Modul i-ch223
Multi-User-Applikation objektorientiert realisieren

Autor
Stephan Metzler

Version 4.0

Index

Dieses Skript vermittelt die Theorie von Enterprise JavaBeans als Applikationsframework zum Erstellen von verteilten Applikationen.

Der Inhalt ist gegliedert in:

A : Java Community

- Theorieaspekt der verteilten Systeme und Übersicht über das JEE Applikationsframework

B : Enterprise JavaBeans

- Betrachtungen der einzelnen EJB Ausprägungen

C : JPA Java Persistence API

- Entity Beans mit Hibernate Persistence Provider (JBoss AS)

D : Best Practice

- Diskussion einiger JEE Idioms und weiterführende Beispiele mit dem JBoss AS

E : Aufsetzen und Konfiguration der Entwicklungsumgebung

- Installationsanleitung für die begleitenden Beispiele

F : Anhang

- Referenzen, Literaturangaben, Online-Quellen sowie die verwendeten Software Versionen

Inhaltsverzeichnis Seite

F Anhang 221

A Java Community

1 Java Plattform

Die Java Plattform ist in drei Editionen aufgeteilt:

JME **Micro Edition**	**JSE** **Standard Edition**	**JEE** **Enterprise Edition**
ME definiert eine robuste, flexible Umgebung für Applikationen in: - consumer devices wie Mobil-Telefone, PDAs - embedded devices ME beinhaltet eine VM und Standart Java APIs für Kommunikationsprozesse.	Die SE definiert: - JRE, Runtime Environment beinhaltet APIs, VM, etc. - JDK, Development Kit beinhaltet JRE sowie Compiler, Debugger, etc. Stellt die Basis für EE dar.	EE definiert einen Standart um Multi-Tier Enterprise Applikationen zu entwickeln. Stellt verschiedene Dienst-leistungen den einzelnen Komponenten zur Verfügung und bietet transparente Lösungen für dessen vielseitiges Verhalten.

1.1 Verteilte Systeme

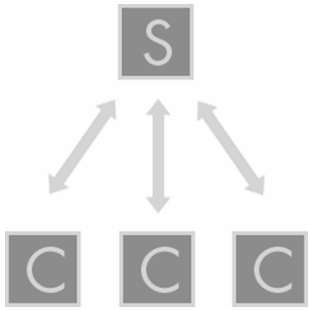
Ein verteiltes System besteht aus mehreren Prozessen, die mittels Nachrichtenaustausch miteinander kommunizieren. Dabei sind wichtige Kriterien:

<u>Gemeinsame Nutzung von Ressourcen</u>

- Zugriff, Verzögerung und Updates müssen zuverlässig und konsistent sein

<u>Offenheit</u>

- Erweiterbarkeit des Systems (Hardware, Software)
- Standardisierung von Schnittstellen erforderlich
- Interprozesskommunikation muss unterstützt werden
- heterogene HW- und SW- Komponenten sind Bestandteil eines offenen Systems



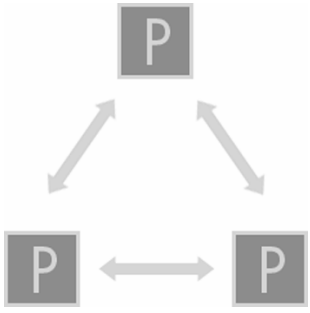
Client Server Architektur

<u>Skalierbarkeit</u>

- Skalierung des Systems ohne Änderungen der System- oder Anwendungssoftware

<u>Fehlertoleranz</u>

- Hardwareredundanz, Softwareredundanz



Peer-to-Peer Architektuer

1.2 OOA Thematik

1.2.1 Java 5

hilfreiche und wichtige Neuerungen im Java-Standard:

1.2.2 Annotations

- Metadaten direkt im Sourcecode
- XML- oder Property-Dateien entfallen
- werden von Codegenerierungstools ausgelesen
- werden zur Laufzeit per Reflection abgefragt
- zentral im Sourcecode abgelegt (Entwickler Vorteil)
- Zuordnung zum Sourcecode ersichtlich

Verwendung von Annotations

- Annotations werden im Sourcecode vor dem zu markierenden Element eingefügt
- @ als Kennzeichnung vorangestellt (Leerzeichen sind erlaubt)
- Parameter in Klammern an den Annotationsnamen angehängt

Annotationsparameter werden als Name-Wert-Paar angegeben, wobei die Reihenfolge der Parameter keine Rolle spielt:

```
@MyAnnotation(parameter1 = "hello", parameter2 = "jee")
```

Hat die Annotation nur einen Parameter, kann der Name weggelassen werden. Bei parameterlosen Annotations ist die Angabe der Klammern optional.

Annotations können sich auf folgende Java-Elemente beziehen:

- Packages
- Klassen
- Interfaces
- Enumerations
- Methoden
- Variablen
- Methodenparameter

Annotations in der Java SE 5.0

- definiert sieben Annotations
  - Standard Annotation (normale Verwendung beim Programmieren)
  - Meta Annotation (Definition neuer Annotationstypen)

Standard Annotations:

- @Deprecated:
  kennzeichnet Methoden und Klassen, die nicht mehr verwendet werden sollen.
- @Override:
  der Compiler stellt sicher, dass die Methode der Basisklasse überschrieben wird.
- @SuppressWarnings:
  Unterdrückt Compilerwarnungen. Warnungen werden als Parameter angegeben.

```
/**
 * @module jee.2007.Metzler
 * @exercise jee010.util.HelloWorld.java
 */
public class HelloWorld {
  @Deprecated
  public String sayHello() {
    return "Hello World";
  }
  @Override
  public String toString() {
    return "Hello World";
  }
}
```

- sayHello() ist veraltet und erzeugt Compilerwarnung
- toString() mit @Override stellt sicher, dass toString() aus Object überschrieben wird

Meta Annotations:

- @Documented
  - zur Erzeugung der Dokumentation bei Javadoc
- @Inherited
  - Annotation-Typ wird vererbt, auch rekursiv, falls keine Annotation in der aktuellen Klasse gefunden wird.
- @Retention
  - Definiert wie lange die Annotation verfügbar ist
    - SOURCE – nur bis zur Compilezeit zur Verfügung
    - CLASS – werden in den Class-Dateien abgespeichert, aber nicht von der VM geladen
    - RUNTIME – werden in der Class-Datei abgelegt und von der VM geladen und stehen somit zur Auswertung per Reflection zur Verfügung
- @Target

- definiert Elemente (Klasse, Methode, Parameter etc.) denen eine Annotation zugeordnet werden kann

Beispiel selbst definierter Annotation

```
/**
 * @module jee.2007.Metzler
 * @exercise jee010.util.Datentyp.java
 */
public enum Datentyp {
  TEXT, NUMMER, DATUM
}
```

```
/**
 * @module jee.2007.Metzler
 * @exercise jee010.util.Datenfeld.java
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Datenfeld {
  String bezeichnung() default "";
  Datentyp datentyp() default Datentyp.TEXT;
}
```

- @Retention(RetentionPolicy.RUNTIME) definiert die Verfügbarkeit
  - RUNTIME : Auswertung zur Laufzeit
  - CLASS : (DEFAULT) – Eintrag in der *.class Datei
  - SOURCE : verfügbar nur zur Compilezeit
- @Target({ElementType.FIELD}) definiert wo die Annotation eingesetzt werden kann
  - TYPE : auf Klassenebene
  - FIELD : für Attribute
  - METHODE : auf Methodenebene
- Parameter
  - Bezeichnung : optional, Default = ""
  - Datentyp : optional, Default = Datentyp.TEXT

Anwendung: Klasse Person annotiert Attribute als Datenfelder

```
 /**
 * @module jee.2007.Metzler
 * @exercise jee010.util.Person.java
 */
@SuppressWarnings( { "serial", "unused", "unchecked" })
public class Person implements java.io.Serializable {
  @Datenfeld(bezeichnung = "Nummer", datentyp = Datentyp.NUMMER)
  private int    nr;
  @Datenfeld(bezeichnung = "Name", datentyp = Datentyp.TEXT)
  private String name;
  @Datenfeld
  private String beschreibung;
}
```

Klasse PersonClient liest die Annotationen aus

```
import java.lang.reflect.Field;
/**
 * @module jee.2007.Metzler
 * @exercise jee010.client.PersonClient.java
 */
public class PersonClient {
  public static void main(String[] args) {
    Person p = new Person();
    Field[] fields = p.getClass().getDeclaredFields();
    for (Field field : fields) {
      Datenfeld datenfeld = field.getAnnotation(Datenfeld.class);
      if (datenfeld != null) {
        System.out.println("\nannotiertes Attribut: "
            + field.getName());
        if (datenfeld.bezeichnung().length() != 0) {
          System.out.println("\tBEZ = " +
                             datenfeld.bezeichnung());
        }
        System.out.println("\tTYP = " + datenfeld.datentyp());
      }
    }
  }
}
```

Ausgabe

```
annotiertes Attribut: nr
  BEZ = Nummer
  TYP = NUMMER

annotiertes Attribut: name
  BEZ = Name
  TYP = TEXT

annotiertes Attribut: beschreibung
  TYP = TEXT
```

1.2.3 Generics

- erlauben die Abstraktion von Typen
- defineren generische Klassen und Methoden, unabhängig von einem konkreten Typ

```
List v = new Vector();
v.add(new Double(1.0));
Double d = (Double)v.get(0);
```

- Cast auf Double ist notwendig denn der Rückgabewert von get(...) ist Objekt
- erlaubt inkompatible Typen in die Liste einzuführen

generische Collections:

```
List<Double> v = new Vector<Double>();
v.add(new Double(1.0));
Double d = v.get(0);
```

- v ist eine Liste von Double
  - Typ in spitzen Klammern definiert den konkreten Typparameter
- Verwendung wird durch Compiler sichergestellt und zur Compilezeit erkannt
- Casts entfallen, da Rückgabewert von get(...) nur Double
- mehrere Typparameter werden durch Komma getrennt (z. B. Hashtable<Long, String>)

Generics verbessert die Lesbarkeit und hilft durch die erhöhte Typsicherheit die Zuverlässigkeit des Codes zu erhöhen.

Wildcards und Bounds

- unabhängiger Code
- Implementierung von Such- oder Sortieralgorithmen
- Wildcardzeichen ist ?
  - Vector<?> steht für einen Vector mit beliebigem Inhalt
- Bounds als Angabe von Schranken
  - extends limitiert auf den angegebenen oder abgeleiteten Typ
  - super limitiert auf den angegebenen oder vererbten Typ

Beispiel:

```
Vector<?> v1;
Vector<String> v2 = new Vector<String>();
v1 = v2;                          // nur String
List<? extends Number> 100;       // Number oder Subtyp
List<? super String> 2;           // String oder Supertyp
```

generische Typen

- Generics sind nicht auf die Verwendung bestehender Klassen beschränkt.
- eigene generische Typen können definiert werden.
- Definition der generischen Klasse MyGeneric, welche zwei Typparameter (T und U) enthält. Die Typparameter können wie normale Typen bei der Definition der Klasse verwendet werden.

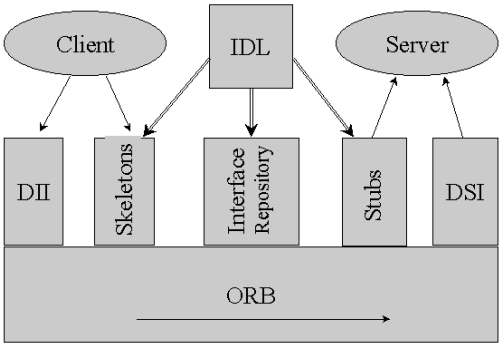
```
/**
 * @module jee.2007.Metzler
 * @exercise jee010.util.MyGeneric.java
 */
public class MyGeneric<T, U> {
  private T key;
  private U value;
  public MyGeneric(T key, U value) {
    this.key = key;
    this.value = value;
  }
  public T getKey() {
    return key;
  }
  public U getValue() {
    return value;
  }
}
```

Generische Methoden

- Definition von generischen Methoden
- Definition der generischen Methode sayHello(...), die zwei Typparameter enthält.

```
/**
 * @module jee.2007.Metzler
 * @exercise jee010.util.GenericMethod.java
 */
public class GenericMethod {
  public static <E, T> void sayHello(E pre, T post) {
    System.out.println(pre.toString() + " hello "
        + post.toString());
  }
  public static void main(String[] args) {
    sayHello("generic", new StringBuffer("world"));
  }
}
```

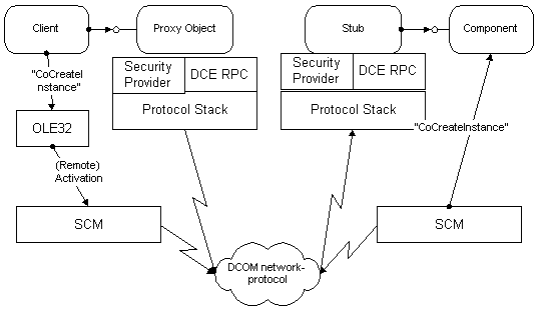
1.3 Technologien für Verteilte Systeme



CORBA

(Common Object Request Broker Architecture):

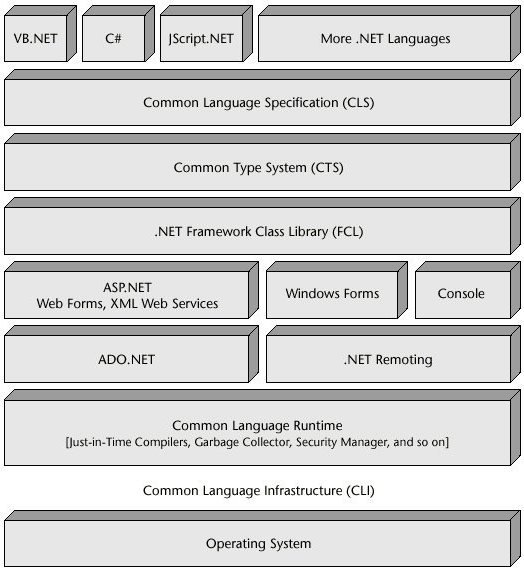
- Plattform- und Sprachunabhängig
- nur "basic distribution"



DCOM

(Distributed COM) Dienste zur Kommunikation zwischen Prozessen

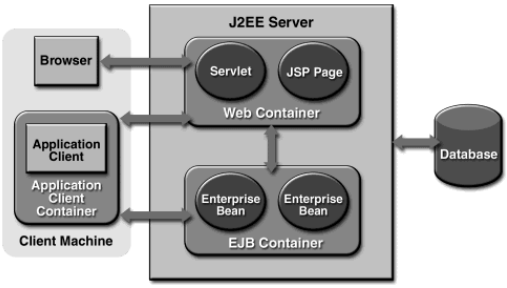
- fokussiert auf Microsoft
- CORBA-DCOM Bridge



.NET

(Microsoft)

- Sprach- und (Plattform-) unabhängiges Framework für Verteilte Systeme



EJB

(Enterprise Java Beans)

- nur JAVA
- entwickelt für Geschäftsprozesse
- Sicherheits- und Verbindungslogik ist integriert

2 JEE Applikationsframework

JEE bezeichnet ein Applikationsframework, ausgelegt auf:

- Performance
- Skalierbarkeit
- Plattformunabhängigkeit

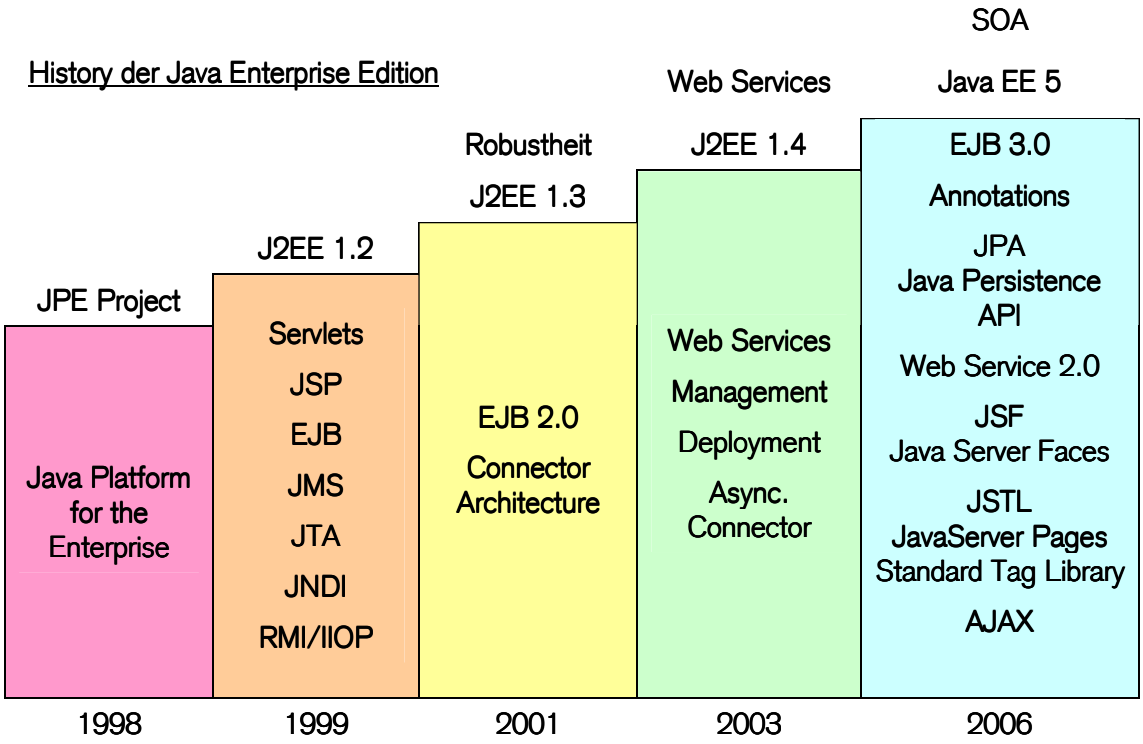
Die Trennung zwischen Client-Tier, Business-Tier und Data-Tier ist sehr sauber implementiert und erlaubt eine weitgehende Unabhängigkeit von den verwendeten Clients und den Systemen zur Datenhaltung im Backend.

JEE Applikations-Server beinhaltet:

- **Components**
  Applets, Servlets, Enterprise Java Beans
- **Containers**
  Web Browsers, Servlet Engines, EJB Containers
- **Resources**
  SQL Datenquellen, Nachrichtensysteme, Mailsysteme

2.1 History

- J2EE 1.2 – 1999
  - Zusammenfassung bestehender Java Technologien zu einem Standard für Applikation Server
  - Trennung von Entwicklungs und Laufzeitaspekten über Deklarationen
- J2EE 1.3 – 2001
  - Detailverbesserungen bei Web Technologien und EJBs
  - Java Connector Architecture als der Standard für Legacy Integration
  - XML Support als Bestandteil der Plattform
- J2EE 1.4 – 12.2003
  - Web Services als integraler Bestandteil der Plattform
  - Vollständiger Support für WS-I Basic Profile
  - Einführung von Management und Tooling APIs
- JEE 1.5 – 05.2006
  - zurück zum POJO, POJI
  - Konvention statt Konfiguration (auf Standard basierend)
  - Reduktion der notwendigen Artefakte (Interfaces, Callbackmethoden, Deploymentdescriptoren)
  - Nutzung von Annotationen für Metadaten (DD für Rollentrennung)
  - Injektion von Ressourcen durch DI (Dependency Injection)
  - Verbesserung der Datenbankabfragen (JPA als Spezifikation)
  - Verbesserung der Testbarkeit (lauffähig ausserhalb AS)
  - Nutzung von gemeinsamem Code durch Interceptoren (Crosscutting Concern)



2.2 Bestandteile

- neue XML/Web Services
  - JAX-B (JSR-222)
  - JAX-WS (JSR-224)
  - StAX (JSR-173)
  - Web Services Metadata (JSR-181)

- neue Web Technologien
  - JSP Standard Tag Library (JSR-52)
  - Java Server Faces 1.2 (JSR-252)

- neue EoD (Ease Of Development = Entwicklungserleichterung)
  - EJB 3.0 und Java Persistence API (JSR-220)
  - Common Annotations (JSR-250)

- geringfügige Anpassungen
  - Java Server Pages / Servlet API
  - JavaMail
  - Java Activation Framework
  - Java Authorisation Contract for Containers
  - Implementing Enterprise Web Services (JSR-109)
  - Management & Deployment API (JSR-77/88)

- Security Änderungen
  - Gleichsetzung der Security Permissions von Web und EJBContainer

- Unterstützung für shared Libraries in Enterprise Archiven
  - Bundled Libraries
    über Class-Path header im Manifest der JAR Files oder über <librarydirectory> Element mit Angabe eines Directories für shared .jar Files im .EAR
  - Installed Libraries
    Spezifikation von Abhängigkeit zu installierten .jar Files per Extension-List Attribut im Manifest eines Archivs

- Deployment Deskriptoren
  - sind eine der Stärken von J2EE
  - Grundlage: J2EE Rollenverständnis: Comp.Developer, Assembler, Deployer
  - Idee nachträglich Settings/Verhalten anpassen zu können
  - Deskriptoren erhöhten Komplexität
  - Nur durch Tools(IDEs,Xdoclet) beherrschbar
  - Migrationshindernis

2.3 Einsatzgebiete

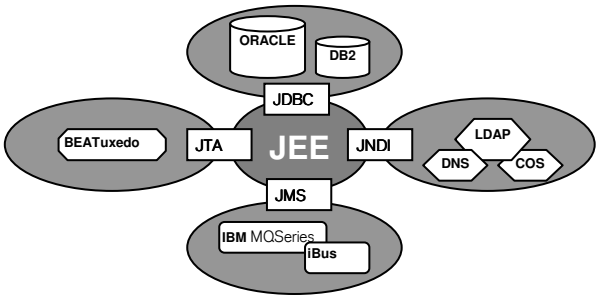
JEE Applikationen finden in den folgenden Gebieten ihren Schwerpunkt:

- Datenbankzugriff
- Dynamische Internetseiten
- Enterprise JavaBeans
- Middle-Tier Server
- Plattformenabhängige Kommunikation
- Sicherheit
- Trennung der Anwenderschwerpunkte
- Verteilte Systeme
- Web-Server-Einsatz

2.4 Architektur

Funktionelle Einheiten werden als Komponenten bezeichnet und zu einer Komponenten-Architektur zusammengefasst. Dabei dominieren die folgenden "Core Technologies":

- **JNDI** (Java Naming and Directory Interface)
  - Zugriff auf "naming" und "directory" Systeme wie LDAP, COS oder DNS
- **JTA** (Java Transaktion API)
  - Zugriff auf Transaction Systeme wie BEA Tuxedo, etc.
- **JDBC** (Java DataBase Connectivity)
  - Zugriff zu relationalen Datenbanken
- **JMS** (Java Messaging Service)
  - Zugriff zu Message Orientated Middleware, z.B. iBus ,IBM MQSeries



2.5 Namenskonventionen

- die Namenskonventionen sind mit JSR 220 nicht mehr so dominant
- orientiren sich am Namensmodell von SUN™
- Ansatz basierend auf einen HelloWorld EJB:

Item	Syntax	Beispiel
Enterprise Bean Name (DD)	<name>Bean	HelloWorldBean
EJB JAR Display Name (DD)	<name>JAR	HelloWorldJAR
Enterprise Bean Class	<name>Bean	HelloWorldBean
Business / Remote Interface	<name>	HelloWorld
Local Business / Remote Interface	<name>Local	HelloWorldLocal
Abstract Schema (DD)	<name>	HelloWorld

2.6 Komponenten

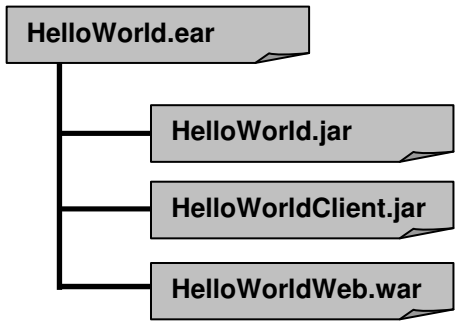
Eine JEE-Anwendung ist eine JEE-Komponente, die über einen JEE-Server verschiedenen Clients zur Verfügung gestellt wird. Die Bestandteile einer JEE-Anwendung werden in eine *.ear Datei (**e**nterprise **ar**chive) gepackt und können aus den folgenden Komponenten bestehen:

- **Enterprise Beans**
  eine oder mehrere Enterprise Beans in einem *.jar Datei (java archive)

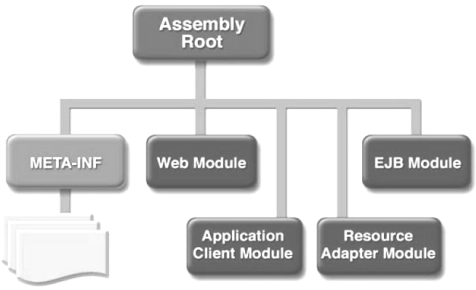
- **WEB-Anwendungen**
  HTML-Seiten, Servlets, JSP, gif- und jpg-Bilder in einer *.war Datei (web application archive)

- **Client-Anwendungen**
  Stand-Alone Java Applikation in einer *.jar Datei (java archive)

- **Deployment Descriptors**
  XML-Dokumente zur Integrationsbeschreibung (sind optionale Bestandteile jeder EAR Komponente)



Das Enterprise Archive folgt nebenstehender Struktur die dem Applikationsserver das Entpacken der Komponenten und das Deployen der JEE Applikation ermöglicht:

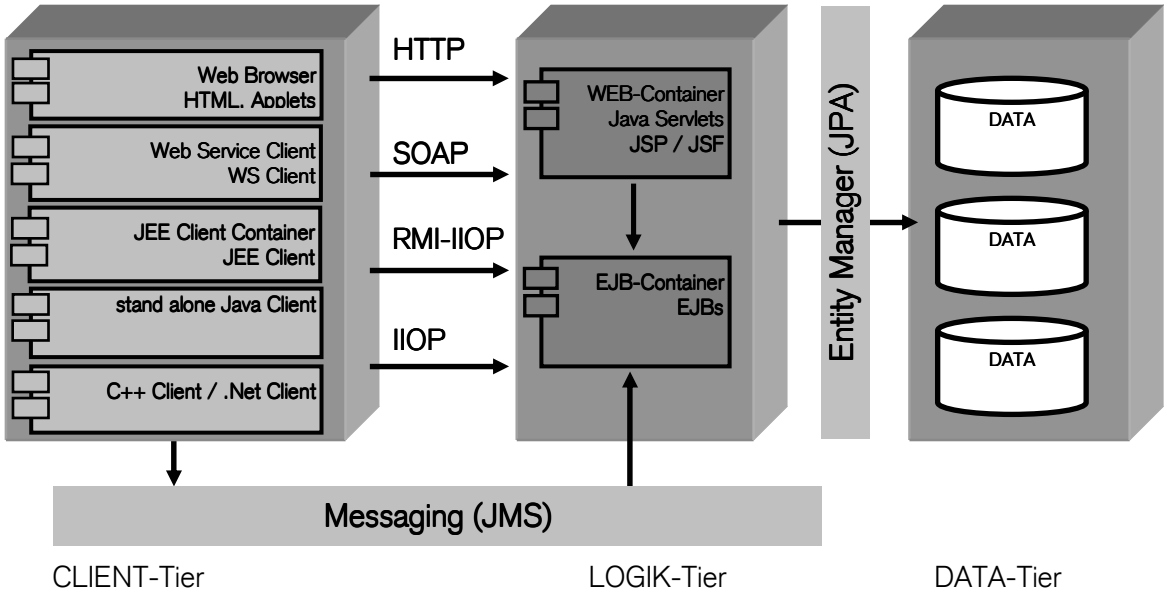


2.7 Multi-Tier Applikation

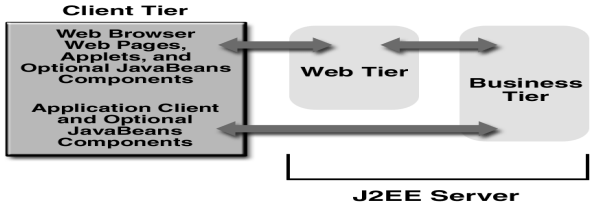
Das Verteilen einer Applikation richtet sich meistens nach dem "seperation of concern" Aspekt der OOA.

3 Schicht Architektur

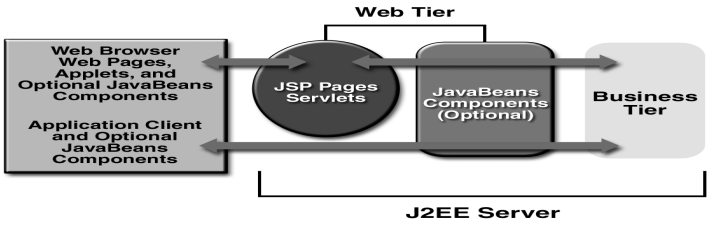
RMI = Remote Method Invocation
IIOP = Internet Inter ORB Protocol



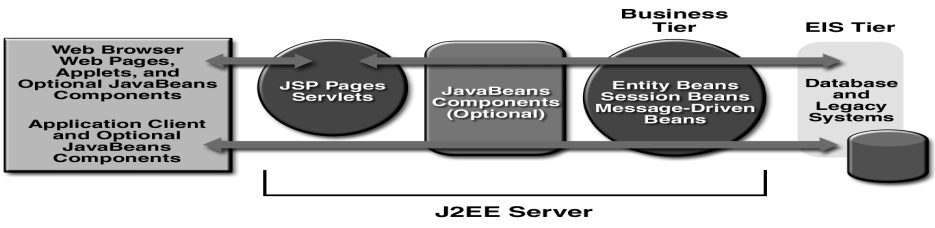
JEE Server mit WEB- und BusinessTier



JEE Server mit JavaBean Komponenten als Interface



JEE Server mit WEB-, BusinessTier und EIS Tier



B Enterprise JavaBeans

3 Enterprise JavaBeans

EJBs sind als Komponenten konzipiert:

- wiederverwendbar
- verteilt einsetzbar
- lose gekoppelt

Verteilungsaspekt (Seperation of Concern) ermöglicht:

- Trennung der Anwendung (Geschäftslogik)
- Bedienerführung (Darstellungslogik)
- Datenerhaltung (Persistenz)

3.1 Definition

Development,	Entwicklung,
Distribution	Verteilung
and	und
Deployment	**Umsetzung** (von Modellen) Militär: Truppenbereitstellung / Installation
of	
Java-Based	Java basierten
Server-Side	serverseitigen
Software Components	Software Komponenten

EJB Technologie definiert

- Interfaces
- Patterns
- Richtlinien

für komponenten-orientierte, verteilte Systeme.

EJB Komponente

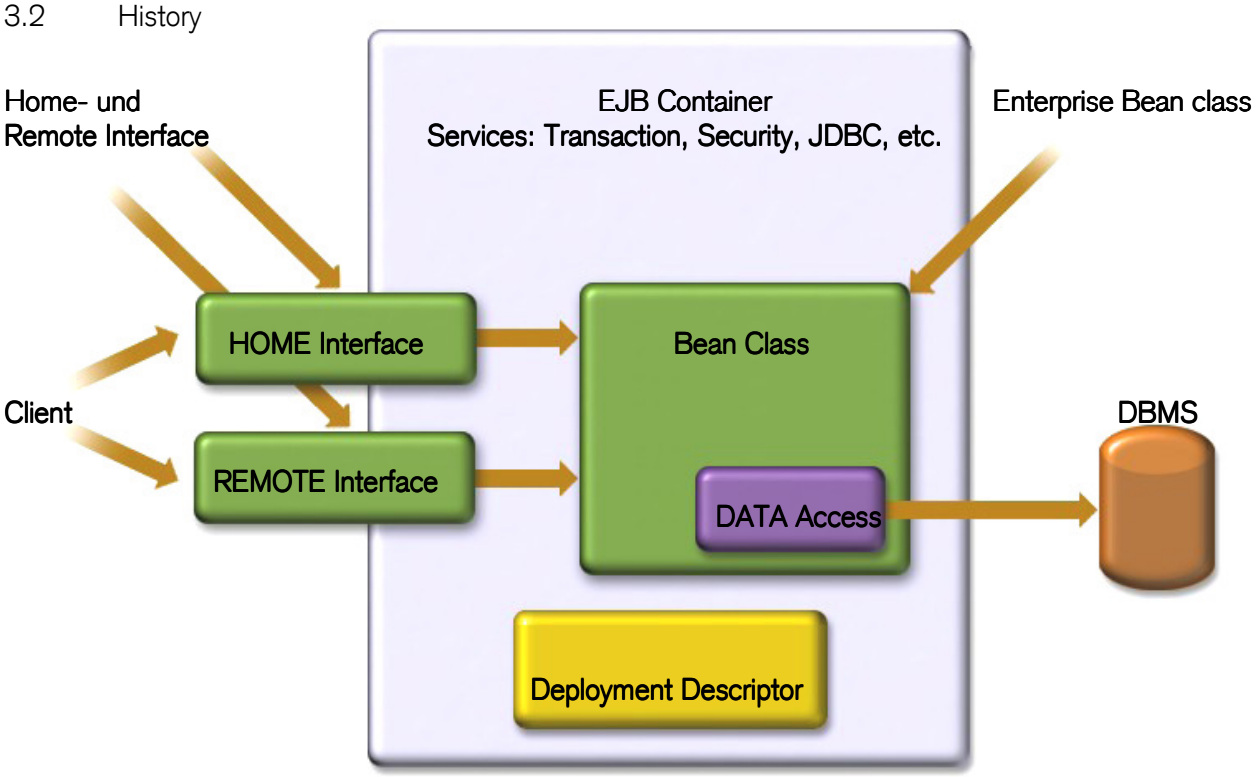
- definiert Funktionalität
- Black Box
- Zugriff nur über Schnittstelle
- Wiederverwendbarkeit

http://www.ejip.net/images/ejb.jpg

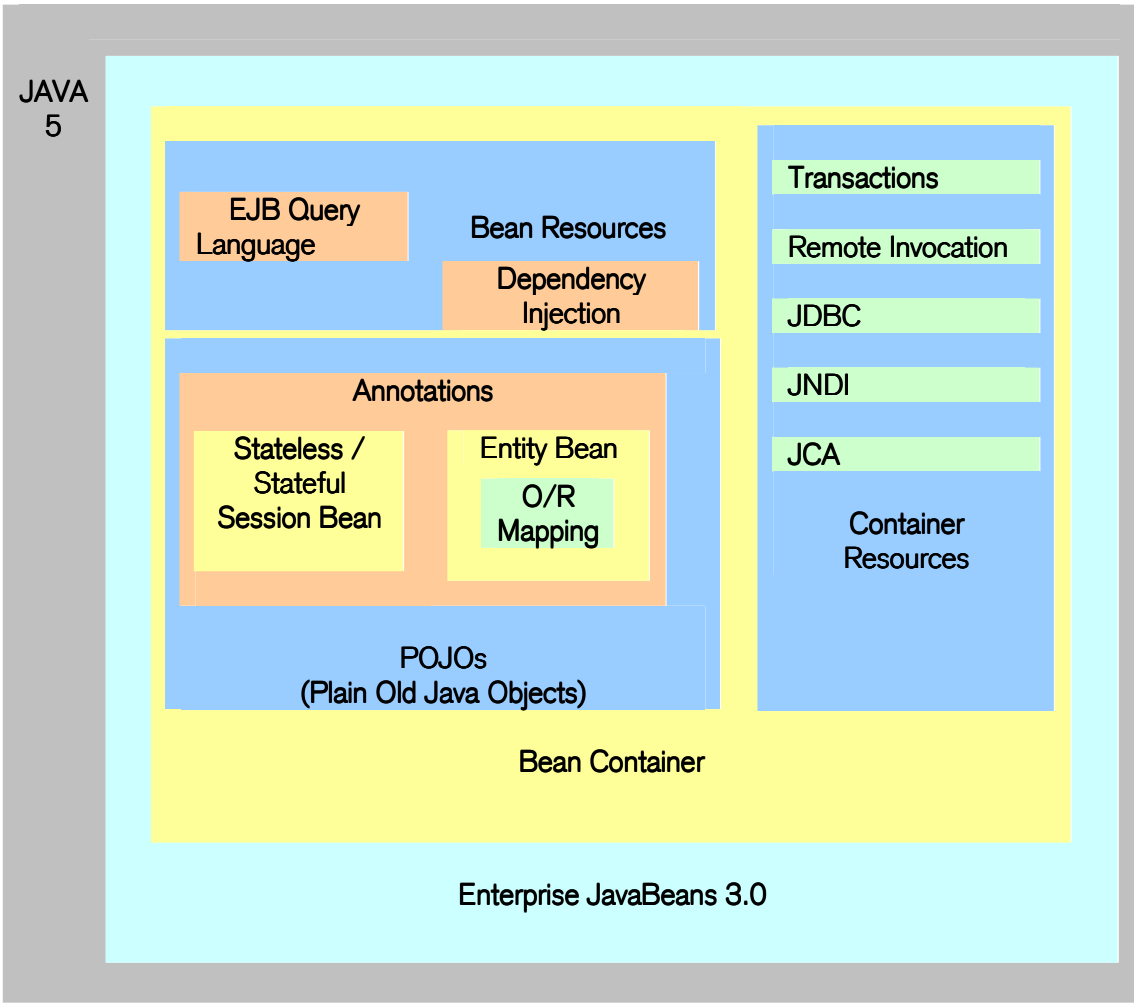
EJB Ziele

- Entwicklung von Geschäftsprozessen (keine Systemprozesse)
- Standard für Komponenten basierte Entwicklung
- Wiederverwendbarkeit

3.2 History

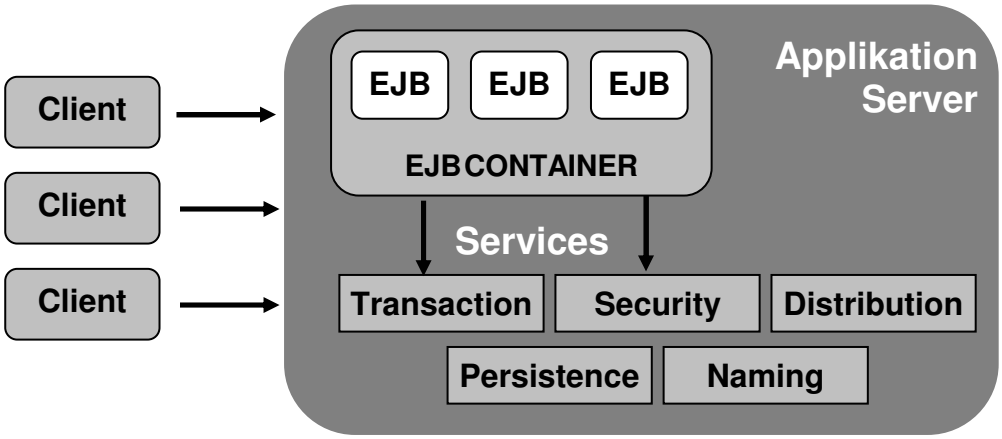


3.3 POJO

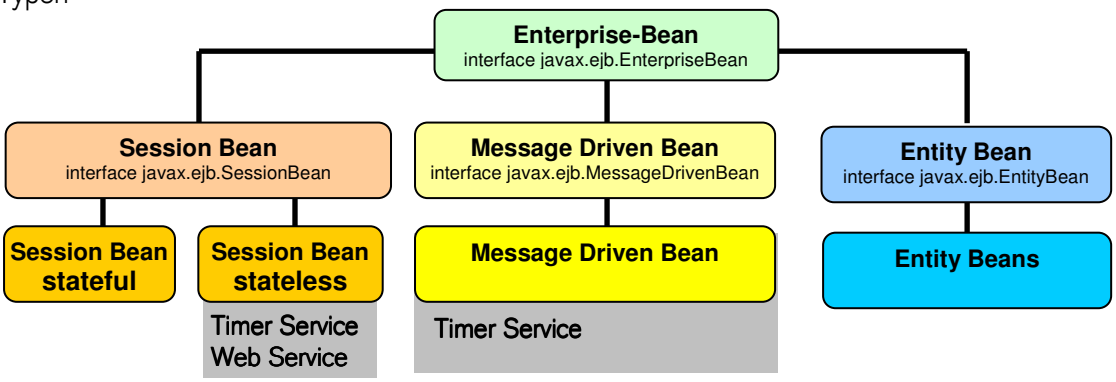


3.4 Architektur

- EJBs existieren innerhalb von Containern
- Applikations-Server stellt Dienste zur Verfügung



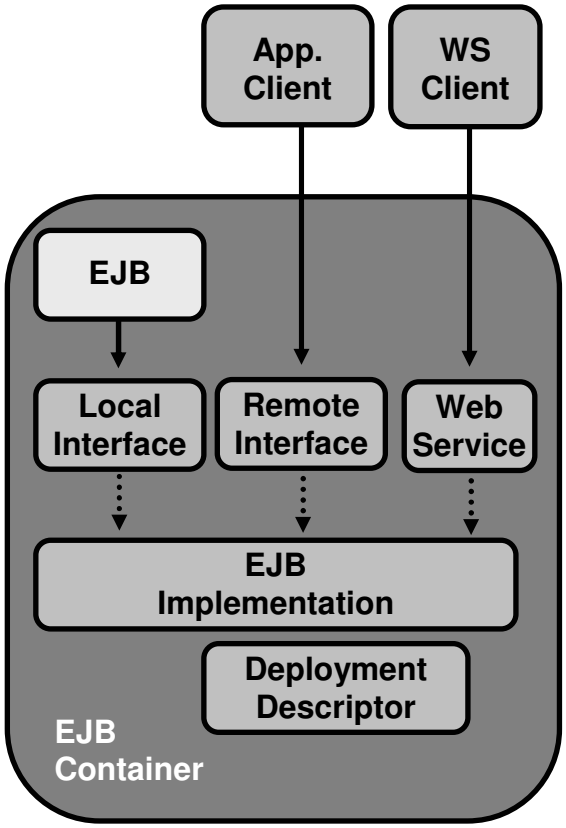
3.5 Typen



	Session Bean	Message Driven Bean	Entity Bean
Zweck	Stellt einen Prozess für einen Client dar, z.B. Geldautomat	Stellt einen Nachrichten-empfänger dar.	Stellt Daten dar. Entitäten oder Objekte im persistenten Speicher (DB)
Ausprägung	**Stateless** Session Bean **Stateful** SessionBean	Message Driven Bean	Entity Bean
Gemeinsame Nutzung	**Stateful** Session Bean dedizierte Client Zuordnung **Stateless** Session Bean hat keine Client Zuordnung	keine direkte Client-zuordnung, "horcht" auf einem JMS-Kanal	Gemeinsame Nutzung durch mehrere Clients möglich
Persistenz	**Transient** Lebenszyklus typischerweise durch Clientprozess bestimmt	**Transient** "stirbt" bei Container Terminisierung	**Persistent** Zustand bleibt im persistenten Speicher (DB) auch nach Container Terminierung erhalten

3.6 Bestandteile

Die EJB stellt ein POJO (plain old Java Object) dar und besteht aus folgenden Komponenten:



Local Interface

- definiert Zugriff (Interface)
  für einen lokalen Client (EJB im Kontainer)
  - Datei: server/HelloWorldLocal.class

Remote Interface

- definiert Zugriff (Business Interface),
  für einen remote Client
  - Datei: server/HelloWorld.class

Web-Service Interface

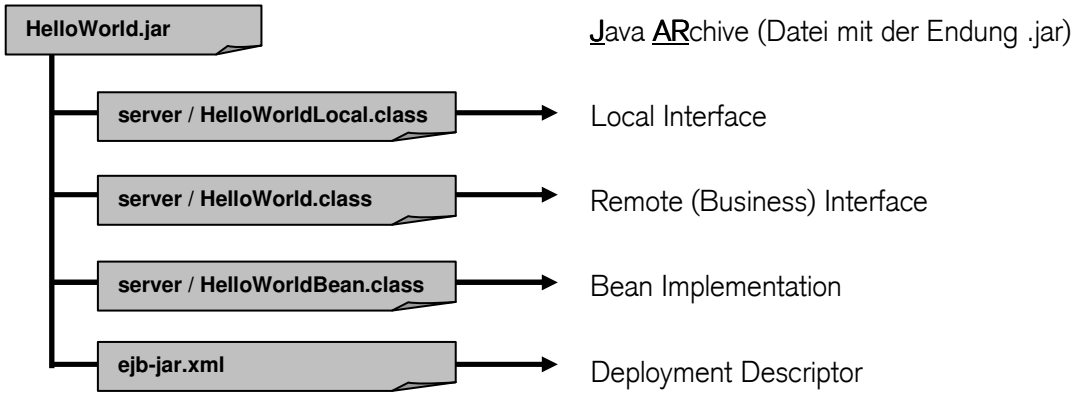
- definiert Zugriff (Business Interface),
  als Web Service
  - Datei: server/HelloWorldWS.class

Bean Implementierung

- definiert und implementiert alle Methoden,
  die in den Interfaces deklariert sind
  - Datei: server/HelloWorldBean.class

Deployment Descriptor

- XML Datei zur Beschreibung der Komponente
  (Dateiangaben, Pfadangaben, Datenbankverbindungen und das Verhalten, z.B:
  Zugriffskontrolle, Transaktionsverhalten)
  - Datei: ejb-jar.xml



4 Hello EJB3 World

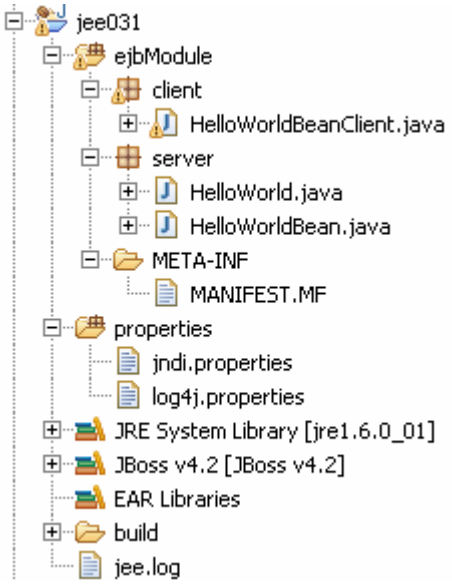
Aufbau, Installieren und Betreiben einer JEE Applikation

4.1 Voraussetzungen

- JSDK: Java Software Development Kit (min. Version 1.5)
- JEE Application Server: Sun GlassFish, **JBoss**, Oracle AS, OpenJPA, Cayenne, …
- Entwicklungsumgebung: **Eclipse**, Java Editor Tool, etc

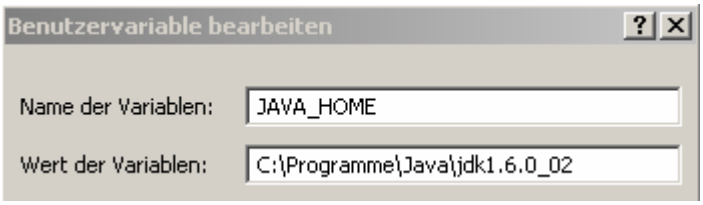
4.2 Organisation

/build	generierte Class Dateien
/properties	Eigenschaftsdateien
/ejbModule	EJB Source Dateien
/client	Source der Client Tier
/server	Source der Middle Tier (EJBs, Servlets und HelperClasses)

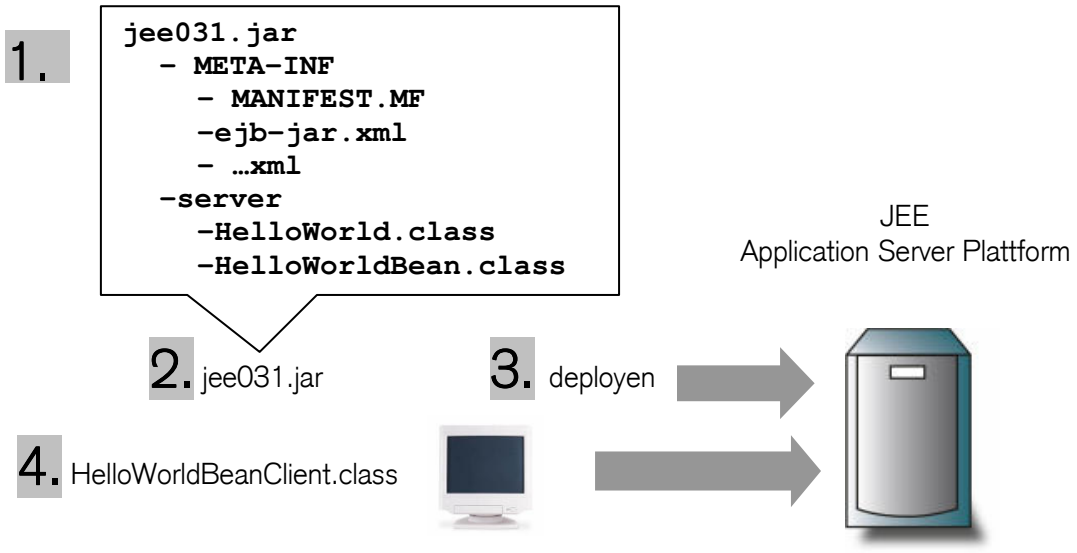


Umgebungsvariablen

Benutzer- oder Systemvariablen definieren oder ergänzen:



4.3 Ziel



4.4 Hello EJB World Java Source Dateien

- das **Remote (Business) Interface** `HelloWorld.java`
- im Verzeichnis `...\jee031\ejbModule\server`

```
package server;

import javax.ejb.*;

/**
 * @module jee.2007.Metzler
 * @exercise jee031.server.HelloWorld.java
 */
@Remote
public interface HelloWorld {
  public String echo(String echo);
}
```

- die **Enterprise BEAN Klasse** `HelloWorldBean.java`
- im Verzeichnis `... \jee031\ejbModule\server`

```
package server;

import javax.ejb.*;

/**
 * @module    jee.2007.Metzler
 * @exercise jee031.server.HelloWorldBean.java
 */
@Stateless
public class HelloWorldBean implements HelloWorld {
  @Override
  public String echo(String echo) {
    return "Server Echo : " + echo;
  }
}
```

- die **Client** Applikations-Klasse `HelloWorldBeanClient.java`
- im Verzeichnis `...\ jee031\ejbModule\client`

```
package client;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import server.HelloWorld;

/**
 * @module jee.2007.Metzler
 * @exercise jee031.client.HelloWorldBeanClient.java
 */
public class HelloWorldBeanClient {
  public static void main(String[] args) {
    try {
      HelloWorld hello = null;
      // http://localhost:8080/jmx-console + service=JNDIView
      // zeigt registrierte Namings
      Context ctx = new InitialContext();
      Object o = ctx.lookup("HelloWorldBean/remote");
      // Zeige Klassen und Interfaces
      System.out.println("CLASS > " + o.getClass().getName());
      Class[] interfaces = o.getClass().getInterfaces();
      for (int i = 0; i < interfaces.length; ++i) {
        System.out.println("implements > "
            + interfaces[i].getName());
      }
      // Instanziierung des Business Interface durch narrowing
      hello = (HelloWorld) PortableRemoteObject.narrow(o,
          HelloWorld.class);
      // Instanziierung des Business Interface durch ProxyObject
      hello = (HelloWorld) o;
      // Instanziierung des Business Interface durch NamingContext
      hello = (HelloWorld) ctx.lookup(o.getClass().getName());
      // Zugriff auf serverseitige Business Logik
      System.out.println(hello.echo("Hello EJB3 World"));
    } catch (NamingException e) { e.printStackTrace();
    }
  }
}
```

Bekanntmachen der Serveradresse durch Properties

- als VM Agrumente
- als Propertie Datei
- als Systemparameter

Properties

```
Context initial = new InitialContext(env);

VM Arguments:
 -Dorg.omg.CORBA.ORBInitalHost=$localshost
 -Dorg.omg.CORBA.ORBInitalPort=$3700

System Properties:
System.setProperty("org.omg.CORBA.ORBInitalHost","localhost");
System.setProperty("org.omg.CORBA.ORBInitalPort","3700");

Environment Properties:
  Properties env = new Properties();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.cosnaming.CNCtxFactory");
  env.put(Context.PROVIDER_URL,
     "iiop://localhost:3700");
```

- als (Property) - Datei laden

- Property Datei im ClassPath
  - java.naming.factory.initial
    - Initialisierungsklasse der Factory für den Namensdienst / JNDI Treiber
  - java.naming.factory.url.pkgs
    - Packages, in dem sich die Init-Klassen befinden
  - java.naming.provider.url
    - Adresse des JNDI-Servers

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost:1099
```

4.5 "HelloWorld" Bean deployen

Mit dem Sun™ Deploytool die notwendigen Deployment Descriptoren menügeführt erstellen und die EJB Komponenten im Sun AS™ deployen.

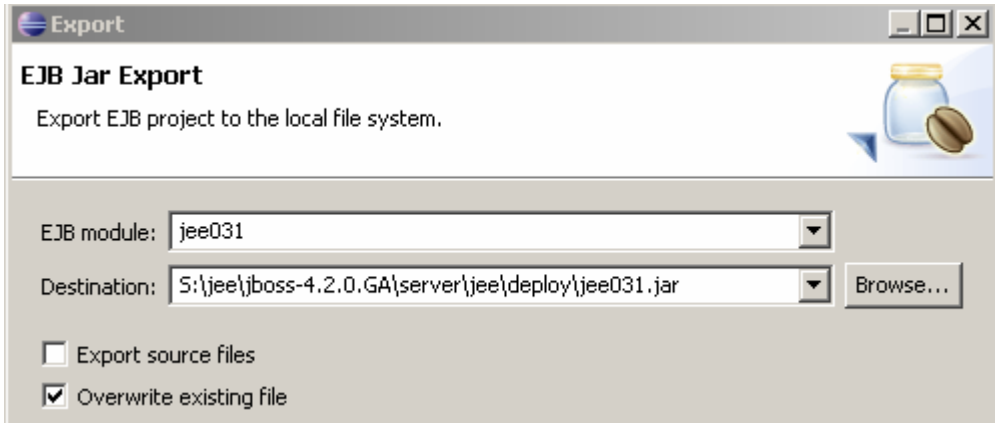
4.5.1 Applikations-Server starten / stoppen

- [JBoss_Home]/bin/run –c jee
- oder PlugIN
  [Show View]+Servers

- Testen ob der Server läuft mit: **`http://localhost:8080`**
- stoppen mit [JBoss_Home]/bin/shutdown –S

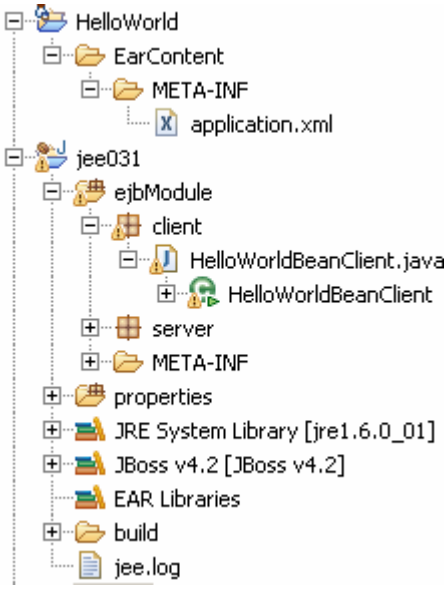
4.5.2 JAR Datei deployen

Java Archive Datei erstellen und ins Verzeichnis [JBoss_Home]/[server]/deploy stellen



<u>JavaARchiv Datei:</u>

```
#  Archive jee031.jar
\server\HelloWorld.class
\server\HelloWorldBean.class
\META-INF\MANIFEST.MF
#
```



<u>oder als EnterpriseARchive</u>

```
#  Archive HelloWorld.ear
META-INF\application.xml
META-INF\MANIFEST.MF
jee031.jar

jee031.jar\server\HelloWorld.class
jee031.jar\server\HelloWorldBean.class
jee031.jar\META-INF\MANIFEST.MF
#
```

- Deployment Descriptoren ergänzen / überschreiben Annotationen

application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:application="http://java.sun.com/xml/ns/javaee/application_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/application_5.xsd"
  id="Application_ID" version="5">
  <module>
    <ejb>jee031.jar</ejb>
  </module>
</application>
```

4.6 Deployment

Das Deployment definiert das serverseitige Bereitstellen der JEE Komponenten. Dazu werden neben den Javakomponenten die Deployment Descriptoren ejb-jar.xml (und weitere) erstellt und gemeinsam in einer EAR Datei dem Applikationsserver übergeben.

