ein Server seinen Dienst anbieten?
- Grundlegende Funktionen
  - Abbildung symbolischer Namen auf Objekt-/Dienstreferenzen (= Bindungen)
  - Programmschnittstelle zum Namensdienst (= lookup API)
    - liefert "Objektreferenz", die für RMI genutzt werden kann
- Analogie: Telefonbuch

Java RMI

- Mechanismus zur Kommunikation
  - zwischen Java-Programmen
  - zwischen Java-Programmen und Applets
- Kommunikationsbeziehungen



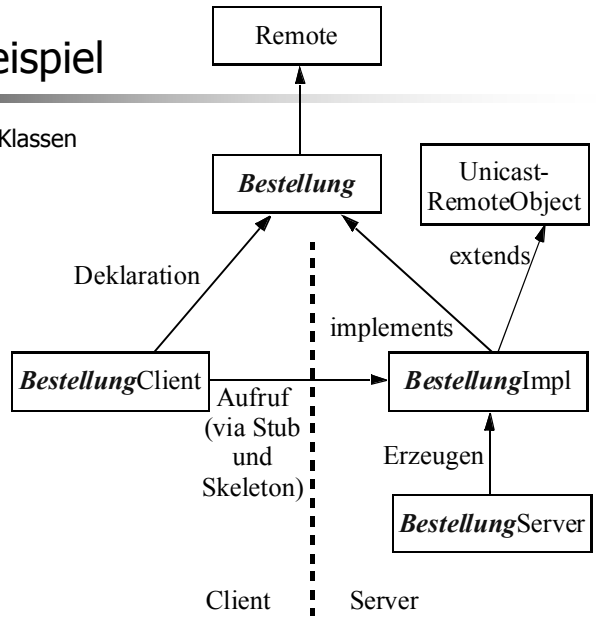
- Aufgaben
  - Lokalisierung entfernter Objekte
  - transparente Kommunikation mit entfernten Objekten
  - Laden von Java-Bytecode für entfernte Objekte

Java RMI (Forts.)

- Stub
  - Herstellen der Verbindung mit der VM, die das entfernte Objekt enthält
  - Marshalling der Methodenargumente
  - Warten, bis Server die Methode abgearbeitet hat und Resultate zurückgibt
  - Unmarshalling (Rückgabewert oder Ausnahme)
  - Weiterreichen des ermittelten Wertes an die ursprüngliche Aufrufstelle des Clients
- Skeleton
  - Unmarshalling (Argumente)
  - Ausführung der Methode
  - Marshalling (Resultat)

Java RMI - Beispiel

- Zusammenspiel der Klassen



Java RMI – Beipiel (Forts.)

```
import java.rmi.*;
import java.util.Date;
public interface Bestellung extends Remote {
        public void neuebestellPosition(int pizzaId, int anzahl)
                                        throws RemoteException;
        public Date getLieferDatum() throws
                RemoteException;
        public Date setLieferDatum(Date neuesDatum) throws
                RemoteException;
        }


...
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
```

Java RMI - Beispiel (Forts.)

```
...
public class BestellungImpl
        extends UnicastRemoteObject
        implements Bestellung {
                private Vector fBestellPositionen;
                private Date fLieferDatum;
                public BestellungImpl(String name) throws RemoteException {
                  super();
                  try {
                     Naming.rebind(name, this); // Registrierung bei Nameserver
                     fBestellPositionen = new Vector();
                     fLieferDatum = null;
                      }
                  catch (Exception e) {
                     System.err.println("Ausnahme: " + e.getMessage());
                     e.printStackTrace();
                                }
                        }
...
```


Java RMI - Beispiel (Forts.)

```
...
public void neueBestellPosition(int pizzaId, int anzahl )
                throws RemoteException {
                // die Klasse bestellPosition sei bekannt
                BestellPosition bestellPosition = new
                        BestellPosition(pizzaId, anzahl);
                fBestellPositionen.addElement(bestellPosition);
                        }
                ... // Impl. v. getLieferDatum und setLieferDatum
        }
```

