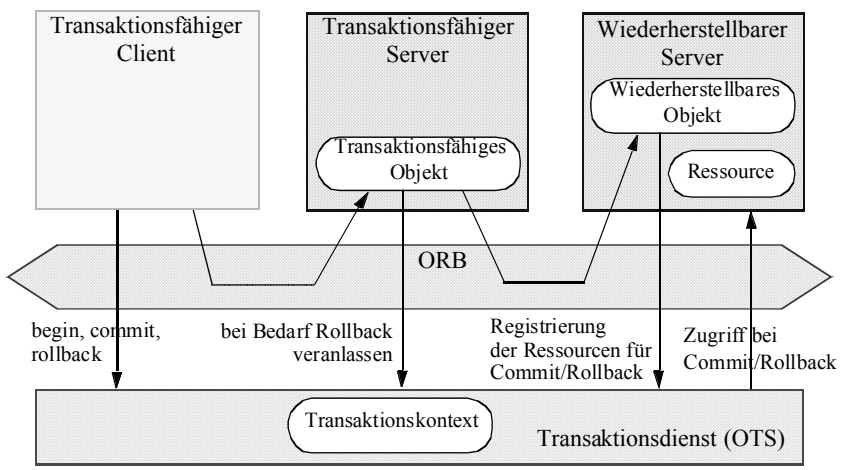
Kapitel 6
Verteilte Objekte und Komponenten
- *Dienste*

Transaktionen

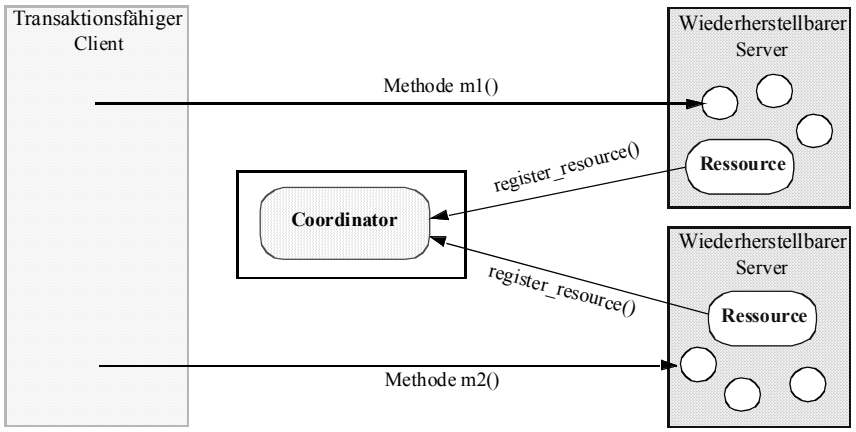
CORBA – Object Transaction Service

- Aufgaben des Object Transaction Service (OTS) entsprechen im wesentlichen denen des TA-Managers im X/OPEN-Modell
  - flache und (optional) geschachtelte Transaktionen
  - ORB-übergreifende TAs
  - X/OPEN DTP
    - Interoperabilität mit "prozeduralen" TA-Managern
- Rollen und Schnittstellen
  - Transaktionsfähiger Client
    - legt Transaktionsgrenzen fest (begin, commit, rollback)
    - nutzt OTS Interface **Current**
  - Transaktionsfähiger Server
    - nimmt an TA teil, verwaltet selbst aber keine wiederherstellbaren Resourcen
    - "implementiert" OTS Interface **TransactionalObject**
      - dient nur als "Markierung" für den ORB zum Propagieren des Transaktionskontext
  - Wiederherstellbarer Server
    - nimmt an TA teil, verwaltet wiederherstellbaren Resourcen
    - implementiert OTS Interface **TransactionalObject** und **Resource**, nutzt **Current** und **Coordinator**
      - nimmt an Commit-Protokoll teil

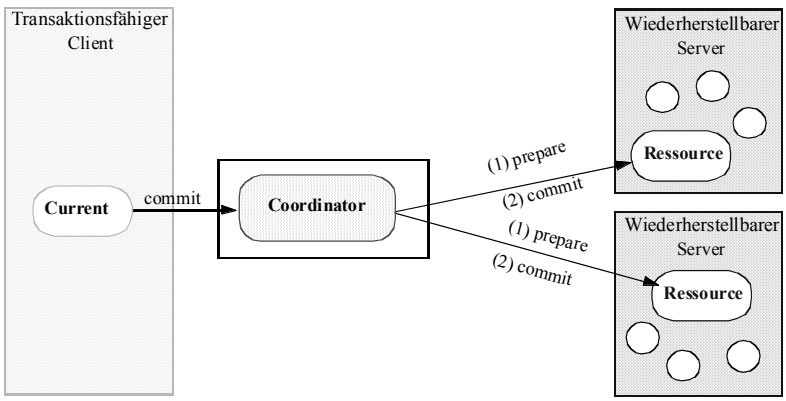
OTS – Elemente und Interaktion



OTS – Registrierung von Resourcen



OTS - Commitphase



CORBA – Concurrency Service

- Concurrency
  - Interfaces zum Anfordern und Freigeben von Sperren auf *shared resources*
  - falls im Dienst eines transaktionalen Clients: Freigabe von Sperren wird durch den OTS veranlasst
  - falls im Dienst eines nicht-transaktionalen Clients: Verantwortlichkeit für die Sperrfreigabe bei Client
  - Serialisierbarkeit
  - R/X/Intention/U-Sperren
  - Methode *try_lock* gibt Kontrolle zurück, statt zu blockieren
  - *Locksets*
    - Menge von Sperren, die später als Einheit freigegeben werden sollen
    - mehrere Locksets können miteinander assoziiert werden
  - *Lock-Coordinator*
    - koordinierte Freigabe von Locksets
    - wird vom OTS genutzt

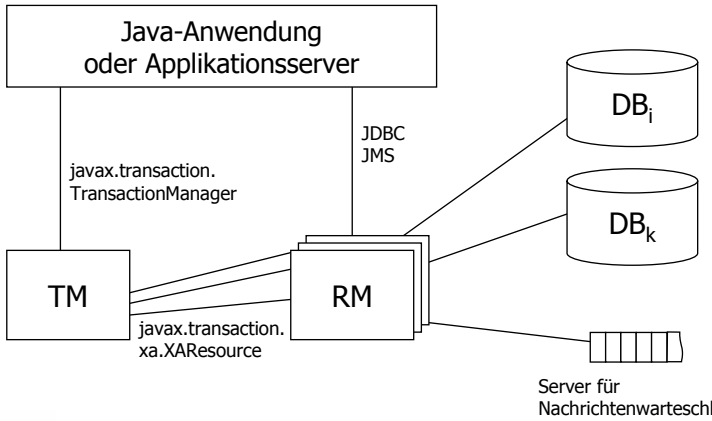
EJBs - Transaktionen

- Transaktionale Eigenschaften
  - programmatic vs. declarative transaction demarcation
  - Transaktionsattribute für Methoden im Deployment Descriptor:

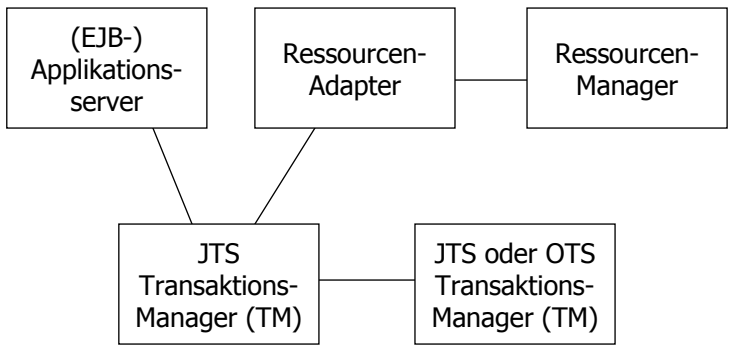
	TA-Attribute	Client-TA	TA in Methode
	Not Supported	Keine T1	Keine keine
	Supports	Keine T1	Keine T1
empfohlen für CMP entity beans	Required	Keine T1	T2 T1
	RequiresNew	Keine T1	T2 T2
	Mandatory	Keine T1	Fehler T1
	Never	Keine T1	Keine Fehler

Explizite (programmatische) Demarkation von Transaktionen

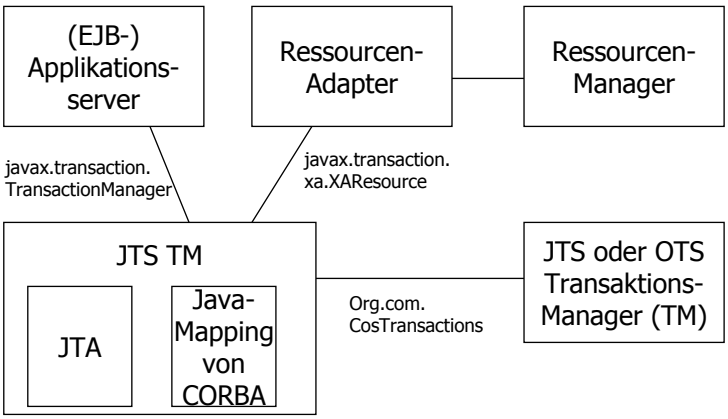
- Nutzung des Java Transaction APIs (JTA)
- UserTransaction Objekt bereitgestellt durch JNDI (oder EJB-Kontext)
- Nicht erlaubt für EntityBeans



Grobarchitektur von JTS



Zusammenspiel von OTS und JTS

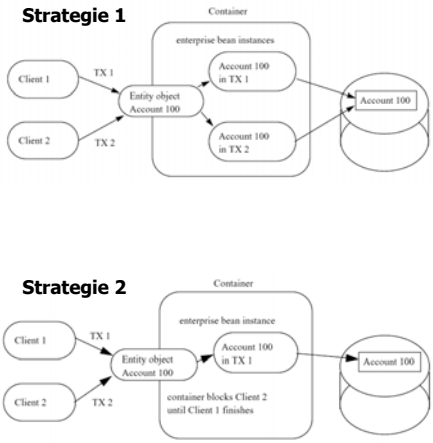


Explizites TA-Management mit JTA

- Kann von EJB Session Beans und EJB-Client, Webkomponenten genutzt werden
  - EJB: im Deskriptor *transaction-type = Bean*
- Demarkation mit Hilfe von JTA UserTransaction
  - *begin()* – erzeugt neue TA, assoziiert mit dem *current thread*
  - *commit()* – beendet TA, current thread nicht mehr mit einer TA assoziiert
  - *rollback()* – setzt TA zurück
  - *setRollbackOnly()* – markiert TA für späteres Rollback
    - Beans mit impl. TA-Mgmnt nutzen hierfür Methode des *EJBContext*
  - *setTransactionTimeout(int seconds)* – setzt timeout limit für TA
  - *getStatus()* – liefert TA Statusinformation
    - active, marked rollback, no transaction, ...
- Stateless SessionBeans
  - *begin()* und *commit()* müssen in der gleichen Methode erfolgen
- Stateful SessionBeans
  - *commit()* muss nicht in der gleichen Methode erfolgen wie *begin()*
  - TA kann über mehrere Methodenaufrufe des gleichen Beans hinweg offen bleiben

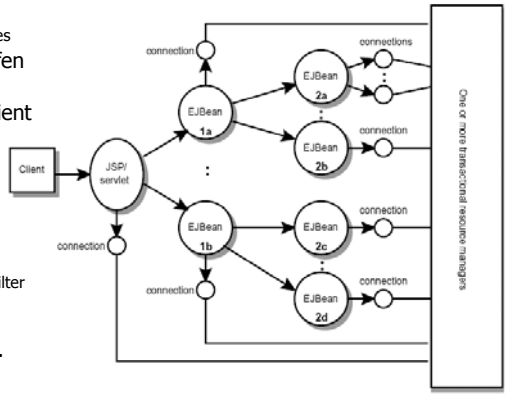
EJB Entity Beans – Concurrency

- Gleichzeitiger Zugriff auf das selbe Entity Bean aus mehrern TAs wird durch den Container kontrolliert
  - mehrere Strategien denkbar in Abhängigkeit von Commit-time Realisierung
- Strategie 1: Kontrolle an das DBMS delegiert
  - mehrere Instanzen des gleichen Beans aktiv
  - Lese- und Schreibzugriffe auf das Bean werden im scope der client TA zum DBMS propagiert
- Strategie 2: Kontrolle im Container realisiert



J2EE Transaktionsunterstützung

- Einbinden der RM in globale TA in der Verantwortung des Containers
  - in Zusammenarbeit mit RM
    - siehe z.B. JDBC XADataSources
- Servlet/JSP Anforderungen dürfen keine TAs offen lassen
- Transaktionskontext im Web Client wird i.A. nicht propagiert
- Optimierungen zur besseren Nutzung von Resourcen, Performanzsteigerung durch Container möglich
  - Lokale TA
    - autom. Umwandlung zu verteilter TA, wenn nötig
  - Connection Sharing
- Zusätzliche Charakteristika (z.B. Konsistenzebenen)
  - RM-spezifische APIs
  - proprietäre Erweiterungen des deployment descriptors



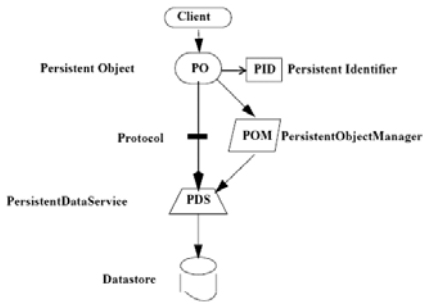
Transaktionsunterstützung – Zusammenfassung