Alternativen zum Bereitstellen dieser Komponenten mittels Deploytool sind:

- manuelles Bereitstellen
  JAR Datei erstellen und dem Applikationsserver übergeben
  (Deployment Descriptoren müssen vorhanden und in Jar Datei verpackt sein!)

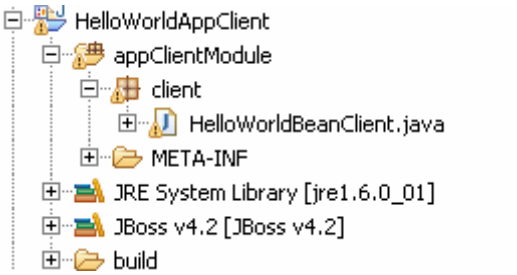
  `copy HelloWorld.jar <JBOSS_HOME>\server\deploy`

- ANT Script
  mit dem Skript Tool von Jacarta Ant™ (http://ant.apache.org) das Packen der Applikation und das Deployen der Komponenten durchführen:

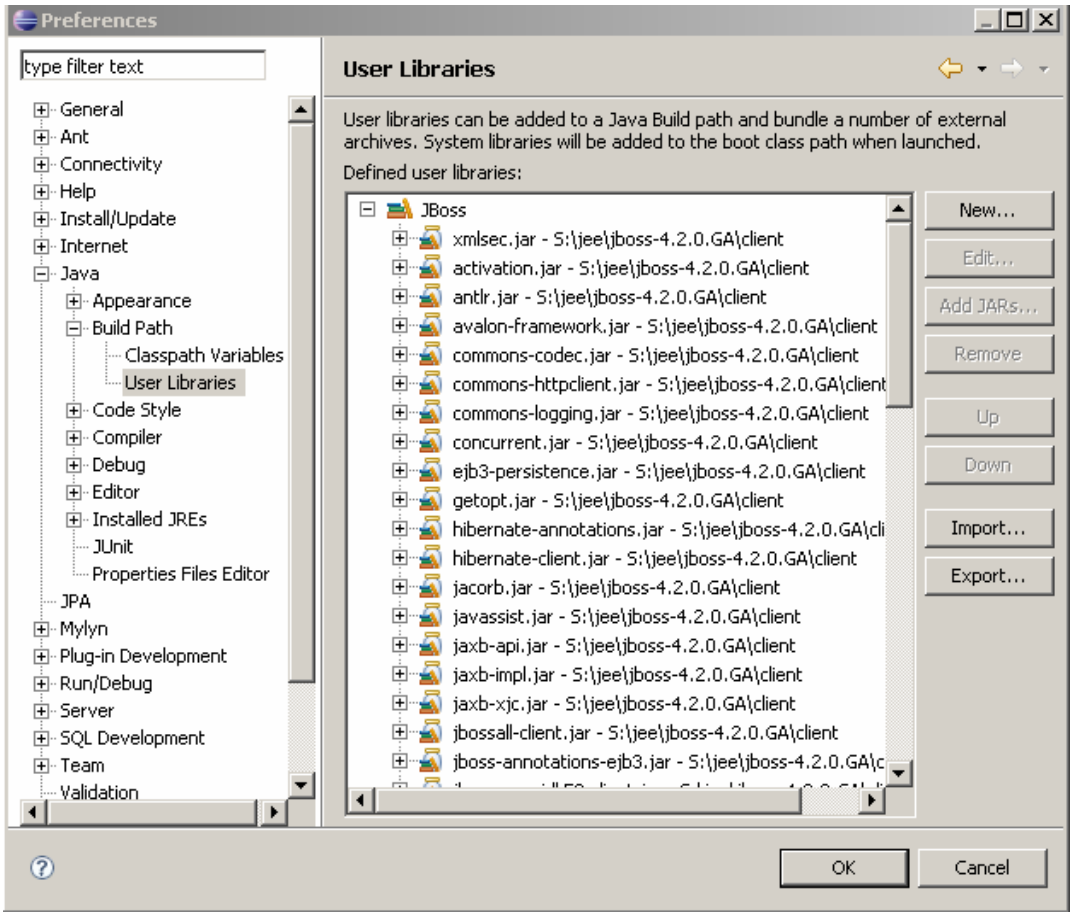
  z.B: **`ant deploy`**

4.7 Client Applikation ausführen

- als Stand-Alone ApplicationClient
- innerhalb der Entwicklungsumgebung



JBoss Client Library als User Library definiert:



Output:

```
CLASS > $Proxy0
implements > server.HelloWorld
implements > org.jboss.ejb3.JBossProxy
implements > javax.ejb.EJBObject
Using ThreadLocal: false
SocketClientInvoker[86c347, socket://127.0.0.1:3873] constructed
SocketClientInvoker[86c347, socket://127.0.0.1:3873] connecting
SocketClientInvoker[86c347, socket://127.0.0.1:3873] added new pool ([]) as
ServerAddress[127.0.0.1:3873, NO enableTcpNoDelay timeout 0 ms]
SocketClientInvoker[86c347, socket://127.0.0.1:3873] connected
reset timeout: 0
removed SocketClientInvoker[86c347, socket://127.0.0.1:3873] from registry
SocketClientInvoker[86c347, socket://127.0.0.1:3873] disconnecting ...
ClientSocketWrapper[Socket[addr=/127.0.0.1,port=3873,localport=2629].18dfef8]
closing
Server Echo : Hello EJB3 World
```

4.7.1 Clientaufruf bei EAR Deployment

- korrekter JNDI Lookup setzen
  - http://localhost:8080/jmx-console/HtmlAdaptor
  - Global JNDI Namespace

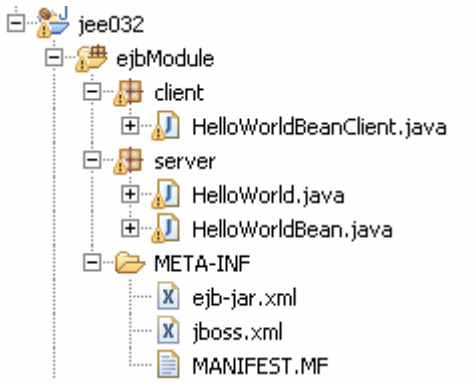
```
+- HelloWorld (class: org.jnp.interfaces.NamingContext)
|   +- HelloWorldBean (class: org.jnp.interfaces.NamingContext)
|   |   +- remote (proxy: $Proxy89 implements
                        interface server.HelloWorld,
                        interface org.jboss.ejb3.JBossProxy,
                        interface javax.ejb.EJBObject)
```

daher

```
Context ctx = new InitialContext();
Object o = ctx.lookup("HelloWorld/HelloWorldBean/remote");
```

4.8 XML Deployment Descriptor

- Alternative zu Annotation
  - Standard vor EJB 3.0 (JSR 220)
- überschreibt Annotationen
  - im Team eher komplex
  - Security durch Assembler



ejb-jar.xml - beschreibt Bean auf logischer Ebene

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldBean</ejb-name>
      <remote>server.HelloWorld</remote>
      <ejb-class>server.HelloWorldBean</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

jboss.xml – beschreibt Bean Applikationsserver spezifisch

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldBean</ejb-name>
      <jndi-name>ejb/hello</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

- Applikationsspezifischer JNDI

```
/**
 * @module jee.2007.Metzler
 * @exercise jee032.client.HelloWorldBeanClient.java
 */
public class HelloWorldBeanClient {
  public static void main(String[] args) {
    try {
      HelloWorld hello = null;
      Context ctx = new InitialContext();
      Object o = ctx.lookup("ejb/hello");
...
```

5 Inside EJB3

- Elemente der EJB-Architektur
- Szenarien und Rollen

5.1 Elemente und Aufgaben der EJB-Architektur

Die wichtigsten Elemente der EJB Architektur sind mit ihren wesentlichsten Aufgaben zusammengefasst:

5.1.1 Java EE Server

- Gewährleistung von :
  - Skalierbarkeit
  - Verfügbarkeit
  - Verbindungs-Management
  - Lastausgleich
  - Fehler-Management
  - Thread- und Prozessmanagement
  - Unterstützung von Clustering
  - Integration von Middleware Diensten
  - Verwalten von System Ressourcen

5.1.2 EJB Kontainer

- stellt den deployten Komponenten zur Verfügung:
  - Laufzeitumgebung
  - standardisierter Zugang zu den Services (JTA, JPA, JNDI, JMS, etc)

- Instanzen Verwaltung
  - life cycle management
  - pooling
  - Ausführen von Callback Methoden
  - Aufrufen von Interceptoren Klassen

- Remote Zugriff
  - Rechner- und Prozessübergreifender Komponentenaufruf
  - Remote Methode Invocation
  - IIOP (CORBA)

- Sicherheit
  - Authorisierung von Komponentenzugriffen
  - deklarativ
  - rollebasiert

- Persistenz
  - Java Persistence API

- Transaktionen
  - 2PC (Two Phase Commit)
  - distributet Transactions
  - CMT (Container Managed Transaction)
  - BMT (Bean Managed Transaction)

- Namens und Verzeichnisdienst
  - JNDI (Java Naming and Directory Interface)
  - Environment stellt Komponenten Information durch lokales Verzeichnis zur Verfügung

- Messaging
  - JMS-API (Java Message Service)
  - asynchroner Nachrichtenaustausch

5.1.3 Enterprise Beans

- Server Komponenten
- kapseln die Geschäftslogik
- Session Beans
  - Abbildung der Use Cases
  - beinhalten Logik
  - definieren Workflow
- Message Driven Beans
  - verarbeiten asynchron empfangene Nachrichten
- Entity Beans
  - definieren persistente Objekte der Java Persistence API

5.1.4 Deployment Descriptor

Der Deployment Descriptor ist eine XML (e**X**tensible **M**arkup **L**anguage) Datei und beschreibt die Deploymenteigenschaften. Der Deployment Descriptor beeinflusst verschiedene Entwicklungslevel (Rollen).

- Deployment Descriptor Beschreibungskriterien (Ausschnitt):
  - wie das Bean installiert werden muss
  - das Deployment (Installieren / Bereitstellen)
  - die EJB Implementation / die Business Interfaces
  - Verbindungen zu anderen Quellen (EJB, DB, ...)
  - die Umgebung des Beans (damit ist das Bean nicht an eine bestimmte Umgebung gebunden, lediglich die Konfiguration muss angepasst werden für eine andere Umgebung)
  - Transaktionsvariablen (Transactions Attributes)
  - Sicherheitsvariablen (Access Control Lists)

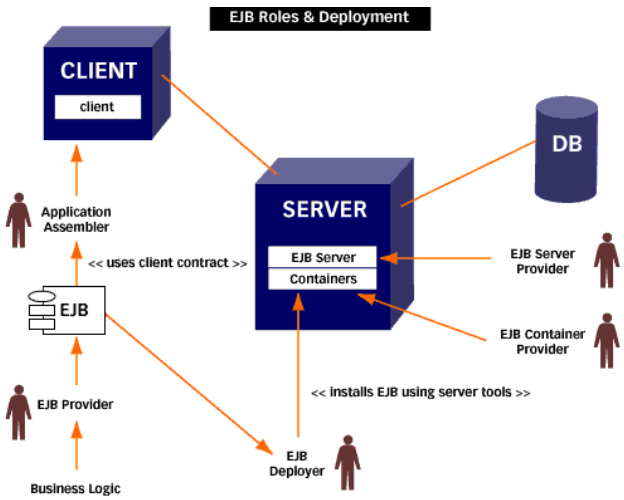
5.1.5 Clients

- thin
  - browser basiert
- fat
  - Java
- CORBA
- EJBs

5.2 Szenarien und Rollen

Durch die Trennung der verschiedenen Schichten ergibt sich neben der daraus resultierenden Robustheit noch die Möglichkeit des parallelen Entwicklungsprozesses. Dieser Aufgabenteilung sind die folgenden Rollen zugeordnet:

- EJB Provider
  - besitzt Wissen im Anwendungsbereich
  - entwickelt die Bean Komponenten
- Application Assembler
  - verbindet die EJB Komponenten zu einer konkreten Anwendung
- EJB Deployer
  - installiert die Applikation (Deployment Unit) in den EJB Container
- Server / Container Provider
  - Hersteller eines JEE Servers, z.B. Sun, BEA, IBM, Borland, JBoss, etc.)
  - Container Provider stellt die Laufzeitumgebung für EJBs zur Verfügung
- Systemadministrator
  - administriert Gesamtsystem, OS, JEE-Server, etc.



In der Praxis kann eine Person mehrere oder sogar alle Rollen übernehmen. Bei grösseren Projekten werden die Rollen jedoch häufig auf Teams aufgeteilt.

5.3 Zugriff auf ein EJB

- EJBs erlaubten den Remote Zugriff mit Hilfe von zwei Java-EE-Technologien
  - JNDI, Zugriff auf einen Namens- und Verzeichnisdienst
  - RMI-IIOP, Protokoll für den entfernten Methodenaufruf
- Funktionalität von EJBs ist über die öffentlichen Methoden ihrer Business Interfaces zugänglich
  - als Proxy Object (Business Interface Referenz)
  - Stub für Remote Client

- Der Client ermittelt zunächst per JNDI eine Referenz auf das Business Interface, welche im Verzeichnisdienst (JNDI) registriert ist.

```
// Connetion to JNDI Service
InitialContext ctx = new InitialContext(ProtocolParameter);

// Business Interface Referenz lookup
Object o = ctx.lookup("HelloWorld/HelloWorldBean/remote");
```

- diese Proxyreferenz wird durch ein entsprechendes Casting zu einer Ausprägung des Business Interface konvertiert

```
HelloWorld hello = null;

// Instanziierung des Business Interface durch ProxyObject
hello = (HelloWorld) o;
```

- als Alternative kann die erhaltene generische Referenz durch Aufruf der statischen Methode narrow der Klasse PortableRemoteObject typsicher in eine Ausprägung des Business Interface konvertiert werden.

```
// Typecasting due to RMI over IIOP
hello = (HelloWorld) PortableRemoteObject.narrow(o,HelloWorld.class);
```

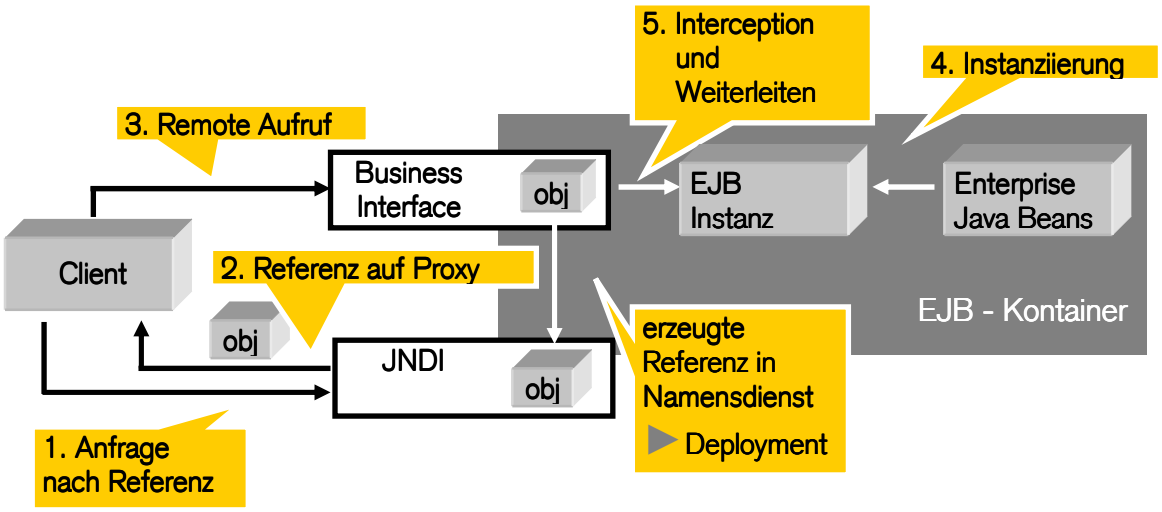
- oder Zugriff auf das Business Interface durch Refactoring des Proxyobjects

```
// Instanziierung des Business Interface durch NamingContext
hello = (HelloWorld) ctx.lookup(o.getClass().getName());
```

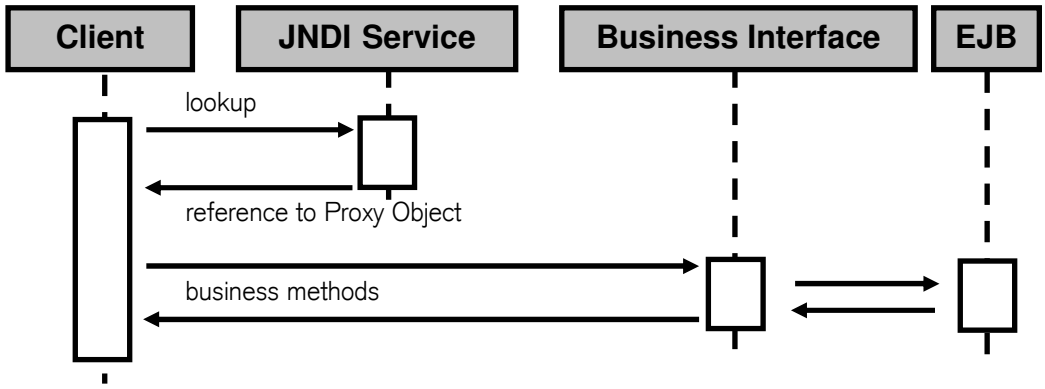
- Zugriff zum EJB über das Proxy
  - dabei wird serverseitig zur Kommunikation mit der EJB ein Proxy Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt. Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

```
// Zugriff auf serverseitige Business Logik
hello.echo("Hello EJB3 World");
```

- Business Methoden lassen sich nur über die erzeugte Objektinstanz aufrufen
- Container fängt die Aufrufe an das Business Object auf und leitet diese weiter
- Interception (Abfangen, Unterbrechen)
  - kontrollieren der Aufrufe
  - Vor- und Nachbearbeitung der Aufrufe
  - Sicherheit, Transaktionen, Persistenz



5.3.1 Sequenz Diagramm



5.3.2 RMI (Remote Method Invocation)

- Remote Method Invocation (Aufruf entfernter Methoden)
- ist Aufruf einer Methode eines entfernten Java-Objekts
- realisiert die Java-eigene Art eines sog. RPC (Remote Procedure Call)
- "Entfernt" bedeutet Objekt ist in einer anderen VM (virtuellen Maschine)
  - auf einem entfernten Rechner
  - auf lokalen Rechner
- Aufruf für das aufrufende Objekt ist wie ein lokaler Aufruf
- besondere Ausnahmen abfangen
  - wie Verbindungsabbruch
  - Server nicht erreichbar

- auf Client-Seite kümmert sich der sogenannte Stub um den Netzwerktransport
  - eine mit dem RMI-Compiler rmic erzeugte Klasse
  - Stub muss entweder lokal oder über das Netz für den Client verfügbar sein

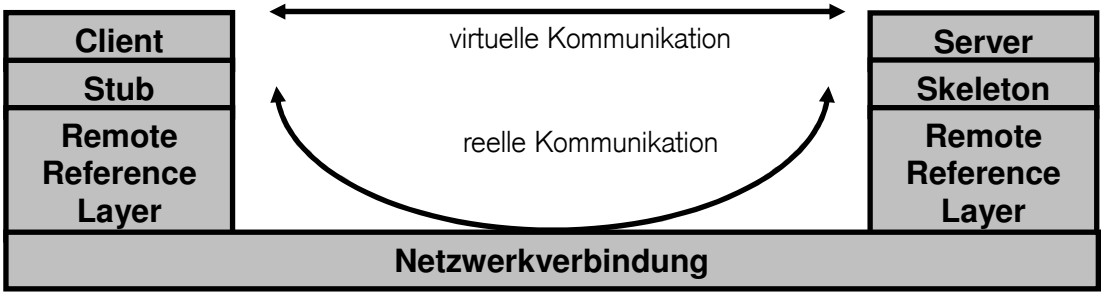
oder

- entfernte Objekte werden durch entferntes Objekt bereitgestellt
  - erste Verbindungsaufnahme benötigt Adresse des Servers und Bezeichners (RMI-URL)
  - Namensdienst auf dem Server liefert eine Referenz auf das entfernte Objekt
  - das entfernte Objekt im Server muss sich unter diesem Namen zuvor beim Namensdienst registriert haben
  - RMI-Namensdienst wird über statische Methoden der Klasse java.rmi.Naming angesprochen
  - Namensdienst ist als eigenständiges Programm implementiert und wird RMI Registry genannt

Kommunikationsprotokoll

- RMI bezeichnet Kommunikationsprotokoll für entfernte Aufrufe zwischen Java-Objekten
- RMI bezeichnet eine Java-Standard-Klassenbibliothek als Teil der JSE
- für RMI sind zwei Ports reserviert
  - Port 1099 ist für die RMI-Registry reserviert, also den Namensdienst
  - Port 1098 für den Activator reserviert
- IIOP (Internet Inter ORB Protokoll) als alternatives Kommunikationsprotokoll
- viele JEE Server Implementation arbeiten mit RMI over IIOP

5.3.3 RMI Kommunikationsmodell



Ablauf

Der Server registriert ein Remote Object bei der RMI-Registry unter einem eindeutigen Namen.

Der Client schaut bei der RMI-Registry unter diesem Namen nach und bekommt eine Objektreferenz, die seinem Remote Interface entsprechen muss.

Der Client ruft eine Methode aus der Objektreferenz auf (dass diese Methode existiert, wird durch das Remote Interface garantiert).

Der Server gibt dem Client die Rückgabewerte dieses Aufrufes, oder der Client bekommt eine Fehlermeldung (z. B. bei einem Verbindungsabbruch).

Activation

Als Ergänzung von RMI steht die sogenannte Activation ("Aktivierung") zur Verfügung. Ähnlich der RMI-Registry ist auch dies eine zentrale Anlaufstelle für RMI-Clients. Jedoch kann der Activator auch RMI-Server-Objekte erzeugen und neue Virtuelle Maschinen starten.

Anwendung

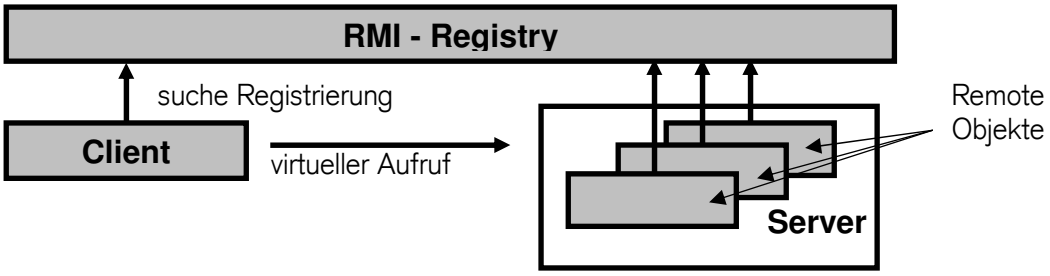
Bei der Kommunikation mit RMI hat der Client den Eindruck, Methoden von Objekten aufzurufen, die auf dem Server liegen. Folgende Kriterien sind Bestandteile von RMI:

- Nahtlose Unterstützung entfernter Aufrufe von Objekten in verschiedenen VM
- Callback-Unterstützung von Server zu Applet
- Integrierung des verteilten Objekt-Modells in Java unter Beibehaltung der Java-Semantik
- Unterschiede zwischen dem verteilten Objekt Modell und dem lokalen Java Objekt Modell sollen transparent gemacht werden
- Schreiben zuverlässiger Anwendungen so einfach wie möglich machen
- Bewahrung der Sicherheitsaspekte der JRE

Jeder Client benötigt zur Benutzung von Serverobjekten, die auf dem Server existieren, eine Beschreibung über die Fähigkeiten eines entfernten Objektes.

- jedes Interface implementiert für entfernte Objekte, direkt oder indirekt, das Interface java.rmi.remote
- dadurch wird das Objekt als ein "Remote-Objekt" gekennzeichnet
- Klassen, die das Interface implementieren, erlauben Clients deren Methoden entfernt (remote) aufzurufen

- Client erhält vom Broker-Tool (RMI-Registry) lediglich ein Stub-Objekt
  - ein Stub ist eine Klasse, die das Remote-Interface implementiert und daher für den Client als Platzhalter für den Zugriff auf das Remote-Objekt dient.
- Remote-Objekt bleibt auf dem Server



- der Stub kommuniziert über eine TCP-Verbindung mit dem als Skeleton bezeichneten Gegenstück auf der Server-Seite
  - das Skeleton kennt das tatsächliche Applikationsobjekt
  - leitet die Anfragen des Stubs an dieses weiter
  - gibt den Rückgabewert an ihn zurück

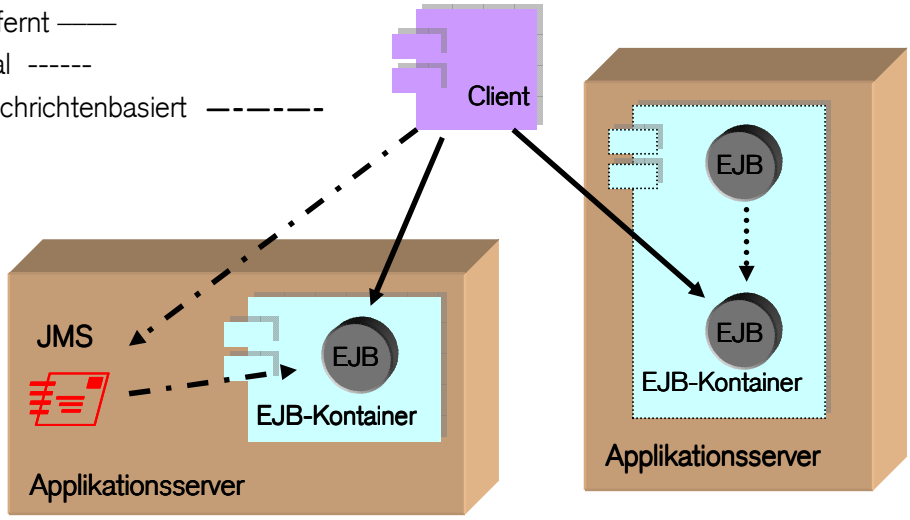
- Stub und Skeleton werden während der Entwicklung mit Hilfe eines Tools generiert und verbergen die komplizierten Details der Kommunikation zwischen Server und Client.

RMI verfolgt mit der Verteilung von Objekten im Netz einen ähnlichen Ansatz wie CORBA (Common Object Request Broker Architecture, siehe beispielsweise http://www.omg.org).

5.4 EJB Kommunikationsmodelle

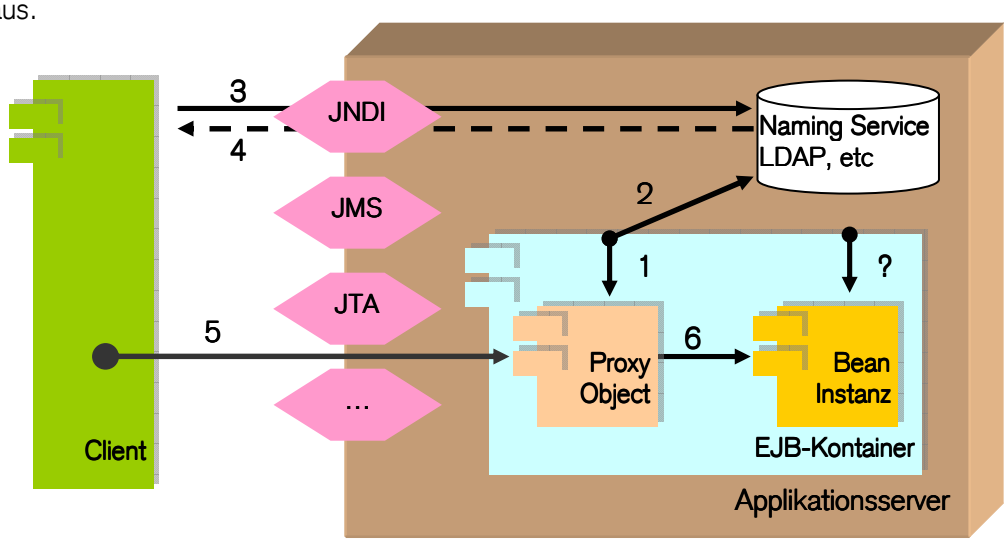
An Prozesse beteiligte EJBs kommunizieren:

- synchron, entfernt ——
- synchron, lokal ------
- asynchron, nachrichtenbasiert —–—–—



5.4.1 synchrone, entfernte Kommunikation

- Aufrufe erfolgen als synchrone Methodenaufrufe
- mittels RMI-IIOP in einem verteilten System
- Übertragung der Daten erfolgt mittels "by-Value" Semantik
  - als Kopie der Originaldaten
  - Änderungen der Daten wirken sich somit nur auf die Originaldaten und nicht auf die Kopie aus.



Ablauf:

EJB-Container erzeugt Proxy Object des Business Interface

EJB-Container registriert Proxy Object beim Namensdienst

Client fordert vom Namensdienst Referenz auf Proxy Object

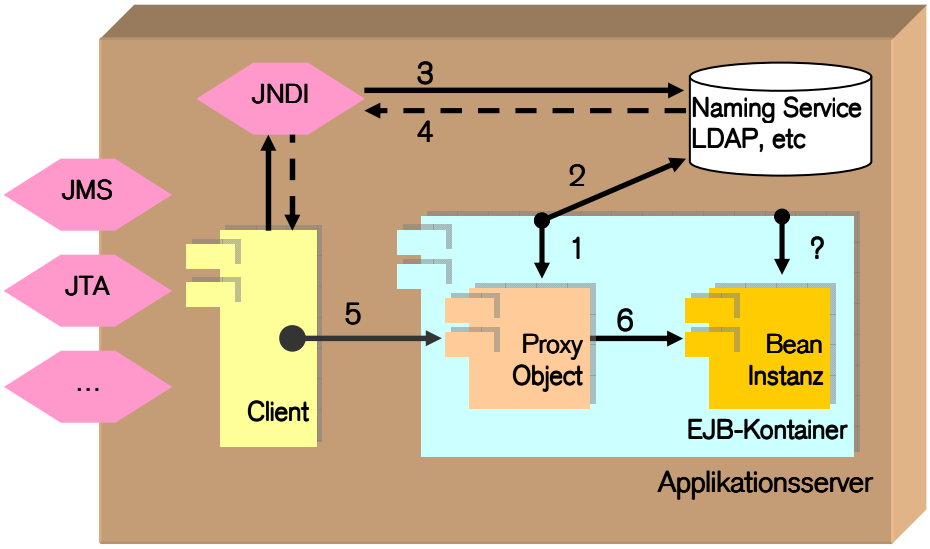
Namensdienst liefert Referenz auf das Proxy Object

Client ruft über das Proxy Object die Geschäftsmethoden der EJB auf

Proxy Object delegiert Methodenaufruf an die EJB-Instanz

5.4.2 synchrone, lokale Kommunikation

- Aufrufe von Diensten einer EJB-Komponente erfolgen durch synchrone Methodenaufrufe
- die Übertragung der Daten erfolgt mittels "by-Referenz" Semantik
  - keine Kopien von Parametern wird an die empfangende Komponente übergeben
  - Adresszeiger auf die Originaldaten im Speicher
  - Ändern sich die übergebenen Daten, so werden die Originaldaten überschrieben



Ablauf:

EJB-Container erzeugt Proxy Object des Business Interface

EJB-Container registriert Proxy Object beim Namensdienst

Client fordert vom Namensdienst Referenz auf Proxy Object

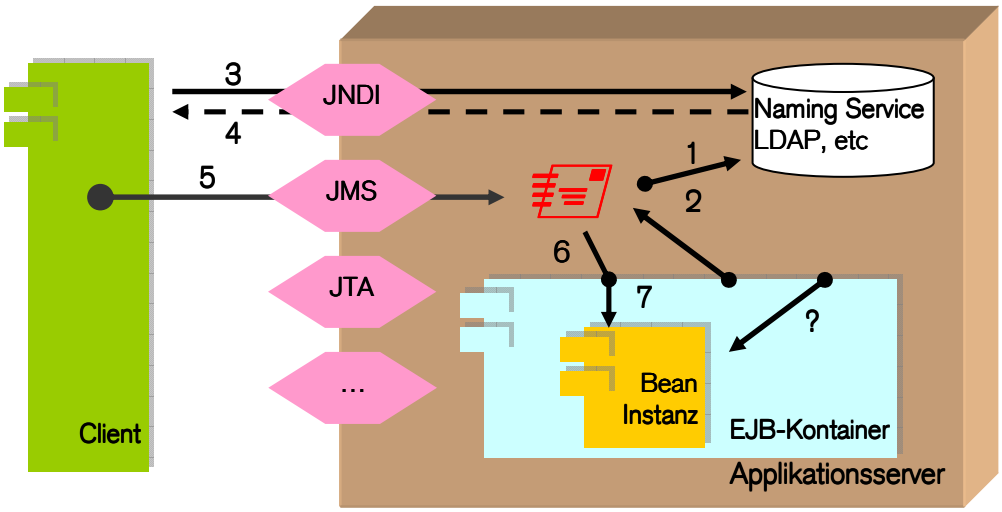
Namensdienst liefert Referenz auf das Proxy Object

Client ruft über das Proxy Object die Geschäftsmethoden der EJB auf

Proxy Object delegiert Methodenaufruf an die EJB-Instanz

5.4.3 asynchron, nachrichtenbasierte Kommunikation

- Schnittstelle zwischen objektorientierten und nicht-objektorientierten Prozessen in verteilten Systemen



Ablauf:

Applikationsserver registriert Nachrichtenempfänger beim Namensdienst

EJB-Container registriert sich als Listener für Nachrichten

Client fordert vom Namensdienst Referenz des Nachrichtenempfängers

Namensdienst liefert Referenz auf Nachrichtenempfänger

Client sendet Nachrichten an Nachrichtenempfänger

Nachrichtenempfänger übermittelt Listener eingegangene Nachricht

Listener delegiert Nachricht an Bean-Instanz

5.4.4 Überblick über die Kommunikationsmodelle

Die folgende Tabelle weist die Aufrufmodelle den verschiedenen EJB-Typen zu:

EJB Typen	Kommunikationsmodell		
	synchron entfernt	synchron lokal	asynchrony nachrichtenbasiert
Stateless Session Bean	✓	✓	
Statefull Session Bean	✓	✓	
Message Driven Bean			✓

6 SLSB Stateless Session Bean (zustandslos)

- modelliert einfacher Geschäftsprozess
  - z.B. Zinsberechnung
- kann keine Daten speichern
- haben keine Identität
- Aufgabe muss mit einem Methodenaufruf erledigt sein, zweimaliger Aufruf des Beans garantiert nicht dieselbe Instanz auf dem Server
- sind oft als Fassaden eingesetzt
  - für Aufrufe an weitere Bean Instanzen
  - Persistenz Kontext
- pooling:
  - mehrere Clients rufen dasselbe Bean auf
  - gemeinsame Nutzung von Sourcen
  - weniger Objekte im Einsatz
  - ABER: Bean ist single-threaded, ein Bean kann nur einen Client behandeln

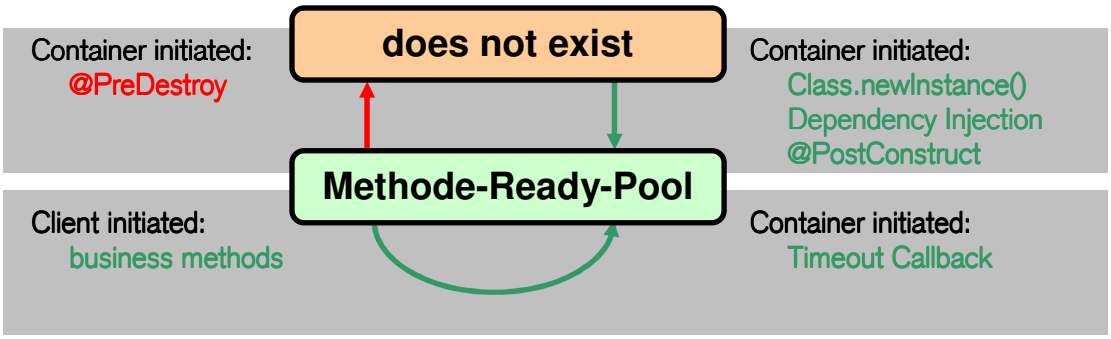
- Kontainer erzeugt während Deployment Beaninstanzen im Pool
  - konfigurierbar
  - "create" Business Methode (@ProConstruct) steht daher nicht immer in direkter Kopplung zur aufrufenden Clientapplikation

Annotation @Stateless			
Parameter	**Beschreibung**	**Typ**	**Default**
name	expliziter Name der Session Bean	String	unqualifizierter Klassenname
mappedName	mappen des Namens auf einen weiteren implementierungsspezifischen (JNDI)-Namen (nicht im Standard-Umfang der Spezifikation)	String	
description	eine Beschreibung für das Bean	String	

Annotation @Remote			
Parameter	**Beschreibung**	**Typ**	**Default**
value	alle Schnittstellen, die entfernt zugreiffbar sein sollen	Class[]	

Annotation @Local			
Parameter	**Beschreibung**	**Typ**	**Default**
value	alle Schnittstellen, die lokal zugreiffbar sein sollen	Class[]	

6.1 Bean Life Cycle



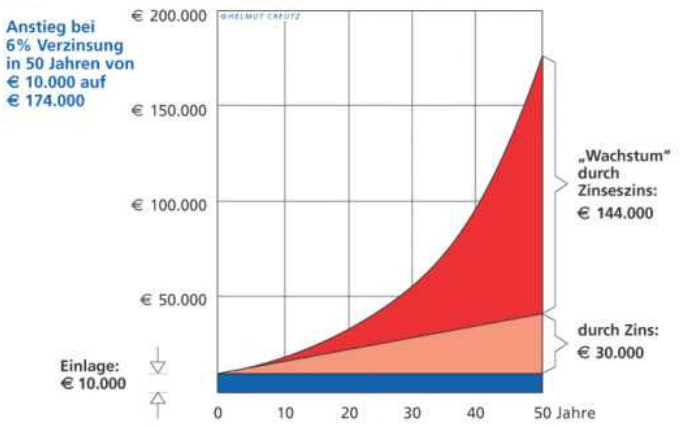
LifeCycleMethoden	
Annotation	Beschreibung
@PostConstruct	Wird vom Container aufgerufen nachdem die Bean durch den Container instanziiert worden ist. Für den Client ist der Aufruf dieser Methode transparent.
@PreDestroy	Ressourcen werden durch den Bean Provider freigegeben. Für den Client ist der Aufruf dieser Methode transparent.
@Remove	Wird vom Container aufgerufen, nachdem der Client eine Methode des Remote-Interface aufgerufen hat, die durch die Bean mit der Annotation @Remove gekennzeichnet ist. In dieser Methode sollten die für die Bean reservierten Ressourcen wieder freigegeben werden. Danach löscht der Container die Bean-Instanz. ACHTUNG: bei SLSB eine NO-OP
Timeout Callback	Bezieht sich auf eine Container Timer Funktion.

6.2 Übung : Zinsberechnung

Erstellen und deployen Sie ein Stateless Session Bean, das als Businessmethode die Zinsberechnung eines Geldbetrages **`amount`** über eine definierte Laufzeit **`years`** zu einem gegebenen Zinssatz **`rate`** mehreren Klienten zur Verfügung stellt. Die Attribute **`amount`** und **`years`** müssen clientspezifisch definiert werden. Der Zinssatz **`rate`** kann als ein Attribut der EJB oder clientspezifisch definiert werden.