Java RMI - Beispiel (Forts.)

```
...
import java.rmi.*;
public class BestellungClient {
        public static void Main(String args[]) {
        try {
                Bestellung bestellung = (Bestellung)
                    Naming.lookup("rmi://berlin:9000/meine_best");
                int pizzaId = Integer.parseInt(args[0]);
                int anzahl = Integer.parseInt(args[1]);
                bestellung.neueBestellPosition(pizzaId, anzahl);
                        }
        catch (Exception e) {
                System.err.println("Systemfehler: " + e);
                        }
                }
        }
```


Java RMI - Beispiel (Forts.)

```
...
import java.rmi.*;
import java.server.*;
public class BestellungServer {
        public static void main(String args[]) {
          try {
            BestellungImpl bestellung =
        new BestellungImpl("meine_best");
             System.out.println("Der Bestellungsserver laeuft");
                  }
          catch (Exception e) {
             System.err.println("Ausnahme: " + e.getMessage());
             e.printStackTrace();
                        }
                }
        }
```

Java RMI - Beispiel (Forts.)

- Übersetzung Quellcode in Java-Bytecode:
  javac Bestellung.java BestellungImpl.java BestellungClient. java BestellungServer.java
- Erzeugung von Stub- und Skeleton-Code:
  rmic BestellungImpl
- administrative Schritte:
  - Starten des Verzeichnisdienstes: ***rmiregistry***
  - Starten des RMI-Servers: ***java BestellungServer***
  - Aufruf eines Clients: ***java BestellungClient***

Java RMI – Parameterübergabe

- Übergabemechanismus abhängig vom Typ des Parameters/Resultats
- Java Parameterübergabe bei lokalen Aufrufen
  - Primitiver Datentyp: pass by value
  - Objekt: pass by reference
- Remote Method Invocation
  - Primitiver Datentyp: pass by value
  - Lokales Objekt: pass by value
    - nutzt RMI Object Serialization
  - Entferntes Objekt: pass by (remote) reference
    - anstelle des Objekts wird serialisierter Stub übergeben

Java RMI – Bewertung

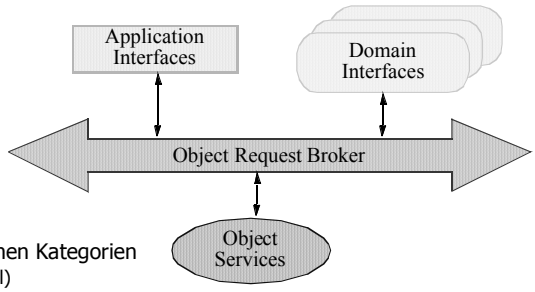
- Java-spezifisch
- Ortstransparenz, Plattformunabhängigkeit
- Nachteile
  - Kein standardisiertes Übertragungsformat/-protokoll
  - Fehlende Unterstützung wichtiger Dienste für verteilte Informationssysteme
    - Transaktionen, Security, ...
- Viele Nachteile werden durch Integration mit CORBA's IIOP behoben
  - RMI-over-IIOP

CORBA - Einführung

- CORBA: **Common Object Request Broker Architecture**
- objektorientierte, universelle Middleware-Plattform
  - Objektbus- und Komponenten-Architektur
  - erweitert RPC zu einem objektorientierten Programmiermodell
  - sprachunabhängig
  - plattformunabhängig
- OMG
  - Industriekonsortium (gegründet 1989, 11 Mitglieder)
  - heute über 1000 Mitglieder
  - keine Standards, keine Referenzimplementierungen, Herstellerunabhängigkeit
  - Produkt: Object Management Architecture (OMA)
    - Objektmodell definiert die Beschreibung von Objektschnittstellen
    - Referenzmodell definiert Interaktionen zwischen Objekten
- erste Produktentwicklungen Anfang der 90er, z. B. 1993 Orbix (erste, kommerziell verfügbare CORBA-Implementierung, für C und C++) von IONA