- Infrastruktur wird durch Transaktionsdienst, Container bereitgestellt
- Realisierung von transaktionalem Verhalten durch
  - explizite Nutzung des Dienstes über vorgegebene Schnittstellen
    - Demarkation, Propagieren des Transaktionskontextes
    - OTS, JTA, JTS
    - erhöht den Programmieraufwand (insbesondere bei OTS, JTS)
    - bietet volle Flexibilität, Kontrolle
  - implizite Nutzung durch Angabe von transaktionalen Charakteristika der Komponenten
    - Container-managed Transactions (EJB)
    - erhöht Produktivität des Programmierers
    - erlaubt Konfiguration des transaktionalen Verhaltens zum Zeitpunkt des deployments
    - in den meisten Fällen ausreichend
- Interoperabilität
  - JTS-OTS Mapping ermöglicht Propagieren des TA-Kontextes Koordinieren von TAs zwischen CORBA und J2EE Applikationen

Persistence, Relationships, Query

CORBA – Persistent Object Service

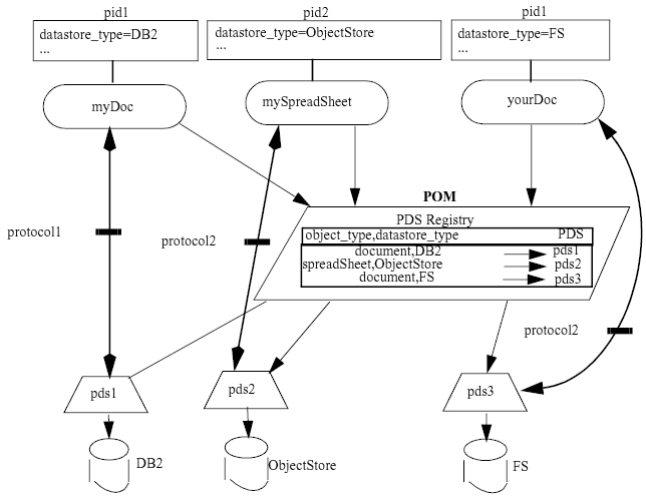
- Ziel: einheitliche Schnittstellen zur Realisierung von Objektpersistenz
- Komponenten des *POS* (*Persistent Object Service*)
  - PO: *Persistent Object*
    - werden identifiziert durch *PID* (*persistent object identifier*)
    - PID beschreibt Objekt-Lokation
  - POM: *Persistent Object Manager*
    - Mediator zwischen POs und PDS
    - realisiert Interface für Persistenz-Operationen
    - interpretiert PIDs
    - implementierungsunabhängig
  - PDS: *Persistent Data Service*
    - Mediator zwischen POM/PO und persistenten Datenspeichern (Datastore)
    - Austausch von Daten zwischen Objekt und Datenspeicher (durch *Protocol* genauer definiert)



CORBA Persistenzmodell

- CORBA Objekt ist für die Realisisierung seiner Persistenz selbst verantwortlich
  - kann Dienste, Funktionen von PDS nutzen
- Client kreiert PID, PO mit Hilfe von Factory-Objekten
- Explizite Kontrolle von Persistenz durch CORBA Client möglich
  - PO Interface
    - connect/disconnect – automatische Persistenz für Dauer der "Verbindung"
    - store/restore/delete – expliziter Transfer von Daten
    - werden an POM, PDS delegiert
  - Vorsicht: CORBA Objektreferenz und PID sind unterschiedliche Konzepte
    - Client kann das gleich CORBA Objekt mit Daten von unterschiedlichen persistenten Objekzuständen "laden"

Persistent Object Manager



Persistenzprotokolle

- CORBA Persistence Service definiert drei Protokolle
  - Direct Access (DA) Protokoll
    - PO speichert persistenten Zustand mit Hilfe von sog. *data objects* (DOs)
      - sind CORBA Objekte, deren interface nur Attribute aufweist
      - werden mit Data Definition Language (IDL subset) definiert
    - DOs können andere DOs, CORBA Objekte persistent referenzieren
  - ODMG'93 Protokoll
    - ähnelt dem DA Protokoll (ist eine Obermenge)
      - eigene DDL (ODL) zur Definition von POs
    - ideal zur Anbindung von OODBMS
  - Dynamic Data Object (DDO) Protokoll
    - "generisches", selbstbeschreibendes DO
      - Methoden zum Lesen, Ändern, Hinzufügen von Attributen und Werten
      - Manipulation von Metadaten
  - Nutzung von DDO zum Zugriff auf Record-basierte Datenquellen (z.B. RDBMS) mit Hilfe eines DataStore CLI Interface
    - SQL CLI für CORBA

CORBA Queries und Relationships

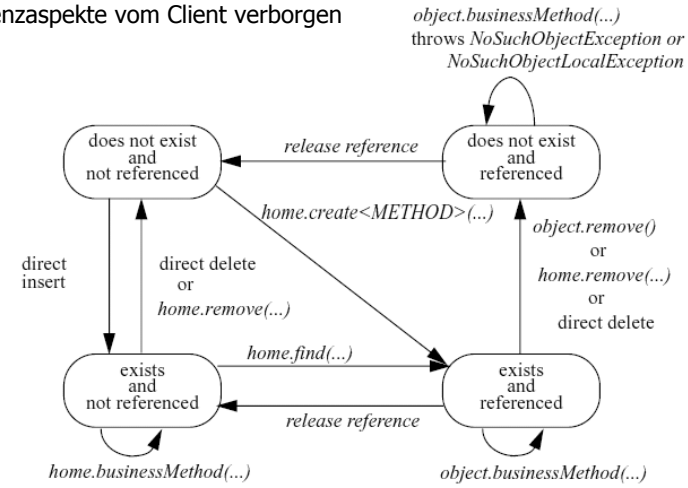
- Query Service
  - mengenorientierte Anfragen zum Auffinden von CORBA-Objekten
  - SQL, OQL
  - Query-Ergebnisse werden durch Collection-Objekte repräsentiert
    - Iteratorkonzept
- Relationship Service
  - Verwaltung von Objektabhängigkeiten
  - Relationship: Typ, Rollen, Kardinalitäten

EJB – Entity Beans

- Verfolgt *transparent persistence* Ansatz
  - persistenzbezogene Operationen (z.B. Abgleichen des Objektzustandes mit der DB) sind dem Client verborgen
  - Realisierung von Persistenzaspekten geschieht ausserhalb der vom Bean realisierten Anwendungslogik
- Entity Bean "implementiert" call-back Methoden für Persistenz
  - ejbCreate – Einfügen des Objektzustands in die DB
  - ejbLoad – Retrieval des persistenten Zustandes von der DB
  - ejbStore – Abspeichern des (geänderten) Zustandes in der DB
  - ejbRemove – Löschen des persistenten Objektzustandes