Ansatz:

Entwicklung einer Geldanlage durch
Zins und Zinseszins:



```java
/**
 * @module    jee.2007.Metzler
 * @exercise jee061.server.ZinsBean.java
 */
@Stateless
public class ZinsBean implements Zins {
  @Override
  public double calculate(double amount, double rate, int years) {
    return amount
        * Math.pow((rate / 100.0) + 1.0, (double) years);
  }
}
```

```java
/**
 * @module    jee.2007.Metzler
 * @exercise jee061.server.Zins.java
 */
@Remote
public interface Zins {
  double calculate(double amount, double rate, int years);
}
```

Eigenschaften

- parameterloser Konstruktor ist erforderlich
  - EJB Container initialisiert Bean über Class.newInstance()
  - wird automatisch erzeugt

- Interfacebezug über @Remote / @Local in der Beanklasse
  - optionaler Name als Parameter

Zugriff auf das Environment

```java
/**
 * @module jee.2007.Metzler
 * @exercise jee062.server.ZinsBean.java
 */
@Stateless(name = "Rate")
@Remote(ZinsRemote.class)
@Local(ZinsLocal.class)
public class ZinsBean {
  private float rate = 0.0f;
  @PostConstruct
  public void injectRate() {
    InitialContext ic;
    try {
      ic = new InitialContext();
```

```
      rate = ((Float) ic.lookup("java:comp/env/Rate"))
          .floatValue();
      System.out.println("Rate set:\t" + rate);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public float calculate(float amount, int years) {
    return amount
        * (float) Math.pow((double) ((rate / 100.0f) + 1.0f),
            (double) years);
  }
  public float getRate() {
    return rate;
  }
}
```

auf Deploymentdescriptor: ejb-jar.xml

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/ejb-
jar_3_0.xsd"
  version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>Rate</ejb-name>
      <env-entry>
        <description>Zinssatz</description>
        <env-entry-name>Rate</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>5.2</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

oder Argument direkt annotiert:

```
/**
 * @module jee.2007.Metzler
 * @exercise jee063.server.ZinsBean.java
 */
@Stateless(name = "Rate")
@Remote(ZinsRemote.class)
@Local(ZinsLocal.class)
public class ZinsBean {
  @Resource(name = "Rate")
  private float rate;
```

6.2.1 EJB Kontext

- Zugriff auf EJBContext
  - Security (z.B. getCallerIdentity)
  - Transaction (z.B. setRollbackOnly)

Methoden des EJB-Kontext (Ausschnitt)	
Methode	**Beschreibung**

lookup()	Shortcut für Objekte des Enterprise Naming Context (ENC)
	- im JNDI ist der ENC unter "java:/comp/env" zu finden
	Einträge des ENC können mit @Resource injiziert werden
	ENC:
	`Object o = ctx.lookup("ejb/myBean");`
	JNDI:
	`Object o = ic.lookup("java:comp/env/ejb/myBean");`

getBusinessObject()	- nur SessionContext - liefert robuste Referenz auf das EJB
getInvoked-BusinessInterface()	- nur SessionContext - liefert das Interface der Geschäftsschnittstelle Local, Remote oder WebService

- Zugriff auf SessionContext (extends EJBContext)
  - Referenz auf aktuelle Bean Implementation (z.B. BusinessObject)

```
/**
 * @module jee.2007.Metzler
 * @exercise jee064.server.ZinsBean.java
 */
@Stateless(name = "Rate")
@Remote(ZinsRemote.class)
@Local(ZinsLocal.class)
public class ZinsBean {
  @Resource
  private SessionContext sc;
  @Resource(name = "Rate")
  private float          rate;
  public float calculate(float amount, int years) {
    return amount
        * (float) Math.pow((double) ((rate / 100.0f) + 1.0f),
            (double) years);
  }
  public float getRate() {
    return rate;
  }
  public Object getRemoteReference() {
    return sc.getBusinessObject(ZinsRemote.class);
  }
  public Object getRemoteInterface(){
    return sc.getInvokedBusinessInterface();
```

```
  }
}
```

oder mit @Ressource Annotation auf Methodenebene

```
/**
 * @module jee.2007.Metzler
 * @exercise jee065.server.ZinsBean.java
 */

...
@Resource
public void setSessionContext(SessionContext sc) {
  this.sc = sc;
}
...
```

Output

```
Proxy:      jboss.j2ee:jar=jee064.jar,name=Rate,service=EJB3
Interface:  interface server.ZinsRemote
```

7 SFSB Stateful Session Bean (zustandsbehaftet)

- modelliert Geschäftsprozess
  (z.B. Shopping Cart)
- kann Daten speichern
- Zustandsbehaftet
  (kann den Zustand eines best. Client speichern)
- mehrmaliger Aufruf des Beans für einen Geschäftsablauf möglich
- mehrere create() Methoden
  (mit unterschiedlichen Signaturen)
- kein pooling

- passivated:
  - wenig Ressourcen und Memory beansprucht
  - dadurch bessere Performance

Annotation @Stateful			
Parameter	**Beschreibung**	**Typ**	**Default**
name	expliziter Name der Session Bean	String	unqualifizierter Klassenname
mappedName	mappen des Namen auf einen weiteren implementierungsspezifischen (JNDI)-Namen (nicht im Standard-Umfang der Spezifikation)	String	
description	eine Beschreibung für das Bean	String	

Annotation @Remove			
Parameter	**Beschreibung**	**Typ**	**Default**
retainIfException	Löschverhalten der Bean bei einer Exception	Boolean	true

7.1 Kompatibilität mit EJB2.1

- EJB 2.1 (und frühere Versionen) definieren Home- und LocalHome-Interface
- für Zustandsverwaltung der Bean

- Kompatibilität durch Annotationen
  - Definition der Remote Home Schnittstelle

Annotation @RemoteHome			
Parameter	**Beschreibung**	**Typ**	**Default**
value	Remote-Home-Interface	Class[]	

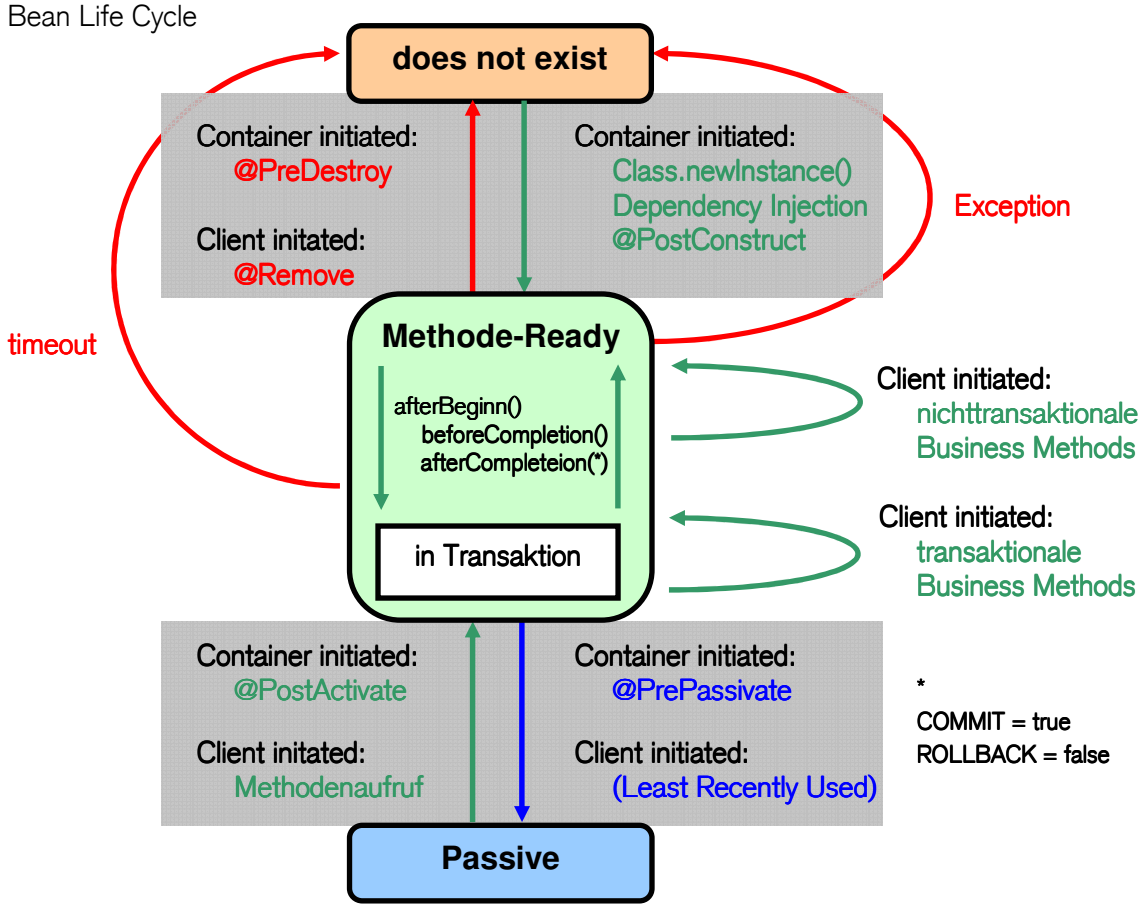
- Definition der Local Home Schnittstelle

Annotation @LocalHome			
Parameter	**Beschreibung**	**Typ**	**Default**
value	Local-Home-Interface	String[]	

- Erzeugung einer SFSB durch einen EJB2.1 Client

Annotation @Init			
Parameter	**Beschreibung**	**Typ**	**Default**
value	korrespondierende Create Methode des Home Interface	String	

7.2 Bean Life Cycle



Bean Life Cycle Methoden

LifeCycleMethoden	
Annotation	Beschreibung
@PostConstruct	Wird vom Container aufgerufen nachdem die Bean durch den Container instanziiert worden ist. Für den Client ist der Aufruf dieser Methode transparent.
@PreDestroy	Ressourcen werden durch den Bean Provider freigegeben. Für den Client ist der Aufruf dieser Methode transparent.
@PrePassivate	Wird durch den Container aufgerufen, bevor die Bean in den passiven Zustand verschoben wird. Ressourcen werden durch den Bean Provider freigegeben. Für den Client ist der Aufruf dieser Methode transparent.
@PostActivate	Der Container holt die passivierte Instanz der Bean in den Arbeitsspeicher zurück und ruft danach die mit @PostActivate markierte parameterlose Methode auf, um Zustand der Bean erneut zu initialisieren.
@Remove	Wird vom Container aufgerufen, nachdem der Client eine Methode des Remote-Interface aufgerufen hat, die durch die Bean mit der Annotation @Remove gekennzeichnet ist. In dieser Methode sollten die für die Bean reservierten Ressourcen wieder freigegeben werden. Danach löscht der Container die Bean-Instanz.
Timeout	Passivierte, verwaiste Beaninstanzen werden vom Container gelöscht. SINNVOLL: saubere Ressourcenverwaltung
Exception	Runtime Exceptions sind als unstabilen Bean Zustand interpretiert. Der Container gibt die Beaninstanz frei. Die laufende Transaktion wird mit einem Rollback beendet. @AnnotationException macht aus einer Runtime Exception eine normale Exception. Der Parameter rollback=true bestimmt das Transaktionsverhalten. Normale Exceptions werden an den Client weitergegeben und hat keinen Einfluss auf die Beaninstanz.

7.3 Übung: Kontoführung

Erstellen und deployen Sie ein Stateful Session Bean zur Kontoführung. Benützen Sie dazu die Variable **`private float accountBalance`**:

- Betrag deponieren (als Float)
- Betrag abheben (als Float)
- Saldo ausgeben (auf Konsole)

7.3.1 Environment Entries

Erweitern Sie die Business Logik und führen Sie auf allen Konten einen Zinssatz. Dabei ist es sinnvoll für alle Konten denselben Zinssatz zu definieren. Benützen Sie dazu die Variable **`private float rate:`**

```
@Resource(name = "RateValue")
private float     rate            = 0.0f;
```

Zins (im Deployment Descriptor) als Environment Entry festlegen

```
<ejb-name>KontoBean</ejb-name>
<env-entry>
   <description>Zinssatz</description>
   <env-entry-name>RateValue</env-entry-name>
   <env-entry-type>java.lang.Float</env-entry-type>
   <env-entry-value>5.25</env-entry-value>
</env-entry>
```

7.3.2 Zinsberechnung über lokales Interface der ZinsBean

@EJB injektioniert das ZinsBean

```
/**
 * @module jee.2007.Metzler
 * @exercise jee071.server.KontoBean.java
 */
@Stateful
public class KontoBean implements Konto {
  @EJB(beanName = "Rate",
       beanInterface = server.ZinsLocal.class,
       name = "Rate")
  private ZinsLocal zins;
```

Annotation @EJB			
Parameter	Beschreibung	Typ	Default
beanInterface	Interface-Typ der referenzierten Bean Remote oder Local	Class	Object.class
beanName	logischer Name der referenzierten Bean name Attribut der @Stateless, @Stateful, @MessageDriven	String	
mappedName	Name der referenzierten Bean im globalen Namensdienst (nicht im Standard-Umfang der Spezifikation)	String	
name	Name, unter dem die referenzierte Bean im lokalen Namensdienst (java:comp/env) der Komponente, die die Annotation verwendet, auffindbar sein soll	String	

Annotation @EJBs			
Parameter	Beschreibung	Typ	Default
value	Liste der EJB Annotationen	EJB[]	

Alternative über einen JNDI lookup

```
/**
 * @module jee.2007.Metzler
 * @exercise jee072.server.KontoBean.java
 */
@Stateful
public class KontoBean implements Konto {
  private float accountBalance = 0.0f;

  @Override
  public void calculateRate(int years) {
    InitialContext ic;
    ZinsLocal zins = null;
    try {
      ic = new InitialContext();
      // auch: ic.lookup(ZinsLocal.class.getName());
      zins = (ZinsLocal) ic.lookup("ZinsBean/local");
    } catch (Exception e) {
      e.printStackTrace();
    }
    accountBalance += zins.calculate(accountBalance, rate, years);
  }
```

7.3.3 Speichern des Kontosaldos in einer (mySQL) Datenbank Tabelle Account

```
DROP TABLE IF EXISTS ACCOUNT;

CREATE TABLE ACCOUNT(
   ID INTEGER PRIMARY KEY,
   DESCRIPTION VARCHAR(20),
   BALANCE FLOAT(5,2)
   );

INSERT INTO ACCOUNT VALUES(1, 'Savings', 100.50);
```

Datenbankanbindung verwalten durch Lifecycle Methoden

```
/**
 * @module jee.2007.Metzler
 * @exercise jee073.server.KontoBean.java
 */
@Stateful
@Remote(Konto.class)
public class KontoBean implements Konto {
  private Connection co          = null;
  private Statement  st          = null;
  private ResultSet  rs          = null;
  final String       userName    = "root";
  final String       password    = "";
  final String       databaseUrl = "jdbc:mysql://localhost:3306/jee";
  @Remove
  @Override
  public void closeAccount() {
    try {
      st = co.createStatement();
      st.executeUpdate("UPDATE account SET balance = "
          + accountBalance + " WHERE ID = 1");
    } catch (SQLException e) {
      System.err.println("Could not update statement '"
          + e.getMessage());
    }
  }
  @PostConstruct
  @PostActivate
  private void getDataSource() {
    try {
      Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException cnfe) {
      System.err.println("Driver not found: " +
       cnfe.getMessage());
    }
    try {
      co = DriverManager.getConnection(databaseUrl, userName,
          password);
    } catch (SQLException sqle) {
```

```
      System.err.println("Could not get connection to db:"
          + sqle.getMessage());
    }
    try {
      st = co.createStatement();
      rs = st.executeQuery
              ("SELECT balance FROM account WHERE ID = 1");
      if (rs.next())
        accountBalance = rs.getFloat("BALANCE");
    } catch (SQLException e) {
      System.err.println("Could not execute statement '"
          + e.getMessage());
    }
  }
  @PreDestroy
  @PrePassivate
  private void closeDataSource() {
    try {
      st.close();
      co.close();
    } catch (SQLException e) {
      System.err.println("Could not get connection to db:"
          + e.getMessage());
    }
  }
}
```

7.3.4 Anbindung einer DataSource

Injektion über @Resource

Annotation @Resource			
Parameter	Beschreibung	Typ	Default
authentication-Type	definiert wer die Anmeldung beim Ressourcen-Manager übernimmt: CONTAINER: die Anmeldedaten werden im Deskriptor hinterlegt BEAN: die Anmeldedaten werden explizit bei der Erzeugung der Verbindung übergeben	AuthenticationTyp	CONTAINER
description	Beschreibung der referenzierten Ressource	String	
mappedName	Name der referenzierten Ressource im globalen Namensdienst (nicht im Standard-Umfang der Spezifikation)	String	

Annotation @Resource			
Parameter	Beschreibung	Typ	Default
name	Name, unter dem die referenzierte Ressource im lokalen Namensdienst (java:comp/env) der Komponente, die die Annotation verwendet, auffindbar sein soll	String	
shareable	definiert ob die Ressource von mehreren Komponenten genutzt werden kann	Boolean	true

Annotation @Resources			
Parameter	Beschreibung	Typ	Default
value	Liste der einzelnen Ressource-Annotationen	Resource[]	

JBoss spezifische DataSource: <JBoss_Home>/server/jee/deploy/mysql-ds.xml

```
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/account</jndi-name>
    <connection-url>
       jdbc:mysql://localhost:3306/jee
     </connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password></password>
   <local-tx-datasource>
<datasources>
```

Injektion der DataSource

```
/**
 * @module jee.2007.Metzler
 * @exercise jee074.server.KontoBean.java
 */
@Stateful
@Remote(Konto.class)
public class KontoBean implements Konto {
@Resource(mappedName = "java:/jdbc/account")
private DataSource ds             = null;
```

7.3.5 lokales Speichern einer serverseitigen Bean Referenz

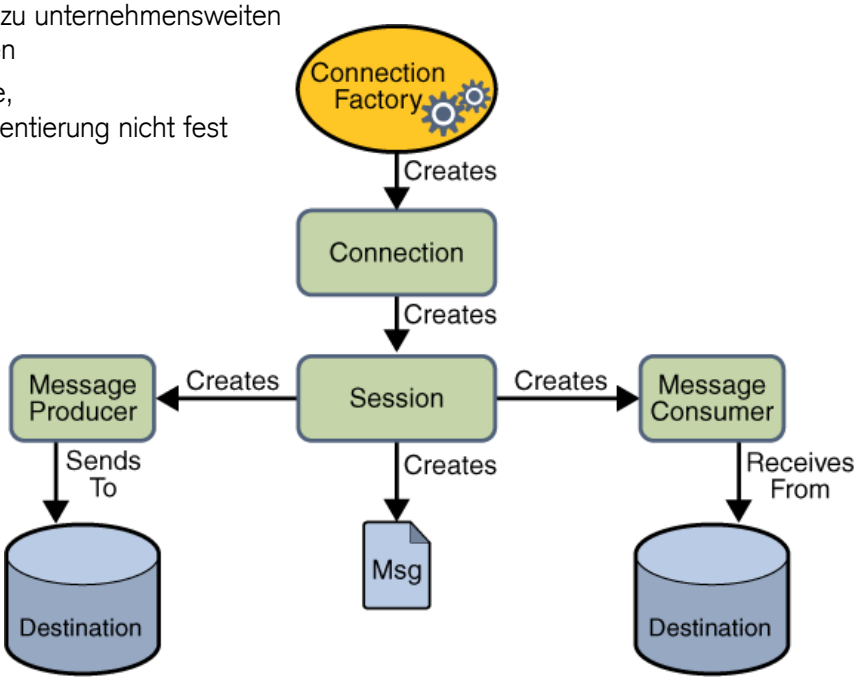
- Handle definiert robuste Referenz auf Objekte
  - definiert Wiederverwendbarkeit
  - Remote Referenz auf EJB3 ist robust
  - POJO muss serialisierbar sein

```
/**
 * @module jee.2007.Metzler
 * @exercise jee074.client.KontoClient_SaveReference.java
 */
public class KontoClient_SaveReference {
  public static void main(String[] args) {
    saveFile();
    System.out.printf("\nSaldo:\t%+9.2f CHF", loadFile()
        .getAccountBalance());
  }
  private static void saveFile() {
    try {
      InitialContext ctx = new InitialContext();
      Konto konto = (Konto) ctx.lookup("KontoBean/remote");
      // write Object Reference to serialized File
      FileOutputStream f = new FileOutputStream(
          "Account_Reference.ser");
      ObjectOutputStream o = new ObjectOutputStream(f);
      o.writeObject(konto);
      o.flush();
      o.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  private static Konto loadFile() {
    Konto anotherKonto = null;
    try {
      // read Object Reference from File
      FileInputStream fi = new FileInputStream(
          "Account_Reference.ser");
      ObjectInputStream oi = new ObjectInputStream(fi);
      // read the object from the file and cast it to a Konto
      anotherKonto = (Konto) oi.readObject();
      oi.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    return anotherKonto;
  }
}
```

8 Message Driven Beans

8.1 JMS

- API für den Zugang zu unternehmensweiten Messaging-Systemen
- definiert Schnittstelle, legt aber die Implementierung nicht fest



Der Vorteil für Software-Hersteller besteht darin, dass sie ihre existierenden Messaging-Produkte mit JMS-Schnittstellen versehen können, ohne sie wesentlich ändern zu müssen.

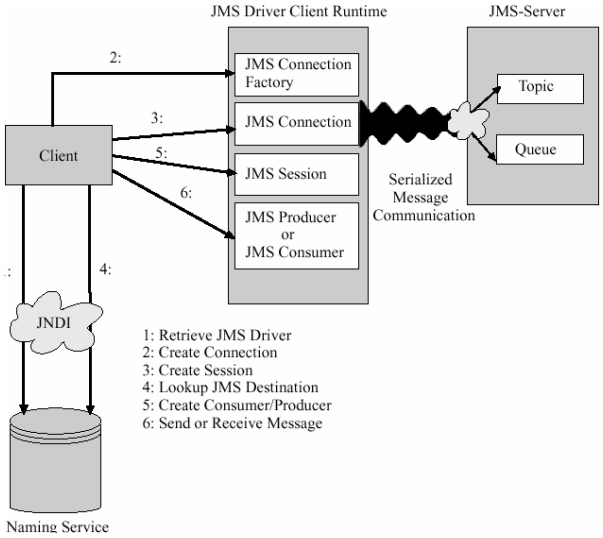
Integration von EJB mit JMS durch Message Driven Bean

- Client greifft per JMS auf MDB zu
- Client findet MDB indirekt über die Auswahl einer Message-Queue
  - Point-to-Point-Verbindung
- oder eines Message-Topics
  - Publish/Subscribe-Mechanismus
- mittels JNDI (Java Naming and Directory Service)
- Message-Driven-Bean ermöglicht nun asynchronen Aufruf von EJBeans
- realisiert nebenläufige Nachrichten
- EJBeans lassen sich nun wie jeder andere JMS Message Listener ansprechen

Nachrichten erzeugen mit der JMS API

Das Erzeugen und Absenden einer Nachricht über JMS erfolgt in sechs Schritten.

1. Treiber lokalisieren
2. Verbindung erzeugen
3. Session erzeugen
4. Destination lokalisieren
5. Consumers / Producers erzeugen
6. Nachricht senden / empfangen



Lokalisieren des JMS-Treibers:

- Treiber für das verwendete JMS-Produkt laden
- lookup(CONNECTION_FACTORY)
  - CONNECTION_FACTORY ist JNDI-Name des Treibers, der die Schnittstelle ConnectionFactory implementiert

Erzeugen der JMS-Verbindung:

- aktive Verbindung zu einem JMS-Provider erzeugen
- kapselt die Netzwerk-Kommunikation
- Analog zu JDBC wird der Treiber verwendet, um eine `Connection` anzufordern
  - TopicConnection
  - QueueConnection

Erzeugen einer JMS Session:

- JMS Session ist ein Hilfsobjekt um Nachrichten zu senden oder zu empfangen
- legt die Eigenschaften des Nachrichtenaustauschs fest
- integriert den Nachrichtenaustausch in eine Transaktion
  - TopicSession
  - QueueSession

Lokalisieren einer JMS Destination:

- JMS Destination ist ein Kommunikationskanal zu dem Nachrichten gesendet oder von dem Nachrichten empfangen werden können
- die Auswahl des Kanals erfolgt über ein lookup(CHANNEL), wobei CHANNEL der JNDI-Name des Kanals ist, der beim Server eingetragen wurde

Erzeugen eines JMS Consumers oder eines JMS Producers:

- Empfänger oder Sender

Senden oder Empfangen der Nachricht:

- Typen von Nachrichten
- Text, Bytes, Streams, Objects und Maps
- Nachricht wird instantiiert
  - Nachricht wird über den Producer gesendet
  - Nachricht wird über den Consumer empfangen

jedes Interface hat zwei Ausprägungen analog zu den beiden Kommunikationsmodellen.

Erzeuger-Interface	Point-to-Point	Publish / Subscribe
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	Queue Connection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, -Browser	TopicSubscriber

8.1.1 MDB Topic / Queue anlegen

- Bevor wir die MDB auf den Server schieben können muss ein entsprechendes Topic oder eine entsprechende Queue deklariert werden
- JBoss AS stellt Standarddeklarationen zur Verfügung
  - topic/testTopic
  - queue/testQueue

- globale Deklaration
  - Datei [JBOSS_HOME]\server\default\deploy\jms\jbossmq-destinations-service.xml öffnen und am Ende folgendes einfügen:

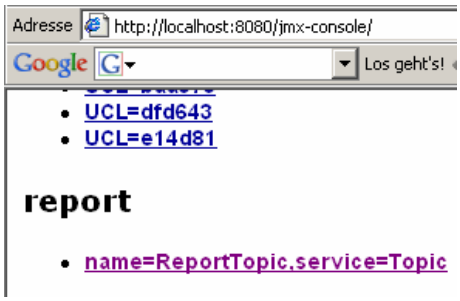
```
<mbean code="org.jboss.mq.server.jmx.Queue"
  name="report:service=Topic,name=ReportTopic">
  <depends optional-attribute-name = "DestinationManager">
    jboss.mq:service=DestinationManager
  </depends>
  <attribute name="JNDIName">
    topic/Report
  </attribute>
</mbean>
```

JBoss Console

```
14:57:38,171 INFO  [ReportTopic] Bound to JNDI name: topic/Report
14:57:38,171 INFO  [testTopic] Bound to JNDI name: topic/testTopic
14:57:38,171 INFO  [securedTopic] Bound to JNDI name: topic/securedTopic
14:57:38,171 INFO  [testDurableTopic] Bound to JNDI name: topic/testDurableTopic
14:57:38,171 INFO  [testQueue] Bound to JNDI name: queue/testQueue
```

JBoss jmx-console

- Deklaration im EJB-Projekt durch Einfügen der Queue durch eine Konfigurations-Datei direkt im EJB-Projekt



report-mt-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean  code="org.jboss.mt.server.jmx.Queue"
          name="report:service=Topic,name=MessageBeanQueue">
    <depends optional-attribute-name="DestinationManager">
       jboss.mt:service=DestinationManager
     </depends>
    <attribute name="JNDIName">
       topic/ReportMDB
     </attribute>
  </mbean>
</server>
```

8.2 Message Driven Bean

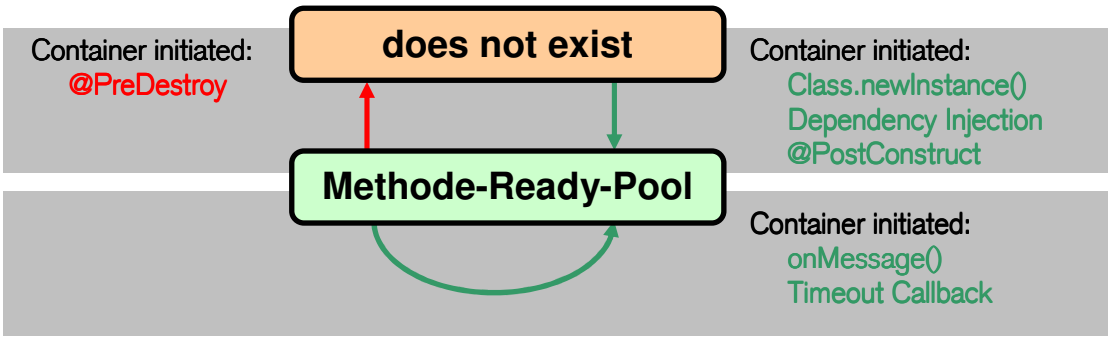
- Session Beans sind Remote-Procedure-Call basierte Komponenten
- werden immer synchron vom Client aufgerufen

- Message-Driven-Bean unterstützt die asynchrone Kommunikation
  - der Client muss nun nicht warten, bis der Server die Aufgabe erledigt hat
  - Message-Bean stellt keine direkte Schnittstelle zur Verfügung
  - Client sendet eine Nachricht an einen Message-Server (Nachrichten-Server)
  - MDB ist durch das Deployment beim Messageserver registriert
  - MDB legt Funktionalität in der onMessage-Methode fest

Annotation @MessageDriven			
Parameter	Beschreibung	Typ	Default
name	expliziter Name der Bean	String	unqualifizierter Klassenname
activationConfig	Providerspezifische Properties für die Konfiguration der vom Container realisierten Anbindung der Bean an das Nachrichtensystem	Activation-Config-Property[]	[]
mappedName	mappen des Namens auf einen weiteren implementierungs-spezifischen (JNDI)-Namen (nicht im Standard-Umfang der Spezifikation)	String	
description	eine Beschreibung für das Bean	String	
messageListener-Interface	Interface für das Nachrichtensystem	Class	Object.class

AnnotationProperty @ActivationConfigProperty			
Parameter	Beschreibung	Typ	Default
acknowledgeMode	- **Auto-acknowledge** = Nachrichten werden vom Container bestätigt und Nachrichten werden nur einmal zugestellt - **Dups-ok- acknowledge** = Nachrichten werden vom Container bestätigt, es können aber Duplikate vorkommen	String	
destination	JNDI der **Queue**- oder **Topic**- Destination	String	
destinationType	Typ der Destination: **javax.jms.Queue** oder **javax.jms.Topic**	String	
messageSelector	Filter für empfangene Nachrichten	String	
subscriptionDurability	- **NonDurable** = Ist MDB nicht bereit so ist Message verloren - **Durable** = JMS-Provider speichert Nachrichten bis MDB bereit	String	

8.3 Bean Life Cycle



LifeCycleMethoden	
Annotation	Beschreibung
@PostConstruct	Wird vom Container aufgerufen nachdem die Bean durch den Container instanziiert worden ist. Für den Client ist der Aufruf dieser Methode transparent.
@PreDestroy	Ressourcen werden durch den Bean Provider freigegeben. Für den Client ist der Aufruf dieser Methode transparent.
onMessage()	Business Methode der MDB
Timeout Callback	Bezieht sich auf eine Container Timer Funktion.

8.4 Übung: Email durch MDB

Erstellen und deployen Sie ein Message Driven Bean (ReportMDB), das bei negativem Kontosaldo über eine MDB ein Email versendet.

Ansatz

```
/**
 * @module jee.2007.Metzler
 * @exercise jee081.server.KontoBean.java
 */
@Stateful
@Remote(Konto.class)
public class KontoBean implements Konto {
  @Resource(mappedName = "TopicConnectionFactory")
  private static TopicConnectionFactory factory;
  @Resource(mappedName = "topic/testTopic")
  private static Topic                  topic;
  ...
  @Override
  public void withdrawMoney(float inAmount) {
    accountBalance -= inAmount;
    if (accountBalance < 0)
      contactReportMDB();
  }
  ...
  private void contactReportMDB() {
```

```
    javax.jms.Connection connect;
    try {
      connect = factory.createConnection();
      Session session = connect.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
      MessageProducer sender = session.createProducer(topic);
      TextMessage msg = session.createTextMessage();
      msg.setText("Balance becomes less than 0");
      sender.send(msg);
      connect.close();
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

ReportMDB

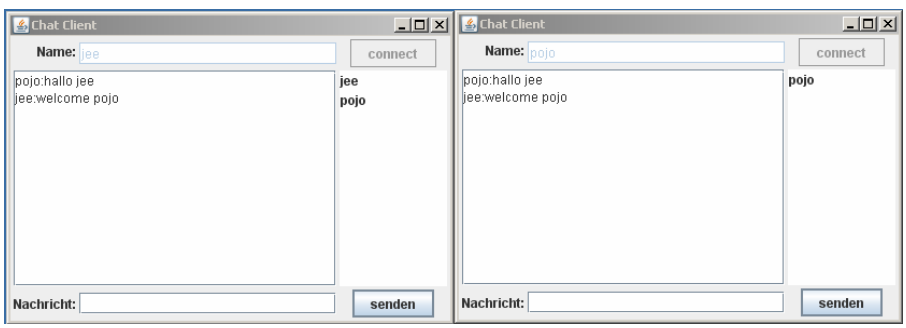
```
/**
 * @module jee.2007.Metzler
 * @exercise jee081.server.ReportMDB.java
 */
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName = "destinationType", propertyValue = javax.jms.Topic"),
  @ActivationConfigProperty(
    propertyName = "destination", propertyValue = "topic/testTopic"),
  @ActivationConfigProperty(
    propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})
public class ReportMDB implements MessageListener {
  @Resource(mappedName = "TopicConnectionFactory")
  ConnectionFactory factory;
  @Resource(mappedName = "topic/testTopic")
  Topic             topic;
  public void onMessage(Message message) {
    System.out.println("\n----------------\n"
        + "ReportMDB - Got message, sending email\n"
        + "----------------");
    // Generate and save email
  }
}
```

8.5 Übung: Chat mit MDB

- Erstellen und deployen Sie ein Message Driven Bean (ChatMDB), das einen Chatroom (Queue) bereitstellt.
- wählen Sie eine eigene Queue : **queue/myChatRoom**

```
/**
 * @module jee.2007.Metzler
 * @exercise jee082.server.ChatMDB.java
 * @author mitp - Enterprise JavaBeans 3.0
 */
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
       propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
       propertyValue = "queue/myChatRoom"),
    @ActivationConfigProperty(propertyName = "messageSelector",
       propertyValue = "MessageFormat = 'ChatMessage'"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
       propertyValue = "Auto-acknowledge") })
public class ChatMDB implements MessageListener {
  @Resource(mappedName = "ConnectionFactory")
  ConnectionFactory factory;
  @Resource(mappedName = "topic/testTopic")
  Topic             topic;
  public void onMessage(Message message) {
    try {
      TextMessage txtMsg = (TextMessage) message;
      String text = txtMsg.getText();
      System.out.println(text);
      if (factory != null && topic != null) {
        Connection connect = factory.createConnection();
        Session session = connect.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer sender = session.createProducer(topic);
        TextMessage msg = session.createTextMessage();
        msg.setText(text);
        sender.send(msg);
        connect.close();
      } else {
        System.out.println("factory or topic not found");
      }
    } catch (JMSException ex) {
      ex.printStackTrace();
    }
  }
}
```

- Swing Client Applikation



9 Interceptoren

- Nutzung von gemeinsamem Code = Crosscutting Concern
- zentrale Codesegmente für die Applikation
- wieder verwendbar
- für Session Beans und Message Driven Beans
- Interceptoren sind Zustandslos
  - können keine Informationen austauschen

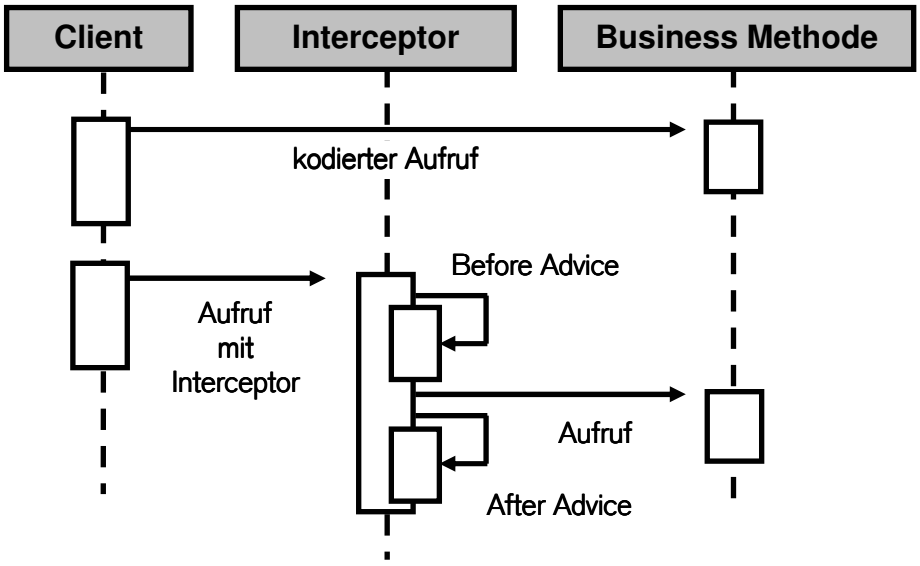
- Interceptoren für Geschäftsmethoden
  - fangen Methodenaufrufe ab
  - führen zentrale Codesegmente aus (z.B. Logging)

- Lebenszyklus Callbacks
  - stellen Callbacks des Kontainers zur Verfügung

9.1 Annotation

Annotation @Interceptors			
Parameter	**Beschreibung**	**Typ**	**Default**
value	alle Interceptoren, die bearbeitet werden sollen	Class[]	

9.2 Funktionsweise



9.3 Übung : Timelogger

Zeitlogger für die Servermethoden per Interceptor

9.3.1 Klassenweiter Interceptor

- durch Annotation @Interceptors
  - mit Klasse als Parameter
- gilt für alle Methodenaufrufe innerhalb der Klasse

```
/**
 * @module jee.2007.Metzler
 * @exercise jee091.server.KontoBean.java
 */
@Interceptors(TimingInterceptor.class)
@Stateful
public class KontoBean implements Konto {
```

- Definition von mehreren Interceptors
  - Aufruf in der Reihenfolge der Deklaration

```
@Interceptors(Interceptor1.class, Interceptor2.class)
@Stateful
public class KontoBean implements Konto {
```

```
/**
 * @module jee.2007.Metzler
 * @exercise jee091.server.TimingInterceptor.java
 */
public class TimingInterceptor {
  @AroundInvoke
  public Object timing(InvocationContext ctx) throws Exception {
    long invoke = System.nanoTime();
    String c = "";
    String m = "";
    try {
      c = ctx.getTarget().getClass().getName();
      m = ctx.getMethod().getName();
      return ctx.proceed();
    } finally {
      long time = System.nanoTime() - invoke;
      System.out.println("Call to " + c + "." + m + " took "
          + NANOSECONDS.toMicros(time) + " Micros");
    }
  }
}
```

Output:

```
Call to server.KontoBean.depositMoney took 57 Micros
Call to server.KontoBean.withdrawMoney took 12 Micros
Call to server.ZinsBean.calculate took 29 Micros
Call to server.KontoBean.calculateRate took 3164 Micros
Call to server.KontoBean.getAccountBalance took 12 Micros
Call to server.KontoBean.tellRate took 12 Micros
Call to server.KontoBean.closeAccount took 545 Micros
```

9.3.2 Interceptoren Methoden

- durch Annotation @AroundInvoke
- Aufrufparameter können analysiert (manipuliert) werden
- Signature der Interceptor-Methode ist unabhängig von EJB Interface
- Verkettung mehrerer Interceptoren Methoden ist möglich
- nur ein Interceptor pro Klasse erlaubt

```
/**
 * @module jee.2007.Metzler
 * @exercise jee092.server.KontoBean.java
 */
@Stateful
public class KontoBean implements Konto {
...
  @AroundInvoke
  public Object abfangen(InvocationContext ic) throws Exception {
    System.out.println(ic.getMethod().getName()
        + " () wird aufgerufen");
    return ic.proceed();
  }
}
```

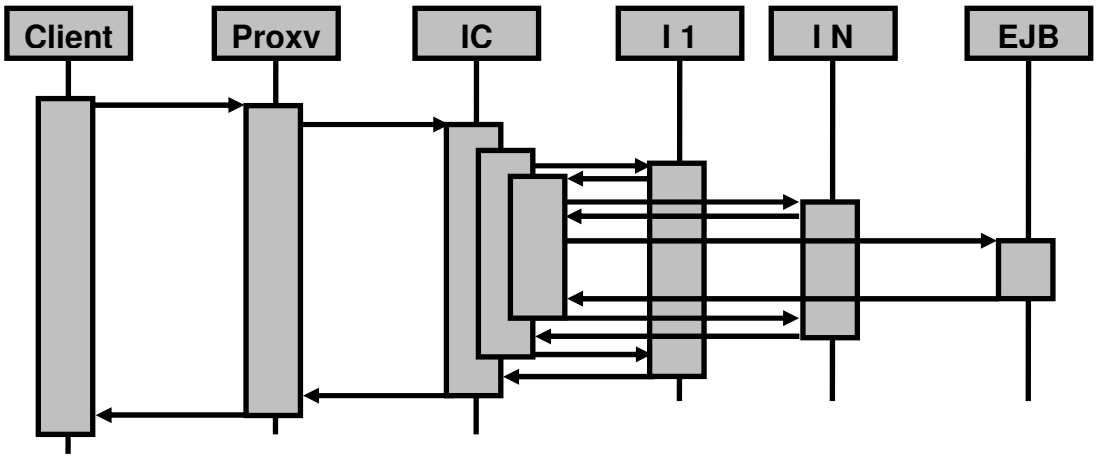
Output:

```
depositMoney () wird aufgerufen
withdrawMoney () wird aufgerufen
calculateRate () wird aufgerufen
getAccountBalance () wird aufgerufen
tellRate () wird aufgerufen
closeAccount () wird aufgerufen
```

- @AroundInvoke leitet sämtliche Methodenaufrufe durch die Interceptor Methode
- Funktionen des Interface InvocationContext enthält Informationen der aufgerufenen Methode
- proceed() definiert den weiteren Ablauf
  - nächste Interceptor-Methode
  - ursprünglich aufgerufene Methode

- stoppt den funktionellen Ablauf (um Sicherheitsverhalten festlegen)
- daher auch throws Exception (als unchecked Exception)
- Return Type Object erlaubt Manipulation der Parameter

9.4 InvocationContext



- Client ruft Business Methode auf
- Proxy erzeugt InvocationContext (IC)
  - Proxy ruft InvocationContext auf
- InvocationContext ruft Interceptor 1 auf (I1)
  - Interceptor 1 beendet mit proceed()
- …
- InvocationContext ruft Interceptor N auf (IN)
  - Interceptor N beendet mit proceed()
- InvocationContext ruft EJB auf (BusinessMethode)
- InvocationContext ruft Interceptor N auf (IN)
- …
- InvocationContext ruft Interceptor N auf (I1)

Methoden des InvocationContext	
Methode	**Beschreibung**
getTarget()	Gibt EJB-Instanz, auf der ein Methodenaufruf erfolgen soll, zurück
getMethode()	Gibt Methode, auf der ein Aufruf erfolgen soll, zurück
getParameters()	liefert die Übergabeparameter
setParameters()	erlaubt die Übergabeparameter zu manipulieren
getEJBContext()	liefert den EJB Kontext der aufzurufenden EJB Instanz damit hat der Interceptor Zugriff auf alle Ressourcen der EJB
getContextData()	stellt Map für die Informationsübergabe zwischen verschiedenen Interceptoren zur Verfügung
proceed()	beendet den Interceptor

9.5 Interceptor als Lifecycle-Callback

- allgemein definierte Lyfecycle-Methode (wieder verwendbar)
- als Interceptor annotiert
- in eigener Klasse definiert

```
public class myCallBack {
  @PostActivate
  void intercept(InvocationContext ic) throws Exception {
  ...
    ctx.proceed();
  }
}
```

List der Callbacks und deren Verwendung

Callback Methoden des EJB Kontainers			
Callback	**Paket**	**-Bean**	**Beschreibung**
@PostConstruct	javax,annotation	Stateless Session- Stateful Session- MessageDriven-	nach der Instanziierung nach Dependency Injection vor der Business-Methode
@PreDestroy	javax,annotation	Stateless Session- Stateful Session- MessageDriven-	vor dem Beenden der Instanz
@PostActivate	javax.ejb	Stateful Session-	nach Reaktivierung der Bean aus dem Sekundärspeicher
@PrePassivate	javax.ejb	Stateful Session-	vor Passivierung der Bean in den Sekundärspeicher
@Timeout	javax.ejb	Stateless Session- Stateful Session- MessageDriven-	durch Timeout eines Timers

9.6 Interceptoren ausschliessen

- @ExcludeClassInterceptors unterdrückt alle für die Klassen definierten Interceptoren auf Methodenebene
  - gilt nicht für @AroundInvoke annotierte Methoden

```
@ExcludeClassInterceptors
public float tellRate() {
  return rate;
}
```

- @ExcludeDefaultInterceptors unterdrückt alle im Deploymentdescriptor definierten Default-Interceptoren

```
@ExcludeDefaultInterceptors
@Interceptors(TimingInterceptor.class)
@Stateful
@Remote(Konto.class)
public class KontoBean implements Konto {
```

10 Timer Service

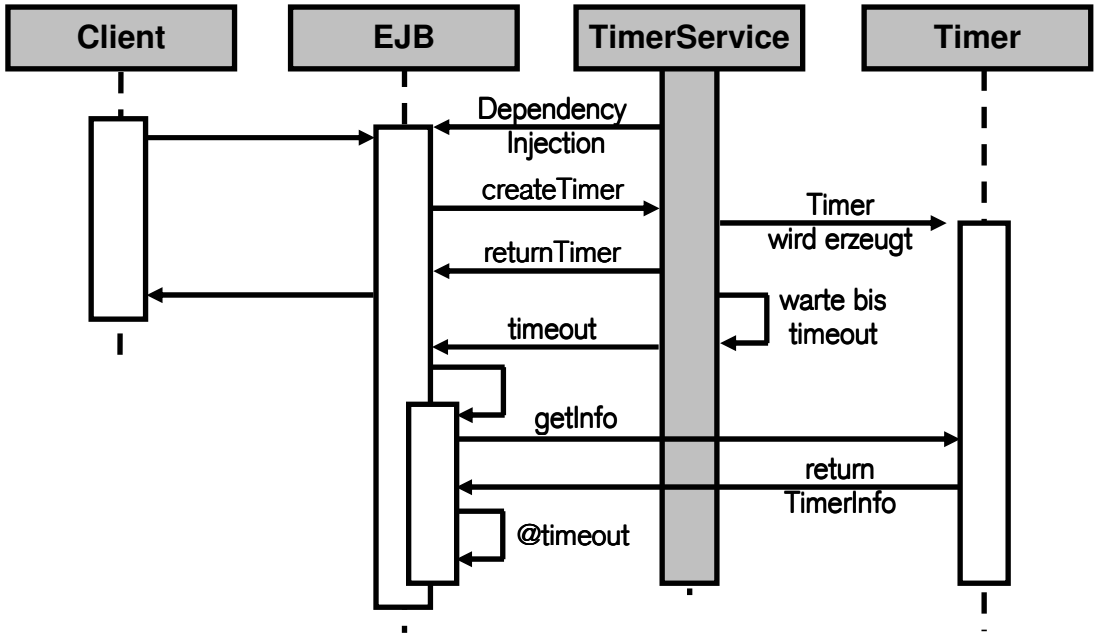
Kontainer Funktionalität um zu vorprogrammierten Zeiten einen Vorgang auszuführen:

- Löschung verwaister Daten
  - jede Nacht alle Datensätze, die älter als eine Woche sind, löschen
- Reporting
  - Bericht über die täglich versendete Anzahl Email
- Monitoring
  - Überprüfung ob ein Dienst (EJB) zur Verfügung steht
- Remainder
  - Nachrichten senden bis eine Rückmeldung eintrifft
- Workflow
  - anstehende Aufträge in eine Warteschlange stellen

10.1 Eigenschaften

- EJB Timer Service
- Batch Jobs duch den Kontainer gestartet und verwaltet
- als Kurzzeit-Timer ungeeignet
- Kontainer benachrichtigt EJB, der Timer definiert
  - Stateless Session Bean
  - Message Driven Bean
  - Version 2.1 erlaubt auch die Benachrichtigung an Entity Beans
- Timer EJB führt die mit @Timeout annotierte Methode aus
- Timer EJB ist Stateless
  - Kontainer führt beliebige Instanz der EJB aus

10.2 Ablauf



10.3 Timer starten

- Timer erzeugen
  - über EJB Kontext – TimerService ts = SessionContext.getTimerService()
  - über Injection - @Ressource javax.ejb.TimerService ts

```
@Stateful
public class myTimerBean implements someInterface {
  @Resource
  SessionContext sc;
  public void setUPTimer() {
    TimerService ts = sc.getTimerService();
    long min15 = 15 * 60 * 1000;
    Timer t = ts.createTimer(min15, "Info über Timer");
  }
   ...
}
```

folgende Methoden stehen zum Erstellen eines Timers zur Verfügung:

TimerService create Methoden				
Methode	**relativ**	**absolut**	**einmalig**	**periodisch**

createTimer(long d, Serializable info)
`long tenDays = 1000 * 60 * 60 * 24 * 10`
`createTimer(tendays,null)`
	✓		✓	

createTimer(long d, long interval, Serializable info)
`long oneMinute = 1000 * 60 * 1`
`long oneHour = 1000 * 60 * 60`
`createTimer(oneMinute, oneHour, null)`
	✓			✓

createTimer(Date expiration, Serializable info)
`Calendar Date = Calendar.getInstance()`
`date.set(2008, Calendar.JANURAY, 1)`
`createTimer(date.getTime(), null)`
		✓	✓	

createTimer(Date ex, long interval, Serializable info)
`Calendar Date = Calendar.getInstance()`
`date.set(2008, Calendar.JANURAY, 1)`
`long oneWeek = 1000 * 60 * 60 * 24 * 7`
`createTimer(date.getTime(), oneWeek, null)`
		✓		✓

10.4 Timer ausführen

- Kontainer ruft @Timeout annotierte Methode beim definierten Zeitintervall auf

```
@Timeout
public void fire(Timer timer) {
  String info = timer.getInfo();
}
```

10.5 Timer beenden

- timer.cancel() löscht den definierten Timer Service

```
public void stopAllMonitors() {
  for (Object o : timerService.getTimers()) {
    // cancel all TimerService
    Timer timer = (Timer) o;
    timer.cancel();
  }
}
```

10.6 Timer Schnittstelle

weite Funktionalitäten im Zusammenhang mit einem Timer:

- getTimeRemaining()
  - verbleibende Zeitspanne bis der Timer abläuft
- getNextTimeout()
  - Zeitpunkt des nächsten Timeout
  - nur bei periodischem Timer
- getHandle()
  - serialisierbare Referenz auf das Timer-Object

10.7 Übung: InterestMonitor

Ein Stateless Session Bean soll als Timer Monitor den Zinszuwachs periodisch anzeigen:

- 1 Million Kontoguthaben
- 10 % Zins pro Jahr
- 0.20 Einheiten Zinszuwachs pro Minute

Timer Service : InterestMonitorBean

```
/**
 * @module jee.2007.Metzler
 * @exercise jee101.server.InterestMonitorBean.java
 */
@Stateless
public class InterestMonitorBean implements InterestMonitor {
  @Resource
  TimerService             timerService;
  private String           monitor = "InterestMonitor";
  public static final long TIMEOUT = 15 * 1000;        // 15 Sec.
  private Konto            konto;
  public void startMonitor() {
    timerService.createTimer(TIMEOUT, TIMEOUT, monitor);
  }
  public void stopAllMonitors() {
    for (Object o : timerService.getTimers()) {
      // cancel all TimerService
      Timer timer = (Timer) o;
      timer.cancel();
    }
  }
  @Timeout
  public void tellInterest(Timer timer) {
    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    System.out.println("Interest gained : "
        + df.format(konto.calculateInterest()) + " on Amount "
        + df.format(konto.getAccountBalance()) + " with "
        + df.format(konto.tellRate()) + " % Interest");
  }
  @PostConstruct
  public void readKontoRef() {
    try {
      // read Object Reference from File
      FileInputStream fi = new FileInputStream("c:\\" + monitor
          + ".ser");
      ObjectInputStream oi = new ObjectInputStream(fi);
      // read the object from the file and cast it to a Konto
      konto = (Konto) oi.readObject();
      oi.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Client : speichert KontoReferenz

```
/**
 * @module jee.2007.Metzler
 * @exercise jee101.client.KontoClientwithInterestMonitor.java
 */
public class KontoClientwithInterestMonitor {
  public static void main(String[] args) {
    final float amount = 1000000f;
    final String monitor = "InterestMonitor";
    try {
      InitialContext ic = new InitialContext();
      // lookup Konto
      Konto konto = (Konto) ic.lookup("KontoBean/remote");
      // create Account and deposit Amount
      konto.depositMoney(amount);
      System.out.println("Account with " + amount
          + " created (InterestRate is : " + konto.tellRate()
          + " %)");
      // write Object Reference to serialized File
      Object or = konto.getRemoteReference();
      FileOutputStream f;
      f = new FileOutputStream("c:\\" + monitor + ".ser");
      ObjectOutputStream os = new ObjectOutputStream(f);
      os.writeObject(or);
      os.flush();
      os.close();
      // lookup InterestMonitor
      InterestMonitor im = (InterestMonitor) ic
          .lookup("InterestMonitorBean/remote");
      // start InterestMonitor
      im.startMonitor();
      System.out.println("TimerService started ...");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Output:

```
gained : 0.05 on Amount 1'000'000.00 with 10.00 % Interest
gained : 0.10 on Amount 1'000'000.00 with 10.00 % Interest
gained : 0.15 on Amount 1'000'000.00 with 10.00 % Interest
gained : 0.20 on Amount 1'000'000.00 with 10.00 % Interest
```

11 Webservice

zwei Szenarien werden zur Verfügung gestellt:

- Session Bean stellt WebService zur Verfügung
  - @WebService exportiert alle Methoden der Bean
  - @WebMethod annotiert die zu exportierende Methode
  - AS (JBoss) erstellt WSDL beim Deployment

- SessionBean ist Client für WebService
  - WSDL Datei (Beschreibungsdatei für den WebService) steht zur Verfügung
  - Session Bean implementiert SEI (Service Endpoint Interface)

11.1 Definition eines Web Service

- @WebService definiert eine Klasse als Webservice und macht alle öffentlichen Methoden sichtbar

Annotation @WebService			
Parameter	**Beschreibung**	**Typ**	**Default**
endpointInterface	vollqualifizierter Name des Service-Endpoint-Interface	String	
name	Name des Port-Typs innerhalb der WSDL-Definition	String	Name der implementierenden Klasse
PortName	Name des Webservice-Ports innerhalb der WSDL-Definition	String	[name]
serviceName	Name des Webservices innerhalb der WSDL-Definition	String	[name]
targetNamespace	Namensraum für Port-Typ und untergeordnete Elemente	String	
wsdlLocation	Angabe einer WSDL-Datei um eine automatische Generierung zu verhindern (eigene Definition zu verwenden)	String	

11.2 WebService Clients

- JAX-WS-Client
  - Remote Client
  - ohne JNDI Zugriff
  - müssen JAX-WS (Java API for XML) implementieren

- Java-EE-Clients
  - enthalten JNDI Kontext
  - lookup gibt Referenz auf SEI (Service Endpoint Interface)

11.3 Übung : WebService zur Zinsberechnung

- ServiceEndpoint Interface deklariert WebService Schnittstelle

ServiceEndpoint Interface

```
/**
 * @module jee.2007.Metzler
 * @exercise jee111.server.ServiceEndpoint.java
 */
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface ServiceEndpoint extends Remote {
  @WebMethod
  public @WebResult(name = "ertrag")
  float calculate(@WebParam(name = "betrag")
  float amount, @WebParam(name = "zins")
  float rate, @WebParam(name = "jahre")
  int years) throws RemoteException;
}
```

WebService

```
/**
 * @module jee.2007.Metzler
 * @exercise jee111.server.ZinsBean.java
 */
@WebService(endpointInterface = "server.ServiceEndpoint")
@Interceptors(TimingInterceptor.class)
@Stateless(name = "Rate")
@Local(ZinsLocal.class)
public class ZinsBean implements ServiceEndpoint {
  public float calculate(float amount, float rate, int years) {
    return amount
        * (float) Math.pow((double) ((rate / 100.0f) + 1.0f),
            (double) years);
  }
```

WebService Client

```
/**
 * @module jee.2007.Metzler
 * @exercise jee111.client.ServiceClient.java
 */
public class ServiceClient {
  public static void main(String[] args) throws Exception {
    URL url = new URL(
        "http://127.0.0.1:8080/ZinsBeanService/ZinsBean?wsdl");
    QName qname = new QName("http://server/", "ZinsBeanService");
    ServiceFactory factory = ServiceFactory.newInstance();
    Service service = factory.createService(url, qname);
    ServiceEndpoint se = (ServiceEndpoint) service
        .getPort(ServiceEndpoint.class);
    float rate = se.calculate(100.0f, 5.0f, 1);
```

```
    System.out.println("Result : " + rate);
  }
}
```

12 JEE Security

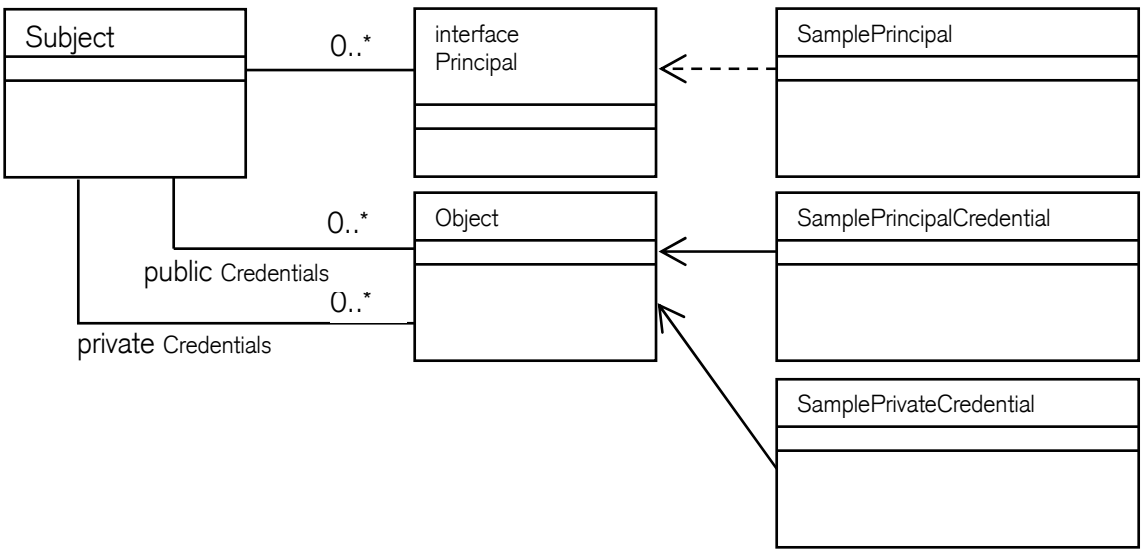
- "so wenig wie möglich" programmatische Sicherheit in die Beans
- viel mehr **deklarative** Sicherheitseinstellungen über den Server tätigen

Securitythematik

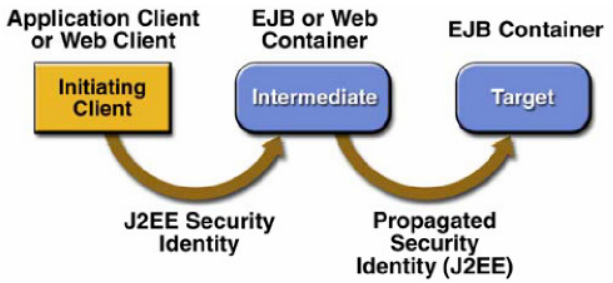
- Authentisierung von Clients
- sichere Kommunikationsformen zwischen Client und Server
- sichere Kommunikation zwischen Containern
- rollenbasierter Zugriffsschutz durch "Security Identity Propagation" innerhalb des Systems
  - **Weiterreichen der Identität** eines EJB-aufrufenden Client

Securitybereitstellung

- Applikation-Client-Authentisierung über den **Java Authentication and Authorisation Service (JAAS)** seit EJB 2.1
- Client wird bei Anmeldung am System als unidentifiziertes Subject angesehen
- Nach erfolgreicher Authentisierung werden Client-Subject (u.A. mehrere) sogenannte **Principals** und **Credentials** zugeordnet
- Principals werden über das `java.security.Principal-Interface` implementiert
  - assoziieren das Client-Subject mit einem Benutzernamen oder –konto
  - sicherheitsrelevante Daten wie etwa ein Kerberos-Ticket
  - Loginpasswort



- Principals werden später wiederum bestimmten Rollen zugeordnet
  - sind Teil der umzusetzenden Sicherheitspolitik des Systems
  - "principal-to-role-mapping"



12.1 Security Roles

Security Rollen werden in verschiedenen Entwicklungsbereichen definiert.

