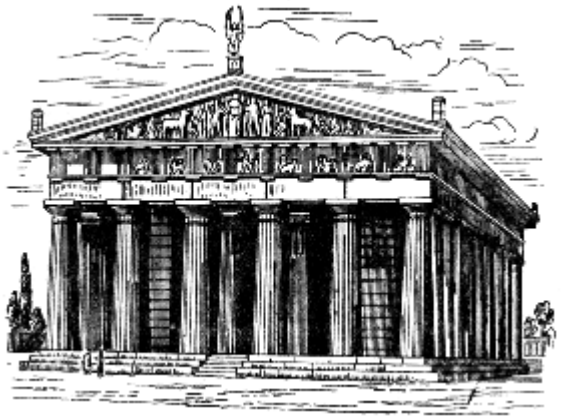
Softwarearchitektur

(Architektur: *αρχή* = Anfang, Ursprung + *tectum* = Haus, Dach)

9. Betriebliche Informationssysteme – Teil 2



Vorlesung Wintersemester 2010 / 11

Technische Universität München
Institut für Informatik
Lehrstuhl von Prof. Dr. Manfred Broy

Dr. Klaus Bergner, Prof. Dr. Manfred Broy, Dr. Marc Sihling

Inhalt

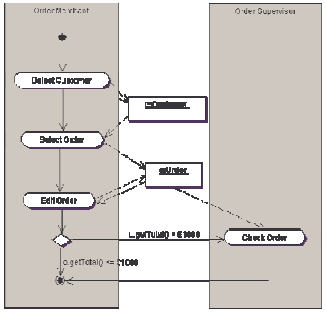
- Rekapitulation Teil 1
  - Schichtenarchitekturen
  - Datenhaltungsschicht
- Anwendungsschicht
  - Aufgaben und Charakteristika
  - Dienste und Middleware
- Literaturhinweise

Inhalt

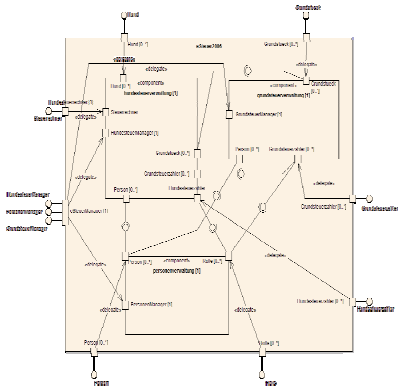
- Rekapitulation Teil 1
  - Schichtenarchitekturen
  - Datenhaltungsschicht
- Anwendungsschicht
  - Aufgaben und Charakteristika
  - Dienste und Middleware
- Literaturhinweise

Logische Schichtenarchitektur

Die Aspekte der fachlichen und technischen Architektur werden vier Schichten zugeordnet. Die Unterstützungsschicht bietet querschnittliche Funktionalität.