CORBA - Referenzmodell

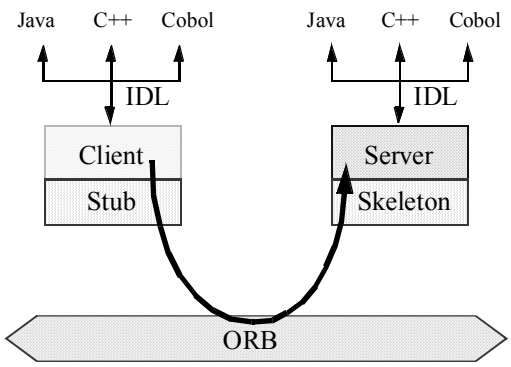
- Object Management Architecture (OMA)



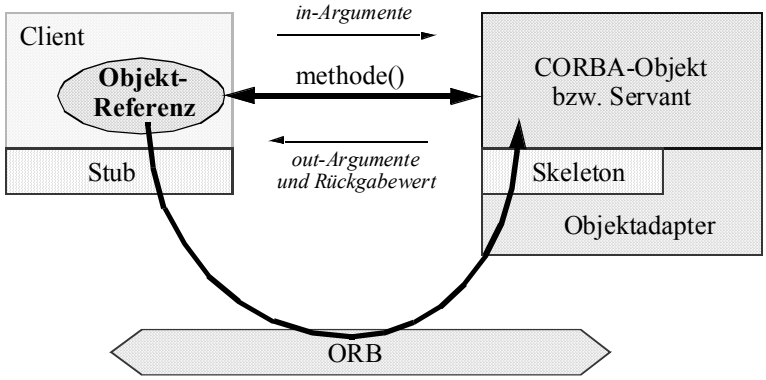
  - Interfaces in unterschiedlichen Kategorien
    - Object Services (horizontal)
    - Domain Interfaces (vertikal)
      - Telekommunikation
      - Finanzdienstleistungen
      - E-Commerce
      - Medizin
      - ...
    - Application Interfaces

CORBA - Ortstransparenz

- Object Request Broker (ORB)
  - Vermittlung der Dienstaufrufe zwischen verschiedenen Interfaces

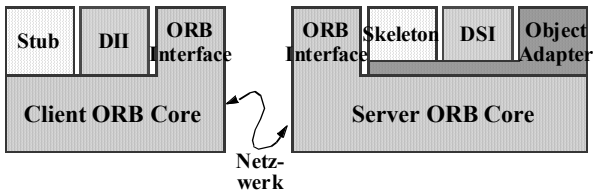


CORBA - Kommunikation



CORBA - Kernkomponenten

- Objektreferenzen (*Interoperable Object References, IOR*)
- *Object Request Broker (ORB)*
- Objektadapter
- *Stubs* und *Skeletons*
- *Dynamic Invocation/Skeleton Interface (DII/DSI)*



- Dienst-spezifisch: *Stub, Skeleton*
- identisch für alle Anwendungen: *ORB Interface, DII, DSI*

CORBA - Objektreferenzen

- Interoperable Objektreferenz (IOR)
  - jede IOR identifiziert genau eine Objektinstanz
  - unterschiedliche IORs können auf dieselbe Objektinstanz verweisen
  - eine IOR kann NIL sein
  - dangling IOR möglich
  - der interne Aufbau von IOR bleibt den Clients verborgen
  - IORs sind streng getypt (erlaubt Typprüfung zur Compilezeit)
  - IORs unterstützen late binding (und damit Polymorphismus)
  - IORs können persistent sein
  - der interne Aufbau ist standardisiert
  - IORs können vom ORB in einen String mit standardisiertem Format umgewandelt und daraus wieder rekonstruiert werden

