Verteilte Systeme
Nachrichten und Java Enterprise

Inhalt dieses Kapitels

Nachrichtenbasierte Middleware

- Message Passing und Message Oriented Middleware
- Zugriff auf Namensverzeichnisse mit JNDI
- Nachrichtenbasierte Anwendungen mit JMS

Enterprise Java Beans

- Komponenten und Container in Java Enterprise
- Programmierung und Deployment von EJBs
- Zugriff auf entfernte Enterprise Java Beans
- Nutzung der Java Persistence API

Nachteile entfernter Aufrufe

Jeder Aufruf kann immer nur einen Empfänger besitzen

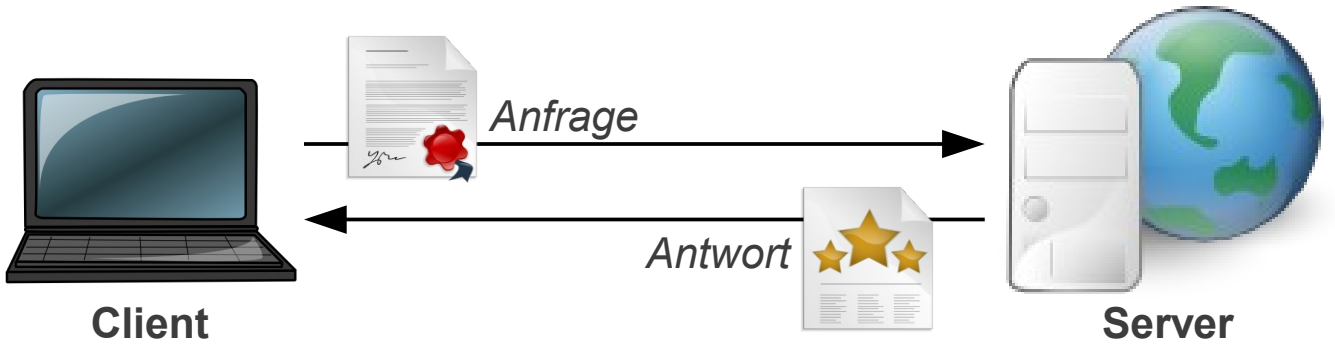
Keine Lastverteilung zwischen mehreren Empfängern vorgesehen

Aufrufe schlagen fehl, wenn der Server nicht erreichbar ist

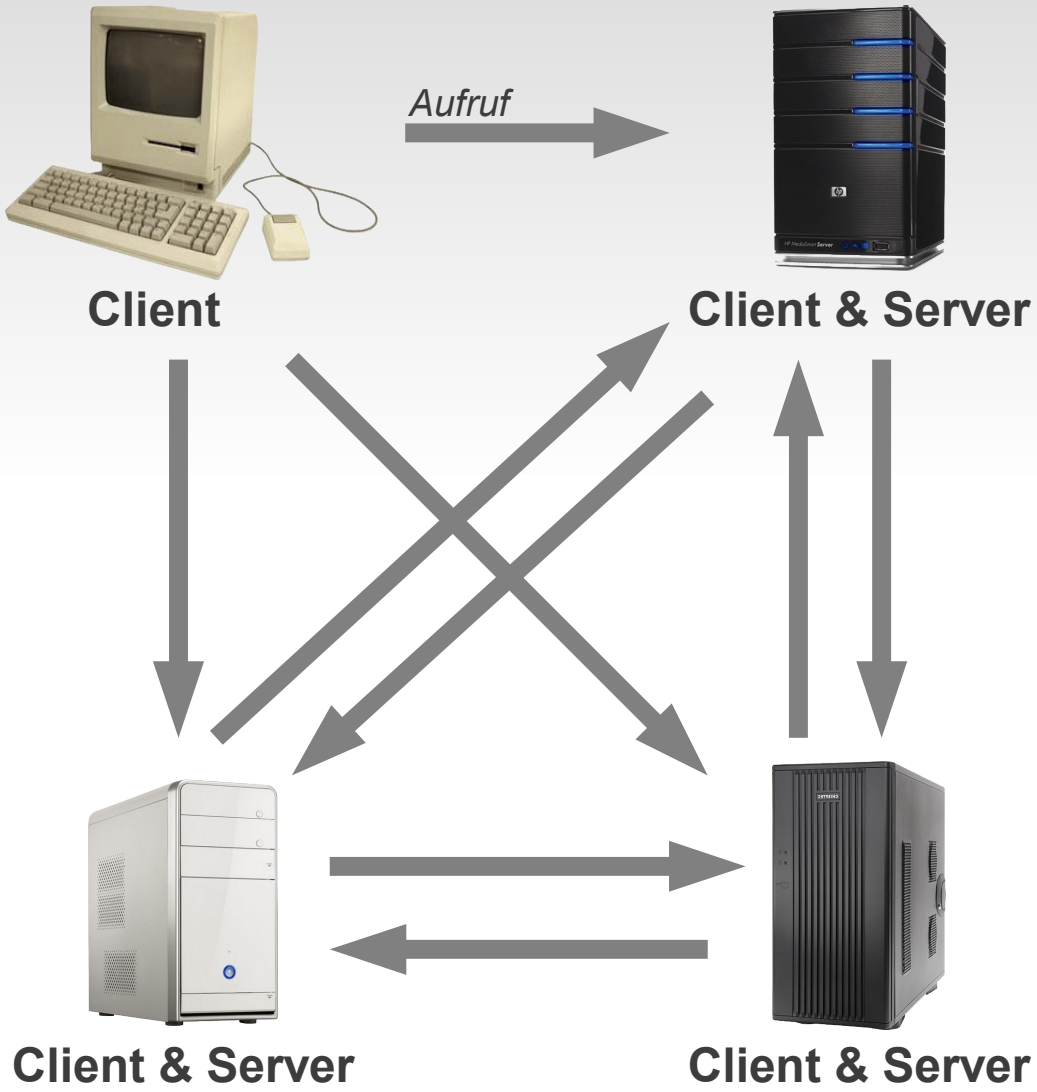
Schlechte Ausnutzung der Nebenläufigkeit, da die Clients immer warten müssen, solange der Server ihre Anfragen bearbeitet

Die feste Rollenverteilung ist für viele Anwendungen ungeeignet

Erhöhte Komplexität bei sehr vielen Systemkomponenten



Beispiel: Erhöhte Komplexität

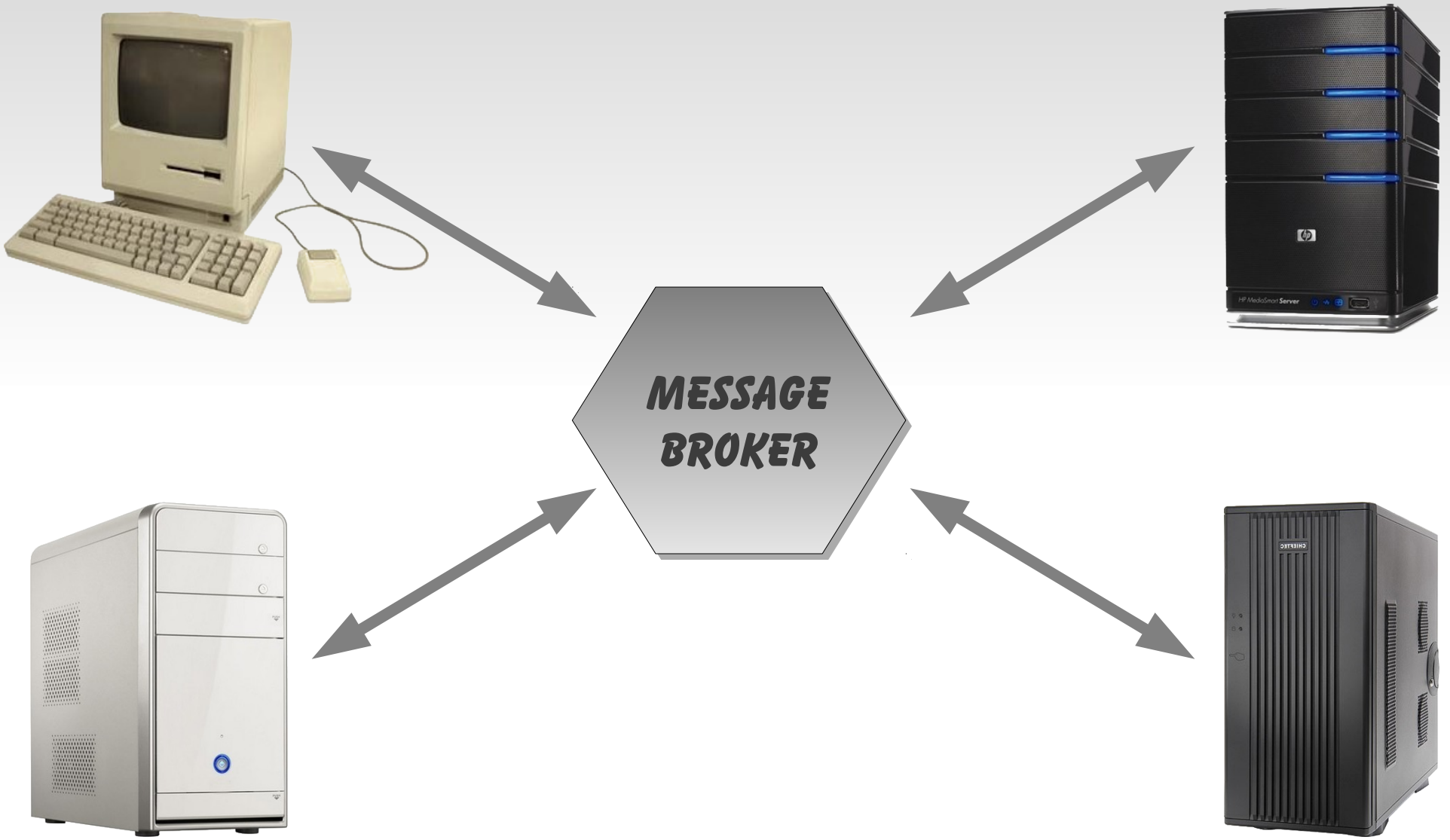


Anzahl der Verbindungen: 9

Verteilte Systeme, in denen viele Komponenten sowohl Client als auch Server sind, können schnell sehr komplex werden, da im Prinzip jedes System jedes andere System aufrufen muss.

Namensdienste (sofern sie von der Middleware überhaupt unterstützt werden) können dabei helfen, diese Komplexität zu verringern, spätestens jedoch, wenn ein Aufruf, der bisher nur an ein System ging, künftig an noch ein zweites System gehen soll, stellen Sie fest, dass Sie aufwendig alle Aufrufer anpassen müssen.

Entkopplung der Systeme



Anzahl der Verbindungen: 4

Entkopplung der Systeme

Entkopplung aller Systeme durch den Message Broker

Keine direkten Aufrufe zwischen den beteiligten Systemen

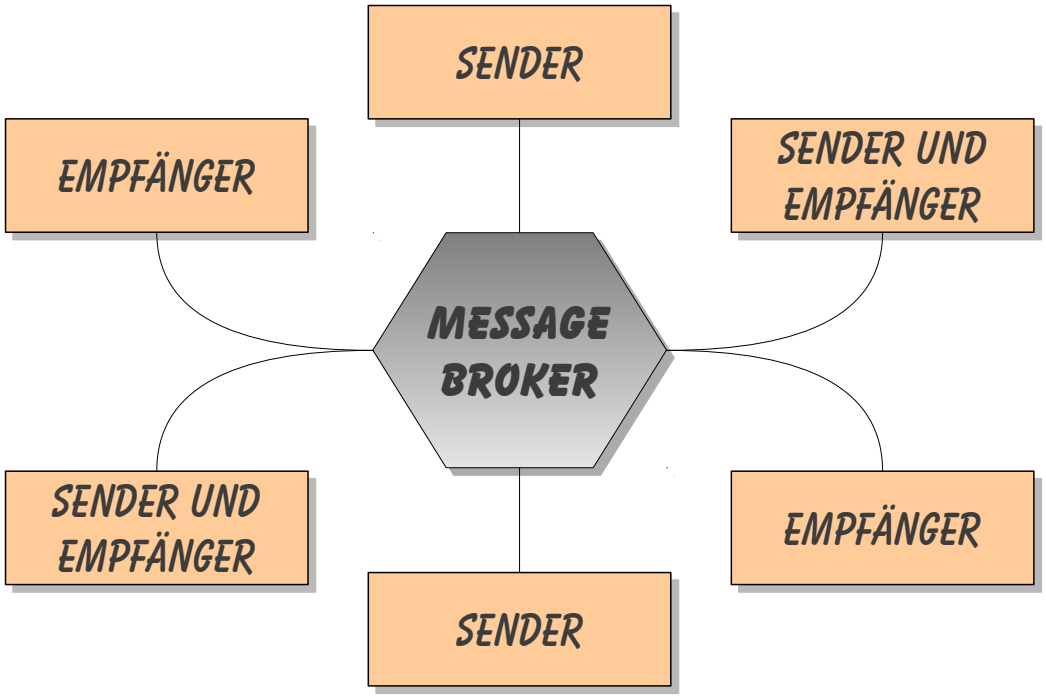
Daher auch kein Unterschied zwischen Client und Server

Jedes System schickt einfach Nachrichten an den Broker

Der Message Broker ermittelt die Empfänger jeder Nachricht und sorgt dafür, dass die Nachrichten zugestellt werden

Falls ein Empfänger nicht erreichbar ist, kann die Zustellung später wiederholt werden

Lastverteilung kann ebenfalls einfach implementiert werden



Asynchroner Nachrichtenaustausch

Asynchroner Aufruf, ähnlich wie E-Mailversand

- Sender übergibt eine Nachricht an die Middleware
- Anschließend wartet der Sender nicht auf eine Antwort
- Die Middleware ordnet die Nachricht in eine Warteschlange ein
- Empfänger rufen die Nachricht aus der Warteschlange ab
- Zwei Arten von Warteschlangen: Mit nur einem Empfänger oder mit mehreren Empfängern ähnlich einer Mailingliste

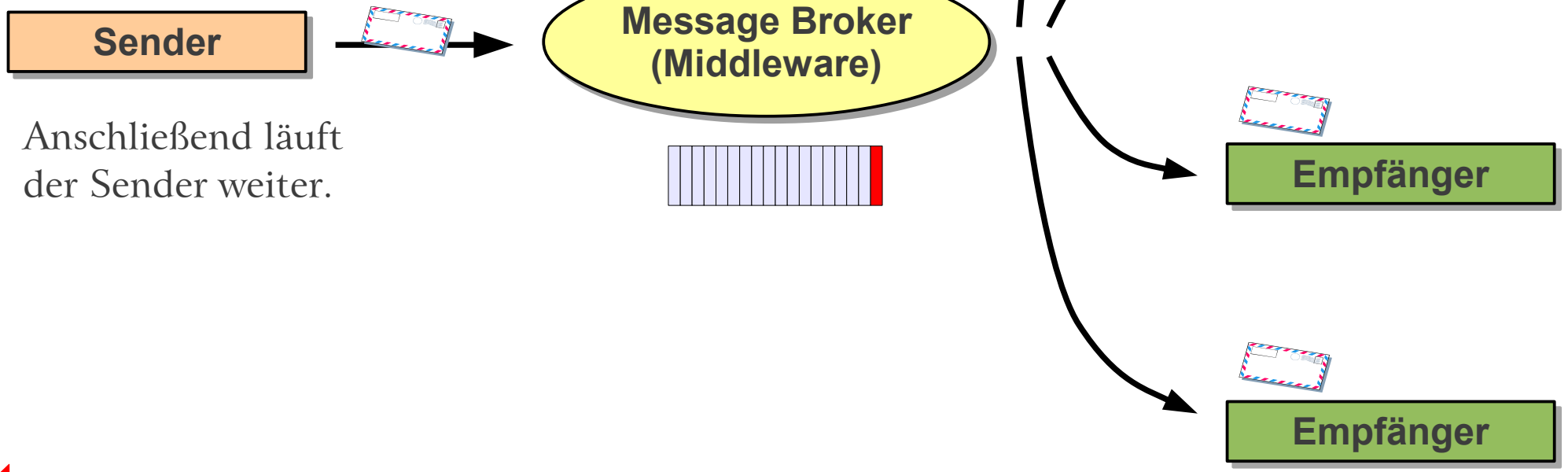
Zwei Arten der Nachrichtenzustellung

- Empfänger prüfen regelmäßig ihre Warteschlangen (Pull)
- Middleware ruft regelmäßig die Empfänger auf (Push)

Beispiel: Nachrichtenversand

Sender übermittelt eine Nachricht an die Middleware.

Anschließend läuft der Sender weiter.



