

J.EJB

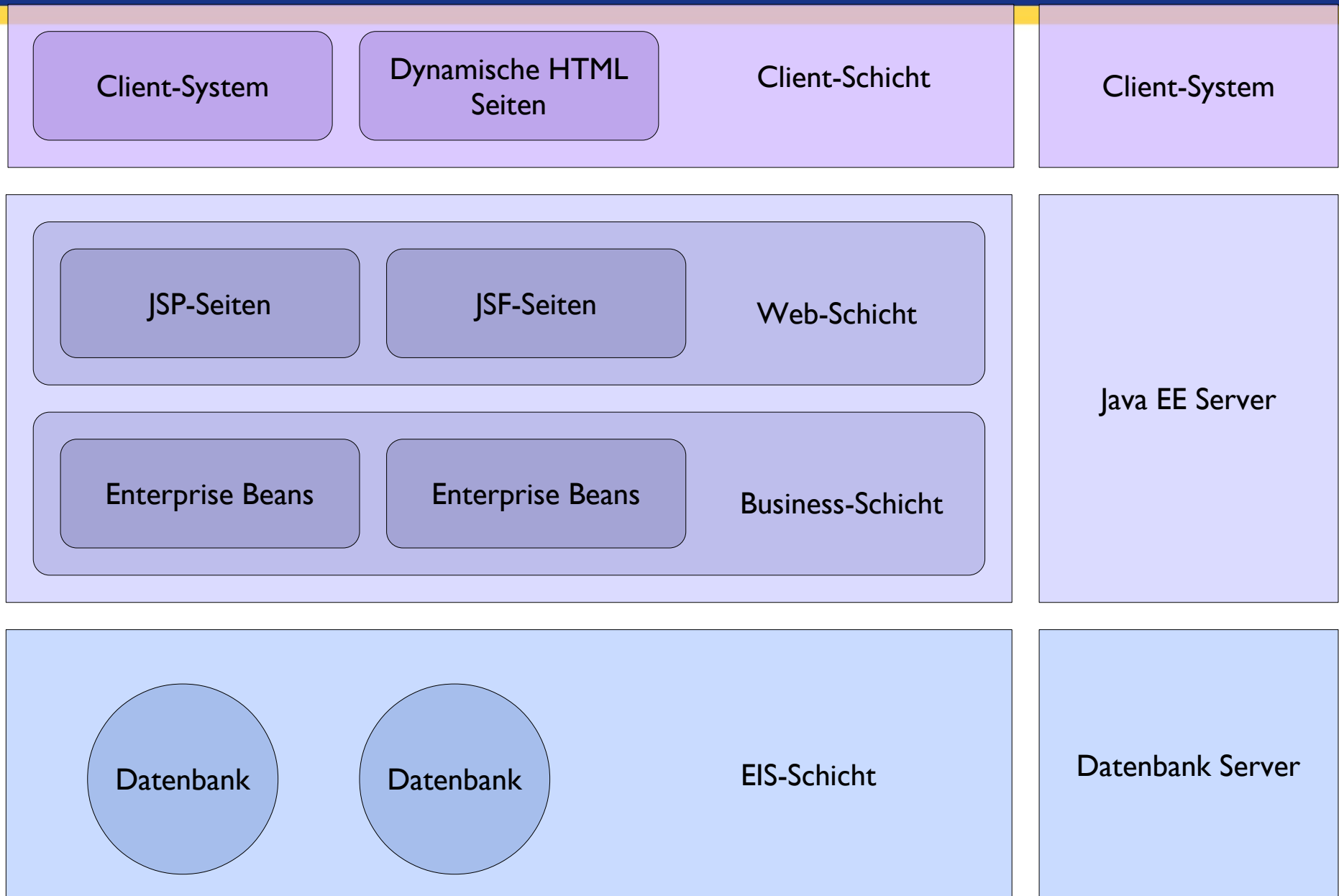
Enterprise JavaBeans (EJB) Technologie und Architektur

www.mathema.de

- ▼ Teil I: Java EE Einführung
- ▼ Teil II: Session Beans
- ▼ Teil III: Messaging
- ▼ Teil IV: Interceptoren
- ▼ Teil V: Injizieren von Service Objekten
- ▼ Teil VI: Java Persistence API
- ▼ Teil VII: Transaktionen
- ▼ Teil VIII: Sicherheit

Teil I

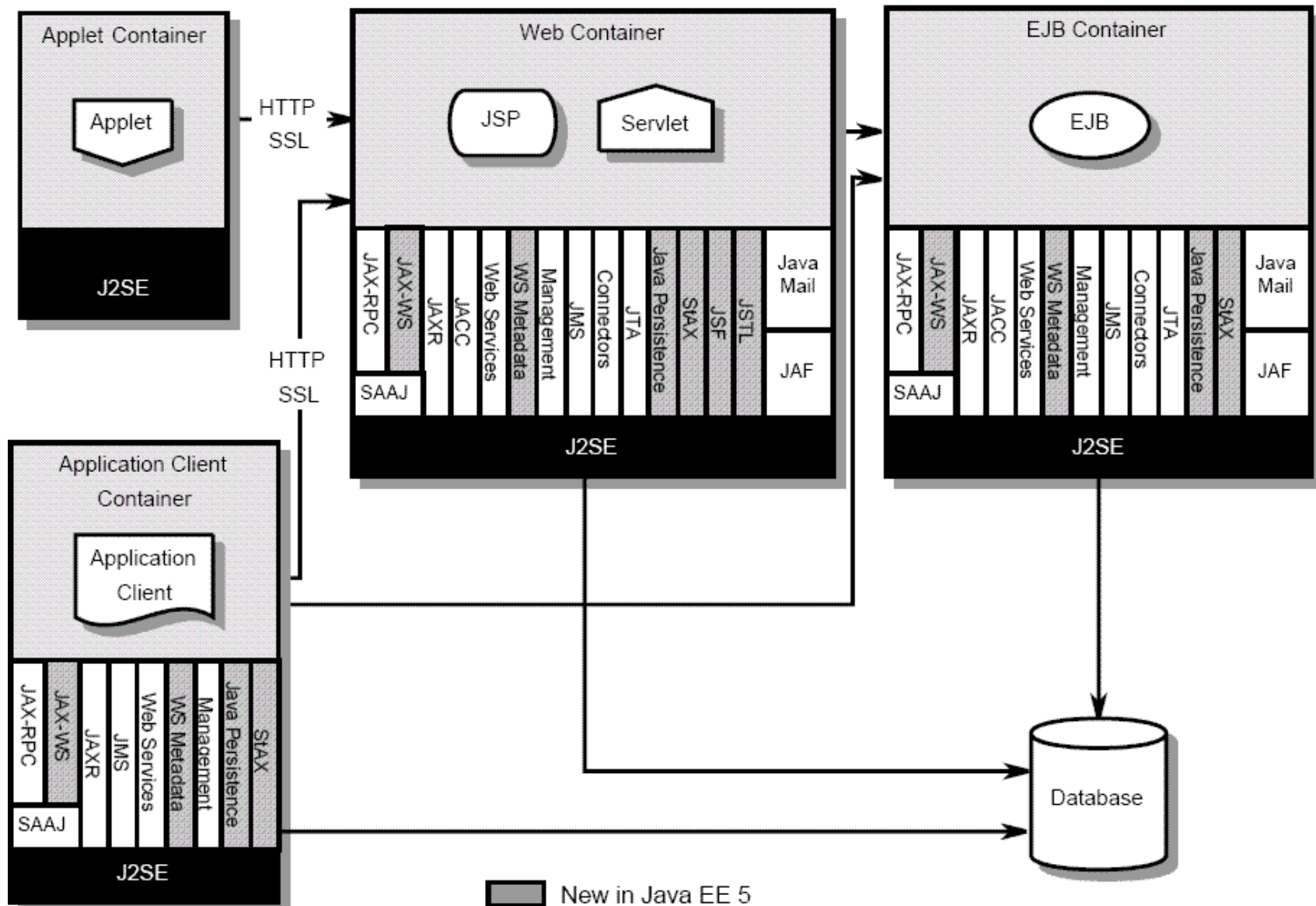
Java EE Einführung



- ▼ Trennung von Technik und Fachlichkeit (Separation of Concerns):
 - Technische Aspekte => Container
 - Fachliche Aspekte => Komponente
- ▼ Wiederverwendbarkeit in anderer Umgebung ohne Änderung des Programmcodes (Write once – install anywhere)
- ▼ Vereinfachung des Zugriffs auf Serverdienste

- ▼ Besitzt klar definierte Funktionalität
- ▼ Erlaubt Zugriff nur über Interfaces
- ▼ Black Box-Prinzip
 - Erlaubt keinen Zugriff auf interne Struktur
 - Implementierung kann ausgetauscht werden
 - Kann in anderem Umfeld wiederverwendet werden
- ▼ Grobe Granularität
- ▼ Nutzbarkeit in verschiedenen Kontexten
 - Lose Kopplung
 - Von außen konfigurierbar

- ▼ Java EE ist ein Standard für die komponentenbasierte Software-Entwicklung
- ▼ Java EE Anwendungen werden aus Komponenten aufgebaut, dazu gehören:
 - Client-Schicht Komponenten
 - Web-Schicht Komponenten
 - Business-Schicht Komponenten
- ▼ Jede Komponente wird in ihrer zugehörigen Umgebung (Container) ausgeführt
 - Client-Komponenten auf dem Client-Rechner
 - Web-Komponenten im Web-Container des Java EE-Servers
 - Business-Komponenten im EJB-Container des Java EE-Servers



▼ Applet Container

- verwaltet die Ausführung von Applets und besteht aus Web Browser und Java Plug-in

▼ Application Client Container

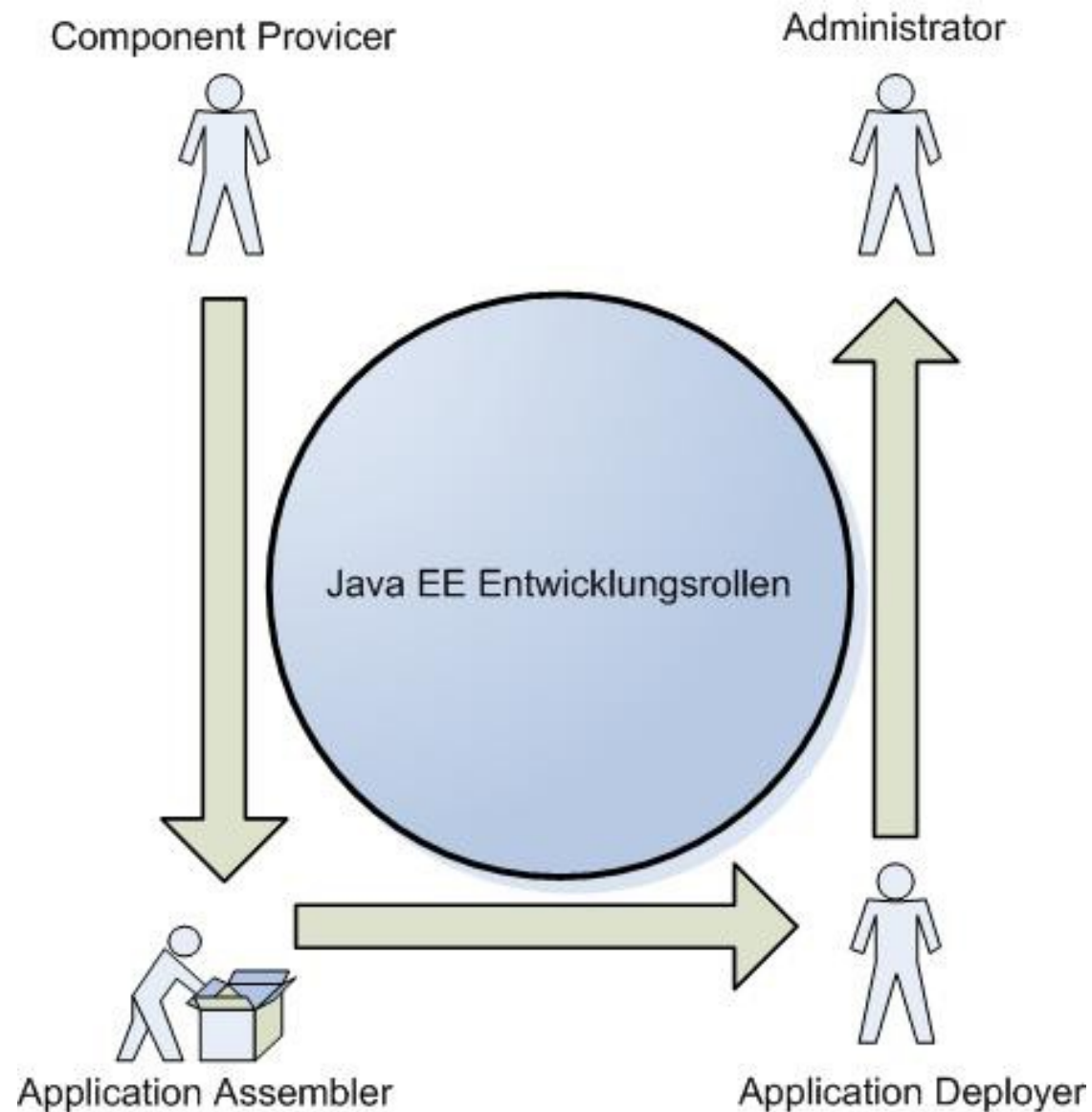
- verwaltet die Ausführung von Clients Komponenten

▼ Web Container

- verwaltet die Ausführung von JSP Seiten und Servlet Komponenten für Java EE Anwendungen

▼ Enterprise Bean Container

- verwaltet die Ausführung von Enterprise Beans für Java EE Anwendungen



▼ Component Provider

- eine Firma oder eine Person, die Web Komponenten oder Enterprise Beans für eine Java EE Anwendung zur Verfügung stellt

▼ Application Assembler

- eine Firma oder eine Person, die Module vom Component Provider bekommt und diese dann in eine JAR Datei mit der Erweiterung .ear verpackt

▼ Application Deployer und Administration

- eine Firma oder eine Person, die für Konfiguration, die Verteilung und die Überwachung von Java EE Anwendungen verantwortlich ist

▼ Java EE Product Provider

- entwirft und stellt die Java EE Plattform APIs zum Verkauf zur Verfügung
- sind Anbieter von Java EE Servern wie BEA, JBoss, Weblogic, ...

▼ Tool Provider

- eine Person oder eine Firma, die Tools für die Entwicklung, für die Verpackung und für Verteilung zur Verfügung stellt
- die meisten Java IDEs wie Eclipse, NetBeans und JDeveloper unterstützen die Entwicklung von Java EE Anwendungen

▼ Enterprise JavaBeans

- stellen ein standardisiertes Modell für die Erstellung serverseitiger Komponenten dar, die die Geschäftsprozesse eines Systems repräsentiert.
- sind architekturelles Modell für die
 - Entwicklung,
 - Verteilung und
 - Installation von Java-basierten, serverseitigen Komponenten

▼ Version 1.0 (1998)

- Erste Spezifikation
- EJB-Server musste SessionBeans unterstützen
- EJB-Server musste den Lebenszyklus der EJBs abbilden

▼ Version 1.1 (1999)

- XML-basierter Deployment-Deskriptor
- EJB-Server mussten persistente Komponenten unterstützen

▼ Version 2.0 (2001)

- Asynchroner Aufruf von EJBs
- Umbau des Modells für persistente Komponenten
- Lokale Interfaces

▼ Version 2.1 (2003)

- Webservice Support

▼ Version 3.0 (2006)

- Reduzierung der Anzahl von Artefakten
- Entfernen von API-Abhängigkeiten
- Implementierung auf Basis von Plain Old Java Objects (POJOs)
- Optionaler Einsatz von Deployment Deskriptoren
- Nutzung von neuen JDK 5 Eigenschaften
- Soweit wie möglich Verwendung von sinnvollen Default-Werten (Configuration by Exception)

▼ Version 3.1 (2009)

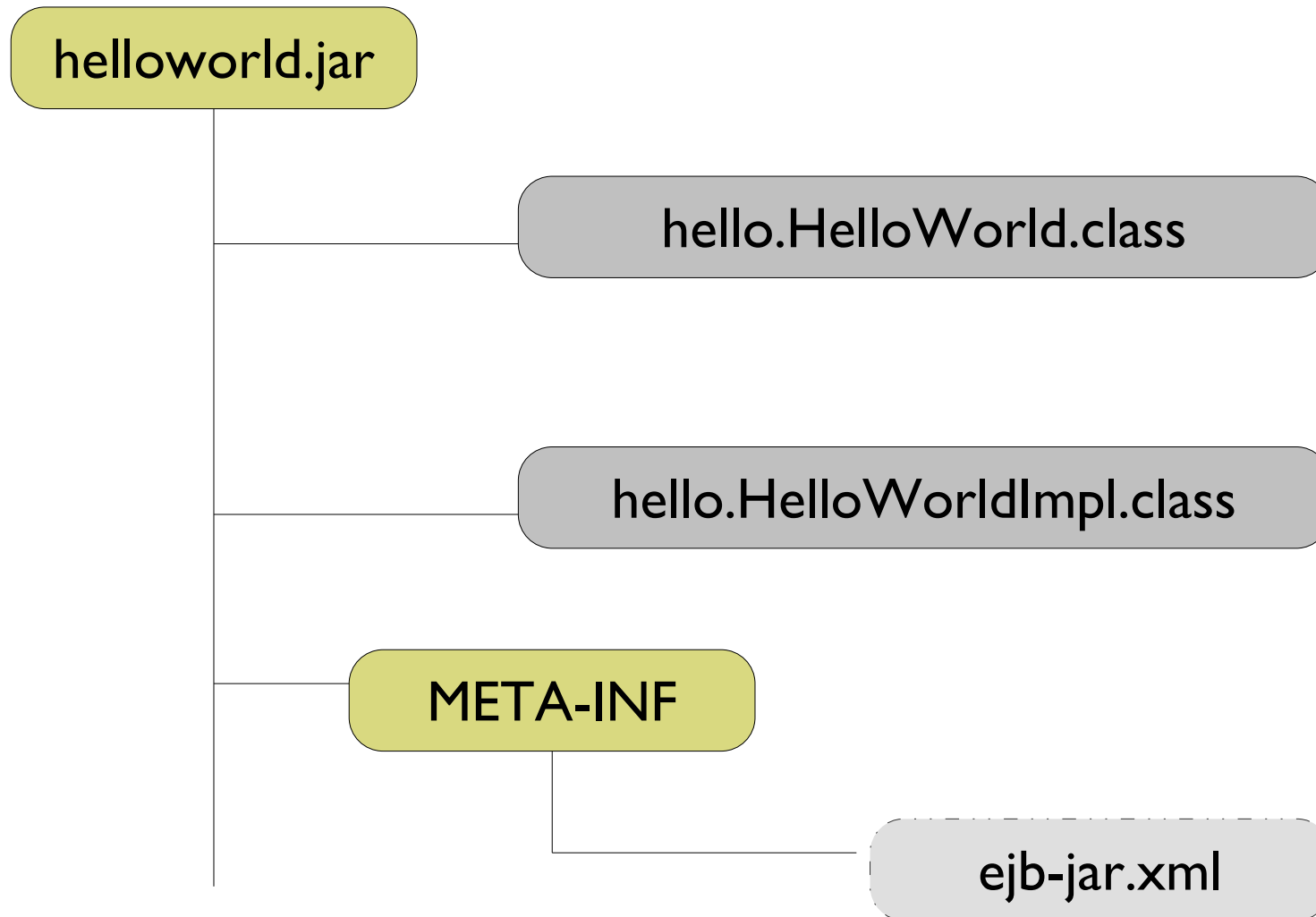
- Vereinfachtes lokale Sicht
- Singleton Bean
- Asynchroner Aufruf von SessionBeans
- Vereinfachtes Deployment
- EJB-lite
- Erweiterungen im Persistenzmodell (JPA 2.0)

▼ Version 3.2 (2013)

- Mehr Feature in EJB-lite
- Entity Beans nur noch optional
- Kleinere Erweiterungen im Persistenzmodell (JPA 2.1)

- ▼ Vereinfachung des Programmiermodells beim Einführung von neuen Features
- ▼ Vereinfachung des Packaging von Komponenten
- ▼ Einfachen Testen von EJB-Anwendungen (durch Embedded Containers)

▼ Minimale EJB 3 Anwendung



- ▼ Eine minimale EJB 3 Anwendung kann in eine JAR-Datei verpackt werden, die folgende Dateien beinhaltet:
 - das Business Interface definiert die Geschäftslogik
 - die Implementierungsklasse realisiert die Geschäftslogik
 - optional kann auch ein Deployment Deskriptor eingefügt werden

- ▼ Eine Packetierung innerhalb einer WAR-Datei ist ebenfalls möglich
 - Was landet wo?
 - Klassen/Interfaces: WEB-INF/classes
 - Deskriptor: WEB-INF/ejb-jar.xml
 - Es gelten die aus dem Webcontainer bekannten Regeln bzgl. Classloading!

- ▼ Während der Kompilierung des Clients werden folgende Klassen benötigt
 - `hello.HelloWorld.class`

- ▼ Während der Ausführung des Clients werden folgende Klassen benötigt
 - `hello.HelloWorld.class`
 - (Server)-spezifische Klassen für die Kommunikation
 - Proxy-Stub
 - Client-Bibliothek des Servers

▼ Component Contract

- Komponente kann Callback-Methoden implementieren
- Einhalten von „Verhaltensregeln“

▼ Container Contract

- Container muss Zugriff auf bestimmte Dienste gewähren:
 - Transaktionsmanagement
 - Sicherheit
 - JNDI (Naming)
 - Automatische Persistenz

▼ EJB-Komponenten laufen

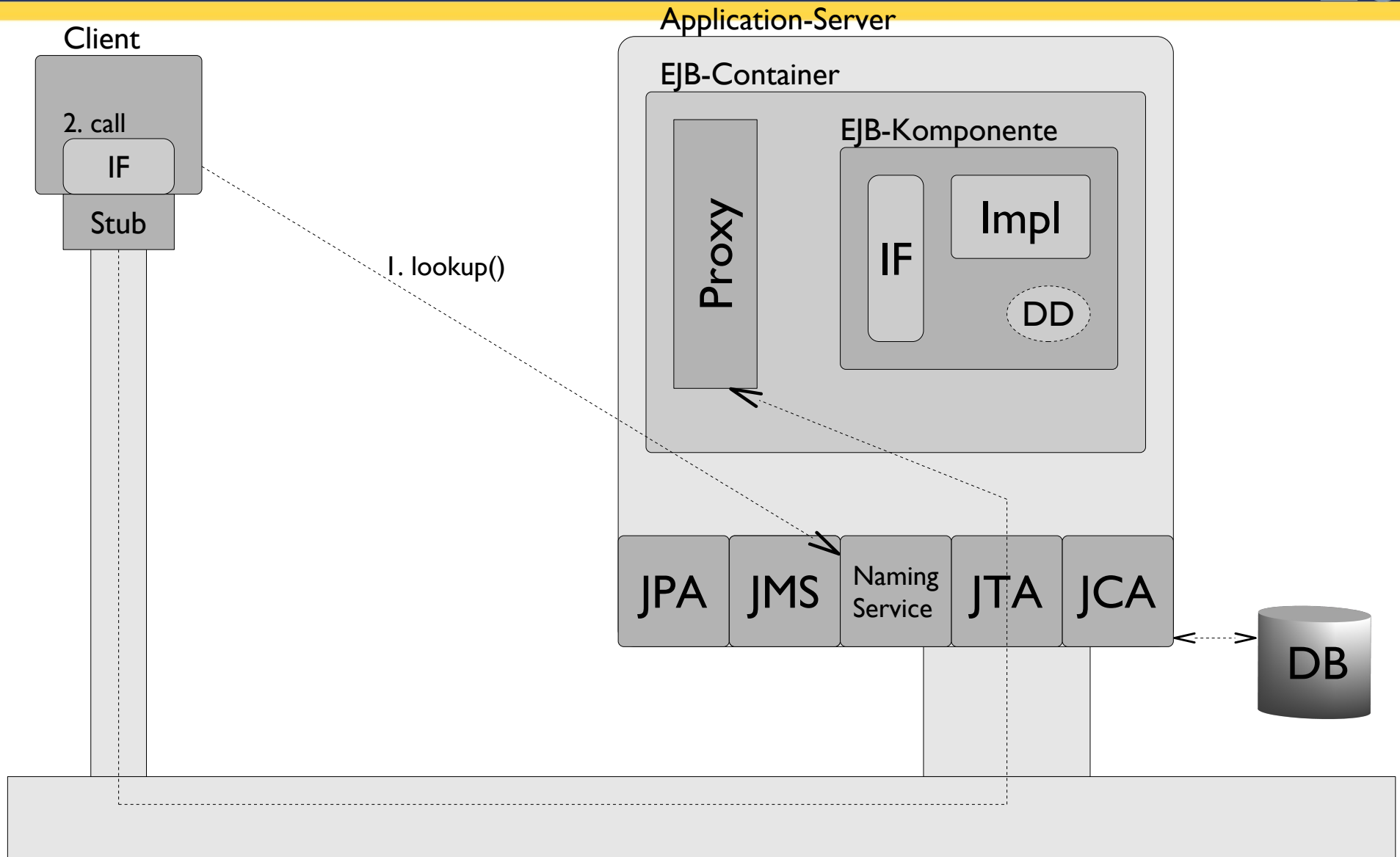
- in einem EJB-Container
- innerhalb eines Java EE Application Servers

▼ Application Server

- Sammlung von verschiedenen Serverdiensten
- Zentrale Konfiguration und Verwaltung

▼ EJB-Container

- Dienst innerhalb eines Application Servers
- Technische Infrastruktur für den Betrieb von EJB-Komponenten



Session Beans (Stateless, Stateful, Singleton)

- Erweiterung für Client-Anwendungen, um Prozesse oder Aufgaben zu verarbeiten
- Zugriff auf Session Beans durch RPC-Protokolle:
 - Java RMI-IIOP
 - CORBA-IIOP
- Meta-Information: `@javax.ejb.Stateless`, `@javax.ejb.Stateful`, `@javax.ejb.Singleton`

Message-Driven Beans

- MDB verarbeitet Nachrichten asynchron aus Systemen wie:
 - JMS
 - Legacy-Systeme
- Meta-Information: `@javax.ejb.MessageDriven`

▼ Komponenten-Interfaces

- Remote Business Interface
 - definiert Methoden, die von einer Anwendung außerhalb des EJB Containers aufgerufen werden können
 - ist ein einfaches Java Interface und wird durch die Meta-Information `@javax.ejb.Remote` markiert
- Local Business Interface
 - definiert Methoden, die von einer Anwendung innerhalb des EJB Containers aufgerufen werden können
 - ist ein einfaches Java Interface und wird durch die Meta-Information `@javax.ejb.Local` markiert
- Message Interface
 - definiert Methoden, die von Messaging Systemen benutzt werden, um Nachrichten zu schicken

▼ Bean Klasse

- beinhaltet Geschäftslogik und kann Remote/Local Interfaces haben
- Session Beans
 - wird mit Meta-Informationen `@javax.ejb.Stateless`, `@javax.ejb.Stateful` oder `@javax.ejb.Singleton` markiert
- Message-Driven Beans (MDB)
 - implementiert eine oder mehrere Methoden `onMessage()`, die vom Message Interface definiert sind
 - MDB-Klasse muss mit der Meta-Information `@javax.ejb.MessageDriven` markiert werden und implementiert das Interface `javax.jms.MessageListener`

▼ Deployment Deskriptor (optional)

- XML-Datei (ejb-jar.xml) zum Überschreiben von Meta-Information
- EJB Container liest diese Datei oder Meta-Informationen

```
package hello;
@javax.ejb.Remote
public interface HelloWorld {
    public String sayHelloTo(String name);
}

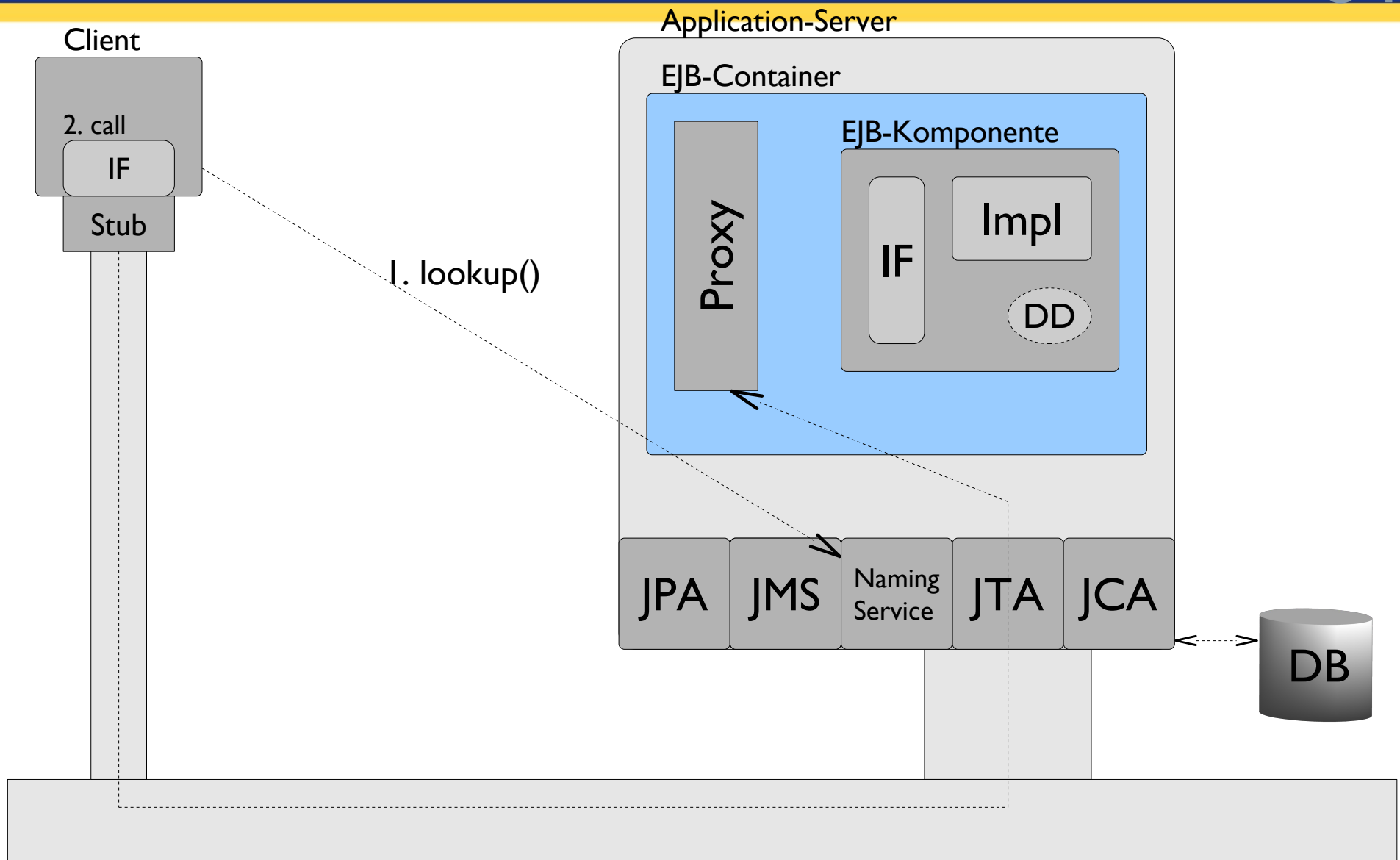
package hello;
@javax.ejb.Stateless (name="HelloWorldBean")
public class HelloWorldImpl implements HelloWorld{
    public String sayHelloTo(String name) {
        System.out.println("How are you " + name );
        return "Great, Thanks";
    }
}
```

```
package hello.clients;

import hello.HelloWorld;
import javax.naming.Context;
import javax.naming.NamingException;

public class Client {
    static public void main(String[] args) {
        try {
            Context jndiContext = new InitialContext();
            HelloWorld helloWorld = (HelloWorld)
                jndiContext.lookup("HelloWorldImpl/remote");
            String response = helloWorld
                .sayHelloTo("JBoss AS");
            System.out.println(response);
        } catch ( NamingException e ) {
            e.printStackTrace();
        }
    }
}
```