CORBA – ORB und Objektadapter

- ORB
  - übernimmt die Netzwerkkommunikation und das Verbindungsmanagement
  - verwaltet Stubs (Client-Seite)
  - bildet Methodenaufrufe auf Objektadapter ab (Server-Seite)
  - bietet Hilfsfunktionen an (z. B. Konvertierung von Objektreferenzen)
  - Scheduling von Dienstaufrufen auf Threads
- Objektadapter: Portable Object Adapter (POA)
  - erzeugt Objektreferenzen
  - bildet CORBA-Methodenaufrufe auf Servants ab
  - aktiviert/deaktiviert/registiert Servants und spezifiziert Server-Zustände (Bereitschaft)
  - Berücksichtigung von Multithreading
- ORB + Objektadapter realisieren den Request Broker

CORBA - Objektaufrufe

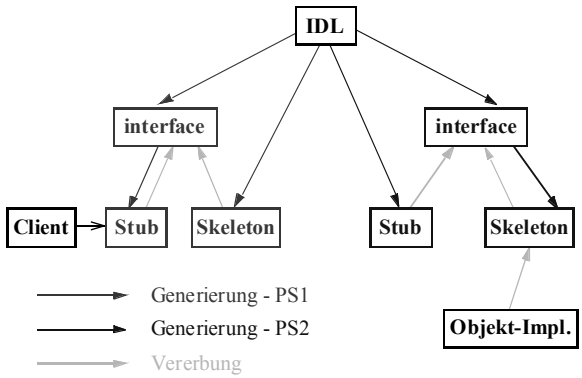
- statische Objektaufrufe
  - Stub und Skeleton werden aus der Dienstbeschreibung generiert
  - Skeleton bildet generischen Aufruf des Objektadapters auf tatsächlichen Methodenaufruf ab
  - statische Typprüfung zur Compilezeit
  - effizienter als dynamische Aufrufe (s. u.)
- dynamische Objektaufrufe
  - Objektschnittstellen/Objektaufrufe werden zur Laufzeit ermittelt (rein syntaktisches Matching, z. B. über ein Interface Repository)
  - DII und DSI sind unabhängig von Dienstbeschreibungen
  - Typüberprüfung zur Laufzeit beim Aufruf
  - unterstützen Flexibilität

CORBA – Sprachunabhängigkeit

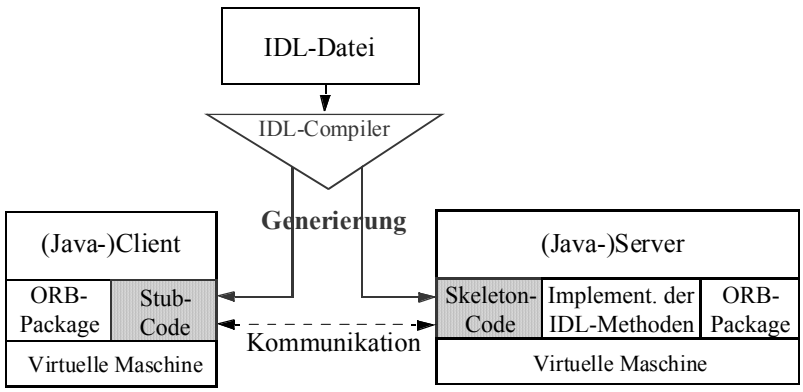
- Ziel: Entwicklung von Clients und Servern in unterschiedlichen Programmiersprachen
  - Trennung von Interface und Implementierung
  - Beschreibung des Interface in programmiersprachen-unabhängiger Interface Definition Language (IDL)
  - Codegenerierung: Stubs, Skeletons
  - Language Mappings; normalerweise IDL-Compiler pro Programmiersprache (PS)