Entity Beans - Client-Perspektive

- Persistenzaspekte vom Client verborgen



Container-Managed Persistence (CMP)

- Bean-Entwickler definiert ein *abstract persistence schema* im Deploymentdeskriptor
  - persistente Attribute (*CMP fields*)
  - Beziehungen
- Zuordnung von Bean-Attributen zu DB-Strukturen in Deployment-Phase
  - abhängig von DB, Datenmodell
  - werkzeugunterstützt
    - *top-down, bottom-up, meet-in-the-middle*
- Container sichert Zustand zu bestimmten Zeitpunkten, verwaltet Beziehungen
  - Bean-Programmierer muss sich kaum mit dem Persistenzmechanismus auseinandersetzen
    - call-back Methoden enthalten keine expliziten DB-Zugriffe
- Zugriff auf CMP fields über Zugriffsmethoden (get*field*(), set*field*(...))
  - für Zugriff innerhalb der Beanmethoden
  - Zugriff durch den Client kann über das remote interface ermöglicht werden
  - ermöglicht zusätzlichen Spielraum für Containerimplementierung
    - lazy loading von einzelnen Attributen
    - updates individuell für geänderte Attribute

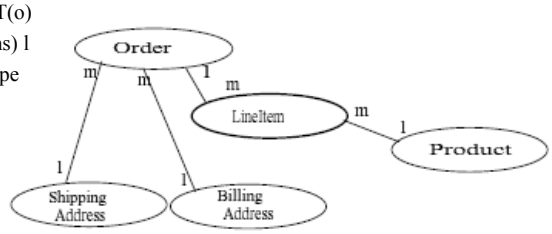
Container-managed Relationships

- Beziehungen können im deployment descriptor definiert werden
  - Teil des abstrakten Persistenzschemas
- Beziehungsart
  - unidirektional ("Referenz")
  - bidirektional
- Beziehungstypen
  - 1:1, 1:n, n:m
- Zugriff auf in Beziehung stehende Objekte über Zugriffsmethoden
  - analog zu CMP fields
  - Java collection interface für mengenwertige Beziehungsattribute
- Container generiert code zur
  - Wartung der Beziehung
  - persistente Speicherung
  - kaskadierendes Delete

EJB Query Language

- Anfragesprache für CMP EntityBeans
  - dient der Definition von benutzerdefinierten Finder-Methoden eines EJB Home Interface
  - bezieht sich auf das im deployment descriptor definierte abstrakte Persistenzschema
  - SQL-ähnlich
- Beispiel:

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.product.product_type
    = 'office_supplies'
```



Bean-Managed Persistence (BMP)

- Callback-Methoden enthalten explizite Datenbankzugriffe
  - Nutzung vor allem bei Legacy-Systemen bzw. komplizierter DB-Abbildung (vom Container nicht unterstützt)
- keine Unterstützung für Container-managed Relationships
- Finder-Methoden
  - müssen in Java implementiert werden
  - keine Unterstützung für EJB-QL

Entity Beans

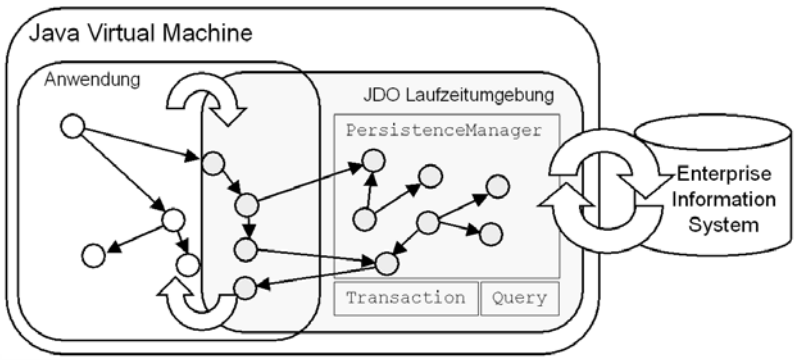
- Probleme
  - Verteilte Komponente vs. persistentes Objekt
    - Granularität
    - potentieller Overhead
      - Lösung in EJB 2.0: local interfaces
      - aber: sem. Differenzen (*call-by-value* vs. *call-by-reference*)
    - Komplexität des Entwicklungsprozesses
  - Fehlende Unterstützung für Vererbung
  - mögl. Performanzprobleme
- Abhilfe?
  - direkte Nutzung von JDBC, stored procedures
    - aufwändig
  - Nutzung eines O/R Mapping-Produktes
    - proprietärer Ansatz
  - Implementieren eines eigene Persistenzframeworks
    - komplex
  - JDO

JDO – Java Data Objects

- JDO gedacht als neuer Standard für Persistenz innerhalb von javabasierten Applikationen.
  - Erste JDO Spezifikation 1.0 veröffentlicht im März 2002 (nach ca. 3 Jahren)
  - Entstanden innerhalb von JCP (Java Community Process)
  - Bereits über 10 vendor implementations und einige Open-Source Projekte.
  - Es gibt *mandatory features* und *optional features* in der Spezifikation (d.h. auch einige optional features sind „standardisiert" → erhöht Portabilität).
- Bestandteile
  - Orthogonale Persistenz
  - native Javaobjekte (Vererbung)
  - Byte-Code-Enhancement
  - Mapping auf Persistenzschicht mit XML-Metadaten
  - Transaktionalität
  - JDO Query Language
  - JDO-API
  - JDO-Identität
  - JDO-Lebenszyklus
  - Integration in Application Server

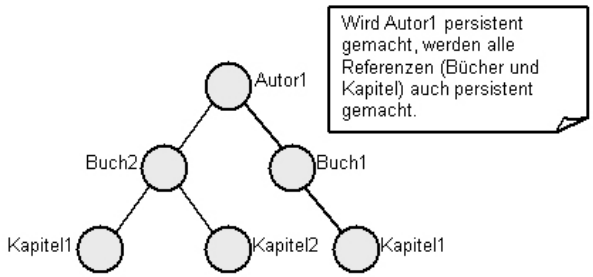
Orthogonale Persistenz in JDO

- Objektbasierte Persistenz, unabhängig von Typ/Klasse
  - Nicht alle Objekte einer Klasse sind *per se* persistent.
- Persistenzlogik ist für die Anwendung nicht sichtbar
  - Interaktion mit transienten und persistenten Objekten ist gleich
- *"persistence by reachability"*



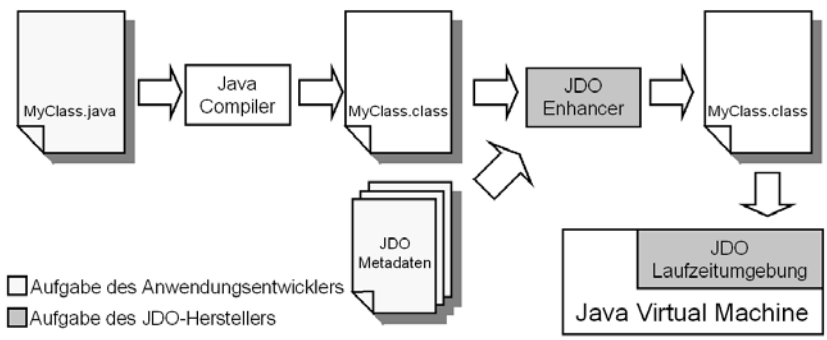
Persistence by Reachability

- alle **`PersistenceCapable`** des gesamter Objektgraph werden persistent gemacht
- kaskadierendes Löschen ist optional



Byte-Code-Enhancement

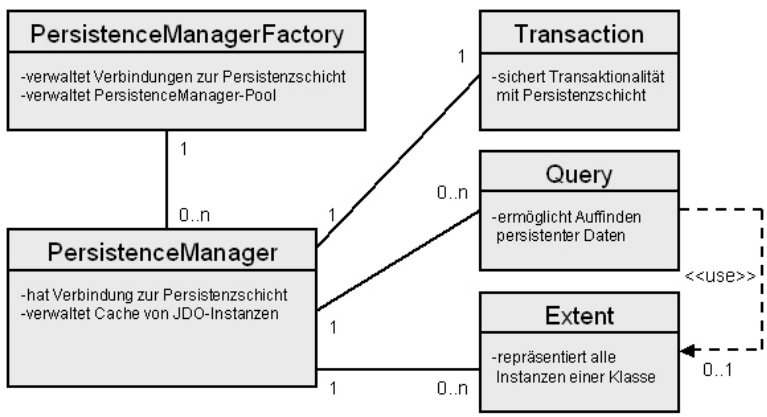
- Java-Byte-Code (*.class) und Metadaten (*.jdo)
- Gleicher Objekttyp (implements PersistenceCapable)
- O/R-Mapping ist herstellerspezifisch



JDO-Metadaten

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE jdo..."jdo.dtd">
<jdo>
  <package name="de.mms_dresden.jobboerse.data.user">
    <class name="Recht">
      <field name="nutzerId">
        <extension key="jdbc" vendor-name="ssibo">
          <extension key="mapping" value="direct" vendor...">
            <extension key="table-name" value="JOB2_RIGHTS".../>
            <extension key="field-name" value="USER_ID" .../>
          </extension>
        </extension>
      </field>
       <field name="rechtId">
     ...
       <extension key="jdbc" vendor-name="ssibo">
         <extension key="table-name" value="JOB2_RIGHTS" .../>
         <extension key="schema-name" value="JOBS" vendor ..."/>
       </extension>
    </class>
  </package>
</jdo>
```