- ▼ Kein Thread-Management
- ▼ Keine statischen Felder (außer read-only)
- ▼ Keine Bildschirmausgabe, keine Tastatureingabe
- ▼ Kein direkter Zugriff auf Verzeichnisse und Dateien
- ▼ Keine Server-Sockets
- ▼ Keine neuen Class-Loader oder Security-Manager
- ▼ Keine native libraries
- ▼ Keine Referenz zu `this` (statt dessen kann `getEJBObject()` oder `getEJBLocalObject()` im `SessionContext` verwendet werden)
- ▼ Keine Verwendung von `System.exit()`



▼ Logisches und programmatisches Konzept

- kein physikalisches Konstrukt

▼ Zwischenhändler zwischen Bean-Klassen und EJB Server

- verwaltet die Erzeugung und Löschung von Bean-Instanzen
- stellt zur Laufzeit einige Dienste (Transaktion, Sicherheit, Concurrency, ...) für die Bean-Klassen zur Verfügung

▼ Container implementiert `javax.ejb.EJBContext`

- versorgt Bean-Klasse zur Laufzeit mit Informationen seiner Umgebung
- Bean-Container Vertrag
 - Session Beans benutzen `javax.ejb.SessionContext`
 - MDBs benutzen `javax.ejb.MessageDrivenContext`

▼ Container generiert automatisch EJB Proxies oder EJB Stubs für Client-Requests

- ▼ Software-Architektur und Programmierschnittstelle (API)
 - zur Integration von heterogenen EIS
- ▼ Diese Architektur definiert eine Menge von skalierbaren, gesicherten und transaktionellen Mechanismen, die
 - die Integration von EIS (Enterprise Information Systemen) mit
 - Java EE Enterprise Servern und anderen Java EE Anwendungen ermöglichen
- ▼ EJB 3 verwendet die Java EE Connector Architecture (JCA),
 - **das** Push Modell wird ebenfalls unterstützt.
(inbound Connector)

- ▼ JAR Datei ist ein plattformunabhängiges Dateiformat
 - für die Kompression, Verpackung und Lieferung von mehreren Dateien zusammen
- ▼ JAR (Java Archive) Werkzeug
 - wurde ursprünglich für ein effizientes Herunterladen von Java Applets entwickelt
 - JAR Werkzeug kann auch für die Verpackung von EJB Definitionen und Klassen benutzt werden
 - mit dem Befehl
jar cf classes.jar Foo.class Bar.class
aufrufbar
- ▼ Paketierung innerhalb einer WAR-Datei möglich
- ▼ Die Beispiele dieser Unterlagen verwenden
 - das ANT Werkzeug für Verpackung und Deployment

- ▼ Deployment ist der Prozess, der durch
 - das Lesen einer EJB JAR Datei,
 - die Änderung oder das Einfügen eines Deployment Deskriptors,
 - die Abbildung von POJOs auf die Datenbank,
 - die Definition von Zugriffskontrollen,
 - und die Generierung von spezifischen Dateien ausgeführt wird.
- ▼ Einige EJB Server-Produkte stellen Deployment-Werkzeuge zur Verfügung, um EJBs im Applikation-Server zu installieren
- ▼ In einigen EJB Server wie JBoss AS wird die EJB JAR Datei in ein *deploy*-Verzeichnis kopiert

```
package hello.clients;
public class Client {
    public static void main(String[] args) {
        try {
            Context jndiContext = getInitialContext();
            HelloWorld helloWorld = (HelloWorld)
                jndiContext.lookup("HelloWorldBean/remote");
            String response = helloWorld
                .sayHelloTo("JBoss AS");
            System.out.println(response);
        } catch ( NamingException e ) {
            e.printStackTrace();
        }
    }

    private static Context getInitialContext() throws NamingException {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        p.put(Context.URL_PKG_PREFIXES,
            "org.jboss.naming:org.jnp.interfaces");
        p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
        return new InitialContext(p);
    }
}
```

- ▼ Enterprise JavaBeans Servers können drei Arten von Sicherheiten unterstützen:
 - Authentifizierung
 - bezeichnet den Vorgang der Überprüfung der Identität eines Gegenübers (Identifikation von Username und Password)
 - Authorization oder Access Control
 - legt eine Sicherheitsrichtlinie an, die festlegt, was ein spezifischer Benutzer machen darf oder nicht
 - Sichere Kommunikation
 - Kommunikationskanäle zwischen Clients und Server sind Bereiche, in denen Sicherheit eine große Rolle spielt
 - Kommunikationskanäle können durch Verschlüsselung gesichert werden
 - hier kommt der Austausch von Schlüsseln ins Spiel

▼ Transaktion besitzt ACID Eigenschaften

- Atomarität,
- Konsistenz,
- Isolation
- und Durabilität

▼ Deklarative Programmierung wird benutzt,

- um Transaktionsattribute zu setzen, die von EJB Server zur Laufzeit ausgewertet werden.

Teil II

Session Beans

▼ Was ist eine Session Bean?

- führt Methoden für den Client aus
- kapselt die Komplexität der Anwendung für den Client
- Session Bean ist ein nicht-persistentes Objekt
- Session Bean wird nicht gemeinsam oder simultan benutzt

▼ Ausprägungen von Session Beans

- Stateful Session Bean
- Stateless Session Bean
- Singleton Session Bean

SLSB

Stateless Session Beans

▼ Eigenschaften

- Stateless Session Beans können keine Daten zwischenspeichern
- bei jedem Aufruf einer zustandslosen Session Bean müssen alle Informationen als Parameter übergeben werden
- Stateless Session Beans sind leicht und schnell

▼ Anwendungen

- sie sind geeignet für
 - Report-Generierung,
 - Batch-Verarbeitung,
 - ...

▼ Session Bean Klasse

- wird mit der Meta-Information `@javax.ejb.Stateless` markiert

▼ Local Business Interface

- wird mit der Meta-Information `@javax.ejb.Local` markiert
- Zugriffe mit call-by-reference
- Zugriff nur innerhalb einer VM
- Kann auch entfallen (vereinfachte lokale Sicht)

▼ Remote Business Interface

- wird mit der Meta-Information `@javax.ejb.Remote` markiert
- Zugriffe mit call-by-value

▼ Web Service Endpoint Interface

- wird mit der Meta-Information `@javax.jws.WebService` markiert

▼ Optionale Anwendung

- es existiert auch die Möglichkeit, die `@javax.ejb.Local`, `@javax.ejb.Remote` und `@javax.jws.WebService` Meta-Informationen in der Session Bean-Klasse zu benutzen

Attribute der Annotation javax.ejb.Remote:

Attribut	Typ	Bedeutung	Default
value	Class[]	Angabe aller Schnittstellen, die entfernt zugreifbar sein sollen.	-

Attribute der Annotation javax.ejb.Local:

Attribut	Typ	Bedeutung	Default
value	Class[]	Angabe aller Schnittstellen, die lokal zugreifbar sein sollen.	-

Attribute der Annotation javax.ejb.Stateless:

Attribut	Typ	Bedeutung	Default
description	String	Beschreibung der EJB. Diese Daten sind zur Anzeige im Deploy-Tool des Application Server gedacht. (optional)	-
mappedName	String	Name, unter dem die EJB im Namensdienst registriert werden soll. Eine Unterstützung dieses Attributs durch die Server-Hersteller ist NICHT verpflichtend! Aus diesem Grund sollte es nicht verwendet werden. (optional)	-
name	String	Logischer Name der EJB. Auch diese Daten sind für die Anzeige im Deploy-Tool gedacht. Innerhalb des Jboss-Servers wird dieser Name zur Erzeugung des Default JNDI-Namens herangezogen.	Name der annotierten Klasse

```
package de.mathema.slsb;
import de.mathema.domain.*;
public interface ProcessPayment {
    public boolean byCreditCard(Customer customer,
        CreditCardDO card, double amount)
        throws ProcessPaymentException;
}
```

```
package de.mathema.slsb;
```

```
@javax.ejb.Local
public interface ProcessPaymentLocal
extends ProcessPayment {
}
```

```
package de.mathema.slsb;
```

```
@javax.ejb.Remote
public interface ProcessPaymentRemote
extends ProcessPayment {
}
```

```
package de.mathema.slsb;
...
@javax.ejb.Stateless
public class ProcessPaymentBean implements
    ProcessPaymentLocal, ProcessPaymentRemote {

    final public static String CREDIT = "CREDIT";
    @Resource(mappedName = "carDatabase")
    DataSource ds;

    public boolean byCreditCard(Customer customer,
        CreditCardDO do, double amount) throws ProcessPaymentException {
        if (cCardDO.expiration.before(new Date())) {
            throw new ProcessPaymentException("Ablaufdatum ist überholt");
        } else {
            return process(customer.getId(), amount, CREDIT, cCardDO.number,
                new Date(cCardDO.expiration.getTime()));
        }
    }

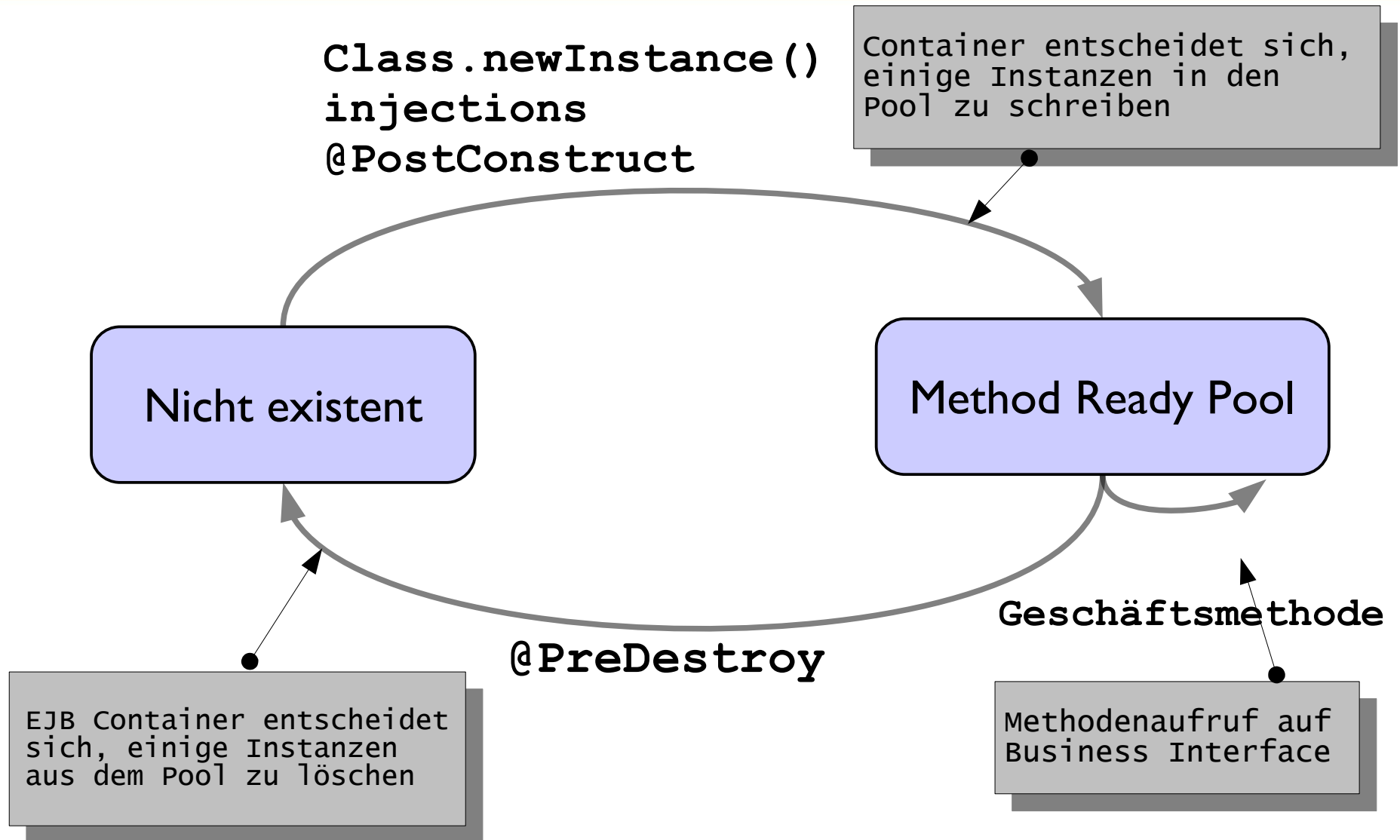
    private boolean process(long customerID, double amount,
        String type, String number, Date creditExpDate)
        throws ProcessPaymentException {
        ...
    }
}
```



```
// Get the Initial Context
InitialContext jndiContext = new InitialContext();

// Lookup for the Bean
ProcessPaymentRemote processPayment = (ProcessPaymentRemote)
    jndiContext.lookup("ProcessPaymentBean/remote");

//Use the ProcessPayment
...
```



▼ Der „Nicht existent“-Zustand

- Bean Objekt ist nicht im Speicher oder noch nicht instanziiert

▼ „Method Ready Pool“

- EJB Container braucht ein Bean Objekt
- nach dem Starten eines EJB Servers wird eine Anzahl von SLSB erzeugt und in den Pool gespeichert
- der Übergang **Nicht existent** => **Method Pool Ready** verursacht die Ausführung von drei Operationen
 - `Class.newInstance()` - Bean wird instanziiert
 - EJB Container injiziert alle EJB Ressourcen, die von der Bean-Klasse mittels Injection oder XML Deskriptor abgefragt wurden
 - EJB Container erzeugt ein Post-Construction-Ereignis
 - Die Bean-Klasse kann dieses Ereignis durch die Markierung einer Methode mit der Meta-Information `@javax.annotation.PostConstruct` registrieren

```
@javax.ejb.Stateless
public class ProcessPaymentBean implements
ProcessPaymentLocal, ProcessPaymentRemote {
    ...

    @javax.annotation.PostConstruct
    public void myInit() {

    }
}
```

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ProcessPaymentBean</ejb-name>
      <post-construct>
        <lifecycle-callback-method>myInit
        </lifecycle-callback-method>
      </post-construct>
    </session>
  </enterprise-beans>
</ejb-jar>
```

▼ Der Übergang *Method Pool Ready* => *Nicht existent*

- EJB Server benötigt die SLSB nicht mehr
- EJB Server reduziert die Anzahl von SLSBs
 - Dieser Prozess beginnt, wenn ein **PreDestroy** Ereignis auf einer SLSB ausgelöst wird
 - Die Bean Klasse kann dieses Ereignis mit der Annotation **@javax.annotation.PreDestroy** registrieren
 - **@PreDestroy** Callback Methode kann alle Clean-up Operationen ausführen

```
@javax.ejb.Stateless
public class ProcessPaymentBean implements
ProcessPaymentLocal, ProcessPaymentRemote {
    ...

    @javax.annotation.PreDestroy
    public void cleanup() {

    }
}
```

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ProcessPaymentBean</ejb-name>
      <pre-destroy>
        <lifecycle-callback-method>cleanup</lifecycle-callback-method>
      </pre-destroy>
    </session>
  </enterprise-beans>
</ejb-jar>
```

▼ Handling der Callbacks in Handlerklasse (Interceptor) auslagern

- keine „Verschmutzung“ der Implementierung
- ein Handler für verschiedene Bean-Implementierungen möglich

▼ Lifecycle-Methoden

- werden in Handler ausgelagert
- werden im Handler annotiert

▼ Handler (Interceptor)

- wird an der Klassendefinition der Bean annotiert
- `@javax.interceptor.Interceptors`

```
@javax.ejb.Stateless
@javax.interceptor.Interceptors({MyCallbackHandler.class})
public class ProcessPaymentBean implements
ProcessPaymentLocal, ProcessPaymentRemote {
    ...
}
```

```
public class MyCallbackHandler{

    @javax.annotation.PreDestroy
    public void cleanup(InvocationContext ic){

    }
}
```


Kein Zustand

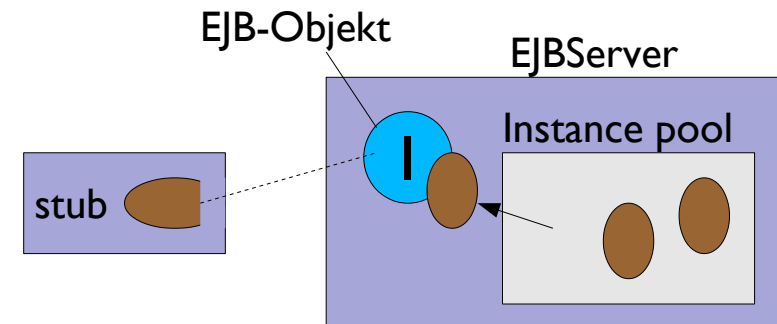
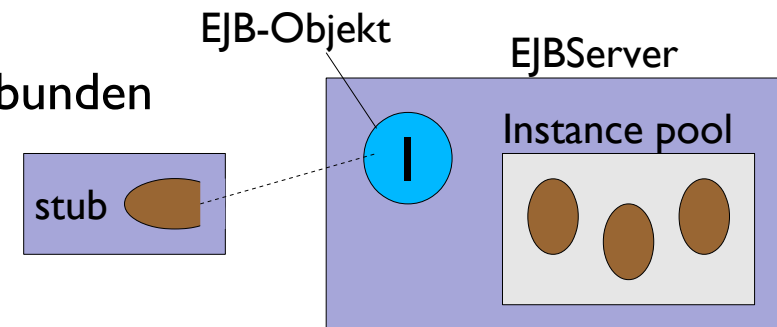
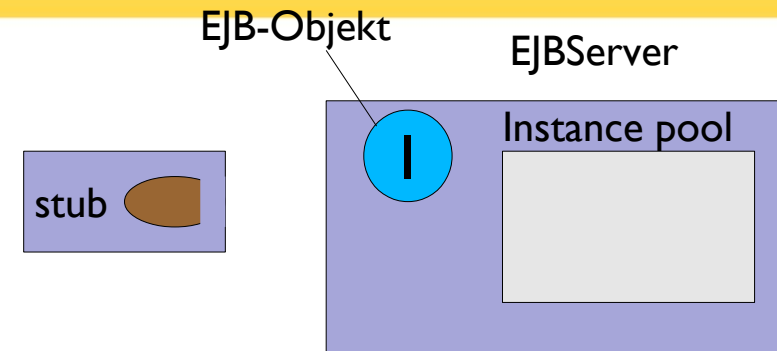
- Beans sind noch nicht instanziiert

Zusammengefasster Zustand

- Beans sind jetzt vom Container instanziiert
- Beans sind noch nicht mit einer EJB-Abfrage verbunden

Verfügbarer Zustand

- Beans sind mit einer EJB- Abfrage verbunden
- Beans sind bereit, einen Methodenaufruf zu beantworten



SFSB

Stateful Session Bean

▼ Stateful Session Bean Klasse

- wird mit der Meta-Information `@javax.ejb.Stateful` markiert
- wird einem Client zugeordnet
- wird nicht in einem Pool von Instanzen gehalten
- behält dialogorientierte Zustände bei
- ist nicht persistent

▼ SFSB hat ein eigenes Gedächtnis

- SFSB kann Daten aus einem Methodenaufruf speichern, damit sie bei einem späteren Aufruf einer anderen Methode wieder zur Verfügung steht

▼ SFSB wird nicht zwischen EJB Objekten ausgetauscht

Attribute der Annotation javax.ejb.Stateful:

Attribut	Typ	Bedeutung	Default
description	String	Beschreibung der EJB. Diese Daten sind zur Anzeige im Deploy-Tool des Application Server gedacht. (optional)	-
mappedName	String	Name, unter dem die EJB im Namensdienst registriert werden soll. Eine Unterstützung dieses Attributs durch die Server-Hersteller ist NICHT verpflichtend! Aus diesem Grund sollte es nicht verwendet werden. (optional)	-
name	String	Logischer Name der EJB. Auch diese Daten sind für die Anzeige im Deploy-Tool gedacht. Innerhalb des Jboss Servers wird dieser Name zur Erzeugung des Default JNDI-Namens herangezogen.	Name der implementierenden Klasse

```
@javax.ejb.Remote
public interface RentingAgentRemote {

    public Customer findOrPersistCustomer
        (String last, String first);

    public void setCarID(long carID);

    public void makeReservation(CreditCardDO card,
        DatePlaceDO datePlace)
        throws IncompleteConversationalStateException;
}
```

```
@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote{
    @PersistenceContext(unitName="carDatabase", type=EXTENDED)
    private EntityManager entityManager;

    @PrePassivate public void passivate(){//close socket connection}
    @PostActivate public void activate(){//open socket connection}

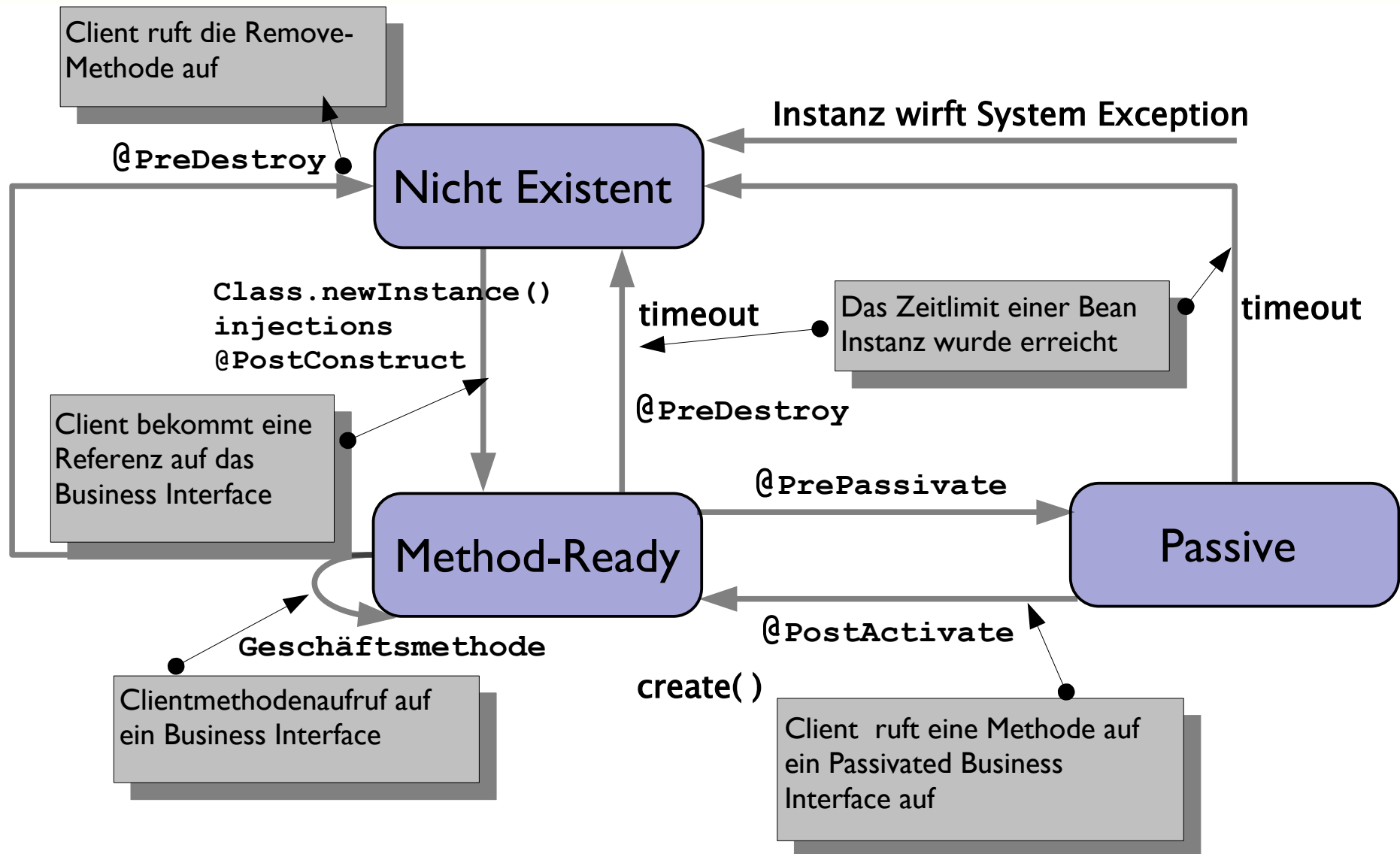
    public Customer findOrPersistCustomer(String last,String first) {
        ...
    }
    public void setCarID( long carID){
        ...
    }

    @javax.ejb.Remove
    public void makeReservation(CreditCardDO card,
        DatePlaceDO dateplace)
        throws IncompleteConversationalStateException {
        ...
    }
}
```

```
// Get the Initial Context
InitialContext jndiContext = new InitialContext();

// Lookup for the Bean
RentingAgentRemote processPayment = (RentingAgentRemote)
    jndiContext.lookup("RentingAgentBean/remote");

//Use the RentingAgent
...
```



- ▼ Der Lebenszyklus für eine Stateful Session Bean ist ähnlich wie der einer Stateless Session Bean
 - Die großen Unterschiede sind:
 - Es gibt keinen Pool aus gleichwertigen Instanzen
 - Es existieren Übergänge für Passivierung und Aktivierung
 - Durch Aufruf einer mit `@javax.ejb.Remove` markierten Methode wird die Bean-Instanz entfernt

- ▼ Stateful Session Beans können passiviert sein, wenn
 - der Zwischenspeicher des EJB Servers voll ist oder
 - die SFSB während ihrer Lebensdauer nicht benutzt wurde

- ▼ Zustände dieser SFSB werden nach der Passivierung im EJB Container bewahrt
 - Die Zustände sind primitive Werte, Serializable Objekte, und spezielle Typen wie:
 - `javax.ejb.SessionContext`,
 - `javax.naming.Context`,
 - `javax.jta.UserTransaction`,
 - `EntityManager`, `EntityManagerFactory`,
 - `javax.sql.DataSource`
 - Referenzen auf andere EJBs, ...
 - Diese Zustände werden automatisch restauriert, wenn die Bean-Instanz aktiviert wird

▼ `@javax.ejb.PrePassivate` Meta-Information

- kennzeichnet die Methode, die vor der Passivierung ausgeführt wird

▼ `@javax.ejb.PostActivate` Meta-Information

- kennzeichnet die Methode, die nach der Aktivierung ausgeführt wird

Stateful Session Bean

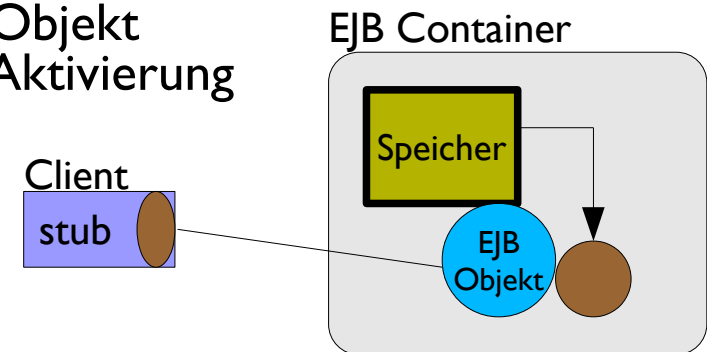
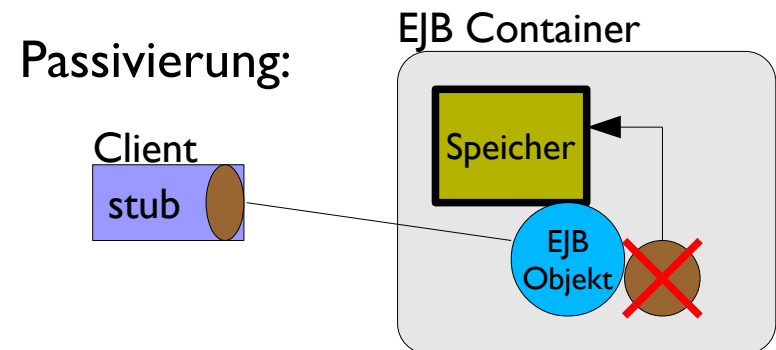
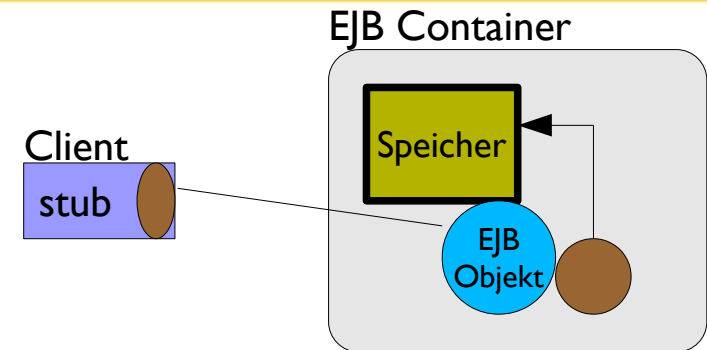
- benutzt den Mechanismus Aktivierung, um Ressourcen zu bewahren
- Langzeitgedächtnis wird benötigt

Passivierung

- Trennen der SFSB Instanz vom EJB Objekt und Speicherung ihres Zustandes

Aktivierung

- Wiederherstellung des gespeicherten Zustandes abhängig von einem bestimmten EJB Objekt



SSB

Singleton Session Bean

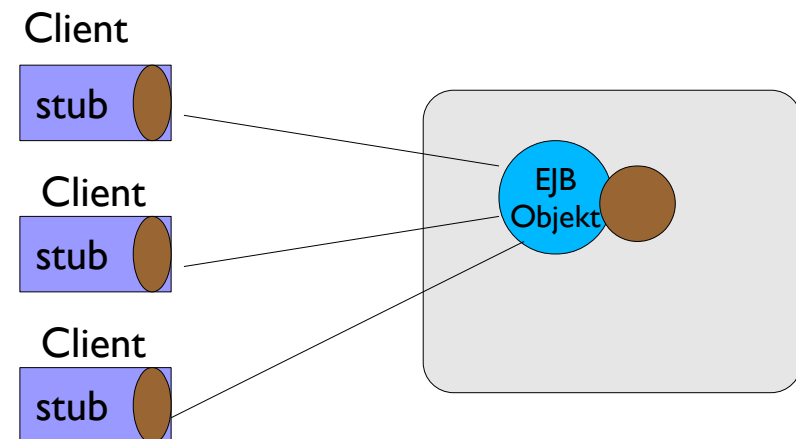
▼ Eine einzige Session Bean-Instanz pro Applikation

- Die selbe Instanz dient mehreren Requests, Sessions/Kientübergreifend
- Aber: im verteilten Fall ein Singleton pro VM!!!