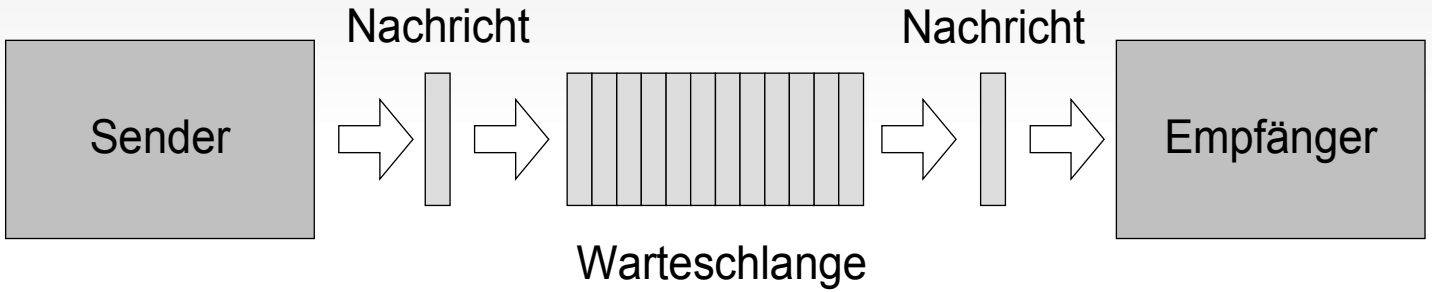
Die Middleware reiht die Nachricht in eine Warteschlange ein.

Anschließend wird die Nachricht zugestellt.

Das Point-To-Point Verfahren

Zwei Prozesse tauschen asynchron Nachrichten miteinander aus. Die gesendeten Nachrichten werden allerdings in einer Warteschlange gesammelt.

Mindestens ein Empfänger überwacht die Warteschlange und liest die darin gesammelten Nachrichten wieder aus.



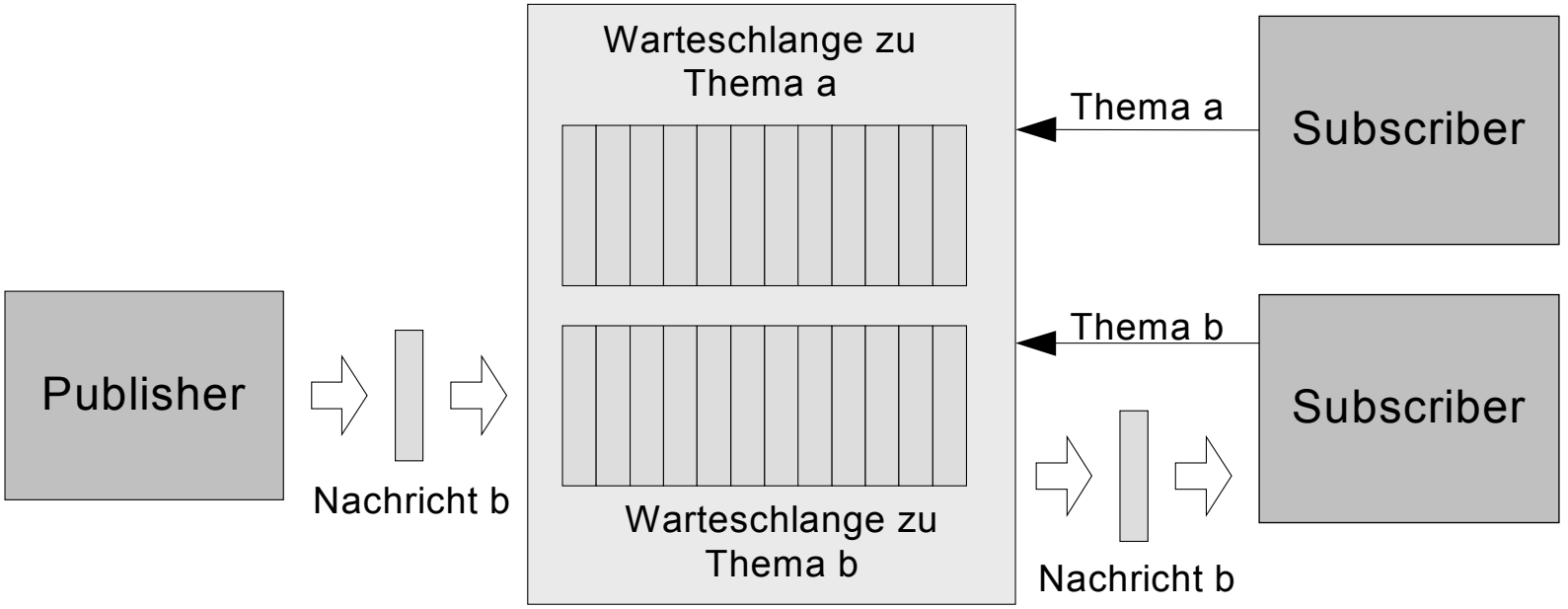
Die Verwaltung der Warteschlange erfolgt nach dem FIFO-Prinzip. Nachrichten mit höherer Priorität werden jedoch immer vor Nachrichten niedrigerer Priorität einsortiert.

Auch wenn mehrere Empfänger eine Warteschlange überwachen, kann jede Nachricht nur von einem Prozess bearbeitet werden, wodurch sich eine automatische Lastverteilung ergibt.

Das Broadcast-Verfahren

Der Message Broker verwaltet themenorientierte Warteschlangen, die von mehreren Empfängern abonniert werden können. Die Nachrichten werden dabei immer an alle registrierten Empfänger gesendet.

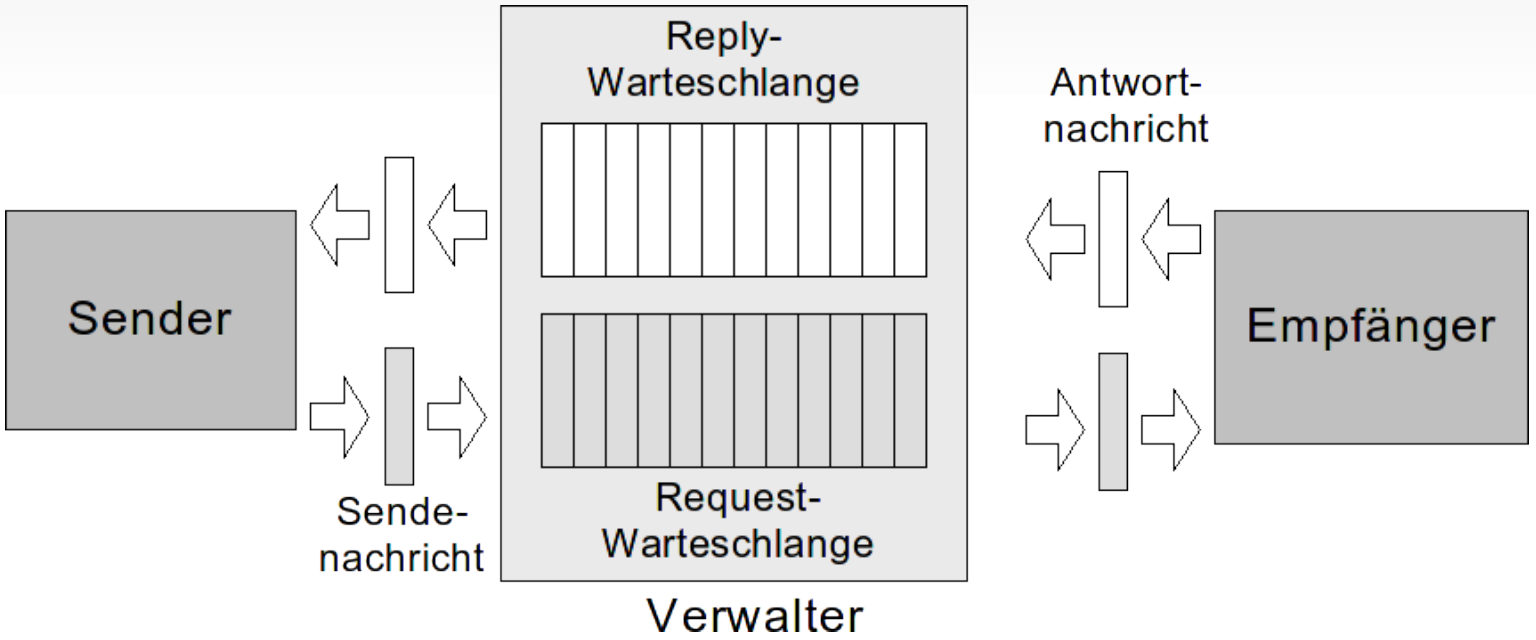
Die Zustellung der Nachrichten wird nicht garantiert, da es auch Themen ohne interessierte Empfänger geben kann. Das Verfahren ist auch bekannt als Fanout oder Publish/Subscribe-Verfahren.



Das Request/Reply-Verfahren

Das Verschicken einer Nachricht und der Antwort darauf erfolgt als eine Einheit. Die Kommunikation gilt nur dann als erfolgreich, wenn beide Nachrichten verschickt wurden.

Jede Nachricht beinhaltet daher ein spezielles Feld, aus welchem der Empfänger die Antwortwarteschlange auslesen kann.

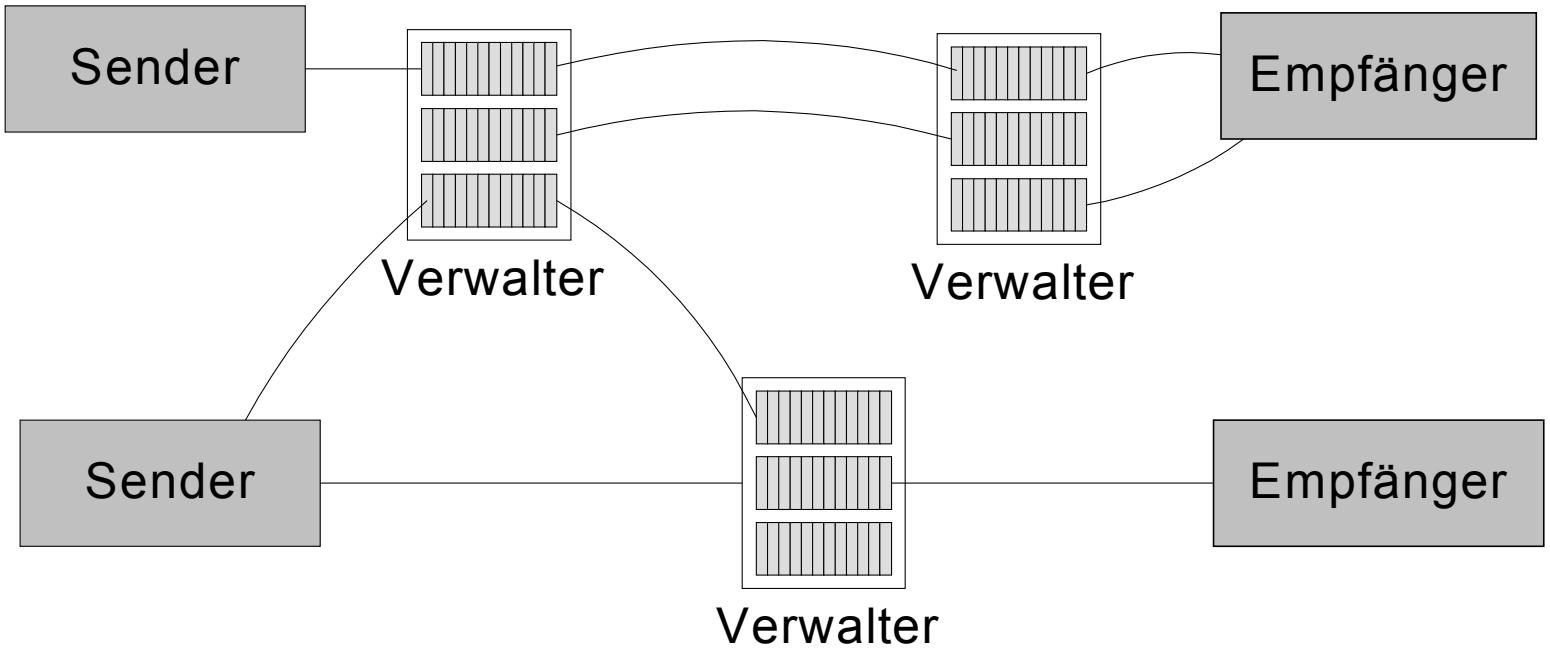


Ermöglicht eine synchrone Kommunikation über ein asynchrones Medium.

Routing von Nachrichten

In der Regel kommuniziert ein Prozess nur mit einem Message Broker. Bei großen Installationen kann es aber vorkommen, dass mehrere Message Broker (zum Beispiel in unterschiedlichen Subnetzen) verwendet werden.

In diesem Fall bilden die Message Broker eine Infrastruktur, in der sie sich gegenseitig Nachrichten weiterleiten. Die Message Broker agieren dann als Router für Nachrichten.



Nachrichtenbasierte Middleware

Message Oriented Middleware (MOM-Server)

- Verbindet mehrere Teilsysteme zu einer verteilten Anwendung
- Unterstützung der verschiedenen Kommunikationsmodelle
- Definiert das Datenformat der ausgetauschten Nachrichten
- Ermöglicht die Administration der verschiedenen Warteschlangen
- Beinhaltet Dienste für Authentifizierung oder Quality of Service

Format der ausgetauschten Nachrichten

- Datenformat von der Middleware vorgegeben
- Kopfdaten werden von der Middleware verwendet, um Zusatzdaten zur Nachricht zu speichern
- Der eigentliche Datenteil ist jedoch frei definierbar

Quality of Service

Mögliche Zustellungsgarantien

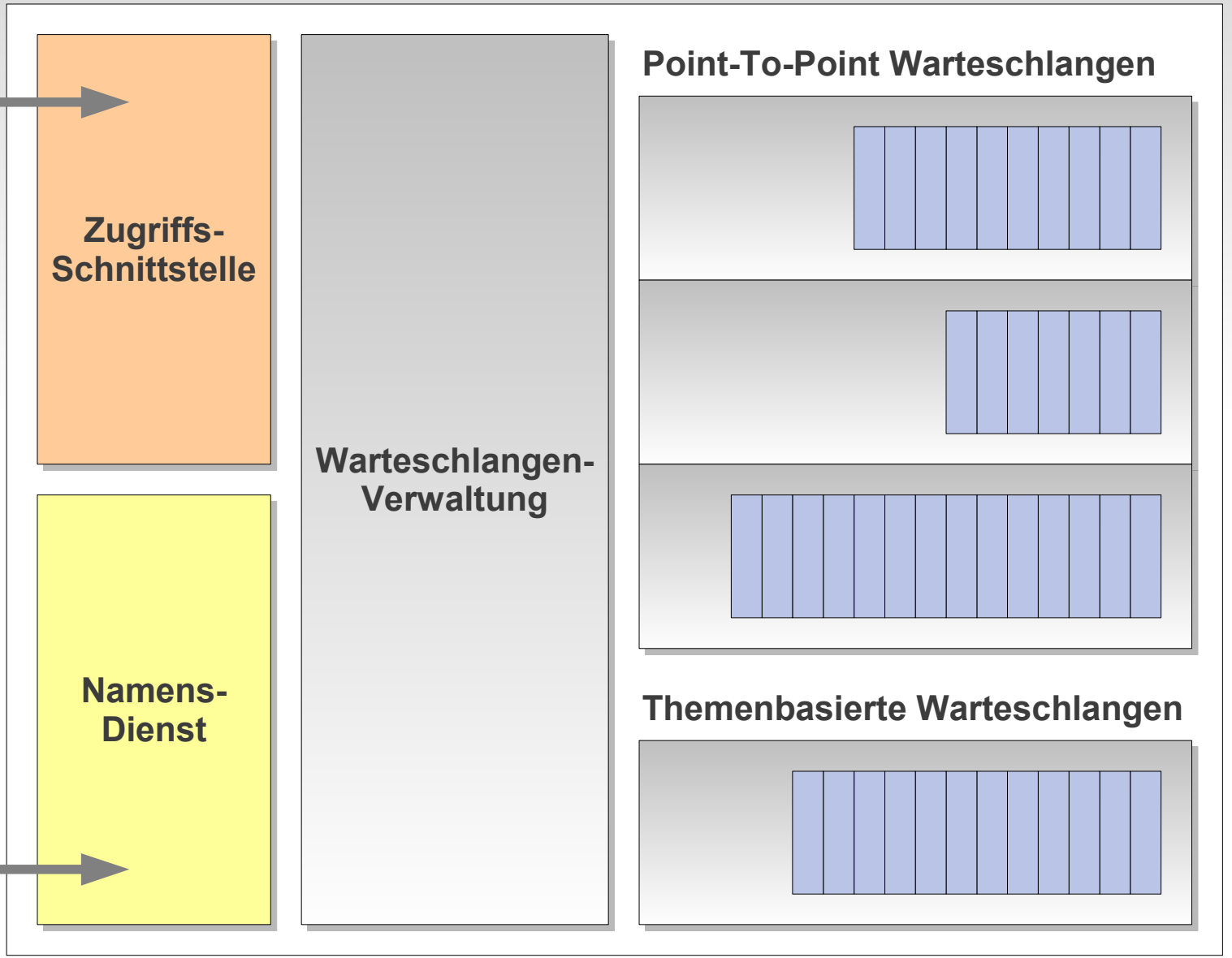
- Keine Garantie, dass eine Nachricht zugestellt wird
- Zwischenspeichern von Nachrichten im Hauptspeicher
- Persistente Warteschlangen, so dass zwischengespeicherte Nachrichten auch einen Neustart des Servers überleben

Priorität und Lebensdauer von Nachrichten

- **Priorität:** Wichtige Nachrichten werden zuerst zugestellt
- **Lebensdauer:** Nachrichten werden verworfen, wenn sie nach Ablauf einer Maximalzeit nicht zugestellt wurden