JDO API



PersistenceManager

- Objekt wird persistent gemacht und wieder gelöscht:

```
1  Autor autor1 = new Autor(„Hans"„Dampf");
2  PersistenceManager pm1 = pmf.getPersistenceManager();
3  pm1.currentTransaction.begin();
4  pm1.makePersistent(autor1);
5  Object jdoID = pm1.getObjectId(autor1);
6  pm1.currentTransaction.commit();
7  pm1.close();

8  // Die Anwendung stellt fest, dass autor1
9  // wieder gelöscht werden muss
10 PersistenceManager pm2 = pmf.getPersistenceManager();
11 pm2.currentTransaction.begin();
12 Autor autor2 = (Autor)pm2.getObjectById(jdoID);
13 pm2.deletePersistent(autor2);
14 pm2.currentTransaction.commit();
15 pm2.close();
```


Transaktionalität

- JDO-Transaktionen auf Objektebene
- Datastore Transaction Management (standard):
  - JDO synchronisiert Transaktion mit der Persistenzschicht
  - Transaktionsstrategie der Persistenzschicht wird übernommen.
- Optimistic Transaction Management (optional):
  - nur eigene JDO-Transaktion auf Objektebene.
  - erst bei Commit erfolgt Synchronisation mit Persistenzschicht.
- Transaktionen und Objektzustand orthogonal

Objekteigenschaft	transaktional	nicht-transaktional
persistent	obligatorisch	optional
transient	optional	obligatorisch (JVM Standard)

JDO Query Language

- Eine JDOQL-Anfrage besteht aus 3 Teilen:
  - *candidate class*: Klasse(n) der erwarteten Ergebnisobjekte
    → Einschränkung auf Klassenebene
  - *candidate collection*: Zu durchsuchende Collection/Extent
    → Einschränkung auf Objektebene (kann weggelassen werden)
  - *filter*: Boolsche Ausdrücke mit JDOQL (optional: andere Sprache)
- JDOQL-Eigenschaften:
  - nicht-manipulativ (kein INSERT, DELETE, UPDATE)
  - liefert JDO-Instanzen (keine Projektionen, Joins)
  - Übergabe an Query als String → Überprüfung erst zur Laufzeit
  - Logische und vergleichende Operatoren: z.B. !,==,>=
  - JDOQL-spezifische Operatoren: Typecast mit (), Navigation mit .
  - keine Methodenaufrufe in JDOQL-String möglich
  - Sortierung durch ascending/descending
  - Variablendeklaration

Query

- JDO-Query mit JDOQL zum Suchen von JDO-Instanzen:

```
1  String suchname = "Dampf";
2  Query q = pm.newQuery();
3  q.setClass(Autor.class);
4  q.setFilter("name == \"" + suchname +"\"");
5  Collection results =(Collection)q.execute();
6  Iterator it = results.iterator();
7  while (it.hasNext()){
8      // über gefundenen Objekten iterieren
9  }
10 q.close(it);
```

JDOQL Beispiele

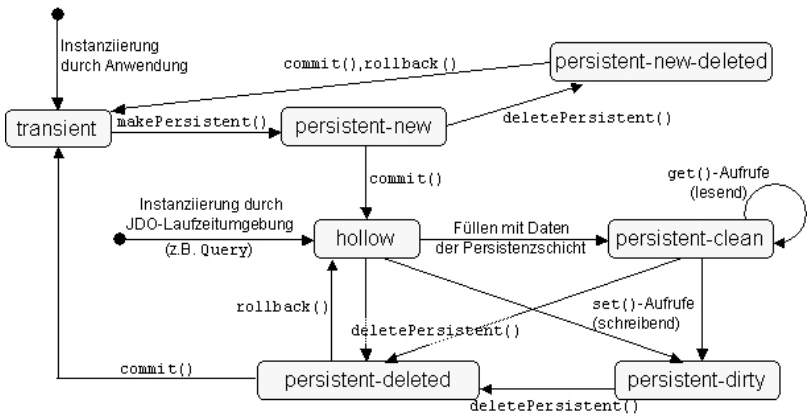
- Sortierung durch Angabe von **ascending** / **descending**:

```
1 Query query = pm.newQuery(Autor.class);
2 query.setOrdering("name ascending, vorname ascending");
3 Collection results = (Collection) query.execute();
```