Bean Provider (EJB-Entwickler)

- "so wenig wie möglich" Sicherheitsfragen
- Sicherheitspolitik nicht hart im Code eingebettet
- später über den Container kontrollierbar und konfigurierbar
- "programmatische Sicherheit" als Alternative, Rollen müssen im Deployment Descriptor referenziert sein

```
/**
 * @module jee.2007.Metzler
 * @exercise jee121.server.KontoBean.java
 */
  public void withdrawMoney(float inAmount) {
    if (inAmount < 1000f) {
      accountBalance -= inAmount;
    } else if (sc.isCallerInRole("admin")) {
      accountBalance -= inAmount;
    } else {
      System.out.println("Withdraw Denied : Caller is "
          + sc.getCallerPrincipal());
    }
...
```

Application Assembler

- stellt EJB-Komponenten zusammen
- konfiguriert diese verkaufsfertig
- deklariert zugeschnittene Sicherheitsrollen
- ordnet diese den Rollen des Bean-Providers zu

```
<assembly-descriptor>
     <security-role>
       <role-name>verwalter</role-name>
     </security-role>
     <security-role>
       <role-name>kunde</role-name>
     </security-role>
...
```

- deklariert die Rechte der einzelnen Methoden
  - Name der erlaubten Methode
  - Name und eventuelle Parameter, um sie eindeutig zu identifizieren
  - *, um alle Methoden für eine Rolle freizugeben

  - Das <method-intf> Element definiert das entspechende Interface 'Home', 'Remote', 'LocalHome' oder 'Local'

```
<method-permission>
        <role-name>verwalter</role-name>
        <method>
          <ejb-name>KontoBean</ejb-name>
          <method-name>*</method-name>
        </method>
      </method-permission>

      <method-permission>
        <role-name>kunde</role-name>
        <method>
          <ejb-name>KontoBean</ejb-name>
          <method-name>depositMoney</method-name>
        </method>
        <method>
          <ejb-name>KontoBean</ejb-name>
          <method-name>withdrawMoney</method-name>
        </method>
...
```

Deployer

- definiert Systemrollen
  - ist auf gute Spezifikation des Assemblers angewiesen
  - muss die Rollenreferenzen des Programmierers kennen

- letztendliche Zuordnung von Benutzern bzw. Benutzergruppen auf Rollen
- Principal-to-Role-Mapping ist jeweils nur für eine einzelne Applikation innerhalb des Containers gültig, d.h. nur für ein Applikations-jar-File

```
        <security-role-ref>
          <role-name>verwalter</role-name>
          <role-link>admin</role-link>
        </security-role-ref>
```

12.2 Client-Tier Sicherheit

ApplicationClients

- JAAS implementiert eine Version des Pluggable Authentication Module (PAM) Frameworks
- erlaubt den Einsatz von eigens entwickelten Loginmodulen
- Loginmodule werden auf Seiten des Clients eingesetzt
- verschiedene Loginvarianten werden unterstützt
  - Name/Passwort-Dialoge
  - externer Hardware und Smartcards
  - Zertifikate weiterreichen

- Loginmodule selbst steuern den Loginvorgang und sammeln Authentisierungsdaten über sogenannte Callbackhandler
- Satusinformationen vom Server werden an den Client zurückgereicht
  - durch Interface javax.security.auth.callback.CallbackHandler

```
AppCallbackHandler handler = new AppCallbackHandler(name, password);
LoginContext lc = new LoginContext("EJBSecurityTest", handler);
lc.login();
```

WebClients

Die Spezifikation erlaubt prinzipiell drei Arten von Webclient-Authentisierung, die über den Server einstellbar sein müssen:

- **http** bzw. http über SSL (**https**) – Es werden http-Formulare mit einfacher Login/Passwort-Abfrage benutzt.
- **formular** – Agiert auch über http/https. Diese Einstellung erlaubt jedoch eigens definierte und umfangreichere Abfragen. Es können Mechanismen wie Cookies bzw. Session-Ids benutzt werden.
- client-certificate – Erlaubt die Authentifizierung von Webclients über Zertifikate. Für diese Option muss zur gegenseitigen Authentisierung (mutual authentication) natürlich auch auf Client-Seite ein X.509-Certificate installiert sein.

Der im Mid-Tier genauer beschriebene rollenbasierte Zugriffsschutz wird auch bei Webapplikationen vom Container übernommen. Es sind auch bei Webanwendungen programmatische statt vom Container übernommene Sicherheitsabfragen möglich: Über das `HttpServletRequest` Interface kann mit der Methode `isUserInRole` festgestellt werden, ob der Benutzer in einer bestimmten Rolle ist. Um die Identität (in Form eines `java.security.Principal`) zu erhalten, kann die Methode `getUserPrincipal` verwendet werden.

12.3 JNDI Sicherheit

- Loging Spoofing
- unberechtigte JNDI Server Einträge

- JEE-Spezifikation rät zur Sicherung der Name-Services und zur Verwendung von "CORBA specified naming services" (Cos-Naming) über eine sichere Verbindung
- Umsetzung ist den Serverentwicklern überlassen, da auch die verschiedenen CORBA-Implementierungen nicht immer eine Sicherung ihrer Namensdienste unterstützen
-

```
// Set up the environment properties
Hashtable h = new Hashtable();
h.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.cosnaming.CNCtxFactory");
h.put(Context.PROVIDER_URL, "iiop://localhost:3700");
h.put(Context.SECURITY_PRINCIPAL, "username");
h.put(Context.SECURITY_CREDENTIALS, "password");
// Get an InitialContext
return new InitialContext(h);
```

12.4 Realms

- Realms definiert eine Gruppe von Benutzern
- folgen alle dem selben Mechanismus zur Authentisierung

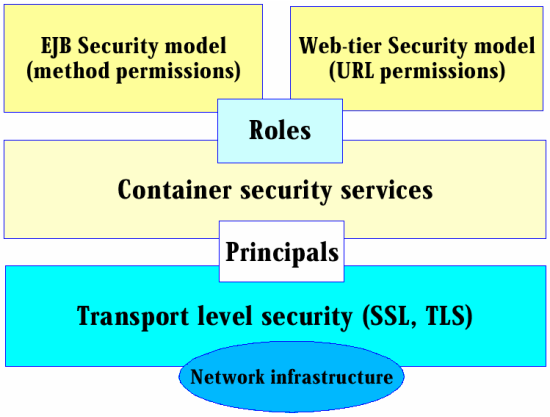
12.5 Legacy Tier Security

- zwei Möglichkeiten, den Container, bzw. die Beans die in ihm laufen, gegenüber Legacy-Systemen wie etwa Datenbanken oder Gateways zu authentisieren
  - Container-Managed
  - Bean-Managed

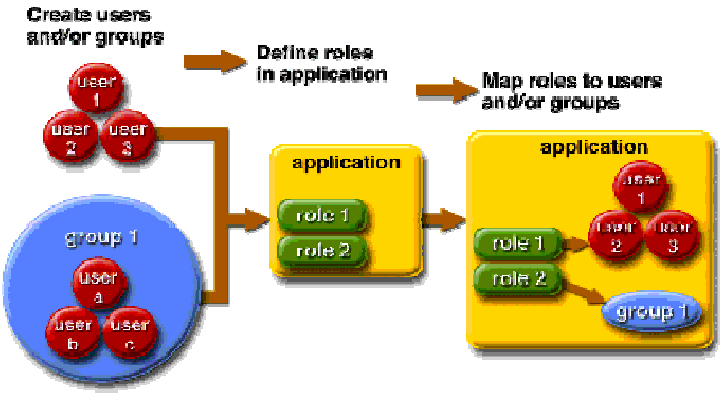
12.6 Declarative vs Programmatic Security

	Web Tier	EJB Tier	deklarative Sicherheit	programmatische Sicherheit
Zugriffskontrolle	Web Resources	Bean Methods	Deployment Descriptor	Programm
Deklaration	web.xml	ejb-jar.xml		
Umsetzung	Web Container	EJB Container	Container	Programm
Codierung	Servlet / JSP	EJB Bean	*"all or nothing"*	Logikbasiert

12.7 JEE Security Architecture

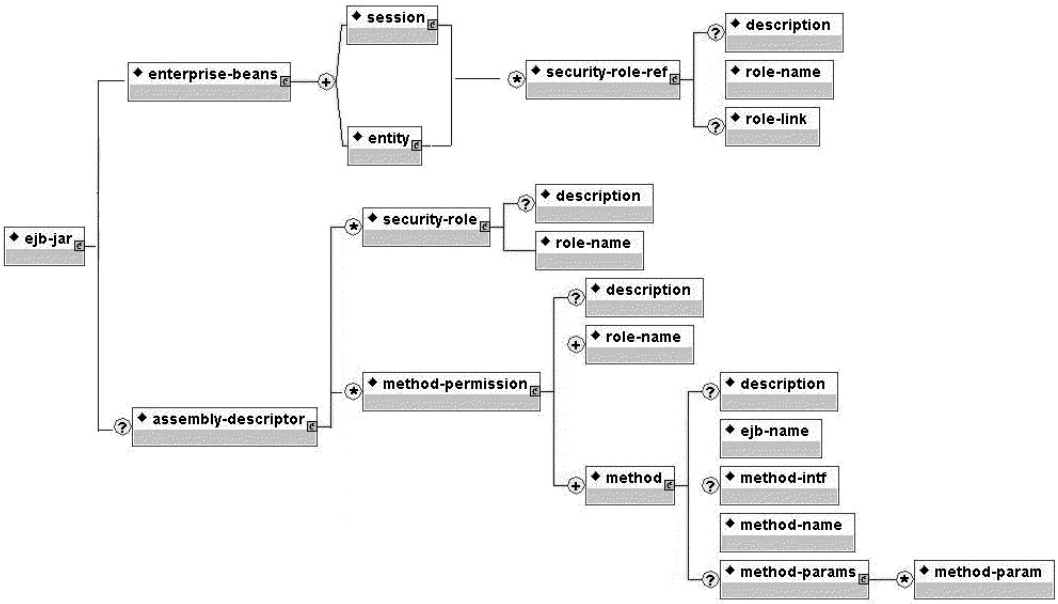


12.8 Role Mapping



12.9 ejb-jar.xml Security Deployment Descriptor Security Elemente

Die folgende Graphik zeigt die Security Related Elements des ejb-jar.xml Deployment Descriptors.



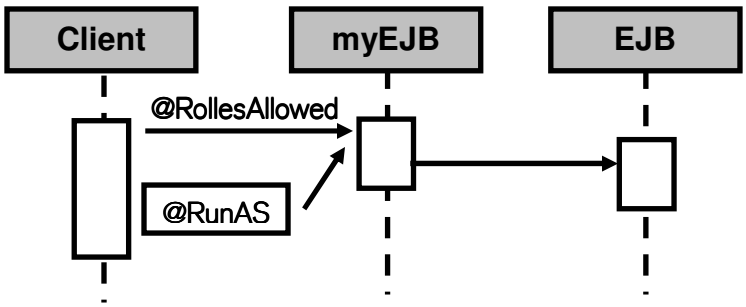
12.10 Security Annotations

EJB Security	
Annotation	**Beschreibung**
@DenyAll	- nur auf Methodenebene - sperrt Zugang zu dieser Methode - sinnvoll bei geerbter Security
@PermitAll	- alle Rollen bekommen Zugriff - auf Klassenebene einsetzbar
@RolesAllowed	- selektioniert Methodenzugriff per Liste
@RunAs	- legt die Rolle zur Ausführung fest, ungeachtet der Client Authorisierung

RunAS

- definiert das weitere Security-Verhalten

Annotaiton @RunAs			
Parameter	**Beschreibung**	**Typ**	**Default**
value	Name der Rolle, die verwendet werden soll	String	




```
@RolesAllowed("guest")
@RunAs("admin")
public class myEJB {
```

- Methoden der Klasse myEJB benötigen die Rolle "guest" damit sie ausgeführt werden können
- Methoden der Klasse myEJB werden in der Rolle "admin" ausgeführt

DeclareRoles

- Auflistung der im Bean verwendeten Rollen
- werden im Zusammenhang mit programmatischer Sicherheit verwendet
  - isCallerInRole()
- annotierte Rollen sind Referenzen
  - werden im Deployment Descriptor auf System Rollen gemappt

Annotation @DeclareRoles			
Parameter	**Beschreibung**	**Typ**	**Default**
value	Auflistung der Rollennamen, die abgefragt werden	String[]	

12.11 EJB Security

definiert rollenbasiertes Sicherheitsmodell:

- falls aktive, bekommen nur Clients mit entsprechenden Rollen Zugriff auf Methoden
  - deklarativ
  - programmatisch
- auf Session- und MessageDrivenBeans anwendbar
- Zugriffsrechte auf EJB Methoden festlegen:
  - ejb-jar.xml Deployment Descriptor
  - per Annotation

Deklarative Sicherheit

- Zugriffsrechte festlegen
  - Security Roles deklarieren

```
<security-role>
   <role-name>verwalter</role-name>
</security-role>
<security-role>
   <role-name>kunde</role-name>
</security-role>
```

  - Method Permission deklarieren

```
<method-permission>
   <role-name>verwalter</role-name>
   <method>
      <ejb-name>KontoBean</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>
```

```
<method-permission>
   <role-name>kunde</role-name>
   <method>
      <ejb-name>KontoBean</ejb-name>
      <method-name>depositMoney</method-name>
   </method>
   <method>
      <ejb-name>KontoBean</ejb-name>
      <method-name>withdrawMoney</method-name>
   </method>
   ...
</method-permission>
```

Programmatische Sicherheit

- Zugriffsrechte festlegen (ejb-jar.xml)
  - Security Roles deklarieren

```
<security-role>
   <role-name>verwalter</role-name>
</security-role>
<security-role>
   <role-name>kunde</role-name>
</security-role>
```

  - Security Role Reference deklarieren

```
<security-role-ref>
   <role-name>admin</role-name>
   <role-link>verwalter</role-link>
</security-role-ref>
```

    Link stellt Beziehung zwischen Rollennamen im Bean-Source-Code und dem Security-Rollennamen dar.

  - Bean-Methoden Implementierung entscheidet über Zugriff

```
public void withdrawMoney(float inAmount) {
  if (inAmount < 1000f) {
    accountBalance -= inAmount;
  } else if (sc.isCallerInRole("verwalter")) {
    accountBalance -= inAmount;
  } else {
    System.out.println("Withdraw Denied : Caller is "
        + sc.getCallerPrincipal());
  }
```

Security Domain

- Security Domain definieren (LDAP, DB, Dateien)
  - definieren User und entsprechende Rollen
  - Datei [JBoss_Home]/server/jee/conf/login-config.xml definiert bestehende Domains
- EJB müssen Teil einer Security Domain sein

Database-Server-Login-Modul

```
<policy>
  <!-- Security domain for JEE Samples -->
  <application-policy name="jee">
    <authentication>
      <login-module code=
          "org.jboss.security.auth.spi.DatabaseServerLoginModule"
        flag="required">
        <module-option name="unauthenticatedIdentity">
          guest
        </module-option>
        <module-option name="dsJndiName">
          java:jdbc/jee
        </module-option>
        <module-option name="principalsQuery">
          SELECT PASSWD FROM JEE_USERS WHERE USERID=?
        </module-option>
        <module-option name="rolesQuery">
          SELECT ROLEID, 'Roles' FROM JEE_ROLES WHERE USERID=?
        </module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

Login-Context `[JBoss-Home]/client/auth.conf`

```
EJBSecurityTest {
org.jboss.security.ClientLoginModule required debug=false;
};
```

Security-Domain `jboss.xml`

```
<?xml version="1.0" encoding ="UTF-8"?>
<jboss>
  <security-domain>java:/jaas/EJBSecurityTest</security-domain>
  <unauthenticated-principal>guest</unauthenticated-principal>
</jboss>
```

12.12 Übung : UsernamePasswordHandler / JNDI Sicherheit

users.properties

```
admin_N=admin_P
chef=bank
system_N=system_N
customer_N=customer_P
client=king
```

roles.properties

```
admin_N=verwalter,kunde
chef=verwalter,kunde
customer_N=kunde
client=kunde
```

User / Passwort Login Context

```
/**
 * @module jee.2007.Metzler
 * @exercise jee121.client.secureKontoClient.java
 */
public class secureKontoClient {
  public static void main(String[] args) {
    // Set User and Password
    String name = "admin_N";
    String pass = "admin_P";
    // Set to actual Path
    System.setProperty("java.security.auth.login.config",
        "s:/jee/jboss-4.2.0.GA/client/auth.conf");
    // [JBoss-Home]/client/auth.conf:
    // EJBSecurityTest {
    // org.jboss.security.ClientLoginModule required debug=false;
    // };
    try {
      UsernamePasswordHandler handler = null;
      handler = new UsernamePasswordHandler(name, new String(pass)
          .toCharArray());
      LoginContext lc = new LoginContext("EJBSecurityTest",
          handler);
      try {
        lc.login();
        System.out.println("Login successfull");
      } catch (LoginException le) {
        System.out.println("Login failed");
        le.printStackTrace();
      }
      InitialContext ctx = new InitialContext();
...
```

JNDI Security

```
/**
 * @module jee.2007.Metzler
 * @exercise jee122.client.secureKontoClient.java
 */
public class secureKontoClient {
  public static void main(String[] args) {
    Properties props = new Properties();
    props.setProperty(
       Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");
    props.setProperty(
       Context.URL_PKG_PREFIXES,"org.jboss.naming.client");
    props.setProperty(
       Context.PROVIDER_URL, "jnp://localhost:1099");
    props.setProperty(
       Context.SECURITY_PRINCIPAL, "customer_N");
    props.setProperty(
       Context.SECURITY_CREDENTIALS, "customer_P");
    props.setProperty(
       Context.INITIAL_CONTEXT_FACTORY,
        "org.jboss.security.jndi.JndiLoginInitialContextFactory");
    try {
      InitialContext ctx = new InitialContext(props);
      Konto konto = (Konto) ctx.lookup("KontoBean/remote");
      konto.depositMoney(1000f);
      konto.withdrawMoney(1999f);
      // konto.calculateRate(1);
      System.out.printf("\nSaldo:\t%+9.2f CHF", konto
          .getAccountBalance());
      // System.out.printf("\nZins:\t%9.2f %%", konto.tellRate());
      konto.closeAccount();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Output **JBoss Server**

```
[STDOUT] Call to server.KontoBean.depositMoney took 6 Micros
[STDOUT] Withdraw Denied : Caller is customer_N
```

annotierte Security

```
/**
 * @module jee.2007.Metzler
 * @exercise jee123.server.KontoBean.java
 */
@DeclareRoles( { "verwalter", "kunde" })
@Interceptors(TimingInterceptor.class)
@Stateful
@Remote(Konto.class)
public class KontoBean implements Konto {
  ...
  @Resource
  private SessionContext                sc;
  @PermitAll
  public void depositMoney(float inAmount) {
    accountBalance += inAmount;
    modificationTime = System.currentTimeMillis();
    if (accountBalance < 0)
      contactReportMDB();
  }
  @RolesAllowed( { "verwalter", "kunde" })
  public void withdrawMoney(float inAmount) {
    if (inAmount < 1000f) {
      accountBalance -= inAmount;
    } else if (sc.isCallerInRole("verwalter")) {
      accountBalance -= inAmount;
    } else {
      System.out.println("Withdraw Denied : Caller is "
          + sc.getCallerPrincipal());
    }
    modificationTime = System.currentTimeMillis();
    if (accountBalance < 0)
      contactReportMDB();
  }
  @PermitAll
  @ExcludeClassInterceptors
  public float getAccountBalance() {
    return accountBalance;
  }
  @PermitAll
  public void calculateRate(int years) {
    accountBalance += zins.calculate(accountBalance, rate, years);
    modificationTime = System.currentTimeMillis();
  }
  @PermitAll
  @ExcludeClassInterceptors
  public float tellRate() {
    return rate;
  }
  @RolesAllowed("verwalter")
  @Remove
  public void closeAccount() {
   ...
```

13 Transaktionen

Transaktion ist eine logisch zusammenhängende Folge von Operationen einer Datenbank, die von einem konsistenten in einen weiteren konsistenten Zusand überführt.

- kleinste, unteilbare und daher an einem Stück ununterbrochen abzuarbeitende Prozesse einer Anwendung
- werden vollständig oder gar nicht abgearbeitet

13.1 Transaktionskonzepte

- ACID-Eigenschaften
  - **Atomic** (atomar)
    eine Transaktion ist Reihe von "primitiven" unteilbaren Operationen. Es werden entweder alle Operationen innerhalb einer Transaktion ausgeführt oder gar keine
  - **Consistent** (konsistent)
    die Änderungen einer Transaktion hinterlassen die Datenbank in einem konsistenten Zustand
  - **Isolated** (isoliert)
    gleichzeitige Transaktionen beeinflussen sich nicht.
  - **Durable** (dauerhaft)
    abgeschlosse Transaktionen sind dauerhaft

13.2 Transaktionsverhalten

- beenden einer Transaktion
  - commit
    erfolgreiche Beendigung einer Transaktion, die Änderungen sind dauerhaft in der Datenbank abgespeichert und für alle sichtbar
  - rollback
    der Abbruch einer Transaktion führt Änderungen am Datenbestand zum Zustand des letzten commit zurück

13.3 Lokale und verteilte Transaktionen

lokale Transaktion (local transactions)

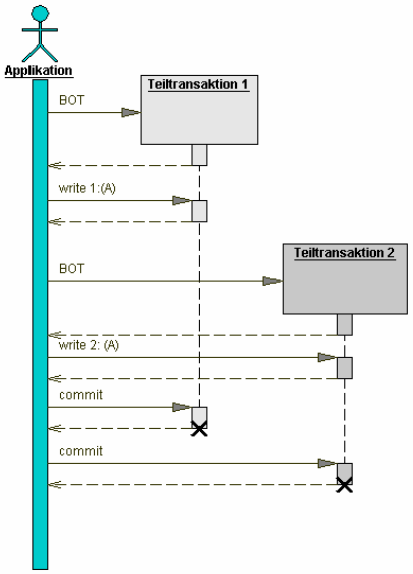
- nur lokale Daten betroffen
- nicht mehrere Ressourcen verwendet

verteilte Transaktion (distributed transaction)

- betreffen mehrere Daten, die auf verteilten Systemen liegen
- müssen in Teiltransaktionen aufgeteilt werden

13.4 Sequenzdiagramm

- die Applikation startet die **Teiltransaktion 1** (BOT = begin of transaction) und ändert den **Wert A**
- von der Teiltransaktion 1 erfolgt eine positive Rückmeldung
- die Applikation startet eine **zweite Teiltransaktion** auf einer zweiten Datenbank und ändert ebenfalls den **Wert A**
- erhält auch hier eine positive Rückmeldung
- der ersten Datenbank wird ein Commit übermittelt und beendet die Teiltransaktion 1
- der zweiten Datenbank wird ein Commit übermittelt und schliesst die gesamte Transaktion positiv ab

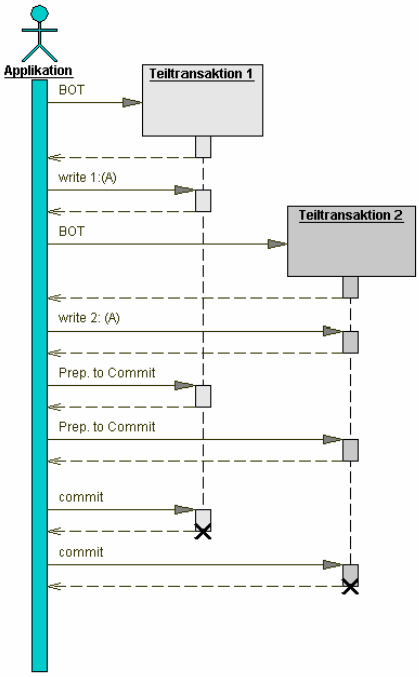


Zwei Phasen Commit

- inkonsistente Zustände sind möglich
  - zweite Commit-Anweisung fällt aus

Um solche Fehlerfälle abzufangen, bedarf es erweiteter Protokolle, z.B. des Zwei-Phasen-Commit-Protokolls (2PC-Protokoll):

- "Prepare-to-Commit" auf allen Knoten
- "Commit Phase"
  - zweite Phase sendet ein Commit an die Knoten



13.5 Isolationsebenen

- komplette Serialisierung der transaktionalen Operationen
  - sicher aber teuer
- Isolationsebene ist Einschränkung im Vergleich zu idealen ACID Transaktionen
  - (nach ANSI-92 SQL):

Isolationsebenen	
Level	Beschreibung
Read Uncommitted	Die Transaktion kann alle Daten lesen, unabhängig davon, ob Write Locks vorhanden sind. Zusätzlich wird die Transaktion in diesem Modus keine eigenen Read Locks absetzen.
Read Committed	Die Transaktion kann nur Daten lesen, welche bereits committed wurden, d.h. es muss gewartet werden, bis offene Write Locks abgeschlossen sind. Für das Lesen von Daten werden neue Read Locks abgesetzt, welche aber bereits nach Abschluss des SQL Statements (d.h. vor Ende der Transaktion!) wieder freigegeben werden.
Repeatable Read	Analog zu Read Committed mit der Ausnahme, dass die Read Locks bis ans Ende der Transaktion aufrecht erhalten werden.
Serializable	Analog zu Repeatable Read wobei zusätzlich sichergestellt wird, dass selbst bei mehrmaligem Ausführen von ein und demselben SQL Statement immer dieselben Resultate gesehen werden.

13.5.1 Probleme bei parallelen Datenbankoperationen

Multi-User Applikationen definieren folgende Problem-Szenarien:

Dirty Read

Innerhalb einer Transaktion (T1) wird ein Datensatz verändert. Dieser veränderte Datensatz wird innerhalb einer zweiten Transaktion (T2) gelesen, bevor T1 abgeschlossen wurde (Commit). Wird nun T1 mit einem Rollback abgebrochen, arbeiten die Operationen innerhalb von T2 mit einem ungültigen Wert.

Non Repeatable Read

Innerhalb einer Transaktion (T1) wird ein bestimmter Datensatz gelesen. Direkt nach dem Lesen von T1, aber noch vor einem Commit von T1, verändert eine zweite Transaktion (T2) diesen Datensatz und wird mit einem Commit beendet. Liest nun T1 diesen Datensatz erneut, wird ein anderer Inhalt zurückgegeben als zu Beginn, obwohl aus der Sicht von T1 der Datensatz nicht verändert wurde.

Phantom Read

Innerhalb einer Transaktion (T1) wird eine Abfrage an die Datenbank gestellt, welche eine bestimmte Anzahl von Ergebnisdatensätzen liefert. Eine zweite Transaktion (T2) ändert den

Inhalt der Datenbank, indem sie neue Datensätze einfügt und mit einem Commit beendet wird. Führt nun T1 die gleiche Abfrage erneut aus, so werden mehr Ergebnisdatensätze als beim ersten Mal gefunden.

mögliche Vorkommen bei Isolationsebenen:

mögliche Reads			
Isolationsebene	Dirty Read	Non Repeatable Read	Phantom Read
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

- mit zunehmendem Isolationsgrad
  - sicherere Transaktion
  - schlechtere Performance, da mehr Sperren innerhalb der Datenbank

13.6 Java Transaction API (JTA) und Java Transaction Service (JTS)

- Java Transaction Service (JTS) spezifiziert die Implementierung eines Transaktionsmanagers
- Java Transaction API (JTA) spezifiziert die Programmierschnittstelle
- jede transaktionsunterstützende (XA-kompatible) Ressource setzt ein Ressourcenmanager ein

Die Koordination von Transaktionen über die Grenzen eines Ressourcenmanagers hinaus wird durch den Transaktionsmanager realisiert und hat folgende Aufgaben:

- Initiierung von Start, Abbruch und Abschluss von Transaktionen.
- Über Transaktionsmanager erfolgt die Realisierung globaler, verteilter Transaktionen. Für eine globale Transaktion werden die beteiligten Ressourcen von dem Transaktions-manager verwaltet.
- Der erfolgreiche Abschluss globaler Transaktionen wird mittels des Zwei-Phasen-Commit-Protokolls koordiniert.
- Die Kooperation eines Transaktionsmanagers mit anderen Transaktionsmanagern kann über CORBA erfolgen.

Der Zugriff auf einen Java Transaction Service erfolgt mit der Java Transaction API, die aus drei Schnittstellen besteht:

javax.transaction.UserTransaction

Mit diesem Interface können Transaktionen gestartet `begin()`, abgebrochen `rollback()` oder abgeschlossen `commit()` werden. Zusätzlich kann man noch den Status einer Transaktion abfragen `getStatus()`, ein Timeout setzen `setTransactionTimeout()`

oder sicherstellen, dass mit einem Rollback `setRollbackOnly()` die Transaktion auf jeden Fall beendet wurde.

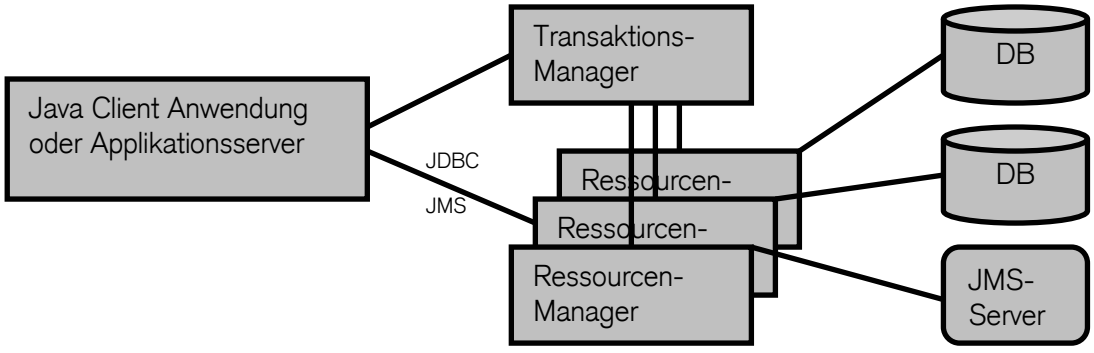
javax.transaction.TransactionManager

Ausser den Methoden der UserTransaction ermöglicht dieses Interface einem Applikationsserver noch zwei zusätzliche Methoden: Mit `suspend()` kann eine Transaktion vorübergehend angehalten und mit `resume()` wieder aufgenommen werden.

javax.transaction.Transaction und javax.transaction.xa.XAResource

Auch in der Transaktions-Schnittstelle wurden alle Methoden aus UserTransaction implementiert. Ausserdem besteht noch die Möglichkeit zum Registrieren `enlistResource()` und zum Deregistrieren einer Ressource `delistResource()` innerhalb der Transaktion; `enlistResource()` erhält als Parameter einerseits die Ressource selbst – als Objekt, das die XAResource-Schnittstelle implementiert – und andererseits einen als Objekt typisierten Parameter, der den Ressourcenmanager identifiziert und mit diesem interagiert, um das Zwei-Phasen-Commit durchzuführen. Die XARessource-Schnittstelle ist ein Java-Mapping des XA-Standards nach der X/Open-Spezifikation. Die wichtigsten Methoden dieser Schnittstelle sind:

- der Beginn der Ausführung mit der Zuordnung zu einer globalen Transaktion durch Aufruf von `start()`
- das Einholen der Entscheidung über ein Commit mittels `prepare()`
- das Beenden der Ausführung mit `end()`
- das Abschliessen / Zurücksetzen der Transaktion mit `commit() / rollback()`



13.7 Transaktionshandling

Um sicherzustellen, dass die Transaktionen einer Applikation den ACID-Prinzipien folgen, sind sorgfältiges Entwerfen und Implementieren nötig. Jede nachträgliche Änderung kann teuer werden. Um die Implementierung zu unterstützen, hat die EJB-Spezifikation zwei Typen von Transaktionsverwaltung vorgesehen:

- implizite (container-managed transactions) Transaktionssteuerung
- explizite (bean-managed transactions) Transaktionssteuerung

Das Element `<transaction-type>` im Deployment-Deskriptor legt fest, welche Transaktionsverwaltung verwendet werden soll. Der Wert ist entweder `Bean` oder `Container`.

13.7.1 Bean Managed Transaction

Mit Hilfe der BMT, ist es möglich, Transaktionen explizit zu steuern.

- Transaktionen mit anderen EJBeans
- Transaktionen mit Datenbanken über JDBC

Java Transaction API (JTA) stellt die Klasse `UserTransaction` zur Verfügung um Transaktionen zu verwalten:

- **`begin()`** – leitet eine neue Transaktion ein.
  Der Thread, der die neue Transaktion erzeugt hat, wird mit dieser verbunden. Falls innerhalb dieser Transaktion Beans verwendet werden, die bereits existierende Transaktionen unterstützen, wird die Transaktion an sie weitergereicht.
- **`commit()`** – beendet die aktuelle mit diesem Thread verbundene Transaktion.
- **`rollback()`** – Transaktion wieder zurückführen
  Um im Falle eines Fehlers die Transaktion wieder zurückzuführen und die schon durchgeführten Aktualisierungen rückgängig zu machen.
- **`setRollBackOnly()`** – merkt die Transaktion für das Zurückführen vor.
  Die Transaktion wird nach ihrer Beendigung in jedem Fall zurückgeführt, ob sie nun erfolgreich war oder nicht.
- **`setTransactionTimeout(int seconds)`** – Lebenszeit festlegen
  Mit dieser Methode wird festgelegt, wie lange die Transaktion existieren darf, bevor ein Timeout auftritt. Falls diese Methode nicht verwendet oder der Wert 0 übergeben wird, gilt der Default-Wert des Transaktions-Managers. Diese Methode muss unmittelbar nach `begin()` aufgerufen werden.
- **`getStatus()`** – Abfrage des aktuellen Transaktions-Status
  Liefert einen Integer-Wert zurück, der mit denen in der Schnittstelle javax.transaction.Status definierten Konstanten verglichen werden kann.

Um eine BMT verwenden zu können, wird das entsprechende <transaction-type> Tag mit Bean beschrieben.

```
<enterprise-beans>
  <session>
    <ejb-name>KontoBean</ejb-name>
    <transaction-type>Bean</transaction-type>
```

Dann wird eine Verbindung zum Transaktions-Manager des EJB-Server hergestellt und über den Container ein `UserTransaction`-Objekt geholt.

```
jndiContext = new InitialContext();
UserTransaction trans = (UserTransaction) jndiContext
 .lookup("java:comp/env/UserTransaction");
```

oder:

```
SessionContext ejbContext;
UserTransaction trans = sc.getUserTransaction();
  try {
    trans.begin();
     ...
    trans.commit();
```

oder:

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateful
public class KontoBean implements Konto {
  @Resource
  UserTransaction ut;
   ...
```

Annotation @TransactionManagement			
Parameter	Beschreibung	Konstante	Default
TransactionManagementType	Transaktionsmanagement	CONTAINER BEAN	CONTAINER

13.7.2 Container-Managed-Transactions

- die Steuerung der Transaktionen wird dem Container überlassen
- kein zusätzlicher Code in die Geschäftslogik
- try-catch-Blocks definiert Transaktion
  - Fehler (catch) führt alle schon durchgeführten Anweisungen zurück (ROLLBACK)
  - keine verschachtelten Transaktionen möglich

CMT auf Bean Ebene

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@RolesAllowed( { "verwalter", "kunde" })
public void withdrawMoney(float inAmount) {
```

Annotation @TransactionAttribute			
Parameter	Beschreibung	Konstante	Default
TransactionAttributeType	Transaktionsattribut	REQUIRED REQUIRESNEW MANDATORY SUPPORTS NOTSUPPORTED NEVER	REQUIRED

13.8 Transaktions-Attribute

- **Required:** Die Bean soll immer in einer Transaktion laufen.
  Falls schon eine Transaktion läuft, wird die Methode ein Teil von ihr, andernfalls wird eine neue Transaktion vom Container gestartet. Die neue Transaktion umfasst allerdings nur die Required-Bean und die von ihr benutzten Beans. Wenn die Required-Methode beendet ist, wird auch die Transaktion beendet.
- **RequiresNew:** Es wird immer eine neue Transaktion gestartet.
  Wenn der aufrufende Client bzw. Bean selbst zu einer Transaktion gehört, wird diese ausgesetzt, bis die "RequiresNew"-Methode beendet ist.
- **Supports:** Die Methode wird zur Transaktion des Aufrufers einbezogen.
  Jedoch nur wenn dieser eine Transaktion hat. Falls der Aufrufer zu keiner Transaktion gehört, wird auch keine neue Transaktion vom Container erzeugt.
- **Mandatory:** Diese Bean-Methode gehört immer zur Transaktion des aufrufenden Clients. Das entspricht dem Gegenteil des `Never`-Attribut. Falls der aufrufende Client mit keiner Transaktion verbunden ist, tritt eine Exception auf.
- **NotSupported:** Methoden können keine Transaktionen unterstützen oder erzeugen.
  Falls der aufrufende Client Teil einer Transaktion ist, wird diese Transaktion während der Ausführung der Methode ausgesetzt (unterbrochen).
- **Never:** Die Methode darf nicht in eine Transaktion einbezogen werden.
  Falls ein Client innerhalb einer Transaktion diese Methode aufrufen sollte, tritt eine Exception auf. Wenn man also vermeiden möchte, dass die Clients, die diese Methode aufrufen, Transaktionen verwenden, dann kann man das mit diesem Attribut erreichen.

Allerdings lassen sich nicht alle Transaktions-Attribute mit allen Beans nutzen. Die folgende Tabelle zeigt, welche Kombinationen möglich sind:

Transaktions-Attribut			
	SLSB	SFSB	MDB
Required	✓	✓	✓
RequiresNew	✓	✓	✗
Mandatory	✓	✓	✗
Supports	✓	✗	✗
NotSupported	✓	✗	✓
Never	✓	✗	✗

13.9 Applikation Exception

- Unterscheidung von Exceptions
  - System Exception : Ausnahme der technischen Infrastruktur
  - Applikation Exception : Ausnahme der Geschäfslogik
    - checked : Subklassen von java.lang.Exception
    - unchecked : Subklassen von java.lang.RuntimeException
- Applikation Exception führen nicht automatisch zu einem transaktionalen Rollback
- ExceptionKlasse muss mit @ApplicationException annotiert wrden

Annotation @ApplicationException			
Parameter	Beschreibung	Typ	Default
rollback	Rollbackverhalten bei einer Exception	Boolean	false

- Als Alternative kann EJBContext.setRollbackOnly() aufgerufen werden

```
/**
 * @module jee.2007.Metzler
 * @exercise jee131.server.IllegalOperationException.java
 */
@ApplicationException(rollback = true)
@SuppressWarnings("serial")
public class IllegalOperationException extends Exception {
  public IllegalOperationException(String reason) {
    System.out.println(reason);
  }
}
```

13.10 JDBC Transaktionen

Die JDBC-Transaktionen sind Transaktionen, die vom Transaktions-Manager des Datenbank-managementsystems (DBMS) verwaltet werden. Mit Hilfe der `java.sql.Connection`-Schnittstelle ist es möglich, Datenbank-Transaktionen selbst zu steuern. Der standardmässig eingeschaltete AutoCommit-Modus sollte vorher mit `setAutoCommit(false)` ausgeschaltet werden. Der Beginn einer Transaktion erfolgt implizit mit der ersten SQL-Anweisung. Es können dann weitere SQL-Anweisungen folgen, bis schliesslich die gesamte Transaktion mit `commit()` bestätigt oder mit `rollback()` zurückgenommen wird.

Beispiel

```
con.setAutoCommit(false);
stmt.executeUpdate("INSERT INTO Table VALUES (val1, val2, ...);");
if (error) {
 con.rollback();
 } else {
 con.commit();
}
```

13.11 Transaktionen der Persistence Unit

verschiedene Anwendungsszenarien (Patterns) für transaktionale Operationen

- basieren auf der Verwendung von
  - JDBC Connections
  - Java Transaction API (JTA)
- optimistischen Locking
  - beim Schreiben wird geprüft ob sich die Daten verändert haben
  - implementiert durch die Verwendung von Versionsattributen @Version
- pessimistisches Locking
  - lesender Client sperrt Daten
  - implementiert durch API die SELECT FOR UPDATE der Datenbank umsetzt
- Repeatable-Read-Semantik definert Caching innerhalb Hibernate Session
  - dabei muss jede Datenbankoperation innerhalb einer Transaktion laufen
  - auch reine Leseoperationen
  - aktiver Auto-Commit-Modus wird durch Hibernate deaktiviert
    - sollte generell nicht in Verbindung mit Hibernate eingesetzt werden

Verwendung von Transaktionen für managed Entity Beans unterscheidet:

- managed : innerhalb eines Java EE Application Servers
  - BMT (Bean Managed Transactions)
  - CMT (Container Managed Transactions)
- non-managed : in einer Java-SE-Anwendung

13.12 Optimistisches Locking

"optimistisches" Locking bedeutet: "meistens geht alles gut"

- hohe Performance
- gute Skalierbarkeit
- keine Überschneidungen von Änderungen durch parallele Transaktionen
  - Auftreten einer Überschneidung wird durch aufwändige, manuelle Korrektur behoben
    - z.B. durch Versionierung der Datensätze
  - oder auch Last Commit Wins
    - Verwirrungen seitens der User

- manuelle Überprüfung mittels Versionsvergleich für jede Entity
- Versionsattribut der Entities wird automatisch beim Commit hochgezählt
- manueller Ansatz nur bei kleinen, trivialen Persistenzlösungen praktikabel
  - manuelle Überprüfung eines komplexen Objektgraphen nicht einfach

13.12.1 Versionsprüfung durch Version Number

- Entity Manager prüft beim Speichern auf Zustandsänderungen der Daten
- Hibernate als Persistenceprivider bietet automatische Versionsüberprüfung
- Attribut version darf durch Applikation nicht verändert werden
- version wird bei jerder Veränderung hochgezählt

```
@Entity
public class MyEntity {
  @Id
  private Long id;

  @Version
  @Column(name="Versionfeld")
  private int version;
  ...

}
```

13.13 Pessimistisches Locking

häufige Kollisionen zwischen konkurrierenden Transaktionen

- restriktives Vorgehen verlangt
- Tabellenzeilen werden explizit gesperrt
  - konkurrierende Transaktion erhält Fehlermeldung
  - konkurrierende Transaktion muss warten
- verschlechtert Gesamtperformance des Systems aufgrund wartenden Transaktionen

Locking-Mechanismus

- um Deadlocks innerhalb der Datenbank zu vermeiden
- Hibernate nutzt für pessimistisches Locking die Datenbank Funktionalität
- Hibernate setzt nie Locks auf Entity-Objekte im Speicher!
- Klasse LockMode definiert verschiedene Lock-Modi
  - es stehen nicht immer alle Modi zur Verfügung (Datenbankabhängig)
  - falls gewünschter Lock-Modus nicht unterstützt wird wählt Hibernate automatisch einen verwandten, verfügbaren Modus

Folgende Lock-Modi werden in LockMode definiert:

LockMode	
Parameter	Beschreibung
NONE	Es wird kein Lock auf Zeilen in der Datenbank gesetzt. Mit diesem Modus wird beim Lesen eines Datensatzes nur auf die Datenbank zugegriffen, wenn sich die entsprechende Entity nicht im Cache befindet.

LockMode	
Parameter	Beschreibung
READ	Dieser Lockmodus weist Hibernate an, direkt auf die Datenbank zuzugreifen und eine Versionsüberprüfung der betroffenen Entities durchzuführen. Sinnvoll z. B. bei Verwendung von Detached Entities. Dieser Lockmodus wird von Hibernate automatisch verwendet, wenn als Isolationsebene Repeatable Read oder Serializable ausgewählt wurde.
UPGRADE	Wird für pessimistisches Locking verwendet. Es wird mit SELECT ... FOR UPDATE ein Lock auf die Tabellenzeilen gesetzt, wenn die Datenbank dieses Feature unterstützt.
UPGRADE_NOWAIT	Verhält sich wie UPGRADE. Durch ein SELECT ... FOR UPDATE NOWAIT, welches in Oracle Datenbanken verfügbar ist, wird die DB angewiesen, nicht darauf zu warten, falls die selektierte Tabellenzeile bereits gelockt ist, sondern eine Locking Exception zu werfen.
WRITE	Ein "interner" Modus, der automatisch von Hibernate verwendet wird, wenn ein Datensatz aktualisiert oder eingefügt wird. Dieser Modus kann nicht explizit durch den User verwendet werden.