Aufbau eines MOM-Servers

Ermöglicht die Benutzung der Middleware



Beinhaltet eine Liste aller Themen und Warteschlagen

Benötigte Java-APIs

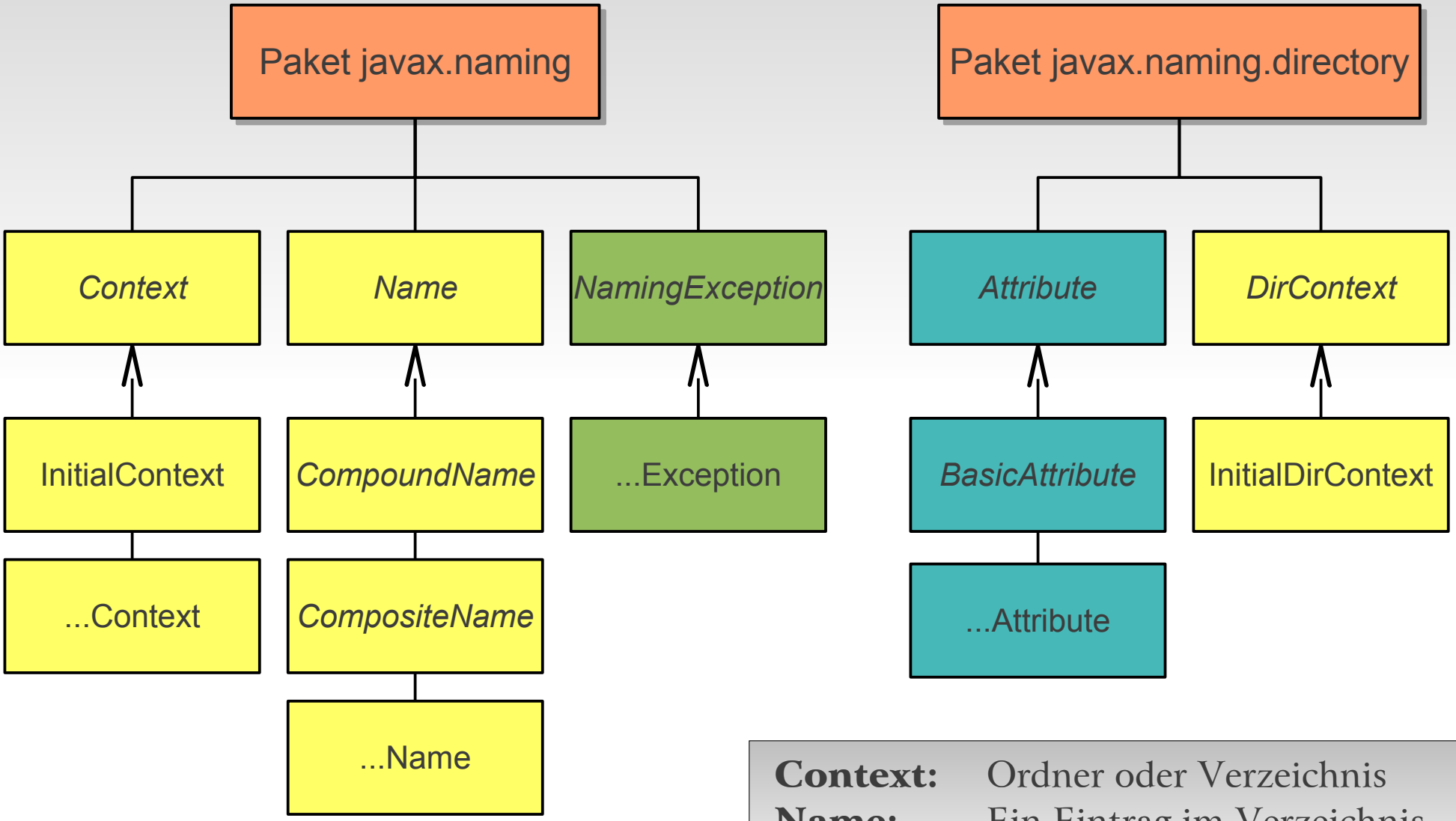
Java Message Services

- Schnittstelle zur Nutzung nachrichtenbasierter Middleware
- Unterstützt alle besprochenen Modelle des Nachrichtenaustauschs
- Benötigt eine Middleware, die als JMS-Provider dient
- Definiert nur den Zugriff und die Nutzung der Middleware, nicht aber, welche Datenformate die Middleware verwendet!

Java Naming and Directory Interface

- Schnittstelle zum Zugriff auf Namens- und Verzeichnisdienste
- Nutzung verschiedener Dienste mit identischen Methodenaufrufen
- Wird verwendet, um benötigte Objekte wie Queues zu suchen

Das Paket javax.naming



Context: Ordner oder Verzeichnis
Name: Ein Eintrag im Verzeichnis
Attribut: Metadaten zu einem Eintrag

Methoden des Interface Context

`public void bind(String name, Object obj)`
Nimmt ein neues Objekt in den Verzeichnisbaum auf und bindet es an den übergebenen Namen

`public void rebind(String name, Object obj)`
Ersetzt ein vorhandenes Objekt im Verzeichnisbaum durch ein anderes

`public void unbind(String name)`
Entfernt ein Objekt aus dem Verzeichnisbaum

`public void rename(String old, String new)`
Gibt einem existierenden Objekt einen neuen Namen

`public Context createSubcontext(String name)`
Erzeugt einen neuen Unterordner

`public Context destroySubcontext(String name)`
Löscht einen Unterordner

Methoden des Interface Context

`public NamingEnumeration<NameClassPair> list(String name)`
Liefert eine Liste aller im übergebenen Subkontext enthaltenen Objekte mitsamt ihrer Namen und Klassennamen

`public NamingEnumeration<Binding> list(String name)`
Liefert zusätzlich noch Referenzen auf die gefundenen Objekte

`public Object lookup(String name)`
Schlägt ein Objekt unter seinem Namen im Verzeichnisbaum nach

`public void close()`
Beendet die Arbeit mit dem Namens- und Verzeichnisdienst. Alle nachfolgenden Methodenaufrufe führen zu Exceptions

Aufbau von JNDI-Namen

Zusammengefasste Namen:

- Ein gültiger Name kann mehr als einen Dienst ansprechen
- **Zum Beispiel:**
  `java.sun.com/products/jndi/index.jsp`
  Beinhaltet eine URL und einen Dateisystempfad
- Solche Namen müssen von mehreren Diensten aufgelöst werden
- Dabei müssen die Regeln aller Dienste beachtet werden

Struktur und Namensbestandteile:

- Elementare Namensbestandteile
- Zusammengesetzte Namen
- Zusammengefasste Namen

Aufbau von JNDI-Namen

Atomic Name / Elementarer Name

- Elementarer Name eines Objekts innerhalb einer Ebene
- Ist nur innerhalb eines Ordners / Verzeichnisses eindeutig
- Kann in keine weiteren Bestandteile zerlegt werden

Compound Name / Zusammengesetzter Name

- Umfasst beliebig viele, elementare Namen
- Bildet den hierarchischen Namen eines Objekts
- Ist innerhalb des Namensdienstes eindeutig

Composite Name / Zusammengefasster Name

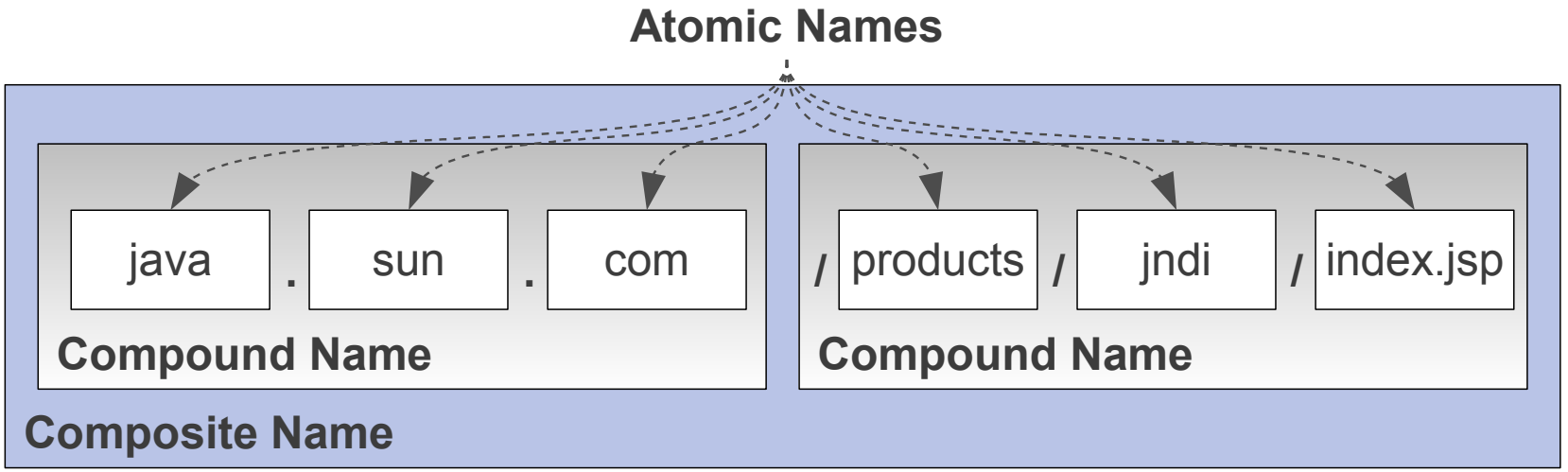
- Umfasst beliebig viele, zusammengesetzte Namen
- Beinhaltet Namen aus mehreren Namensdiensten

Beispiel: Aufbau von Namen

Elementare Namen: Nur eindeutig innerhalb einer Ebene
`java`, `sun`, `com`, `products`, `jndi`, `index.jsp`

Zusammengesetzte Namen: Eindeutig innerhalb eines Namensdienstes
`java.sun.com`, `/products/jndi/index.jsp`

Zusammengefasster Name: Umfasst mehrere Namensdienste
`java.sun.com/products/jndi/index.jsp`



Benutzung von JNDI

Vorgehen bei der Entwicklung

- Erzeugen eines Konfigurationsobjekts (Environment)
- Initialen Kontext erzeugen und Environment übergeben
- Mit dem Namens- und Verzeichnisdienst arbeiten
- Verbindung über den initialen Kontext schließen

Inhalt der JNDI-Konfiguration

- Name der Server Provider Klasse des Namensdienstes
- Hostname und Portnummer falls der Dienst nicht lokal läuft
- Sonstige Konfigurationsparameter des Server Providers

Beispiel: InitialContext erzeugen

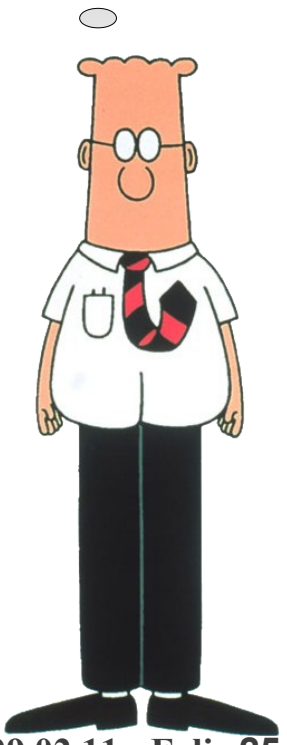
```
try {
    // Environment mit Verbindungsdaten befüllen
    Hashtable<String, String> env = new Hashtable<String, String>();

    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL,
            new File("~/testfiles").toURI().toString());

    // Initialen Kontext erzeugen
    Context ctx = new InitialContext(env);

    ////////////////////////
    // Arbeiten mit JNDI //
    ////////////////////////

    // Initialen Kontext schließen
    ctx.close();
} catch (NamingException ex) {
    ex.printStackTrace();
}
```

Wenn kein Environment erzeugt wird, werden die Werte in der Datei **jndi.properties** gesucht.

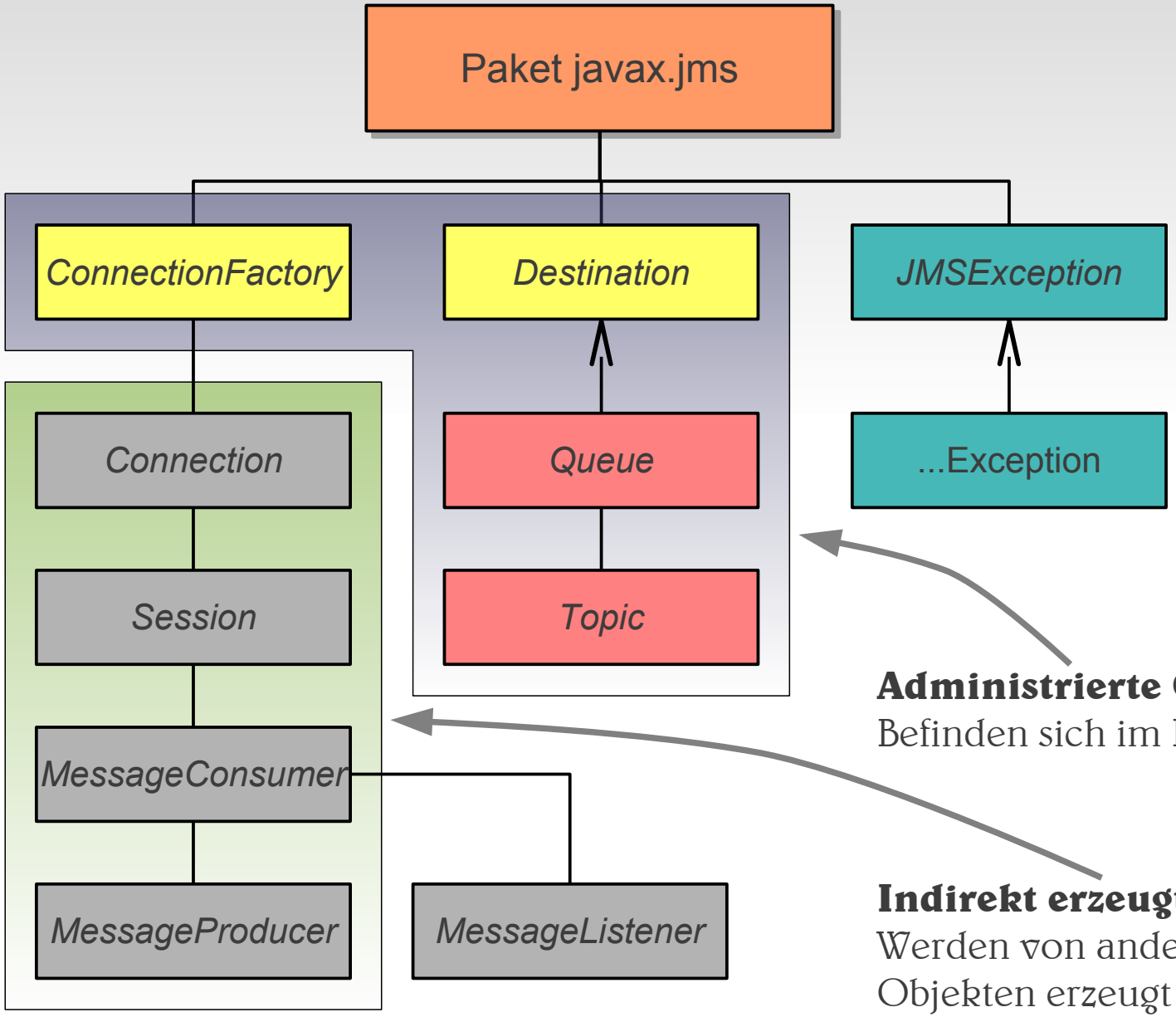
Beispiel: Arbeiten mit JNDI

```
// Auflisten aller Kindelemente
NamingEnumeration<NameClassPair> children = ctx.list(".");

for (; children.hasMoreElements(); ) {
    NameClassPair child = children.nextElement();
    System.out.println("" + child);
}

// Auflisten mit Objektreferenzen
NamingEnumeration<Binding> bindings = ctx.listBindings(".");

for (; bindings.hasMoreElements(); ) {
    Binding binding = bindings.nextElement();
    System.out.print(binding.getName() + ": ");
    System.out.println("" + binding.getObject());
}

// Binden von Objekten
ctx.bind("dlb", new Person("Dilbert", "Engineer"));
ctx.bind("wly", new Person("Wally", "Nobody actually knows"));
ctx.bind("phb", new Person("Pointy Haired Boss", "Manager"));

// Nachschlagen eines Objekts
Person dilbert = (Person) ctx.lookup("dlb");
```

NamingEnumeration unterstützt nicht das Iterator-Interface!