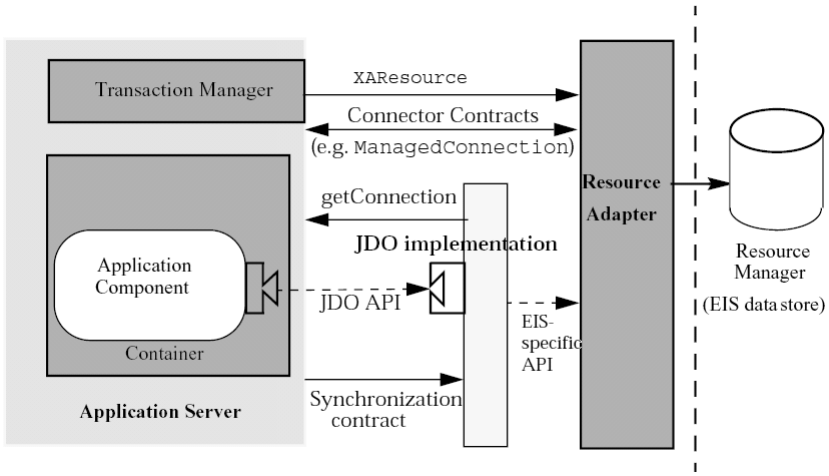
- Variablendeklaration durch **declareVariables**

```
1  String filter = "buecher.contains(meinBuch) &&  " +
2                  "(meinBuch.name == \"Core JDO\")";
3  Query query = pm.newQuery(Autor.class, filter);
4  query.declareVariables("Buch meinBuch");
5  Collection results = (Collection) query.execute();
```


JDO-Lebenszyklus



J2EE Integration



Zusammenfassung

- Unterstützung von Objektpersistenz auf unterschiedlichen Abstraktionsebenen
  - CORBA
    - standardisierte "low-level" APIs
    - sehr mächtig, flexibel, aber kein einheitliches Modell für Komponentenentwickler
      - unterschiedliche Protokolle
    - explizite vs. implizite (transparente) Persistenz
  - EJB/J2EE
    - persistente Komponenten
      - CMP: Container für Persistenz, Wartung von Beziehungen verantwortlich
    - einheitl. Programmiermodell
    - transparente Persistenz
  - JDO
    - persistente Javaobjekte
    - orthogonale Persistenz
- Anfragemechanismen
  - CORBA: Anfragen auf Objektkollektionen mit SQL, OQL
    - persistentes Objektschema?
  - EJB: Anfragen auf abstraktem Persistenzschema
    - eingeschränkte Funktionalität, nur zur Definition von Finder-Methoden
    - mehr oder minder SQL subset
  - JDO: Anfragen auf Kollektionen, Extents
    - eingeschränkte Funktionalität
    - eigener Sprachansatz
  - keine Anfragen über verteilte Datenquellen hinweg unterstützt

Sicherheit

Sicherheit

- Grundkonzepte, Aufgabenbereiche
  - Authentisierung
  - Zugriffskontrolle, Autorisierung
  - Kommunikationssicherheit
    - Vertraulichkeit
    - Integrität
  - Logging, Auditing
    - Nachweisbarkeit, Unleugbarkeit
- Sicherheitsregeln (security policy)
  - unter welchen Bedingungen darf ein Objekt/Benutzer auf ein Objekt zugreifen
  - welche Informationen werden zur Authentifikation verlangt
  - …
- Realisierung von Sicherheitsaspekten (aus Komponentensicht)
  - deklarativ/administrativ – für Komponenten spezifiziert, vom Container garantiert
  - programmatisch – von der Komponente implementiert, mglw. unter Nutzung von Standard-APIs

CORBA - Sicherheit

- Aufgaben
  - Authentisierung, Identität von Benutzern und von Diensten
  - Autorisierung, Zugriffskontrolle
  - Sicherheits-Log
  - Verschlüsselte Datenübertragung
  - Unleugbarkeit, Verantwortlichkeit
  - Verwaltung/Administration von Sicherheitseinstellungen
- unterschiedliche Sichten
  - Client-Anwendung (Authentisierung)
  - Server-Anwendung (Rechte, Kontrollen)
  - Systemverwalter (Verwaltung von Zugriffsrechten, Log-Zugriff)
  - Schutzdienst-/ORB-Entwickler (interne Nutzung)
- Referenzmodell - CORBA Security Reference Model (SRM)
  - allgemeines Rahmenwerk
  - unabhängig von bestimmten Sicherheitstechnologien

CORBA Sicherheit – Konformanzebenen

- Level 1: Schutz transparent für die Anwendung
  - auf einem sicheren ORB
  - Authentisierung von Benutzern (*principal*) durch das System
  - Zugriffskontrolle
  - sichere Übertragung
  - Logging
- Level 2: Anwendung selbst nutzt Schutzdienst
  - Client
    - Authentisierung von Benutzern
    - kontrollierte Weitergabe von Rechten
  - Server
    - Ändern von Zugriffsrechten
    - Prüfen von Identität und Rechten von Aufrufern

Sicherer Objektaufruf

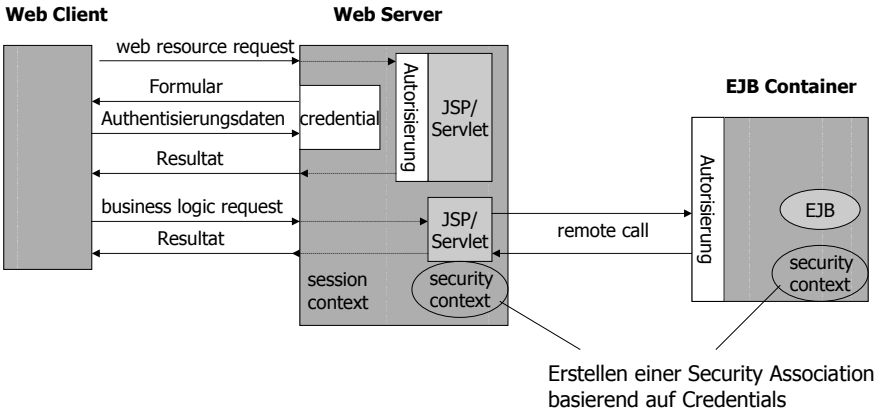
- Aufbau einer sicheren Verbindung
  - beidseitige Authentisierung
  - Mitteilung der Berechtigungen des Clients an den Server
    - *Credential*-Objekt
      - Sicherheitsmerkmale: Identität, Rolle des Benutzers (z. B. Administrator), Gruppe, Autorisierungsrang (z. B. 'geheim'), *Capabilities* (Recht, bestimmte Methoden bestimmter Objekte aufzurufen), ...
  - Aufbau eines Sicherheitskontextes (security context)
  - Schutz vor Modifikation der Nachrichten, z. B. Signaturen
  - Schutz vor Abhören (Verschlüsselung)
- Sichere Protokolle
  - Secure Inter-ORB-Protocol (SECIOP)
  - SSL
- Zugriffskontrolle und Logging
  - auf Client- und Server-Seite möglich
  - durch den ORB und möglicherweise durch die Anwendung