- Lock-Modi erlauben pessimistisches Locking auf feingranularer Ebene

13.14 Übung : Kontotransfer als Transaktion

- Übertrag von Konto zu Konto
- negative Saldi durch Transaktion verhindern

```
/**
 * @module jee.2007.Metzler
 * @exercise jee131.server.BankBean.java
 */
@TransactionManagement(TransactionManagementType.CONTAINER)
@Stateless
@Remote(Bank.class)
public class BankBean implements Bank {
  @EJB
  private Konto      konto;
  @Resource
  private EJBContext sc;
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void transfer(Konto from, Konto to, float amount)
      throws IllegalOperationException {
    try {
      to.depositMoney(amount);
      from.withdrawMoney(amount);
    } catch (IllegalOperationException e) {
      sc.setRollbackOnly();
    }
  }
```

BankClient

```
final float amount = 10.0f;
InitialContext ctx = new InitialContext();
final Bank bank = (Bank) ctx.lookup("BankBean/remote");
final Konto from = bank.getAccount("Savings");
final Konto to = bank.getAccount("Checking");
printAccount(from);
printAccount(to);
Thread t1 = new Thread() {
 public void run() {
  try {
   System.out.println(
    "Thread started : transfer Savings -> Checking : " + amount);
   bank.transfer(from, to, amount);
   System.out.println("Thread finished");
  } catch (Exception e) {
    e.printStackTrace();
  }
 }
};
t1.start();
Thread.sleep(1000);
printAccount(from);
printAccount(to);
from.closeAccount("Savings");
to.closeAccount("Checking");
```

Output:

```
Login successfull
   OWNER : Savings  BALANCE : 100.0
   OWNER : Checking  BALANCE : 200.0
Thread started : transfer Savings -> Checking : 10.0
Thread finished
   OWNER : Savings  BALANCE : 90.0
   OWNER : Checking  BALANCE : 210.0
```

13.14.1 Transaktionsverhalten

Testen und überprüfen Sie anhand der Methoden-Transaktionsattribute das transaktionale Verhalten der Applikation:

Klasse	BankBean	KontoBean		Transaktions-Verhalten
Methoden	transfere()	depositMoney () withdrawMoney ()	getAccountBalance () getDescription ()	
Transaktionsattribut	Required	Required	Required	✓
	Required	RequiresNew	Required	✘
	Supports	Required	Supports	✘
	Required	Supports	NotSupported	✘
	Required	Supports	Required	✓
	NotSupported	Required	Required	✘
	NotSupported	NotSupported	NotSupported	✘

C JPA Java Persistence API

14 Entity Beans

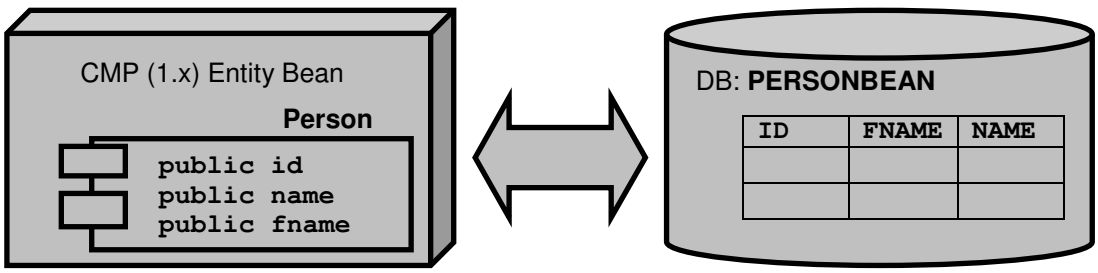
Abbilden von persistenten Geschäftsdaten in die Logik Tier

- Entity Bean (bis Version 2.1) als EJB
  - Mapping durch Deploymentdeskriptoren
- POJO als Mapper zwischen OO und Relationaler Schicht (EJB 3.0)
  - Mappingverhalten per Annotationen
  - durch Deploymentdeskriptoren (überschreiben Annotationen)

14.1 Entity Bean als EJB

- modelliert Abbildungen von persistenten Geschäftsdaten (DB)
- mehrere Clients können auf diese Entities zugreifen
- verwendet identifizierbare Daten (primary key)
- CMP - Container Managed Persistence:
  - Container speichert Bean in DB
  - Bean-Provider muss sich nicht kümmern
  - exakter Mechanismus ist nicht definiert !
  - jedes Objekt hat einen Primary Key
- BMP - Bean Managed Persistence:
  - Bean-Provider ist verantwortlich, dass die Daten in der DB gespeichert werden
  - Realisation mittels SQL, ...
  - höherer Aufwand als CMP
  - mehr Flexibilität als CMP

14.1.1 CMP



Deployment Descriptor ejb-jar.xml (Ausschnitt)

```
<enterprise-beans>
  <entity>
    <ejb-name>PersonBean</ejb-name>
    <home>ejb.PersonHome</home>
    <remote>ejb.Person</remote>
    <ejb-class>ejb.PersonBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>false</reentrant>
```

```
      <cmp-version>1.x</cmp-version>
      <cmp-field>
        <field-name>fname</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>id</field-name>
      </cmp-field>
      <primkey-field>id</primkey-field>
    </entity>
  </enterprise-beans>
```

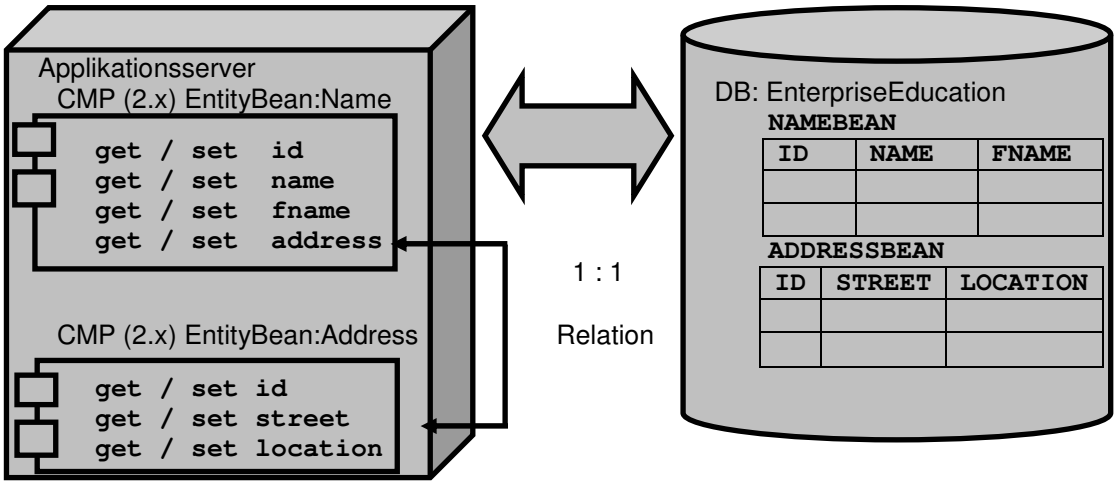
Deployment Descriptor cmp-mappings.xml (Ausschnitt)

```
<entity-mapping>
  <ejb-name>PersonBean</ejb-name>
  <table-name>PERSONBEAN</table-name>
  <cmp-field-mapping>
    <field-name>fname</field-name>
    <column-name>PERSONBEAN.FNAME</column-name>
  </cmp-field-mapping>
  <cmp-field-mapping>
    <field-name>id</field-name>
    <column-name>PERSONBEAN.ID</column-name>
  </cmp-field-mapping>
  <cmp-field-mapping>
    <field-name>name</field-name>
    <column-name>PERSONBEAN.NAME</column-name>
  </cmp-field-mapping>
</entity-mapping>
```

14.1.2 CMR

Enterprise JavaBeans 2.0 erweitern die frühere Version 1.x mit folgenden Eigenschaften:

- erweiterte Funktionalität für Container-Managed Persistence für Entity Beans
- unterstützt CMR (Container-Managed Relationships)
- definiert EJB-Query Language (EJB-QL) für **`select`** und **`find`** Query Methoden
- unterstützt zusätzlich Locale Remote and Locale Home Interfaces um den Zugriff zwischen Beans im gleichen Container zu ermöglichen



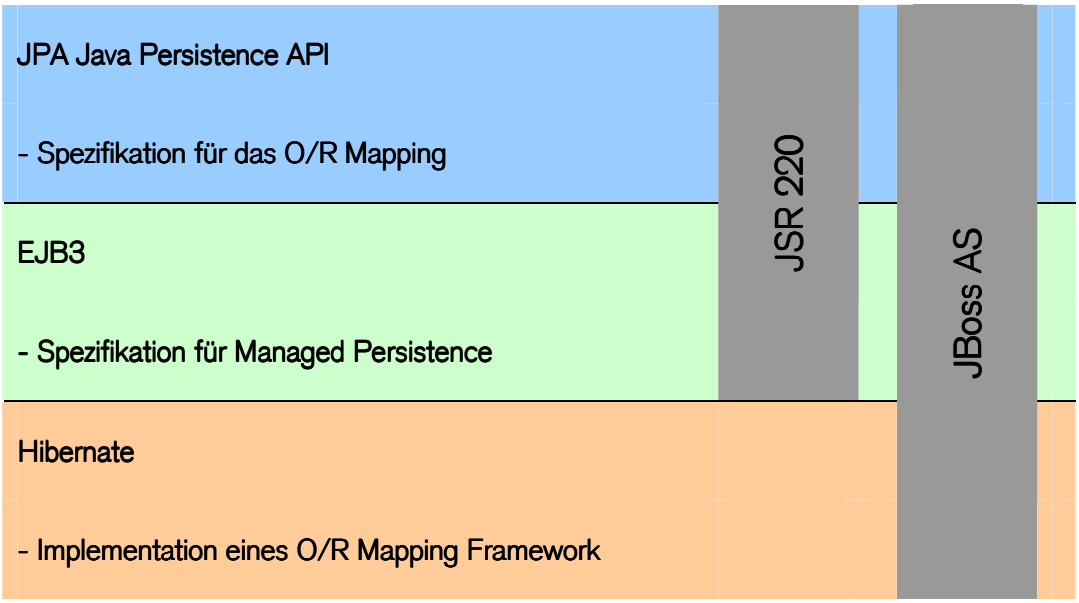
14.2 Entity Bean als POJO

- seit EJB 3.0 normale Java Klassen (POJOs) als persistente Geschäftsobjekte
  - mit BusinessKey (Primärschlüssel)
  - mit Annotationen
  - weder Klasse noch Methoden dürfen final sein
  - darf keine eingebettete Klasse sein
  - darf kein Interface sein
  - darf keine Enum sein
  - muss Standard-Konstruktor besitzen (öffentlich und parameterlos), ev. durch Compiler bereitgestellt
  - persistente Attribute als public deklariert und durch Getter/Setter proklariert

POJO als EntityBeab

```
@Entity
public class Person implements Serializable {
  @Id
  private Long        id;
  private String      name;
  @ManyToMany(targetEntity = Address.class)
  private Collection address;
  public Person() {
  } // Default Konstruktor
  public Collection<Address> getAddress() {
    return address;
  }
  public void setAddress(Collection<Address> address) {
    this.address = address;
  }
  public Long getId() { return id; }
  public void setId(Long id) { this.id = id; }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
}
```

15 Begriffserklärungen



15.1 Java Persistence API

- spezifiziert durch JSR 220
- spezifiziert Persistenz Framework
  - (Definition) aus Hibernate, TopLink und JDO entstanden
- nicht auf den Einsatz unter Java EE begrenzt
  - lauffähig unter Java SE
  - als normale Java Anwendungen
  - ausserhalb eines Java EE Containers
- Darstellung der Entitäten durch POJOs (Plain old Java Objects)
- Definition der Metadaten durch Annotations
  - XML (Deployment) Deskriptor-Dateien zur Angabe der Metadaten als Alternative
  - XML (Deployment) Deskriptor-Dateien überschreiben Annotationen

Ziel

- Java EE zu vereinfachen.

15.2 EJB 3.0

- Spezifikation für Managed Persistence
- Spezifikation standardisiert Java Persistence API
- JSR 220 durch Java Community Process Programm: http://jcp.org/en/jsr/detail?id=220

Ziel

- Standardisierung der Basis-API
- Standardisierung von Metadaten eines objekt-relationalen Persistenz-Mechanismus
- definiert aber kein fertiges Framework, sondern ist Spezifikation

15.3 Hibernate

- Implementation eines O/R Mapping Framework
- Gavin King ist Gründer von Hibernate
- Hibernate Core
  - Definition mit XML
- Hibernate Annotation
  - Definition per Annotations
- Hibernate Entity Manager
  - Zugriff auf den Persistenzkontext

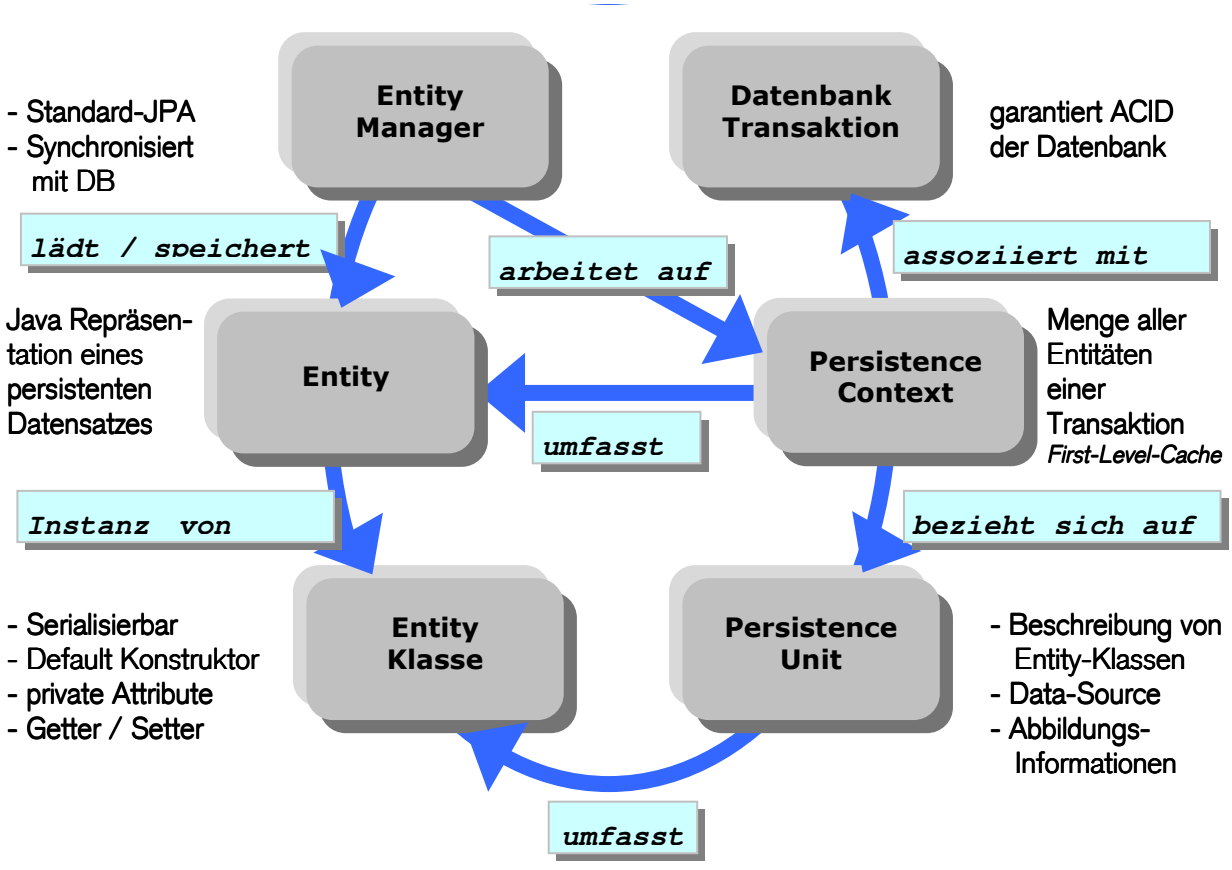
Hibernate ist als Open-Source ein objekt-relationaler Persistenz- und Query-Service:

> Hibernate is a powerful, high performance object/relational persistence and query service. Hibernate lets you develop persistent classes following object-oriented idiom- including association, inheritance, polymorphism, composition, and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API.
>
> *http://www.hibernate.org*

16 Persistence Provider

JPA erfordert einen Persistence Provider:

- Entity
  - Java Repräsentation eines persistenten Datensatzes
- Entity Klasse
  - serialisierbar / default Konstruktor /private Attribute
  - getter / setter Methoden
- Persistence Unit
  - Beschreibung von Entity-Klassen
  - Data-Source
  - Abbildungsinformationen
- Entity Manager
  - JPA Standard Interface
  - Synchronisiert mit der Datenbank
- Persistenzkontext
  - Menge aller Entitäten einer Transaktion
  - First Level Cache
- Datenbank-Transaktion
  - garantiert ACID Verhalten der Datenbank



Persistenz-Provider implementiert JPA

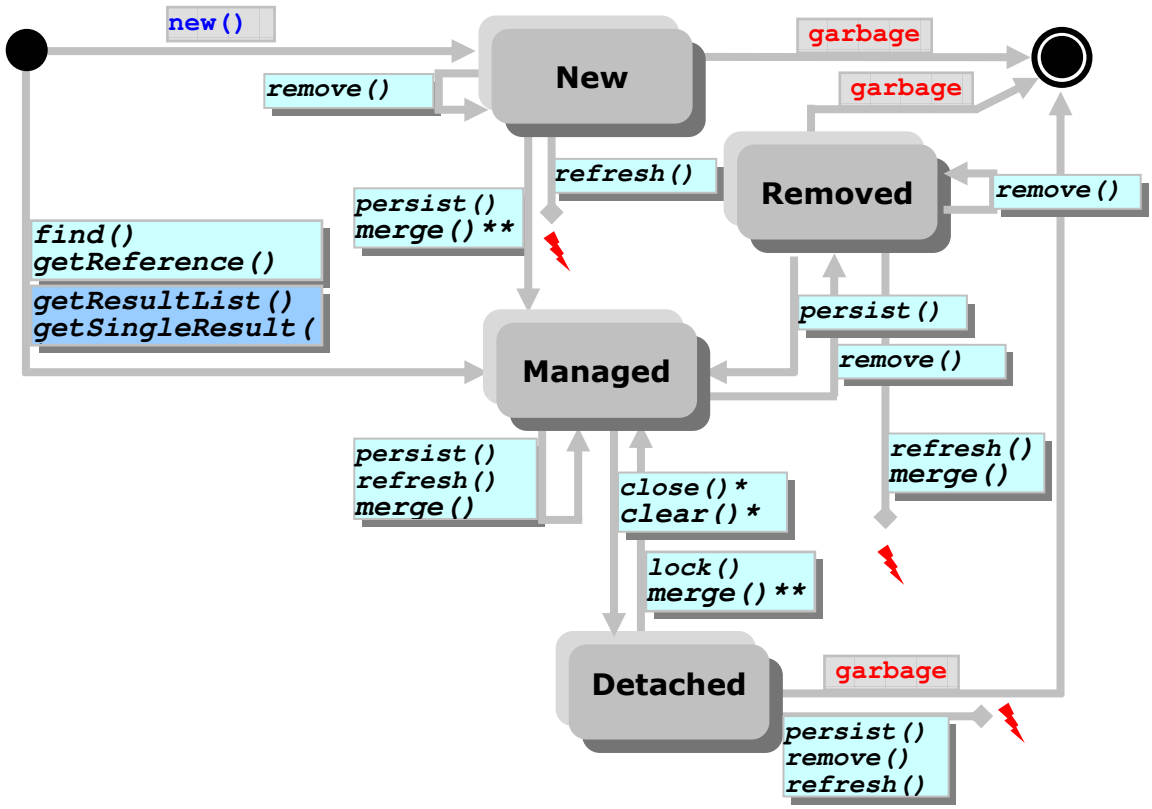
16.1 Lebenszyklus

- der Persistence Provider verwaltet die Zustände eines persistenten Objektes transparent
  - Hibernate ist transparentes O/R-Mapping Framework
  - Entities ist ihr eigener Zustand nicht bekannt
  - es ist nicht direkt möglich, eine Entity nach ihrem aktuellen Zustand zu fragen

16.1.1 Objektzustände

Eine Entity in Hibernate definiert vier verschiedene Zustände:

- Transient / Removed
- Persistent
- Detached



Objekt - Zustandsdiagramm

- Zustandsänderung durch Operationen des Persistenzkontext
  - ☐ Methoden der EntityManager-Instanz
  - ☐ Methoden der Query-Instanz

* betrifft alle Instanzen des Persistenzkontext

** erzeugt persistente Instanz, Originalobjekt bleibt erhalten

New

- durch new erzeugtes Entity-Objekt
- keine Verbindung zwischen der Entity und Entity Manager
- Zustand der Entity nicht persistent abgespeichert
- normales Java-Objekt POJO
- Garbage Collector entsorgt Objekte ohne Referenz
- keine Repräsentation der Entity innerhalb der Datenbank
- alle Daten der Entity sind flüchtig und gehen verloren (Garbage Collector)
- Entities sind transient, nicht Teil einer Transaktion
- Rollback nicht anwendbar (in Transaktion)
- Primärschlüsselfelder, von Hibernate erzeugt, sind noch nicht gesetzt

Übergang von New zu Managed mit:

- persist()
- merge()
- Referenzierung durch persistente Entity (abhängig von Kaskadierungseinstellungen der Assoziation)

Manged

- Entity ist einer Hibernate Session zugeordnet
- Entity hat Primärschlüssel zugewiesen
- (auch ohne) Repräsentation innerhalb der Datenbank
- persist() veranlasst keine Datenbankaufrufe !
- Entities im persistenten Zustand sind Teil einer Transaktion
- Änderungen können durch Rollback rückgängig gemacht werden
- Änderung von Attributen wird erkannt
  - UPDATE innerhalb der Datenbank
  - INSERT innerhalb der Datenbank
- find() – erzeugt persistentes Entity
  - ohne Übergang von New nach Managed

Übergang von Managed zu Removed:

- remove()

Achtung: remove() löscht die Daten der Entity innerhalb der Datenbank, Java-Objekt bleibt verfügbar, vorausgesetzt es wird noch referenziert

Detached

- schliessen des Managers mit close(...)
- Zuordnung der Entities zu Session endet
- Entities sind nicht mehr Teil einer Transaktion
- Änderungen werden nicht mehr mit Datenbank synchronisiert
- Java-Objekte enthalten trotzdem (veraltete) persistente Daten
- Serialisieren (und Übertragen) einer Entity in einen anderen Prozess
  - Transferieren des POJO in eine Remote-Anwendung, (Client-Request)
- Transfer-Objekte zwischen Präsentationsschicht und Datenbankschicht

Übergang von Detached zu Managed:

- public Object **merge**(Object object):
  - Zustand der übergebenen Entity wird in Persistenzkontext übertragen
  - Kopie der Entity wird in Persistenzkontext aufgenommen
  - existiert Primärschlüssel in Persistenzkontext werden Daten der Entity kopiert
  - falls keine Entity in Persistenzkontext ist:
    - neue Entity wird aus der Datenbank geladen und die Daten kopiert
    - neue Entity wird erstellt und die Daten kopiert
  - gibt Referenz auf die persistente Entity zurück

- public void **lock**(Object object, LockMode lockMode):
  - bindet Detached Entity wieder an Persistenzkontext
  - Detached Entity darf nicht verändert worden sein
  - Parameter lockMode definiert verschiedene Lock-Modi, welche in Verbindung mit Transaktionen eine Rolle spielen

16.2 Bestandteile eines O/R Mapping Frameworks

16.2.1 Entities

- leichtgewichtige, persistente Objekte
- zentraler Teil der Java Persistence API
- lösen die schwergewichtigen Entity Beans ab
- abstrakte wie auch konkrete Java-Klassen werden verwendet
- Vererbung, polymorphe Assoziationen und polymorphe Abfragen sind unterstützt
- nicht von einer bestimmten Basisklasse abgeleitet
- eigene Vererbungshierarchie kann verwendet werden

Entities und Java-Klassen können beliebig kombiniert werden; folgende Bedingungen müssen erfüllt sein:

- Entity-Klasse muss mit der Annotation @Entity markiert werden
- Entity-Klasse muss einen parameterlosen Konstruktor enthalten (public oder protected)
- Entity-Klasse muss eine Top-Level-Klasse sein, darf nicht als final deklariert sein
- Methoden oder persistente Attribute der Klasse dürfen nicht final sein
- Das Interface Serializable muss implementiert werden, falls Instanzen der Klasse "by-value" übertragen werden sollen
- Jede Entity muss einen Primärschlüssel enthalten

Innerhalb der Entity-Klassen können die persistenten Attribute durch Annotations mit Mapping- und anderen Metadaten versehen werden. Zwei Methoden stehen zur Verfügung:

- Direkte Markierung der Klassenattribute
- Markierung der Getter-Methoden

Die Markierung persistenter Attribute muss innerhalb einer Klasse identisch sein. Es dürfen also nicht beide Arten gleichzeitig innerhalb derselben Klasse verwendet werden. Ausserdem können keine Attribute als persistent markiert werden, die mit dem Javaschlüsselwort transient deklariert wurden. Als persistente Attribute einer Entity sind folgende Typen erlaubt:

- alle primitiven Javatypen und deren Wrapperklassen
- java.lang.String
- serialisierbare Javaklassen (z. B. java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp)
- selbstdefinierte serialisierbare Javaklassen
- byte[], Byte[], char[] und Character[]
- Enums
- java.util.Collection, java.util.Set, java.util.List, java.util.Map
- Collections von Entity-Klassen
- Embeddable Klassen

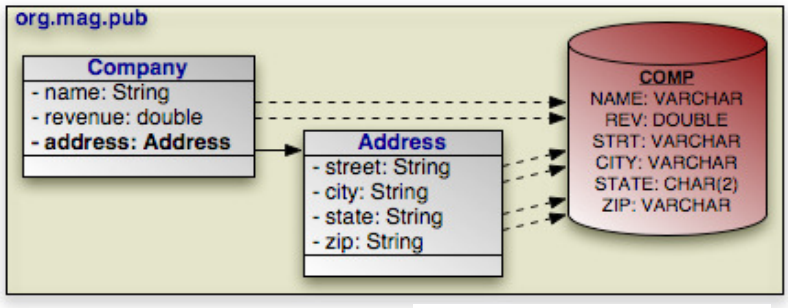
16.2.2 Primärschlüssel

- mittels der Annotation @Id gekennzeichnet
- einfaches Attribut der Klasse
- alle primitiven Javatypen
- java.lang.String
- java.util.Date
- java.sql.Date
- Primärschlüsselklasse
  - die Klasse muss das Serializable Interface implementieren
  - die Methoden equals und hashcode müssen definiert werden
  - die Klasse muss public sein und einen parameterlosen Konstruktor enthalten
- zusammengesetzter Schlüssel
  - Instanzvariable vom Typ der Primärschlüsselklasse wird definiert (Embeddable Klasse)
  - Entity-Klasse enthält einzelne Felder des Primärschlüssels

Innerhalb einer Vererbungshierarchie von Entities darf nur ein Primärschlüssel definiert werden. Ausserdem muss er in der obersten Klasse der Hierarchie deklariert sein.

16.2.3 Embeddable Klassen

- dieselben Bedingungen wie die Entity-Klasse
- durch Annotation @Embeddable markiert (nicht @Entity)
- keinen Primärschlüssel, da ihre persistenten Attribute auf die gleiche Tabellenzeile gemappt werden wie die der sie umschliessenden Entity
- gehören direkt zur Entity und können nicht von mehreren Entity Objekten referenziert werden



http://openjpa.apache.org

16.2.4 Relationship

Die Java Persistence API unterstützt Beziehungen (Relationships) zwischen Entities:

- 1 : 1 durch die Annotations @OneToOne abgebildet
- 1 : n durch die Annotations @ OneToMany abgebildet
- n : 1 durch die Annotations @ ManyToOne abgebildet
- n : m durch die Annotations @ ManyToMany abgebildet

Defaultmappings erlauben in den meisten Fällen eine Beziehung zwischen zwei Entity-Klassen zu definieren.

Beispiel einer ManyToMany Beziehung Person-Address

```
@Entity
public class Person implements Serializable {

   @Id
   private Long id;
   private String name;

   @ManyToMany(targetEntity=Address.class)
   private Collection address;

   public Person() {
   } // Default Konstruktor

   public Collection<Address> getAddress() {
      return address;
   }

   public void setAddress(Collection<Address> address) {
      this.address = address;
   }
```

```
@Entity
public class Address {
   @ManyToMany(mappedBy = "address")
   Collection<Person> personen;
}
```

16.2.5 bidirektionale Beziehungen

- unterscheiden zwischen "Besitzer" und der referenzierten Seite
- referenzierte Seite muss auf ihren Besitzer durch Angabe mappedBy der Relationship Annotation @OneToOne, @OneToMany oder @ManyToMany verweisen.
  - bei 1 : 1- Beziehungen enthält die Besitzerseite den Fremdschlüssel in der Datenbank.
  - "n"-Seite einer bidirektionalen Beziehung muss die "Besitzer"-Seite sein.
  - bei n : m-Beziehungen kann die Besitzerseite frei gewählt werden.

einfache 1:n Relation

```
@Entity
public class Book {
   private Publisher publisher;

   @ManyToOne
   public Publisher getPublisher() {
      return publisher;
   }
...
}
```

```
@Entity
public class Publisher {
   private Collection<Book> books = new HashSet();

   @OneToMany(mappedBy = "book")
   public Collection<Book> getBooks() {
      return books;
   }
...
}
```

16.2.6 Vererbung

- Polymorphe Assoziationen sind Beziehungen auf verschiedene konkrete Klassen innerhalb einer Vererbungshierarchie
- Polymorphe Abfragen liefern verschiedene konkrete Klassen einer Vererbungshierarchie zurück
- abstrakte Klassen können mit @Entity definiert werden
- abstrakte Klassen können nicht direkt instanziiert werden
- Abfragen können sich auf abstrakte Entities beziehen, liefern Instanzen von konkreten Subklassen als Ergebnis

abstrakte Entity-Klasse

```
@Entity
@Table(name = "BOOK")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Book {
   @Id
   private Long id;
   private String title;
}
```

```
@Entity
@DiscriminatorValue(value = "Paperback")
public class Paperback extends Book {
   protected Integer priceDiscount;

   public Integer getPriceDiscount() {
      return priceDiscount;
   }
}
```

- Klasse Book ist als abstract deklariert
- Klasse Paperback ist eine konkrete Entity-Klasse, die von Book abgeleitet ist
- Klasse Paperback erbt alle Attribute von Book wie id und title

Eine Entity kann auch von einer Klasse abgeleitet sein, die selbst nicht als @Entity definiert ist und @MappedSuperclass und entsprechende Mapping-Annotations enthält.

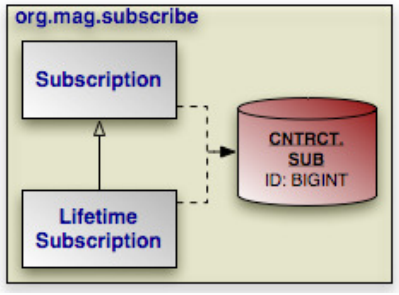
- wird nicht direkt einer Tabelle zugeordnet
- Attribute werden auf die Tabellen der abgeleiteten Entity-Klassen gemappt
- kann nicht direkt mittels einer Abfrage gesucht werden
- Verwendung in einer Entity-Relationship ist nicht direkt möglich

Das Ableiten von einer solchen nicht-Entity-Klasse ist z. B. dann sinnvoll, wenn mehrere Entities die gleichen Attribute enthalten sollen, denn die gemappten Attribute werden vererbt und sind Teil jeder Subklasse und werden dementsprechend beim Mapping aus der Datenbank berücksichtigt.

Entity-Klassen können auch von Klassen abgeleitet werden, die weder als @Entity noch als @MappedSuperclass markiert sind.

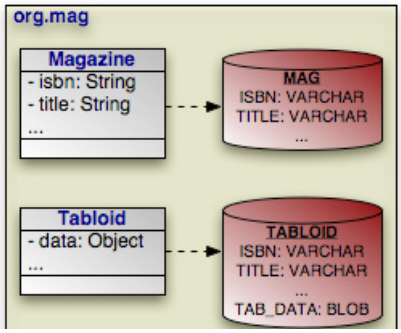
- werden nicht von einem Entity Manager verwaltet
- werden nicht persistent abgespeichert

Für das Mapping der Entity-Hierarchien auf die konkreten Tabellen einer Datenbank stehen folgende Strategien zur Verfügung:



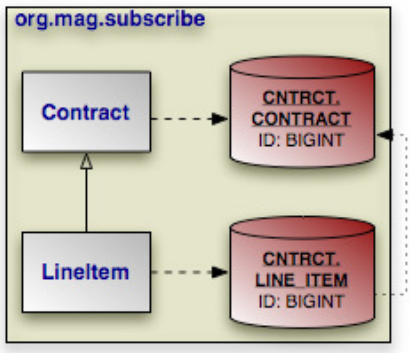
SINGLE_TABLE

- eine Tabelle für die Abbildung einer Hierarchie
- Unterscheidungsmerkmal ist die Diskriminator Spalte
- Polymorphe Abfragen sind möglich



TABLE_PER_CLASS

- Eine Tabelle pro konkreter Entity-Klasse
- Unterstützung durch Implementierung optional
- jede Tabelle enthält ALLE Attribute einer Klasse inklusive der geerbten
- keine polymorphen Abfragen möglich
- SQL UNION als Behelf



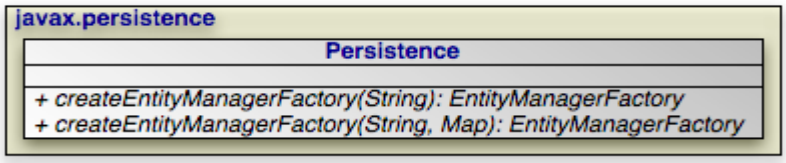
JOINED

- eine Tabelle je Superklasse und eine Tabelle je abgeleitete Entity-Klasse
- eine Tabelle je Klasse einer Vererbungshierarchie
- jede Tabelle enthält nur spezifische Attribute der jeweiligen Klasse
- polymorphe Abfragen sind möglich (nicht sehr performant)

http://openjpa.apache.org

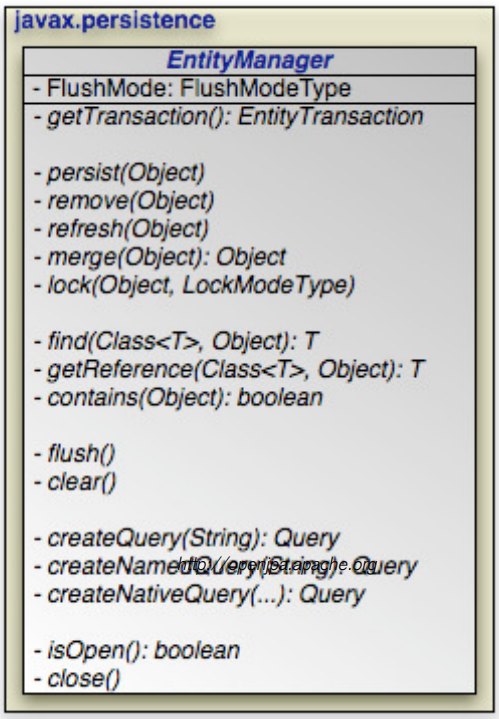
16.3 Persistenzkontext

- erzeugt EntityManager Factory
- definiert durch persistence.xml



http://openjpa.apache.org

16.4 EntityManager



- verwaltet die Entities
- stellt die API für den Zugriff auf einen Persistenzkontext bereit
- Persistenzkontext bezeichnet Menge von Entity-Objekten die innerhalb der Datenbank höchstens ein Java-Objekt im Kontext representieren
- einem Entity Manager ist immer genau ein Persistenzkontext zugeordnet, als:
  - Container-Managed Entity Manager
  - Application-Managed Entity Manager

Container-Managed Entity Manager

- kommen innerhalb eines Java EE Containers zum Einsatz
- für das Erzeugen und Schliessen des Entity Managers verantwortlich
- der Java EE Container verwaltet JTA (Java Transaction API) Transaktionen
- Verwaltung geschieht transparent für die Anwendung
- Zugriff auf den Entity Manager erhält die Anwendung durch:
  - Dependency Injection
  - JNDI
- die Angabe der Annotation @PersistenceContext ist notwendig

Application-Managed Entity Manager

- Entity Manager wird von der Anwendung selbst verwaltet
- zum Erzeugen wird eine EntityManagerFactory verwendet
- Transaktionen durch:
  - JTA
  - lokale Transaktion (z. B. JDBC Transaktion)

Für die Gültigkeitsdauer eines Persistenzkontexts gibt es zwei Alternativen:

- transaction-scoped:
  - entspricht der Dauer einer Transaktion
  - nach Abschluss der Transaktion wird der Persistenzkontext geschlossen
- extended:
  - ist unabhängig von einer Transaktion
  - kann mehrere Transaktionen umfassen
  - der Persistenzkontext muss manuell (Entity Manager) geschlossen werden

Methoden der EntityManager API (Ausschnitt):

```
public void persist(Object entity);
```

Übergang zu managed Entity Instanz; commit() oder flush() persistiert Objekt in der Datensource.

- ist entity eine neue Entity wird diese zur managed Entity
- ist entity eine bestehende managed Entity so wird nichts geschehen
- ist entity eine removed Entity wird diese zur managed Entity
- ist entity eine detached Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.PERSIST annotiert sind

```
public void remove(Object entity);
```

Übergang zu removed Entity Instanz; commit() oder flush() löscht Objekt in der Datensource.

- ist entity eine neue Entity so wird nichts geschehen
- ist entity eine bestehende managed Entity so wird diese gelöscht
- ist entity eine removed Entity so wird nichts geschehen
- ist entity eine detached Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.REMOVE annotiert sind

```
public void refresh(Object entity);
```

Synchronisiert die Entity Instanz mit der Datensource. Änderungen innerhalb der Entity werden dabei überschrieben.

- ist entity eine neue Entity so wird nichts geschehen
- ist entity eine bestehende managed Entity so wird diese mit der Datensource synchronisiert
- ist entity eine removed Entity so wird nichts geschehen

- ist entity eine detached Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.REFRESH annotiert sind

```
public void merge(Object entity);
```

Übergang von einer detached Entity zu einer managed Entity Instanz.

- ist entity eine detached Entity so wird diese in die existierende managed Instanz von entity kopiert oder eine neue Kopie einer managed entity Instanz erstellt
- ist entity eine neue Entity so wird eine neue managed Instanz entity* als Kopie erstellt
- ist entity eine bestehende managed Entity so wird nichts geschehen
- ist entity eine removed Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.MERGE annotiert sind

```
public void lock(Object entity, LockModeType mode);
```

Sperrt die Entity Instanz:

- LockModeType.READ
  - andere Transaktionen können das Objekt lesen aber nicht ändern
- LockModeType.WRITE
  - andere Transaktionen können das Objekt weder lesen noch schreiben

```
public <T> T find(Class<T> entity, Object primaryKey);
```

Sucht ein Objekt:

- lädt die entity aus der Datenbank, welche dem Primärschlüssel zugeordnet ist
- liefert null, falls keine Entity gefunden wurde

```
public <T> T getReference(Class<T> entity, Object primaryKey);
```

Lädt ein Proxy (bzw. eine Referenz) einer Entity, welche durch den angegebenen Primärschlüssel eindeutig identifiziert wird

- übrige Felder der Entity können zu einem späteren Zeitpunkt nachgeladen werden
- Vorgehen ist sinnvoll, wenn eine Entity einer anderen zugeordnet werden soll und der eigentliche Inhalt gar nicht benötigt wird

```
public boolean contains(Object entity);
```

Prüft ob sich die Entity im Persistenzkontext des Entity Managers befindet.

```
public void flush();
```

Synchronisiert die Datensource mit den Entity Objekten.

- weist den EntityManager an, den Zustand aller Entites die dem Persistenzkontext zugeordnet sind, mit der Datenbank zu synchronisieren
- Synchronisation geschieht dabei nur in eine Richtung; der Inhalt der Entity-Objekte im Persistenzkontext wird nicht aktualisiert, falls sich der Inhalt der entsprechenden Datenbankfelder geändert haben sollte.
- Synchronisation wird nur ausgeführt, wenn eine Transaktion aktiv ist

```
public void clear();
```

Leert den Persistenzkontext des Entity Managers.

- Persistenzkontext des Entity Managers wird geleert, alle verwalteten Entities werden detached Entities
- Synchronisation mit der Datenbank findet nicht statt

Verwendung des EntityManagers

```
// emf = EntityManagerFactory ist Application Managed
// Neuen EntityManager erstellen
   EntityManager em = emf.createEntityManager();

// Transaktion starten
   em.getTransaction().begin();
   Book myBook = em.find(Book.class, new Long(1));
   myBook.setTitle("Hibernate 3.0");
   Book mySecondBook = new Book();
   mySecondBook.setId(2);
   mySecondBook.setTitle("Hibernate 3.2");
   em.persist(mySecondBook);
   em.getTransaction().commit();
// Ende der Transaktion
}
```

- neuer EntityManager mit Hilfe der EntityManagerFactory wird erzeugt
- neue Transaktion wird begonnen
- find(...)-Methode lädt eine Instanz aus der Datenbank

- setTitle(...) weist dieser Entity ein neuer Titel zu
- Änderungen sind aber noch nicht persistent in der Datenbank abgelegt
- neue Book-Instanz wird erzeugt und deren ID und Title gesetzt
- mit em.persist(...) definiert diese neue Entity zu einer verwalteten Entity
- mit em.getTransaction().commit() werden alle Änderungen in der Datenbank abgespeichert

Die Ausführung ist ausserhalb eines Java EE Containers ausgelegt, da die Erzeugung eines EntityManager sowie das manuelle Starten und Beenden einer Transaktion innerhalb eines Containers von diesem übernommen würde.

Exception des EntityManagers		Methoden des EntityMangers									
-Exception	Beschreibung	persist	merge	remove	find	getReference	flush	refresh	lock	contains	joinTransction
Persistence-	Superklasse, wird geworfen wenn Operation nicht ausgeführt werden kann Transaktionsstatus wird (meistens) auf RollbackOnly gesetzt						✓		✓		
EntityExists-	PK existiert bereits	✓				✓					
EntityNotFound-	Entity existiert nicht (mehr)										
IllegalArgument-	Objekt ist keine Entity	✓	✓	✓	✓	✓		✓	✓	✓	
NoResult-	Query.getSingleResult() liefert kein Ergebnis Transaktion wird nicht zurückgerollt										
NoUniqueResult-	Query.getSingleResult() liefert mehr als ein Ergebnis Transaktion wird nicht zurückgerollt										
OptimisticLock-	DB Inhalt wurde geändert										
Rollback-	EntityTransaction.Rollback kann nicht durchgeführt werden										
Transaction-Required-	Operation wird nicht innerhalb einer Transaction aufgerufen (bei container manged Entity Manager)	✓	✓	✓			✓	✓	✓		✓

16.5 Entity Listener

Die neue Java Persistenz API bietet die Möglichkeit, über bestimmte Ereignisse während des Lebenszyklus einer Entity informiert zu werden. Hierfür sind zwei Mechanismen definiert:

deklarieren von Callback-Methoden

- Methoden werden unmittelbar nach oder vor dem jeweiligen Ereignis automatisch aufgerufen
- eignet sich um den Persistenzmechanismus durch beliebige eigene Funktionalitäten zu erweitern

Folgende Annotations stehen zur Verfügung:

- @PrePersist
  - wird vor der Ausführung der persist Operation des EntityManagers aufgerufen
- @PreRemove
  - wird vor der Ausführung von remove aufgerufen
- @PostPersist
  - wird nach der Ausführung von persist aufgerufen
  - wird auch ausgeführt, nachdem die jeweiligen INSERT Statements innerhalb der Datenbank durchgeführt wurden
- @PostRemove
  - wird nach der Ausführung von remove aufgerufen
- @PreUpdate
  - wird aufgerufen, bevor ein UPDATE innerhalb der Datenbank ausgeführt wird
- @PostUpdate
  - wird nach einem UPDATE-Statement ausgeführt
- @PostLoad
  - wird aufgerufen, nachdem eine Entity in den aktuellen Persistenzkontext geladen wurde sowie nach jedem refresh der Entities

zuweisen einer Entity-Listener-Klasse

- mit Annotation @EntityListeners
- Methoden werden automatisch aufgerufen

aufzurufende Methoden müssen folgende Bedingungen erfüllen:

- Callback-Methode muss die folgende Signatur haben

```
void <METHODENNAME>()
```

- Methode einer Entity-Listener-Klasse muss die folgende Signatur haben

```
void <METHODENNAME> (<KLASSE> o)
```

- Typ des Parameters o muss der Klasse der Entity entsprechen, auf die sich die Entity-Listener- Klasse bezieht.

Definitionen für die Verwendung von Callback Methoden

- mehrere Methoden mit verschiedenen Callback Annotations können markiert werden
- mehrfache Markierung einer Methode durch verschiedene Callback Annotations ist zulässig
- mehrere Methoden können nicht mit der gleichen Annotation versehen werden, d. h., es kann immer nur eine Methode pro Ereignis definiert werden
- aber mehrere Listener pro Entity können definiert werden, Reihenfolge der Aufrufe hängt von der Reihenfolge der Definition innerhalb der @EntityListener Annotation ab
- gleichzeitige Verwendung von Entity Listener und Callback-Methoden gemeinsam in einer Entity ist möglich, Methoden der Entity Listener werden vor den Callback-Methoden aufgerufen

Verwendung von Callback-Methoden und Entity Listenern

```
@Entity
public class Book1 {
   @PostPersist
   protected void postPersistBook() { ... }
}

@Entity
@EntityListeners(
   PaperbackListener.class, PaperbackListener2.class)
public class Paperback extends Book { ... }

public class PaperbackListener {
   @PostPersist
   protected void postPersistMethod1(Paperback book) {...}
}

public class PaperbackListener2 {
   @PostPersist
   protected void postPersistMethod2(Paperback book) {...}
}
```

Das Beispiel zeigt die Verwendung von Callback-Methoden und Entity Listenern innerhalb einer Entity-Vererbungshierarchie. Wird ein Paperback mittels persist() in der Datenbank abgespeichert, werden die Listenermethoden in der folgenden Reihenfolge ausgeführt:

- postPersistMethod1(...)
- postPersistMethod2(...)
- postPersistBook()

16.6 Queries

- können in SQL formuliert werden
- werden durch EJB Query Language (EJB QL) definiert (erweiterte Features)
  - Massen-Updates und –Deletes
  - Joins
  - Group By
  - Subqueries
- EJB QL ist portable, da Datenbank unabhängig (gegenüber SQL)

Definition und Verwendung von Queries:

```
public Query createQuery(String ejbqlString)
```

- erzeugt ein Query-Objekt, mit dem die Abfrage ausgeführt werden kann
- übergebene Query-String muss als EJB-QL-Abfrage formuliert sein

```
public Query createNamedQuery(String name)
```

- erzeugt ein Query-Objekt aus einer so genannten Named Query
- vordefinierte Abfrage in EJB QL oder SQL, welche in den Metadaten (XML Dateien oder Annotations) hinterlegt wurde

```
public Query createNativeQuery(String sqlString)
```

- erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String

```
public Query createNativeQuery(String sqlString, Class resultClass)
```

- erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String
- Class Parameter resultClass bestimmt den Typ der Ergebnisse, welche von dieser Query zurückgeliefert werden

```
public Query createNativeQuery(String sqlString, String resultSetMapping)
```

- erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String
- Mapping definiert die JDBC Ergebnisse auf die jeweiligen Entities zu mappen

Das Query Interface enthält eine Reihe von Methoden zur Angabe von Parametern sowie zur Durchführung der durch das Query-Objekt repräsentierten Abfrage. Queries können auch als sogenannte "Named Queries" statisch innerhalb der XMLMetadaten oder als Annotation definiert werden:

```
@NamedQuery(
   name="findAllBooksWithTitle",
   query="SELECT b FROM Book b WHERE b.title LIKE :bookTitle"
)
```

Eine so definierte Query kann folgendermassen verwendet werden:

```
EntityManager em = ...
List books = em.createNamedQuery("findAllBooksWithTitle")
   .setParameter("bookTitle", "Hibernate").getResultList();
```

- Queries sind polymorph:
  - Abfrage liefert nicht nur die Instanzen der Entity-Klasse, sondern auch Instanzen von abgeleiteten Entity-Klassen, wenn diese die Abfragebedingungen erfüllen

- Queries in SQL formuliert:
  - wenn EJB QL nicht ausreicht
  - datenbankspezifische SQL Features
  - nicht mehr datenbankunabhängig
  - Ergebnis einer SQL formulierten Abfrage sind skalierbar

  - SqlResultSetMapping deklariert die Javaklassen der Entities:

```
Query q = em.createNativeQuery(
  "SELECT b.id, b.title, b.publisher, p.id, p.name" +
  "FROM Book b, Publisher p " +
  "WHERE (b.title LIKE 'Hibernate%') " +
  "AND (b.publisher = p.id)" +
  ,"BookPublisherResults"
);

@SqlResultSetMapping(name="BookPublisherResults",
  entities={
     @EntityResult(entityClass=Book.class),
     @EntityResult(entityClass=Publisher.class)
  }
)
```

17 Hibernate als Persistenzframework

- implementiert die standardisierten Annotations
- deklariert Bedingungen durch Validators
- stellt mit dem EntityManager eine Persistenzengine zur Verfügung

17.1 Hibernate Annotations

- Verwendung der Hibernate Annotations (neu)
- Metadaten im XML-Format
- als XDoclet-Kommentare

Die Hibernate Annotations fügen eine weitere Möglichkeit hinzu, um für Hibernate relevante Metadaten zu deklarieren, welche sich auch mit den bisherigen Lösungen mischen lassen.

Hibernate-spezifische Annotations (zusammen mit EJB Annotations):

- @org.hibernate.annotations.Entity
  erlaubt Einstellungen, die über den Umfang der EJB3-Spezifikation hinausgehen. Ist kein Ersatz der Annotation @javax.persistence.Entity, sondern kann zusätzlich angegeben werden.
  - mutable definiert ob die Entity veränderbar ist
- selectBeforeUpdate
  definiert, dass Hibernate nie ein SQL UPDATE ausführt, ohne zu überprüfen, ob sich eine Entity tatsächlich verändert hat
- optimisticLock
  erlaubt die Festlegung der Lock Strategie:
  - OptimisticLockType.VERSION
  - OptimisticLockType.NONE
  - OptimisticLockType.DIRTY
  - OptimisticLockType.ALL
- @org.hibernate.annotations.GenericGenerator
  ermöglicht die Angabe eines Hibernate ID Generators
  mit der EJB3 Annotation @Id lassen sich so alle in Hibernate verfügbaren ID Generierungsmechanismen nutzen
- @org.hibernate.annotations.AccessType
  legt die Art, wie die Persistenz-Engine auf Attribute einer Entity zugreift, bis auf Attribute-Ebene individuell fest
  z. B. kann ein bestimmtes Attribut definiert werden, auf das nur über die Getter- und Setter-Methoden zugegriffen wird, während auf die übrigen Attribute innerhalb derselben Entity direkt, also auf die Instanzvariablen, zugegriffen wird

Die hier vorgestellten Annotations sind nur ein kurzer Auszug aus den Erweiterungen, die das Hibernate Annotations Projekt bereithält. So sind unter anderen auch zusätzliche Annotations für den Umgang mit Polymorphie, Assoziations, Collections und Queries enthalten.