CORBA – Interface Definition Language

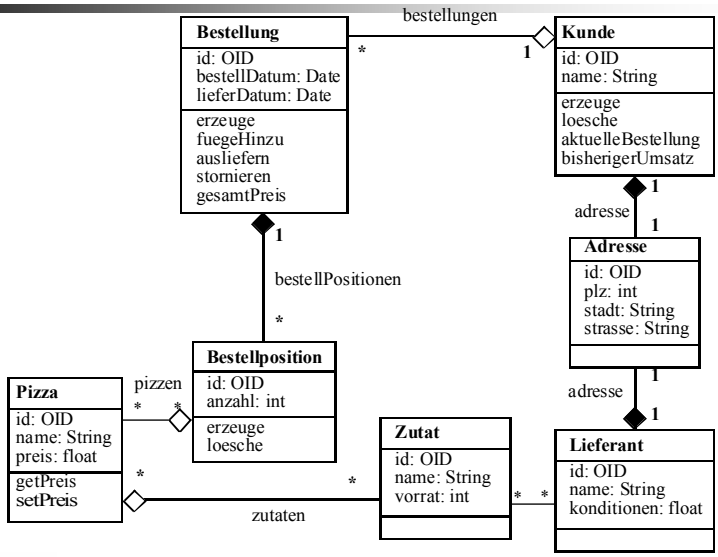
- IDL definiert (rein deklarativ):
  - Typen
  - Konstanten
  - Objekt-Interfaces (Attribute, Methoden und Exceptions)



CORBA – IDL Compiler



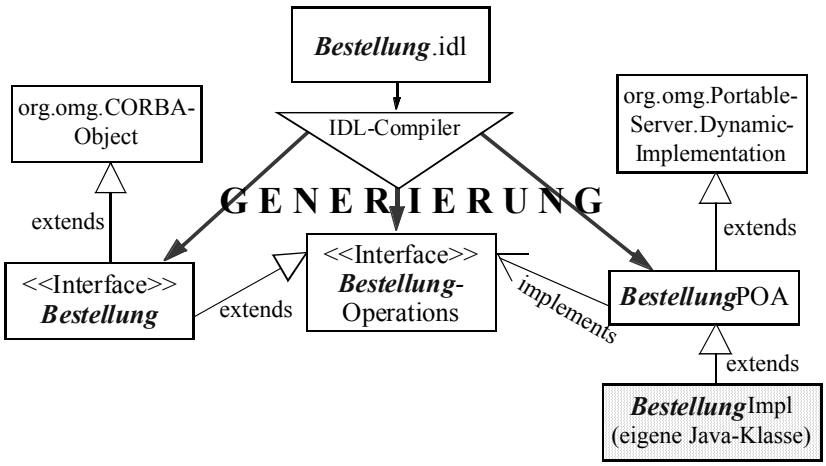
CORBA – Beispiel: Pizza-Service



CORBA – IDL-Spezifikation Bestellungsdienst

```
Module PizzaService {
    interface Bestellungsdienst {
        void neueBestellung(in long kundennr, out long bestellNr);
        void weitereBestellPosition(        in long bestellnr,
                                    in long pizzanr,
                                    in long anzahl);
                };
    interface Auslieferungsdienst {
        long auslieferung(in long kundennr);
                };
        };
interface Bestellung {
        readonly attribute long id; // nur Get-Methode
        attribute Datum lieferDatum;        // Datum sei ein IDL-Interface
        void neueBestellPosition(in long pizzaId, in long pizzaAnz);
        };
```

CORBA - Code-Generierung



CORBA – Dienstreferenzen

- enthalten alle Informationen zum Senden eines Requests
  - minimal
    - Aufenthaltsort des Dienstes (IP-Adresse des Rechnerknotens, Identifikation des Prozesses)
    - vom Server unterstütztes Interaktions-/Kommunikationsprotokoll
  - ergänzend
    - alternative Kommunikationsprotokolle
    - weitere Anlaufstellen (z. B. bei einem Server-Lastverbund)
    - Dienstgüteanforderungen
    - herstellerspezifische Information ohne Einfluss auf die Interoperabilität
  - sollte erweiterbar sein
  - *Interoperable Object Reference (IOR)*