▼ Defacto: Stateful Session Bean mit dem Lifecycle einer Stateless Session Bean

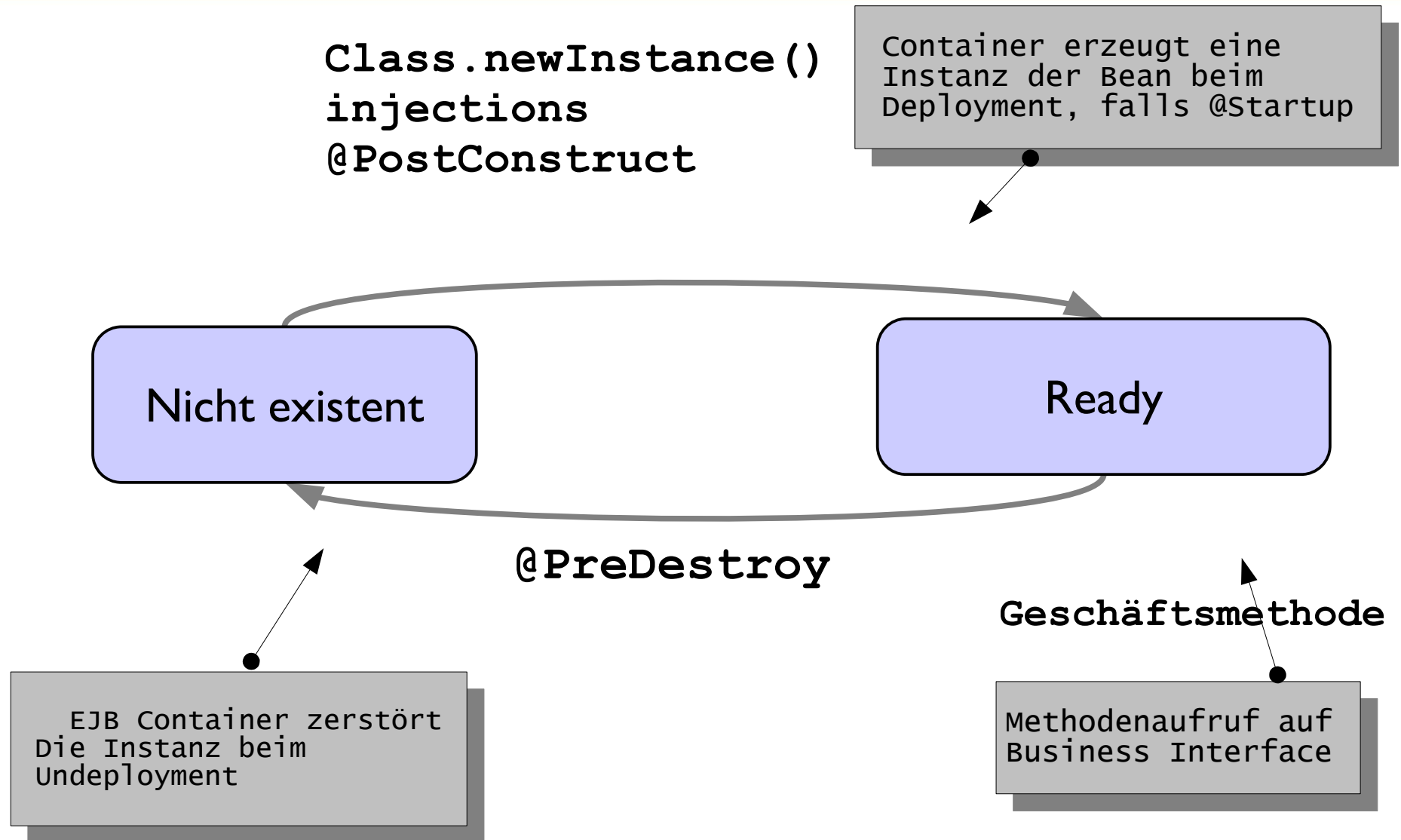
▼ Annotationen

- `@Singleton`
- `@Startup`
- `@DependsOn`



Attribute der Annotation javax.ejb.Singleton

Attribut	Typ	Bedeutung	Default
description	String	Beschreibung der EJB. Diese Daten sind zur Anzeige im Deploy-Tool des Application Server gedacht. (optional)	-
mappedName	String	Name, unter dem die EJB im Namensdienst registriert werden soll. Eine Unterstützung dieses Attributs durch die Server-Hersteller ist NICHT verpflichtend! Aus diesem Grund sollte es nicht verwendet werden. (optional)	-
name	String	Logischer Name der EJB. Auch diese Daten sind für die Anzeige im Deploy-Tool gedacht. Innerhalb des Jboss Servers wird dieser Name zur Erzeugung des Default JNDI-Namens herangezogen.	Name der implementierenden Klasse




```
@Startup
@Singleton
public class SingletonHelloImpl implements Hello {
    public String hello() {
        return "Hello";
    }
}

@Singleton
@DependsOn("SingletonHelloImpl")
public class SingletonGoodbyeImpl implements Goodbye {
    ...
}
```

■ Parallelität

- Singletons werden aus Client-Sicht immer paralll genutzt
- Eventuelle Synchronisierung notwendig
 - Container-managed-concurrency
 - Bean-managed-concurrency
- Annotationen
 - `@ConcurrencyManagement`
 - `@Lock`
 - `@AccessTimeout`

Attribute der Annotation `javax.ejb.ConcurrencyManagement`:

Attribut	Typ	Bedeutung	Default
value	Concurrency-Management-Type	Art der Steuerung des parallelen Zugriffs. Möglicher Werte sind: - <code>ConcurrencyManagementType.BEAN</code> - <code>ConcurrencyManagementType.CONTAINER</code>	CONTAINER

Attribute der Annotation `javax.ejb.Lock`:

Attribut	Typ	Bedeutung	Default
value	LockType	Art des benötigten Locks. Mögliche Werte sind: - <code>LockType.READ</code> - <code>LockType.WRITE</code>	WRITE

Attribute der Annotation javax.ejb.AccessTimeout:

Attribut	Typ	Bedeutung	Default
value	long	<p>Zeitspanne für die ein Thred bei der Anfrage eines Locks blockieren darf. Werte < -1 sind nicht erlaubt.</p> <p>Werte mit besonderer Bedeutung:</p> <p>0 – kein paralleler Zugriff erlaubt, Thread kehrt sofort mit Fehler zurück.</p>	-
unit	TimeUnit	<p>Einheit der angegebenen Zeitspanne. Gültige Werte sind:</p> <ul style="list-style-type: none">- TimeUnit.DAYS- TimeUnit.HOURS- TimeUnit.MICROSECONDS- TimeUnit.MILLISECONDS- TimeUnit.MINUTES- TimeUnit.NANOSECONDS- TimeUnit.SECONDS	MILLISECONDS

▼ Beispiel: Container Managed Concurrency

```
@Startup
@Singleton
public class SingletonCounterImpl implements Counter {
    long counter;

    @Lock(LockType.READ)
    public long getCounterValue() {
        return counter;
    }

    @Lock(LockType.WRITE)
    public long incrementAndGetCounterValue() {
        counter++;
        return counter;
    }
}
```

■ Beispiel: Bean Managed Concurrency

```
@Startup
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class SingletonCounterImpl implements Counter {
    volatile long counter;

    public long getCounterValue() {
        synchronized(this){
            return counter;
        }
    }

    public String incrementAndGetCounterValue() {
        synchronized(this){
            counter++
            return counter;
        }
    }
}
```

Asynchroner Aufruf einer Session Bean

▼ Session Beans können asynchrone Business-Methoden anbieten

- Mögliche Rückgabetypen
 - Void
 - Future<V>
- Für den Client nur indirekt unterscheidbar
- Neue Annotationen
 - @Asynchronous

Beispiel

```
public interface AsyncDoIt {
    Future<String> doIt();
}

@Stateless
public class AsyncDoItImpl implements AsyncDoIt {
    @Asynchronous
    public Future<String> doIt() {
        // Irgendwas langwieriges... ,)
        String result = ...
        return new javax.ejb.AsyncResult<String>(result);
    }
}
```

▼ Abbruch der Verarbeitung in Nachhinein möglich

- `Future<V>.cancel(boolean mayInterruptIfRunning)`
- `SessionContext.isCancelled()`

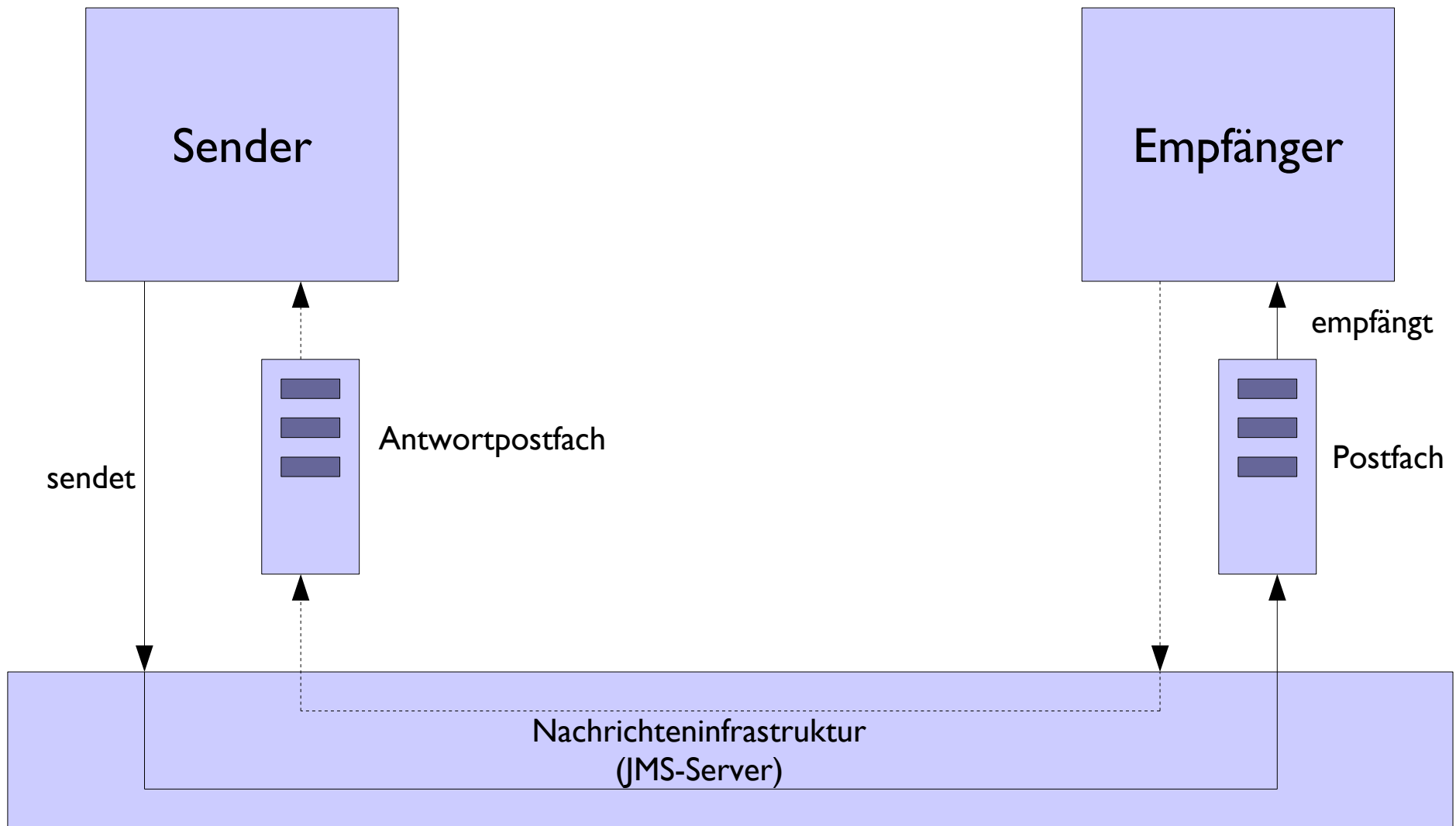
▼ Exceptions

- Exception wird beim Aufruf von `Future<V>.get()` geworfen
 - `java.lang.concurrent.ExecutionException.getCause()`
- **Void-Methoden** werfen keine Exceptions

Teil III

Messaging

Messaging Grundlagen (JMS)



▼ Messaging ermöglicht zwei oder mehreren Anwendungen, Informationen auszutauschen

- in Form von Nachrichten
- über virtuelle Kanäle (Destinations)
- in Enterprise-Messaging Systemen (Message-Oriented Middleware)
- Consumer kann
 - asynchron oder
 - synchron implementiert werden

▼ Messaging besteht aus

- Message Producers,
- Message Middleware und
- Message Consumers

- ▼ Asynchrone Kommunikation
- ▼ Entkopplung von Systemen
- ▼ Zuverlässigkeit
- ▼ Unterstützung für vielfache Sender und Empfänger

- ▼ JMS ist eine anbieterneutrale Schnittstelle für
 - das Erstellen,
 - das Senden,
 - das Empfangen,
 - und das Lesen von Nachrichten
- ▼ Aufgebaut aus:
 - JMS Provider (Server)
 - JMS Clients
- ▼ JMS API kann für den Zugriff auf Enterprise-Messaging (MOM) Systeme benutzt werden

▼ JMS Provider (oder Messaging System)

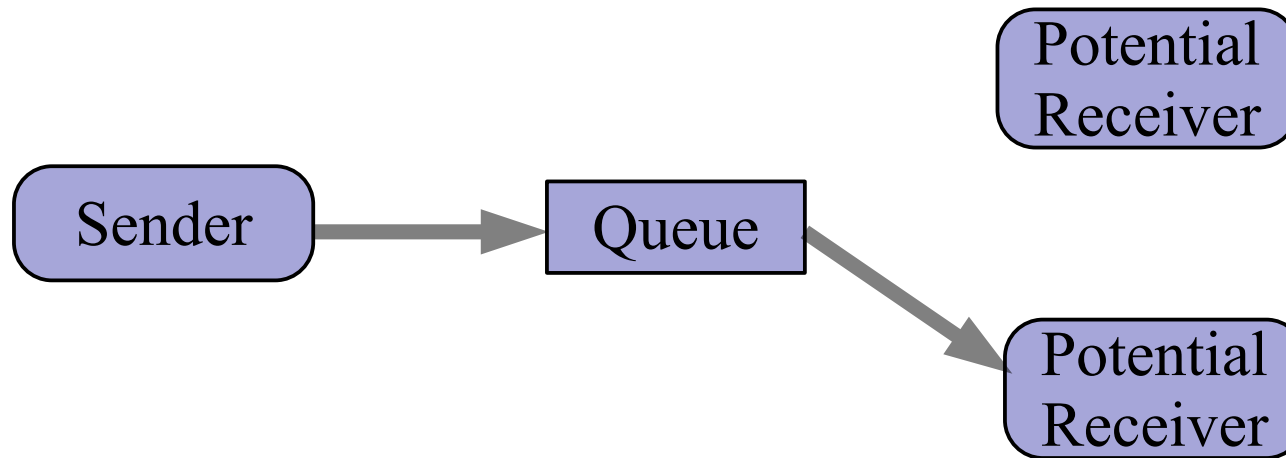
- behandelt das Routing und die Lieferung von Nachrichten

▼ JMS Client

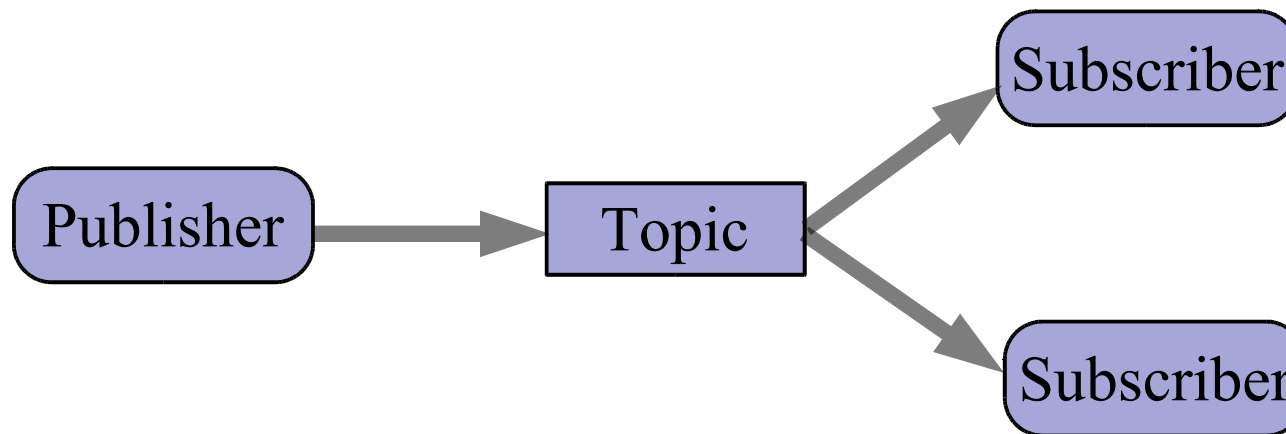
- *Message producer* : sendet Nachrichten
- *Message consumer* : empfängt Nachrichten

▼ Verwaltete Objekte

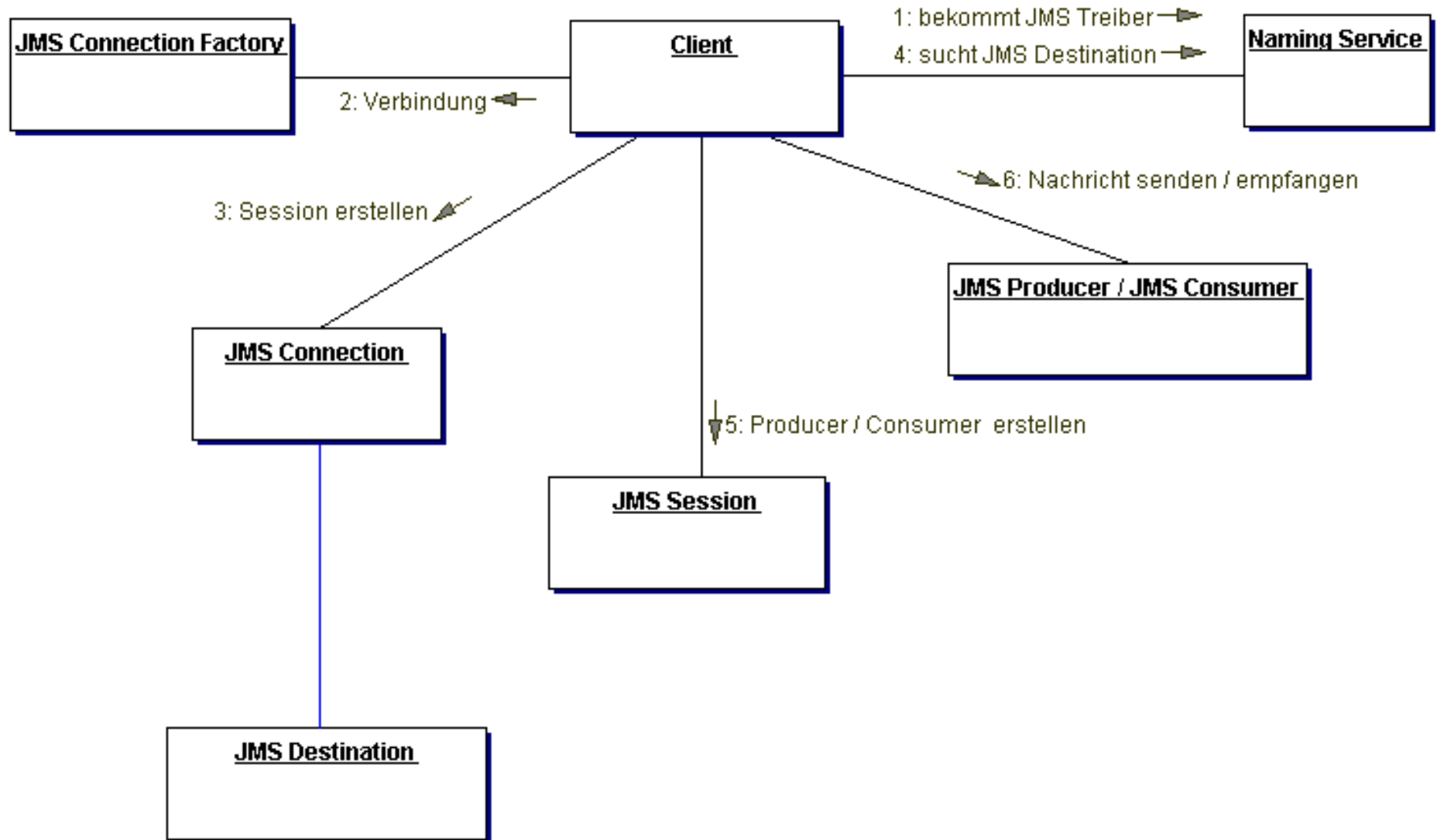
- Vorkonfigurierte JMS Objekte, die von einem Administrator für JMS Clients erstellt werden
- JMS Connection Factories
 - `javax.jms.ConnectionFactory`
- JMS Destinations
 - `javax.jms.Destination`



- ▶ One-to-one / Many-to-one
- ▶ Destination (virtueller Kanal): Queue (`javax.jms.Queue`)
- ▶ Eine Nachricht kann nur einmal konsumiert werden
- ▶ Nachrichten werden zwischengespeichert, falls diese nicht übergeben wurden



- ▼ One-to-many / Many-to-many
- ▼ Destination (virtueller Kanal): Topic (`javax.jms.Topic`)
- ▼ Eine Nachricht wird an alle Abonnenten verteilt
- ▼ Ein dauerhaftes Abonnement ist möglich



- ▼ Lokalisierung der JMS ConnectionFactory Instanz mittels JNDI
- ▼ Nutzung der JMS Connection, um eine Verbindung zu erstellen
- ▼ Nutzung von JMS Session, um eine Sitzung zu erstellen
- ▼ Lokalisierung einer JMS Destination mittels JNDI
- ▼ Ein JMS Producer oder ein JMS Consumer wird erzeugt
- ▼ Nachricht wird gesendet oder empfangen

```
//Erzeuge einen JNDI Kontext
InitialContext jndiCtx = new InitialContext(
    System.getProperties());

//1: Erhalte eine ConnectionFactory durch JNDI
ConnectionFactory factory =
    (ConnectionFactory)
        jndiCtx.lookup("TopicConnectionFactory");
```

- ▶ Der InitialContext erwartet JNDI Provider Informationen, die in eine Datei `jndi.properties` geschrieben werden können
- ▶ Der JNDI Name für die `ConnectionFactory` ist `"TopicConnectionFactory"`

```
//2: Benutze Connection Factory,  
//    um eine JMS Connection zu erzeugen  
Connection connection = factory.createConnection();  
  
//3: Benutze Connection,  
//    um eine Session zu erzeugen  
Session session = connection.  
    createSession(false, Session.AUTO_ACKNOWLEDGE);
```

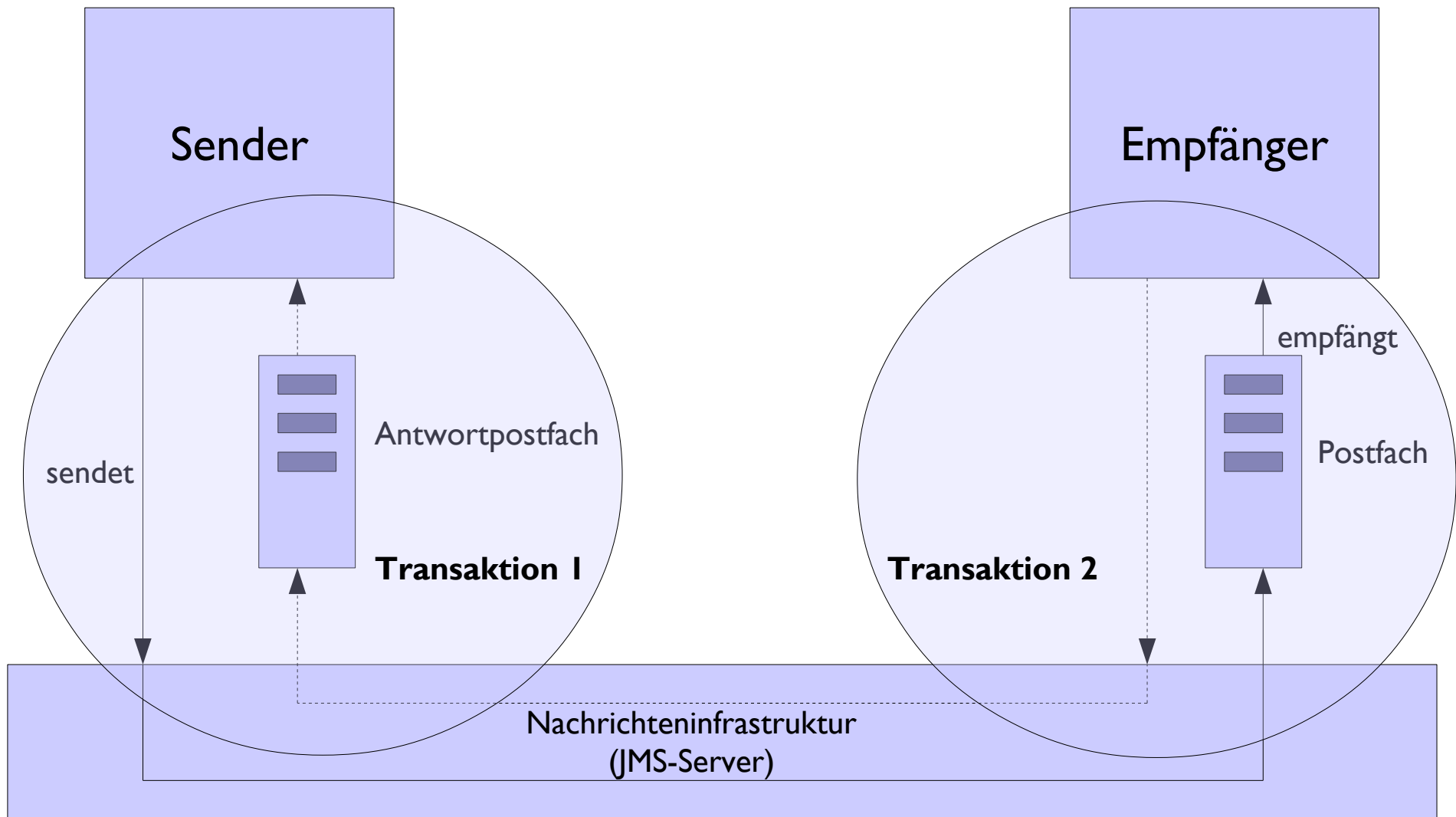
- ▼ Wenn die Session erzeugt wird, werden zwei Parameter weitergegeben
 - Transaktionsverhalten der Session (transaktional / nicht-transaktional)
 - Acknowledge Mode
 - AUTO_ACKNOWLEDGE
 - CLIENT_ACKNOWLEDGE
 - DUPS_OK_ACKNOWLEDGE

```
//4: Erhalte das Thema für das Tutorial
Destination topic =
    (Destination)jndiCtx.lookup("topic/tutorial");

//5: Erzeuge einen Sender für das Thema
MessageProducer publisher = session.createProducer(topic);

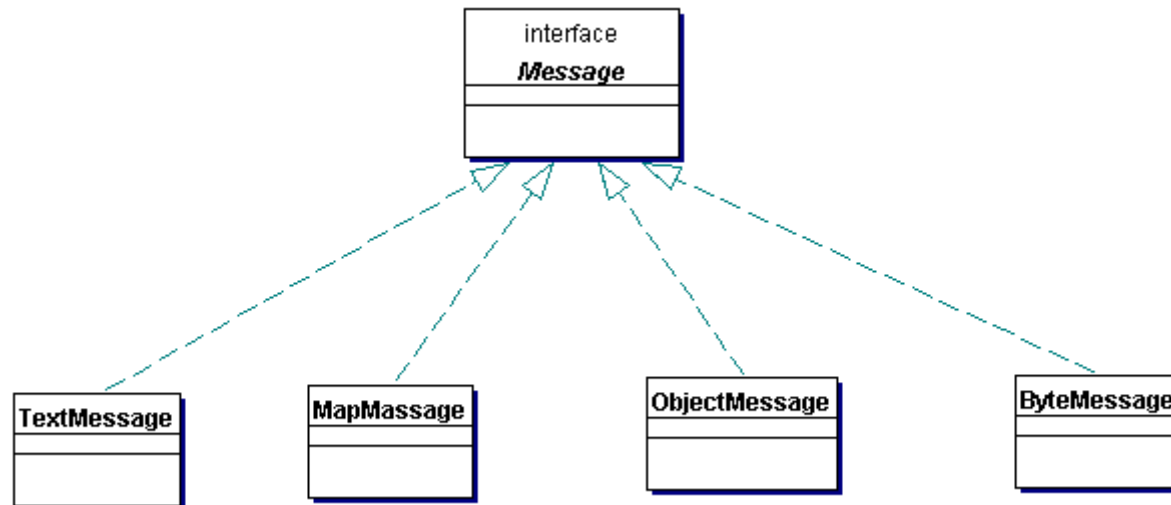
//6: Erzeuge eine Text Nachricht
TextMessage message = session.createTextMessage();
message.setText("This is a simple message.");

//7: Sende die Text Nachricht
publisher.send(message);
```