17.2 Hibernate Validator

- ermöglicht Deklaration von Bedingungen, die eine gültige Entity erfüllen muss
- Constraints, werden als Annotations direkt in der betroffenen Entity deklariert, wobei sowohl Methoden wie auch Instanzvariablen markiert werden können
- entsprechende Validator Implementierung führt die tatsächliche Überprüfung aus
- Deklaration zentral, Überprüfung innerhalb einer mehrschichtigen Anwendung
- Constraints werden direkt über die Entity- Objekte überprüft, z. B. in Form von Entity Listeners, die vor einem Insert oder Update aufgerufen werden.

Die Hibernate Validator enthaltenen Constraints decken ein breites Spektrum ab:

- @Length: Überprüfung von Stringlängen
- @Max: Maximalwert für numerische Werte
- @Min: Minimalwert für numerische Werte
- @NotNull: Überprüfung auf NULL
- @Past: Überprüft, ob ein Datum in der Vergangenheit liegt
- @Future: Überprüft, ob ein Datum in der Zukunft liegt
- @Pattern: Überprüft, ob ein Attribut einem regulären Ausdruck entspricht
- @Range: Wertebereich für numerische Werte
- @Size: Überprüfung der Länge von Collections, Arrays und Maps
- @AssertFalse: Überprüft boolean Werte auf false
- @AssertTrue: Überprüft boolean Werte auf true
- @Valid: Führt eine Überprüfung des markierten Objekts durch

Definition eines neuen Constraints:

- Definition einer Annotation
- Implementierung einer Validatorklasse

Hibernate Validator zur Überprüfung einer Book-Entity

```
@Entity
public class Book implements Serializable {

   @Id
   private Long id;

   @Length(max=100)
   @NotNull
   private String title;
   ...
}
```

Der Titel eines Buches darf maximal 100 Zeichen lang sein und darf nicht null sein.

Definition von Constraints

```
ClassValidator bookValidator = new ClassValidator( Book.class );
InvalidValue[] validationMessages = addressValidator.getInvalidValues(book);
```

Für jede verletzte Bedingung liefert ein so genannter ClassValidator ein InvalidValue- Objekt, welches Details über die nicht erfüllte Bedingung enthält. Die ClassValidator Instanz sollte normalerweise nur einmal pro Anwendung erzeugt und danach gecached werden.

17.3 Hibernate EntityManager

- implementiert durch die Java Persistence API
- Einsatz innerhalb eines Java EE Containers
- Einsatz als Standalone Lösung ausserhalb eines Containers

Hibernate EntityManager setzt auf den Hibernate Core auf und verwendet diesen als Persistenzengine.

18 Kontoverwaltung als Anwendungsbeispiel

- Inhaber verwalten
- Konto anlegen
- Suchfunktionen

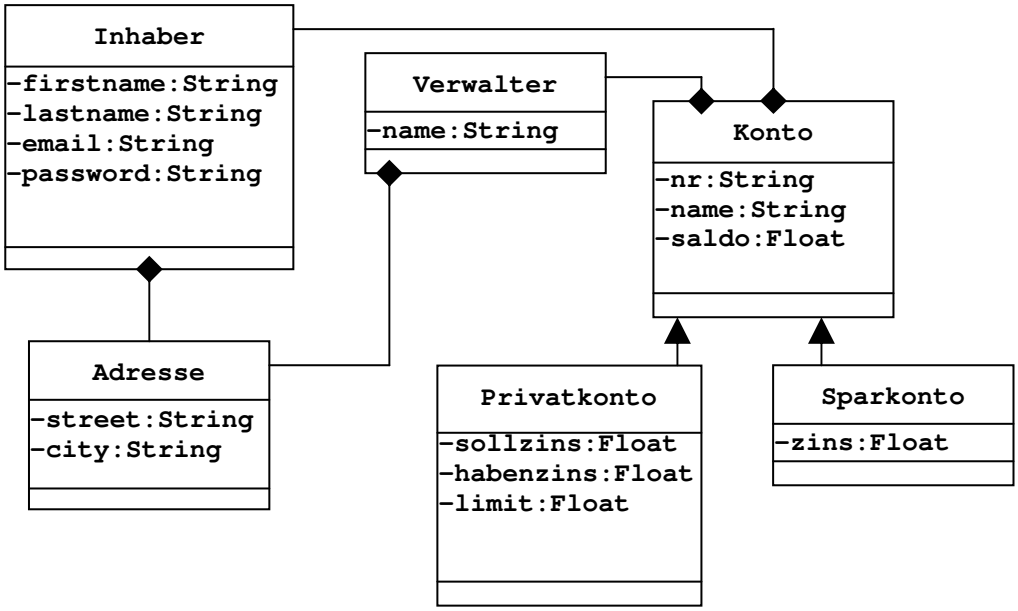
18.1 Anwendungsfälle

- Grundlegende Funktionen der Anwendung:
  - Registrierung eines Inhabers
  - Anmelden eines Inhabers
  - Erfassen eines Kontos
  - Suche nach Inhaber- / Kontoinformationen

18.2 Klassendiagramm

- Klassen bzw. Entities:
  - Inhaber mit Adresse (Email-Adresse als eindeutiger Identifikator)
  - Konto (eindeutige Nummer) als Privatkonto oder Sparkonto
  - Verwalter

Klassendiagramm



18.3 Konfiguration

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="JavaEE">
    <jta-data-source>java:jdbc/jee</jta-data-source>
    <properties>
      <property  name="hibernate.hbm2dll.auto"
                 value="create-drop" />
    </properties>
  </persistence-unit>
</persistence>
```

definiert

- Persistenz Unit
- Data Source
- Properties

18.4 POJO als Entity

```
/**
 * @module jee.2007.Metzler
 * @exercise jee181.server.Inhaber.java
 */
@Entity
public class Inhaber {
  @Id
  private Long   id;
  private String firstname;
  private String lastname;
  private String email;
  private Inhaber() {
  }
  public Inhaber(String firstname, String lastname, String email)
{
    this.firstname = firstname;
    this.lastname = lastname;
    this.email = email;
  }
  public Long getId() {
    return id;
  }
  private void setId(Long id) {
    this.id = id;
  }
  public String getFirstname() {
    return firstname;
```

```
  }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }
  public String getLastname() {
    return lastname;
  }
  public void setLastname(String lastname) {
    this.lastname = lastname;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
}
```

einfaches POJO (Plain Old Java Object):

- Annotations der Java Persistence API
- ohne externe Abhängigkeiten
- Getter- und Setter-Methoden
- Java- Beans-Nameskonventionen
- id als eindeutigen Bezeichner für persistente Klasse
- Standardkonstruktor (auch private)
- @Entity vor der Klassendefinition
- @Id vor dem Getter für id (Primärschlüssel)
- @Transient für nicht persistente Felder

- @Table definiert das Datenbankschema

Annotation @Table			
Parameter	**Beschreibung**	**Typ**	**Default**
name	Tabellenname	String	Klassenname
schema	Tabellenschema	String	datenbank-spezifisches Schema
catalog	Tabellenkatalog	String	datenbank-spezifischer Katalog
uniqueConstraint	Unique-Constraints, die auf der Tabelle definiert werden sollen	UniqueConstraint[]	

Entity persistieren

Klasse erweitern mit:

- @Table (auf Klassenebene)
- @NamedQueries
  - für optionale Suchfunktionalität

Inhaber.java

```
/**
 * @module jee.2007.Metzler
 * @exercise jee181.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
@NamedQuery(name = "Inhaber.findAll", query = "from Inhaber i")
public class Inhaber implements Serializable {
  @Id
  private Long   id;
  private String firstname;
  private String lastname;
  private String email;
```

BankBean als Fassade

```
/**
 * @module jee.2007.Metzler
 * @exercise jee181.server.BankBean.java
 */
@Stateful
@Remote(Bank.class)
public class BankBean implements Bank {
  @PersistenceContext(unitName = "JavaEE")
  private EntityManager manager;
  public void addInhaber(Inhaber i) throws InhaberException {
    if (manager.find(Inhaber.class, i.getId()) != null) {
      throw new InhaberException("Inhaber bereits vorhanden");
    }
    manager.persist(i);
  }
...
```

BankClient

```
/**
 * @module jee.2007.Metzler
 * @exercise jee181.client.BankClient.java
 */
public class BankClient {
  public static void main(String[] args) {
    try {
```

```
        InitialContext ctx = new InitialContext();
        Bank bank = (Bank) ctx.lookup("BankBean/remote");
        Inhaber i1 = new Inhaber(1L, "my_First_1", "my_Last_1",
            "my_Email_1@email.com");
        bank.addInhaber(i1);
...
```

t_inhaber

id [BIGINT]	firstname [VARCHAR(255)]	lastname [VARCHAR(255)]	email [VARCHAR(255)]
1	my_First_1	my_Last_1	my_Email_1@email....
2	my_First_2	my_Last_2	my_Email_2@email....
3	my_First_3	my_Last_3	my_Email_3@email....

Insert-Statement wird ausgeführt durch :

- manager.persist()
- als Alternative :
  - manager.merge()

19 Generatorstrategien

- @Id definiert Primary Key Attribut
- @GeneratedValue definiert Strategie zur Generierung des Primärschlüssels

persistente Datenbankobjekte haben Datenbankidentität:

- zwei gleiche Objekte besitzen denselben Primärschlüssel
- Verwalten der Primärschlüssel übernimmt Persistenz-Unit
  - oder definieren des Primärschlüssels (natürlicher Schlüssel, z.B. Personalnummer)

19.1 Anforderungen an Primärschlüssel

- darf nicht null sein
- kann nie verändert werden (natürlicher Schlüssel ?)
- ist in einer Tabelle über alle Einträge eindeutig

19.2 Datenbankidentität, Objektidentität, Objektgleichheit

- Objektidentität (JAVA : ==) Referenz-Semantic
  - zwei Objekte sind identisch, wenn sie dieselbe Adresse im Speicher der Java VM (Virtual Maschine) haben
- Objektgleichheit (JAVA : equals()) Value-Semantic
  - zwei Objekte sind gleich, wenn sie denselben Inhalt wie in der Methode equals() definiert haben. Falls equals() aus der Klasse java.lang.Object nicht überschrieben wurde, wird die Objektidentität zurückgegeben

Praxis

- equals() und hashcode() immer überschreiben
  - equals() soll geschäftsrelevante, eindeutige Daten vergleichen, z.B. Email
  - hascode() soll gute Verteilung definieren (mehr, aber echte Daten)

Problematisch bei

- Objekte in Collections (Sets, Maps, etc.)
  - da Implementation durch java.util.HashSet
- einbinden von Detached Objekten
  - bestehendes Objekt wird mit neuer Session verlinkt

(mögliche) Implementierung von equals() und hashcode() für die Klasse Inhaber:

```
@Override
public boolean equals(Object obj) {
   if (this == obj) return true;
   if (!(obj instanceof Inhaber)) return false;
   final Inhaber inhaber = (Inhaber) obj;
   return !(email != null ? !email.equals(inhaber.email)
        : inhaber.email != null);
}
@Override
public int hashCode() {
   return 29 * (email != null ? email.hashCode() : 0);
}
```

Primärschlüssel ist für Objektgleichheit nicht geeignet:

- Inhaber hat n Adressen
- zwei Adressen (Rechnungsadresse und Lieferadresse) werden hinzugefügt (über Collections!)
- Objektgleichheit (über Primärschlüssel bestimmt) überschreibt die erste Adresse beim Hinzufügen der zweiten Adresse
  - Primary Key wird generiert durch die Datenbank
  - Primary Key ist NULL solange das Objekt nicht persistiert ist
- beide Adressen sind transient und Primärschlüssel null

```
   // erste Adresse hinzufügen
   inhaber.addAddress(new Address("Street 1", "City 1"));
   // zweite Adresse hinzufügen
   inhaber.addAddress(new Address("Street 2", "City 2"));
...
   // erzeugt nur eine Adresse für den Inhaber
   session.update(inhaber);
```

19.3 Generators für die Primärschlüssel

- @Id definiert Feld als Primärschlüssel
- @GeneratedValue wählt ein Id-Generator
- @SequenceGenerator definiert die Sequezparameter
- @TableGenerator nutz seperate Datenbanktabelle

Annotation @GeneratedValue			
Parameter	**Beschreibung**	**Parameter**	**Default**
strategy	Strategie zum definieren des Primary Keys	AUTO IDENTITY SEQUENCE TABLE	

Generator-Strategien

- AUTO
  - wählt entsprechend der darunterliegenden Datenbank eine Strategie (TABLE, IDENTIY, SEQUENCE).
  - Entspricht der Hibernate-Generatorstrategie native.
- TABLE
  - IDs sind in einer eigenen Tabelle.
  - Entspricht der Hibernate-Generatorstrategie hilo. (Hi/Lo Algorithmus)
- IDENTITY
  - unterstützt Identity Columns, die es beispielsweise in MySQL, HSQLDB, DB2 und MS SQL Server gibt.
  - Entspricht der Hibernate-Generatorstrategie identity.
- SEQUENCE
  - unterstützt Sequences, die es beispielsweise in PostgreSQL, Oracle und Firebird gibt.
  - Entspricht der Hibernate-Generatorstrategie sequence.

Annotation @SequenceGenerator			
Parameter	**Beschreibung**	**Typ**	**Default**
name	eindeutiger Name innerhalb der Persistence-Unit kann von anderen Entites referenziert werden	String	
sequenceName	Datenbank abhängiges Sequenz-Objekt	String	abhängig vom Persistence-Provider

Annotation @SequenceGenerator			
Parameter	Beschreibung	Typ	Default
initalValue	Anfangswert für die Generierung	int	0 oder 1
allocationSize	Schrittweite	int	50

Annotation @TableGenerator			
Parameter	Beschreibung	Typ	Default
name	eindeutiger Name innerhalb der Persistence-Unit kann von anderen Entities referenziert werden	String	
table	Tabellenname	String	abhängig vom Persistence-Provider
schema	Tabellenschema	String	datenbank-spezifisches Schema
catalog	Tabellenkatalog	String	datenbank-spezifischer Katalog
pkColumnName	Spalte für den Namen der Generatoren	String	abhängig vom Persistence-Provider
valueColumnName	Spalte für den Zählerstand des PK	String	abhängig vom Persistence-Provider
pkColumnValue	Generatorname	String	abhängig vom Persistence-Provider
initialValue	Anfangswert für die Generierung	int	0 oder 1
allocationSize	Schrittweite	int	50
uniqueConstraints	Definition von Constraints	Unique-Constraint[]	

19.4 proprietäre Stategien

Hibernate bietet Hibernate-eigene Annotation (@org.hibernate.annotations.GenericGenerator) für die Definition weiterer Generatoren an.

```
@Id
@GeneratedValue(generator = "useridgen")
@GenericGenerator( name = "useridgen",
                   strategy = "seqhilo",
                   parameters = {
     @Parameter( name = "max_lo", value = "5"),
     @Parameter(name = "sequence", value = "mysequence") })
public Long getId() {
   return this.id;
}
```

@GenericGenerator parametrisiert:

- einen eindeutigen Namen, identisch zu Annotation @GeneratedValue
- eine Generatorstrategie
- ev. weitere Parameter …

weitere Generatoren können selbst erstellt werden:

- Interface IdentifierGenerator muss implementiert werden

Liste der Hibernate Generatoren im Anhang.

20 Beziehungen

- Beziehungen zwischen Klassen – Relation zwischen Tabellen
- Abbilden eines Objektmodells (Vererbung) durch Relationen in einem Datenmodel
- Java Objekte werden in Relationen abgebildet
  - als Komponente
  - durch Assoziationen
- uni-direktional
- bi-direktional (@mappedBy gibt führendes Ende an)
- N Seite als Collection
  - java.lang.List (Reihenfolge der Elemente nicht gegeben ▶ @OrderBy)
  - java.lang.Set
  - java.lang.Map
  - java.lang.Bag

20.1 Komponenten

Entity-Komponente

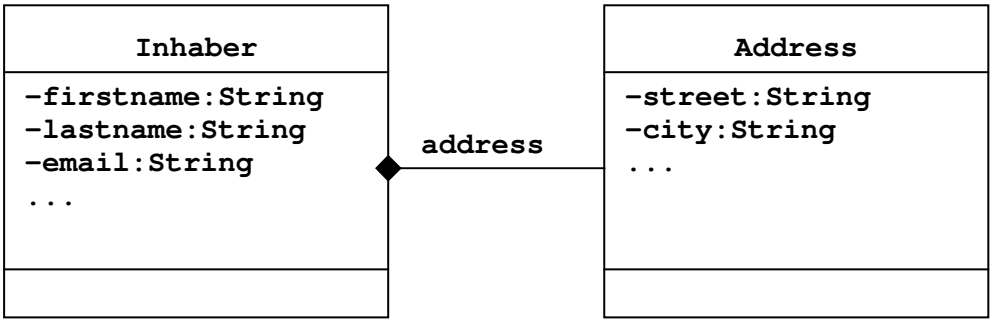
- haben einen Primärschlüssel
- haben einen Lebenszyklus

Value-Komponente

- haben keine Datenbankidentität
- haben keinen Primärschlüssel
- gehören zu einer Entity und ihr Zustand wird innerhalb der Tabelle der dazugehörigen Entity gesichert
- Ausnahme sind Collections von Value-Typen
- typisch sind einfache Objekte vom Typ String
- Lebensdauer eines Value-Typ ist immer an den Lebenszyklus der enstpr. Entity gebunden

Komponenten ermöglichen die Abbildung mehrerer Klassen auf eine Tabelle.

Beziehung Inhaber zu Address



Komposition

- strengere Form der Aggregation
- die Lebensdauer eines Objektes Address ist dabei an das Objekt Inhaber gebunden
- eine Address wird also immer mit oder nach dem Inhaber erzeugt
- und mit dem User zerstört
- Komposition wird in UML (http://www.omg.org) mit einer gefüllten Raute dargestellt

Aggregation

- dargestellt mit einer leeren Raute
- beschreibt eine schwache Beziehung zwischen Objekten
- Objekt ist ein Teil eines anderen Objekts
- kann aber im Gegensatz zur Komposition auch alleine existieren

Die Implementierung mit Java macht keinen Unterschied. Aber Object Address ist für Persistenzprovider ein Value-Typ, hat keinen Primärschlüssel.

Komponenten Implementierung mit Annotationen der Klasse Address.java

```
/**
 * @module jee.2007.Metzler
 * @exercise jee201.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
@NamedQuery(name = "Inhaber.findAll", query = "from Inhaber i")
public class Inhaber implements Serializable {
  @Id
  private Long    id;
  private String  firstname;
  private String  lastname;
  private String  email;
  @Embedded
  // optional
  private Address address;
   ...
```

```
**
 * @module jee.2007.Metzler
 * @exercise jee201.server.Address.java
 */
@Embeddable
public class Address {
  private String street;
  private String city;
  private Address() {
  }
```

```
  public Address(String street, String city) {
    this.street = street;
    this.city = city;
  }
  public String getStreet() {
    return street;
  }
  public void setStreet(String street) {
    this.street = street;
  }
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
  @Override
  public String toString() {
    return " " + this.street + " " + this.city;
  }
}
```

- Annotation @Embeddable definiert Komponente auf Klassenebene
- Entity Inhaber mit dem Attribut address muss nicht gekennzeichnet werden
  - @Embedded ist optional

t_inhaber

id [BIGINT]	firstname [VARCHAR(255)]	lastname [VARCHAR(255)]	email [VARCHAR(255)]	street [VARCHAR(255)]	city [VARCHAR(255)]
1	my_First_1	my_Last_1	my_Email_1@email....	my_Street_1	my_City_1
2	my_First_2	my_Last_2	my_Email_2@email....	my_Street_2	my_City_2
3	my_First_3	my_Last_3	my_Email_3@email....	my_Street_3	my_City_3

- Persistenzprovider definiert als Spaltennamen für Komponenten den Attributnamen
  - Annotation @Column definiert den Spaltennamen benutzerspezifisch
  - name, unique, nullable, insertable, updatable, lenght, precusion, scale, etc.

Annotation @Column			
Parameter	**Beschreibung**	**Typ**	**Default**
name	Spaltenname	String	Name des Attributs
unique	definiert eindeutiges Schlüsselfeld kann alternative über uniqueConstraint auf Tabellenebene deklariert werden	Boolean	false
nullable	erlaubt Null-Werte hat für primitive Java Typen keine Wirkung	Boolean	true

Annotation @Column			
Parameter	Beschreibung	Typ	Default
insertable	definiert die Insert-Funktionalität des Persistence Providers für dieses Attribut	Boolean	true
updatable	definiert die Update-Funktionalität des Persistence Providers für dieses Attribut	Boolean	true
columnDefinition	Fragment der SQL Typendefinition	String	entsprechend referenzierte Spalte
table	Namen der Tabelle, welche die Spalte enthält	String	Primärtabelle
length	Spaltengrösse nur für String anwendbar	int	255
precision	Anzahl der Stellen nur für numerische Spalten anwendbar	int	herstellerabhängig
scale	Anzahl der Nachkommastellen nur für numerische Spalten anwendbar	int	0

```
@Column(name = "address_street")
  public String getStreet() {
    return street;
  }
  public void setStreet(String street) {
    this.street = street;
  }
  @Column(name = "address_city", length = 50, insertable = false)
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
```

- Annotationen @AttributeOverrides und @AttributeOverride überschrieben die Attribute der Komponente von der Entity aus
  - ermöglicht Wiederverwendung der Komponente

Annotation @AttributeOverride			
Parameter	Beschreibung	Typ	Default
name	Attribut	String	
column	Spaltenname	@Column	

Attribute überschreiben mit @AttributeOverrides

```
/**
 * @module jee.2007.Metzler
 * @exercise jee202.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
@NamedQuery(name = "Inhaber.findAll", query = "from Inhaber i")
public class Inhaber implements Serializable {
  @Id
  private Long    id;
  private String  firstname;
  private String  lastname;
  private String  email;
  @Embedded
  @AttributeOverrides( {
      @AttributeOverride( name = "street",
        column = @Column(name = "userstreet")),
      @AttributeOverride( name = "city",
        column = @Column(name = "usercity", length = 50)) })
  private Address address;
...
```

t_inhaber

id [BIGINT]	firstname [VARCHAR(255)]	lastname [VARCHAR(255)]	email [VARCHAR(255)]	userstreet [VARCHAR(255)]	usercity [VARCHAR(50)]
1	my_First_1	my_Last_1	my_Email_1@email....	my_Street_1	my_City_1
2	my_First_2	my_Last_2	my_Email_2@email....	my_Street_2	my_City_2
3	my_First_3	my_Last_3	my_Email_3@email....	my_Street_3	my_City_3

20.2 Assoziationen

- Verbindung mindestens zweier Klassen
  - erlaubt das Navigieren von der einen zur anderen Klasse
- binäre Assoziation ist eine Beziehung zwischen zwei Klassen
- reflexive Assoziation definiert Beziehung zu sich selbst

Kardinalität beschreibt den Grad einer Beziehung zwischen zwei Klassen:

Eins-zu-Eins (1:1)

- eine Klasse steht mit einer anderen Klasse in Beziehung

Eins-zu-Viele/Viele-zu-Eins (1:n/n:1)

- eine Klasse steht mit mehreren anderen Klassen in Beziehung

Viele-zu-Viele (n:n)

- mehrere Klassen stehen mit mehreren anderen Klassen in Beziehung

Bei allen Assoziationen unterscheidet man:

- unidirektionale Beziehungen:
  - Inhaber führt zu Adresse
  - über Adresse kann kein Inhaber definiert werden
  - nur von Inhaber kann zu Adresse navigiert werden

- bidirektionale Beziehungen:
  - Inhaber führt zu Adresse
  - Adresse führt zu Inhaber
    - Entity Address muss Getter und Setter für den User definieren

20.3 Überschreiben von Assoziationen

- Analog zum Re-Mapping von Attributen
  - Mappen von Beziehungen
- @AssociationOverride

Annotation @AssociactionOverride			
Parameter	**Beschreibung**	**Typ**	**Default**
name	Attribut	String	
joinColumns	Spaltenname	@JoinColumn	

Beispiel

```
@Entity
@AssociationOverride(name = "address", joinColumns = "a_id")
public class Inhaber implements Serializable {
  @Id
  private Long    id;
  private String  firstname;
  private String  lastname;
  private String  email;
  private Address address;
...
```

20.4 1:1-Beziehungen

- bei 1:1 Beziehungen haben die beiden Tabellen den gleichen Primary-Key:
  - Primary-Key (Relation) selbst erzeugen
  - Generatorstrategie verwenden

Annotation @OneToOne			
Parameter	**Beschreibung**	**Typ**	**Default**
targetEntity	Entitätsklasse	Class	Typ des Attributs
cascade	Angabe der Kaskadierungsoperationen	CascadeType[]	
fetch	definiert Ladeverhalten des Attribut	FetchType	EAGER
optional	erlaubt Null-Werte gibt an ob die Assoziation belegt sein muss hat für primitive Java Typen keine Wirkung	Boolean	true
mappedBy	definiert Attribut der führenden Entity bei Bidirektionalen Beziehungen	String	

identischer Primärschlüssel selbst erzeugen

- Annotation @OneToOne kennzeichnet Address als 1-zu-1-Beziehung
- Annotation @PrimaryKeyJoinColumn
  - beide in Beziehung stehende Objekte haben immer denselben Primärschlüssel

Unidirektionale 1-zu-1-Beziehung mit gemeinsamem Primärschlüssel

```
/**
 * @module jee.2007.Metzler
 * @exercise jee203.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
@NamedQueries( {
 @NamedQuery( name = "Inhaber.findAll",
  query = "from Inhaber i"),
 @NamedQuery( name = "Inhaber.findLastName",
  query = "select i.id from Inhaber i where i.lastname = :name")
})
public class Inhaber implements Serializable {
  @Id
  @GeneratedValue
  private Long    id;
  private String  firstname;
  private String  lastname;
  private String  email;
```

```
  @OneToOne
  @PrimaryKeyJoinColumn
  private Address address;
...
```

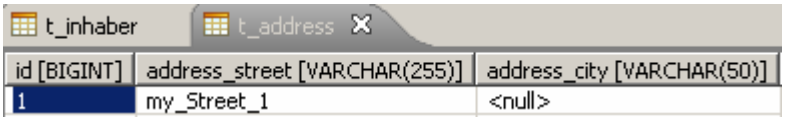
```
/**
 * @module jee.2007.Metzler
 * @exercise jee203.server.Address.java
 */
@Entity
@Table(name = "T_ADDRESS")
public class Address implements Serializable {
  private Long   id;
  private String street;
  private String city;
  private Address() {
  }
  public Address(Long id, String street, String city) {
    this.id = id;
    this.street = street;
    this.city = city;
  }
  @Id
  public Long getId() {
    return id;
  }
...
```

- Primärschlüssel für die Entity Address muss selbst gesetzt werden
- persist(inhaber) erzeugt einen Primärschlüssel für den Inhaber

```
/**
 * @module jee.2007.Metzler
 * @exercise jee203.client.BankClient.java
 */
public class BankClient {
  public static void main(String[] args) {
    try {
      InitialContext ctx = new InitialContext();
      Bank bank = (Bank) ctx.lookup("BankBean/remote");
      Inhaber i1 = new Inhaber("my_First_1", "my_Last_1",
          "my_Email_1@email.com");
      bank.addInhaber(i1);
      long id = bank.getInhaber(i1.getLastname());
      Address a1 = new Address(id, "my_Street_1", "my_City_1");
      i1.setAddress(a1);
      bank.addAddress(a1);
      ...
```

t_inhaber | t_address

id [BIGINT]	firstname [VARCHAR(255)]	lastname [VARCHAR(255)]	email [VARCHAR(255)]
1	my_First_1	my_Last_1	my_Email_1@email....



t_inhaber | t_address

id [BIGINT]	address_street [VARCHAR(255)]	address_city [VARCHAR(50)]
1	my_Street_1	<null>

- address_city ist null
  - insertable = false

```
@Column(name = "address_city", length = 50, insertable = false)
public String getCity() {
  return city;
}
```

bi-direktionale 1:1 Beziehung

- Klasse Address kann Inhaber referenzieren
- getInhaber() in der Entity Address mit Annotation @OneToOne
  - Attribut mappedBy definiert die "owning"-Seite der Beziehung

```
/**
 * @module jee.2007.Metzler
 * @exercise jee205.server.Address.java
 */
@Entity
@Table(name = "T_ADDRESS")
public class Address implements Serializable {
  @Id
  @GeneratedValue
  private Long    id;
  private String  street;
  private String  city;
  private Inhaber inhaber;
  @OneToOne(mappedBy = "address")
  public Inhaber getInhaber() {
    return inhaber;
  }
  ...
```

20.5 Generierung des Primary Key

Generatorstrategie foreign

- Hibernate Funktionalität
  - bindet Persistenz-Provider
- 1-zu-1-Beziehung mit gemeinsamem Primärschlüssel
- beide Entities haben denselben Primärschlüssel
- Parameter property referenziert Komponente
- Primärschlüssel in der Entity Address wird mit der Generatorstrategie foreign generiert

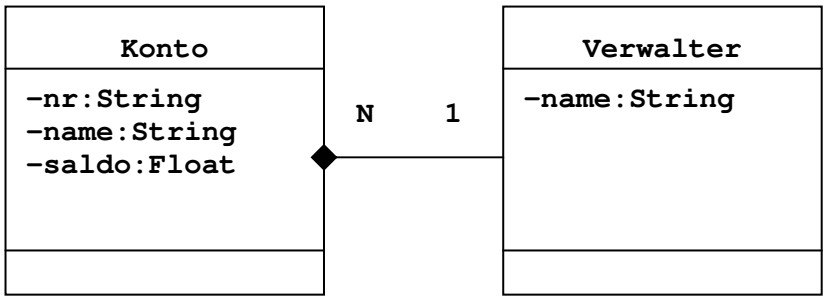
```
import org.hibernate.annotations.*;

/**
 * @module jee.2007.Metzler
 * @exercise jee204.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
public class Inhaber implements Serializable {
  @Id
  @GeneratedValue(generator = "foreign")
  @GenericGenerator( name = "foreign",
                     strategy = "foreign",
                     parameters = {
     @Parameter(name = "property", value = "address") })
  private Long    id;
```

- Generatorstrategie AUTO erzeugt Address Primärschlüssel automatisch
  - id in Address darf nicht mehr gesetzt werden
  - persist() sichert Inhaber (inkl. Addresse), wenn Inhaber mit Adresse definiert sit

```
/**
 * @module jee.2007.Metzler
 * @exercise jee204.client.BankClient.java
 */
public class BankClient {
  public static void main(String[] args) {
    try {
      InitialContext ctx = new InitialContext();
      Bank bank = (Bank) ctx.lookup("BankBean/remote");
      Inhaber i1 = new Inhaber("my_First_1", "my_Last_1",
          "my_Email_1@email.com");
      Address a1 = new Address("my_Street_1", "my_City_1");
      i1.setAddress(a1);
      bank.addInhaber(i1);
     ...
```

20.6 1:n und n:1-Beziehungen

- jedes Konto hat einen Verwalter
- ein Verwalter kann kein, eins oder mehrere Konten verwalten



Annotation @ManyToOne			
Parameter	**Beschreibung**	**Typ**	**Default**
targetEntity	Entitätsklasse	Class	Typ des Attributs
cascade	Angabe der Kaskadierungsoperationen	CascadeType[]	
fetch	definiert Ladeverhalten des Attributs	FetchType	LAZY
optional	erlaubt Null-Werte gibt an ob die Assoziation belegt sein muss hat für primitive Java Typen keine Wirkung	Boolean	true

@ManyToOne Attribute

- cascade [ALL, MERGE, PERSIST, REFRESH, REMOVE,..]
  - merge( ), persist( ) usw. wird an die referenzierten Objekte durchgereicht
  - Default = kein Durchreichen

- fetch [EAGER, LAZY]
  - EAGER : Fetch-Strategie lädt die Beziehungsobjekte sofort mit
  - LAZY : lädt erst beim Zugriff auf Beziehung das entsprechende Objekt nach

- referencedColumnName defniert Name des referenzierten Feldes

1-zu-n bzw. n-zu-1-Beziehung zwischen Konto und Verwalter

```
/**
 * @module jee.2007.Metzler
 * @exercise jee205.server.Konto.java
 */
@Entity
@Table(name = "T_Konto")
public class Konto implements Serializable {
  private Long      id;
  private String    nr;
  private String    name;
  private Float     saldo;
  private Verwalter verwalter;
  public Konto(String nr, String name, Float saldo,
      Verwalter verwalter) {
    super();
    this.nr = nr;
    this.name = name;
    this.saldo = saldo;
    this.verwalter = verwalter;
  }
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  public String getNr() {
    return nr;
  }
  public void setNr(String nr) {
    this.nr = nr;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public Float getSaldo() {
    return saldo;
  }
  public void setSaldo(Float saldo) {
    this.saldo = saldo;
  }
  @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
  @JoinColumn(nullable = false)
  public Verwalter getVerwalter() {
    return verwalter;
  }
  public void setVerwalter(Verwalter verwalter) {
    this.verwalter = verwalter;
  }
}
```

Annotation @OneToMany			
Parameter	**Beschreibung**	**Typ**	**Default**
targetEntity	Entitätsklasse	Class	Typ des Attributs
cascade	Angabe der Kaskadierungsoperationen	CascadeType[]	
fetch	definiert Ladeverhalten des Attributs	FetchType	LAZY
optional	erlaubt Null-Werte gibt an ob die Assoziation belegt sein muss hat für primitive Java Typen keine Wirkung	Boolean	true
mappedBy	definiert Attribut der führenden Entity	String	

Beispiel

```
/**
 * @module jee.2007.Metzler
 * @exercise jee206.server.Verwalter.java
 */
@Entity
@Table(name = "T_Verwalter")
public class Verwalter implements Serializable {
  private Long       id;
  private String     name;
  private Set<Konto> kontos = new HashSet<Konto>();
  public Verwalter(String name, Set<Konto> kontos) {
    super();
    this.name = name;
    this.kontos = kontos;
  }
  @Id
  @GeneratedValue
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  @OneToMany(mappedBy = "verwalter")
  public Set<Konto> getKontos() {
    return kontos;
  }
```

```
  public void setKontos(Set<Konto> kontos) {
    this.kontos = kontos;
  }
}
```

Annotation @JoinColumn			
Parameter	Beschreibung	Typ	Default
name	Spaltenname	String	Name des Attributs
unique	definiert eindeutiges Schlüsselfeld kann alternativ über uniqueConstraint auf Tabellenebene deklariert werden	Boolean	false
nullable	erlaubt Null-Werte hat für primitive Java Typen keine Wirkung	Boolean	true
insertable	definiert die Insert-Funktionalität des Persistence Providers für dieses Attribut	Boolean	true
updatable	definiert die Update-Funktionalität des Persistence Providers für dieses Attribut	Boolean	true
columnDefinition	Fragment der SQL Typendefinition	String	entsprechend referenzierte Spalte
referenced-ColumnName	Name der Spalte auf die referenziert wird	String	PK der referenzierten Tabelle

Beziehungen mit Verbindungstabelle

- 1:n und/oder n:1 Beziehung mit Verbindungstabelle
  - Annotation @JoinTable, @JoinColumns definiert Tabellen-, Feldnamen
  - interessant bei bestehendem DB-Schema

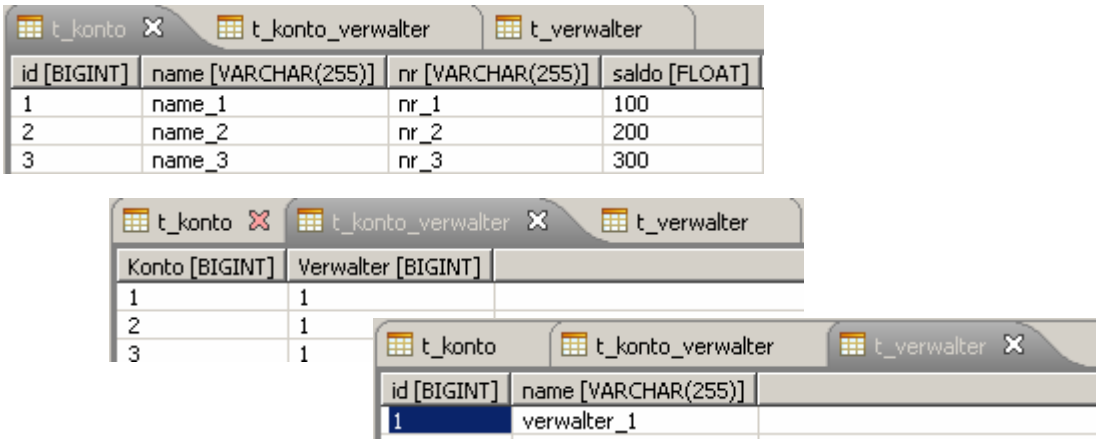
Annotation @JoinTable			
Parameter	Beschreibung	Typ	Default
name	Name der Relationstabelle	String	
schema	Tabellenschema	String	datenbank-spezifisches Schema
catalog	Tabellenkatalog	String	datenbank-spezifischer Katalog
joinColumns	PK Spalten der führenden Entity	JoinColumn[]	
inverseJoinColumn	PK Spalten der nichtführenden Entity	JoinColumn[]	
uniqueConstraint	Unique-Constraints, die auf der Relationstabelle definiert werden sollen	UniqueConstraint[]	

viele-zu-eins mit Verbindungstabelle: T_KONTO_VERWALTER

```
/**
 * @module jee.2007.Metzler
 * @exercise jee207.server.Konto.java
 */
@ManyToOne( cascade = CascadeType.ALL,
            fetch = FetchType.LAZY)
@JoinTable( name = "T_KONTO_VERWALTER",
     joinColumns = {@JoinColumn( name = "Konto") },
     inverseJoinColumns = { @JoinColumn(name = "Verwalter") })
  public Verwalter getVerwalter() {
    return verwalter;
  }
```

Die Join-Tabelle wird per Default aus den Namen der beiden Tabellen zusammengesetzt

- oder Annotation @JoinTable
  - Attribut joinColumns
  - Attribut inverseJoinColumns



t_konto | t_konto_verwalter | t_verwalter

id [BIGINT]	name [VARCHAR(255)]	nr [VARCHAR(255)]	saldo [FLOAT]
1	name_1	nr_1	100
2	name_2	nr_2	200
3	name_3	nr_3	300

t_konto | t_konto_verwalter | t_verwalter

Konto [BIGINT]	Verwalter [BIGINT]
1	1
2	1
3	1

t_konto | t_konto_verwalter | t_verwalter

id [BIGINT]	name [VARCHAR(255)]
1	verwalter_1

falls keine Kaskadierung für die Beziehung zwischen Konto und Verwalter definiert wurde ist ein expliziter Aufruf von persist(verwalter) notwendig

bi-direktionale One-To-Many Beziehung

- setzen der Beziehungen auch auf der referenzierten Seite
  - addKonto für die Entity Verwalter

```
/**
 * @module jee.2007.Metzler
 * @exercise jee208.server.Verwalter.java
 */
  // setzen der Beziehungen auf der referenzierten Seite
  public void addKonto(Konto konto) {
    this.kontos.add(konto);
    konto.setVerwalter(this);
  }
  public void deleteKonto(Konto konto) {
    this.kontos.remove(konto);
    konto.setVerwalter(null);
  }
```

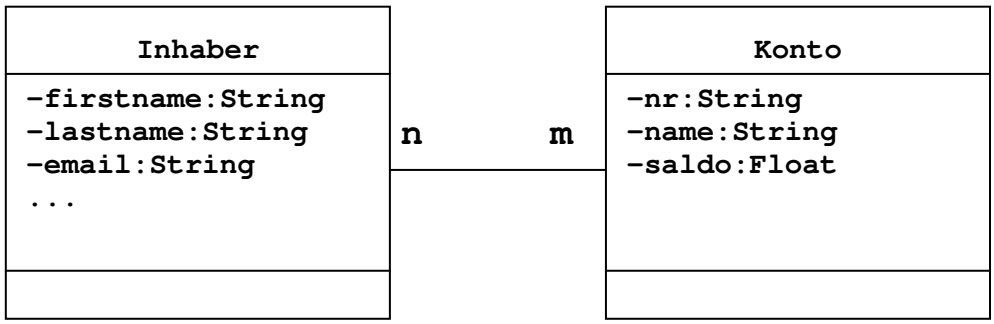
- Speichern einer Verwalter- Konto Beziehung

```
/**
 * @module jee.2007.Metzler
 * @exercise jee208.client.BankClient.java
 */
public class BankClient {
  public static void main(String[] args) {
    try {
      InitialContext ctx = new InitialContext();
      Bank bank = (Bank) ctx.lookup("BankBean/remote");
      Konto k1 = new Konto("nr_1", "name_1", 100.0f, null);
```

```
      Konto k2 = new Konto("nr_2", "name_2", 200.0f, null);
      Konto k3 = new Konto("nr_3", "name_3", 300.0f, null);
      Set<Konto> kontos = new HashSet<Konto>();
      Verwalter v = new Verwalter("verwalter_1", kontos);
      v.addKonto(k1);
      v.addKonto(k2);
      v.addKonto(k3);
      bank.addVerwalter(v);
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }
}
```

20.7 n:m-Beziehungen

- immer mit einer Verbindungstabelle (Assoziationstabelle)
- Annotation @ManyToMany
- Beispiel : Inhaber und Konto




```
/**
 * @module jee.2007.Metzler
 * @exercise jee209.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
public class Inhaber implements Serializable {
  @Id
  @GeneratedValue
  private Long       id;
  private String     firstname;
  private String     lastname;
  private String     email;
  private Address    address;
  private Set<Konto> kontos = new HashSet<Konto>();
  ...
  @ManyToMany
  public Set<Konto> getKontos() {
    return kontos;
  }
  public void setKontos(Set<Konto> kontos) {
    this.kontos = kontos;
```

```
  }
  ...
}
```

Konto Entity definiert die "owning"-Seite der bidirektionalen Beziehung mit mappedBy:

```
**
 * @module jee.2007.Metzler
 * @exercise jee209.server.Konto.java
 */
@Entity
@Table(name = "T_Konto")
public class Konto implements Serializable {
  private Long         id;
  private String       nr;
  private String       name;
  private Float        saldo;
  private Verwalter    verwalter;
  private Set<Inhaber> inhabers = new HashSet<Inhaber>();
  ...
  @ManyToMany(mappedBy = "kontos")
  public Set<Inhaber> getInhabers() {
    return inhabers;
  }
  public void setInhabers(Set<Inhaber> inhabers) {
    this.inhabers = inhabers;
  }
  public void addInhabers(Inhaber inhaber) {
    this.inhabers.add(inhaber); // Konto zu Inhaber-Set
    inhaber.getKontos().add(this); // Inhaber zu Konto-Set
  }
```

- Setzen der bi-direktionalen Beziehungen:
  - Inhaber.setKontos(Set<Book> kontos)
  - Konto.addInhaber(Inhaber inhaber)

- n-zu-m-Beziehung zwischen Inhaber und Konto resultiert in folgenden Tabellen:
  - Join-Table : t_inhaber_t_konto

20.8 Kaskadieren von Persistenzoperationen

Transitive Persistenz bezeichnet alle Objekte, die von einem persistenten Objekt erreichbar sind werden selbst auch persistent

- erlaubt die Weitergabe von Entity-Operationen : persist(), merge(), delete()
- mit Annotationen : CascadeType
- bei Beziehungen : @OneToOne, @OneToMany / @ManyToOne, @ManyToMany

```
@ManyToOne(cascade = CascadeType.PERSIST)
```

- A.persist() persistiert auch alle an A hängenden Objekte B

- weitere CascadeTypes überschreiben die CascadeTypes der Java Persistence API
  - Annotation @org.hibernate.annotations.Cascade
  - definiert CascadeType mit Annotation @org.hibernate.annotations.CascadeType

```
@OneToMany
@org.hibernate.annotations.Cascade( {
     @org.hibernate.annotations.CascadeType.SAVE_UPDATE,
     @org.hibernate.annotations.CascadeType.MERGE })
public List<B> getB() {
   return b;
}
```

Übersicht der möglichen CascadeTypes von JPA und Hibernate

CascadeTypes		
JPA javax.persistence. CascadeType	HIBERNATE org.hibernate.annotations. CascadeType	Beschreibung
PERSIST	PERSIST	Übergang zu managed Entity Instanz; commit() oder flush() persistiert Objekt in der Datensource
MERGE	MERGE	Übergang von einer detached Entity zu einer managed Entity Instanz
REMOVE	REMOVE	Übergang zu removed Entity Instanz; commit() oder flush() löscht Objekt in der Datensource
REFRESH	REFRESH	Synchronisiert die Entity Instanz mit der Datensource. Änderungen innerhalb der Entity werden dabei überschrieben
	DELETE	Entspricht javax.persistence.REMOVE
ALL		Entspricht cascade = { PERSIST, MERGE, REMOVE, REFRESH }
	SAVE_UPDATE	save(Object entity) oder update(Object entity) wird an die in Beziehung stehenden Objekte mit diesem CascadeType durchgereicht
	REPLICATE	replicate(Object entity, ...) Objekt wird unter Verwendung der existierenden Id in der Datenbank abgelegt
	DELETE_ORPHAN	Alle persistenten Entities, die aus einer Beziehung zu einer persistenten Entity, beispielsweise aus

CascadeTypes		
JPA javax.persistence. CascadeType	HIBERNATE org.hibernate.annotations. CascadeType	Beschreibung
		einer Collection, genommen wurden (Objekte, die nicht mehr referenziert werden), werden automatisch gelöscht
	LOCK	lock(Object entity, ...) Objekt wird mit angegebenem LockMode versehen, um beispielsweise Versionschecks oder pessimistisches Locking durchzuführen
	EVICT	evict(Object entity) Objekt wird aus der Session gelöscht und damit verhindert, dass Änderungen in die Datenbank geschrieben werden
	ALL	Beinhaltet … und SAVE_UPDATE, DELETE, EVICT und LOCK.

Beispiel transitiver Persistenz

- Verwalter und Konto Beziehung definiert CascadeType.Persist und CascadeType.Merge
  - kein expliziter Aufruf durch persist(konto) notwendig

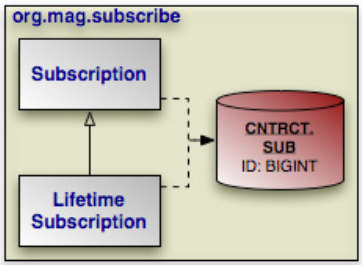
```
@OneToMany( mappedBy = "verwalter",
            cascade = { CascadeType.PERSIST,
                        CascadeType.MERGE
})
public Set<Konto> getKontos() {
    return kontos;
}
```

21 Persistenzabbildung

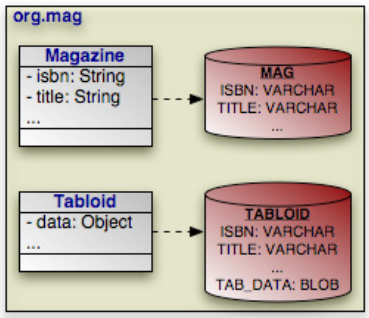
21.1 Vererbung

- JPA definiert Vererbung in relationale Datenbanken
- drei verschiedene Vererbungsstrategien:

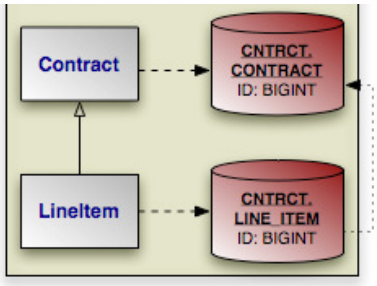
  - SINGLE_TABLE
    - eine Tabelle für die Abbildung einer Hierarchie
    - Unterscheidungsmerkmal ist die Diskriminator Spalte
    - Polymorphe Abfragen sind möglich



  - TABLE_PER_CLASS
    - eine Tabelle pro konkreter Entity-Klasse
    - Unterstützung durch Implementierung optional
    - jede Tabelle enthält ALLE Attribute (auch Superclass)
    - keine polymorphen Abfragen möglich
    - SQL UNION als Behelf

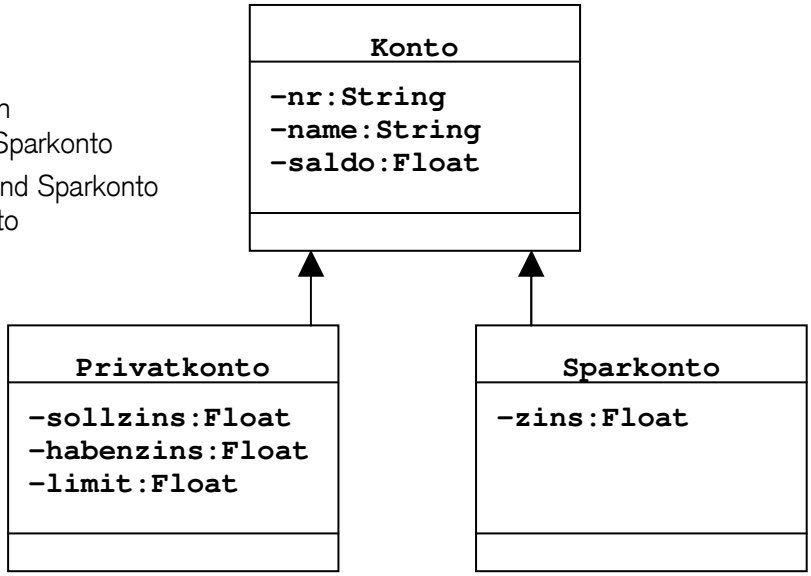


  - JOINED
    - eine Tabelle je Superklasse und abgeleitete Klasse
    - eine Tabelle je Klasse einer Vererbungshierarchie
    - jede Tabelle enthält nur spezifische Attribute der Klasse
    - polymorphe Abfragen sind möglich (nicht performant)



http://openjpa.apache.org

- Vererbungshierarchie von Konto, Privatkonto und Sparkonto
  - Entities Privatkonto und Sparkonto erben von Entity Konto



21.1.1 SINGLE_TABLE

- einfachste Möglichkeit
- Defaultstrategie
- Klassen werden in einer einzigen Tabelle abgebildet
- Annotation @Inheritance bestimmt Vererbungsstrategie
- Annotation @DiscriminatorValue definiert mapping zwischen Objekt und Tabellen-Eintrag
- Annotation @DiscriminatorColumn definiert Typ und der Name der Discriminatorspalte
  - beschreibt Vererbung, z.B: Sparkonto oder Privatkonto
  - Defaultwerte : Name (DTYPE), Typ (String)

Beispiel mit SINGLE_TABLE

- abstrakte Klasse Konto bestimmt Vererbungsstrategie mit InheritanceType
- konkrete Klassen Privatkonto und Sparkonto definieren Discriminatorvalue

Konto.java mit Vererbungsstrategie SINGLE_TABLE

```
/**
 * @module jee.2007.Metzler
 * @exercise jee211.server.Konto.java
 */
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "T_Konto")
public abstract class Konto implements Serializable {
  private Long      id;
  private String    nr;
  private String    name;
  private Float     saldo;
  private Verwalter verwalter;
...
```

Privatkonto.java

```
/**
 * @module jee.2007.Metzler
 * @exercise jee211.server.Privatkonto.java
 */
@Entity
@DiscriminatorValue(value = "Privatkonto")
public class Privatkonto extends Konto {
  private Float sollzins;
  private Float habenzins;
  private Float kredit;
...
```

Sparkonto.java

```
/**
 * @module jee.2007.Metzler
 * @exercise jee211.server.Sparkonto.java
 */
@Entity
@DiscriminatorValue(value = "Sparkonto")
public class Sparkonto extends Konto {
  private Float zins;
...
```

Bankclient.java

```
Konto k1 = new Privatkonto("nr_1", "name_1", 100.0f, null,
           5.5F, 4.5F, 10000.0F);
Konto k2 = new Sparkonto("nr_2", "name_2", 200.0f, null,
           5.0F);
Set<Konto> kontos = new HashSet<Konto>();
Verwalter v = new Verwalter("verwalter_1", kontos);
v.addKonto(k1);
v.addKonto(k2);
bank.addVerwalter(v);
```

Tabellenstruktur: Single_Table

t_konto

DTYPE [VARCHAR(31)]	id [BIGINT]	name [VARCHAR(255)]	nr [VARCHAR(255)]	saldo [FLOAT]	zins [FLOAT]	sollzins [FLOAT]	habenzins [FLOAT]	kredit [FLOAT]
Privatkonto	1	name_1	nr_1	100	<null>	5.5	4.5	10000
Sparkonto	2	name_2	nr_2	200	5	<null>	<null>	<null>
<null>	<null>	<null>	<null>	<null>	<null>	<null>	<null>	<null>

Vorteil der SINGLE_TABLE-Strategie

- gute Performance, da polymorphe Abfragen nur immer eine Tabelle betreffen
  - geeignet für wenig Attribute
  - polymorphe Abfragen

Nachteil der SINGLE_TABLE-Strategie:

- grosse Vererbungshierarchien erzeugen sehr grosse Tabellen
- redundante Spalten
- schlechte Datenintegrität da Felder der abgeleiteten Klassen NULL Werte haben

Annotation @DiscriminatorColumn			
Parameter	**Beschreibung**	**Typ**	**Default**
name	Spaltenname	String	DTYPE
discriminatorType	Spaltentyp [STRING, CHAR, INTEGER]	DiskriminatorTyp	STRING
columnDefinition	Fragment der SQL Typendefinition	String	entsprechend referenzierte Spalte
lenght	Spaltenbreite nur für STRING	int	31

21.1.2 TABLE_PER_CLASS

- jede konkrete Klasse in eigene Tabelle
- Tabellen eindeutig den Entities zugeordnet
- kein Unterscheidungsfeld (Discriminator) notwendig

Die Persistenzprovider sind nach der Spezifikation der Java Persistence API (EJB 3.0) nicht verpflichtet, die TABLE_PER_CLASS-Strategie bereitzustellen.

Beispiel mit TABLE_PER_CLASS

- abstrakte Klasse Konto bestimmt Vererbungsstrategie
- Tabellen für konkrete Entitäten erhalten auch die Attribute der geerbten Klasse

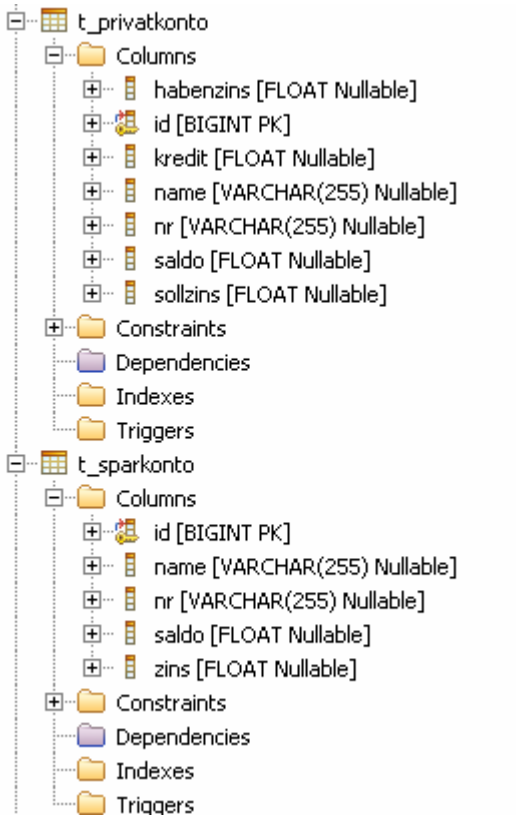
```
/**
 * @module jee.2007.Metzler
 * @exercise jee212.server.Konto.java
 */
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Table(name = "T_Konto")
public abstract class Konto implements Serializable {
...
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  public Long getId() {
    return id;
  }
...
```

```
/**
 * @module jee.2007.Metzler
 * @exercise jee212.server.Privatkonto.java
 */
@Entity
@Table(name = "T_PRIVATKONTO")
public class Privatkonto extends Konto {
```

```
/**
 * @module jee.2007.Metzler
 * @exercise jee212.server.Sparkonto.java
 */
@Entity
@Table(name = "T_SPARKONTO")
public class Sparkonto extends Konto {
```

ACHTUNG: JBoss verlangt TABLE-Generation-Stategy bei "Mapped Tables"

Tabellenstruktur: Table_Per_Class



Vorteil der TABLE_PER_CLASS –Strategie

- geeignet für Veerbungshierarchien ohne polymorphe Abfragen und Beziehungen
- Abfragen auf konkrete Klassen sehr einfach und performant:

```
SELECT id, name, ...
FROM Privatkonto
```

Nachteil der TABLE_PER_CLASS -Strategie:

- Tabellenstruktur ist abhängig von den Superklassen
- polymorphe Abfragen nicht optimal unterstützt
  - Abfragen über alle Books:

```
SELECT id, name, ...
FROM Privatkonto
WHERE ...