CORBA – "On the wire"

- Datenformat:
  - Aufgaben
    - definiert die Kodierung von Datentypen
    - definiert die Verantwortlichkeit für notwendige Konvertierungen
  - *Common Data Representation* (CDR)
- Kommunikationsprotokoll
  - definiert die Interaktionen zwischen Client und Server
    - Nachrichtenaufbau
    - Nachrichtensequenzen
  - CORBA 2.0: *General Inter-ORB Protocol* (GIOP)
  - *Internet-Inter-ORB-Protocol* (IIOP)
    - konkretisiert GIOP für TCP/IP
    - Internet als "Backbone-ORB"
  - Kür: Environment-Specific Inter-ORB Protocols (ESIOP)
    - Beispiel: DCE Common Inter-ORB Protocol (DCE-CIOP)

CORBA – Namensdienst

- hierarchische Namen
  - Kontexte (Mengen von Bindungen) sind CORBA-Objekte
  - alle Bindungen verweisen auf Dienstreferenzen
- standardisierte Schnittstellen
  - Abfragen/Eintragen/Entfernen/Ändern von Bindungen
  - Iterieren über alle Bindungen eines Kontextes
- "Namensgraph" verfügt über mindestens einen initialen Kontext
  - Einstiegspunkte in den Namensgraph
- unterstützt verteilte Namensdienste
  - da Kontexte wiederum CORBA-Objekte sind
  - initialer Namenskontext über ObjektID **NameService**

CORBA Object Services

- Ziel: Erweiterung der Basisfunktionalität des ORB durch zusätzliche Systemdienste
  - Naming, Life Cycle, Events, Persistence, Concurrency Control, Transaction, Relationship, Externalization, Query, Licensing, Properties, Time, Security, Trading, Collections
- Nutzung eines Dienstes
  - Funktionalität über CORBA-IDL Schnittstellen definiert
  - CORBA-Objekt ruft Methode eines Dienstobjekts
    - Beispiel: NameService
  - CORBA-Objekt implementiert für einen Dienst definierte Schnittstellen
    - Beispiel: TransactionalObject

COM+

- COM+ erstmals mit *Windows 2000* ausgeliefert
  - Integration von COM, DCOM und MTS
- *Component Object Model (COM*, 1993)
  - Kommunikation von COM-Komponenten innerhalb eines Prozesses oder zwischen verschiedenen Prozessen auf dem gleichen Rechnerknoten
  - COM-Objekt kann mehrere Schnittstellen bereitstellen
  - jede Schnittstelle enthält Menge funktional zusammengehörender Methoden
  - COM-Client interagiert mit COM-Objekt, in dem eine Referenz auf eine der Schnittstellen benutzt wird
  - jede Schnittstelle muss bestimmtem Speicherlayout folgen
  - Spezifikation auf binärer Ebene ermöglicht Nutzung verschiedener Programmiersprachen (heute vor allem *Java, C++, VisualBasic*)

COM+ (Forts.)

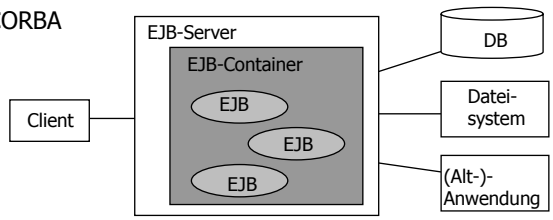
- *Distributed COM (DCOM*, 1996)
  - Kommunikation über verschiedene Rechnerknoten hinweg
  - baut auf DCE-RPC auf
- *Microsoft Transaction Server (MTS*, 1996)
  - ermöglicht Kommunikation über DCOM mit Windows- und Unix-Clients
  - Speicherung von Objekten und Koordination verteilter TA
  - Applikationsserver-Laufzeitumgebung
- *COM+ Services*
- *COM-CORBA-Bridge* verfügbar