▼ JMS definiert vier Message Typen

- `javax.jms.Message` ist das Basisinterface
- `javax.jms.TextMessage` kann nur `java.lang.String` annehmen
- `javax.jms.MapMessage` kann nur Namen-Werte Paare annehmen
- `ObjectMessage` kann nur serialisierbare Objekte annehmen
- `ByteMessage` kann nur Arrays von Bytes annehmen



```
@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote{

    Destination topic;

    // ...

    @javax.ejb.Remove
    public void sendMessage(String message) throws JMSEException{
        InitialContext initialContext = new InitialContext();
        ConnectionFactory factory = (ConnectionFactory) initialContext
            .lookup("TopicConnectionFactory");
        Connection connection = factory.createConnection();
        Session session = connection.
            createSession(true, Session.AUTO_ACKNOWLEDGE );
        MessageProducer publisher = session.createProducer(topic);
        TextMessage textMsg = session.createTextMessage();
        textMsg.setText(message);
        publisher.send(textMsg);
        connection.close();
    }
}
```

MDBs

Message-Driven Beans

▼ MDBs

- sind asynchrone Nachrichtenkonsumenten
- ermöglichen die parallele Verarbeitung von Nachrichten
- sind nicht persistent

▼ Eigenschaften von MDB Instanzen

- serverseitig
- zustandslos (aus der Client-Sicht),
- transaktional

▼ Clients können nicht auf eine MDB direkt zugreifen

▼ Bestandteile einer MDB

- Bean Klasse
- Messagelister Interface
- optionaler Deployment Deskriptor

- ▼ MDB Instanzen sind einer Queue bzw. einem Topic zugeordnet und werden bei der Ankunft einer Nachricht vom Container aufgerufen
- ▼ MDBs müssen ein Messagelistener Interface implementieren
(z.B. `javax.jms.MessageListener`)
- ▼ MDBs werden mit der `@javax.ejb.MessageDriven` gekennzeichnet
- ▼ MDBs unterstützen die Abarbeitung von asynchronen Nachrichten aus beliebigem Messaging-System
- ▼ MDBs benutzen JCA 1.5 für die Integration
- ▼ Konfiguration über das Attribut `activationConfig()` der Annotation `@MessageDriven`

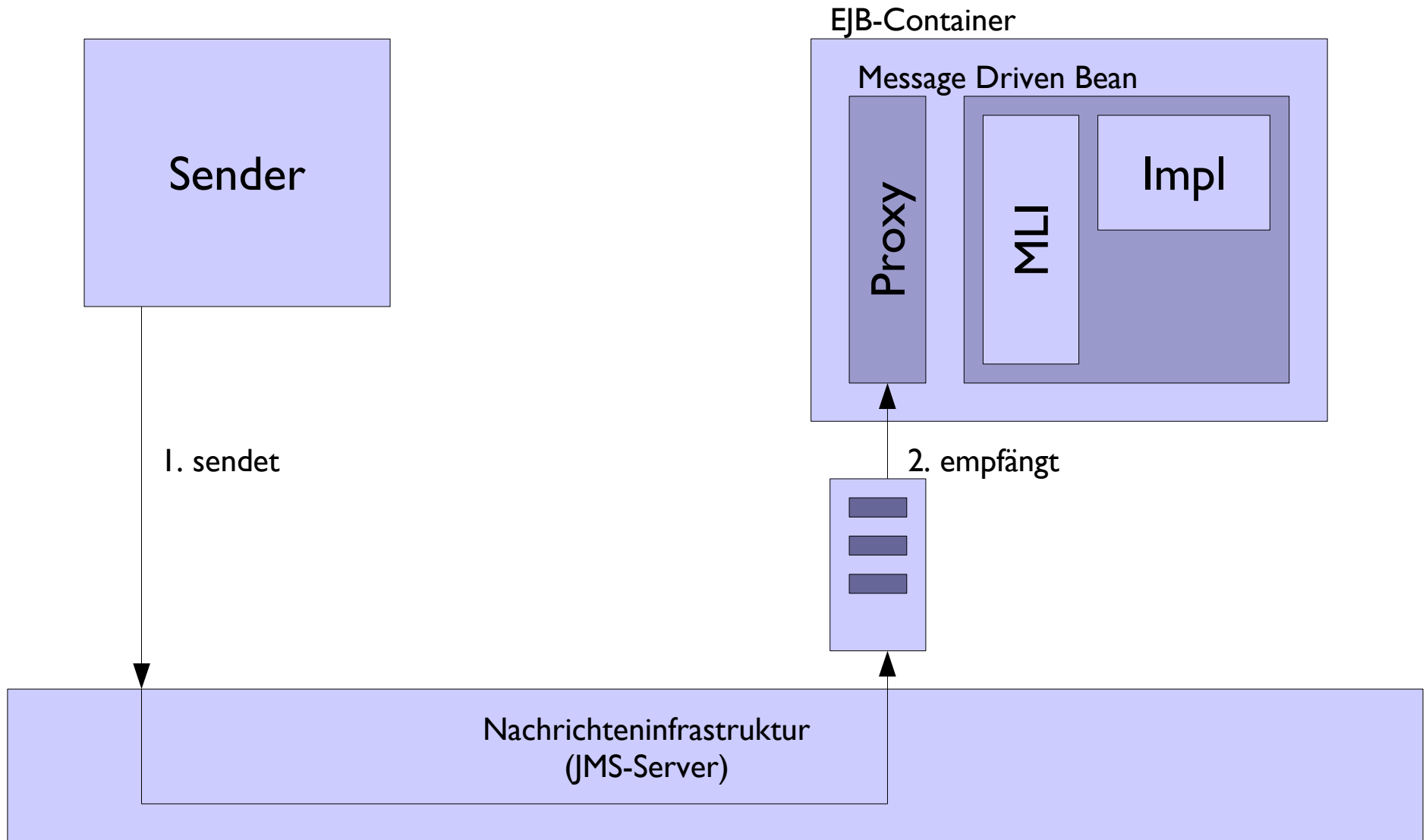
```
package de.mathema.mdb;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;

@javax.ejb.MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(
        propertyName = "destination",
        propertyValue = "queue/A"),
    @ActivationConfigProperty(
        propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge")
})
public class ReservationHandlerBean
    implements javax.jms.MessageListener {
    public void onMessage(Message msg) {
        // Geschäftslogik
        ...
    }
}
```

```
package javax.jms;  
  
public interface MessageListener{  
    void onMessage(Message message) ;  
}
```

▼ MessageListener.onMessage()

- nur für JMS Message Driven Bean.
- wird vom MDB Container aufgerufen, wenn eine neue Nachricht in Queue/Topic ankommt
- beinhaltet die Geschäftslogik der MDB.
- realisiert i.d.R. den Einsprung in die weitere Anwendung



Attribut	Typ	Bedeutung	Default
activationConfig	ActivationConfigProperty[]	Provider-spezifische Properties für die Konfiguration der vom Container realisierten Anbindung der Bean an das Nachrichtensystem.	Leeres Array
description	String	Beschreibung der EJB. Diese Daten sind zur Anzeige im Deploy-Tool des Application-Servers	-
mappedName	String	Name, unter dem die EJB im Namensdienst registriert werden soll. Eine Unterstützung dieses Attributs durch die Server-Hersteller ist NICHT verpflichtend! Aus diesem Grund sollte es nicht verwendet werden. (optional)	-
messageListenerInterface	Class	Klasse des implementierenden Interfaces zur Entgegennahme der Nachrichten. Diese Property wird nur benötigt, wenn die Bean neben diesem Interface noch weitere implementiert.	Object.class
name	String	Logischer Name der EJB. Auch diese Daten sind für die Anzeige im Deploy-Tool gedacht. Innerhalb des Jboss-Servers wird dieser Name zur Erzeugung des Default JNDI-Namens herangezogen.	Name der annotierten Klasse

- ▼ EJB 3.0 definiert eine Menge von Eigenschaften für die JMS-basierten MDBs, um den Messaging-Dienst zu beschreiben
 - Destination Type
 - Message Selector
 - Acknowledge Mode
 - Subscription Durability
- **@ActivationConfigProperty** definiert Namen-Werte Paare, um die MDBs zu konfigurieren

Attribut	Typ	Bedeutung	Default
propertyName	String	Name des Konfigurationsparameters	keiner, muss angegeben werden
propertyValue	String	Wert des Konfigurationsparameters	keiner, muss angegeben werden

```
@javax.ejb.MessageDriven(activationConfig={  
    @ActivationConfigProperty(  
        propertyName="messageSelector",  
        propertyValue="MessageFormat='Version 3.4'") })
```

- ▶ ist ein JMS Konzept, welches die Filterung von Nachrichten an das Ziel erlaubt
- ▶ benutzt Message Eigenschaften und Headers als Kriterium in einem bedingten Ausdruck (boolean)
- ▶ Syntax: Subset SQL-92
- ▶ Bezeichner: Name einer Eigenschaft oder JMS Header Name

```
@javax.ejb.MessageDriven(activationConfig={  
    @ActivationConfigProperty(  
        propertyName="acknowledgeMode",  
        propertyValue="Auto-acknowledge")  
    })
```

▼ auto-acknowledge

- MDB Container sendet beim Entnehmen der Message aus der Queue eine Empfangsbestätigung (einmalige Zustellung der Nachricht ist garantiert).

▼ dups-ok-acknowledge

- Nachricht kann potentiell mehrfach zugestellt werden (Duplikate müssen im Bean-Code verarbeitet werden).

▼ client-acknowledge

- nicht erlaubt!

```
@javax.ejb.MessageDriven(activationConfig={
    @ActivationConfigProperty(
        propertyName="destinationType",
        propertyValue="javax.jms.Topic")
    @ActivationConfigProperty(
        propertyName="subscriptionDurability",
        propertyValue="Durable")
})
```

- ▶ Eine Message-Driven-Destination, welche den Zieltyp (**Queue** oder **Topic**) enthält
- ▶ Bei Verwendung eines **javax.jms.Topic**, kann die Verbindung als **Durable** oder **NonDurable** deklariert werden

```
<?xml version="1.0"?>
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>ReservationHandlerBean</ejb-name>
      <ejb-class>
de.mathema.mdb.ReservationHandlerBean
      </ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <transaction-type>Container</transaction-type>
      <message-destination-type>
        javax.jms.Queue
      </message-destination-type>
      <activation-config>
        <activation-property>
          <activation-config-property-name>destinationType
          </activation-config-property-name>
          <activation-config-property-value>javax.jms.Queue
          </activation-config-property-value>
        </activation-property>
        ...
      </activation-property>
      </activation-config>
    </message-driven>
  </enterprise-beans></ejb-jar>
```

```
package javax.ejb;
public interface EJBContext{
    public Principal getCallerPrincipal();
    public boolean isCallerInRole(String roleName);
    public UserTransaction getUserTransaction() throws IllegalStateException;
    public Object lookup(String name);
    ...
}

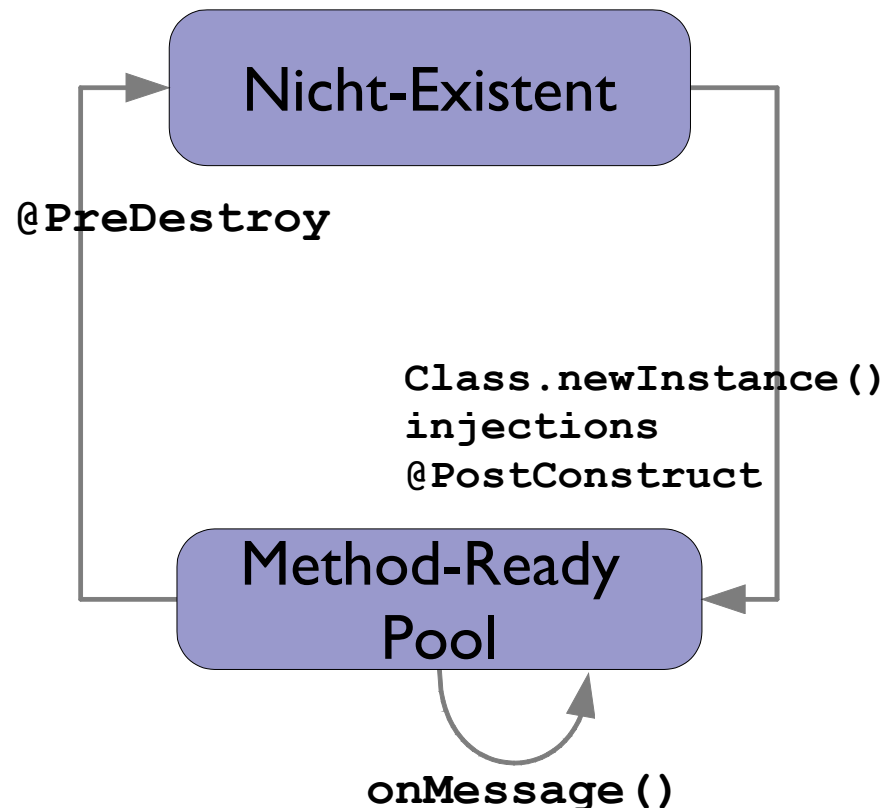
public interface MessageDrivenContext extends EJBContext {
}
```

- ▼ **MessageDrivenContext** ist ein leeres Interface, das alle Methoden von **EJBContext** erbt.
- ▼ kann durch **@javax.annotation.Resource** injiziert werden

- Lebenszyklus einer Message-Driven Bean verhält sich ähnlich wie der von Stateless Session Beans

- OnMessage () :**

- nach dem Eintreten der Nachricht wird **onMessage** von MDB Container initialisiert

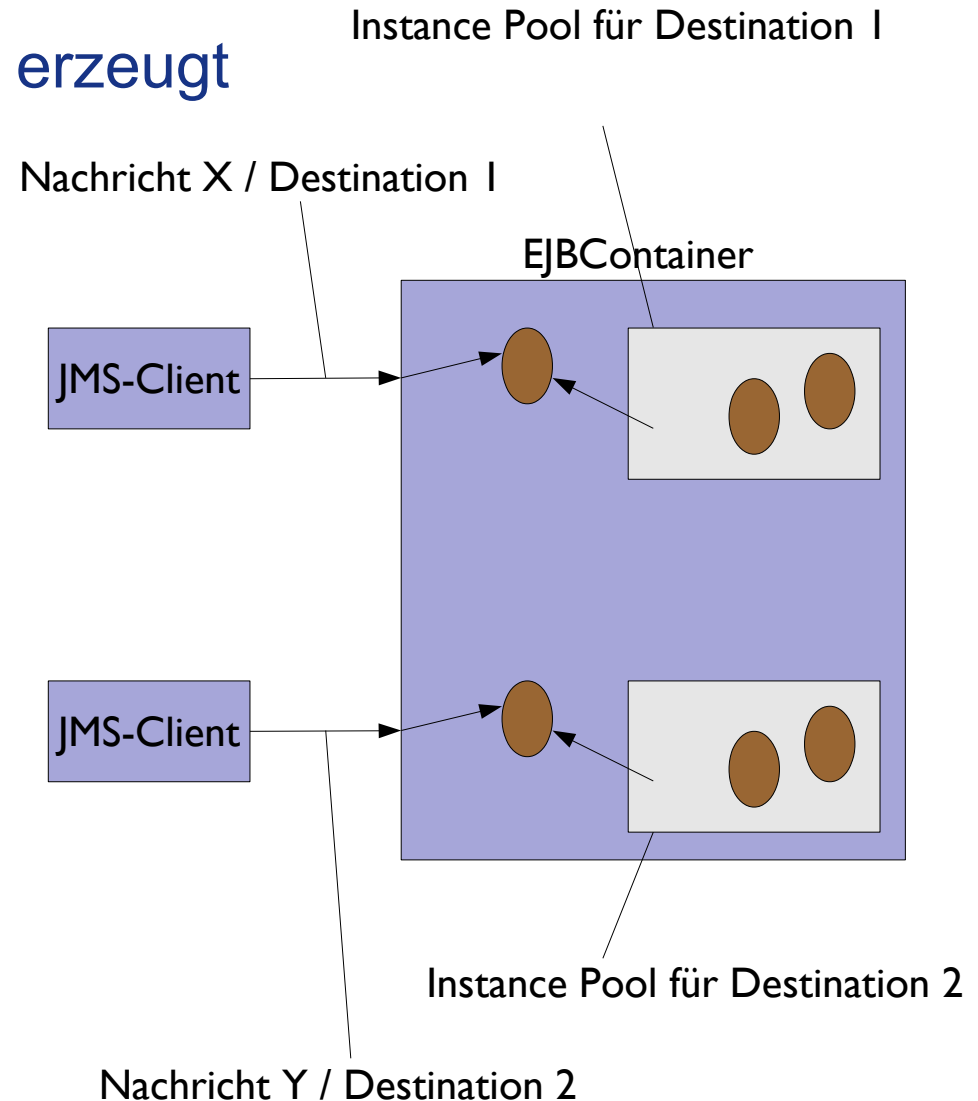


▼ Pool wird nach Typen von MDBs erzeugt

▼ Message Destination ist eine eindeutige Adresse für Senden und Empfangen von Nachrichten

▼ EJB Container

- leitet die Nachrichten in die richtige Kanäle
- wählt die richtigen JMS-MDBs für die Bearbeitung von Nachrichten
- gibt die Instanz in den Pool zurück, falls diese nicht mehr gebraucht wird



▼ MDBs sind als EJBs single-Threaded

- pro Instanz erfolgt zu einem Zeitpunkt immer nur die Abarbeitung einer Nachricht.

▼ ABER mehrere MDBs arbeiten für eine Queue/Topic

- Parallele Abarbeitung verschiedener Nachrichten ist möglich
- Reihenfolge der Nachrichten ist nicht garantiert

▼ Ablauf beim Erhalt einer Poison Message

- Nachricht wird zugestellt
- Empfänger entnimmt die Nachricht
- Empfänger stößt bei der Abarbeitung der Nachricht auf einen Fehler
- Entnahme der Nachricht wird wg. Tx-Rollback rückgängig gemacht
- Empfänger setzt wieder neu auf
- Empfänger entnimmt die Nachricht
- Empfänger stößt bei der Abarbeitung der Nachricht auf einen Fehler
- ...

▼ Typische Voraussetzungen für eine Poison Message sind:

- Die MDB wirft bei Abarbeitung eine System Exception
- Sie verwendet das Transaktionsattribut “Required“
=> Rollback wird durchgeführt
- Das Ziel ist eine Queue, oder es wird eine dauerhafte “Subscription“ verwendet.
- Nachrichten werden nicht ungültig (`msg.getJMSExpiration()=0`).

▼ Symptome einer Poison Message:

- Die Verarbeitung der MDB schlägt fehl und es wird eine Exception geworfen, die eine Rollback initiiert.
- Der JMS-Provider wartet auf Bestätigung und sendet die selbe Nachricht nochmals.

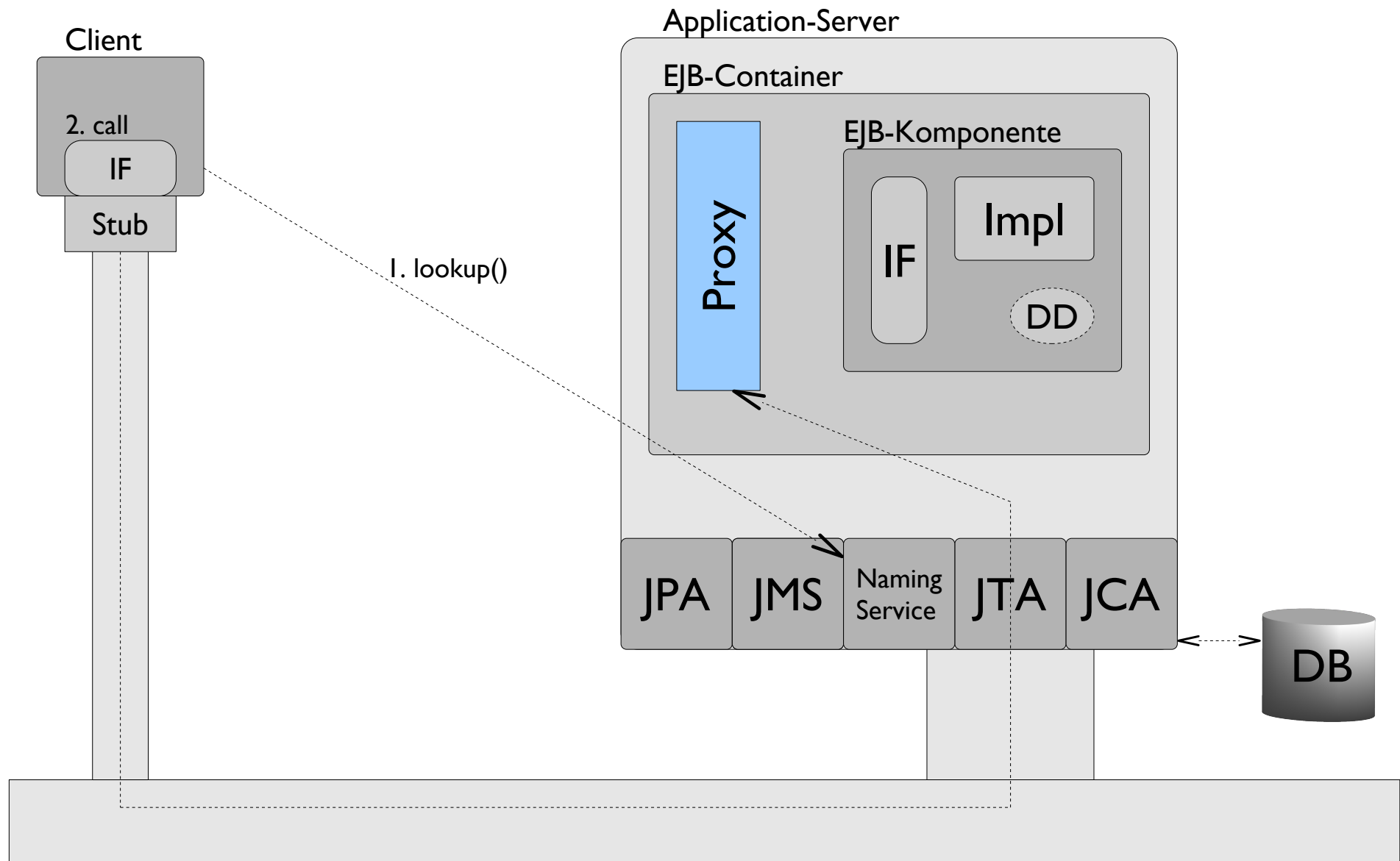
▼ Vorsichtsmaßnahmen um Poison Messages zu vermeiden:

- Einsatz von Bean Managed Transactions
- Setzen des Transaktionsattributs “NotSupported“
- Keine System Exceptions aus MDB
 - Application Exceptions sind hier nicht möglich
- Im Falle eines Fehlers:
 - Fehlernachricht loggen und normal fortfahren
 - Nachricht an eine andere Queue zur weiteren Behandlung weiterleiten
- Einsatz von Poison Message Queues
 - Nachrichten werden nach einer bestimmten Anzahl an Zustellversuchen als Poison Messages markiert.
 - Eine Nachricht wird nach 3-5 Zustellversuchen als Poison Message bezeichnet.

Teil IV

Interceptoren

- ▼ Wiederkehrende Aufgaben bei client-seitigen Aufrufen „verschmutzen“ fachlichen Code
 - Tracing
 - Accounting
 - Zeitmessung
 - ...
- ▼ Lösung: Auslagerung dieses Codes in Interceptoren
 - bereits vorgestellt: Lifecycle Callback Event Interceptor
 - jetzt verallgemeinert: Business Method Interceptor



- ▼ Eingriff vor bzw. nach dem Methodenaufruf („around invoke“)
- ▼ Implementierung durch einfache Java-Klasse
- ▼ Definition von Interceptoren möglich für
 - alle Businessinterfaces eines ejb-jars
 - Lifecycle-Event-Callback-Methoden einer EJB-Implementierung
 - Timer-Methoden einer EJB-Implementierung
- ▼ **Verwendete Annotationen**
 - `@javax.interceptor.AroundInvoke`
 - `@javax.interceptor.AroundTimeout`
 - `@javax.interceptor.Interceptors`

```
@javax.ejb.Stateless
@javax.interceptor.Interceptors({MyTracingInterceptor.class})
public class ProcessPaymentBean implements
ProcessPaymentLocal, ProcessPaymentRemote {
    ...
}
```

```
public class MyTracingInterceptor{

    @javax.interceptor.AroundInvoke
    public Object trace(InvocationContext ic) throws Exception{
        log.trace("Invoking method: "+ic.getMethod());
        return ic.proceed();
    }
}
```

```
<interceptor-binding>
  <ejb-name>*</ejb-name>
  <!-- Mehrere Interceptoren werden in Reihenfolge ausgefuehrt-->
  <interceptor-class>MyTracingInterceptor</interceptor-class>
</interceptor-binding>
```

```
<interceptor-binding>
  <ejb-name>ProcessPaymentBean</ejb-name>
  <interceptor-class>MyTracingInterceptor</interceptor-class>
</interceptor-binding>
```

```
<interceptor-binding>
  <ejb-name>ProcessPaymentBean</ejb-name>
  <interceptor-class>MyTracingInterceptor</interceptor-class>
  <method-name>byCreditCard</method-name>

  <!-- Methodensignatur noch genauer spezifizierbar
  <method-params>
    <method-param>java.lang.String</method-param>
    <method-param>java.lang.String</method-param>
  </method-params>
-->
```

Teil V

Injizieren von Service Objekten

- ▼ Jeder EJB Container in einem Applikation Server besitzt ein internes Register:
 - Enterprise Naming Context (ENC)
- ▼ Enterprise Naming Context
 - wird von JNDI implementiert
 - ist das persönliche Adressbuch des EJB Containers
- ▼ Enterprise Naming Context beinhaltet Adressen von vielen Java EE Diensten, die von EJB Objekten benutzt werden können
 - `javax.ejb.TimerService`
 - `javax.transaction.UserTransaction`,
 - `org.omg.CORBA.ORB`
 - JNDI ENC kann auch andere EJB Objekte, JMS Objekte, JCA Ressourcen und primitive Typen aufnehmen

▼ Zwei Wege:

- über ejb-jar.xml Deployment Deskriptor oder
 - `<ejb-name>`, `<ejb-local>`, `<ejb-ref-name>`, `<ejb-ref-type>`, `<local>`, und `<ejb-link>`
- über Meta-Informationen
 - `@javax.ejb.EJB`

```
package javax.ejb;
public @interface EJB{
    String name() default "";
    Class beanInterface() default
        java.lang.Object.class;
    String beanName() default "";
    String mappedName() default "";
    String description() default "";
}
```

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>RentingAgentBean</ejb-name>
      <ejb-local-ref>
        <ejb-ref-name>
          ejb/ProcessPayment
        </ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>
          de.mathema.slsb.ProcessPaymentLocal
        </local>
        <ejb-link>ProcessPaymentBean</ejb-link>
      </ejb-local-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

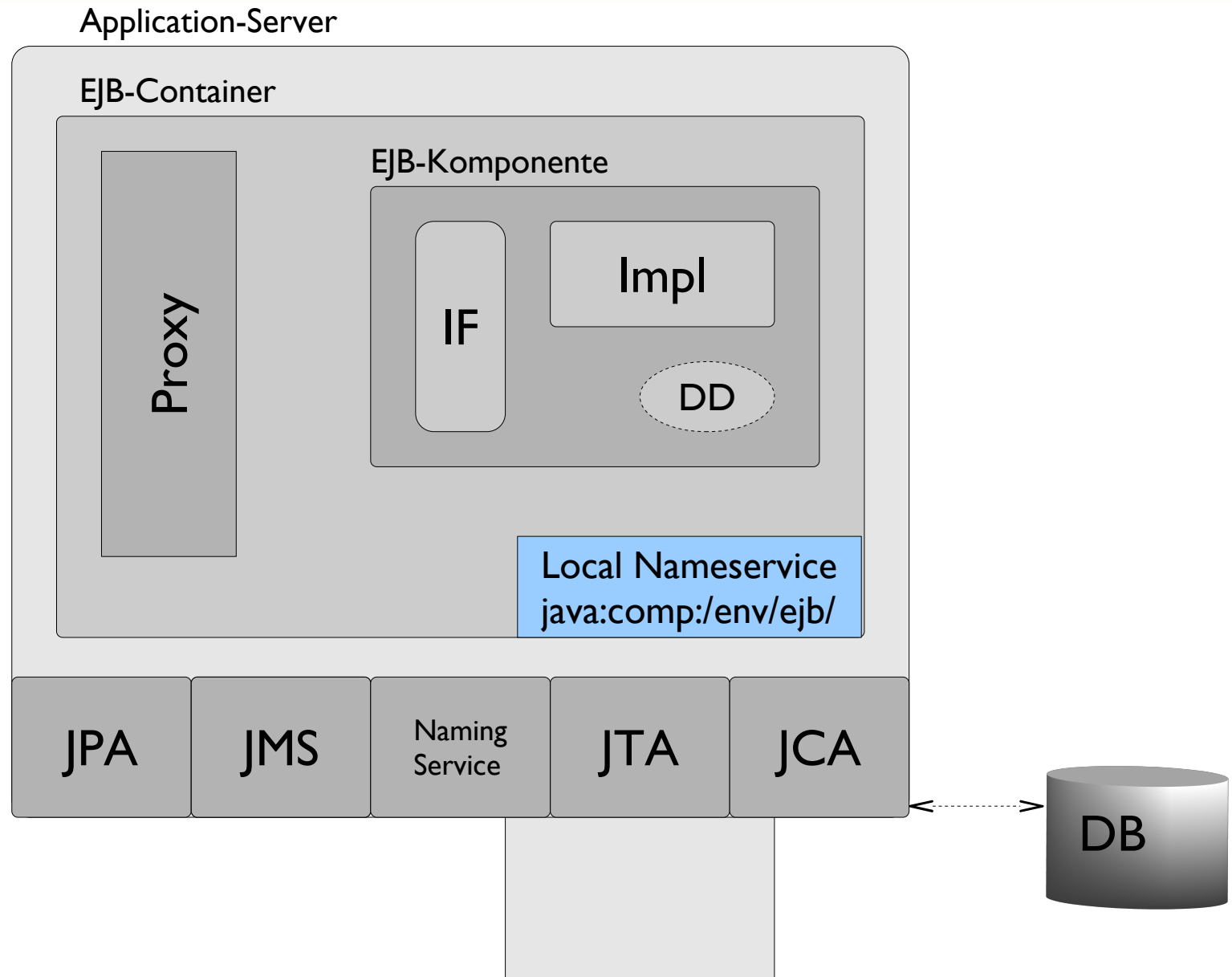


```
package de.mathema.sfsb;
...
@javax.ejb.Stateful
@javax.ejb.EJB(name="ejb/ProcessPayment",
    beanInterface=ProcessPaymentLocal.class,
    beanName="ProcessPaymentBean")
public class RentingAgentBean
    implements RentingAgentRemote{

    @javax.persistence.PersistenceContext
    (unitName="carDatabase", type=EXTENDED)
    private EntityManager entityManager;

    ...

}
```