SELECT id, name, zins
FROM Sparkonto
WHERE ...
```

- Hibernate als Persistenceprovider unterstützt SQL-Unions:

```
SELECT * FROM (
   SELECT id, name, ..., 'P' AS type
   FROM Privatkonto
   UNION
   SELECT id, name, zins 'S' AS type
   FROM Sparkonto )
WHERE ...
```

21.1.3 JOINED

- jede abstrakte und jede konkrete Klasse hat eigene Tabelle

```
/**
 * @module jee.2007.Metzler
 * @exercise jee213.server.Konto.java
 */
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "T_Konto")
public abstract class Konto implements Serializable {
...
```

Tabellenstruktur: Joined

t_konto | t_privatkonto | t_sparkonto

id [BIGINT]	name [VARCHAR(255)]	nr [VARCHAR(255)]	saldo [FLOAT]	verwalter_id [BIGINT]
1	name_2	nr_2	200	2
2	name_1	nr_1	100	3

t_konto | t_privatkonto | t_sparkonto

id [BIGINT]	sollzins [FLOAT]	habenzins [FLOAT]	kredit [FLOAT]
2	5.5	4.5	10000

t_konto | t_privatkonto | t_sparkonto

id [BIGINT]	zins [FLOAT]
1	5

- jede Tabelle hat einen Primärschlüssel id

- Primärschlüssel id ist auch Fremdschlüssel zur Superklasse t_konto
- JOINED-Strategie benötigt keinen Discriminator
  - optionale Definition möglich für Implementierungen anderer Persistenzprovider

Vorteil der JOINED-Strategie:

- Datenbankintegrität wird nicht verletzt
  - Attribute in den Subklassen haben keine NULL Werte
- polymorphe Abfragen und Assoziationen sind möglich

Nachteil der JOINED-Strategie:

- komplexe Abfragen
- schlechte Performance
- Abfrage auf eine konkrete Klasse benötigt Inner-Joins

```
SELECT p.id, p.sollzins, k.name
FROM t_privatkonto p
INNER JOIN t_konto k ON k.id = p.id
WHERE ...
```

- für Abfragen über alle Kontos sind Outer Joins notwendig

```
SELECT p.id, p.sollzins, s.zins
FROM t_konto AS k
LEFT OUTER JOIN t_privatkonto AS p ON k.id = p.id
LEFT OUTER JOIN t_sparkonto AS s ON k.id = s.id
WHERE ...
```

21.2 Sekundärtabellen

- @SecondaryTable deklariert Instanzvariable email als Value-Typ
- @SecondaryTable gibt die Tabelle der Email-Adressen an
- @JoinColumn enthält Tabellenspalte mit Fremdschlüssel der Entity
- @Column definiert den Namen der Tabellenspalte der Email-Adressen

Annotation @SecondaryTable			
Parameter	**Beschreibung**	**Typ**	**Default**
name	Tabellenname	String	
schema	Tabellenschema	String	datenbank-spezifisches Schema

Annotation @SecondaryTable			
Parameter	Beschreibung	Typ	Default
catalog	Tabellenkatalog	String	datenbank-spezifischer Katalog
uniqueConstraint	Unique-Constraints, die auf der Tabelle definiert werden sollen	Unique-Constraint[]	
pkJoinColumns	PKs der Tabellen über die die Beziehung hergestellt werden soll	PrimaryKey-CoinColumn[]	alle PKs der Primärtabelle

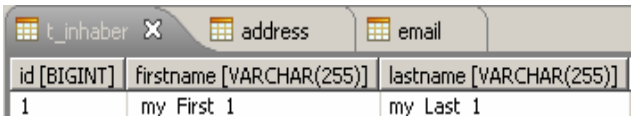
Entity Inhaber definiert Adresse und Email in Sekundärtabellen

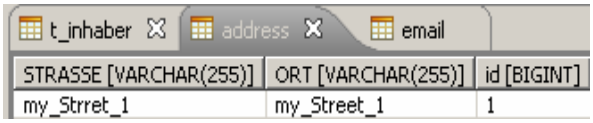
```
/**
 * @module jee.2007.Metzler
 * @exercise jee214.server.Inhaber.java
 */
@SuppressWarnings( { "serial", "unused", "unchecked" })
@Table(name = "T_Inhaber")
@SecondaryTables( {
     @SecondaryTable(name = "ADDRESS"),
     @SecondaryTable(name = "EMAIL") })
@Entity
public class Inhaber implements Serializable {
  private Long   id;
  private String firstname;
  private String lastname;
  private String email;
  private String street;
  private String city;
  @Column(name = "STRASSE", table = "ADDRESS")
  public String getStreet() {
    return street;
  }
  public void setStreet(String street) {
    this.street = street;
  }
  @Column(name = "ORT", table = "ADDRESS")
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
  @Column(name = "EMAIL", table = "EMAIL")
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
```

Inhaber mit Adresse und Email speichern:

```
/**
 * @module jee.2007.Metzler
 * @exercise jee214.client.BankClient.java
 */
Inhaber i = new Inhaber(" my_First_1", "my_Last_1",
          "my_Email_1@email.com", "my_Strret_1", "my_Street_1");
bank.addInhaber(i);
```

folgende Tabellen werden generiert:





22 Collections

Assoziationen zwischen Entities werden als Collections umgesetzt.

- auch Verwendung von Value Types
  - repräsentieren keine eigenständigen Entities innerhalb der Datenbank

- Collections als Attribute von persistenten Klassen
  - durch Deklaration des Typs als Interface

Folgende Interfaces sind möglich:

- java.util.Set / SortedSet
- java.util.Collection
- java.util.List
- java.util.Map / SortedMap
- selbst definierte Interfaces durch org.hibernate.usertype.UserCollectionType

Die Verwendung von Collections in persistenten Klassen erlaubt nur die erwähnten Interfaces

folgender Code wirft eine Laufzeit-Exception:

```
/**
 * @module jee.2007.Metzler
 * @exercise jee221.client.BankClient.java
 */
Verwalter v = bank.getVerwalter(1L);
Set kontos = v.getKontos();                    // ok
HashSet my_kontos = (HashSet) v.getKontos();   // Exception
```

- Hibernate als Persistenzprovider ersetzt HashSet während persist(...) durch eigene Implementierung
  - durch Entity referenzierte Collection-Instanzvariablen werden persistiert
  - referenzlose Collection-Instanzvariablen werden gelöscht
- zwei Entities dürfen nicht auf gleiche Collection-Instanz verweisen
  - Hibernate unterscheidet nicht zwischen einer Null-Referenz auf eine Collection und einer leeren Collection

22.1 Collections mit Index

- Indexierte Collections z. B. List, Map oder Arrays
  - erlauben Zugriff über Index
- müssen Indexspalte (bzw. ein Key bei Maps) definieren
  - enthält den Index der Collection
  - List und Array erlauben nur Integer
  - Map erlaubt beliebiger Basistyp

23 Datenbankabfragen

- Entities in einer Datenbank mittels Abfragen finden
- Query Interface
  - zentrale Schnittstelle zur Ausführung von Abfragen
  - JPQL (Java Persistence Query Language)
- Hibernate Query Language (HQL)
  - starke, komplett objektorientierte Abfragesprache
- Criteria API
  - Abfragen durch Aufruf einer normalen API formuliert
- Abfragen per SQL formuliert

23.1 Query Interface

- zentrale Schnittstelle zur Ausführung von Abfragen
- immer wenn Primärschlüssel einer Entity nicht bekannt ist
- Suchkriterien zum Finden der Entities
- Query-Instanz wird mit Hilfe des EntityManager erzeugt
- String-Parameter muss eine gültige JPQL-Abfrage definieren
- oder mit createNativeQuery( ) ein SQL String definieren
  - Datenbankabhängig

```
/**
 * @module jee.2007.Metzler
 * @exercise jee231.server.BankBean.java
 */
 public Vector<Inhaber> getInhaber(String lastname,
     String firstname) {
   Query q = manager.createQuery("SELECT  i FROM Inhaber i  "
       + "WHERE i.lastname LIKE :lastname "
       + "AND i.firstname LIKE :firstname");
   q.setParameter("lastname", lastname);
   q.setParameter("firstname", firstname);
   List liste = q.getResultList();
   Vector<Inhaber> erg = new Vector<Inhaber>();
   for (Object o : liste) {
     erg.add((Inhaber) o);
   }
   return erg;
 }
```

23.1.1 Methoden des Query Interface

QueryInterface	
Methode	**Beschreibung**
List getResultList( )	Das Ergebnis wird als Liste zurückgegeben, das heisst, das Resultat der Abfrage befindet sich komplett im Speicher
Object getSingleResult( )	Ist von vornherein bekannt, dass eine Abfrage nur ein einziges Objekt zurück liefert, kann diese Methode verwendet werden, um den Umweg über die Liste zu sparen
int executeUpdate( )	Ermöglicht UODATE und DELETE Anweisungen, die sich auf eine ganze Reihe von Entity Beans auswirken könnnen.
Query setMaxResults() **Query setFirstResults()**	erlaubt ausschnittweises oder seitenweises Eingrenzen der Ergebnismenge
Query setHint()	setzen herstellerspezifischer Parameter, z.B. org.hibernate.timeout
Query setParameter()	setzt die Parameter einer dynamischen Query
Query setFlushMode()	setzt den Synchronisationsmodus AUTO synchronisiert Persistenzkontext und DB vor der Abfrage COMMIT Persistenzkontext synchronisiert erst bei COMMIT

23.2 Ausführen von Abfragen

Query Interface liefert:

- einfache Instanzen der entsprechenden Wrapperklassen (Integer, Long, etc.)
- Object Array

```
List liste = q.getResultList();
Vector<Inhaber> erg = new Vector<Inhaber>();
for (Object o : liste) {
  erg.add((Inhaber) o);
}
```

23.3 Eingrenzung der Abfrage

- setMaxResults(int maxResults) grenzt die Ergebnismenge ein
- setFirstResult(int firstResult) setzt einen Zeiger in der Ergebnismenge
  - Listenweise Ausgabe der Elemente

```
Query q = manager.createQuery("FROM Inhaber i  "
   + "WHERE i.lastname LIKE :lastname "
   + "AND i.firstname LIKE :firstname");
q.setParameter("lastname", lastname);
q.setParameter("firstname", firstname);
q.setFirstResult(5).setMaxResults(3);
```

23.4 dynamische Abfragen

- benannte Parameter

Verwendung von benannten Parametern

```
/**
 * @module jee.2007.Metzler
 * @exercise jee232.server.BankBean.java
 */
    Query q = manager.createQuery("FROM Inhaber i  "
        + "WHERE i.lastname LIKE :lastname "
        + "AND i.firstname LIKE :firstname");
    q.setParameter("lastname", lastname);
    q.setParameter("firstname", firstname);
```

23.5 namedQuery

- Definition per Metadaten
- als JPQL oder SQL

Annotation @NamedQuery(...)

```
/**
 * @module jee.2007.Metzler
 * @exercise jee233.server.Inhaber.java
 */
@SuppressWarnings( { "serial", "unused", "unchecked" })
@Table(name = "T_Inhaber")
@Entity
@NamedQueries( {
    @NamedQuery( name = "Inhaber.findAll",
                 query = "from Inhaber i"),
    @NamedQuery( name = "Inhaber.findLastName",
                 query = "select i.lastname from Inhaber i
                          where i.lastname like :name") })
public class Inhaber implements Serializable {
```

Abfragen mit namedQueries

```
/**
 * @module jee.2007.Metzler
 * @exercise jee233.server.BankBean.java
 */
  public Vector<String> getInhaber(String lastname) {
    Query q = manager.createNamedQuery("Inhaber.findLastName");
    q.setParameter("name", lastname);
    List liste = q.getResultList();
    Vector<String> erg = new Vector<String>();
    for (Object o : liste) {
      erg.add((String) o);
    }
    return erg;
  }
```

23.6 JPQL – Java Persistence Query Language

- Abfragesprache
- nicht case sensitive
- aber : Javaklassen und Properties entsprechend korrekt angeben
  - objektorientiert
    - Abfragen von Objekten über deren Vererbungsbeziehung
- "from inhaber"
  - JPQL : Abfrage der Entities vom Typ Inhaber oder einem Subtyp
  - SQL : Abfrage der Daten aus der Tabelle inhaber

23.6.1 from

- Abfrage der Entity-Klasse

```
from server.Inhaber
```

```
from Inhaber          // Hibernate-Mappings Attribute
                        auto-import = true (DEFAULT)
```

```
from Inhaber as i     // mit Alias
from Inhaber i
```

```
from Inhaber, Konto   // kartesisches Produkt
```

23.6.2 where

- Einschränkung der Ergebnismenge

```
from Inhaber as i
where i.fristname = 'my_First_1'
```

```
from Inhaber
where firstname = 'my_First_1'
```

- Formulierung von Bedingungen
  - praktisch alle SQL bekannten Ausdrücke

Bedingungen	
Ausdruck	**Beschreibung**
and, or, not	logische Operatoren zur Verknüpfung von Ausdrücken
+, -, *, /	mathematische Operatoren
=, >=, <=, <>, !=, like	Vergleichsoperatoren
( )	Klammern zur Gruppierung von Ausdrücken
is null, is not null	Vergleiche auf Null
in, not in, between, is empty, is not empty, member of, not member of	Ausdrücke zum Umgang mit Mengen und Bereichen (Collections etc.)
case ... when ... then ... else ... end, case when ... then ... else ... end	Bedingungen mit Alternativen
||, concat(..., ...)	String Verknüpfungen
current_date(), current_time(), current_timestamp(), second(...), minute(...), hour(...), day(...), month(...), year(...)	Verwendung von Zeit und Datum
substring(), trim(), lower(), upper(), length(), locate()	Verwendung von Strings
abs(), sqrt(), bit_length(), mod()	sonstige mathematische Operatoren
str()	Umwandlung in einen String
cast(... as ...), extract(... from ...)	Casts zur Umwandlung in HQL-Datentypen

Bedingungen	
Ausdruck	Beschreibung
size(), minelement(), maxelement(), minindex(), maxindex()	sonstige Ausdrücke zum Umgang mit Collections
sign(), rtrim(), sin()	sonstige SQL-Funktionen

Beispiele:

```
from Inhaber i
where i.kontos.size > 3        // mit mehr als drei Konten
```

```
from Inhaber i
where i.firstname between 'A' and 'M'
```

```
from Inhaber  i
where i.firstname in ('Markus', 'Martin')
```

```
from Konto k
where k.name like '%konto%'
```

23.6.3 order by

- Sortierung der Ergebnismenge
  - nur über die aktuelle Klasse (Datenbank Tabelle)

```
from Inhaber
order by lastname
```

```
from Inhaber
order by lastname asc, firstname desc
```

23.6.4 Verbundoperationen

- liefern Kombinationen der beteiligten Entities
  - inner join / join
    - 1:1-Zuordnung. Elemente ohne Zuordnung sind ausgeschlossen
  - left outer join / left join
  - 1:1-Zuordnung. inklusive Elemente ohne Zuordnung der linken Tabelle
  - right outer join / right join

    - inklusive Elemente ohne Zuordnung der rechten Tabelle
- full join
    - kartesisches Produkt

```
select i, konto              // alle Inhaber mit Konten
from Inhaber i
inner join i.konten konto
```

```
select i, k
from Inhaber i
inner join i.konten k
with k.name like '%konto%'
```

```
from Verwalter v               // mit fetch Strategie
left join fetch v.konten
```

- Fetch-Joins
- erlaubt Abfragen von Beziehungen und Collections inklusive der 'Parent Objects'
- überschreibt outer-join und lazy Deklaration für Beziehungen und Collectionen
- Einschränkungen von Fetch-Joins
    - Ergebnismenge kann nicht iteriert werden
    - with - Konstrukt nicht geeignet
    - setMaxResults() / setFirstResult() kann nicht verwendet werden
        - Duplikate der Parents sind möglich

23.6.5 select

- definiert Ergebnismenge der Objekte und Attribute

```
select k                     // Instanzen der FROM Entities
from Inhaber i               // also Konto-Instanzen
inner join i.konten k
```

```
select i, k                  // Liste von Object[]
from Inhaber i               // Object[0] ist Inhaber Object
inner join i.konten k        // Object[1] ist Konto Object
```

```
select k, k.name             // Liste von Object[]
from Inhaber i               // Object[0] ist Inhaber Object
inner join i.konten k        // Object[1] ist String Object
```

```
select new list(i, k)          // List als Ergebnistyp
from Inhaber i
inner join i.konten k
```

```
select new InhaberKonto(i, k)  // typisiertes Java Object
from Inhaber i                 // InhaberKOnto als Ergebins
inner join i.konten k          // (selbst definierte Javaklasse)
```

23.6.6 Aggregat-Funktionen

- zur Konsolidierung oder Verdichtung der Ergebnismenge
- distributive Funktionen
  - sum( ), min( ), max( ), count(*), count( ), count(distinct ...), count(all ...)
- algebraische Funktionen
  - avg( )

```
select count(*)            // Anzahl aller Inhaber
from Inhaber
```

```
select count(distinct i.firstname)
from Inhaber i             // Anzahl aller eindeutiger Namen
```

23.6.7 group by

- Gruppieren von Aggregat-Funktionen

```
select v.name, count(k)    // Liste aller Verwalter
from Konto k               // mit Anzahl Konten
inner join k.verwalter v
group by v.name
from Inhaber
```

23.6.8 Polymorphe Abfragen

- SQL kennt das Konzept der Vererbung nicht
- JPQL unterstüzt polymorphe Abfragen
  - liefert Ergebnismenge von Entity-Instanzen einer gemeinsamen Vererbungshierarchie

```
from Konto                 // liefert auch
                           // Instanzen von Subklassen
```

```
from java.lang.Object      // alle persistenten Objekte
```

23.6.9 Subqueries

- falls die darunterliegende Datenbank Subqueries verwendet

```
from Konto kavg                 // Konten die mehr Saldo als
where kavg.saldo >              // das Durchschnittskonto haben
   ( select avg(k.size)
     from Konto k )
```

```
from Konto k                    // Wertemenge > 1
where not (k.nr, k.name) in
   ( select pk.saldo
     from Privatkonto pk )
```

24 Datentypen

- Datenbankzugriffe resultieren in Performance-Probleme
  - geeignete Fetching-Strategie wählen
  - geeigneter Typ wählen

24.1 Fetching-Strategien

JPA Spezifikation definiert zwei Fetching-Strategien:

- Lazy Loading
  - Daten erst laden wenn ein Zugriff auf das Attribut oder das Objekt erfolgt
  - Persistence Provider ist nicht verpflichtet Lazy Loading zu unterstützen
  - Umfang einer Umsetzung von Lazy Loading ungenau definiert
- Eager Loading
  - Daten sofort vollständig laden

Fetching-Strateg bei Attributen

```
/**
 * @module jee.2007.Metzler
 * @exercise jee251.server.Inhaber.java
 */
@Table(name = "T_Inhaber")
@Entity
public class Inhaber implements Serializable {
  private Long    id;
  private String  firstname;
  private String  lastname;
  @Basic(fetch = FetchType.LAZY)
  private String  email;
  private Address address;
```

Fetching-Strateg bei Beziehungen

```
  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
  private Address address;
```

- JPA unterstützt Lazy Loading
- Entity muss vom EntityManager verwaltet werden
- Zugriff auf nicht geladenes Attribut einer detached Entity wirft Exception

Annotation @Basic			
Parameter	**Beschreibung**	**Typ**	**Default**
fetch	definiert Ladeverhalten des Attribut	FetchType	EAGER
optional	erlaubt Null-Werte	Boolean	true

Annotation @Basic			
Parameter	Beschreibung	Typ	Default
	hat für primitive Java Typen keine Wirkung		

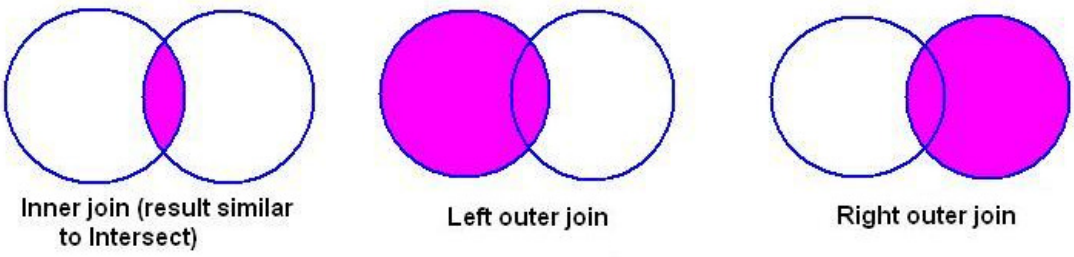
Fetching-Strategie

Fetching	
Annotation	Defaultwert
@Basic(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@OneToOne(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@ManyToOne(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@OneToMany(fetch = FetchType.EAGER/LAZY)	FetchType.LAZY
@ManyToMany(fetch = FetchType.EAGER/LAZY)	FetchType.LAZY

JPQLFetch Joins

- Spezifikation definiert für EJB QL-Abfragen Fetch Joins
- Beziehungen zwischen Entities bei Ausführung der Abfrage laden
- als Inner Join implementiert
- Loading Strategie (EAGER / LAZY) anwendbar
- Hibernate bietet zusätzliche Optionen:
  - Batch-Fetching
  - Subselect-Fetching

Join Strategien



Fetch Join Syntax

```
from [left [outer]] / [inner] join fetch association
```

- Fetch Join verwendet Fetch Strategien
  - Collections von Entities werden bei Bedarf der Ergebnismenge hinzugefügt
  - effizientes Laden von Entities und deren Abhängigkeiten

24.2 grosse Objekte

- umfangreiche Daten (Streams) werden in spezielle Datentypen abgelegt
  - meistens generische Objekte oder Strings
- @Lob deklariert Attribut als **L**arge **OB**ject
  - Datenbank ist verantwortlich für die Abbildung

```
@Basic
@Lob
private String grossesAttribut;
```

- @Lob wird oft im Zusammenhang mit @Basic verwendet, um LazyLoading für Large Objects zu definieren.

24.3 Datums- und Zeitwerte

- Datums- und Zeitwerte sind für die Speicherung in der Datenbank nicht vordefiniert
  - java.util.Calender
  - java.util.Date
- @Temporal definiert Datenbankrepräsentation
  - DATE entspricht java.sql.Date
  - TIME entspricht java.sql.Time
  - TIMESTAMP entspricht java.sql.Timestamp

Annotation @Temporal			
Parameter	**Beschreibung**	**Typ**	**Default**
TemporalType	definiert Speichertyp	DATE TIME TIMESTAMP	TIMESTAMP

```
@Temporal(TemporalType.DATE)
private java.sql.Date startDatum;
```

24.4 Aufzählungen

- Java SE 5.0 kennt Aufzählungstyp ENUM
- @Enumerated deklariert Aufzählungstyp und definiert Art der Persistierung
  - ORDINAL : Wert wird als Zahl beginnend mit 0 gespeichert
  - STRING : Wert wird als String gespeichert

Annotation @Enumerated			
Parameter	Beschreibung	Typ	Default
EnumType	definiert Art der Persistierung	ORDINAL STRING	

```
public enum KontoTyp {
  PRIVATKONTO, SPARKONTO, ANLEGEKONTO
}
```

```
@Enumerated(EnumType.ORDINAL)
private KontoTyp      kontoTyp;
```

D Best Practice

25 Design Patterns

25.1.1 Zweck

- Pattern heisst auf Deutsch übersetzt Muster oder Beispiel
- Design Patterns stellen wiederverwendbare Lösungen zu häufig auftretenden Problemen in der Software Entwicklung dar
- Design Patterns kommen aus der Praxis und wiederspiegeln die Erfahrung von Experten
- Die Verwendung solcher Muster erhöht die Qualität der Software und erleichtert die Kommunikation zwischen den einzelnen Entwicklern

25.1.2 Geschichtliches

- Idee stammt ursprünglich aus der Architektur
- 1977 hat Christopher Alexander ein Buch über die Verwendung von Mustern in der Architektur geschrieben
- Ward Cunningham und Kent Beck nahmen 1987 die Idee von Alexander auf und entwarfen fünf Patterns für die User-Interface Programmierung
- Durchbruch der Software Patterns erst in den 90er Jahren
- 1994 wurde das Buch Design Patterns von den Autoren Erich Gamma, Richard Helm, John Vlissides und Ralph Johnson geschrieben
- Design Patterns entwickelte sich zu einem der Einflussreichsten Computer Bücher überhaupt und ist auch unter dem Begriff Gang of Four bekannt.

25.2 JEE Pattern

Im Zusammenhang mit JEE Applikationen sind spezielle Anforderungen erwünscht wie:

- Transparenz
- Stabilität
- Performance

- JEE Entwurfsmuster stellen Idioms dar
  - also programmiersprache- und oft auch releaseabhängige Lösungen

Die diskutierten Idioms sind nicht EJB spezifisch, sondern gelten im Umgang mit Serverkomponenten.

25.2.1 Service Locator

- Suche und Erzeugung von EJB ist zeitintensiv und fehleranfällig
  - Namensdiensten (JNDI)
  - Komponentenverteilung (RMI-IIOP)
  - Factorycaching (Home Interface)

- Service Locator Idiom stellt ein Entwurfsmuster zur Kapselung der Technologie und eine Performancesteigerung zur Verfügung.

Problem:

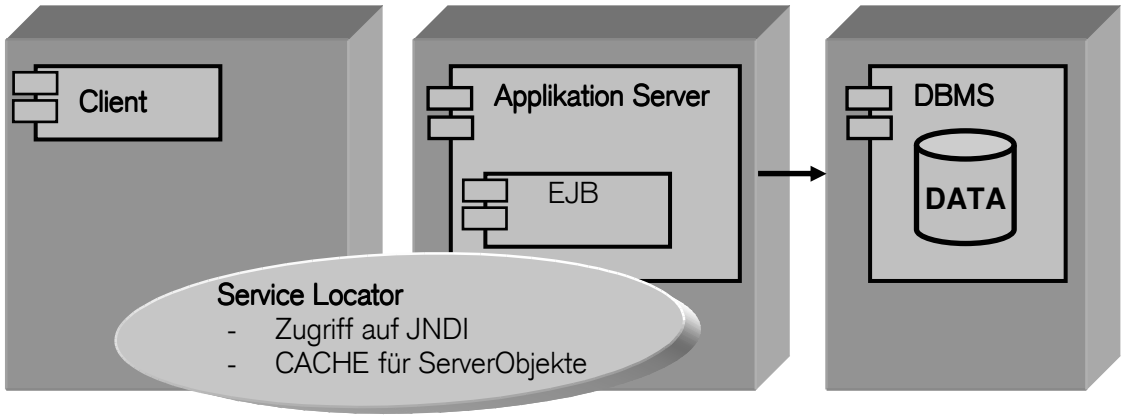
Bei einem Zugriff auf eine EJB muss zuerst ein Namensdienst initialisiert und die Verbindung zum Server hergestellt werden. Dazu werden produktspezifische Informationen benötigt. Die meisten JNDI Implementierungen basieren auf CORBA-Technologie, die eine Interpretation des Servers verlangt (z.B: "iiop://localhost:3700"). Dies kann bis zu mehreren Sekunden in Anspruch nehmen, da der Name zuerst von einem DNS aufgelöst werden muss.

Lösung:

Das Ermitteln von Dienstleistungen (z.B. JNDI Context) wird in den Locator verlagert. Dadurch verschwindet das sich wiederholende Downcasting über die narrow Methode von PortableRemoteObject im Service Locator.

Die folgenden Anforderungen werden an den Service Locator gestellt:

- bereits instanziierte Instanz des Home Objektes und Ressourcen werden gecached
- Handels (robuste Referenzen) werden gecached
- JNDI Technologie ist transparent für den Client



Implementation:

Der Service Locator kapselt die Zugriffe auf das JNDI und stellt ein Cache für die Serverschnittstellenobjekte zur Verfügung. Der Service Locator wird meist als Singelton

implementiert, damit auch nur tatsächlich eine Instanz des `InitialContext` pro VM existiert.

ServiceLocator to cache HomeObject (Ausschnitt)

```
public EJBHome getHome(String name,boolean cached) throws
Exception{
    Object temp = null;
    if(cached) temp = this.homeCache.get(name);
    EJBHome home = null;

    if(temp !=null && temp instanceof EJBHome){
      return (EJBHome)temp;
    }else{
      Object ref = context.lookup(name);
      EJBMetaData meta = ((EJBHome)ref).getEJBMetaData();
      home = (EJBHome) PortableRemoteObject.narrow(
              ref,meta.getHomeInterfaceClass());
      if(!meta.isSession() || meta.isStatelessSession())
        this.homeCache.put(name,home);
    }
    return home;
  }

  public static ServiceLocator getInstance() throws Exception{
    if(instance==null) instance = new ServiceLocator();
    return instance;
  }


  public EJBHome getHome(String name) throws Exception{
    return getHome(name,true);
  }

  public static void main(String[] args) throws Exception{
    Hashtable hash = new Hashtable();
      hash.put("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
      hash.put("java.naming.provider.url",
                 "iiop://localhost:3700");
    }
  }
```

ServiceLocatorClient

```
public class ServiceLocatorClient {
  public static void main(String args[]) throws Exception {
    HomeFinder finder = ServiceLocator.getInstance();
    EJBHome home = finder.getHome("Test");
    System.out.println("HomeClass " + home.getClass().getName());
    TestHome testHome = (TestHome) home;
    Test test = testHome.create();
  }
}
```

25.2.2 Business Delegate

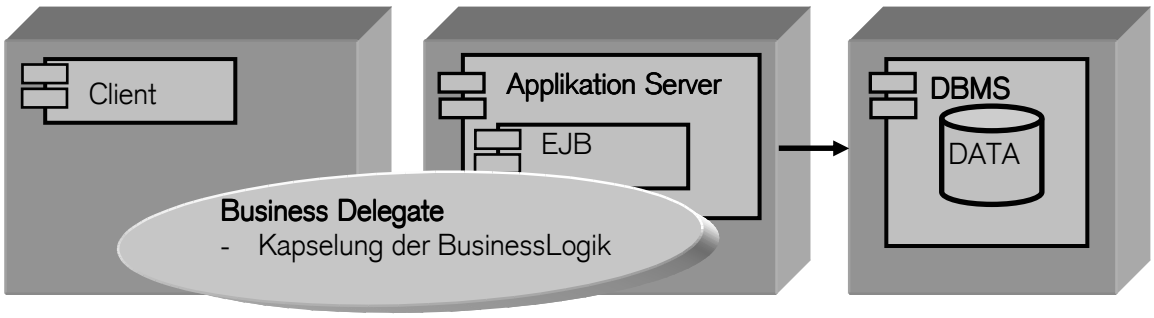
- Remotezugriff erfordert die Implementation entsprechender Technologie (z.B. RMI-IIOP)
- Client ist nicht an Technologie, sondern an Geschäftslogik interessiert
- Business Delegate Idiom ermöglicht Kapselung der Technologie
  - Vereinfachung der Schnittstelle zur Geschäftslogik

Problem:

Bei einem Zugriff auf die Geschäftslogik in einem EJB-Kontainer ist oft ein Remotezugriff notwendig. Ein Remotezugriff ist jedoch viel komplexer und zeitintensiver als ein Lokalzugriff. Zusätzlich müssen noch Exceptions (`HostNotFoundException`, `StubNotFoundException`, etc.) abgefangen werden.

Lösung:

Die einzelnen Beans werden über eine zentrale Session Bean angesprochen. Diese hält die Instanzen der verschiedenen Beans und stellt diese dem Client in einer Schnittstelle zur Verfügung und kann auch Informationen an die anderen Beans weitergeben. Der Client produziert "Events" die vom Business Delegate Idiom "verteilt" werden und hat einen einfachen Zugriff auf die Geschäftslogik zur Verfügung.



Implementation:

Das Business Delegate stellt die Schnittstelle zwischen Client und Geschäftslogik zur Verfügung. Dabei kapselt dieses Idiom die Zugriffslogik. Das Business Delegate wird vom Client instanziiert und benutzt. Die notwendigen Services werden durch den Service Locator beschafft.

Business Delegate Interface (wird dem Client zur Verfügung gestellt)

```
public interface BusinessDelegateIF {
  public CustomerVO getCustomer(String id);
  public CustomersVO[] getAllCustomerForZip(String zipPrefix);
  public CustomersVO[] getAllCustomers();
  ...
}
```

Business Delegate (Kapselung der BusinessMethoden)

```
public CustomerVO getCustomer(String id){
  CustomerVO customer = null;
 try{
   customer = this.customer.findByPrimaryKey(id);
 }catch(RemoteException e){
   throws new ProblemWithBusinessMethodeException(this.e);
 }
 return customer:
}
...
```

Das Business Delegate Idiom ist in EJB 3.0 obsolet

25.2.3 Value Object

- limitierender Faktor in JEE Applikationen ist die Datenübertragung über das Netzwerk
  - von der Data Tier bis zur Client Tier und umgekehrt
- Value Object Idiom verbessert die Performance und minimiert die Remote Methodenaufrufe

Problem:

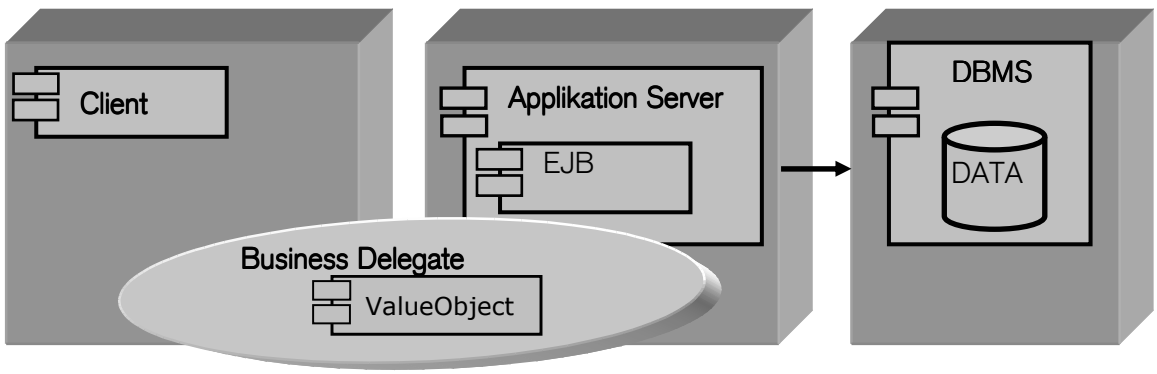
Der Zugriff auf ein EJB erfolgt oft über Remote Methodenaufrufe. Oft wird jedoch nur eine Teilmenge der benötigten Daten übermittelt (z.B: getter / setter Methoden). Sinnvoller ist es die gesammte benötigte Datenmenge (Datensatz) in einem Methodenaufruf zu übermitteln.

Enterprise Beans haben in der Regel ganze Gruppen von Attributen, die vom Client in der Gruppe gebündelt abgerufen und gesetzt werden können. Auf diese Weise muss nicht für jedes Attribut einzeln eine Methode zum Setzen und Abfragen aufgerufen werden. Der Client erhält die Attribute in einem Value Object gebündelt.

Lösung:

Die vielen "get"-Aufrufe werden bereits im EJB Container zusammengefasst und dem Client als Objekt übermittelt. Dieses Kontainer-Objekt besteht nur aus Zugriffslogik und keiner Geschäftslogik, ist somit ein Value Objekt und entspricht folgenden Anforderungen:

- minimieren der Remote Methodenaufrufe an den Server
- Die Geschäftsdaten werden als Objekt (Value Object) an den Client übergeben
- das Auslesen der Objekt Attribute findet clientseitig statt



Implementation:

Das Value Objekt wird durch eine serialisierbare Klasse repräsentiert, die lediglich zu Datenhaltungszwecken dient. Das Value Object wird auf der entfernten Maschine erzeugt und gefüllt, über das Netzwerk transferiert und lokal ausgelesen.

Value Object als serialisierbare Klasse

```
public class PersonVO implements Serializable{
  public String id        = null;
  public String fname     = null;
  public String name      = null;
  public PersonVO(){}

  public PersonVO(String id, String fname, String name) {
    this.init(id,fname,name);
  }
  protected void init(String id, String fname, String name);
    this.id = id;
    this.fname = fname;
    this.name = name;
  }
  public PersonVO(PersonVO person){
    this.init(person.getId(),person.getFname(),person.getName());
  }
  public String getId(){ return this.id; }
  public String getFname(){ return this.fname; }
  public String getName(){ return this.name; }
  public void setId(String id){this.id = id;}
  public void setFname(String fname){this.fname = fname;}
  public void setName(String name){this.name = name;}
}
```

Grundsätzlich können Value Objects von folgenden Komponenten erzeugt werden:

- Data Access Objects
- Session Façade (einer Session Bean, die den Zugriff auf ein Entity Bean managed)
- Business Delegate
- Session Bean
- Entity Bean
- Komponente der Präsentationsschicht

Es handelt sich also grundsätzlich um Komponenten, die in der Geschäftslogik oder in der Integrationsschicht liegen und in der Lage sind, benötigte Daten zu beschaffen. Meistens werden jedoch die Value Objects von den Entity Beans erzeugt, da diese die notwendige "Persistence Schicht" repräsentieren.

Die CMP Entity Bean der EJB 2.0 ist ein "Value Object"

Das Value Object Idiom ist in EJB 3.0 obsolet

25.2.4 Session Facade

- Geschäftslogik liegt typischerweise im EJB Container
  - durch Komponenten (EJBs) abgebildet
  - Client muss Zusammenhang der einzelnen Komponenten im Container kennen

- Session Facade Idiom kapselt die Kontainer Komponenten durch ein Session Bean

Problem:

Entity Beans bilden persistente Objekte ab, repräsentieren aber lediglich den Zustand der darunterliegenden Datenbank. Die EJBs sind in Bezug auf Wiederverwendbarkeit, Transaktionssteuerung sowie Zustandserhaltung nicht, oder nur mit sehr hohem Aufwand verbunden, vom Client trennbar.

Wenn ein Client auf verschiedene Entity Beans zugreift und verschiedene Methoden der Entity Beans aufruft, um eine Aufgabe zu erledigen, kann dies verschiedene Probleme mit sich bringen. Zum einen erhöht sich der Netzwerk Traffic, des weiteren wird die Workflow Logik auf den Client verlagert, zum anderen sind die Methodenaufrufe durch die client-seitige Inszenierung möglicherweise nicht in einen notwendigen Transaktionszusammenhang gestellt.

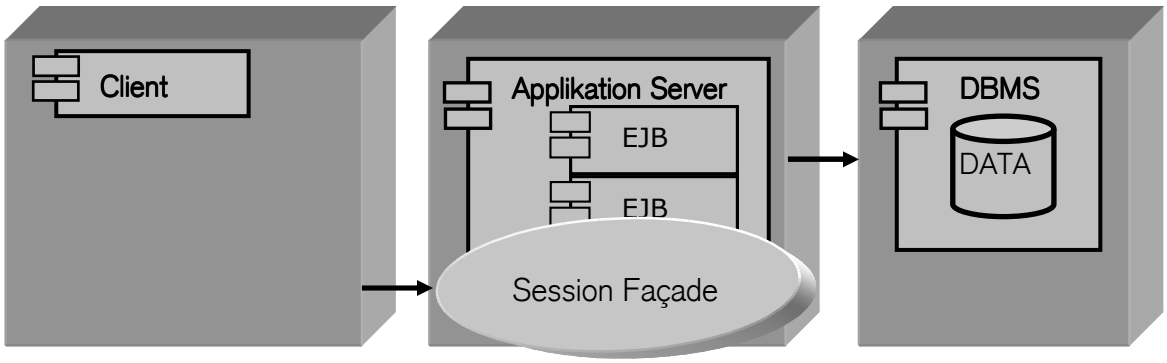
Die Entity Beans mit weiterer Processing Logik aufzublähen, ist nicht der richtige Lösungsansatz. Da die Entity Bean Schicht von der langfristigen Perspektive her aus verschiedenen Sichten und Aufgaben gesehen wird, wird der Einbau von Anwendungslogik zu Zwecken der Performance Verbesserung eher zu Wartungsproblemen führen.

Deshalb sollte die Anwendungslogik, die eine Kette von Aufrufen von Entity Bean Methoden widerspiegelt, in einer Session Bean implementiert werden. Der Client ruft dann **eine** Methode der Session Bean auf, um seine Aufgabe zu erledigen. Die Methode der Session Bean übernimmt die Aufrufe der Methoden der verschiedenen Entity Beans.

Lösung:

Was der Client möchte ist eine Schnittstelle, die ein Use Case abbildet. Dem Client kann es egal sein, wieviele Entity oder Session Beans diese Schnittstelle implementiert. Für den Client ist nur die korrekte Abbildung des Anwendungsfalles wichtig. Das Session Façade definiert eine neue "Schicht", die als Session Bean implementiert ist und folgende Anfgorderungen erfüllt:

- Abschirmung und Entkoppelung des Client vor Subsystemen (Entity Bean, etc.)
- Erhalt der Unabhängigkeit der Subsysteme
- Vereinfachung der Business Schnittstelle
- Implementierung von allgemeinen Diensten wie Caching, Authorisierung, etc.
- Zustandsverwaltung des Client
- Transaktionssteuerung



Implementation:

In einer zusätzlichen Schicht, implementiert mit einer Session Bean, werden diese Anforderungen untergebracht. Falls ein Caching gefordert ist, muss die Session Bean statefull sein. Diese EJB entspricht in ihrer Funktionalität dem klassischen GoF Façade Pattern. Sie bietet eine einfache Schnittstelle und verbirgt die Komplexität der Subsysteme vor dem Client.

Session Façade um einen Datensatz zu ändern; eine Transaktion wird gestartet (Ausschnitt)

```
public void updatePerson (PersonVO person) throws Exception{
  try{
    getPerson(person.getId()).remove();
    this.createPerson(person);
  }catch(Exception e){
    throw new Exception("PersonBean.updatePerson " + e);
   }
}
```

25.3 JEE Entwurfsmuster

JEE Idioms		
Entwurfsmuster	**Beschreibung**	**Bedeutung für EJB 3.0**
Service Locator	kapselt teure Technologien	relevant
Business Delegate	kapselt Businesslogik stellt vereinfachter Client Zugriff zur Verfügung	obsolet, da keine RemoteExeptions in BusinessInterface
Session Facade	kapselt die Kontainer Komponenten	durch die Spezifikation erzwungen
Value Object	kapselt Business Data	obsolet, Custom DTO relevant
Message Façade **Service Actvator**	nutzt asynchrone Kommunikation	relevant
EJB Command	beschreibt Möglichkeit zur Gestaltung der Session Bean - Entkopplung durch den Client	relevant
EJB Home Factory	Cache von entfernten Objekten	obsolet
Business Interface	stellt Konsistenz zwischen Beanklasse und Businessinterface sicher	obsolet
DTO Factory	Erzeugung eines DTO durch Factory	für Custom DTO relevant
Data Transfer **Hash Map**	generische Variante zum DTO	relevant
Value List Handler	serverseitiges Speichern von Ergebnismengen	relevant
Data Transfer Row Set	disconnected RowSet für Serveranwendung ohne Entity Beans	relevant
Data Access Object	kapselt den Datenbank Zugriff	relevant mit JPA jedoch leistungsfähiger
Version Number	zusätzliches Attribut kontrolliert Lost-Update-Problematik	in Spezifikation integriert
Muster zur Generierung von Primärschlüsselwerten	Generierung von Business Keys	in Spezifikation integriert

26 Programmiereinschränkungen (7 Regeln)

1 Threads
kein Thread Management oder Synchronisation in (EJB) Beans
- ist Aufgabe des Applikationsservers
- Bean = einzelner Thread, keine Kongruenz

2 statische Variablen
keine schreibenden Zugriffe auf Klassenvariablen durchführen
- verschiedene Prozesse bei Cluster-Deployment
- statische Attribute mit final deklarieren

3 AWT (Abstract Windwing Toolkit)
- kein stdin / -out verwenden
- Ausgabe- / Eingabestream sind nicht definiert
- Portabilierproblematik

4 Dateizugriff
kein direkter Zugriff auf Dateien oder Directories
- keine Verwendung von java.io-Package
- keine native Libraries
- nicht definierter Speicherort bei Cluster-Deployment

5 Socket-Listener
keine Beans an Sockets binden
- Kontainer steuert Bean Life Cycle, daher Bean nicht als Socket Server einsetzen
- Bean Instanzen dürfen Socket Clients sein

6 this
keine this – Referenz als Parameter oder Ereignis eines Methodenaufrufes
- SessionContext.getBusinessObject(Class businessInterface) benützen

7 Systemproperty
EJBs dürfen Laufzeitumgebung nicht beeinflussen
- keine Class-Loader erzeugen, abfragen
- JVM anhalten
- Security Manager installieren

27 Deployment Descriptoren

- Beschreiben und Bereitstellen der serverseitigen Komponenten
  - Annotationen (EJB 3.0)
  - Deployment Descriptoren (überschreiben Annotationen)

ejb-jar.xml Deployment-Descriptor

```
...
 <enterprise-beans>
  <entity>
    <ejb-name>Anwendung1EJB</ejb-name>
    <home>Anwendung1Home</home>
    <remote>Anwendung1</remote>
    <ejb-class>Anwendung1Bean</ejb-class>
    ...
  </entity>
  <session>
    <ejb-name>Anwendung2EJB</ejb-name>
    <home>Anwendung2Home</home>
    <remote>Anwendung2</remote>
    <ejb-class>Anwendung2Bean</ejb-class>
    ...
  </session>
  ...
 </enterprise-beans>
```

Verpackt in eine jar-Datei (EJB 2.0)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
EnterpriseJavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
```

Verpackt in eine jar-Datei (EJB 2.1)

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/JEE"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/JEE
                             http://java.sun.com/xml/ns/JEE/ejb-jar_2_1.xsd"
         version="2.1">
```

Verpackt in eine jar-Datei (EJB 3.0)

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
```

Zwischen den Versionen ist man von DTDs auf XML-Schema umgestiegen.