Autorisierung

- Client
  - Level 1
    - automatische Zuordnung von *Credentials* bei Login des Benutzers
  - Level 2
    - Aufrufe mit eingeschränkten Rechten möglich
    - (De-)Aktivierung von Sicherheitsfunktionalität
- Server
  - Level 1
    - Objekte sind einer *Security Domain* mit festgelegten *Policies* zugeordnet
      - Kontrollattribute beschreiben die *policies* für die jeweiligen Zielobjekte im Detail
        - *Access Control Lists* (ACLs)
        - *Labels* (vertraulich, ...)
    - ORB prüft Zugriffe gemäß dieser *Policies* basierend auf vorgelegten Berechtigungen sowie Zielobjekt/Methode
  - Level 2
    - *Current*-Objekt enthält Sicherheitskontext (liefert *Credentials* des Aufrufers)
    - Änderung der Zugriffs-*Policies* für eigene Objekte
    - Zugriffskontrolle durch Anwendung selbst (flexibler)

J2EE Server Security

- Beispielszenario



Containerbasierte Sicherheit

- Deklarative Sicherheit
  - Deployment Deskriptor erlaubt Beschreibung von Sicherheitsaspekten
    - Sicherheitsrollen
    - Zugriffskontrolle
    - Authentisierungsanforderungen
  - Abbildung der Sicherheitsmerkmale auf die Sicherheitsumgebung des J2EE Servers während der Deploymentphase
- Programmatische Sicherheit
  - Anwendungskomponenten implementieren Sicherheitsaspekte
  - Standardisierte Schnittstellen
    - Webkomponenten
      - isUserInRole (HttpServletRequest)
      - getUserPrincipal (HttpServletRequest)
    - EJB-Komponenten
      - isCallerInRole (EJBContext)
      - getCallerPrincipal (EJBContext)

Autorisierung

- Basierend auf Konzept der Sicherheitsrollen (security role)
  - logische Gruppierung von Benutzern
  - werden für Komponenten definiert (application provider)
  - werden während der Deploymentphase auf Benutzer(-gruppen) abgebildet
- Zugriffkontrolle mit Hilfe der Credentials
  - Name – bei Abbildung der Rolle auf indiv. Benutzer
  - Gruppenattribut – bei Abbildung der Rolle auf Benutzergruppe

Authentisierung

- Webkomponenten
  - HTTP Basic Authentication
    - Userid und Passwort (unverschlüsselt)
    - Aus Sicherheitsgründen oft Kombination mit SSL, ...
  - Formularbasierte Authentifications
    - HTML-forms, JSPs, Servlets
    - Anpassen der Benutzerschnittstelle
    - Nachteile: s.o.
  - HTTPS Client Authentication
    - basierend auf Public Key Certificates
- Single Sign-on
  - Login-Session Kontext
    - anwendungsübergreifend
    - beinhaltet Zugriff auf EJBs
  - Nur innerhalb der gleichen security domain gefordert
- Lazy Authentication
  - Authentisierung erfolgt nur beim Zugriff auf geschützte Resourcen

Sicherheit und Verteilung

- Erfordert Aufbau von Sicherheitsbeziehung (security association)
  - Kommunikationssicherheit
  - gemeinsamer Sicherheitskontext
- Wird von den beteiligten Containern aufgebaut
  - Verteilung hier innerhalb eines J2EE Serverprodukts
- Einbeziehen von Resource Managern erfordert komplexere Massnahmen
  - Authentisierung über Grenzen von security policy domains hinweg
    - Vorkonfigurierte Sicherheitsidentität
    - Programmatische Authentisierung
    - weitere Massnahmen wünschenswert (principal mapping, caller impersonation, credentials mapping)
  - Zusätzliche Möglichkeiten im Rahmen der J2EE Connector Architecture

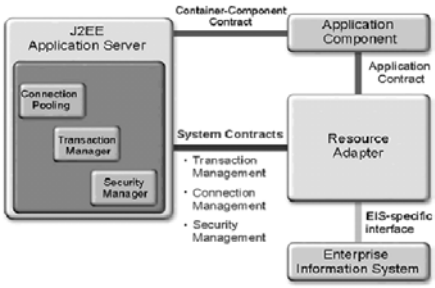
Konnektoren

Zugriff auf Betriebliche Informationssysteme

- Zugriff auf (SQL) Datenbanken kann über standardisierte Schnittstellen erfolgen
  - z.B. in J2EE explizit über SQL + JDBC/SQLJ, oder implizit durch CMP
  - Interoperabilität auf Systemebene durch wohldefinierte Schnittstellen
    - XXXDataSource für Connection Pooling, Transaktionskoordination, ...
- Zugriff auf bzw. Interaktion mit betr. Informationsystemen?
  - Beispiele
    - Enterprise Resource Planning (ERP), Customer Relationship Management (CRM)
      - SAP, Baan, Peoplesoft, Siebel, Oracle, ...
    - Transaktionssysteme basierend auf TP-Monitoren
      - CICS, Encina, Tuxedo, ...
    - Nicht-relationale DBS
      - IMS, ...