- ▼ Durch den Kontext `java:comp:/env` mit dem Namen der Bean
 - `JndiContext`
`lookup("java:comp:/env/ejb/ProcessPayment")`
- ▼ Durch Nutzung von Unter-Interfaces des `javax.ejb.EJBContext` Interface
 - `javax.ejb.SessionContext`
 - `javax.ejb.MessageDrivenContext`
 - `SessionContext` und `MessageDrivenContext` können in Session oder Message-Driven Beans mittels `@javax.annotation.Resource` injiziert werden

- ▼ JNDI-Lookups sind unnötiger technischer Overhead
- ▼ Daher: Einführung eines Dependency Injection-Mechanismus
- ▼ Durch Meta-Informationen
 - **@javax.ejb.EJB**
 - Injiziert EJB Objekt Stubs (kann auf Felder oder Setter-Methoden angelegt werden)
 - **@javax.annotation.Resource**
 - Injiziert Java EE Dienste
 - Injiziert Environment Entries

```
@javax.ejb.Stateful
@javax.ejb.EJB(name="ejb/ProcessPayment",
    beanInterface=ProcessPaymentLocal.class,
    beanName="ProcessPaymentBean")
public class RentingAgentBean implements RentingAgentRemote{

    @javax.annotation.Resource
    private javax.ejb.SessionContext ejbContext;
    ...
    @javax.ejb.Remove
    public void makeReservation(CreditCardDO card, TimePlaceDO place)
        throws IncompleteConversationalStateException {
        try{
            ...
            ProcessPaymentLocal payment =
                ProcessPaymentLocal ejbContext.lookup("ejb/ProcessPayment");
            payment.byCreditCard
                (this.customer, card, this.car.getPrice());

        }catch (EJBException e) {
            throw new EJBException(e);
        }
    }
}
```

```
@javax.ejb.Stateful
public class RentingAgentBean
    implements RentingAgentRemote{

    @javax.ejb.EJB ProcessPaymentLocal payment;
    ...
}
```

```
@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote{
    private ProcessPaymentLocal payment;
    ...
    @javax.ejb.EJB
    public void setProcessPayment(ProcessPaymentLocal payment){
        this.payment = payment;
    }
    ...
}
```

```
package de.mathema.sfsb;  
...  
@javax.ejb.Stateful  
public class RentingAgentBean  
    implements RentingAgentRemote{  
  
    ...  
    @javax.annotation.Resource(mappedName="java:/DefaultDS")  
    private javax.sql.DataSource oracleDS;  
    ...  
}
```

```
@javax.ejb.Stateful
public class RentingAgentBean
    implements RentingAgentRemote{

    @javax.ejb.EJB ProcessPaymentLocal payment;
    ...
}
```

```
@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote{
    private ProcessPaymentLocal payment;
    ...
    @javax.ejb.EJB
    public void setProcessPayment(ProcessPaymentLocal payment){
        this.payment = payment;
    }
    ...
}
```


▼ Environment Entries sind Konfigurationsparameter

- werden „von aussen“ vorgegeben
- DD kann im nachhinein verändert werden
- Annotation dient nur als „default“ und kann per XML überschrieben werden

```
...  
    <env-entry>  
        <env-entry-name>creditCardMinAmount</env-entry-name>  
        <env-entry-type>java.lang.Integer</env-entry-type>  
        <env-entry-value>25</env-entry-value>  
        <injection-target>  
            <injection-target-name>creditCardMinAmount</injection-target-name>  
        </injection-target>  
    </env-entry>  
...
```

Teil VI

Java Persistence API

Grundlagen

▼ Eigene Spezifikation seit Java EE 5

- Java Persistence API 1.0
- Abstraktion über JDBC

▼ Spezifikation in JEE 6/7

- Java Persistence API 2.0/2.1

▼ Standard für Object-to-Relational-Mapping (ORM)

- JPA kann Java Objekte automatisch auf eine relationale Datenbank abbilden

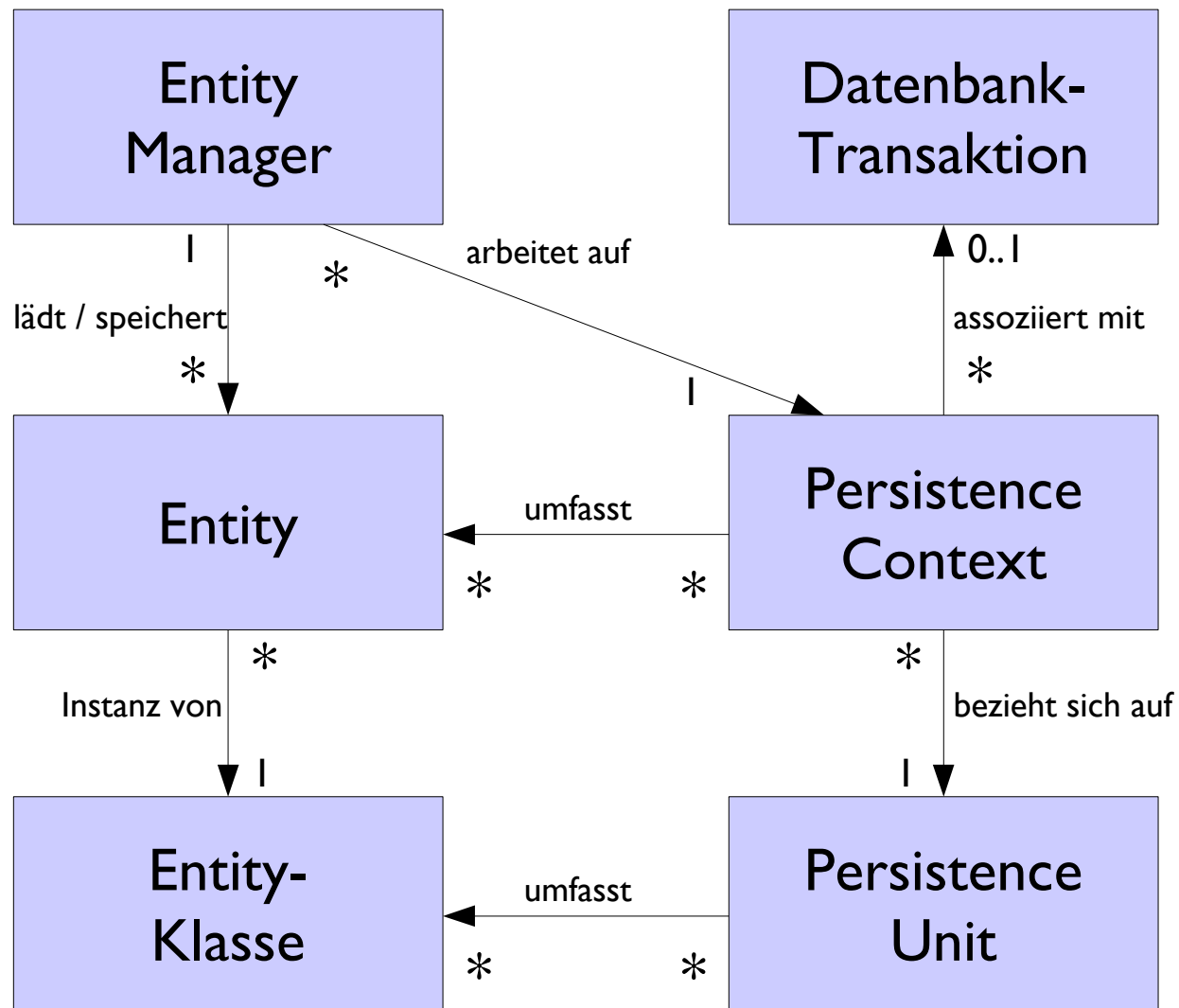
▼ Bereitstellung einer Abfragensprache (JPQL), die mit Java Objekten arbeitet

▼ In JPA spielt `javax.persistence.EntityManager` eine der wichtigsten Rollen

- ▼ Leichtgewichtiges Domänenobjekt
- ▼ Persistentes Objekt
- ▼ JavaBeans Klasse
 - mit einigen Regeln
- ▼ Plain Old Java Object (POJOs)
- ▼ Der Operator new()
 - alloziert Entities
 - new Customer();

```
@Entity
public class Customer {
    private int id;
    private String name;

    @Id @GeneratedValue
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String n) {
        this.name = n;
    }
}
```



▼ Persistence Context

- Menge von attached/managed Objekten
- wird von EntityManager verwaltet
- wenn ein Persistence Context geschlossen ist, werden alle Managed Persistence Objekte als
 - Detached und
 - Unmanaged markiert

▼ Typen von Persistence Contexts

- **Transaction-Scoped**
 - Existiert für die Dauer einer Transaktion
- **Extended**
 - Existiert über Transaktionsgrenzen hinweg

- ▶ Applikation Server Managed Persistence Contexts können Transaction-Scoped sein.
- ▶ EntityManager Instanzen injizieren mit dem
 - `@javax.persistence.PersistenceContext` oder
 - seiner ***XML*** äquivalenten Beschreibung
 - können Transaction-Scoped sein

```
@javax.persistence.PersistenceContext(unitName="carDatabase")
private javax.persistence.EntityManager em;

@Transactional(REQUIRED)
public User updateCustomer(long id, String firstName){
    // JTA Transaktion fängt an
    User u = em.find(User.class, id);
    u.setFirstName("new name");
    return u;
    // JTA Transaktion endet
}
```



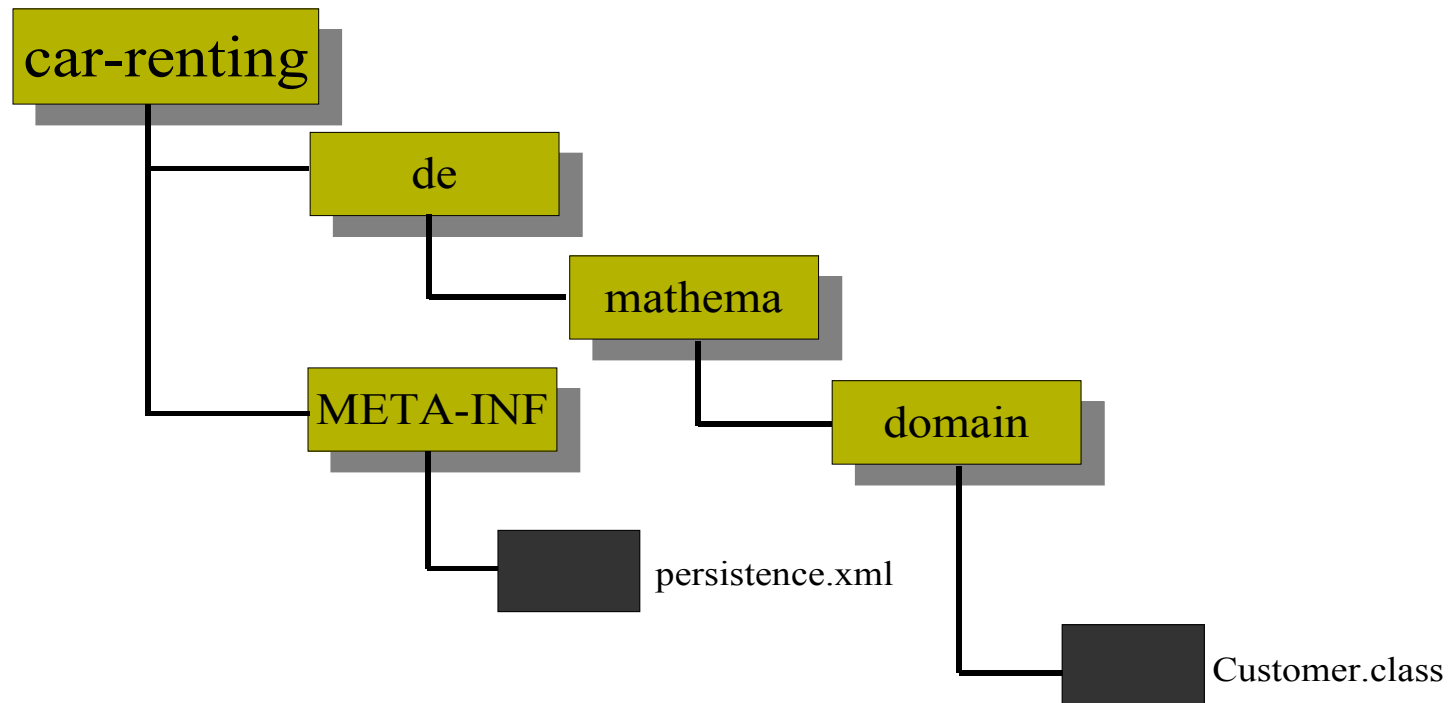
```
@PersistenceContext(unitName="carDatabase", type=EXTENDED)
private javax.persistence.EntityManager em;
...
User user = null;
getTransaction().begin(); //JTA beginnt Transaktion 1
user = em.find(User.class, 1);
getTransaction().commit(); //JTA beendet Transaktion 1

getTransaction().begin(); //JTA beginnt Transaktion 2
user.setFirstName("Serge");
em.flush();
getTransaction().commit(); // User Instanz bleibt managed
                           // und Änderung wird synchronisiert
```

- ▶ Persistence Contexts können auch länger als eine Transaktion leben
- ▶ EntityManager behält den gleichen Persistence Context für seinen ganzen Lebenszyklus
- ▶ Extended Persistence Context kann nur in eine **Stateful Session Bean** injiziert werden


▼ Persistence-Unit

- Menge von Klassen, die auf eine bestimmte Datenbank abgebildet werden
- wird in einer Datei persistence.xml definiert
 - Anforderung für die Java Persistence Spezifikation
 - **persistence.xml** definiert eine oder mehrere Persistence-Units
- Diese Datei befindet sich im **META-INF/** Verzeichnis von:
 - einer **einfachen JAR Datei** im CLASSPATH eines regulären Java SE Programms
 - einer **EJB-JAR Datei**: Persistence-Unit kann mit einem EJB Deployment eingefügt werden
 - einer **JAR Datei** in dem **WEB-INF/lib** Verzeichnis in einer Web Archivdatei (.war)
 - einer **JAR Datei** in der Wurzel einer Enterprise Archivdatei (.ear)
 - einer **JAR Datei** in dem EAR **lib/** Verzeichnis



```
public class Persistence {  
    public static EntityManagerFactory createEntityManagerFactory(  
        String unitName);  
    public static EntityManagerFactory createEntityManagerFactory(  
        String persistenceUnitName, java.util.Map properties);  
}
```

```
EntityManagerFactory factory =  
    createEntityManagerFactory("carDatabase");  
...  
EntityManager em = factory.createEntityManager();  
...  
factory.close();
```




```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="carDatabase">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto"
value="create-drop"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```


- ▼ Die `createEntityManager()` gibt eine `EntityManager` Instanz zurück
- ▼ Der **Map** Parameter überschreibt oder erweitert die Eigenschaften einer ***persistence.xml*** Datei
- ▼ `EntityManagerFactory.close()` schließt die Factory
- ▼ `EntityManagerFactory.isOpen()` überprüft die Validität der Factory

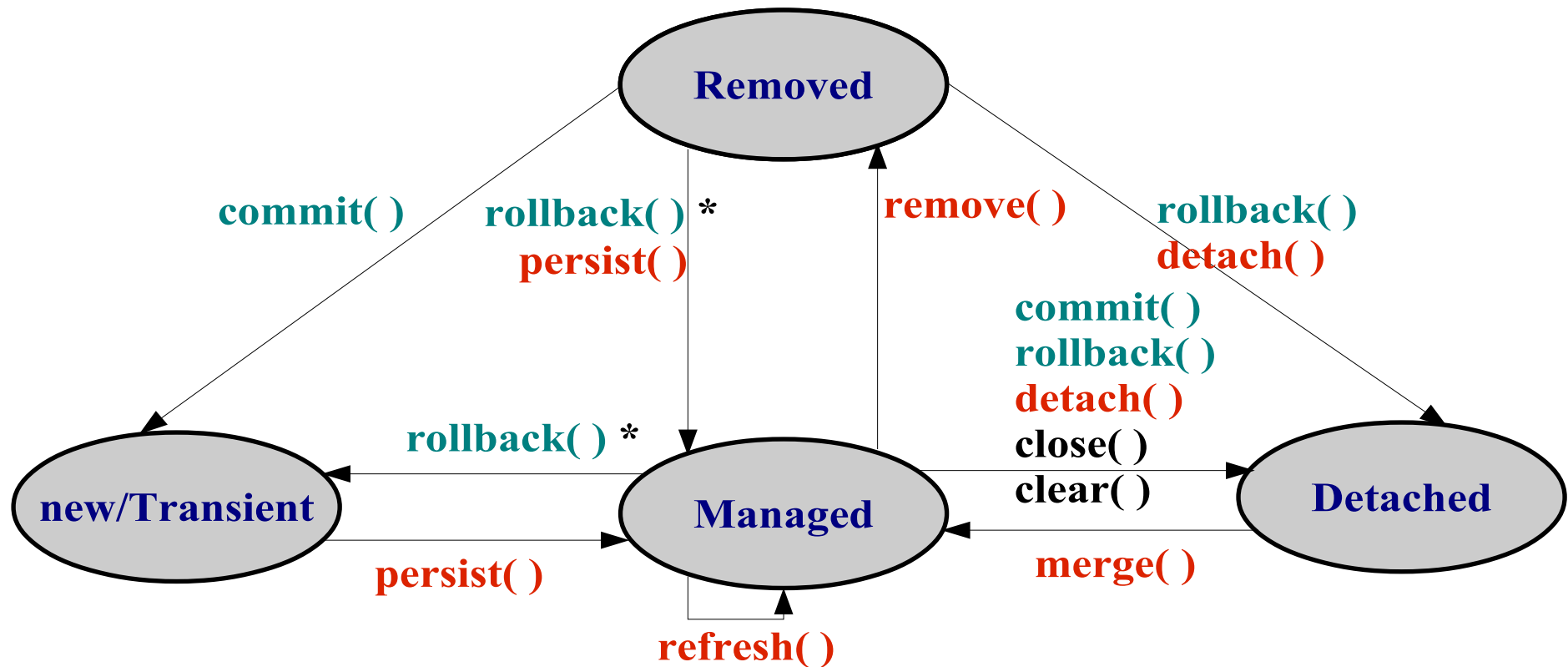
```
package javax.persistence;  
  
public interface EntityManagerFactory {  
    EntityManager createEntityManager();  
    EntityManager createEntityManager(Map map);  
    void close();  
    public boolean isOpen();  
}
```

```
package javax.persistence;

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceProperty[] properties() default {};
    PersistenceContextType type() default /* type can be EXTENDED */
        PersistenceContextType.TRANSACTION; /* Default value */
}
```

```
@javax.ejb.Stateless
public class RentingAgentBean implements RentingAgentRemote {
    @PersistenceContext(unitName="carDatabase") 
    private javax.persistence.EntityManager entityManager;
}

@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote {
    @PersistenceContext(unitName="carDatabase", type=EXTENDED) 
    private javax.persistence.EntityManager extendedEntityManager;
}
```



* = Extended Persistence Context


```
persist(Object)
merge(T) : T
remove(Object)
refresh(Object)
refresh(Object, LockModeType)
lock(Object, LockModeType)
find(Class<T>, Object) : T
find(Class<T>, Object, LockModeType) : <T>
getReference(Class<T>, Object) : T
detach(Object)
getDelegate() : Object
contains(Object) : boolean
getCriteriaBuilder() : CriteriaBuilder
void flush()
void clear()
setFlushMode(FlushModeType)
getFlushMode() : FlushModeType
createQuery(String) : Query
createNamedQuery(String) : Query
createNativeQuery(String) : Query
createNativeQuery(String, Class) : Query
joinTransaction()
getTransaktion: EntityTransaction
close()
isOpen()
```

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

```
Customer customer = new Customer();  
customer.setFirstName("Francis");  
entityManager.persist(customer);
```

- ▼ **persist()** markiert eine neue Instanz als Managed Instanz
 - Nach **flush** oder **commit** wird die neue Instanz in die Datenbank geschrieben
 - **IllegalArgumentException** wird geworfen, falls die neue Instanz als **Detached** markiert ist
 - **TransactionRequiredException** wird geworfen, wenn diese Methode auf einen Transaction Scope Persistence Context aufgerufen wird

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

```
@javax.ejb.TransactionAttribute(TransactionAttributeType.REQUIRED)
public void removeUser(long id) {
    User user = entityManager.find(User.class, id);
    entityManager.remove(user); }
```

- ▼ **remove ()** markiert Managed Instanz als Removed Instanz
 - Nach flush oder commit wird die Managed Instanz aus der Datenbank gelöscht
 - **IllegalArgumentException** wird geworfen, falls diese Instanz als **Detached** markiert ist
 - **TransactionRequiredException** wird geworfen, wenn diese Methode auf einen Transaction Scope Persistence Context aufgerufen wird

```
@javax.ejb.TransactionAttribute(TransactionAttributeType.REQUIRED)  
public void refreshUser(long id) {  
    User user = entityManager.find(User.class, id);  
    entityManager.refresh(user);  
  
    // alternativ auch mit implizitem Lock moeglich  
    entityManager.refresh(user, LockModeType.READ);  
}
```

▼ **refresh()** aktualisiert eine Managed Instanz mit dem neuen Zustand

- **IllegalArgumentException** wird geworfen, falls die Instanz eine Detached Instanz ist
- **TransactionRequiredException** wird geworfen, falls die Methode auf einen Transaction-Scoped PC aufgerufen wird
- **EntityNotFoundException** wird geworfen, falls diese Instanz aus der Datenbank gelöscht wurde

```
public <T> T merge(T entity) ;
```

```
@javax.ejb.TransactionAttribute(TransactionAttributeType  
    .REQUIRED)  
public User updateUser(User detachedUser) {  
    User copyUser = entityManager.merge(detachedUser) ;  
    return copyUser ;  
}
```

- merge () gibt eine Kopie einer Managed Instanz von einer gegebenen Detached Instanz zurück
 - IllegalArgumentException wird geworfen, falls eine Instanz als **Removed** Instanz markiert ist
 - TransactionRequiredException wird geworfen, falls die Methode auf einen Transaction Scoped Persistence Context aufgerufen wird

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

- ▼ **lock ()** sperrt eine gegebene Instanz durch einen bestimmten LockMode
- ▼ **javax.persistence.LockModeType**
 - **READ**
 - andere Transaktionen können das Objekt nur gleichzeitig lesen
 - **WRITE**
 - andere Transaktionen können nicht das Objekt gleichzeitig lesen oder schreiben

```
public CriteriaBuilder getCriteriaBuilder();
```

- ▼ Erzeugt ein Objekt zur programmatischen Erstellung von Abfragen
- ▼ Mehr dazu später (Criteria-API)

```
public void detachCustomer (int customerId){  
    Customer c = (Customer) em.find(Customer.class,  
                                     customerId) ;  
    if(c != null)  
        em.detach(c) ;  
}
```

- ▼ Entkoppelt eine Entity von ihrem Persistence-Context
- ▼ Änderungen an der Entity sind nicht mehr persistent
- ▼ Wiederaufnahme in den Persistence-Context über `merge`


```
public Query createQuery(String ejbqlString);  
  
public Query createNamedQuery(String name);  
  
public Query createNativeQuery(String sqlString);  
public Query createNativeQuery(String sqlString, Class resultClass);  
public Query createNativeQuery(String sqlString, String resultSetMapping);
```

```
Query query = entityManager.  
    createQuery("c from Customer c where id = 1");  
Customer cust = (Customer) query.getSingleResult();
```

```
public <T> T find(Class<T> entityClass, Object primaryKey);  
  
public <T> T find(Class<T> entityClass, Object primaryKey,  
                  LockModeType lockMode);
```

```
Customer customer = entityManager.find(Customer.class, 1);
```

- ▼ **find()** gibt **null** zurück, falls die Instanz nicht in der Datenbank existiert
 - die Parameter von **find()** sind:
 - Klasse der Persistence-Entity
 - Primärschlüssel der Persistence-Entity
 - die Methode **find()** benutzt Java Generics, um das Casting zu vermeiden
 - **IllegalArgumentException** für falsche Parameter

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

```
Customer customer = null;
try{
    customer = entityManager.getReference(Customer.class, 1);
}catch (EntityNotFoundException notFound) {
    //logische Besserung
}
```

▼ **getReference ()** liefert eine Referenz auf die (noch nicht geladene) Instanz des Objekts, wirft aber

- **EntityNotFoundException** für nicht existierende Entities
- **IllegalArgumentException** für falsche Parameter.
- die Parameter sind
 - Klasse der Persistence-Entity
 - Primärschlüssel der Persistence-Entity

contains ()

- hat als Parameter die Instanz der Persistence-Entity und
- gibt **true** zurück, falls die Instanz vom Persistence Context verwaltet wird

```
Customer werner = new Customer("Eberling", "Werner");
em.persist(werner);

//The new Customer should be managed now
Assert.assertTrue("Customer should be managed now, after
    Call to persist.", em.contains(werner) );
```

```
//Setzt den Flush Modus für alle Objekte im PC
public void setFlushMode(FlushModeType flushMode);

//Gibt den Flush Modus für alle Objekte im PC
public FlushModeType getFlushMode();

//Löscht den Persistence Context und alle Managed
//Objekte werden detached Objekte
public void clear();
```

- Der FlushMode kontrolliert, ob die transaktionellen Änderungen vor der Ausführung von Abfragen mit der Datenbank synchronisiert wird
- javax.persistence.FlushModeType** hat zwei Konstanten
 - COMMIT** Synchronisation nur am Ende der Transaktion
 - AUTO** Synchronisation kann auch vor der Ausführung von Abfragen geschehen

```
public EntityTransaction getTransaction();
```

```
public interface EntityTransaction {  
    public void begin();  
    public void commit();  
    public void rollback();  
    public boolean isActive();  
}
```

- **EntityManager.getTransaction()** wird in einer Nicht-Java EE Umgebung angewandt
 - **begin()** wirft **IllegalStateException**, falls **EntityTransaction** schon aktiv ist
 - **commit()** und **rollback()** wirft **IllegalStateException**, falls eine Transaktion noch nicht aktiv ist

```
public void close() ;  
  
public boolean isOpen() ;
```

- ▼ Freigabe von allen Ressourcen einer EntityManager Instanz durch die Methode `close()`
 - Der Persistence Context wird beendet
 - Alle Managed Entities der EntityManager Instanz werden Detached Entities
 - Alle vorhandenen **Query** Instanzen werden ungültig
- ▼ `isOpen()` auf einer geschlossenen EntityManager Instanz wirft eine `IllegalStateException`

Einfache Mappings

▼ Persistence Entities

- sind einfache Java Klassen
- werden wie ein normales Java Objekt alloziert und durch den Dienst EntityManager automatisch verwaltet
- benötigen einen Default-Konstruktor
- müssen der JavaBeans Konvention entsprechen und serialisierbar sein

▼ Java Persistence API erfordert nur zwei Annotationen von Meta-Informationen

- `@javax.persistence.Entity` bildet eine Klasse auf eine Datenbank ab
 - `@Entity` hat ein `name ()` Attribut, um die Entity in einer EJB QL zu referenzieren
- `@javax.persistence.Id` markiert eine Eigenschaft als Primärschlüssel

- ▼ alle andere Eigenschaften der Klasse werden auf Spalten der Datenbank mit dem gleichen Namen abgebildet
- ▼ XML Mapping kann anstatt Annotations benutzt werden
 - **orm.xml** in **META-INF/**
 - Deklarieren der Mapping Datei in ***persistence.xml*** (<mapping-file/>)

```
package de.mathema.domain;

@javax.persistence.Entity
public class Customer implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private Long id;
    private String lastName;
    private String firstName;

    @javax.persistence.Id
    public Long getId() {return id;}
    public void setId(Long id) {this.id = id;}

    public String getFirstName() {return this.firstName;}
    public void setFirstName(String firstName){
        this.firstName = firstName;}

    public String getLastName() {return this.lastName;}
    public void setLastName(String lastName) {
        this.lastName = lastName;}

    public String toString() {
        return "Customer#" + getId() + "(" + getLastName() + ")";
    }
}
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd" version="1.0">
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

▶ <entity-mappings> das Wurzelelement

- <entity> definiert die Entity Klasse und den Typ des Zugriffs: **PROPERTY** oder **FIELD**
- <attributes> hat <id> als Unterelement
 - <id> definiert ein Attribut als Primärschlüssel

▶ Der OR-Mapper nimmt alle anderen Eigenschaften der Klasse als persistente Eigenschaften an

▼ @javax.persistence.Table

- sagt dem EntityManager Dienst den Namen der Tabelle, auf den die Entity abgebildet wird
- diese Annotation wird oft in einer Bottom-Up Strategie angewandt
- **name()** Attribut legt den Namen der Tabelle einer Datenbank fest
- **schema()** und **catalog()** identifizieren den Katalog und das Schema in einer relationalen Datenbank

▼ @javax.persistence.Column

- beschreibt, wie eine bestimmte Eigenschaft auf eine Spalte in der Datenbank abgebildet wird
- **name()** Attribut legt den Namen der Spalte fest
- **length()** Attribut bestimmt die Länge einer Eigenschaft

```
package de.mathema.domain;

@javax.persistence.Entity
@javax.persistence.Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
    @javax.persistence.Id
    @javax.persistence.Column(name="CUSTID",nullable=false,
        columnDefinition="integer")
    private long id;
    @javax.persistence.Column(name="LAST_NAME",length=50,nullable=false)
    private String lastName;
    @javax.persistence.Column(name="FIRST_NAME",length=50,nullable=false)
    private String firstName;

    public long getId() {return id;}
    public 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;}
}
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd" version="1.0">
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table name="CUSTOMER_TABLE"/>
    <attributes>
      <id name="id">
        <column name="CUSTID"
          nullable="false"
          column-definition="integer"/>
      </id>
      <basic name="firstName">
        <column name="FIRST_NAME"
          nullable="false"
          lenght="50"/>
      </basic>
      ...
    </attributes>
  </entity>
</entity-mappings>
```

```
package javax.persistence;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```