27.1 EJB Deployment Descriptors

- ejb-jar.xml
- jboss.xml
- application.xml
- application-client.xml
- jboss-client.jar
- jbosscmp-jdbc.xml

27.1.1 ejb-jar.xml

beschreibt Bean auf logischer Ebene

- Namen der Java-Klassen
- Namen der Attribute
- Transaktionsverhalten
- Security Mapping
- Spezifikation der Query-Methoden
- Resource Referenzen
- ejb-jar.xml ENC Elements (ENC = Enterprise Naming Context)

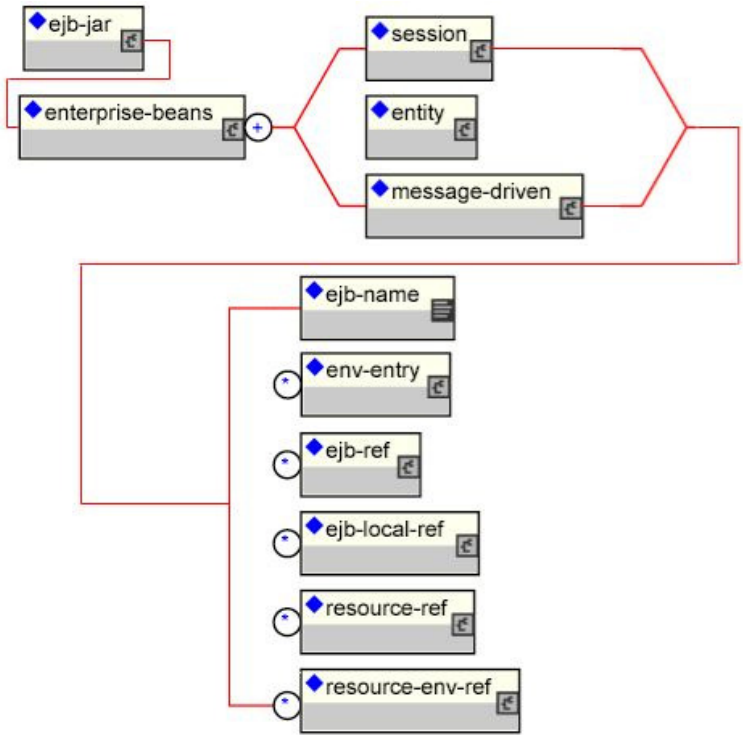
- Name: ejb-jar.xml
- Ort: META-INF-Verzeichnis der Jar-Datei
- Root-Element: ejb-jar

ejb-jar.xml Elementbeschreibung	
Element	**Beschreibung**
description?	Textbeschreibung der ejb-jar-Datei
display-name?	Name, der von Tools benutzt wird
small-icon?	Pfad, innerhalb der jar-Datei, für jpeg-Bilder (16x16)
large-icon?	Pfad, innerhalb der jar-Datei, für jpeg-Bilder (32x32)
enterprisebean	nähere Beschreibung/ Definition der Beans
relationships?	definiert CMP Beziehungen
assembly-descriptor?	definiert Transaktionen und Security
ejb-client-jar?	definiert JAR-Datei, die Stubs und Interfaces für Remote-Clients enhält

ejb-jar.xml Session Bean Elementbeschreibung	
Element	**Beschreibung**
ejb-name	Spitzname des Beans, dient als Referenz
ejb-class	Session-Bean Klasse + Package-Name
home?	Home-Interface
remote?	Remote-Interface
local-home?	Local-Home-Interface
local?	Local-Interface
session-type	Stateful | Stateless
transaction-type	Container | Bean
description?	Textbeschreibung der ejb-jar-Datei
display-name?	Name, der von Tools benutzt wird
small-icon?	Pfad, innerhalb der jar-Datei, für jpeg-Bilder (16x16)
large-icon?	Pfad, innerhalb der jar-Datei, für jpeg-Bilder (32x32)
env-entry*	Definition von Umgebungsvariablen
ejb-ref*	Definition von Referenzen zu anderen Beans
ejb-local-ref	Definition von Referenzen zu lokalen Beans
security-role-ref*	definiert Sicherheitsrollen
security-identity*	definiert wie Sicherheitskontexte ermittelt werden können
resource-ref*	definiert Referenzen zu Ressourcen (z.B. JDBC)
resource-env-ref*	bindet Ressourcen-Factories zu JNDI Namen

- assembly-descriptor
  - ermöglicht die Festlegung von Zugriffsberechtigungen, Transaktionen und das generelle Zusammenspiel von Beans einer Applikation

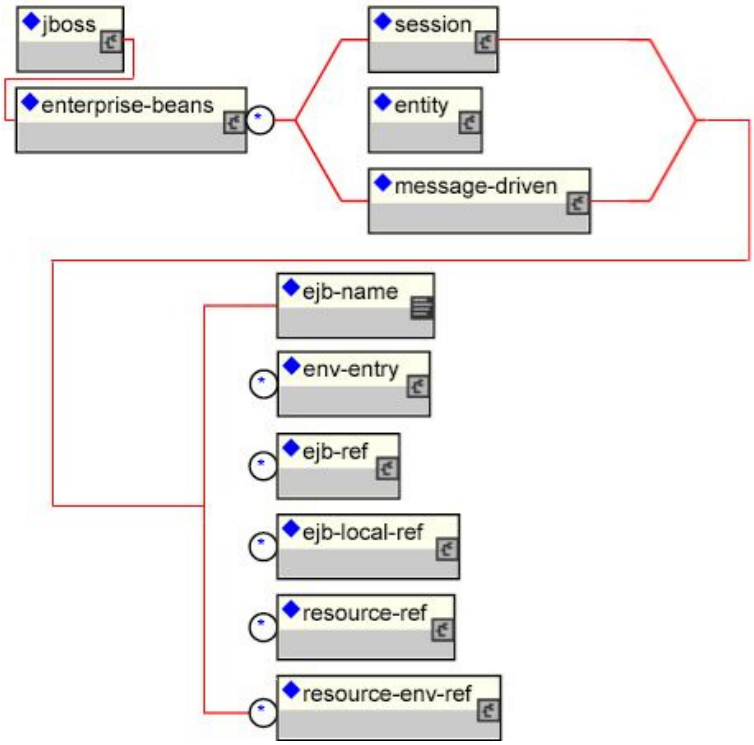
ejb-jar.xml Assembly-Descriptor Elementbeschreibung	
Element	**Beschreibung**
security-role*	Security-Rollen können definiert werden
method-permission*	spezifiziert für die einzelnen Methoden, die Zugriffsrechte
container-transaction*	Definition der Transaktionen, der einzelnen Methoden
exclude-list?	Liste von Methoden, die von niemandem aufgerufen werden dürfen



27.2 jboss.xml

bescheibt Informationen für den Application Server

- JNDI-Mapping: Name unter dem das Home-Interface gefunden wird
- Resource Referenzen



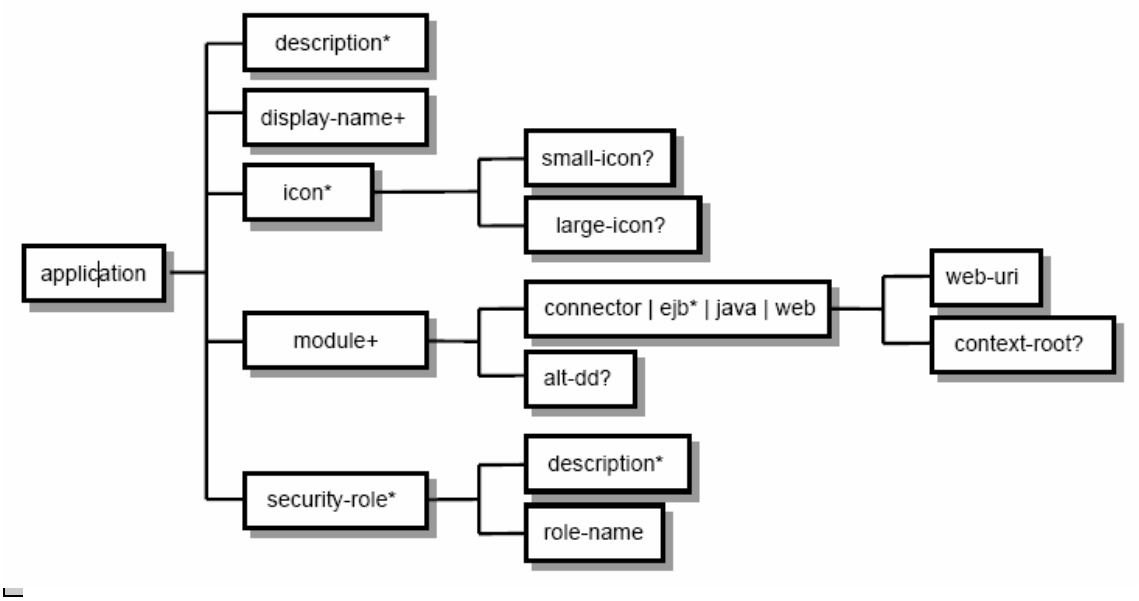
jboss.xml ENC Elements

- (ENC = Enterprise Naming Context)

```
<jboss>
 <enterprise-beans>
  <session>
   <ejb-name>moneyexchange</ejb-name>
   <jndi-name>ejb/moneyexchange</jndi-name>
   <resource-ref>
    <res-ref-name>jdbc/ExchangeTable</res-ref-name>
      <jndi-name>java:/mySQLDS</jndi-name>
   </resource-ref>
  </session>
 </enterprise-beans>
</jboss>
```

application.xml : beschreibt eine ganze Enterprise Applikation

- die graphische Repräsentation zeigt die Struktur des JEE application.xml Schemas

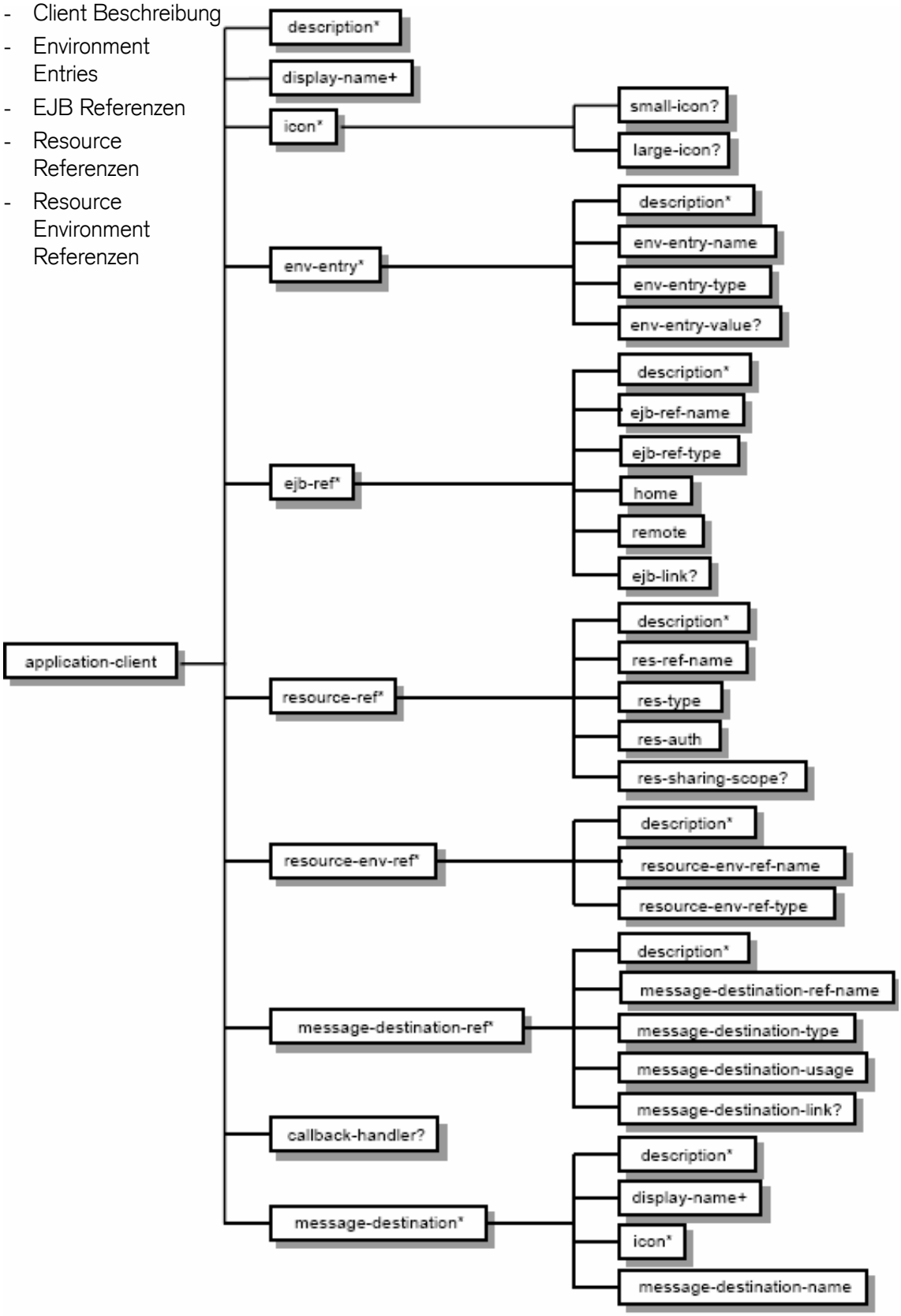



```
<?xml version="1.0" encoding="UTF-8"?>
<application
   xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:application=
      "http://java.sun.com/xml/ns/javaee/application_5.xsd"
   xsi:schemaLocation=
      "http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/application_5.xsd"
   id="Application_ID"
   version="5">
   <module>
      <ejb>jee031.jar</ejb>
   </module>
</application>
```

27.2.1 application-client.xml

beschreibt First Tier Client Program

- Client Beschreibung
- Environment Entries
- EJB Referenzen
- Resource Referenzen
- Resource Environment Referenzen



27.2.2 jboss-client.xml

bescheibt Informationen für den Application Server

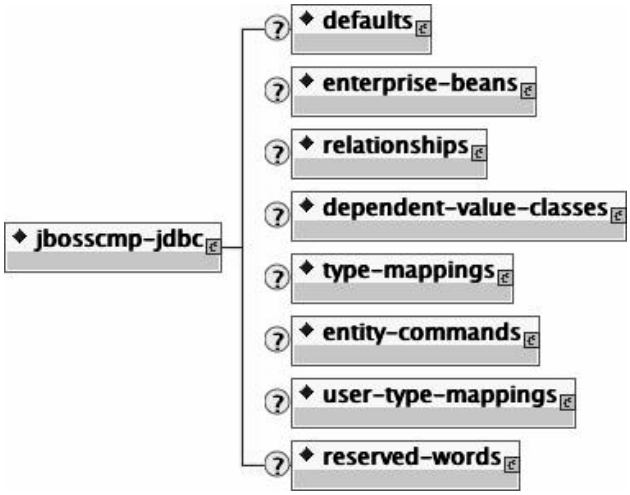
- Versionsnummern,....
- JNDI-Mapping: Name unter dem das Home-Interface gefunden wird
- Resource Referenzen

```
<jboss-client>
   <jndi-name>MoneyExchangeClient</jndi-name>
   <ejb-ref>
      <ejb-ref-name>ejb/MoneyExchange</ejb-ref-name>
      <jndi-name>moneyexchange</jndi-name>
   </ejb-ref>
</jboss-client>
```

27.2.3 jbosscmp-jdbc.xml

beschreibt O/R-Mapping für die Datenbank (applikationsserverspezifisch)

- Datasource: Information für den DB-Connect im Application Server
- jndi-Name: Name unter dem das Home-Interface gefunden wird




```
<jbosscmp-jdbc>
   <defaults>
      <datasource>java:/DefaultDS</datasource>
      <datasource-mapping>MySQL</datasource-mapping>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
   </defaults>
   <enterprise-beans>
      <entity>
         <ejb-name>Product</ejb-name>
         <table-name>ProductTable</table-name>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

E Aufsetzen und Konfiguration der Entwicklungsumgebung

28 Entwicklungstools für JEE

Der Markt bietet neben kommerziellen AS Produkten, die mehr oder weniger hohe Lizenzkosten beinhalten auch einige Open-Source Projekte an. Dieses Skript und die aufgeführten Beispiele fokussieren bewusst auf OS Tools um das Testen der Lerninhalte so einfach wie möglich zu gestalten.

Die folgenden Installationsanleitungen und die Beschreibung der Entwicklungs-Tools basieren auf den im Skript aufgeführten Beispielsübungen und deren Lösungen.

Die Beispiele in diesem Skript basieren auf den nachfolgend installierten Versionen. Neue Releases sind meist "down-kompatible", eventuelle Anpassungen an den Source Code der Beispiele müssen gemäss den entsprechenden Handbüchern gemacht werden.

28.1 Java J2SE / JEE Installation

Neben einem JDK (z.B. JDK1.5) muss für die Entwicklung von JEE Applikationen die JEE Umgebung von Sun™ installiert werden (z.B. JEESDK1.4). Beide Dateien, sowie etliche Dokumentationen können unter http://java.sun.com bezogen und lokal installiert werden.

J2SE installieren:

ausführen:	`jdk-1_5….exe`
von:	`[CD]\install\j2se` oder von http://java.sun.com

29 Eclipse als Entwicklungsumgebung

Eclipse.org ist die Plattform für mehr Hintergrundinformationen betreffend der Eclipse Entwicklungsumgebung:

- Basiert auf Initiative von IBM
- Mitarbeit von Borland, webgain, GNX, serena, Rational, redhat, SuSe, TogetherSoft, ...
- Entwickelt von mehr als 1200 Software Entwicklern von mehr als 150 führenden Software Herstellern aus mehr als 60 verschiedenen Ländern

(einige) Plattform Komponenten:

- Ant (integration of Ant scripting)
- Compare (universal compare facility)
- Debug (generic execution and debug framework)
- Help (platform help system)

29.1 Installation

Die aktuelle Version von Eclipse kann als ZIP Datei von http://www.eclipse.org oder direkt unter http://mirror.switch.ch/mirror/eclipse/downloads/ bezogen werden und zu einem lokalen Verzeichnis (z.B. c:\jee) dekomprimieren (Verzeichnisstruktur sollte keine Leerzeichen enthalten, wie z.B. "Eigene Dateien").

Für die in diesem Script abgebildeten Beispiele wurde die jee-europa Version von Eclipse verwendet:

entpacken:	`eclipse-jee-europa-win32.zip`
von:	`[CD]\install\eclipse`
nach:	`c:\jee`

bevorzugte Plug-Ins installieren, z.B: XMLBuddy:

entpacken:	`xmlbuddy_2.0.72.zip`
von:	`[CD]\install\eclipse plugins`
nach:	`...\jee\eclipse\plugins`

weitere Eclipse Plug-Ins unter:

http://www.eclipseplugincentral.com

29.2 Eclipse starten

Im Insallationsverzeichnis auf das entsprechende Icon klicken.

30 JBoss AS

JBoss war ursprünglich als EJB Kontainer konzipiert, wurde aber zu einem kompletten Applikations Server für JEE Applikationen weiterentwickelt.

Entstanden 1999 - JBoss ist das Produkt einer OpenSource Developer Community die das erklärte Ziel hatte, den besten JEE-compliant Applikationsserver auf dem Markt zu entwickeln

- 1000 Developer weltweit
- JBoss ist heute wahrscheinlich der verbreitetste EJB Server
- 4000 Downloads des EJB Servers pro Tag (mehr als manche kommerzielle Hersteller insgesamt haben)
- Die Distribution erfolgt unter LGPL License, d.h. JBoss kostet keine Lizenzgebühren.
- JBoss hat sich als eine vitale Alternative zu überpreisten EJB Servern im kommerziellen Umfeld entwickelt.

30.1 Installation

Die aktuelle Version von JBoss kann als ZIP Datei von http://www.jboss.org bezogen werden und zum lokalen Verzeichnis `C:\jee` dekomprimieren (Verzeichnisstruktur sollte keine Leerzeichen enthalten, wie z.B. "Eigene Dateien")

entpacken:	`jboss-4.2.0.GA.zip`
von:	`[CD]\install\jboss` oder `http://www.jboss.org`
nach:	`...\jee\`

`JAVA_HOME` Umgebungsvariable als Benutzer- oder Systemvariablen entsprechend der installierten JDK Version definieren/ergänzen:

30.2 JBoss AS starten / stoppen

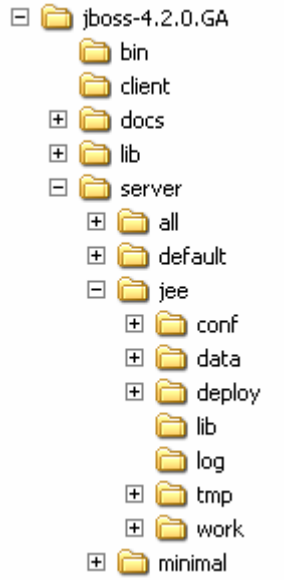
- das `run.bat` Skript in `\bin` Verzeichnis der JBoss Installation ausführen
- mit `http://localhost:8080` den JBoss Server testen.
- CTRL+C, `shutdown.bat` oder über die Management Konsole `http://localhost:8080/jmx-console` kann der JBoss Applikations Server gestoppt werden.

30.3 JBoss AS administrieren

- über JMX Konsole : http://localhost:8080/jmx-console/
- über WEB Konsole : http://localhost:8080/web-console/

30.4 JBoss Verzeichnisstruktur

Das Installationsverzeichnis enthält fünf Ordner:



- **`bin`** Skript Files
- **`client`** JAR Dateien für Client Applikationen, `jbossall-client.jar` enthält sämtliche Libraries
- **`docs`** enthält XML DTD für JBoss Referenzen sowie example JEE Connector Architecture Dateien für das Anbinden von externen Ressourcen (z.B. MySQL, etc.)
- **`lib`** JBoss microkernel JAR Libraries
- **`server`** die verschiedenen server Konfigurationen
  - **`all`** startet alle zur Verfügung stehenden Dienste wie: RMI-IIOP, Clustering und WEB-Service
  - **`default`** started in der standart Konfiguration
  - **`jee`** (erstellter) benutzerspezifischer Server für die EnterpriseEducation Beispiele. (als Kopie des **`default`** Ordner)
  - **`minimal`** minimum Requirement für den JBoss Server mit: JNDI und URL Deployment Scanner ohne: WEB- oder EJB Container, ohne JMS

Der entsprechende Server ist, wenn gestartet (z.B. `run -c default`), auch zugleich das entsprechende Verzeichnis, indem alle Deployments stattfinden und alle LOGs geschrieben werden. Die Verzeichnisstruktur beschreibt:

- **`conf`** enthält die jboss-service.xml Datei, die den core Service spezifiziert
- **`data`** Verzeichnis für die Hypersonic Datenbank
- **`deploy`** HOT-Deployment Verzeichnis für JAR, WAR und EAR Dateien
- **`lib`** JAR Dateien für die Server Konfiguration und Ressourcen z.B. "fremde" JDBC Drivers, etc.
- **`log`** Verzeichnis für die Logging Informationen
- **`tmp`** Temporäres Verzeichnis für Deployment, etc.
- **`work`** JSP-Kompilationsverzeichnis für Tomcat

30.5 Benutzerspezifischer Server einrichten

Als Default wird der "default" Server gestartet. Alle serverspezifischen Informationen sind im entsprechenden Verzeichnis. Mit den Beispielsapplikationen benutzen wir mySQL als Datenbank. Um einen entsprechenden benutzerspezifischen Server "jee" für die EnterpriseEducation Beispiele zu definieren sind die folgenden Schritte massgebend:

[JBoss_Home]\server\default Verzeichnis kopieren zu "jee"

kopieren:	Verzeichnis
von:	`[JBoss_Home]\server\default`
nach:	`[JBoss_Home]\server\jee`

Der JDBC Treiber wird als "MySQL Connector" auf der offiziellen Website von MySQL angeboten, wobei die im Root des heruntergeladenen TAR- oder ZIP Verzeichnis befindliche JAR-Datei von Bedeutung ist. Diese im JAR-Format befindliche Java-Bibliothek muss in den `/lib` Ordner der JBoss Installation kopiert werden - damit erhält der Application Server den Zugriff darauf, und die Voraussetzung, mit dem MySQL Server kommunizieren zu können.

JDBC Driver mysql-connector-java-5.0.6-ga-bin.jar oder aktuellen Connector/J von http://www.mysql.com ins Verzeichnis [JBoss_Home]\server\ee\lib kopieren:

kopieren:	`mysql-connector-java-5.0.6-bin.jar`
von:	`[CD]\install\mysql` `http://www.mysql.com`
nach:	`[JBoss_Home]\server\jee\lib`

Die logische Repräsentation der MySQL Datenbank innerhalb der JEE Programme wird nicht, wie vermutet, durch ein direktes Verwenden des JDBC-Treibers geschaffen, sondern über eine abstrakte Datenbankschnittstelle, DataSource genannt. Durch eine neu anzulegende Konfigurationsdatei im `/deploy` Ordner der JBoss Installation können DataSources zentral durch den Namens- und Verzeichnisdienst verwaltet werden.

In `/docs/examples/jca/mysql-ds.xml` befindet sich eine Vorlage, die den Verhältnissen entsprechend angepasst werden muss.

mySQL DataSource Konfigurationsdatei **`mysql-ds.xml:`**

kopieren:	`mysql-ds.xml`
nach:	`[JBoss_Home]\server\jee\deploy`
anpassen:	benutzerspezifisch konfigurieren

```
<datasources>
  <local-tx-datasource>
    <jndi-name>
       jdbc/account
     </jndi-name>
     <connection-url>
       jdbc:mysql://localhost:3306/jee
     </connection-url>
     <driver-class>
       com.mysql.jdbc.Driver
     </driver-class>
     <user-name>
       root
     </user-name>
    <password></password>
...
```

30.6 Benutzerspezifischer JBoss-Server starten

Um den neu erstellten Server "ee" zu starten muss vorgängig mySQL installiert, sowie die entsprechende Datenbank "ee" mit den entsprechenden Benutzerrechten erstellt sein.

- cd\ C:\ee\ jboss-4.0.4.GA\bin\
- run -c jee

Das entsprechende Script um den Server zu stoppen ist: `shutdown -S`

JBoss Log Console

```
[Server] Starting JBoss (MX MicroKernel)...
[Server] Release ID: JBoss [Trinity] 4.2.0.GA (
  build: SVNTag=JBoss_4_2_0_GA date=200705111440)
[Server] Home Dir: S:\jee\jboss-4.2.0.GA
[Server] Home URL: file:/S:/jee/jboss-4.2.0.GA/
[Server] Patch URL: null
[Server] Server Name: jee
[Server] Server Home Dir: S:\jee\jboss-4.2.0.GA\server\jee
[Server] Server Home URL: file:/S:/jee/jboss-4.2.0.GA/server/jee/
[Server] Server Log Dir: S:\jee\jboss-4.2.0.GA\server\jee\log
[Server] Server Temp Dir: S:\jee\jboss-4.2.0.GA\server\jee\tmp
[Server] Root Deployment Filename: jboss-service.xml
[ServerInfo] Java version: 1.5.0_07,Sun Microsystems Inc.
[ServerInfo] Java VM: Java HotSpot(TM)
  Client VM 1.5.0_07-b03,Sun Microsystems Inc.
[ServerInfo] OS-System: Windows XP 5.1,x86
[Server] Core system initialized
[WebService] Using RMI server codebase: http://127.0.0.1:8083/
...
[testTopic] Bound to JNDI name: topic/testTopic
[testQueue] Bound to JNDI name: queue/testQueue

[Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-8080
[AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
[Server] JBoss (MX MicroKernel) [4.2.0.GA
  (build: SVNTag=JBoss_4_2_0_GA date=200705111440)]
  Started in 17s:609ms
```

31 mySQL Datenbankserver installieren und Datenbank erstellen

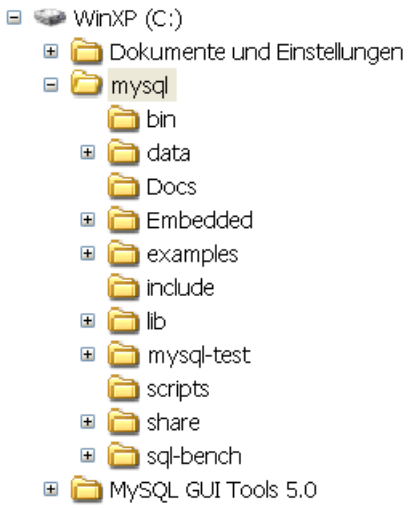
mySQL ist eine der verbreitetsten Open Source Datenbanken und wird von vielen Organisationen kommerziell benutzt.

31.1 Installation

entpacken:	`mysql-noinstall-5.0.41-win32.zip`
von:	`[CD]\install\mysql` oder `http://www.mysql.com`
nach:	`c:\` **(entziptes Verzeichnis umbenennen zu mysql!)**

Without installer (unzip in C:\)

entziptes Verzeichnis umbenennen zu mysql



mySQL kann bei der Installation und beim Starten des DBMS Dienstes Probleme verursachen wenn diese mySQL Distribution nicht im c:\ (root) Verzeichnis liegt:

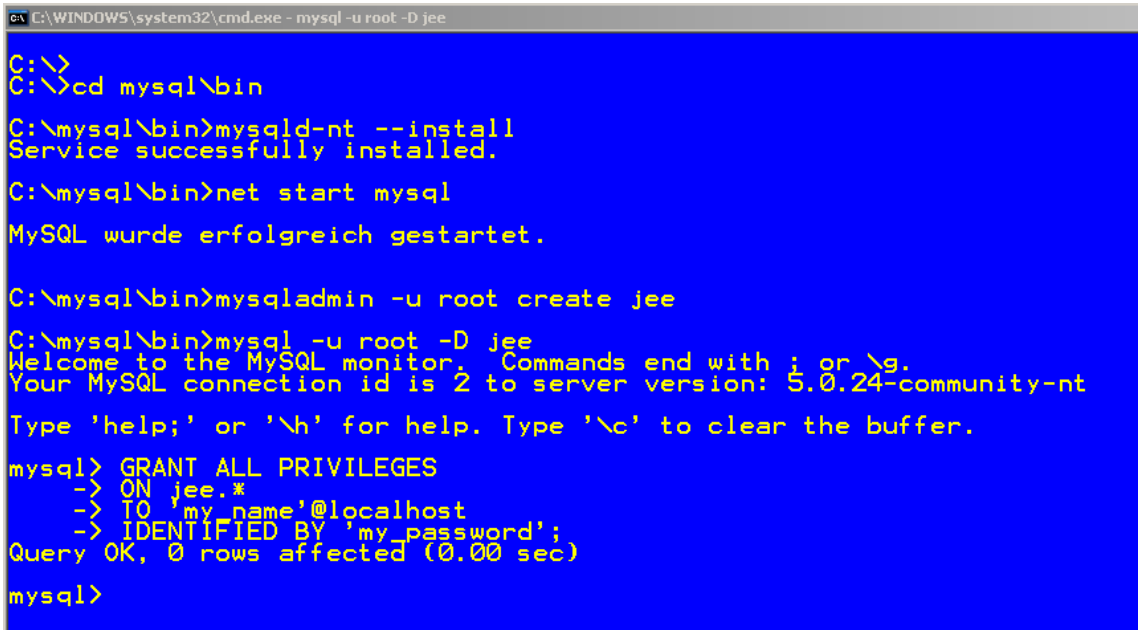
31.2 mySQL-Tools

Zur windowbasierten Administration von mySQL DBMS können verschiedene Tools installiert werden, z.B:

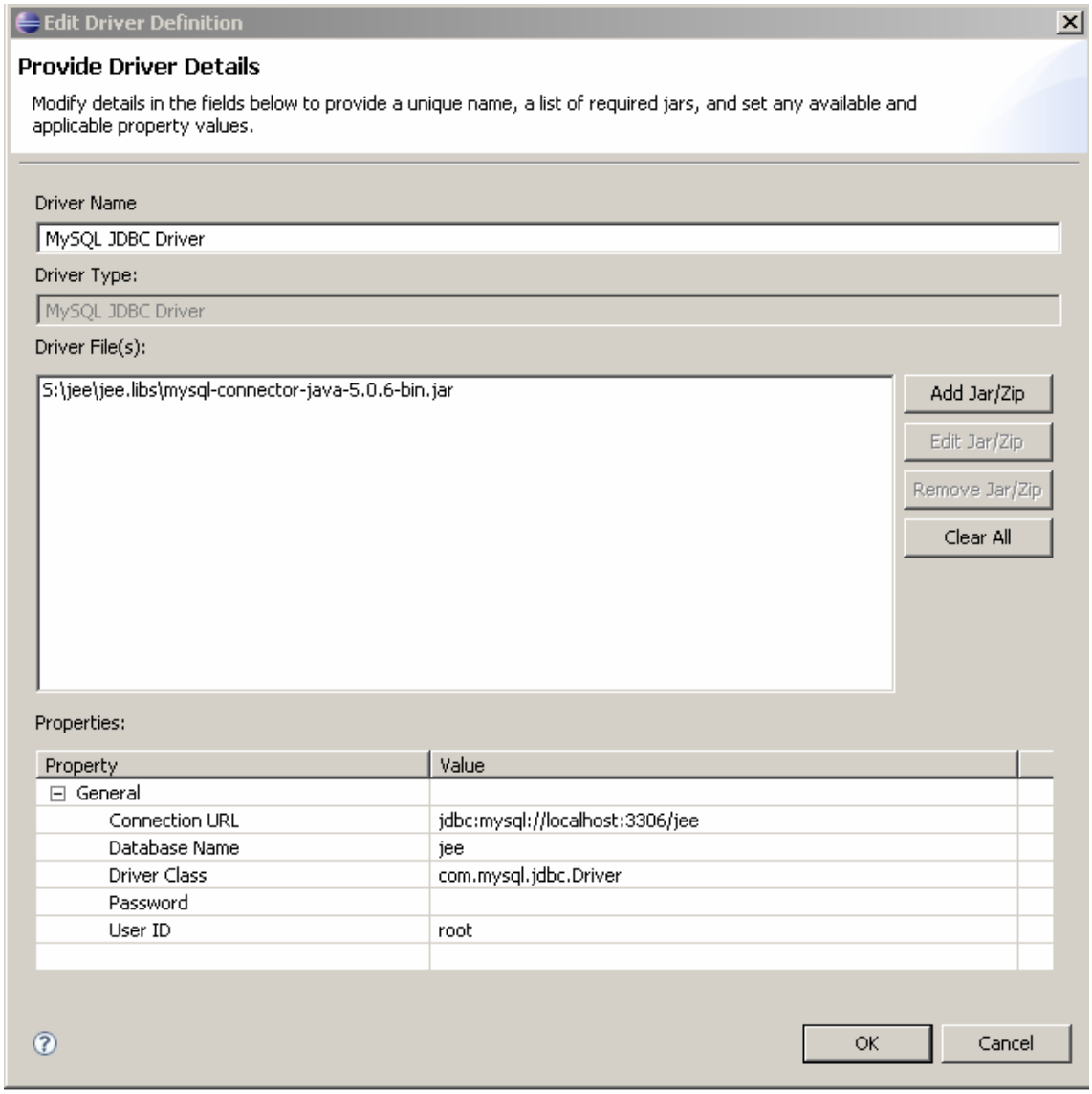
entpacken:	`mysql-gui-tools-noinstall-5.0-r12-win32`
von:	`[CD]\install\mysql` oder `http://www.mysql.com`
nach:	`c:\`

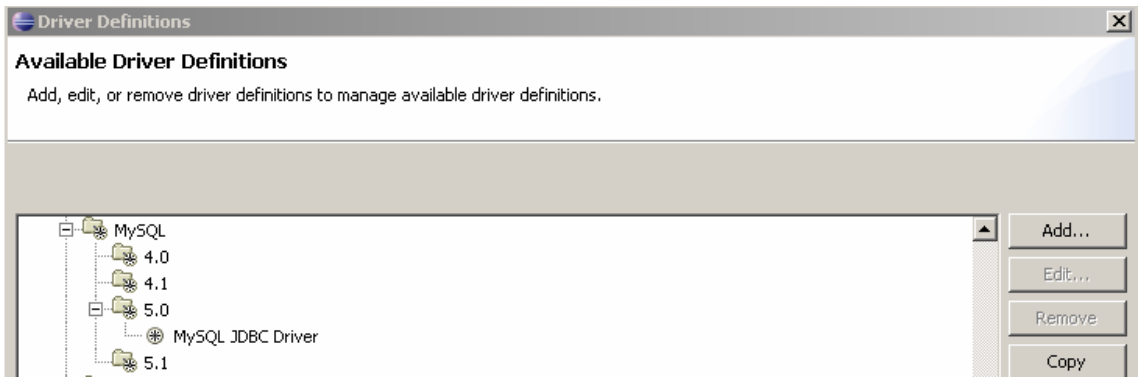
31.3 EnterpriseEducation DB anlegen

Mit Administrationstool oder per Konsole den Datenbankserver starten und die Datenbank "jee" erstellen:



31.4 Eclipse Plugin mit DB verbinden





F Anhang

32 Hibernate als Persistenzprovider

32.1 Hibernate Mapping Types

In Hibernate sind bereits Mapping Types für eine Vielzahl von Javatypen (primitive wie auch Klassen) enthalten. Die Hibernate Mapping-Typen werden stets klein geschrieben.

Hibernate-Mapping-Typ		
Typ	**JAVA**	**SQL**
string	java.lang.String	VARCHAR
short	short oder java.lang.Short	SMALLINT
integer	int oder java.lang.Integer	INTEGER
long	long oder java.lang.Long	BIGINT
float	float oder java.lang.Float	FLOAT
double	double oder java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC
character	java.lang.String	CHAR(1)
byte	byte oder java.lang.Byte	TINYINT
boolean	boolean oder java.lang.Boolean	BIT
yes_no	boolean oder java.lang.Boolean	CHAR(1) ('Y' oder 'N')
true_false	boolean oder java.lang.Boolean	CHAR(1) ('T' oder 'F')
date	java.util.Date oder java.sql.Date	DATE
time	java.util.Date oder java.sql.Time	TIME
timestamp	java.util.Date oder java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY oder BLOB
text	java.lang.String	CLOB
serializable	java.io.Serializable (beliebige Klasse welche das Interface implementiert)	VARBINARY oder BLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

32.2 Hibernate SQL Dialects

Hibernate-SQL-Dialect	
RDBMS	**Dialect**
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i/10g	org.hibernate.dialect.Oracle9Dialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

32.3 Package org.hibernate

32.3.1 Interfaces

Interface Summary	
Interface	**Beschreibung**
Criteria	Criteria is a simplified API for retrieving Entities by composing Criterion objects.
Filter	Type definition of Filter.
Interceptor	Allows user code to inspect and/or change property values.
Query	An object-oriented representation of a Hibernate Query.
ScrollableResults	A result iterator that allows moving around within the results by arbitrary increments.
Session	The main runtime interface between a Java application and Hibernate.
SessionFactory	Creates Sessions.
SQLQuery	Allows the user to declare the types and select list injection points of all entities returned by the query.
StatelessSession	A command-oriented API for performing bulk operations against a database. A stateless session does not implement a first-level cache nor interact with any second-level cache, nor does it implement transactional write-behind or automatic dirty checking, nor do operations cascade to associated instances.
Transaction	Allows the application to define units of work, while maintaining abstraction from the underlying transaction implementation.

32.3.2 Classes

Class Summary	
Klasse	**Beschreibung**
CacheMode	Controls how the session interacts with the second-level cache and query cache.
ConnectionReleaseMode	Defines the various policies by which Hibernate might release its underlying JDBC connection.
EmptyInterceptor	An interceptor that does nothing.
EntityMode	Defines the representation modes available for entities.
FetchMode	Represents an association fetching strategy.
FlushMode	Represents a flushing strategy.
Hibernate	Provides access to the full range of Hibernate built-in types.
LockMode	Instances represent a lock mode for a row of a relational database table.
ReplicationMode	Represents a replication strategy.
ScrollMode	Specifies the type of JDBC scrollable result set to use underneath a ScrollableResults

32.3.3 Exceptions

Exception Summary	
Exception	**Beschreibung**
AssertionFailure	Indicates failure of an assertion: a possible bug in Hibernate.
CallbackException	Should be thrown by persistent objects from Lifecycle or Interceptor callbacks.
DuplicateMappingException	Raised whenever a duplicate for a certain type occurs.
HibernateException	Any exception that occurs inside the persistence layer or JDBC driver.
InstantiationException	Thrown if Hibernate can't instantiate an entity or component class at runtime.
InvalidMappingException	Thrown when a mapping is found to be invalid.
JDBCException	Wraps an SQLException.
LazyInitializationException	Indicates access to unfetched data outside of a session context.
MappingException	An exception that usually occurs at configuration time, rather than runtime, as a result of something screwy in the O-R mappings.
MappingNotFoundException	Thrown when a resource for a mapping could not be found.
NonUniqueObjectException	This exception is thrown when an operation would break session-scoped identity.
NonUniqueResultException	Thrown when the application calls Query.uniqueResult() and the query returned more than one result.
ObjectDeletedException	Thrown when the user tries to do something illegal with a deleted object.
ObjectNotFoundException	Thrown when Session.load() fails to select a row with the given primary key (identifier value).
PersistentObjectException	Thrown when the user passes a persistent instance to a Session method that expects a transient instance.
PropertyAccessException	A problem occurred accessing a property of an instance of a persistent class by reflection, or via CGLIB.
PropertyNotFoundException	Indicates that an expected getter or setter method could not be found on a class.
PropertyValueException	Thrown when the (illegal) value of a property can not be persisted.
QueryException	A problem occurred translating a Hibernate query to SQL due to invalid query syntax, etc.
QueryParameterException	Parameter invalid or not found in the query
SessionException	Thrown when the user calls a method of a Session that is in an inappropropriate state for the given call (for example, the the session is closed or disconnected).
StaleObjectStateException	A StaleStateException that carries information about a particular entity instance that was the source of the failure.
StaleStateException	Thrown when a version number or timestamp check failed, indicating that the Session contained stale data (when using long transactions with versioning).
TransactionException	Indicates that a transaction could not be begun, committed or rolled back.
TransientObjectException	Thrown when the user passes a transient instance to a Session method that expects a persistent instance.
TypeMismatchException	Used when a user provided type does not match the expected one
UnresolvableObjectException	Thrown when Hibernate could not resolve an object by id, especially when loading an association.
WrongClassException	Thrown when Session.load() selects a row with the given primary key (identifier value) but the row's discriminator value specifies a subclass that is not assignable to the class requested by the user.

http://www.hibernate.org/hib_docs/v3/api/org/hibernate/package-summary.html

32.4 Hibernate ID-Generatoren

Generator		
Strategie	**Parameter**	**Beschreibung**
native	–	Wählt entsprechend der darunterliegenden Datenbank eine Strategie (identity, sequence oder hilo). Welche Strategie für eine Datenbank gewählt wird, ist in den Dialekten definiert.
uuid	separator	Gibt einen String mit Länge von 32 ausschliesslich hexadezimalen Zeichen zurück. Optional kann ein Separator zwischen den UUID-Komponenten mit generiert werden.
hilo	table column max_lo	Dieser Generator nutzt einen Hi/Lo-Algorithmus, um numerische IDs (Long, long, int, ...) zu erzeugen. Optional können die Spaltennamen für die Hi/Lo- Tabelle angegeben werden (Defaultwerte: hibernate_unique_key und next_hi). Mit max_lo kann die Anzahl der IDs bestimmt werden, die erzeugt werden, bevor wieder ein Datenbankzugriff erfolgt, um den Max-Value zu erhöhen. Der Generator kann nicht mit einer eigenen Connection oder eine über JTA (Java Transaction API) erhaltene Connection verwendet werden, da Hibernate in der Lage sein muss, den "hi"-Value in einer neuen Transaktion zu holen.
seqhilo	sequence max_lo parameters	Dieser Generator kombiniert einen Hi/Lo-Algorithmus mit einer darunterliegenden Sequence, die die Hi-Values generiert. Die Datenbank muss Sequences unterstützen, wie zum Beispiel Oracle und PostgreSQL. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".
identity	–	Dieser Generator unterstützt Identity Columns bzw. autoincrement, die es beispielsweise in MySQL, HSQLDB, DB2 und MS SQL Server gibt.
select	key (required)	Die ID wird über ein Select mit einem eindeutigen key erhalten. Der Primärschlüssel wird von der Datenbank, zum Beispiel mit einem Trigger, vergeben.
sequence	sequence parameters	Dieser Generator unterstützt Sequences, die es beispielsweise in PostgreSQL, Oracle und Firebird gibt. Die ID kann vom Type Long, long, int etc. sein. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".
assigned	–	Bei assigned muss die ID selbst gesetzt werden, vor dem Aufruf von save(). Nützlich bei natürlichen Keys. Das ist zugleich auch die Default-Strategie, falls die Annotation @GeneratedValue nicht angegeben wurde.
increment	–	Der maximale Primärschlüssel-Wert einer Tabelle wird beim Start der Anwendung gelesen und dann jedes Mal erhöht, wenn ein Insert erfolgt. Sollte nicht in einer Cluster-Umgebung benutzt werden.
foreign	property (required)	Benutzt die ID eines assoziierten Objekts, welches beim Parameter property angegeben werden muss. Wird gewöhnlich in Verbindung mit 1-zu-1-Beziehungen verwendet.
guid	–	Nutzt die von MS SQL Server oder MySQL generierte GUID.

33 Quellenangaben

Die Liste der Quellenangaben stellt einen Ausschnitt der vorhandenen Literatur und Onlinequellen dar. Bei der Beschaffung von weiterführenden Informationen ist auf die Kompatibilität zu der verwendeten AS Version und zur EJB Technologie zu achten.

33.1 Bücher

J2EE, Einstieg für Anspruchsvolle

Thomas Stark
Addison-Wesley
ISBN 3-8273-2184-0

Core J2EE Patterns, Best Practices and Design Strategies

Deepak Alur, Joh Crupi, Dan Malks
Pearson Professional Education
ISBN 0-13-142246-4

J2EE Patterns, Entwurfsmuster für die J2EE

Adam Bien
Addison-Wesley
ISBN 3-8273-2124-7

Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison-Wesley
ISBN 3-8273-1862-9

Mastering Enterprise JavaBeans

Ed Roman, Rima Patel Sriganesh, Gerald Brose
Wiley
ISBN 0-7645-7682-8

Enterprise Java Security

Marco Pistoia, Nataraj Nagaratnam, Larry Koved, Anthony Nadalin
Addison-Wesley
ISBN 9-780321-118899

Java Persistence with Hibernate

Bauer, King
Manning
ISBN 1-932394-88-5

J2EE Security

Pankaj Kumar
Addison-Wesley
ISBN 0-13-140264-1

33.2 Online Quellen

SUN

http://java.sun.com

JBOSS

http://www.jboss.org

HIBERNATE

http://www.hibernate.org

weitere

http://developers.sun.com
http://www.J2EE-develop.de
http://forum.java.sun.com
http://www.galileocomputing.de/openbook/java2
http://www.unix.org.ua/orelly/java-ent
http://www.jeckle.de
http://www.ejbkomplett.net
http://www.theserverside.com/tt/books/wiley/masteringEJB3/index.tss
http://www-306.ibm.com/software/websphere
http://www.pc-ware.de/de/PC-Ware/ssl/oracle/appl_server.htm

Technologien

http://ch.sun.com
http://www.jboss.com
http://tomcat.apache.org
http://ant.apache.org
http://www.mysql.de
http://www.hibernate.org

34 Installierte Software

Die für die Ausführung der Beispiele verwendeten Softwareversionen:

Java JSE

- jdk-1_5_0_12-windows-i586-p

Applikations Server

- JBoss-4.2.0.GA

mySQL

- mysql-noinstall-5.0.24-win32
- mysql-gui-tools-noinstall-5.0-r12-win32
- mysql-connector-java-5.0.6-bin

Entwicklungsumgebung

- eclipse-jee-europa-win32
  - Plugin : xmlbuddy_2.0.72