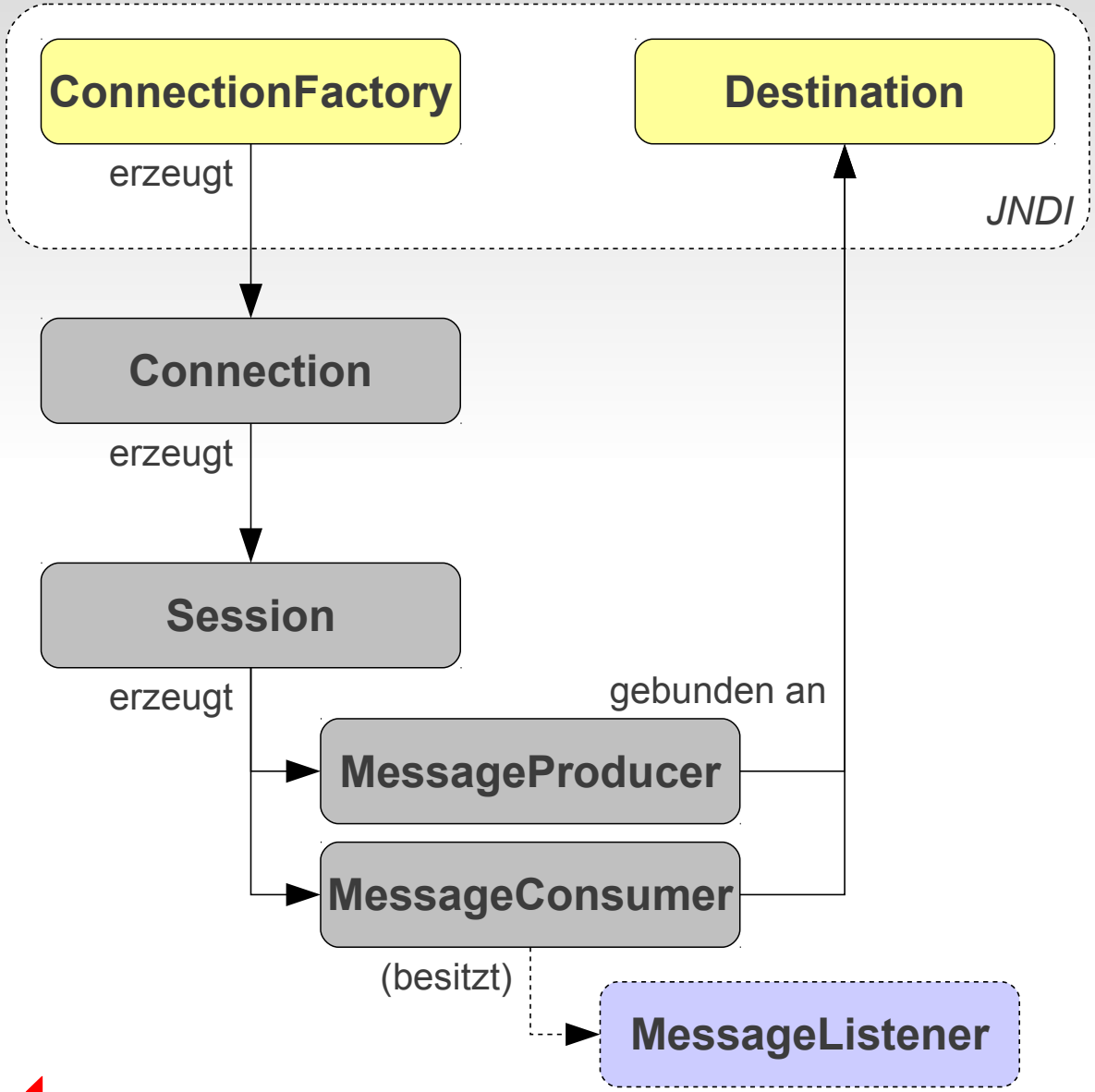
Das Paket javax.jms



Administrierte Objekte
Befinden sich im Namensdienst

Indirekt erzeugte Objekte
Werden von anderen Objekten erzeugt

Wofür so viele Klassen?



ConnectionFactory
Erzeugt ein Connection-Objekt

Connection
Stellt eine geöffnete Verbindung zur Middleware dar. Hier wird die Authentifizierung vorgenommen.

Session
Eine aktive Sitzung. Diese bestimmt die geforderten QoS-Eigenschaften und die Transaktionalität aller Vorgänge.

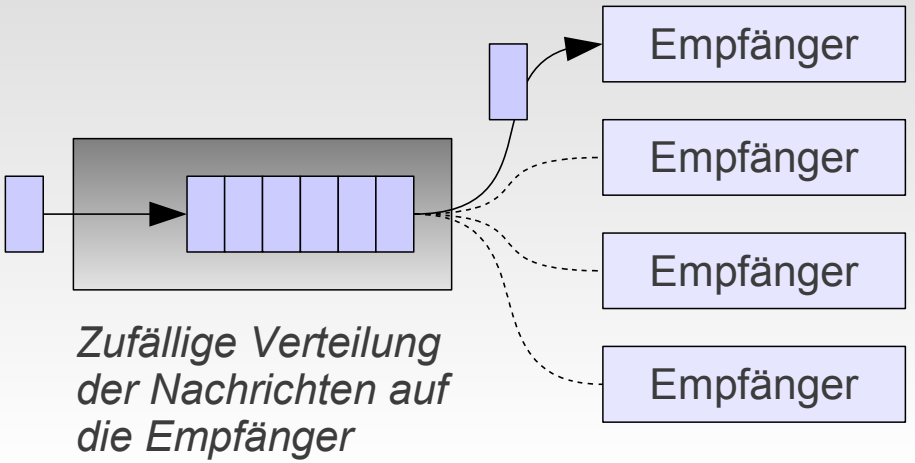
Producer, Consumer, Listener
Objekte zum Senden und Empfangen von Nachrichten

Destination
Queue oder Topic zur Verwendung mit einem Producer oder Consumer

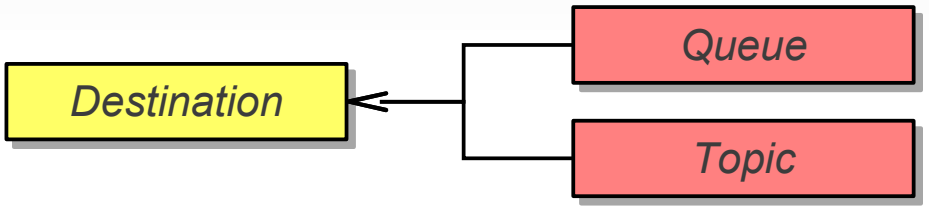
Destinations in JMS

Queues

- Point-To-Point Warteschlangen
- Nur ein Empfänger je Nachricht
- Garantierte Zustellung möglich

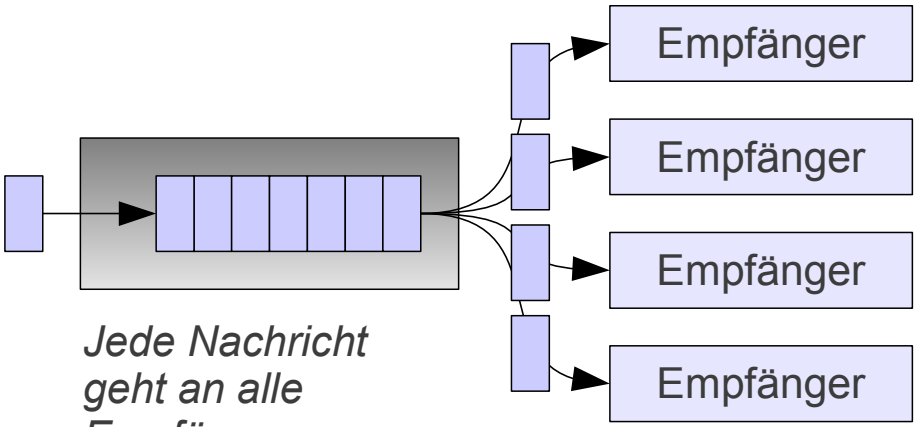


Zufällige Verteilung der Nachrichten auf die Empfänger



Topics

- Themenbasierte Warteschlangen
- Nachricht geht an alle Empfänger
- Zustellung wird nicht garantiert



Jede Nachricht geht an alle Empfänger

Nachrichtenarten in JMS

TextMessage
Ermöglicht den Versand eines beliebigen Textes. Wird häufig verwendet, um XML-Dokumente zu verschicken.

ByteMessage
Kapselt ein uninterpretiertes Byte-Array mit Binärdaten.

StreamMessage
Beinhaltet interpretierte Binärdaten im Java Byteformat.

MapMessage
Besteht aus einem assoziativen Array mit Name/Wert-Paaren. Namen sind immer Strings, die Werte beliebige Variablen oder Objekte.

ObjectMessage
Transportiert ein serialisiertes Objekt, welches daher das `Serializable` Interface implementieren muss. Setzt eine Java-Anwendung als Empfänger voraus.

Wichtige Methoden

MessageProducer

```
public void send(Message msg)
```

Versendet die übergebene Nachricht

```
public void send(Message msg, int deliveryMode,
int priority, long timeToLive)
```

Versand mit Persistenzbedingung, Priorität und Lebensdauer

MessageConsumer

```
public void setMessageListener( … )
```

Übergabe eines MessageListener, der auf neue Nachrichten reagiert

```
public Message receive()
public Message receive(long timeout)
public Message receiveNoWait()
```

Manueller Empfang einer einzelnen Nachricht ohne MessageListener

Beispiel: Nachrichten senden

```
import javax.naming.*;
import javax.jms.*;

public class SenderProgram {
    public static void main(String[] args) throws Exception {
        // Verbindung zum Message Broker herstellen
        InitialContext jndi = new InitialContext();

        ConnectionFactory connectionFactory = (ConnectionFactory)
                            jndi.lookup("/ConnectionFactory");
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);

        // Zugriff auf eine Point-To-Point Warteschlange
        Queue queue = (Queue) jndi.lookup("/queue/ExampleQueue");
        MessageProducer queueProducer = session.createProducer(queue);

        // Zugriff auf eine themenorientierte Warteschlange
        Topic topic = (Topic) jndi.lookup("/topic/ExampleTopic");
        MessageProducer topicProducer = session.createProducer(topic);
```

Beispiel: Nachrichten senden

```
        // Nachrichten verschicken
        Message m1 = session.createTextMessage("Hallo du da!");
        queueProducer.send(m1);

        Message m2 = session.createTextMessage("Hallo Leute!");
        topicProducer.send(m2);

        // Verbindung trennen
        if (jndi != null) {
            jndi.close();
        }

        if (connection != null) {
            connection.close();
        }
    }
}
```

Beispiel: Nachrichten empfangen

```
import javax.naming.*;
import javax.jms.*;

public class ReceiverProgram {
    public static void main(String[] args) throws Exception {
        // Verbindung zum Message Broker herstellen
        InitialContext jndi = new InitialContext();

        ConnectionFactory connectionFactory = (ConnectionFactory)
                          jndi.lookup("/ConnectionFactory");
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false,
                          Session.AUTO_ACKNOWLEDGE);

        // Zugriff auf eine Point-To-Point Warteschlange
        Queue queue = (Queue) jndi.lookup("/queue/ExampleQueue");
        MessageConsumer queueConsumer = session.createConsumer(queue);

        // Zugriff auf eine themenorientierte Warteschlange
        Topic topic = (Topic) jndi.lookup("/topic/ExampleTopic");
        MessageConsumer topicConsumer = session.createConsumer(topic);
```

Beispiel: Nachrichten empfangen

```
        // Nachrichten manuell empfangen
        TextMessage m1 = (TextMessage) queueConsumer.receive();
        TextMessage m2 = (TextMessage) topicConsumer.receive();

        // Verwendung eines MessageListeners
        queueConsumer.setMessageListener(new MessageListener() {
            public void onMessage(Message msg) {
                Empfang aus der Point-To-Point Warteschlange
            }
        });

        topicConsumer.setMessageListener(new MessageListener() {
            public void onMessage(Message msg) {
                Empfang aus der themenorientierten Warteschlange
            }
        });

        // Verbindung trennen
        jndi.close();
        connection.close();
    }
}
```

Enterprise Java Beans

Klassifizierung von Middleware

Kommunikationsorientierte Middleware

- Setzt direkt auf den Netzwerkdiensten des Systems auf
- Vereinheitlicht das Datenformat verschiedener Systeme
- Stellt eine einheitliche Kommunikationsinfrastruktur bereit
- Bietet verschiedene Paradigmen zur Programmierung an

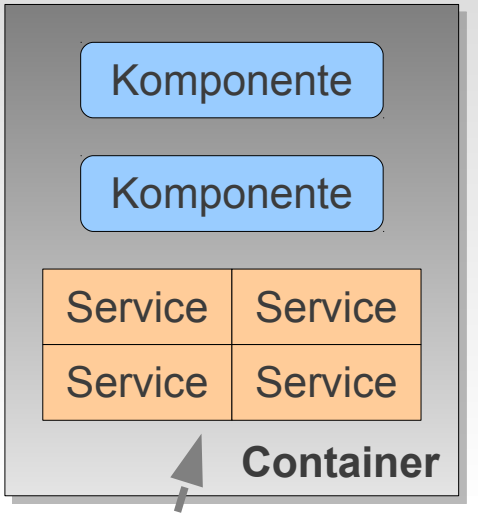
Anwendungsorientierte Middleware

- Erweitert die kommunikationsorientierte Middleware
- Beinhaltet eine **Laufzeitumgebung** für echte **Komponenten**
- Stellt der Anwendung verschiedene Zusatzdienste bereit

Container und Komponenten

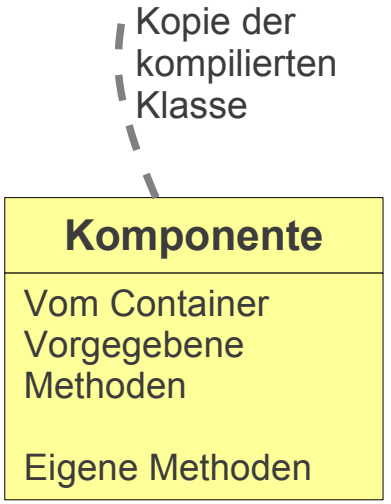
Laufzeitumgebung / Container

- Ist ein fertiges Programm ohne Funktion
- Wird durch Softwarekomponenten erweitert
- Erzeugt/Zerstört die Komponentenobjekte
- Bietet den Komponenten verschiedene Services



Softwarekomponenten

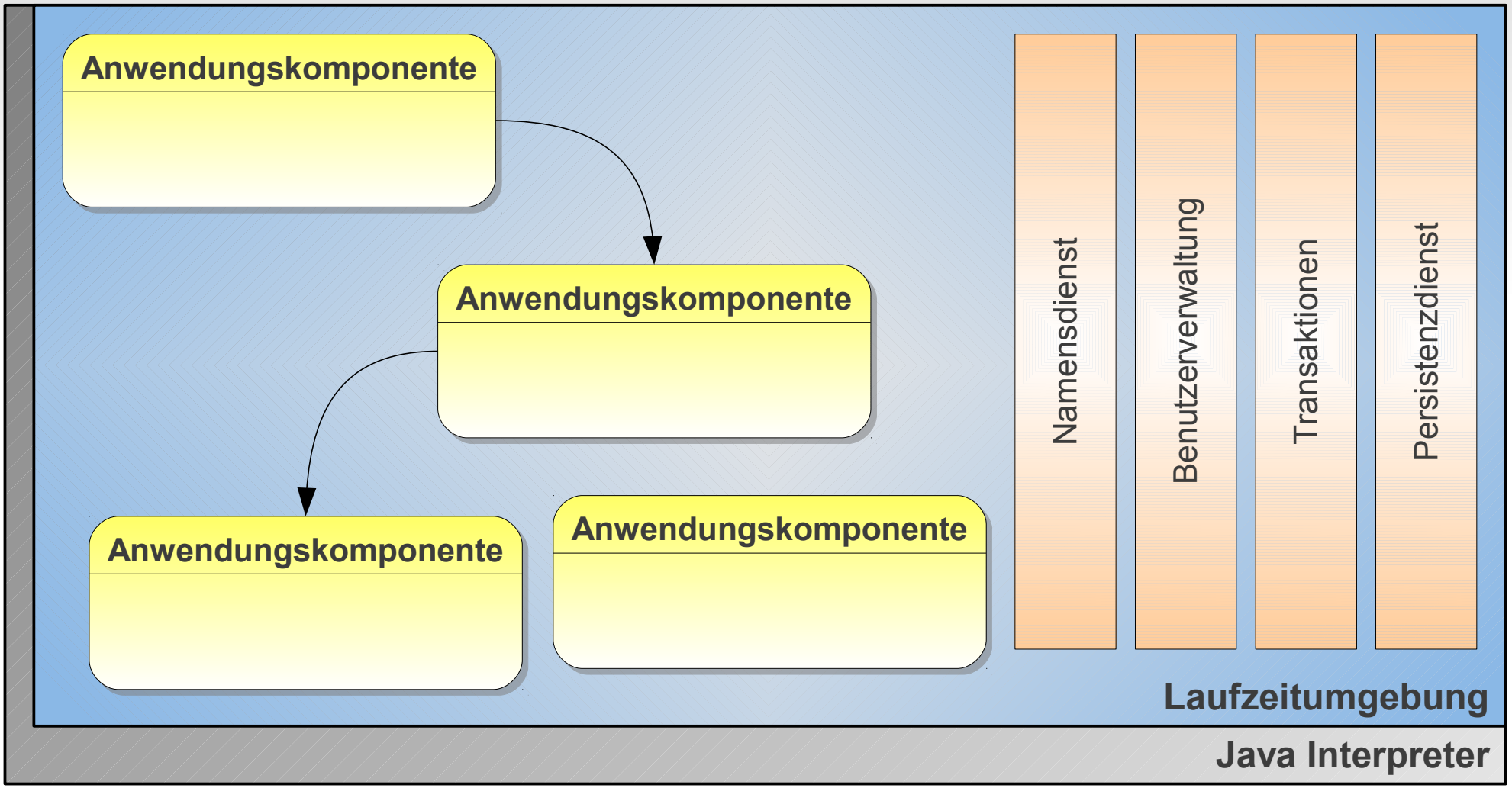
- Kapseln einzelne Funktionen einer Anwendung
- Werden als gewöhnliche Klassen programmiert
- Methoden vom Container vorgeschrieben
- Aufruf der Methoden ebenfalls vom Container