```
package javax.persistence;

@Target({}) @Retention(RUNTIME)
public @interface UniqueConstraint {
    String[] columnNames();
}
```


- ▼ @Column hat ein äquivalentes XML-Element **<column>**
- ▼ Insertable() & updateable()
 - spezifizieren, ob eine Spalte in **SQL INSERT** oder **UPDATE** beigefügt werden soll
- ▼ <column> Element ist ein Unterelement von Attributen:
 - <id>
 - <basic>
 - <temporal>
 - <lob>
 - <enumerated>

```
package javax.persistence;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {

    String name() default "";
    boolean unique()

        default false;
    boolean nullable()

        default true;
    boolean insertable()

        default true;
    boolean updatable()

        default true;
    String columnDefinition()

        default "";

    String table() default "";
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```

```
package javax.persistence;  
@Target({METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface Id {}
```

```
package javax.persistence;  
@Target({METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface GeneratedValue {  
    GenerationType strategy() default  
        GenerationType.AUTO;  
    String generator() default "";  
}  
  
package javax.persistence;  
public enum GenerationType {  
    TABLE, SEQUENCE, IDENTITY, AUTO  
};
```

- ▶ **@javax.persistence.Id** identifiziert einen Primärschlüssel für eine Tabelle
- ▶ **@javax.persistence.GeneratedValue** für die automatische Generierung von Primärschlüsseln
 - Generierung erfordert einen bestimmten **GenerationType**
 - **AUTO** (Default Wert)
 - **IDENTITY** (**AUTO** kann durch **IDENTITY** ersetzt werden)
 - andere Typen mit zusätzlichen Meta-Informationen

```
package de.mathema.domain;

@javax.persistence.Entity
@javax.persistence.Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
    private long id;
    ...
    @javax.persistence.Id
    @javax.persistence.GeneratedValue
    @javax.persistence.Column(name="CUSTID", nullable=false,
                              columnDefinition="integer")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table name="CUSTOMER_TABLE"/>
    <attributes>
      <id name="id">
        <generated-value strategy="AUTO"/>
        ...
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

```
package javax.persistence;
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
}
```

- ▶ ***name()***: benutzt Name von **@TableGenerator** und von **@Id**
- ▶ ***table()***: beschreibt die Definition der Tabelle
- ▶ ***pkColumnName()***: identifiziert den generierten Schlüssel
- ▶ ***valueColumnName()***: Name des Zählers für den generierten Schlüssel
- ▶ ***pkColumnValue()***: der Primärschlüssel
- ▶ ***allocationSize()***: Größe der Inkrementierung

```
# Die Strategie TABLE bestimmt eine relationale Tabelle für die
# Generierung von numerischen Schlüsseln
CREATE TABLE CUST_GENERATOR_TABLE
(
  PRIMARY_KEY_COLUMN VARCHAR not null,
  VALUE_COLUMN long not null
);
```

```
@javax.persistence.Entity
public class Customer implements java.io.Serializable {
    private long id;
    private String LastName;
    private String FirstName;
    @TableGenerator(name="CUST_GENERATOR", table="CUST_GENERATOR_TABLE",
        pkColumnName="PRIMARY_KEY_COLUMN",
        valueColumnName="VALUE_COLUMN",
        pkColumnValue="CUSTID",
        allocationSize=20)
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE,
        generator="CUST_GENERATOR")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table-generator name="CUST_GENERATOR" table="CUST_GENERATOR_TABLE"
      pk-column-value="PRIMARY_KEY_COLUMN"
      value-column-name="VALUE_COLUMN"
      pk-column-value="CUSTID"
      allocation-size="20"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE" generator="CUST_GENERATOR"/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

■ **<table-generator>** ist ein Unterelement von **<entity>**

- ihre Attribute sind genau so wie die der `@javax.persistence.TableGenerator` Annotation
- nicht vergessen den Generator **<generated-value>** zu setzen!

```
package javax.persistence;  
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)  
public @interface SequenceGenerator {  
    String name();  
    String sequenceName() default "";  
    int initialValue() default 1;  
    int allocationSize() default 50;  
}
```

- ▼ **name()**: Name, der von dem **generator** Attribut des **@javax.persistence.GeneratedValue** benutzt wird
- ▼ **sequenceName()**: definiert die Sequence Tabelle, die aus der Datenbank benutzt wird

```
package de.mathema.domain;
import javax.persistence.*;

@Entity
public class Customer implements java.io.Serializable {
    private long id;
    private String LastName;
    private String FirstName;
    @SequenceGenerator(name="CUSTOMER_SEQUENCE",
        sequenceName="CUST_SEQ")

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="CUSTOMER_SEQUENCE")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}

    @Column(name="FIRST_NAME",length=50,nullable=false)
    public String getFirstName() {return FirstName;}
    public void setFirstName(String firstName) {FirstName = firstName;}

    @Column(name="LAST_NAME",length=50,nullable=false)
    public String getLastName() {return LastName;}
    public void setLastName(String lastName) {LastName = lastName;}
}
```



```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <sequence-generator name="CUSTOMER_SEQUENCE"
      sequence-name="CUST_SEQ"
      initial-value="0"/>
    <attributes>
      <id name="id">
        <generated-value strategy="SEQUENCE" generator="CUSTOMER_SEQUENCE"/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

■ **<sequence-generator>** ist ein Unterelement von **<entity>**

- ihre Attribute sind genau so wie die der
@javax.persistence.SequenceGenerator Annotation
- nicht vergessen den Generator im Tag **<generated-value>** zu setzen

- ▼ EJB 2.1 Primärschlüssel Stil
- ▼ Java Persistence API stellt Annotations zur Verfügung:
 - `@javax.persistence.IdClass`
 - `@javax.persistence.EmbeddedId`
 - `@javax.persistence.Embeddable`
- ▼ automatische Generierung wird nicht von Composite Keys und Primary Key Classes unterstützt
- ▼ Klasse als Primärschlüssel muss diese Anforderungen erfüllen:
 - Klasse muss `java.io.Serializable` implementieren
 - Klasse muss einen No-Arg Konstruktor haben
 - Klasse muss die **`equals()`** und **`hashCode()`** Methode implementieren

```
package de.mathema.domain;

import javax.persistence.*;

@Entity
@IdClass(CustomerPK.class)
public class Customer implements
    java.io.Serializable {
    private String firstName;
    private String lastName;
    private long ssn;
    public String getFirstName()
    {return firstName;}
    public void setFirstName
    (String firstName)
    {this.firstName = firstName;}

    @Id
    public String getLastName()
    {return lastName;}
    public void setLastName
    (String lastName)
    {this.lastName = lastName;}

    @Id
    public long getSsn(){return ssn;}
    public void setSsn(long ssn)
    {this.ssn = ssn;}
}
```

```
package de.mathema.domain;
public class CustomerPK
    implements java.io.Serializable {
    private String lastName;
    private long ssn;

    ...

    public boolean equals(Object obj){
        if (obj==this) return true;
        if (!(obj instanceof CustomerPK))
            return false;
        CustomerPK pk = (CustomerPK)obj;
        if (!lastName.equals(pk.lastName))
            return false;
        if (ssn != pk.ssn) return false;
        return true;
    }

    public int hashCode( ){
        return lastName.hashCode()
            +(int)ssn;
    }
}
```

```
<entity-mappings>
  <entity class="com.titan.domain.Customer"
    access="PROPERTY">
    <id-class>de.mathema.domain.CustomerPK</id-class>
    <attributes>
      <id name="lastName" />
      <id name="ssn" />
    </attributes>
  </entity>
</entity-mappings>
```

```
package de.mathema.domain;

import javax.persistence.*;

@Entity
public class Customer
    implements
        java.io.Serializable {
    private String firstName;
    private CustomerPK pk;

    public String getFirstName()
    {return firstName;}
    public void setFirstName
        (String firstName)
    {this.firstName=firstName;}

    @EmbeddedId
    public CustomerPK getPk()
    {return pk;}
    public void setPk(CustomerPK pk)
    {this.pk=pk;}
}
```

```
package de.mathema.domain;
import javax.persistence.Column;

@Embeddable
public class CustomerPK
    implements java.io.Serializable {
    private String lastName;
    private long ssn;

    @Column(name="CUSTOMER_LAST_NAME")
    public String getLastName()
    {return this.lastName;}
    public void setLastName(String lastName)
    {this.lastName=lastName;}

    ...
    public boolean equals(Object obj){
        if(obj == this) return true;
        if(!(obj instanceof CustomerPK))
            return false;
        CustomerPK pk = (CustomerPK)obj;
        if(!lastName.equals(pk.lastName))
            return false;
        if(ssn != pk.ssn) return false;
        return true;
    }
    public int hashCode(){
        return lastName.hashCode()
            +(int)ssn;
    }
}
```

```
<entity-mappings>
<embeddable class="de.mathema.domain.CustomerPK"
  access-type="PROPERTY">
  <embeddable-attributes>
    <basic name="lastName">
      <column name="CUSTOMER_LAST_NAME"/>
    </basic>
    <basic name="ssn">
      <column name="CUSTOMER_SSN"/>
    </basic>
  </embeddable-attributes>
</embeddable>
<entity class="de.mathema.domain.Customer" access="PROPERTY">
  <attributes>
    <embedded-id name="pk">
      <attribute-override name="lastName">
        <column name="LAST_NAME"/>
      </attribute-override>
      <attribute-override name="ssn">
        <column name="SSN"/>
      </attribute-override>
    </embedded-id>
  </attributes>
</entity>
</entity-mappings>
```

```
package de.mathema.domain;
import javax.persistence.*;
@Entity
public class Customer
    implements java.io.Serializable {
    private long id;
    private Address address;
    ...
    @Embedded
    public Address getAddress() {
        return address;
    }
}
```

```
package de.mathema.domain;

import javax.persistence.Embeddable;

@Embeddable
public class Address
    extends Serializable{
    private String street;
    private String city;
    private String state;

    ...
}
```

- ▶ Java Persistence API ermöglicht, in Persistence Entities eingebettete Java Objekte zu benutzen
- ▶ Eingebettete Java Objekte (dependent objects)
 - haben keine Identität
 - werden mit `@javax.persistence.Embedded` markiert

```
<entity-mappings>
<embeddable class="de.mathema.domain.Address"
  access-type="PROPERTY"/>
<entity class="de.mathema.domain.Customer"
  access="PROPERTY">
  <attributes>
    <id name="id"/>
    <embedded name="address">
      <attribute-override name="street">
        <column name="CUST_STREET"/>
      </attribute-override>
      <attribute-override name="city">
        <column name="CUST_CITY"/>
      </attribute-override>
      <attribute-override name="state">
        <column name="CUST_STATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
</entity-mappings>
```



```
@Entity
public class Customer
    implements java.io.Serializable {
    private long id;
    @Embedded Address address;
    ...
}
```

```
@Embeddable
public class Address {

    @Embedded Street street
}
```

```
@Embeddable
public class Street {

    String streetName;
    String hsNr;
}
```

▼ Embeddables können auch in Collections verwaltet werden

- Speicherung erfolgt in eigener Tabelle

```
@Entity
public class Customer {

    @ElementCollection
    Set<Phone> phoneNumber = new HashSet<Phone>();
    ...
}
```

```
@Embeddable
public class Phone {

    String countryCode;
    String regionalCode;
    String callNumber;

}
```

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface CollectionTable {
    String name() default "";
    String catalog() default "";
    JoinColumn[] joinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

▼ **@javax.persistence.Transient**

- der Persistence Manager ignoriert alle Eigenschaften, die mit **@javax.persistence.Transient** markiert sind

▼ **@javax.persistence.Basic** und FetchType (LAZY, EAGER)

- **@javax.persistence.Basic**
 - markiert alle Java primitive oder Java Wrapper Typen
 - hat Attribute **fetch()** und **optional()**

- ▼ **@Temporal** und **TemporalType**(DATE, TIME, TIMESTAMP)
 - verwendbar für `java.util.Date` oder `java.util.Calendar`
- ▼ **@javax.persistence.Lob** ermöglicht die Markierung von primitiven Typen als `java.sql.Blob` oder `java.sql.Clob`
 - **Blob** für Java Typen **`byte[]`**, **`Byte[]`** oder **`java.io.Serializable`**
 - **Clob** für Java Typen **`char[]`**, **`Character[]`** oder **`java.lang.String`**

▼ **@javax.persistence.Enumerated**

- wird verwendet, um Java enum Typen zu markieren

▼ **@javax.persistence.Version**

- wird für die Optimistic Concurrency benutzt

▼ **@javax.persistence.SecondaryTable**

- Java Persistence API ermöglicht durch @SecondaryTable die Abbildung einer Persistence Entity auf eine oder mehrere Tabelle(n)

▼ **@javax.persistence.Access**

- definiert die Zugriffsvariante für eine Attribut der Entität

▼ Constraint-Angaben bisher nur für DDL-Skripte

- `@Column(nullable=false, length=256)`
- `@Basic(optional=false)`
- ...
- Keine Berücksichtigung durch den Persistence-Provider
- Attributwerte müssen nicht mit diesen Angaben übereinstimmen!

- ▼ Eigener Standard innerhalb der JEE
- ▼ Framework zur Validierung von Bean-Attributen
 - Vordefinierte Menge von Standard-Constraints
 - Möglichkeit zur Implementierung eigener Constraints
 - In den Persistence-Provider integriert
- ▼ Prüfung der Constraints
 - Vor der Persistierung
 - Vor einem Update
 - Vor der Löschung (optional)

▼ Vordefinierte Constraints in `javax.validation.constraints`

```
@Entity
public class Kunde {
    @NotNull
    @Size(min=2,max=25)
    String name;

    @NotNull
    @Size(min=2,max=25)
    String vorname;

    @Embedded
    @Valid
    Adresse adresse;
}
```

Implementierung eigener Constraints möglich

```
@Constraint(validatedBy = KundenNummerValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface KundenNummer {
    String message() default "{de.mathema.schulung.KundenNummer.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class KundenNummerValidator implements
    ConstraintValidator<KundenNummer,String> {

    //...
}
```

Beziehungen

▼ die unidirektionale **One-to-one** Beziehung

- ein Kunde hat eine unidirektionale Beziehung mit der Adresse

▼ die bidirektionale **One-to-one** Beziehung

- ein Kunde hat eine bidirektionale Beziehung mit der Kreditkarte

▼ die unidirektionale **One-to-many** Beziehung

- ein Kunde hat eine unidirektionale Beziehung mit dem Telefon

▼ die bidirektionale **One-to-many / Many-to-One** Beziehung

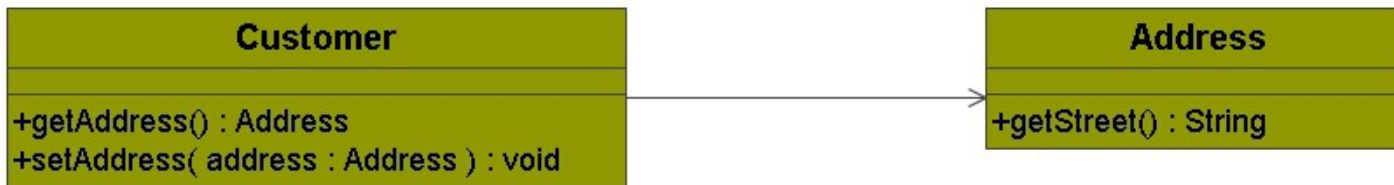
- ein Anbieter hat eine bidirektionale Beziehung mit dem Auto

▼ die bidirektionale **Many-to-many** Beziehung

- eine Reservierung kann viele Fahrer haben und jeder Fahrer kann für viele Reservierungen eingetragen sein

▼ @OneToOne beschreibt eine Beziehung, die

- mit @JoinColumn auf einen Fremdschlüssel,
- @JoinColumns auf einen zusammengesetzter Primärschlüssel oder
- @PrimaryKeyJoinColumn auf Primärschlüssel von beiden Persistence-Entities abgebildet wird



```
@javax.persistence.Entity
public class Customer implements java.io.Serializable{
    ...
    private Address address;
    ...
    @javax.persistence.OneToOne(cascade={CascadeType.ALL})
    @javax.persistence.JoinColumn(name="ADDRESSID")
    public Address getAddress() {return this.address;}
    public void setAddress(Address address){this.address = address;}
}
```

```
package javax.persistence;
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

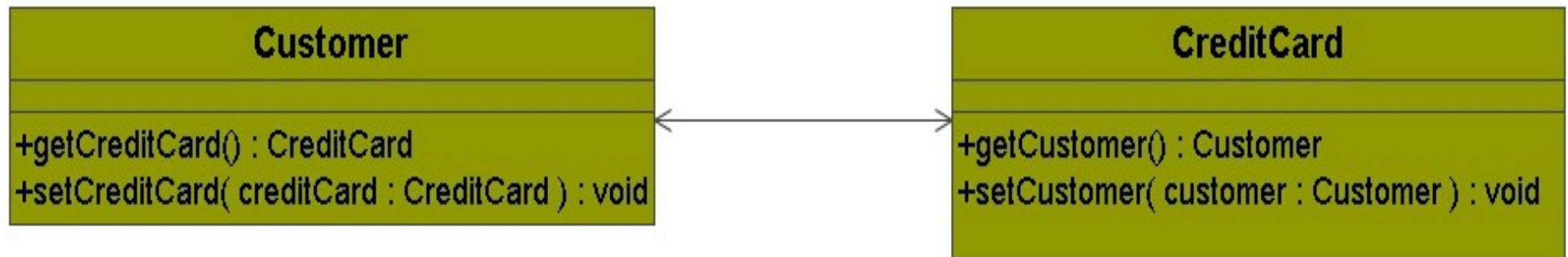
Attribut	Typ	Bedeutung	Default
targetEntity	Class	Entitätsklasse, die Ziel der Assoziation ist; nur erforderlich, wenn dies nicht aus dem Attributstyp hervorgeht (optional)	Typ des Attributs
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	EAGER
optional	Boolean	gibt an, ob die Assoziation belegt sein muss oder nicht (optional)	true
mappedBy	String	darf an nur einer Seite der Beziehung – der nicht-führenden – angegeben werden	

```
package javax.persistence;
public @interface JoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="address"
        targetEntity="de.mathema.domain.Address"
        fetch="LAZY"
        optional="true">
        <cascade>ALL</cascade>
        <join-column name="ADDRESSID"/>
      </one-to-one>
    </attributes>
  </entity></entity-mappings>
```


■ Für eine bidirektionale Beziehung wird

- `@javax.persistence.OneToOne` in dem referenzierten Objekt mit dem Attribut ***mappedBy()*** angewandt



```
package de.mathema.domain;
@javax.persistence.Entity
public class Customer implements
    java.io.Serializable{
    private CreditCard creditCard;

    ...

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="CREDITCARDID")
    public CreditCard getCreditCard()
    {return creditCard;}

    // Sicherstellung der Multiplizität
    // und Navigierbarkeit
    public void setCreditCard
        (CreditCard creditCard)
    {this.creditCard = creditCard;}
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class CreditCard implements
    java.io.Serializable{
    private Customer customer;

    ...

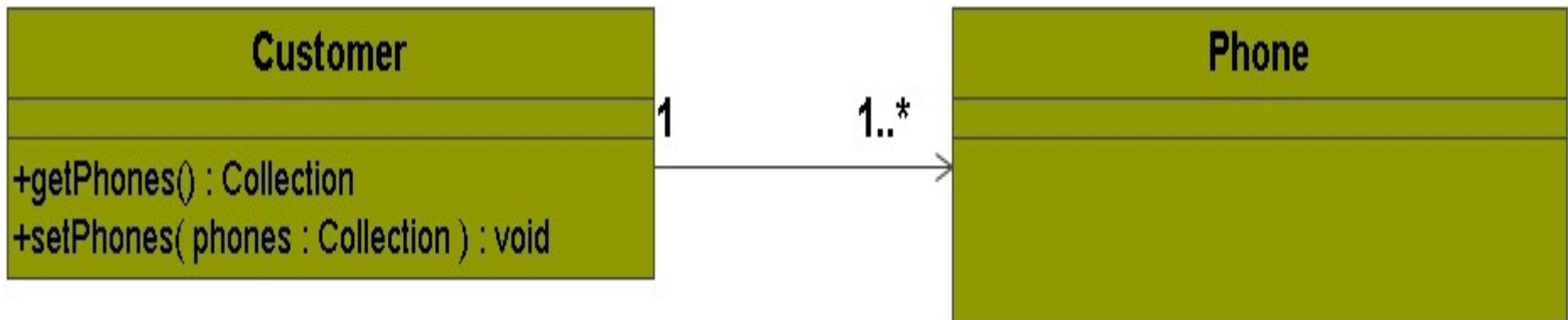
    @OneToOne(mappedBy="creditCard")
    public Customer getCustomer()
    {return customer;}

    // Sicherstellung der Multiplizität
    // und Navigierbarkeit
    public void setCustomer
        (Customer customer)
    {this.customer = customer;}

    }
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="creditCard"
        target-entity="de.mathema.domain.CreditCard"
        fetch="LAZY">
        <cascade-all/>
        <join-column name="CREDITCARDID"/>
      </one-to-one>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.CreditCard"
    access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="customer"
        target-entity="de.mathema.domain.Customer"
        mapped-by="creditCard"/>
    </attributes>
  </entity>
</entity-mappings>
```

- **@javax.persistence.OneToOne** wird für die Deklaration einer One-to-many Beziehung angewandt und benutzt
 - die Annotation **@javax.persistence.JoinColumn** oder
 - die Annotation **@javax.persistence.JoinTable** für die Abbildung auf Fremdschlüssel



```
package javax.persistence;
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

```
package javax.persistence;
public @interface JoinTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Primärschlüssel eigener Seite

Primärschlüssel anderer Seite

Attribut	Typ	Bedeutung	Default
targetEntity	Class	wie bei @OneToOne, hier jedoch etwas relevanter für den Fall der Anwendung untypisierter Kollektionen (optional)	Inhaltstyp der Collection, bei Anwendung von Generics
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	LAZY
mappedBy	String	Pflichtattribut im Gegensatz zu @OneToOne	

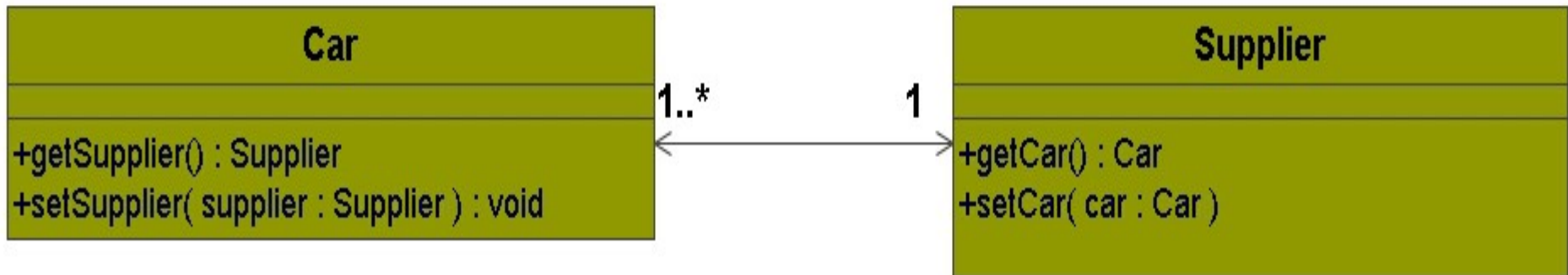
```
package de.mathema.domain;
@javax.persistence.Entity
public class Phone implements java.io.Serializable {
    private long id;
    ...
    @javax.persistence.Column(name="NUMBER")
    public String getNumber() {return number;}
    public void setNumber(String nummer) {this.number = nummer;}
    ...
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class Customer implements Serializable {
    private Address address;
    private Collection<Phone> phones;
    ...
    @javax.persistence.OneToMany(cascade={CascadeType.ALL})
    @JoinTable(name="CUSTOMER_PHONE",
        joinColumns={@JoinColumn(name="CUSTOMERID")},
        inverseJoinColumns={@JoinColumn(name="PHONEID")})
    public Collection<Phone> getPhones() {return phones;}
    public void setPhones(Collection<Phone> phones) {this.phones = phones;}
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-many name="phones"
        targetEntity="de.mathema.domain.Phone">
        <cascade-all />
        <join-table name="CUSTOMER_PHONE">
          <join-column name="CUSTOMERID" />
          <inverse-join-column name="PHONEID" />
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
</entity-mappings>
```


■ die Beschreibung dieser Beziehung geschieht mit Hilfe von Annotations:

- `@javax.persistence.ManyToOne`
 - `@javax.persistence.JoinColumn` wird für die Abbildung auf einen Fremdschlüssel verwendet
- `@javax.persistence.OneToOne`
 - das Attribut ***mappedBy()*** wird nur für bidirektionale Beziehungen verwendet



```
package javax.persistence;

public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

- ▼ die Attributdefinition von `@javax.persistence.ManyToOne` ähnelt der von `@javax.persistence.OneToOne`

Attribut	Typ	Bedeutung	Default
targetEntity	Class	Entitätsklasse, die Ziel der Assoziation ist; nur erforderlich, wenn dies nicht aus dem Attributstyp hervorgeht (optional)	Typ des Attributs
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	EAGER
optional	Boolean	gibt an, ob die Assoziation belegt sein muss oder nicht (optional)	true
mappedBy	String	darf an nur einer Seite der Beziehung – der nicht-führenden – angegeben werden	

```
package de.mathema.domain;
@javax.persistence.Entity
public class Car implements Serializable {
    ...
    private Supplier supplier;
    ....
    @javax.persistence.ManyToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="SUPPLIERID")
    public Supplier getSupplier() {return supplier;}
    public void setSupplier(Supplier supplier) {this.supplier = supplier;}
}
```

```
package de.mathema.domain;

@javax.persistence.Entity
public class Supplier implements Serializable{
    private Set<Car> cars = new HashSet<Car>();
    ...
    @javax.persistence.OneToMany(mappedBy="supplier")
    public Set<Car> getCars() {return cars;}
    public void setCars(Set<Car> cars) {this.cars = cars;}
}
```

Die bidirektionale One-to-many / Many-to-One Beziehung: XML Beispiel

221

```
<entity-mappings>
  <entity class="de.mathema.domain.Supplier" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-many name="car"
        target-entity="de.mathema.domain.Car" fetch="LAZY"
        mapped-by="supplier">
      </one-to-many>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Car" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-one name="supplier"
        target-entity="de.mathema.domain.Supplier"
        fetch="EAGER">
        <join-column name="SUPPLIERID" />
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

- ▼ Die Many-to-many Beziehung wird durch die `@javax.persistence.ManyToMany` definiert
- ▼ `@javax.persistence.JoinTable` definiert die Tabelle (Link Tabelle), die eine große Rolle in der Beziehung spielt.



```
package javax.persistence;
public @interface ManyToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class Reservation implements Serializable {
    ...
    private Set<Driver> drivers = new HashSet<Driver>();
    // Sicherstellung der Multiplizität und Navigierbarkeit
    @javax.persistence.ManyToMany
    @javax.persistence.JoinTable(name="RESERVATION_DRIVER",
        joinColumns={@JoinColumn(name="RESERVATION_ID")},
        inverseJoinColumns={@JoinColumn(name="DRIVER_ID")})
    public Set<Driver> getDrivers() {return drivers;}
    public void setDrivers(Set<Driver> drivers) {this.drivers = drivers;}
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class Driver implements Serializable{
    ...
    private Collection<Reservation> reservations = new ArrayList<Reservation>();
    // Sicherstellung der Multiplizität und Navigierbarkeit
    @javax.persistence.ManyToMany(mappedBy="drivers")
    public Collection<Reservation> getReservations() {return reservations;}
    public void setReservations(Collection<Reservation> reservations)
    {this.reservations = reservations;}
    ...
}
```



```
<entity-mappings>
  <entity class="de.mathema.domain.Reservation" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-many name="drivers"
        target-entity="de.mathema.domain.Driver" fetch="LAZY">
        <join-table name="RESERVATION_DRIVER">
          <join-column name="RESERVATION_ID" />
          <inverse-join-column name="DRIVER_ID" />
        </join-table>
      </many-to-many>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Driver" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-many name="reservations"
        target-entity="de.mathema.domain.Reservation" fetch="LAZY"
        mapped-by="drivers">
      </many-to-many>
    </attributes>
  </entity>
</entity-mappings>
```