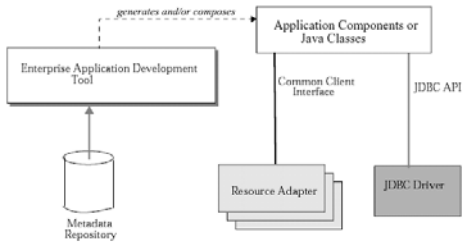
J2EE Connector Architecture (JCA)

- Standardisierte Interoperabilität mit EIS
- Resource Adapter (Konnektor)
  - EIS-spezifische Komponente
  - implementiert Client-Schnittstelle (application contract) zum EIS, nutzbar durch EJBs, Webkomponenten
    - entweder standardisiert (Common Client Interface, CCI)
    - oder EIS-spezifisch
  - kooperiert mit J2EE Anwendungsserver über sog. system-level contracts
    - Verbindungsmanagement, Transaktionen, Sicherheit, …

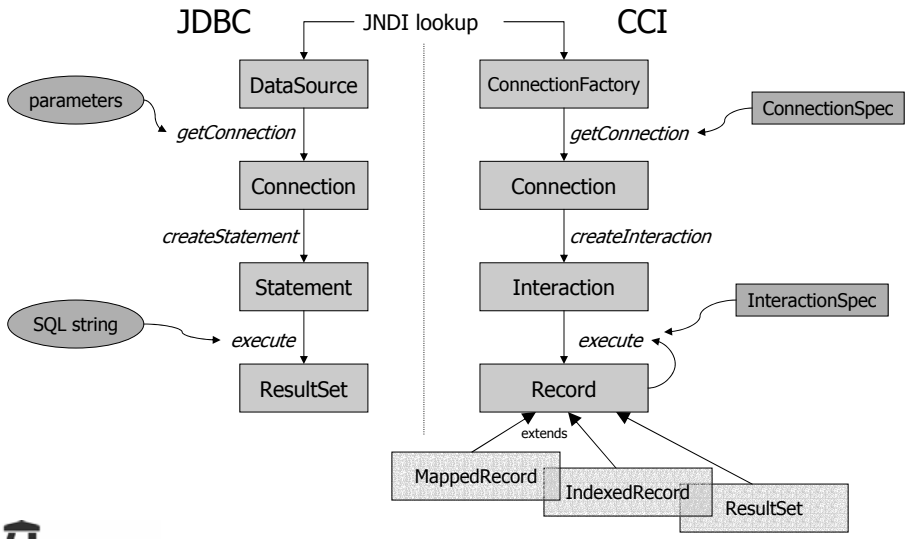


Common Client Interface

- Generische Schnittstelle zum Aufruf von EIS Funktionen (remote function calls)
- Nutzung vor allem durch Anwendungsentwicklungswerkzeuge, EAI Frameworks
  - Generierung von EJB Wrapper-Klassen für EIS Funktionen (vgl. EJB CMP Tooling)
  - Erfordert standardisierte Darstellung von EIS Metadaten
    - nicht im Rahmen der JCA definiert

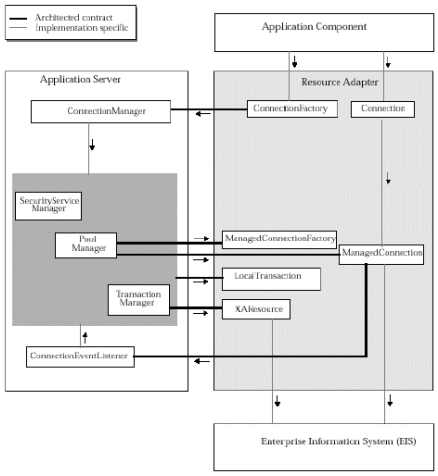


CCI Schnittstellen im Vergleich zu JDBC



JCA System-Level Contracts

- Anwendungsserver implementiert einen ConnectionManager
  - generisch, für beliebige EIS Verbindungen
  - interagiert mit weiteren Diensten des AS
    - connection pooling, transactions, security
- Resource Adapter (RA) erzeugt Verbindungen (ConnectionFactory)
- Anforderung einer Verbindung durch Applikation gelangt über ConnectionFactory zum ConnectionManager
  - PoolManager sucht nach passender Verbindung im Pool, bzw. initiiert Erzeugen einer neuen Verbindung
    - RA hilft bei Auswahl einer geeigneten Verbindung
  - RA hält ConnectionManager über Zustand der Verbindung auf dem Laufenden (ConnectionEventListener)



JCA – Transaktionsmanagement

- Ein Resource Adapter (bzw. der RM des EIS) unterstützt
  - globale Transaktionen
    - Koordination durch TA-Manager des Anwendungsservers
    - XA-compliant (RA implementiert XAResource-Schnittstelle)
      - one-phase optimization durch TA-Manager beim Zugriff auf eine einzige Resource
  - lokale Transaktionen
    - erlaubt Umgehen des globalen TA-Managers aus Performanzgründen, falls im voraus (zum Zeitpunkt des deployments) klar ist, dass keine globalen TAs benötigt werden
  - keine Transaktionen

JCA – Security

- Sicherheitsarchitektur erlaubt folgende Möglichkeiten zur Feststellung des sog. *resource principal*
  - Component-managed sign-on
    - die Anwendungskomponente legt resource principal (evtl. dynamisch) fest
  - Container-managed sign-on
    - resource principal, sign-on information (z.B. useride, password) zum Zeitpunkt des deployments für EIS beschrieben
    - Optionen
      - configured identity: resource principal festgelegt, unabhängig vom initiating principal
      - principal mapping: resource principal wird abhängig vom initiating principal aufgrund einer Abbildung bestimmt, erbt keine zusätzlichen Sicherheitseigenschaften vom initiating principal
      - caller impersonation: Identität, credentials des Aufrufers werden an das EIS delegiert
      - credentials mapping: Abbildung über security domains hinweg
- Abstimmung des verwendeten Authentisierungsmechanismus
  - BasicPassword, KerbV5, ...
- Zugriffskontrolle kann durch EIS oder Anwendungsserver durchgeführt werden
- Kommunikationssicherheit durch Aufbau von Secure Associations zum RA

JCA – weitere System Level Contracts

- Lifecycle Management
  - Start, Stop, Wiederanlauf eines resource adapters im Addressraum des Anwendungsservers
- Work Management
  - erlaubt Nutzung des anwendungsserverseitigen Thread-Managements zur Abwicklung von Aufgaben des RA
- Inbound Communication
  - EIS kann über den RA Anwendungskomponenten aufrufen
    - EJBs
    - Messaging
  - Schließt das Propagieren von externem TA-Kontext durch den RA ein
    - TA wird durch einen TA-Manager ausserhalb des J2EE Anwendungsservers koordiniert

Konnektoren – Zusammenfassung

- Ziel: Einbinden von existierenden EIS als zusätzliche Resource Manager in verteilte Komponentenumgebung über sog. Resource Adapter
  - Einheitliches Verbindungsmodell, Sicherheitsmodell für aufrufende Anwendungskomponenten
  - Einheitlich Aufrufschnittstelle zu beliebigen EIS
    - Werkzeugunterstützung, kombiniert mit Metadatenverwaltung
- Standardisierte Schnittstellen, Interaktionen
  - Der gleiche RA ist in allen J2EE-konformen Umgebungen einsetzbar
  - J2EE-Server realisiert benötigte Infrastruktur nur einmal, für beliebige EIS
- Wichtiges Architekturkonzept für Enterprise Application Integration
  - große Anzahl von J2EE-Konnektoren für verschiedene EIS auf dem Markt erhältlich