Beispiel: Laufzeitumgebung

Anwender startet die Laufzeitumgebung

```
dennis@localhost:~$ jboss/bin/run.sh
```



Java Enterprise Edition

Offener Standard für eine javabasierte Middlewareplattform

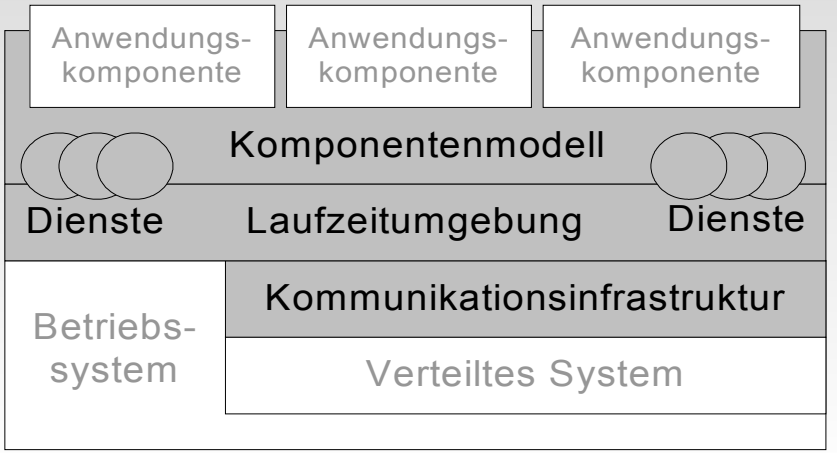
Definiert eine Anwendungsorientierte Middleware basierend auf Containern

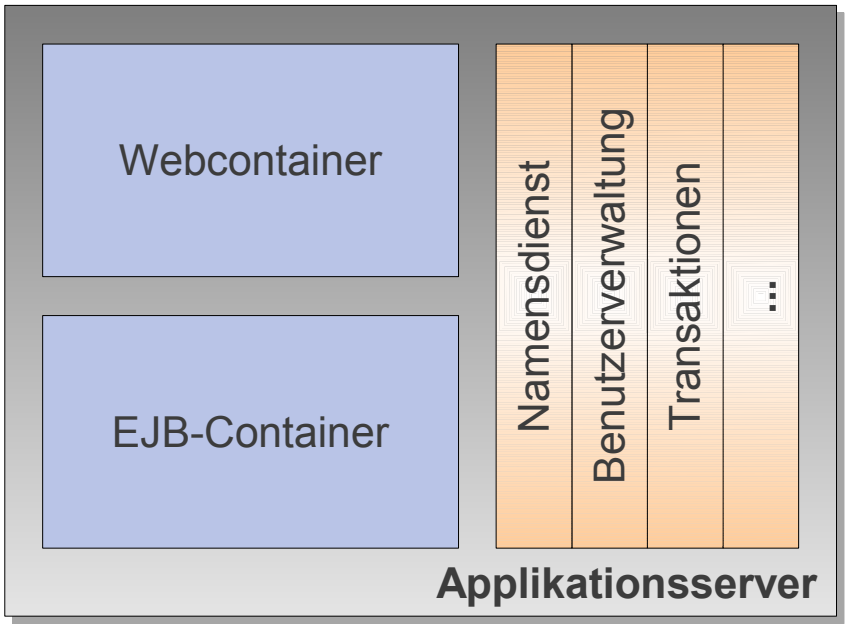
Unterstützt alle bisher kennengelernten Programmiermodelle

- Entfernte Methodenaufrufe
- SOAP-Webservices
- RESTful Webservices
- Java Message Services

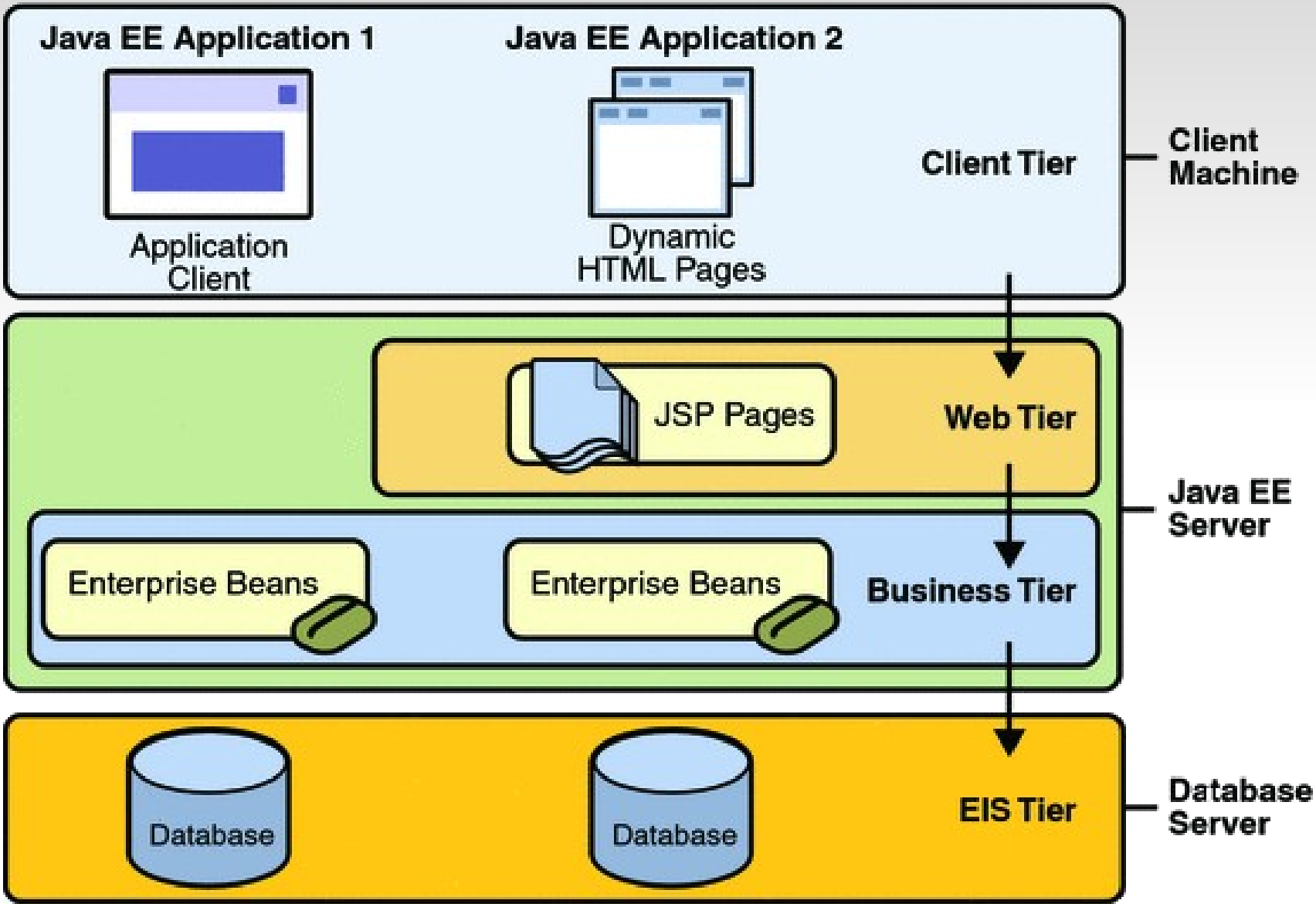
Aktuelle Version: Java EE 6

Viele Implementierungen vorhanden

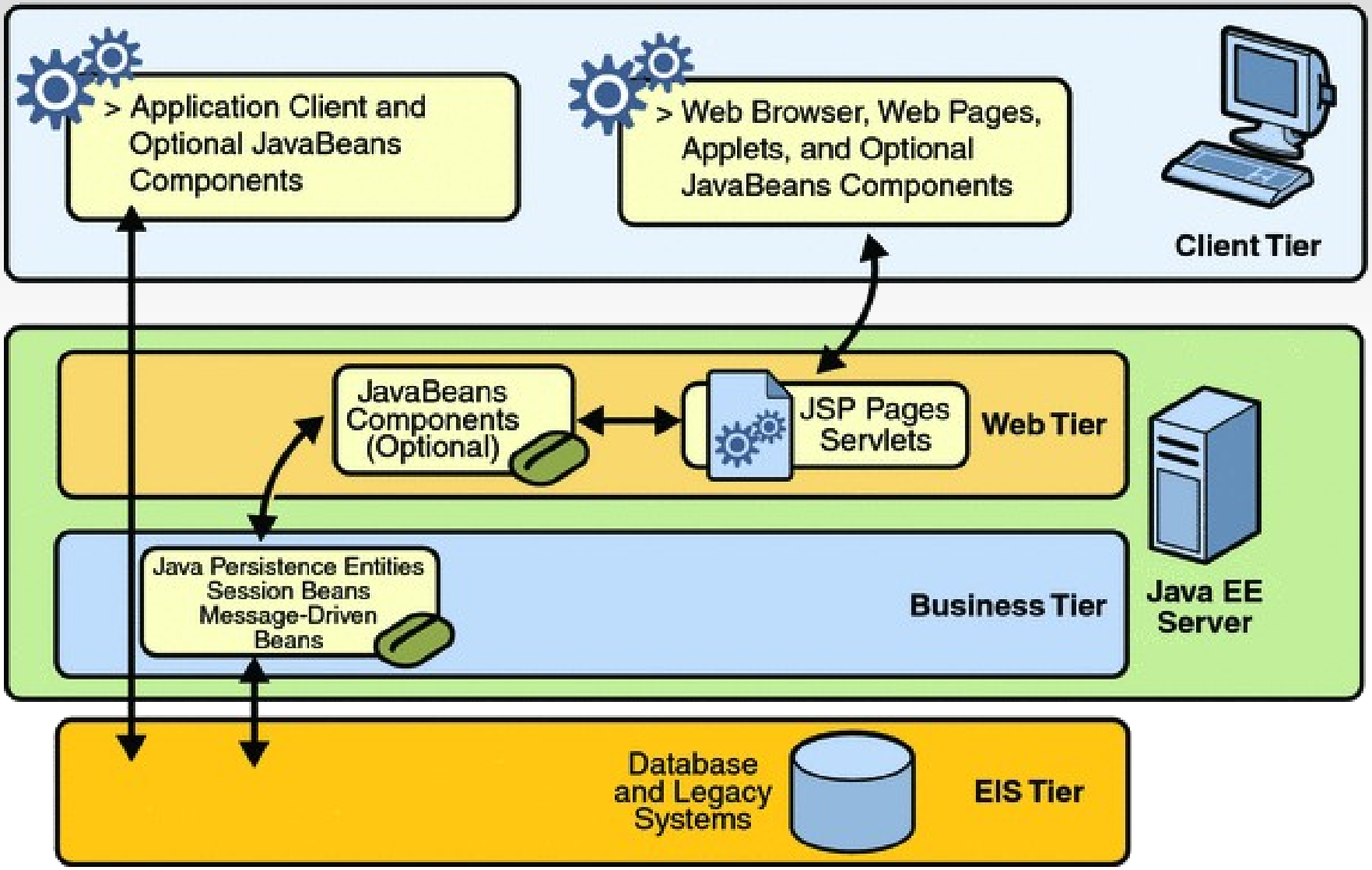




Container in Java Enterprise



Zusammenarbeit der Container



Bestandteile von Java EE

Webprogrammierung
Servlets
Java Server Pages
Java Server Faces

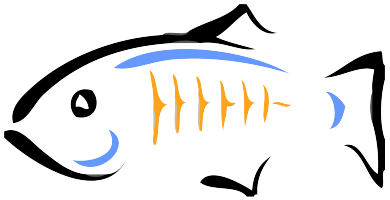
Entfernter Prozeduraufruf
JAX-RPC
JAX-WS

Entfernte Methodenaufrufe
Enterprise Java Beans

Nachrichtenaustausch
Java Messaging Services

Persistenzdienst
Java Database Connection
Java Persistence API
Java Transaction API

Verzeichnisdienst
Java Naming and Directory Interface



Serverseite einer Anwendung

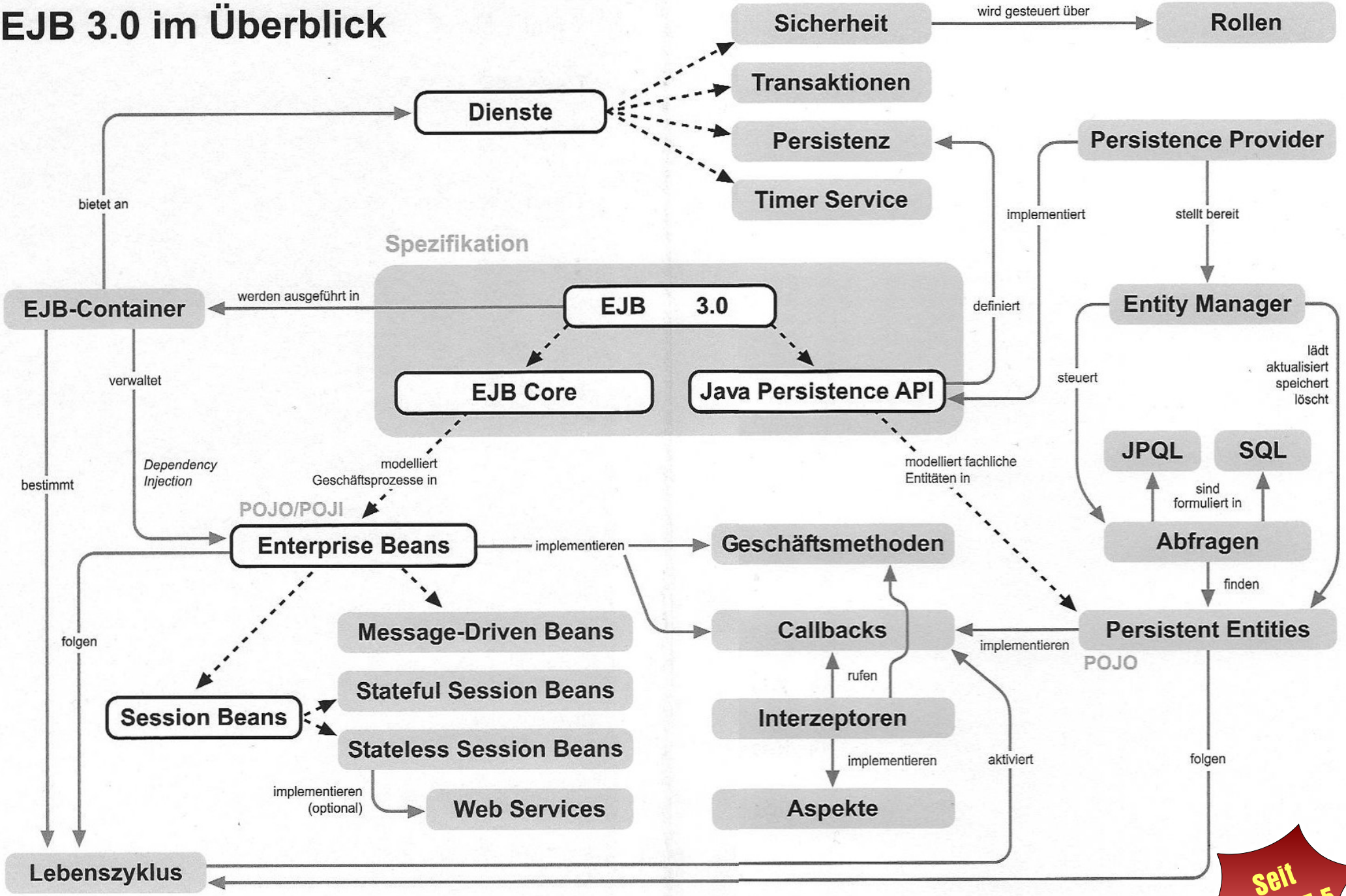
Enterprise Java Beans

- Softwarekomponenten für entfernte Methodenaufrufe
- Methodenaufruf erfolgt lokal oder entfernt über RMI
- Aufruf auch als Webservice oder mit JMS möglich
- Implementieren die Fachlogik einer Java EE-Anwendung

Persistence Entities

- Objekte, welche die Datensätze einer Tabelle kapseln
- Werden automatisch in Datenbankeinträge konvertiert
- Ein Objekt entspricht dabei immer einer Tabellenzeile

EJB 3.0 im Überblick



Enterprise Java Bean-Typen

Stateful Session Beans

- Sind für die Dauer einer Sitzung an einen Client gebunden
- Attributwerte bleiben zwischen den Aufrufen erhalten