- Das automatische Schreiben/Lesen/Löschen/Aktualisieren von Beziehungen einer persistenten Instanz in die Datenbank
- setze das Attribut `cascade()` mit einem der Typen aus `javax.persistence.CascadeType` in den Beziehungen `@OneToOne`, `@OneToMany`, `@ManyToOne` und `@ManyToMany`

```
package javax.persistence;  
public enum CascadeType {  
  
    ALL, //ist eine Kombination aus allen Richtlinien  
  
    PERSIST, //kaskadiert entityManager.persist() Operationen  
  
    MERGE, //kaskadiert entityManager.merge() Operationen  
  
    REMOVE, //kaskadiert entityManager.remove() Operationen  
  
    REFRESH //kaskadiert entityManager.refresh() Operationen  
}
```

```
package de.mathema.domain;

@javax.persistence.Entity
public class Customer implements Serializable{
    ...
    private Address address;
    ...
    @javax.persistence.OneToOne
        (cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    @javax.persistence.JoinColumn(name="ADDRESS_ID")
    public Address getAddress() {return this.address;}
    public void setAddress(Address address)
        {this.address = address;}
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-one name="address"
        targetEntity=
          "de.mathema.domain.Address" fetch="LAZY"
        optional="true">
        <cascade-persist />
        <cascade-remove />
        <primary-key-join-column />
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

▼ LAZY und EAGER Fetching

- durch ***fetch()*** in allen Objektbeziehungen

▼ LAZY Fetching

- das annotierte Attribut kann bei Bedarf nachgeladen werden

▼ EAGER Fetching

- alle Attribute werden sofort geladen

▼ LAZY Fetching in Verbindung mit FETCH JOIN Operation

- verbessert die Manipulation von Managed Objekten

Vererbung

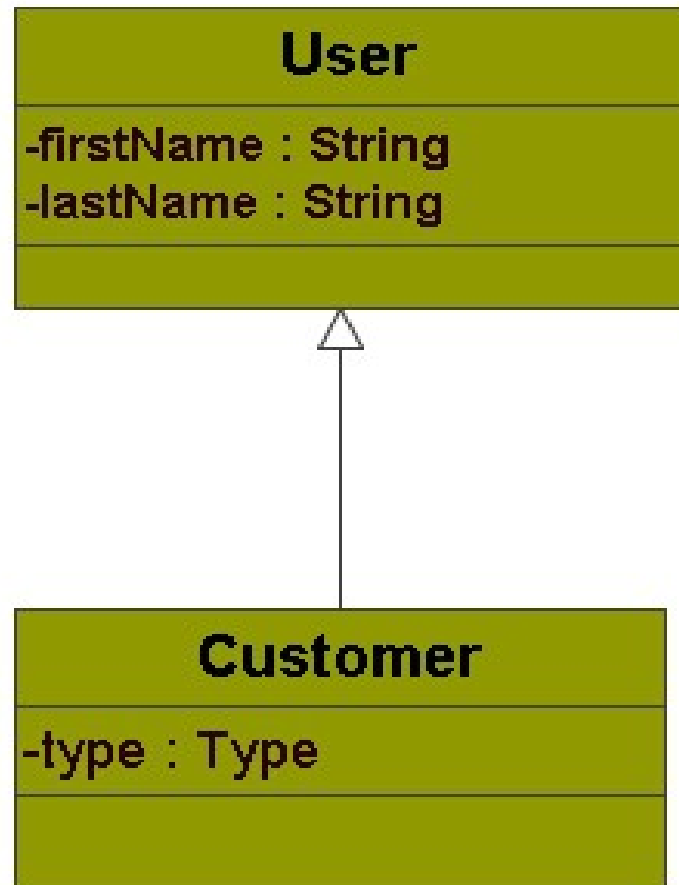
- ▼ EJB 3.1 Persistence API unterstützt drei Strategien für die Vererbungsabbildung
 - Single Table Per Class Hierarchy (default)
 - Table Per Concrete Class
 - Table Per Subclass
- ▼ Java Persistence API unterstützt zusätzlich
 - Implizite Polymorphie
 - Polymorphe Abfrage

```
package javax.persistence;
public @interface Inheritance
    InheritanceType strategy()
default SINGLE_TABLE;
}
```

```
package javax.persistence;
public enum InheritanceType
{
    SINGLE_TABLE,
    TABLE_PER_CLASS,
    JOINED
};
```

```
package javax.persistence;
public @interface DiscriminatorColumn
{
    String name() default "DTYPE";
    DiscriminatorType
discriminatorType() default STRING;
    String columnDefinition()
default "";
    int length() default 31;
}
```

```
package javax.persistence;
public @interface
    DiscriminatorValue {
    String value();
}
```

USER
id integer primary key not null, firstName varchar(255), lastName varchar(255), type varchar(255), DISCRIMINATOR varchar(31) not null

▼ Einzige Tabelle für die gesamte Klassenhierarchie

- erfordert eine zusätzliche Discriminator-Spalte

▼ Vorteile

- Performanz
- Einfachheit

▼ Nachteile

- Spalten für die Attribute einer Unterklasse können den NULL-Wert annehmen

```
package de.mathema.domain;

@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR",
    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("USER")
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
@DiscriminatorValue("CUST")
public class Customer extends User implements Serializable{

}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="SINGLE_TABLE" />
    <discriminator-column name="DISCRIMINATOR"
      discriminator-type="STRING" />
    <discriminator-value>USER</discriminator-value>
    <attributes>
      <id>
        <generated-value />
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer">
    <discriminator-value>CUST</discriminator-value>
  </entity>
</entity-mappings>
```

User
id integer primary key not null, firstName varchar(255), lastName varchar(255), ...

Customer
id integer primary key not null, firstName varchar(255), lastName varchar(255), type varchar(255), ...

▼ Diese Strategie ist nicht zu empfehlen:

- Eine Tabelle für jede reale Unterklasse
- Polymorphie wird nicht unterstützt
- Evolution des Schemas wird immer komplexer

```
package de.mathema.domain;

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
public class Customer extends User implements Serializable{
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="TABLE_PER_CLASS"/>
    <attributes>
      <id>
        <generated-value/>
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer"/>
</entity-mappings>
```

- ▼ Alle Unterklassen besitzen ihre eigene Tabelle
- ▼ Gemeinsame Benutzung von gleichen Primärschlüsseln
- ▼ Vorteile
 - Komplette Normalisierung
 - Datenintegrität und Evolution des Schemas sind einfach
- ▼ Nachteil
 - Weniger performant als Table Per Class Hierarchy

User
id integer primary key not null, firstName varchar(255), lastName varchar(255), ...

Customer
id integer primary key not null, type varchar(255), ...


```
package de.mathema.domain;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
public class Customer extends User implements Serializable{
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="JOINED" />
    <attributes>
      <id>
        <generated-value/>
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer" />
</entity-mappings>
```

Abfragen

Java Persistence Query API

- völlig ausgearbeitetes Java Interface
- `javax.persistence.Query` wird zur Laufzeit über `javax.persistence.EntityManager` generiert

```
package javax.persistence;

public interface EntityManager {
    public Query createQuery(String jpqlString);
    public Query createNamedQuery(String name);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString,
        Class resultClass);
    public Query createNativeQuery(String sqlString,
        String resultSetMapping);
}
```

- ▼ String-Basiert
- ▼ Dynamisch und statisch möglich
- ▼ ABER
 - Keine Typsicherheit
 - Prüfung erst zur Laufzeit
- ▼ Typischere Queries über CriteriaBuilder
 - Benötigt Metamodell der Entitäten
 - Erstellt per AnnotationProcessor (statisch)
 - Abfrage am EntityManager (dynamisch)

```
package javax.persistence;

public interface Query {
    public List getResultList();
    public Object getSingleResult();
    public int executeUpdate();
    public Query setMaxResults(int maxResult);
    public Query setFirstResult(int startPosition);
    public Query setHint(String hintName, Object value);
    public Query setParameter(String name, Object value);
    public Query setParameter(String name,
        Date value, TemporalType temporalType);
    public Query setParameter(String name, Calendar value,
        TemporalType temporalType);
    public Query setParameter(int position, Object value);
    public Query setParameter(int position, Date value,
        TemporalType temporalType);
    public Query setParameter(int position, Calendar value,
        TemporalType temporalType);
    public Query setFlushMode(FlushModeType flushMode);
}
```

```
try{
Query query = entityManager.createQuery(
    "select c from Customer c where c.firstName='"+firstName+
    "' and c.lastName='" + lastName +"'");
customer = (Customer)query.getSingleResult();
}catch(EntityNotFoundException notFound){
}catch(NonUniqueResultException nonUnique){
}
```

```
Query query = entityManager.createQuery(
    "select c from Customer c where c.firstName='"+firstname+
    "' and c.lastName='" + lastName +"'");
java.util.List<Customer> customers = query.getResultList();
```

▼ Query API unterstützt

- benannte Parameter(:name)
- JDBC-artige Parameter

```
...
Query query = manager.createQuery
("from Customer c where " +
"c.firstName=:firstName");

query.setParameter
(firstName, firstName);
...
```

```
...
Query query = manager.createQuery
("from Customer c where " +
"c.firstName=?1");
query.setParameter(1, firstName);
....
```


▼ Begrenzung auf das ResultSet

- Query API hat zwei eingebaute Methoden für dieses Szenario
- `Query.setMaxResult()`
- `Query.setFirstResult()`

```
public List getCustomer(int max, int index) {  
    Query query = manager.createQuery("from Customer cust");  
    return query.setMaxResults(max) .  
        setFirstResult(index) .  
        getResultList();  
}
```

▼ JPQL

- ist vollständig objektorientiert
- unterstützt Vererbung, Polymorphie und Assoziationen

▼ JPQL Features

- FROM, SELECT, WHERE, HAVING, ORDER BY, GROUP BY, JOIN Operationen,
- Aggregationsfunktionen
- Polymorphe Abfrage
- Projektion
- Dynamische Abfragen und benannte Parameter
- Subqueries
- Bulk Update
- Delete Update

▼ Abfrage sind case-insensitive

- **SeLeCT = sELEct = SELECT**
- *SELECT cust.lastName FROM Customer cust*
- ausgenommen sind Java Klassen Namen und Eigenschaften
 - **de.mathema.domain.CUSTOMER != de.mathema.domain.Customer**

▼ FROM-Klausel

- alle Instanzen vom Typ Customer
 - *FROM de.mathema.domain.Customer*
 - Alias "AS" kann in der Abfrage benutzt werden
 - *FROM Customer AS cust*

▼ Projektion

- *SELECT c.lastName, cc.number FROM Customer c, CreditCard cc
WHERE c.creditCard = cc*

▼ IN Operator

- *SELECT d FROM Reservation As r, IN(r.drivers) d*

▼ LEFT JOIN

- *SELECT c.firstName, c.lastName, p.number FROM Customer c LEFT JOIN c.phones p*

▼ FETCH JOIN

- *SELECT c FROM Customer c LEFT JOIN FETCH c.phones*

▼ DISTINCT

- *SELECT DISTINCT c FROM Reservation AS res, IN (res.drivers) cust*

▼ Aggregationsfunktionen

- AVG(), SUM(), MIN(), MAX(), COUNT(*), COUNT(), ...
 - *SELECT COUNT(cust) FROM Customer AS cust*

▼ Polymorphe Abfrage

- Abfrage auf eine Basisklasse gibt auch die Unterklassen zurück
- *FROM User user*

▼ WHERE-Klausel

- *FROM Customer cust WHERE cust.id = 1*

▼ ORDER BY-Klausel

- *FROM Customer cust ORDER BY cust.firstName ASC*

▼ GROUP BY-Klausel

- *SELECT c.lastName, COUNT(c) FROM Customer c GROUP BY c.id*

▼ Bulk UPDATE

- *UPDATE* Reservation res SET res.car.price = (res.car.price + 6) WHERE EXISTS (SELECT cust FROM res.customers cust WHERE cust.firstName = 'Serge' AND cust.lastName='Pagop')

▼ Bulk DELETE

- *DELETE* Reservation res WHERE EXISTS (SELECT cust WHERE cust.firstName = 'Francis' AND cust.lastName='Pouatcha')
- Vor der Ausführung von Bulk Operationen empfiehlt die Java Persistence API, die Methoden *EntityManager.flush()* und *EntityManager.clear()* auszuführen

▼ Subqueries

- *SELECT s FROM Supplier s WHERE (SELECT COUNT(c) FROM s.cars c) > 10*

- Formulierung einer nativen Abfrage mittels `EntityManager.createNativeQuery()`

```
Query query = entityManager.createNativeQuery  
(  
    "SELECT p.id, p.number, p.type  
      FROM PHONE AS p",  
    Phone.class  
);
```

```
package javax.persistence;
public @interface NamedQueries {
    NamedQuery [] value ();
}
```

```
package javax.persistence;
public @interface NamedQuery {
    String name();
    String query();
    QueryHint[] hints()
    default {};
}
```

```
package de.mathema.domain;
@NamedQueries({
    @NamedQuery(name="getCustomer",
        query="SELECT c FROM Customer AS c WHERE c.lastName=:lastname")
})
@Entity
public class Customer extends User implements Serializable{
    ...
}
// in der SLSB
Query query =
    entityManager.createNamedQuery("getCustomer") .
    setParameter("lastName", lastName);
```



```
<entity-mappings>
  <named-query name="getCustomer">
    <query>
      SELECT c FROM Customer
      AS c WHERE c = :lastName
    </query>
  </named-query>
</entity-mappings>
```

- „Einfache“ Variante der Criteria-API
- Nicht typsicher

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
Root<Kunde> kunde = cq.from(Kunde.class);
Path<String> name = kunde.get("name");
cq.select(kunde).
    where(cb.in(name).value("Müller")
        .value("Huber").value("Mayer"));
Query query = em.createQuery(cq);
```

```
@Entity
public class Kunde {
    @NotNull
    @Size(min=2,max=25)
    String name;
    @NotNull
    @Size(min=2,max=25)
    String vorname;
    @ElementCollection
    Set<Telefon> telefonnummern;
}
```

```
@StaticMetamodel(Kunde.class)
public class Kunde_ {
    public static volatile
        SingularAttribute<Kunde, String> name;
    public static volatile
        SingularAttribute<Kunde, String> vorname;
    public static volatile
        SetAttribute<Kunde, Telefon> telefonnummern;
}
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
Root<Kunde> kunde = cq.from(Kunde.class);
Path<String> name = kunde.get(Kunde_.name);
cq.select(kunde).
    where(cb.in(name).value("Müller").value("Huber").value("Mayer"));
Query query = em.createQuery(cq);
```

//Äquivalentes Query via JPA-QL

```
Query q = em.createQuery(
    "select k from Kunde k where k.name in ('Müller','Huber','Mayer')");
```

Entity Callbacks

- Java Persistence API stellt für die aktive Steuerung des Lebenszyklus von Persistence Entities folgende Meta-Informationen zur Verfügung:
 - `@javax.persistence.PrePersist`
 - tritt vor dem Aufruf von `EntityManager.persist()` auf
 - `@javax.persistence.PostPersist`
 - tritt nach dem Aufruf von `EntityManager.persist()` auf
 - `@javax.persistence.PreRemove`
 - tritt vor dem Aufruf von `EntityManager.remove()` auf
 - `@javax.persistence.PostRemove`
 - tritt nach dem Aufruf von `EntityManager.remove()` auf
 - `@javax.persistence.PreUpdate`
 - tritt vor der Synchronisation von Änderungen in die Datenbank auf
 - `@javax.persistence.PostUpdate`
 - tritt nach der Synchronisation von Änderungen in die Datenbank auf
 - `@javax.persistence.PostLoad`
 - tritt nach dem Aufruf von `manager.find()` oder `manager.getReference()` auf

```
package de.mathema.domain;
...
@Entity
public class Customer extends User implements Serializable{
    public enum Type{PRIVATE, ORGANISATION}

    private CreditCard creditCard;
    private Type type;
    private Collection<Phone> phones = new ArrayList<Phone>();
    private Collection<Reservation>
        reservations = new ArrayList<Reservation>();

    ...

    @PostPersist
    protected void afterPersist(){
        System.out.println("After Persist");
    }

    @PostLoad
    protected void afterLoad(){
        System.out.println("After Load");
    }
}
```

- ▼ EntityListener kann auf definierten Ereignissen des Lebenszyklus einer Entity angebracht werden
 - `@PrePersist` – wenn die Anwendung `persist()` aufruft
 - `@PostPersist` – nach dem SQL `INSERT`
 - `@PreRemove` – wenn die Anwendung `remove()` aufruft
 - `@PostRemove` – nach dem SQL `DELETE`
 - `@PreUpdate` – wenn Container feststellt, daß eine Instanz „dirty“ ist
 - `@PostUpdate` – nach dem SQL `UPDATE`
 - `@PostLoad` – nach dem Aufladen einer Instanz

▼ EntityListeners Annotation

```
package javax.persistence;  
public @interface EntityListeners {  
    Class[] value();  
}
```



```
package de.mathema.listeners;
...
public class AuditListener {
    @PostPersist
    void postInsert(Object object){
        System.out.println("Inserted entity: " + object.getClass().getName());
    }
    @PostLoad
    void postLoad(Object object){
        System.out.println("Loaded entity: " + object.getClass().getName());
    }
}
```

```
package de.mathema.domain;

@Entity
@EntityListeners({AuditListener.class})
public class Customer extends User implements Serializable{
    public enum Type{PRIVATE, ORGANISATION}
    private CreditCard creditCard;
    ...
}
```

```
<entity class="de.mathema.domain.Customer">
  <entity-listeners>
    <entity-listener
      class="de.mathema.listeners.AuditListener">
    </entity-listener>
  </entity-listeners>
</entity>
```

Teil VII

Transaktionen

- ▼ Transaktion (TX) involviert üblicherweise einen Austausch zwischen zwei Parteien
- ▼ **ACID**: die vier Prinzipien des Transaktionsservices
 - **A**tomarität: **TX** wird entweder ganz oder gar nicht ausgeführt
 - **K**onsistenz: konsistenter Datenzustand nach Beendigung der **TX**
 - **I**solation: **TX** beeinflussen sich nicht gegenseitig
 - **D**urabilität: das Ergebnis einer **TX** ist dauerhaft
- ▼ besteht aus einem oder mehreren DB Zugriffen
 - Rollback
 - wenn einer der Zugriffe scheitert
 - durch einen programmatischen Einsatz
 - Commit
 - wenn alle Zugriffe erfolgreich verlaufen

- ▼ Transaktions-Manager
- ▼ Applikation Server
- ▼ Ressource-Manager
- ▼ Anwendung (EJB Komponenten)

▼ Client benötigt atomare Operation auf viele Datenbanken

▼ Beispiele:

- Der Client möchte auf zwei separate Datenbanken schreiben
- Der Client versucht, die atomare Operation abzuschließen (commit)

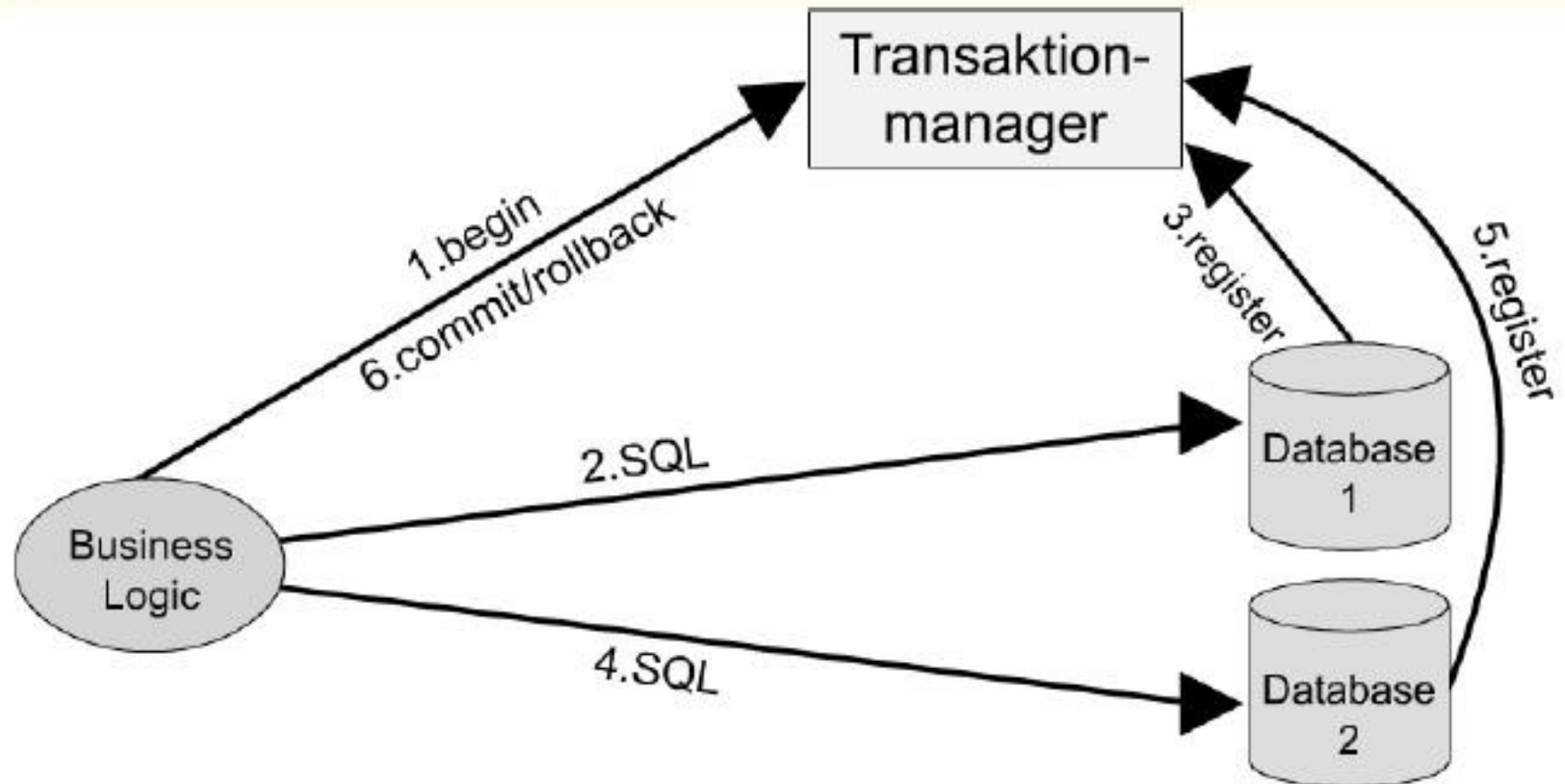
▼ Problem:

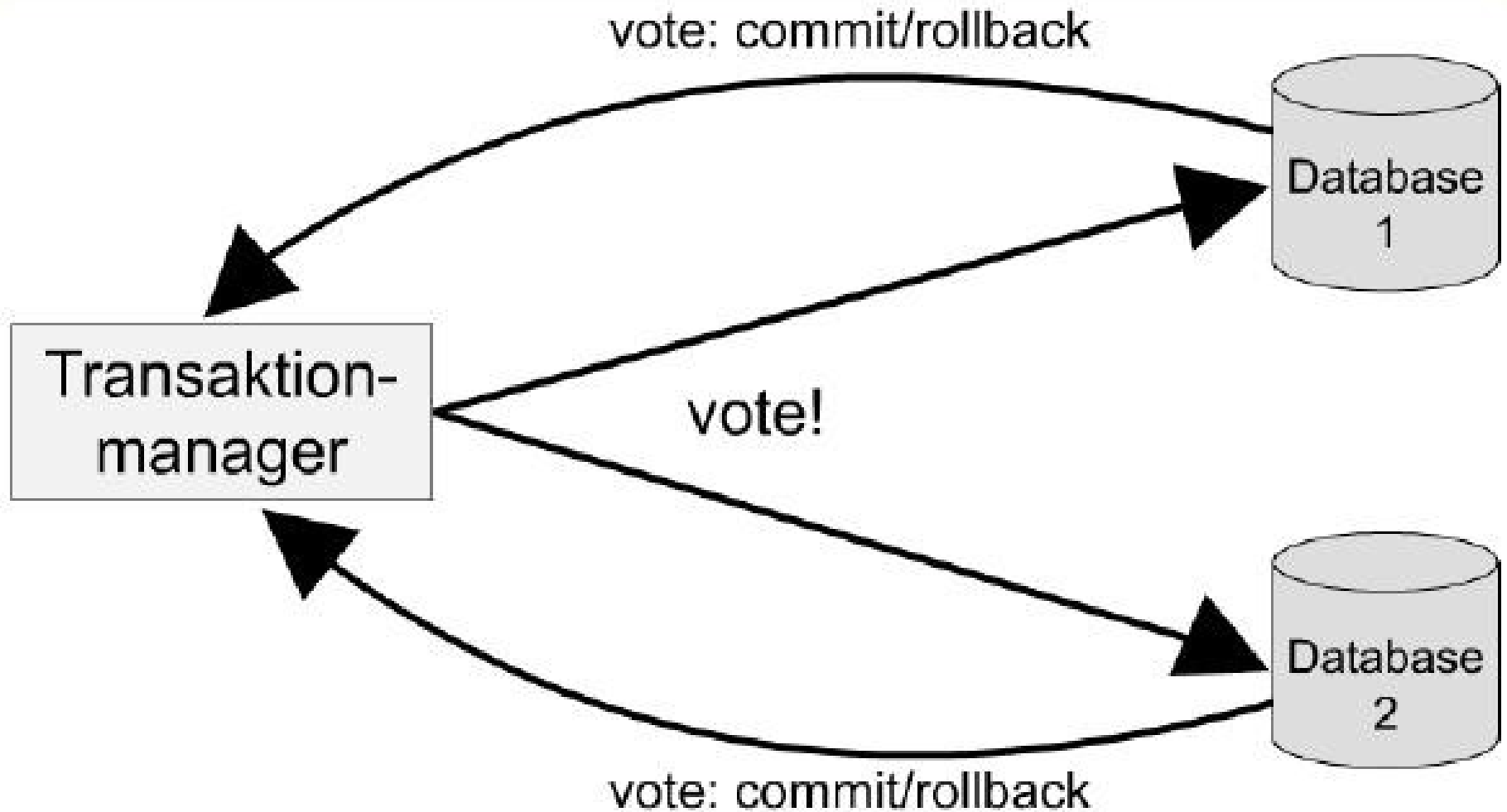
- Was passiert, wenn Datenbank 1 ihren Teil der atomaren Operation mit Commit abschließt, aber Datenbank 2 ein Rollback verursacht?

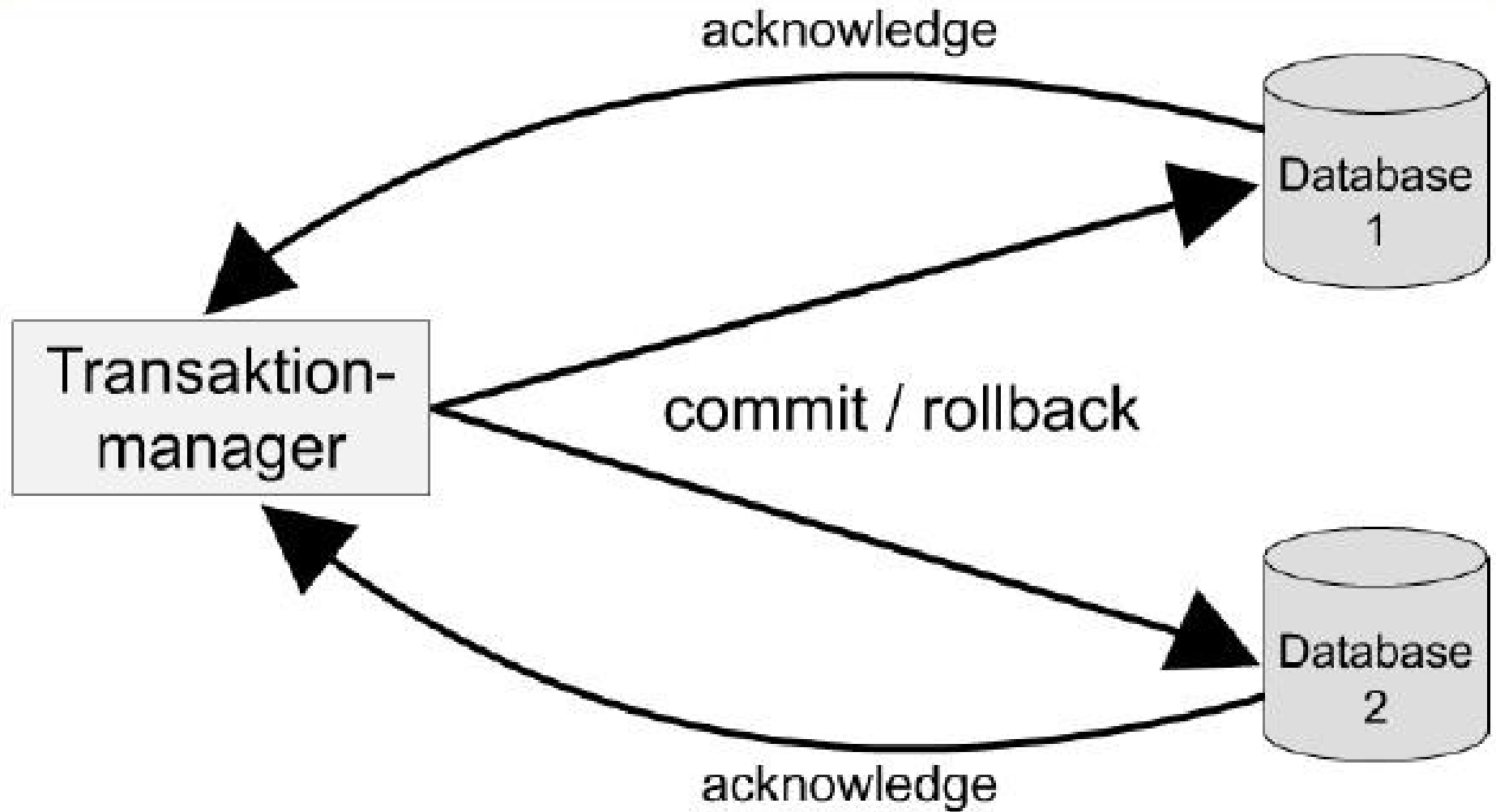
▼ Lösung:

- Verteilte Transaktionen müssen vom Transaktions-Manager koordiniert werden

▼ Transaktions-Manager benutzt **Two-Phase Commit** Protokoll







▼ Java Transaction API (JTA)

- Abstraktionsschicht für den Transaktions-Manager
- `javax.transaction.UserTransaction` Interface muss vom Applikation Server implementiert werden