Server-seitige Komponentenmodelle

- Probleme mit CORBA
  - Relativ hoher Programmieraufwand für Serverobjekte
    - Nutzung von Diensten (Transaktionen, Sicherheit, ...)
      - wird zur Entwicklungszeit festgeschrieben
    - Resourcenmanagement, Lastbalancierung
      - proprietäre Schnittstellen, falls durch Objektadapter unterstützt
- Standardisiertes Server-seitiges Komponentenmodell
  - Entwickler von Serverkomponente konzentriert sich auf Anwendungslogik
    - Nutzung von Diensten erfolgt in vielen Fällen durch Konfiguration der Anwendungskomponente zum Zeitpunkt des Deployments
      - Codegenerierung
    - Resourcenmanagement, Lastbalancierung durch wird vom Server übernommen
      - Komponenten erfüllen bestimmte Voraussetzungen zum Ablauf auf dem Server
  - Serverkomponenten sind portabel

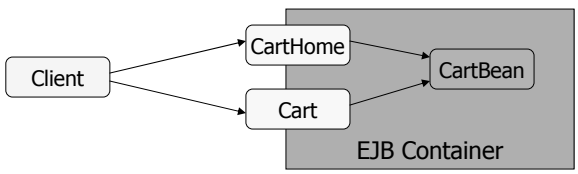
Enterprise JavaBeans (EJBs)

- Standard für Server-seitige Komponenten in Java
  - Kapselung von Anwendungslogik
    - *Business Object Components*
    - Präsentationslogik durch weitere, komplementäre Komponentenmodelle unterstützt
      - Servlets, JSPs
  - EJB Container
    - Ablaufumgebung für EJB
      - Bereitstellung von Diensten und Ablaufkontext
    - *Bean-container contract*
      - EJB realisiert call-back Methoden
- Interoperabilität mit CORBA
  - Aufruf: RMI/IIOP
  - Dienste



EJBs - Bestandteile

- Enterprise Java Bean (EJB) besteht aus (ejb-jar file):
  - einer Klasse, die die Geschäftslogik implementiert (*Bean*)
  - einem Remote-Interface, das die Methoden beschreibt
  - einem Life-Cycle-Interface (*Home* interface)
    - Erzeugen, Aufsuchen, Löschen
  - einem Deployment-Deskriptor
  - einer Primary-Key-Klasse, die persistente Bean-Objekte eindeutig identifiziert
- Client interagiert mit einem Bean über EJB object und EJB home
  - zum Deploymentzeitpunkt generierter Code



EJB - Grundtypen

- *Session Beans*
  - realisiert geschäftl. Aktivität/Prozess
  - zustandslos (*stateless*), bzw. flüchtiger Zustand (*stateful*) für die Dauer einer Session (*conversational state*)
- *Entity Beans*
  - repräsentiert (dauerhaftes) Geschäftsobjekt/-konzept
  - persistenter Zustand
  - *Primary-Key* ermöglicht eindeutigen Zugriff
- *Message-driven Beans*
  - asynchron, botschaftenorientiert (JMS)
  - erleichtert Intergration mit existierenden Anwendungen

Entity Beans

- Persistentes Objekte
  - Zustand i.a. durch ein DBMS verwaltet
  - Home interface enthält Methoden zum Erzeugen, Aufsuchen, Löschen
    - Home.findByPrimaryKey(...)
    - individuelle Finder-Methoden
  - Ein Entity (Instanz) kann von vielen Clients genutzt werden
    - unter Beachtung von Concurrency, Transaktionen
- Persistenzmechanismen
  - bean-managed, container-managed
- Relationships
  - Wartung von Beziehungen zwischen Entities
- Query
  - EJB-QL
    - Spezifikation der Bedeutung von benutzerdefinierten Finder-Methoden