Stateless Session Beans

- Werden von mehreren Clients gleichzeitig benutzt
- Daher keine feste Zuordnung zu nur einem Client möglich
- Kompatibel zu JAX-WS: Können Webservices implementieren

Message Driven Beans

- Bilden die Schnittstelle zu einem JMS-Provider
- Sind `MessageListener` für eine Queue oder Topic

Struktur einer EJB

Struktur einer Enterprise Java Bean

- **Business Interface:** Definiert die entfernt aufrufbaren Methoden
- **Local Interface:** Definiert die lokal aufrufbaren Methoden
- Beanklasse, welche die beiden Interfaces implementiert

Vorgehen bei der Entwicklung

- Business Interface definieren, falls benötigt
- Local Interface definieren, falls benötigt
- Beanklasse der Implementierung ausprogrammieren
- Deployment Descriptor schreiben, falls benötigt
- Anwendung verpacken und auf den Server kopieren

Business und Local Interface

Business Interface / Remote Interface

- Definiert die entfernt aufrufbaren Methoden einer EJB
- Nur benötigt, wenn die EJB entfernt aufrufbar sein soll
- Ist ein normales Interface, ohne besondere Merkmale
- Interface muss nur mit `@Remote` ausgezeichnet werden

Local Interface

- Definiert die übrigen, nicht entfernten Methoden einer EJB
- Wird ab Java Enterprise 6 nicht mehr benötigt
- Ist ebenfalls ein normales Interface, ohne Besonderheiten
- Interface muss mit `@Local` ausgezeichnet werden

Ein Interface kann nicht Remote und Local zugleich sein

Beispiel: EJB mit Local Interface

```java
import javax.ejb.Remote;

@Remote
public interface MisterRemote {
    String getFirstName();
    String getLastName();
    int setAge();
}
```

```java
import javax.ejb.Local;

@Local
public interface MisterLocal {
    void setFirstName(String name);
    void setLastName(String name);
    void setAge(int age);
}
```

```java
import javax.ejb.Stateless;

@Stateless
public class MisterBean implements MisterLocal, MisterRemote {
    public void setFirstName(String name) { ... }
    public void setLastName(String name) { ... }
    public void setAge(int age) { ... }

    public String getFirstName() { ... }
    public String getLastName() { ... }
    public int getAge() { ... }
}
```

Beispiel: Local No Interface-View

```java
import javax.ejb.Remote;

@Remote
public interface MisterRemote {
    String getFirstName();
    String getLastName();
    int setAge();
}
```

Kein Local Interface nötig!

```java
import javax.ejb.Stateless;

@Stateless
public class MisterBean implements MisterRemote {
    public void setFirstName(String name) { ... }
    public void setLastName(String name) { ... }
    public void setAge(int age) { ... }

    public String getFirstName() { ... }
    public String getLastName() { ... }
    public int getAge() { ... }
}
```

EJB-Annotationen

@Local

Kennzeichnet das Local Interface einer EJB. Kann aber auch eine EJB markieren, um das Local Interface der EJB zu benennen

@Remote

Kennzeichnet das Business Interface einer EJB. Kann auch die Klasse auszeichnen

@Asynchronous

Markiert Methoden, die im Hintergrund ausgeführt werden

@Stateless

Kennzeichnet die Implementierung einer EJB als Stateless Session Bean

@Stateful

Kennzeichnet die Implementierung einer EJB als Stateful Session Bean

@MessageDriven

Kennzeichnet die Implementierung einer EJB als Message Driven Bean

Beispiel: Message Driven Bean

```
import javax.ejb.*;
import javax.jms.*;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(
        propertyName = "destination",
        propertyValue = "queue/mailOrders"),
})
public class MessageDrivenBean implements MessageListener {
    @Resource
    private MessageDrivenContext mdc; // Dependency Injection

    public void onMessage(Message msg) {
        System.out.println("Hoppla, eine Nachricht!");
    }
}
```

Beispiel: Asynchrone Methoden

```
import javax.ejb.*;
import java.util.concurrent.*;


@Stateful
public class AsyncBean {
    @EJB private BillingBean billing;

    // Asynchrone Methode mit Rückgabewert
    @Asynchronous
    public Future<String> sendConfirmationMail() {
        return new AsyncResult<String>("Bestellbestätigung");
    }


    // Asynchrone Methode ohne Rückgabewert
    @Asynchronous
    public void archiveOrder() {
        ...
    }
```

Implementiert das Future-Interface

Rückgabewert

Beispiel: Asynchrone Methoden

```
    public void processOrder() {
        this.billing.chargeAmount(this.amount);
        Future<String> progress = this.sendConfirmationMail();

        // Prüfen, ob die Methode noch läuft
        if (!progress.isDone() && !progress.isCancelled()) {
            System.out.println("Läuft noch ...");

            // Nicht unterbrechen, wenn sie schon läuft
            // Falls sie noch nicht läuft, abbrechen (false)
            progress.cancel(false);
        }

        // Auf das Ergebnis warten
        String msg = progress.get();
    }
}
```

Lebenszyklus-Annotationen

`@PostConstruct`

Kennzeichnet eine Methode, die nach Erzeugung einer Beaninstanz und nach der Dependency Injection ausgeführt wird

`@PreDestroy`

Markiert eine Methode, die aufgerufen wird, bevor die Bean aufgeräumt wird

`@Remove`

Methoden mit dieser Annotation führen dazu, dass eine Bean zerstört wird, nachdem eine der Methoden aufgerufen wurde

`@PrePassivate`

Wird aufgerufen, kurz bevor die Bean passiviert wird, also wenn das Beanobjekt kurzfristig aus dem Speicher entfernt wird, um den Speicher für ein anderes Objekt freizumachen

`@PostActivate`

Gegenstück zu ***`@PrePassivate`****. Wird aufgerufen, nachdem eine passivierte Bean wieder in den Speicher geladen wurde*

Beispiel: Lifecycle Callbacks

```
import javax.annotations.*;
import javax.ejb.*;

@Stateful
public class LifecycleBean {
    @PostConstruct
    public void initShoppingCart() {
        System.out.println("Ich wurde soeben erzeugt.");
    }

    @PreDestroy
    public void cleanUpShoppingCart() {
        System.out.println("Jetzt ist es gleich vorbei mit mir.");
    }

    @PrePassivate // Kurzzeitiges Entfernen aus dem Speicher
    public void passivateShoppingCart() {
        System.out.println("Ich lege mich gleich schlafen.");
    }
```

Beispiel: Lifecycle Callbacks

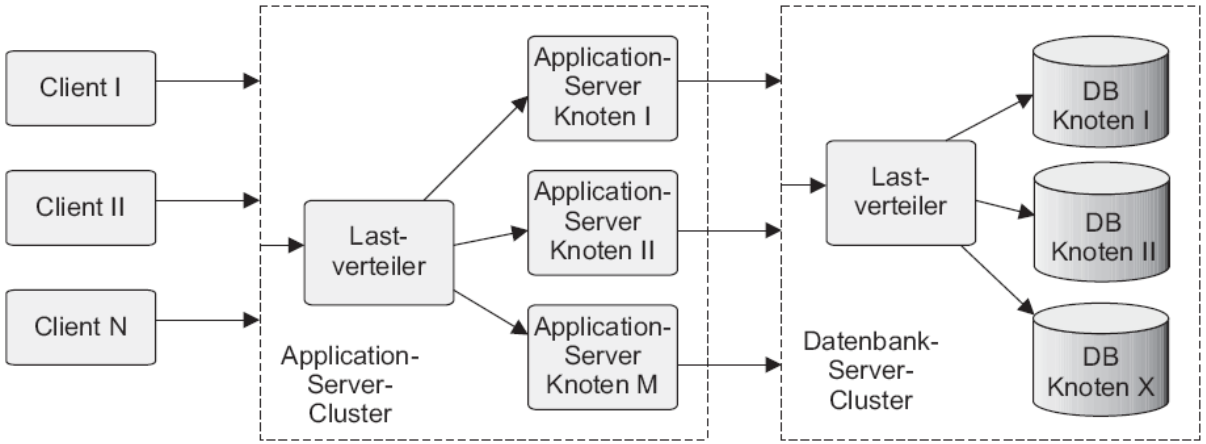
```
    @PostActivate // Wieder Einladen in den Speicher
    public void activateShoppingCart() {
        System.out.println("Jetzt bin ich wieder im Speicher.");
    }

    @Remove
    public void checkoutAndPay() {
        System.out.print("Wenn man mich aufruft, ");
        System.out.println("wird die Bean zerstört.");
    }

    @Remove
    public void abortProcess() {
        System.out.println("Es kann mehrere Remove-Methoden geben.");
    }
}
```

Einschränkungen bei der Entwicklung

Java Enterprise ist für den performanten Clusterbetrieb ausgelegt

Daher sieht der Standard folgende Einschränkungen vor:

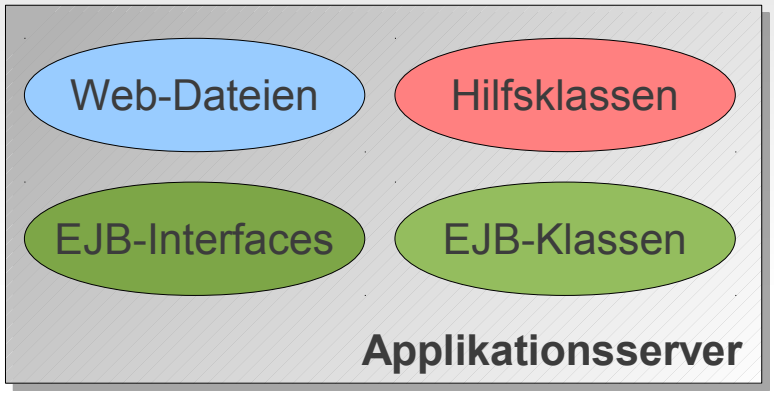
- Alle statischen Variablen müssen auch **`final`** sein
- Es dürfen keine eigenen Threads gestartet werden
- Keine Verwendung von Swing oder anderen UI-Toolkits
- Zugriff auf Dateien und Verzeichnisse nur über Servermethoden
- Es dürfen keine eigenen **`ServerSockets`** erzeugt werden
- **`System.exit()`** (würde den Applikationsserver stoppen)
- Zugriff auf native Systembibliotheken sind nicht zugelassen



Deploymentstrategien

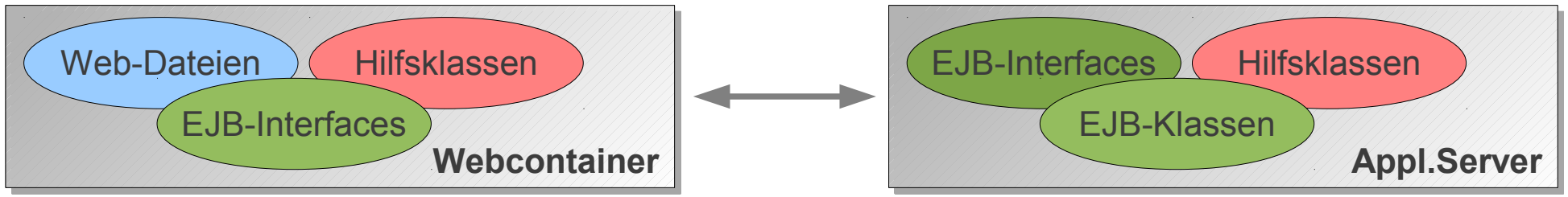
Einfaches Deployment

Alle Enterprise Java Beans, Hilfsklassen und Webdateien werden im selben Applikationsserver betrieben. Eine Aufteilung auf mehrere Server findet nicht statt.



Verteiltes Deployment

Trennung der Webanwendung von der Geschäftslogik. Die Webanwendung läuft in einem eigenständigen Webcontainer unabhängig von den Enterprise Java Beans.



Mögliche Verpackungen

JAR-Datei (Java Archive)

- Beinhaltet alle EJBs und von diesen benötigte Hilfsklassen
- Kann direkt im Applikationsserver deployed werden
- Die EJBs sind dann kein Teil einer bestimmten Anwendung

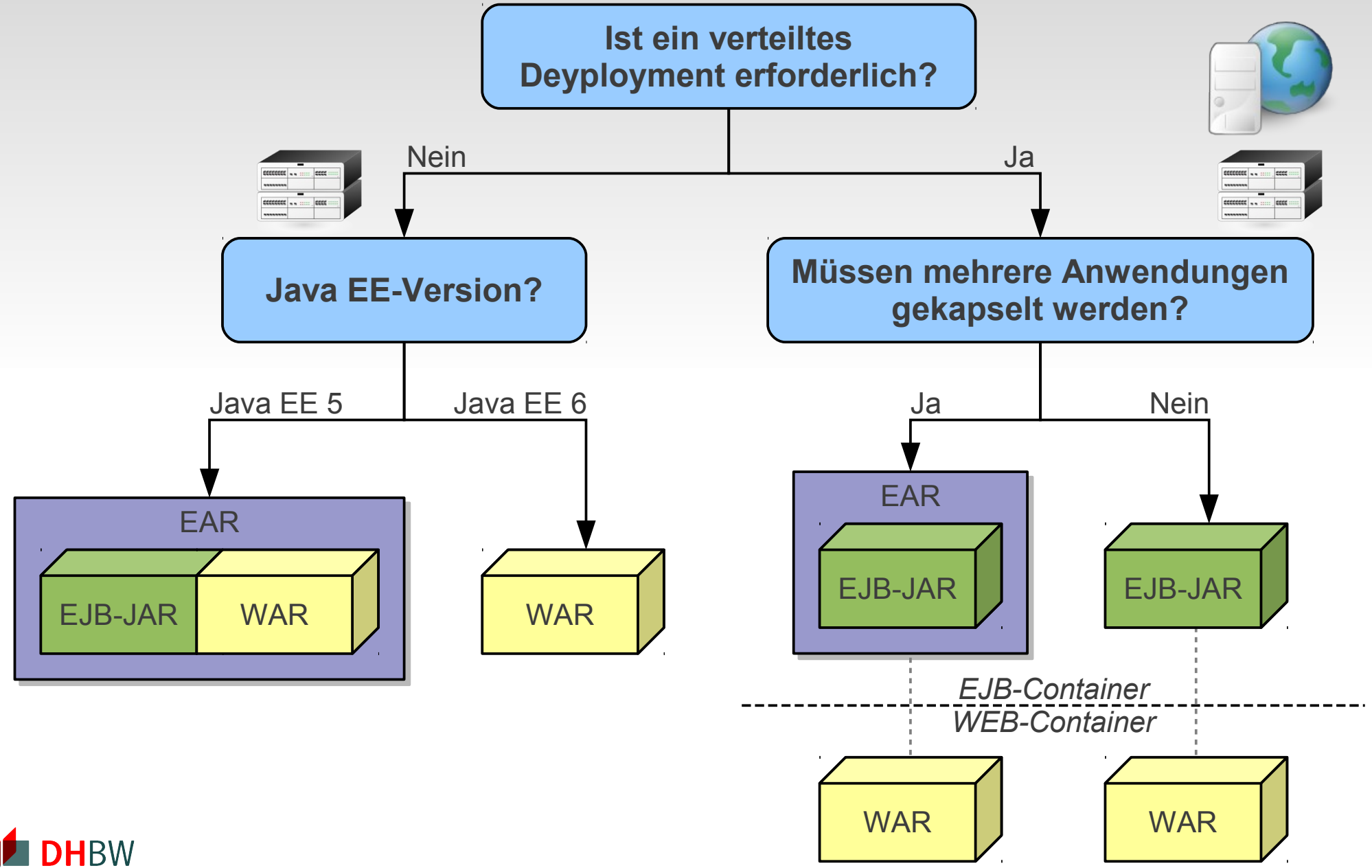
WAR-Datei (Web Archive)

- Beinhaltet Webdateien, Servlets, Java Server Pages …
- Kann seit Java EE 6 auch Enterprise Java Beans beinhalten

EAR-Datei (Enterprise Archive)

- Beinhaltet keine Klassen, sondern nur JAR- und WAR-Archive
- Fast die JARs und WARs zu einer Anwendung zusammen

Wann was verwenden?