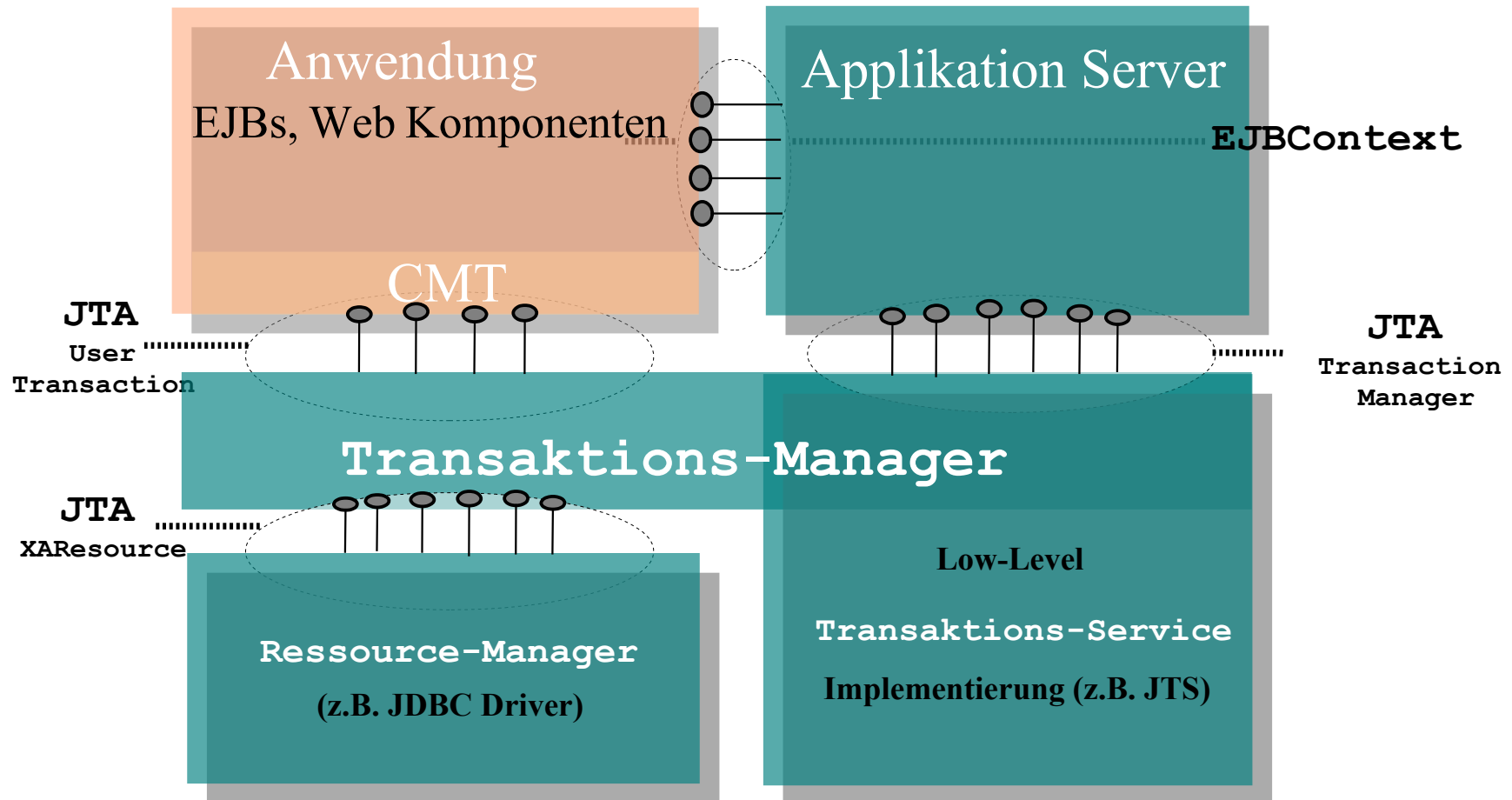
▼ Java Transaction Service (JTS)

- implementiert
 - Transaktions-Manager auf der hohen Ebene
 - CORBA OTS auf der niedrigen Ebene
- kann vom Applikation Server benutzt werden

▼ CORBA Object Transaction Service (OTS)

- verteilte Transaktionen (Two-Phase Commit)
- unterstützt inter-ORB Interoperabilität

- ▼ Java EE Applikation Server muss Two-Phase Commit (2PC) unterstützen
- ▼ Verteilte Transaktion kann innerhalb
 - Web Komponenten
 - EJB Komponenten
 - Application Client (wenn der Client-Container 2PC unterstützt)
gestartet werden
- ▼ Ressource-Managers können:
 - Datenbanken (MySQL, ORACLE, ...)
 - Connectors
 - JMS Providers (JBoss Messaging, IBM MQSeries, ...)
- ▼ **Anmerkung:** Nicht alle Ressource-Manager unterstützen Two-Phase Commit



▼ Container-Managed Transaction

- spezifiziere Enterprise Bean mit der Annotation `@javax.ejb.TransactionManagement` (`javax.ejb.TransactionManagementType.CONTAINER`)
- Transaktionsabgrenzungen können durch die Annotation `@javax.ejb.TransactionAttribute` oder im XML Deployment Deskriptor gesetzt werden

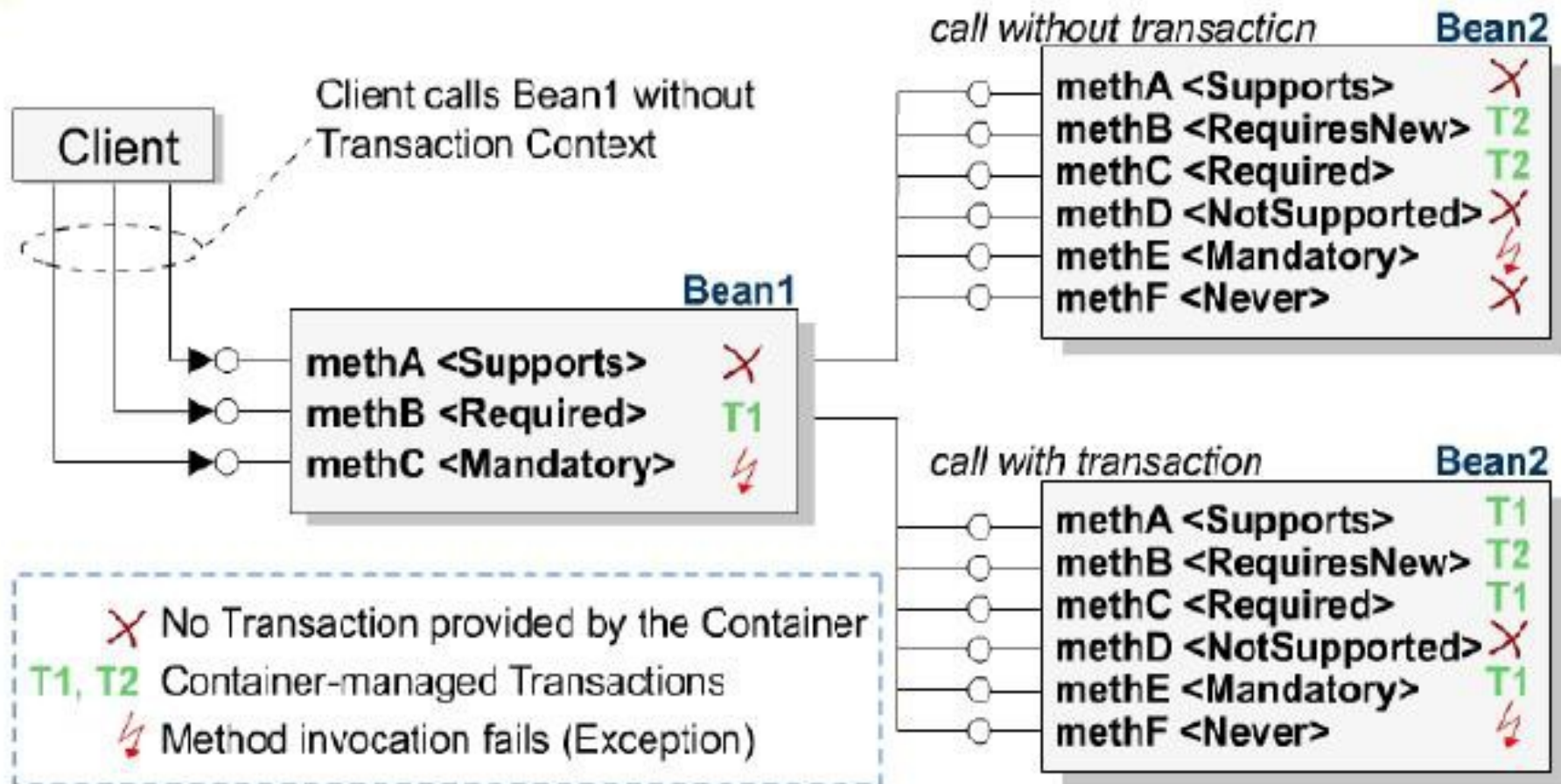
▼ Bean-managed Transaction

- spezifiziere Enterprise Bean mit der Annotation `@javax.ejb.TransactionManagement` (`javax.ejb.TransactionManagementType.BEAN`)
- Transaktionsabgrenzungen werden programmatisch gelöst

▼ Client-managed Transaction

- Transaktion wird vom Client gestartet/committed
- Kontext der Transaktion breitet sich auf die Bean aus

- ▼ Transaktionen können vom Container am Anfang/Ende einer Methode gestartet/committed werden
- ▼ Transaktionsverhalten von EJBs kann durch das Setzen von Attributen in die Annotation `@javax.ejb.TransactionAttribute` oder im EJB Deployment Deskriptor kontrolliert werden
- ▼ Transaktionsattribute sind:
 - **NotSupported**
 - **Supports**
 - **Required**
 - **RequiresNew**
 - **Mandatory**
 - **Never**



```
package javax.ejb;

public enum TransactionAttributeType {
    MANDATORY,
    REQUIRED,
    REQUIRES_NEW,
    SUPPORTS,
    NOT_SUPPORTED,
    NEVER
}
```

```
package javax.ejb;

public @interface TransactionAttribute {
    TransactionAttributeType value()
        default TransactionAttributeType.REQUIRED;
}
```



```
@javax.ejb.Stateful
//Default-Wert des @javax.ejb.TransactionManagement (CONTAINER)
@javax.ejb.TransactionAttribute(NOT_SUPPORTED)
public class RentingAgentBean implements RentingAgentRemote{

    public Customer findOrPersistCustomer(String last,String first){...}

    @javax.ejb.TransactionAttribute(REQUIRED) //überschreibt NOT_SUPPORTED
    public ConfirmationDO makeReservation(CreditCardDO cc, TimePlaceDO p)
        throws IncompleteConversationalStateException {
        ...
    }
}
```

▼ @javax.ejb.TransactionAttribute

- kann für eine Methode angelegt werden oder
- auf einer Session Bean Klasse benutzt werden, um das Default Transaktionsattribut zu setzen

```
<ejb-jar>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>RentingAgentEJB</ejb-name>
        <method-name> * </method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>RentingAgentEJB</ejb-name>
        <method-name>
          makeReservation
        </method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

- ▼ Erhält eine `javax.transaction.UserTransaction` aus `javax.ejb.EJBContext` oder `@javax.annotation.Resource`
- ▼ Interface von `javax.transaction.UserTransaction`
 - `public void begin();`
 - `public void commit();`
 - `public void rollback();`
 - `public void getStatus();`
 - `public void setRollbackOnly();`
 - `public void setTransactionTimeout(int seconds);`
- ▼ programmatisches Starten und Beenden von Transaktionen
- ▼ nur für Session Beans

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ProcessPaymentBean
    implements ProcessPaymentLocal{
    @Resource SessionContext ejbContext;

    public void someMethod( )
        throws TransactionRolledbackException {
        try {
            UserTransaction ut =
                ejbContext.getUserTransaction( );
            ut.begin( );
            ...
            ut.commit( );
        } catch(IllegalStateException ise) {...}
        catch(SystemException se) {...}
        catch(HeuristicRollbackException hre) {...}
        catch(HeuristicMixedException hme) {...}
        ...
    }
}
```

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ProcessPaymentBean
    implements ProcessPaymentLocal{

    @Resource UserTransaction ut;

    public void someMethod( )
        throws TransactionRolledbackException {
        try {
            ut.begin( );
            ...
            ut.commit( );
        } catch(IllegalStateException ise) {...}
            catch(SystemException se) {...}
            catch(HeuristicRollbackException hre) {...}
            catch(HeuristicMixedException hme) {...}

    }
}
```

- ▼ Transaktionen werden vom Applikation Server verwaltet
 - direktes Commit oder Rollback auf der JDBC Verbindung ist nicht erlaubt (Anwendung auf CMT und BMT)
- ▼ Rollback kann erzwungen werden
 - CMT: `EJBContext.setRollbackOnly()`
 - BMT: `UserTransaction.setRollbackOnly()`

▼ **Rollback:**

Zustände müssen vor dem Beginn der Transaktion wiederhergestellt werden

▼ **Commit:**

Zustände müssen in die Datenbank geschrieben werden

▼ **Wer ist dafür verantwortlich?**

■ **Stateful Session Beans:**

Wiederherstellung von Zuständen muss vom Bean Provider durch die Nutzung des `javax.ejb.SessionSynchronization` Interfaces erledigt werden

▼ Wenn Session Beans

- sich über die Commits und Rollbacks Zustände informieren wollen
- die gleichen Zustände wie gespeichert in der Datenbank haben wollen

▼ dann müssen sie das Interface **`SessionSynchronisation`** implementieren

- `public void afterBegin() ;`
- `public void beforeCompletion() ;`
- `public void afterCompletion(boolean committed) ;`

▼ Nur Stateful Session Beans mit CMT können dieses Interface implementieren

▼ Daten in der Datenbank werden automatisch verwaltet

▼ `public void afterBegin() ;`

- wird nach dem Beginn einer Transaktion aufgerufen
- Bean speichert die Zustände, die beim Rollback benötigt werden

▼ `public void beforeCompletion() ;`

- wird nach dem Start des Two-Phase Commits aufgerufen
- Bean schreibt Daten aus dem Cache in die Datenbank

▼ `afterCompletion(boolean committed) ;`

- wird aufgerufen, nachdem die Commit oder Rollback auf der Transaktion ausgeführt wurde
- Bean muss die Zustände zurücksetzen, wenn `committed==false` (Rollback)

- ▼ Möglich, falls der Applikation Server die Ausbreitung des Transaktionskontexts unterstützt
- ▼ Das Interface `javax.transaction.Usertransaction` kann referenziert werden:
 - JNDI ENC
 - Injizierung der Ressource

```
//JNDI ENC
Context jndiCntx = new InitialContex( );
UserTransaction ut = (UserTransaction)
jndiCntx.lookup("java:comp/UserTransaction");

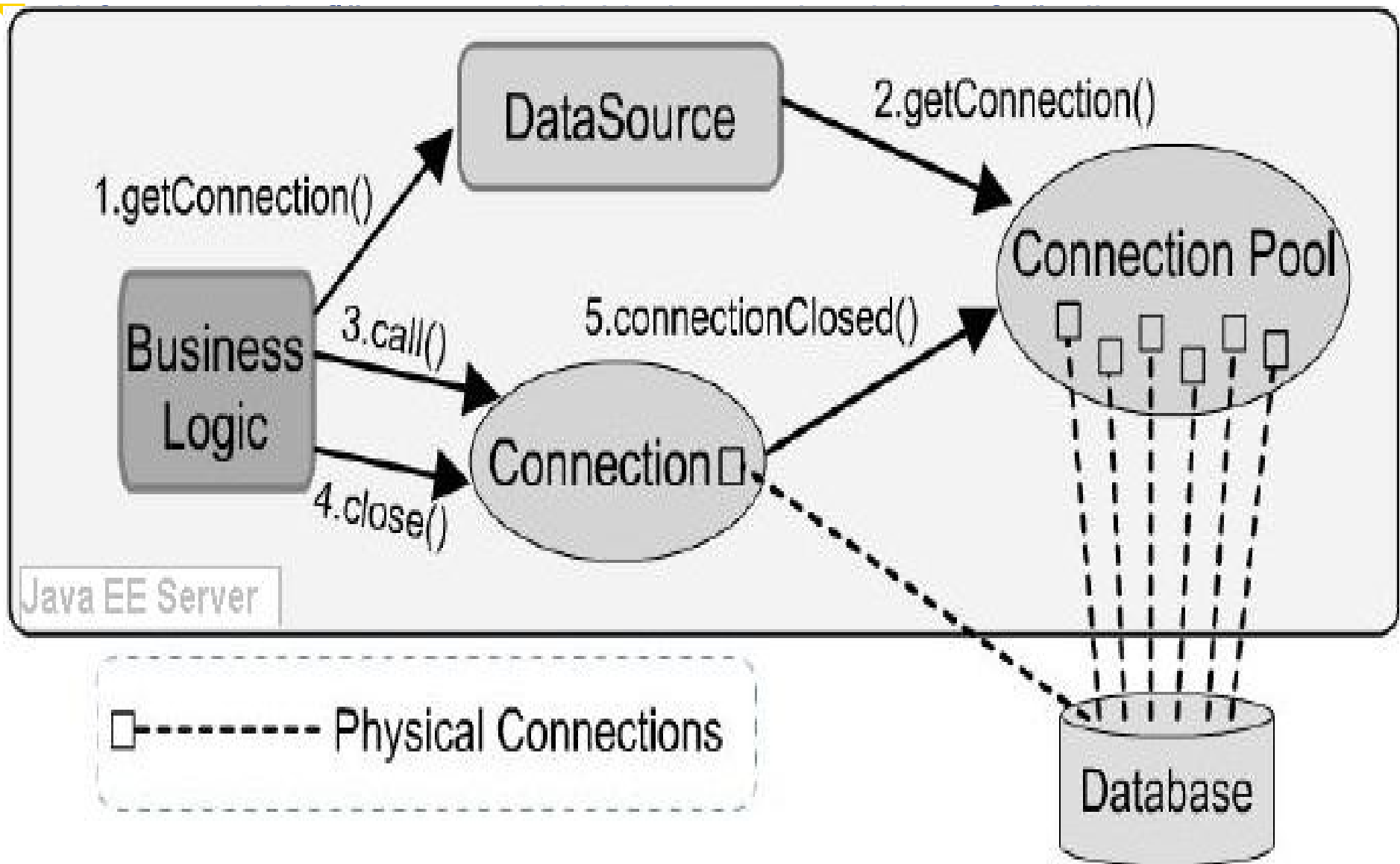
//Injizierung der Resource
@Resource UserTransaction ut;
```

▼ Vor dem Ende der Transaktion

- BMT Session Beans: Vor `commit()` und `rollbackk()`
- CMT Session Beans mit `SessionSynchronisation`: `beforeCommit()`
- CMT Session Beans ohne `SessionSynchronisation`: **vor** dem Ende jeder Methode

▼ In Methoden, die mit `@javax.ejb.PrePassivate` annotiert werden

▼ Am einfachsten: am Ende jeder Methode



▼ Zwei Typen von Exceptions:

- System Exceptions (unchecked Exceptions)
 - Exceptions und Fehler, die von Systemsdiensten verursacht werden
 - `java.lang.NullPointerException`
 - `java.lang.IndexOutOfBoundsException`
 - `java.lang.OutOfMemoryError`
 - `javax.transaction.RollbackException`
 - ...
- Application Exceptions (checked Exceptions)
 - Exceptions, die von der Geschäftslogik geworfen werden
 - `IncompleteConversationalState`
 - `PaymentException`

- ▶ Alle **RuntimeExceptions**, **Errors** und unerwartete Exceptions aus dem Applikation Server oder Ressourcen
- ▶ Bean Provider Verantwortlichkeiten
 - **Errors** und **RuntimeExceptions** müssen im Container verbreitet werden
 - Andere Exceptions müssen in einer neuen **EJBException** abgefangen und verpackt werden
- ▶ Applikation Server Verantwortlichkeiten
 - Kommunikation zu den Clients als **RemoteException**
 - automatisches Rollback der Transaktion
 - Bean Instanz wird abgeworfen
- ▶ **RuntimeException** und **RemoteException** Unterklassen können durch die Nutzung der Annotation **@javax.ejb.ApplicationException** in Applikation Exceptions umgeformt werden

▼ Methoden der Geschäftslogik

- benutzerdefinierte Exceptions

▼ werden unverändert an den Client weitergegeben

- müssen im Remote Business Interface deklariert werden

▼ kein automatisches Rollback der Transaktion

- muss manuell durchgeführt werden, falls die Zustände einer Bean inkonsistent bleiben

▼ `@javax.ejb.ApplicationException`

- kann verwendet werden, um eine Application Exception zu erzwingen und die Transaktion automatisch zum Rollback zu führen

```
package javax.ejb;
public @interface ApplicationException{
    boolean rollback() default false;
}
```

```
@ApplicationException(rollback=true)
public class ProcessPaymentException extends Exception {
    public ProcessPaymentException() {
        super();
    }
    public ProcessPaymentException(String message) {
        super(message);
    }
}
```



```
<ejb-jar>
  <assembly-descriptor>
    <application-exception>
      <exception-class>j
        java.sql.SQLException</exception-class>
      <rollback>true</rollback>
    </application-exception>
  </assembly-descriptor>
</ejb-jar>
```

- ▼ **java.sql.SQLException** kann jetzt als Application Exception benutzt werden – sie verursacht ein Rollback

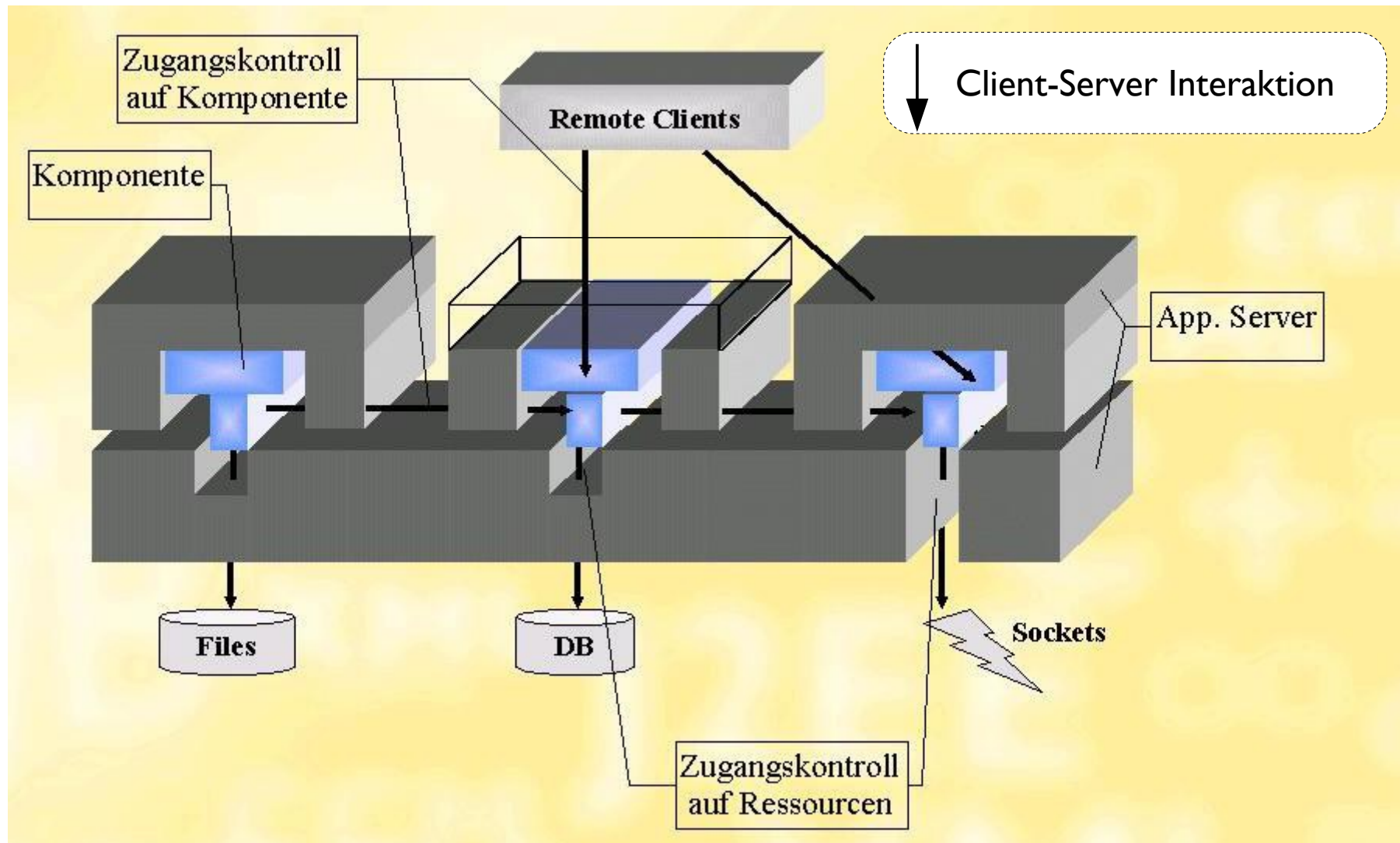
- ▼ Das bestmögliche Isolation Level sollte gewählt werden
 - Read Uncommitted, Read Committed, Repeatable Read oder Serializable
 - kein EJB Problem, sondern ein Datenbank-Problem
- ▼ Problem der Nebenläufigkeit kann durch optimistisches Locking gelöst werden
 - Verwende `@javax.persistence.Version` aus der Java Persistence API
- ▼ Programmatische Sperre
 - durch `javax.persistence.EntityManager.lock()`
- ▼ Lokale Transaktion
 - Server kann globale Transaktion(two-phase Commit) vermeiden, falls die Transaktion nur aus einem Ressource-Manager besteht
 - muss im Deployment Deskriptor spezifiziert werden

▼ Caching

- Aufwand durch Laden von Zuständen am Anfang einer Transaktion kann vermindert werden, wenn diese Zustände im Cache gehalten werden

Teil VIII

Sicherheit



- ▶ Client Anwendung muss auf eine Identität des Benutzers zugreifen können z.B. Password, PIN, ...
- ▶ Viele Applikations-Server führen die Authentifizierung mittels des JNDI API durch
- ▶ Ein anderer Mechanismus kann durch die Verwendung JAAS Spezifikation implementiert werden

```
public Context getInitialContext() throws Exception{
    Properties env = new Properties();
    env.setProperty(Context.SECURITY_PRINCIPAL, this.user);
    env.setProperty(Context.SECURITY_CREDENTIALS, this.password);
    env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
        "org.jboss.security.jndi.JndiLoginInitialContextFactory");
    return new InitialContext(env);
}
```

- ▼ Benutzer einer EJB Anwendung werden in Gruppen eingeteilt
 - Gruppe wird als Rolle bezeichnet
 - Benutzer können dabei mehreren Gruppen angehören
- ▼ Je nach Gruppenzugehörigkeit erteilt oder sperrt die EJB-Anwendung das Zugriffsrecht auf Ressourcen
- ▼ Access Control in EJB wird
 - auf Per-Method-Basis
 - und Per-Role-Basis festgesetzt
- ▼ Autorisierung in EJB
 - Deklariert Rollen, auf die programmatisch zugegriffen werden kann
 - Legt Berechtigungen für jede Methode fest
- ▼ Autorisierung kann durch Java Annotations oder durch den Deployment Deskriptor `ejb-jar.xml` festgelegt werden

▼ `@javax.annotation.security.RolesAllowed()`

- Die Werte von `@RolesAllowed` enthalten eine Liste von Rollen, die festlegt, welchen Benutzern einer Gruppe die Ausführung einer Methode erlaubt ist

▼ `@javax.annotation.security.PermitAll`

- spezifiziert, dass alle Gruppen die annotierte Methode ausführen können

▼ `@javax.annotation.security.DenyAll`

- alle Methoden können nicht ausgeführt werden

▼ `EJB Container`

- wirft eine `javax.ejb.EJBAccessException`, falls ein Client auf eine nicht erlaubte Methode zugreift


```
package javax.annotation.  
    Security;  
  
public @interface  
    RolesAllowed{  
    String[] value();  
}
```

```
package javax.annotation.  
    Security;  
  
public @interface  
    PermitAll{  
}
```

```
package javax.annotation  
    .security;  
  
public @interface  
    DenyAll{  
}
```

```
package de.mathema.slsb;

...
@javax.ejb.Stateless
@javax.annotation.security.RolesAllowed("AUTHORIZED_CUSTOMER")
public class ProcessPaymentBean implements
ProcessPaymentLocal, ProcessPaymentRemote {
    ...
    public boolean byCreditCard(Customer customer,
        CreditCardDO card, double amount)
        throws ProcessPaymentException {
        ...
    }

    private boolean process(long customerID, double amount,
        String type, String creditNumber,
        java.sql.Date creditExpDate)
        throws ProcessPaymentException {
        ...
    }
}
```

```
<ejb-jar version="3.0">
```

```
  <assembly-descriptor>
```

```
    <security-role>
```

```
      <description>
```

Diese Rolle repräsentiert einen bevollmächtigten Kunden

```
    </description>
```

```
      <role-name>AUTHORIZED_CUSTOMER</role-name>
```

```
    </security-role>
```

```
  <method-permission>
```

```
    <role-name>AUTHORIZED_CUSTOMER</role-name>
```

```
    <method>
```

```
      <ejb-name>ProcessPaymentBean</ejb-name>
```

```
      <method-name>byCreditCard</method-name>
```

```
    </method>
```

```
  </method-permission>
```

```
  </assembly-descriptor>
```

```
</ejb-jar>
```

- ▼ Anstatt der Anwendung der @RolesAllowed Annotation oder des **<method-permission>** Elements auf Methoden kann auch die @RunAs Annotation oder das **<run-as>** Element auf eine Enterprise Bean spezifiziert werden, um eine spezifische Rolle festzusetzen
- ▼ Message-Driven Beans (MDBs) können die @RunAs Annotation benutzen

```
package de.mathema.sfsb;  
...  
  
@javax.ejb.Stateful  
@javax.annotation.security.RunAs("AUTHORIZED_CUSTOMER")  
public class RentingAgentBean implements RentingAgentRemote{  
    ...  
}
```

```
<ejb-jar version="3.0">  
  <enterprise-beans>  
    <session>  
      <ejb-name>RentingAgentBean</ejb-name>  
      <security-identity>  
        <run-as>  
          <role-name>AUTHORIZED_CUSTOMER</role-name>  
        </run-as>  
      </security-identity>  
    </session>  
  </enterprise-beans></ejb-jar>
```

▼ **javax.ejb.EJBContext** Interface definiert zwei Methoden

- `javax.security.Principal getCallerPrincipal()`
- `boolean isCallerInRole(String roleName);`

▼ **Verwendung**

- vermeide programmatische Sicherheit, wenn die automatische Autorisierung möglich ist
- sinnvoll, wenn ein Benutzer genau eine Session Bean benutzen darf

▼ **User Role Namen können mittels**

- `@javax.annotation.security.DeclareRoles()` annotiert
- oder im ejb-jar.xml Deployment Deskriptor beschrieben werden

```
package javax.annotation.security;  
  
public @interface DeclareRoles{  
    String[] value();  
}
```

```
package de.mathema.slsb;
...
@javax.ejb.Stateless
@javax.annotation.security.DeclareRoles("JUNIOR_RENTING_AGENT")
public class ProcessPaymentBean implements
    ProcessPaymentLocal, ProcessPaymentRemote {
    ...
    @Resource SessionContext ctx;
    @Resource double maximumAmount = 80.0;

    private boolean process(long id, double amount,
        String type, String number, Date creditExpDate)
        throws ProcessPaymentException {

        if (amount > maximumAmount &&
            ctx.isCallerInRole("JUNIOR_RENTING_AGENT")) {
            throw new ProcessPaymentException(
                "Agent is not authorized to make such a large purchase. " +
                "Manager approval is required.");
        }
    }
    ...
}
```



```
<ejb-jar version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>ProcessPaymentBean</ejb-name>
      <security-role-ref>
        <role-name>JUNIOR_RENTING_AGENT</role-name>
      </security-role-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```