Session Beans

- Realisierung von Aktivitäten, Vorgängen, Prozessen
  - isoliert Client von Detailsicht auf Entities
  - Reduzierung des Nachrichtenaufkommens zwischen Client und Serverkomponenten
- Session Beans sind transient
  - Beaninstanz existiert (logisch) nur für die Dauer einer "Session"
    - Home.create(...), Home.remove()
- zustandslose (*stateless*) Session-Bean
  - Zustand nur für einen Methodenaufruf
- zustandsbehaftete (*stateful*) Session-Bean
  - Zustand über Methodenaufrufe hinweg
  - Zuordnung von Bean-Instanz zu Client notwendig
- nicht persistent, können jedoch persistente Daten manipulieren
  - Beispiel: Nutzung von JDBC, SQLJ zum Zugriff auf RDBMS

Beispiel

- Lokalisieren des Home Interface

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)
    initialContext.lookup("java:comp/env/ejb/cart");
```

- Erzeugen des Session-Objekts

```
cartHome.create("John", "7506");
```

- Aufrufen der Anwendungslogik

```
cart.addItem(66);
cart.addItem(22);
...
```

- Löschen des Session-Objekts

```
cart.remove()
```

Deployment

- EJB ist Server-unabhängig
- Erinnerung: Anpassung an die Spezifika des Servers
  - Bekanntmachung der Klassen und Interfaces
  - Verbinden von Bean-Attributen mit DB-Strukturen
  - Konfiguration bzgl. Transaktionsverwaltung
  - Konfiguration bzgl. Sicherheit
  - Setzen von Umgebungsvariablen
  - Erzeugung von Glue-Code
- *Deployment Descriptor*
  - XML-Datei, entsprechende DTD wird vorgegeben
  - Beschreibung von:
    - Typ, Name
    - Persistenzart
    - Klasse, Interfaces, Primary-Key
    - persistente Felder bei Container-Verantwortlichkeit
    - Umgebungsvariablen, benötigte Ressourcen (DBS, etc.)
    - Referenzen zu den Home-Interfaces von benutzten EJBs
    - Transaktionsparameter der einzelnen Methoden
    - Referenzen auf Sicherheitsrollen

EJB Resourcenmanagement

- Traditionelle Aufgabe eines (Komponenten)-TP Monitors
  - Pooling von Resourcen, Lastmanagement, - balancierung
- EJB Spezifikation
  - *Instance Pooling* und *Instance Swapping*
    - EJB Server verwaltet kleine Anzahl von Enterprise Beans
      - Wiederverwendung, dynamische Zuordnung zur Bearbeitung von Anforderungen
    - Möglich durch die "Indirektion" des Zugriffs auf das Bean über EJB Objekt
    - I.a. nur für **stateless SessionBeans** und für **EntityBeans** anwendbar
  - *Passivation* und *Activation*
    - Zustand des Beans kann auf getrennt vom Bean abgespeichert werden (*passivation*)
      - erlaubt das Freigeben von Resourcen (Speicher), wenn bean längere Zeit nicht genutzt wird
    - Bei Bedarf kann das Bean reaktiviert werde (*activation*)
    - Nutzung von Java Serialization als Mechanismus
    - kann auch für **stateful SessionBeans** genutzt werden
- "Verboten" für EJB Entwickler:
  - Erzeugen von Threads, Nutzung von Synchronisationsprimitiven
  - Graphische Funktionalität, I-O,
  - Netzwerkoperationen
  - JNI

CORBA Component Model

- nach Einführung und raschem Markterfolg von EJBs in Zugzwang geraten
- enthält nun CORBA Component Model (CCM) als Middle-Tier-Infrastruktur
  - übernimmt Konzepte, die sich bei EJB 'bewährt' haben
  - Unterscheidung zwischen Implementierung und Deployment
  - Container (Transaktionen, Persistenz, Zugriffsschutz, Ereignisse)
  - Interoperabilität mit EJBs
  - Vorteil: CORBA-Komponenten können in mehreren Programmiersprachen realisiert werden