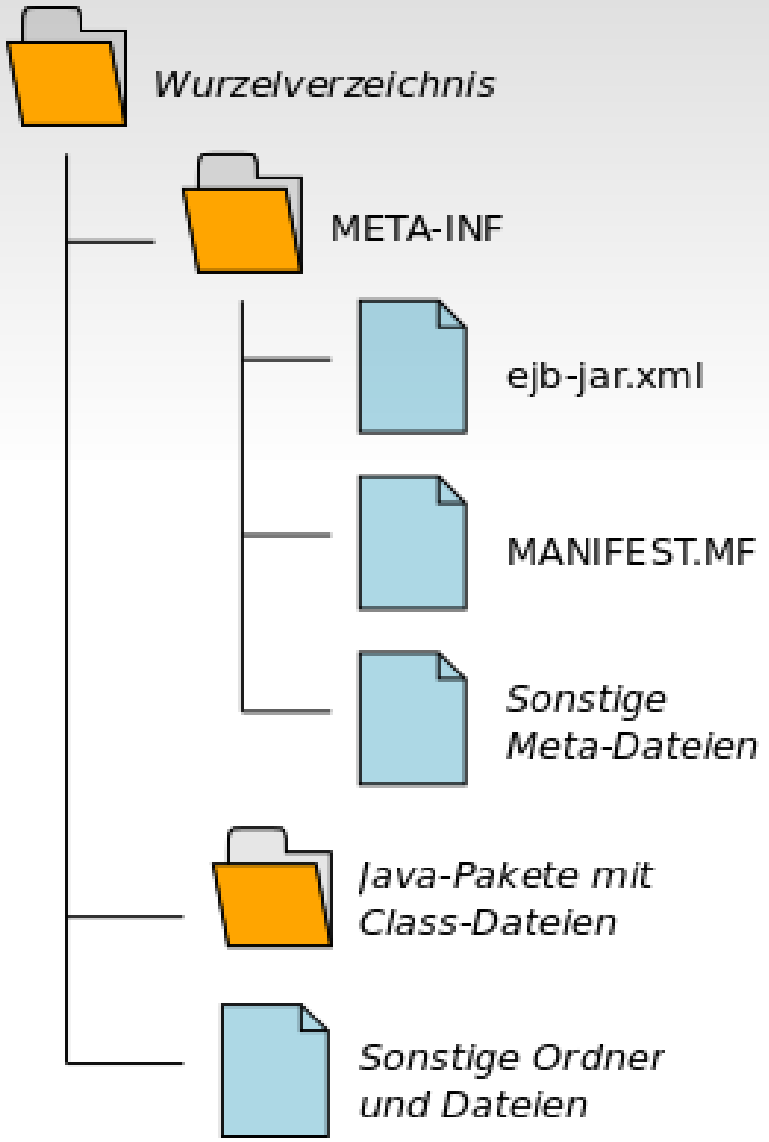
Struktur einer JAR-Datei

ZIP-Datei mit der Endung `.jar`

Alle Klassen und Pakete befinden sich im Wurzelverzeichnis

Inhalt des Ordners `META-INF`

- Eine optionale Manifestdatei, um das Archiv zu beschreiben
- Ein optionaler Deployment Descriptor, der alle Enterprise Java Beans konfiguriert, wenn diese keine Annotationen verwenden



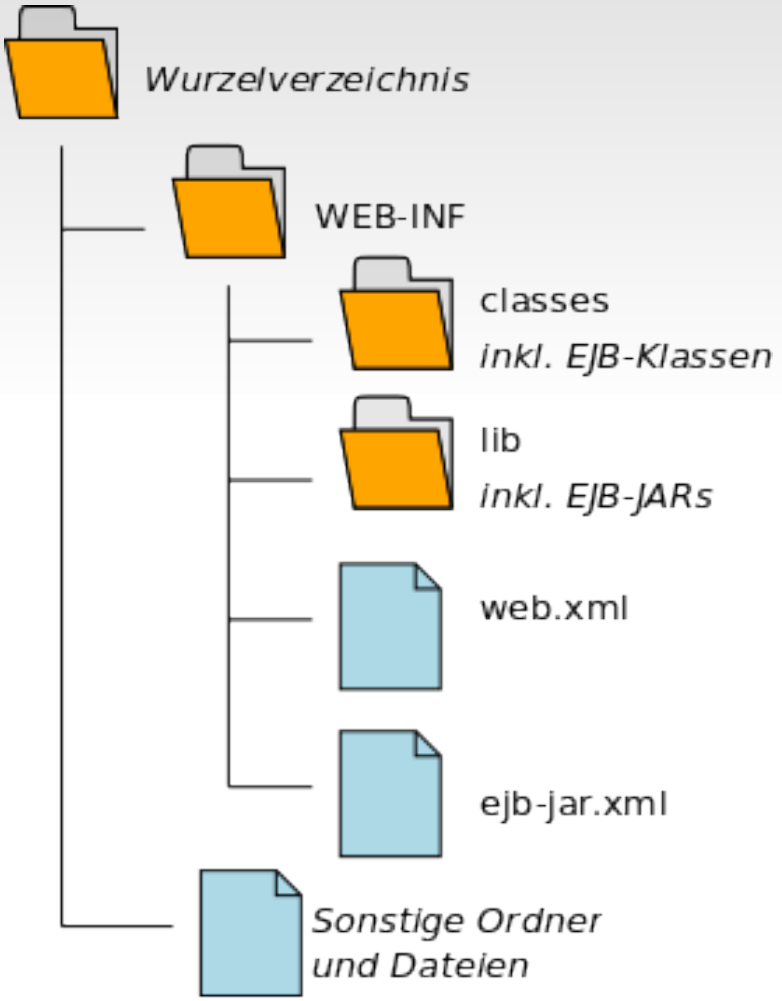
Struktur einer WAR-Datei

Seit Java EE 6 können EJBs direkt in einer Webanwendung ohne EAR-Archiv deployed werden

Die EJBs werden dann einfach zu den anderen Klassen der Anwendung kopiert

`ejb-jar.xml` auch hier optional

Die EJBs können alternativ als JAR-Dateien in `WEB-INF/lib` aufgenommen werden

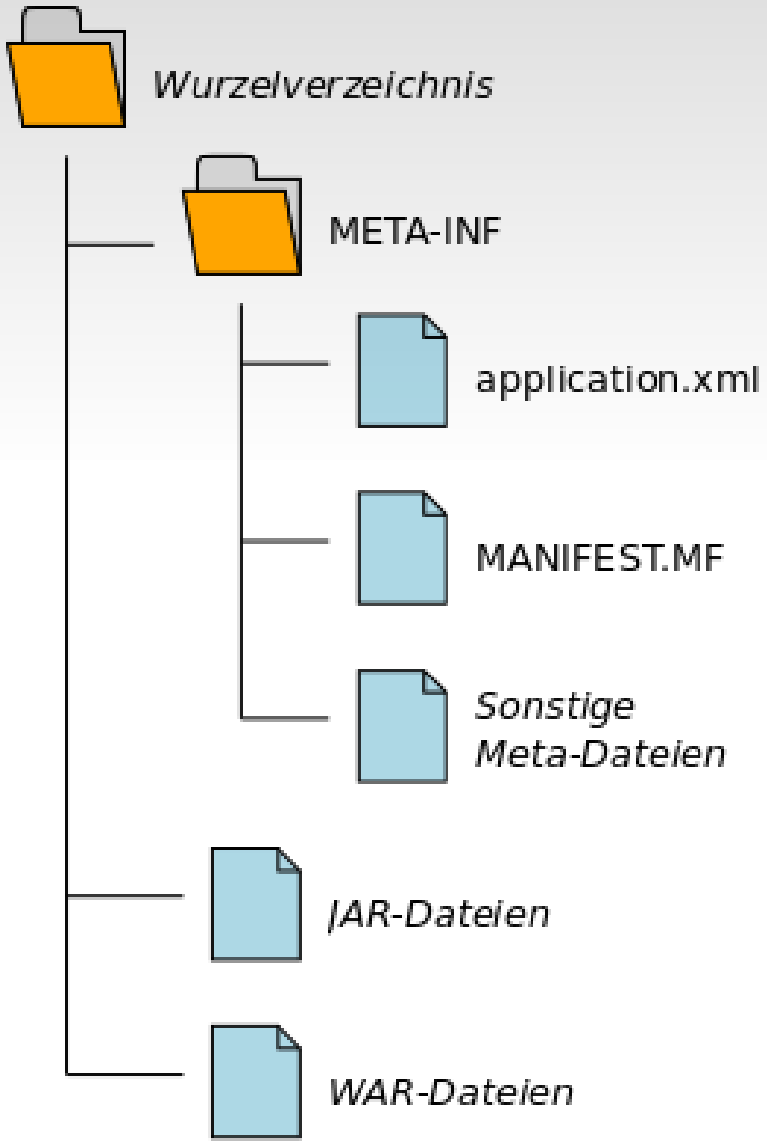


Struktur einer EAR-Datei

ZIP-Datei mit der Endung `.ear`

Fasst EJB- und WAR-Dateien zu einer Anwendung zusammen

Alle enthaltenen Archive befinden sich im Wurzelverzeichnis

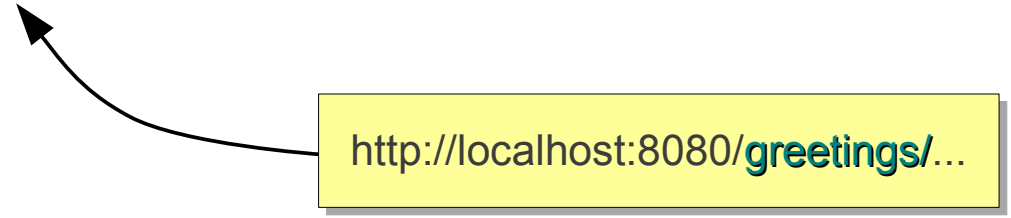


Inhalt des Ordners `META-INF`

- Optionale Manifestdatei
- Application Descriptor (Pflicht)

Beispiel: application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
    <!-- Name und Beschreibung der Anwendung -->
    <display-name>Hallo, Welt-Anwendung</display-name>
    <description>Beispiel für ein EAR-Paket</description>

    <!-- Auflistung der enthaltenen JAR-Dateien -->
    <module>
        <ejb>helloworld-ejb.jar</ejb>
    </module>
    ...

    <!-- Auflistung der enthaltenen Webanwendungen -->
    <module>
        <web>
            <web-uri>helloworld-web.war</web-uri>
            <context-root>greetings</context-root>
        </web>
    </module>
    ...
</application>
```



Zugriff auf (entfernte) EJBs

Der klassische Ansatz: Namensdienst

- EJBs werden wie alle Ressourcen mit JNDI verwaltet
- Das Objekt muss also im Namensdienst gesucht wurden
- **Namensschema:** `/EAR_Name/JAR_Name/Bean!Interface`
- **Beispiel:** `java:global/test_ear/ejb_jar/MisterBean`

Der moderne Weg: Dependency Injection

- Automatische Befüllung von Variablen, wenn sie mit der Annotation `@EJB` oder `@Resource` markiert sind
- Kein expliziter Zugriff auf den Namensdienst nötig
- Im Hintergrund findet ein automatischer JNDI-Lookup statt

Zugriff auf (entfernte) EJBs

Java SE (Ohne Applikationsserver): JNDI-Lookup

```
import javax.naming.*;

InitialContext jndi = new InitialContext();
MisterBean mb = (MisterBean) jndi.lookup("MisterBean");
...
jndi.close();
```

Java EE (Mit Applikationsserver): Dependency Injection

```
import javax.ejb.*;

public class MissesBean {
    @EJB private MisterBean misterBeanSimple;

    @EJB(beanInterface=MisterBean.class,
         lookup = "java:global/example_app/MisterBean")
    private MisterBean misterBeanComplex;
    ...
}
```

Pfad ist abhängig davon, wie das Objekt deployed wurde

Java Persistence API

Persistenz in Java Enterprise

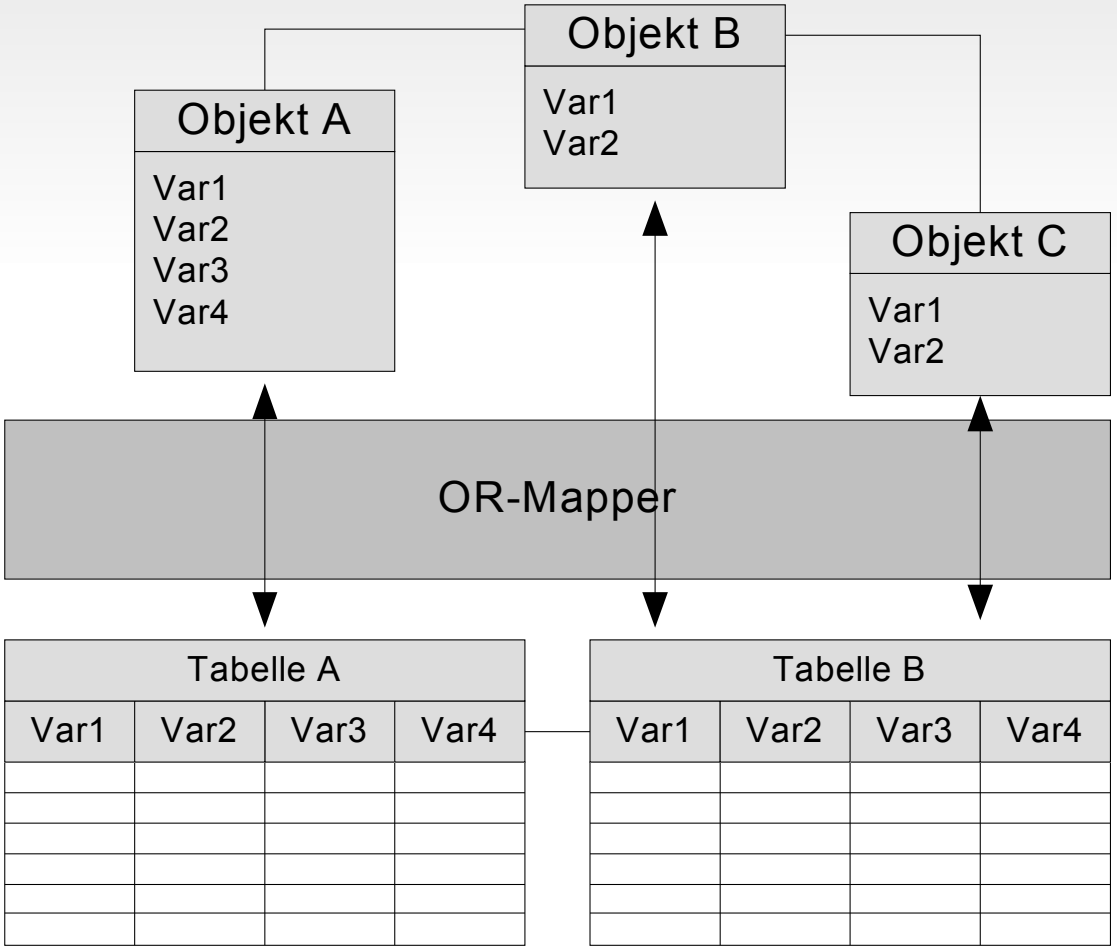
- Neue API zur Benutzung von O/R-Mappern ab Java EE 5
- Baut auf dem alten Java Database Connection-Standard auf
- Benötigt zusätzlich zu JDBC jedoch einen Persistence Provider, welcher den O/R-Mapper zur Verfügung stellt
- Läuft in jeder Javaumgebung, auch in Java SE

Aufgaben des O/R-Mappers

- Datensätze finden und in normale Javaobjekte konvertieren
- Javaobjekte in das relationale Datenbankmodell überführen
- Einheitlicher Zugriff auf unterschiedliche Datenbanken
- Datenbankobjekte (Tabellen …) anlegen und aktualisieren

Objektrelationales Mapping

Die Modellierung des Datenmodells und der Zugriff darauf erfolgen mit normalen Klassen. Ein O/R-Mapper kümmert sich um die Übersetzung zwischen Objekten und Datenbankeinträgen.



Mappingstrategien

Eine Tabelle pro Vererbungshierarchie

- Nur eine Tabelle für alle miteinander verwandten Klassen
- Verwendung eines Diskriminators (Kennzeichen) je Datensatz, der die abgelegte Klasse identifiziert

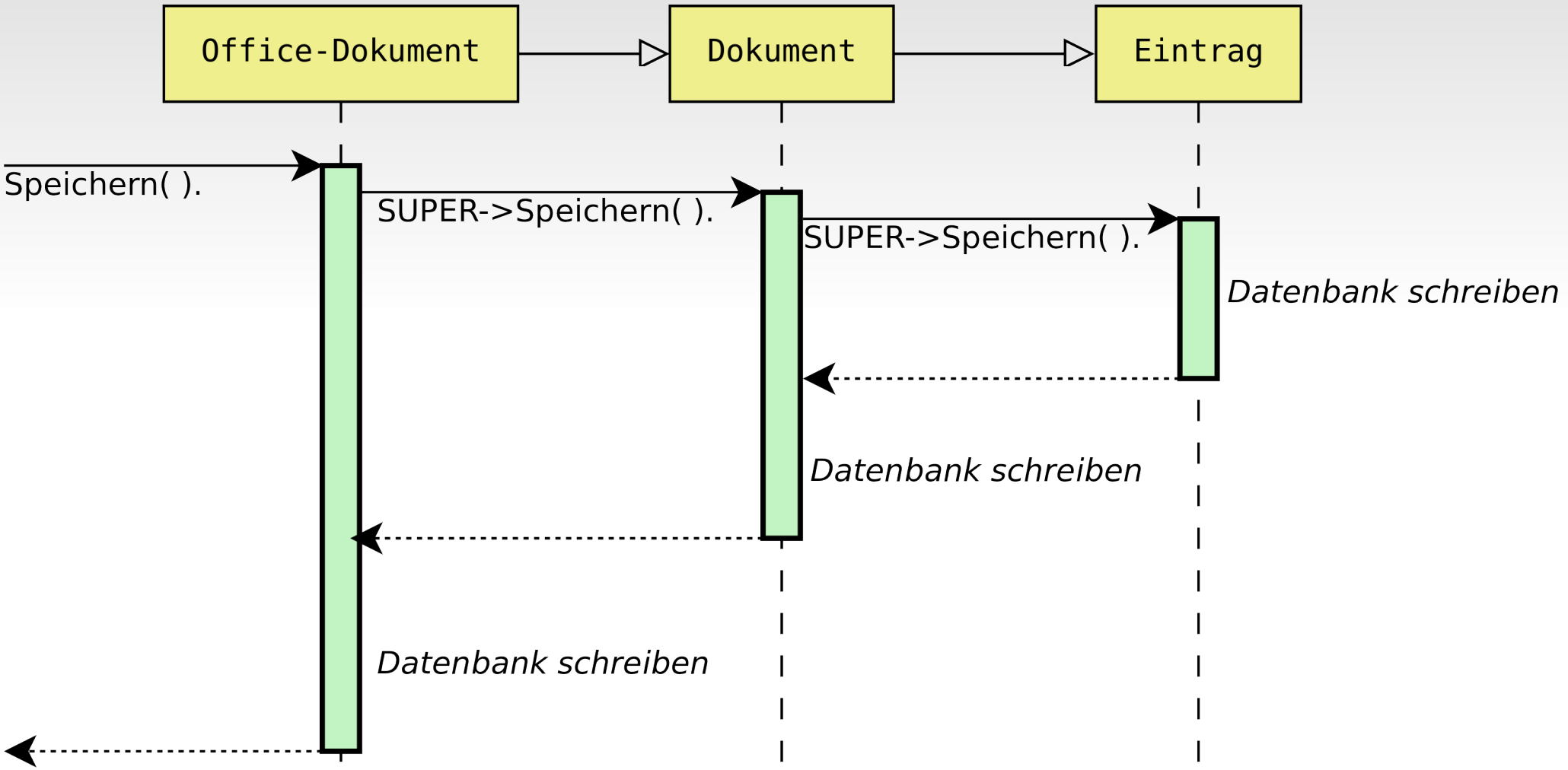
Eine Tabelle pro konkreter Klasse

- Die Vererbungshierarchie wird gänzlich ignoriert
- Jede Klasse wird in einer eigenen Tabelle abgelegt
- Die vererbten Felder finden sich in allen Tabellen wieder

Eine Tabelle pro Unterklasse

- Aufteilung von vererbten Klassen zu je einer eigenen Tabelle
- Jede Tabelle enthält nur die in einer Klasse neu definierten Felder

Beispiel: Polymorphe Datenbankoperationen



JPA-Annotationen

Annotation	Beschreibung
`@Entity`	Kennzeichnung einer Klasse als persistente Klasse
`@Inheritance`	Festlegung, wie Vererbungshierarchien in der Datenbank abgelegt werden
`@Table`	Vorgabe der Tabelle, in welcher die Objekte abgelegt werden
`@Id`	Markierung eines Attributs als eindeutiges Schlüsselattribut
`@IdClass`	Klasse für einen zusammengesetzten Schlüssel
`@GeneratedValue`	Aktivierung der automatischen Hochzählung der ID
`@Column`	Definition von Spalteneigenschaften
`@Temporal`	Kennzeichnung eines Attributs als zeitwertig
`@Transient`	Kennzeichnung eines nicht persistenten Attributs
`@NamedQuery`	Definition einer benannten Abfrage auf Objekte der Klasse
`@NamedQueries`	Definition mehrerer benannter Abfragen auf die Klasse

Beispiel: Persistente Klasse

```
import javax.persistence.*;

@Entity
@Inheritance(stragey=InheritanceStrategy.JOINED)
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String firstName;
    private String lastName;

    public void setId(int id) { this.id = id; }
    public int getId() { return this.id; }

    public void setFirstName(String firstName) { ... }
    public String getFirstName() { ... }

    public void setLastName(String lastName) { ... }
    public String getLastName() { ... }
}
```

Beispiel: Persistente Klasse

```
import javax.persistence.*;
import java.util.Date;

@Entity
@Table(name="t_person")
public class NamedPerson extends Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Temporal(TemporalType.DATE)
    private Date birthday;

    @Transient
    private int age;  // Feld wird nicht gespeichert

    @Column(name="p_name", nullable=false, unique=true, length=30)
    private String name;

    // ...
}
```

Verwendete Aufzählungen

InheritanceType: Abbildung von Vererbung

- **SINGLE_TABLE** *Eine Tabelle je Hierarchie*
- **TABLE_PER_CLASS** *Eine Tabelle je Klasse*
- **JOINED** *Mehrere Tabellen*

GenerationType: Automatische ID-Generierung

- **AUTO** *Automatische Auswahl*
- **IDENTITY** *Über eine SQL-Identität*
- **SEQUENCE** *Über eine SQL-Sequenz*
- **TABLE** *Tabelle mit aktuellem Stand*

TemporalType: Ausprägung von Zeitfeldern

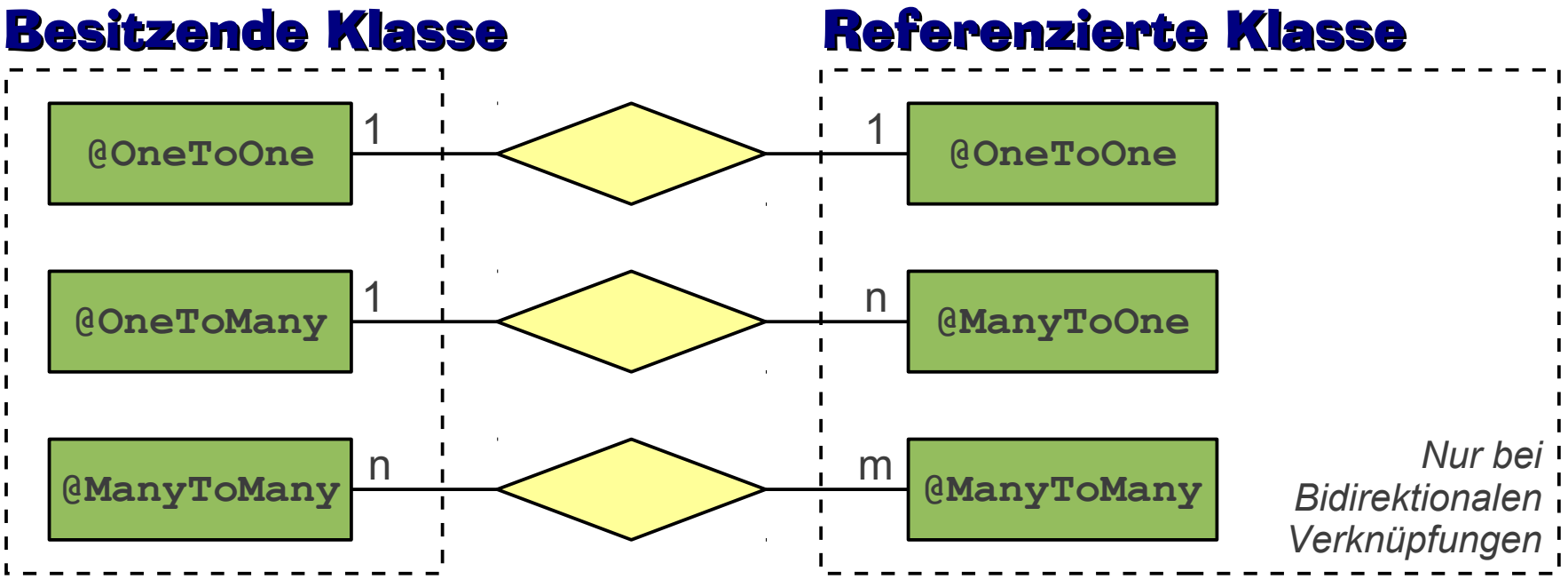
- **DATE** *Datumswert*
- **TIME** *Uhrzeitwert*
- **TIMESTAMP** *Datum und Uhrzeit*

Verknüpfungen

Abbildung von relationalen Verknüpfungen, indem ein persistentes Objekt als Attribut eines anderen Objekts verwendet wird

Dekoration des Attributs mit speziellen Annotationen nötig

Unidirektionale Verknüpfungen: Nur in eine Richtung
Bidirektionale Verknüpfungen: In beide Richtungen



Beispiel: 1:1-Verknüpfung

```
import javax.persistence.*;

@Entity
public class Person {
    @Id @GeneratedValue private int id;

    @OneToOne(optional=false)
    @JoinColumn(name="ADDRESS_ID") // Wie @Column, optional
    private Address address;
}
```

```
import javax.persistence.*;

@Entity
public class Address {
    @Id @GeneratedValue private int id;

    @OneToOne(optional=false, mappedBy="address")
    private Person person; // Optional
}
```

Beispiel: 1:1-Verknüpfung

```
import javax.persistence.*;

@Entity
public class Person {
    @Id @GeneratedValue private int id;

    @OneToOne(optional=false)
    @MapsId
    private Address address;
}
```

Verknüpfung über den-selben Schlüssel

```
import javax.persistence.*;

@Entity
public class Address {
    @Id @GeneratedValue private int id;
}
```

Beispiel: 1:n-Verknüpfung

```
import javax.persistence.*;
import java.util.List;

@Entity
public class Invoice {
    @Id @GeneratedValue private int id;

    @OneToMany(cascade={CascadeType.ALL})
    private List<InvoicePosition> positions;
}
```

Unidirektional

```
import javax.persistence.*;

@Entity
public class InvoicePosition {
    @Id @GeneratedValue private int id;
}
```

Beispiel: 1:n-Verknüpfung

Bidirektional

```
import javax.persistence.*;
import java.util.List;

@Entity
public class Invoice {
    @Id @GeneratedValue private int id;

    @OneToMany(mappedBy="invoice", cascade={CascadeType.ALL})
    private List<InvoicePosition> positions;
}
```

```
import javax.persistence.*;

@Entity
public class InvoicePosition {
    @Id @GeneratedValue private int id;

    @ManyToOne(fetch=FetchType.LAZY)
    private Invoice invoice;
}
```

Beispiel: n:m-Verknüpfung

```
import javax.persistence.*;
import java.util.List;

@Entity
public class Student {
    @Id @GeneratedValue private int id;

    @ManyToMany
    @JoinTable(name="LECTURES") // Ähnlich wie @JoinColumn, optional
    private List<Lecture> lectures;
}
```

```
import javax.persistence.*;
import java.util.List;

@Entity
public class Lecture {
    @Id @GeneratedValue private int id;

    @ManyToMany(mappendBy="lectures")
    private List<Student> students;
}
```

Verwendung der Objekte

Zugriff auf persistente Objekte nur via EntityManager möglich

Der EntityManager kapselt die eigentliche O/R-Funktionalität

Konfiguration durch einen Persistence Descriptor erforderlich

Interface `javax.persistence.EntityManager`

- Methoden zum Suchen / Finden von Objekten
- Methoden zum Speichern von Objekten
- Methoden zum Löschen von Objekten

Ein EntityManager je Persistence Unit

- **Java SE:** Zugriff über die Entity Manager Factory
- **Jave EE:** EntityManager wird vom Container bereitgestellt

Beispiel: Persistence Descriptor

```
<persistence>
    <persistence-unit name="jpa-test" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>de.dhbw.example.Person</class>
        <class>de.dhbw.example.Customer</class>
        <class>de.dhbw.example.Vendor</class>

        <properties>
            <property name="hibernate.connection.driver_class"
                      value="org.hsqldb.jdbcDriver"/>

            <property name="hibernate.connection.url"
                      value="jdbc:hsqldb:data/examples"/>

            <property name="hibernate.connection.username" value="uname"/>
            <property name="hibernate.connection.password" value="paswd"/>

            <property name="hibernate.hbm2ddl.auto" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Zugriff auf den EntityManager

Java SE (Ohne Applikationsserver)

```
import javax.persistence.*;

EntityManagerFactory emf = Persistence.createEntityManagerFactory(
                                "exampleUnit");
EntityManager em = emf.createEntityManager();
...
emf.close();
```

Java EE (Mit Applikationsserver)

```
import javax.persistence.*;

public class MisterBean {
    @PersistenceContext(unitName="exampleUnit")
    private EntityManager em;

    ...
}
```

Methoden des Entity Managers

`EntityTransaction getTransaction()`
Erzeugt ein neues Objekt für die Transaktionssteuerung

`Object find (class, primaryKey)`
Sucht einen einzelnen Datensatz in der Datenbank

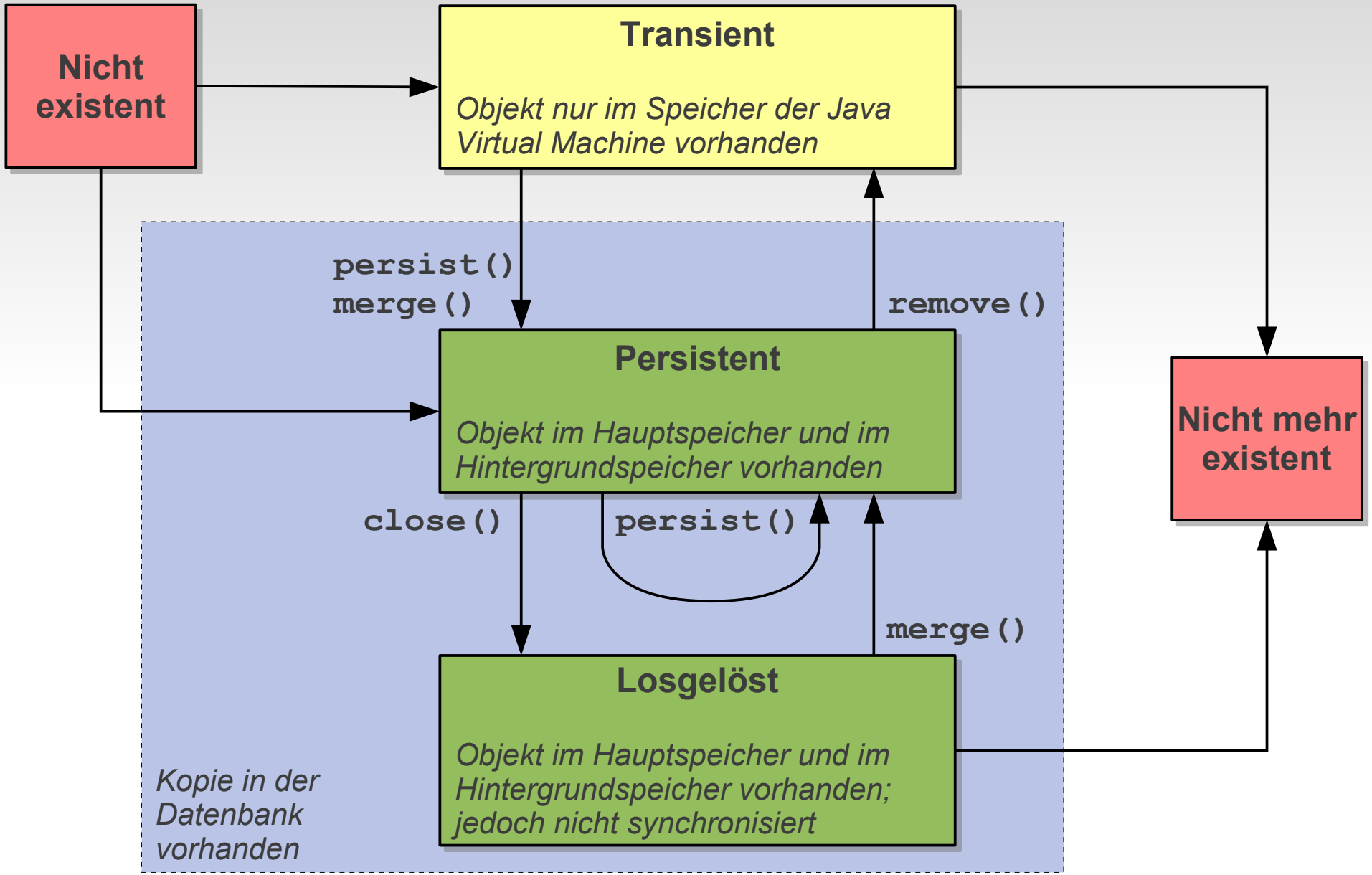
`void lock(object, lockMode)`
Setzt eine Schreib- oder Lesesperre auf einen Datensatz

`void persist(Object)`
Speichert das übergebene Objekt in der Datenbank (Neuanlage)

`void merge(Object)`
Speichert das übergebene Objekt in der Datenbank (DB-Update)

`void remove(Object)`
Löscht das übergebene Objekt aus der Datenbank

Persistenter Lebenszyklus



Beispiel: Entity Manager

```
public class MisterBean {
    @PersistenceContext EntityManager em;

    public void changePerson() {
        Person person = em.find(Person.class, 2850);
        em.lock(person, LockModeType.WRITE);
        person.setName("Mister Bean");
        person.setSize(182);
        em.merge(person);
    }

    public void changeInvoices() {
        List<Invoice> invoices = em.createQuery(
                    "SELECT i FROM Invoice i WHERE i.amount < :amount")
                    .setParameter("amount", 250)
                    .getResultList();

        for (Invoice invoice : invoices) {
            em.lock(invoice, LockModeType.WRITE);
            invoice.setCurrency("EUR");
            em.merge(invoice); // Entsperrt das Objekt wieder!
        }
    }
}
```


B
mf


lang hat's gedauert ...


WIR SIND AM ENDE
DER VORLESUNG
ANGELANGT


That's all Folks!
A WARNER BROS. CARTOON