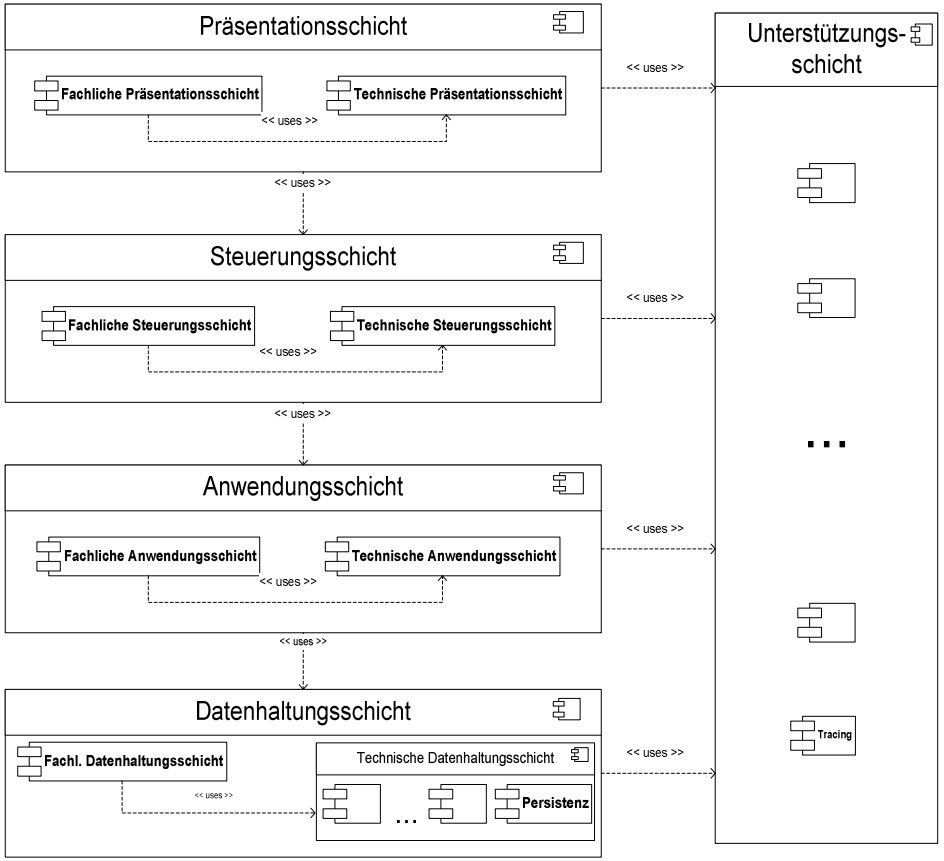
GUIs zu Aktivitäten



Workflows zu Prozessen

Fachliche Komponenten

Datenlayout zu Objekten

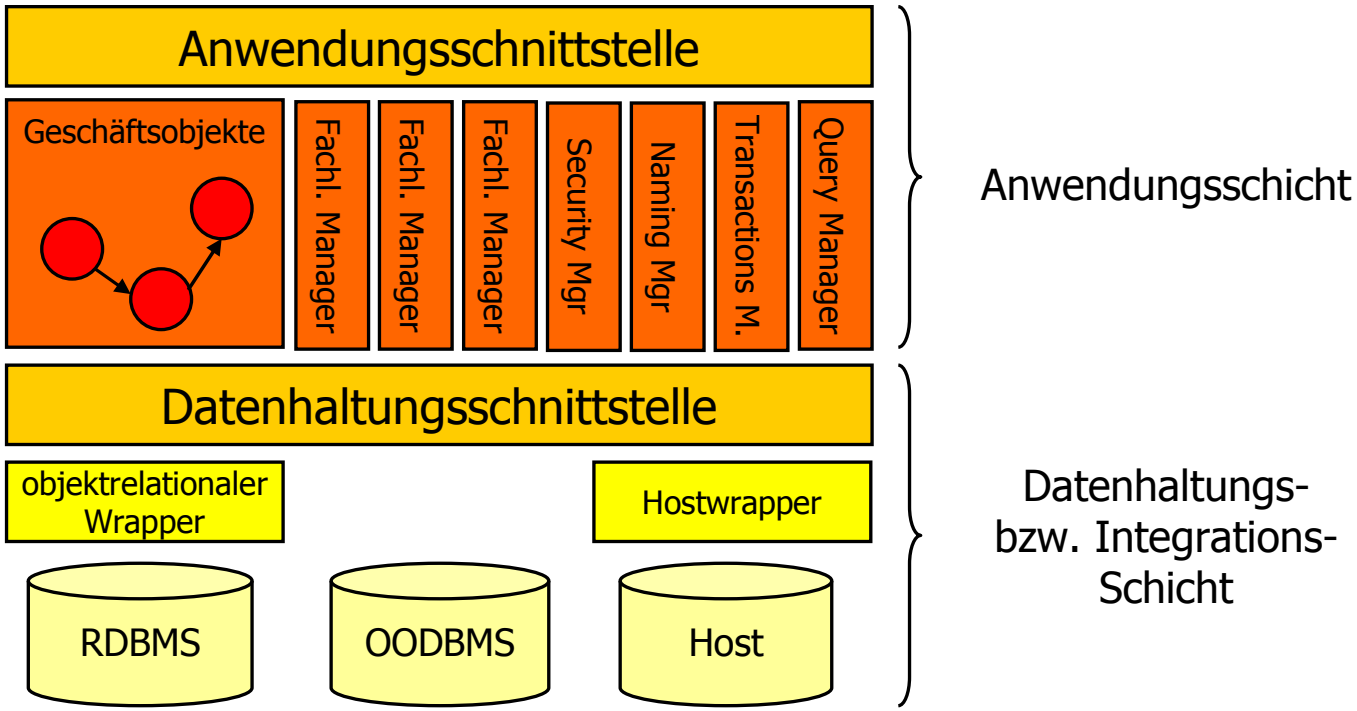


Aufgaben der Anwendungsschicht

- Zugriff auf Anwendungsfunktionalität
  - Geschäftsobjekte inklusive Daten und Operationen
  - Sonstige Funktionalität und Anwendungsdienste
- Basis für darauf aufbauende Schichten
  - Unterschiedliche Präsentationen (Multi-Channel Presentation)
  - Voraussetzungen für Ansteuerung durch Steuerungsschicht
- Einfaches Programmiermodell für Anwendungsprogrammierer
  - Verstecken der Datenhaltungsschicht vor anderen Schichten
  - Kapselung proprietärer technischer Dienste der Anwendungsschicht hinter Schnittstellen
  - Transaktionen als Mittel für Programmierer, so zu entwickeln, als gäbe es nur einen Benutzer und keine Parallelität

Aufgabe der Anwendungsschicht

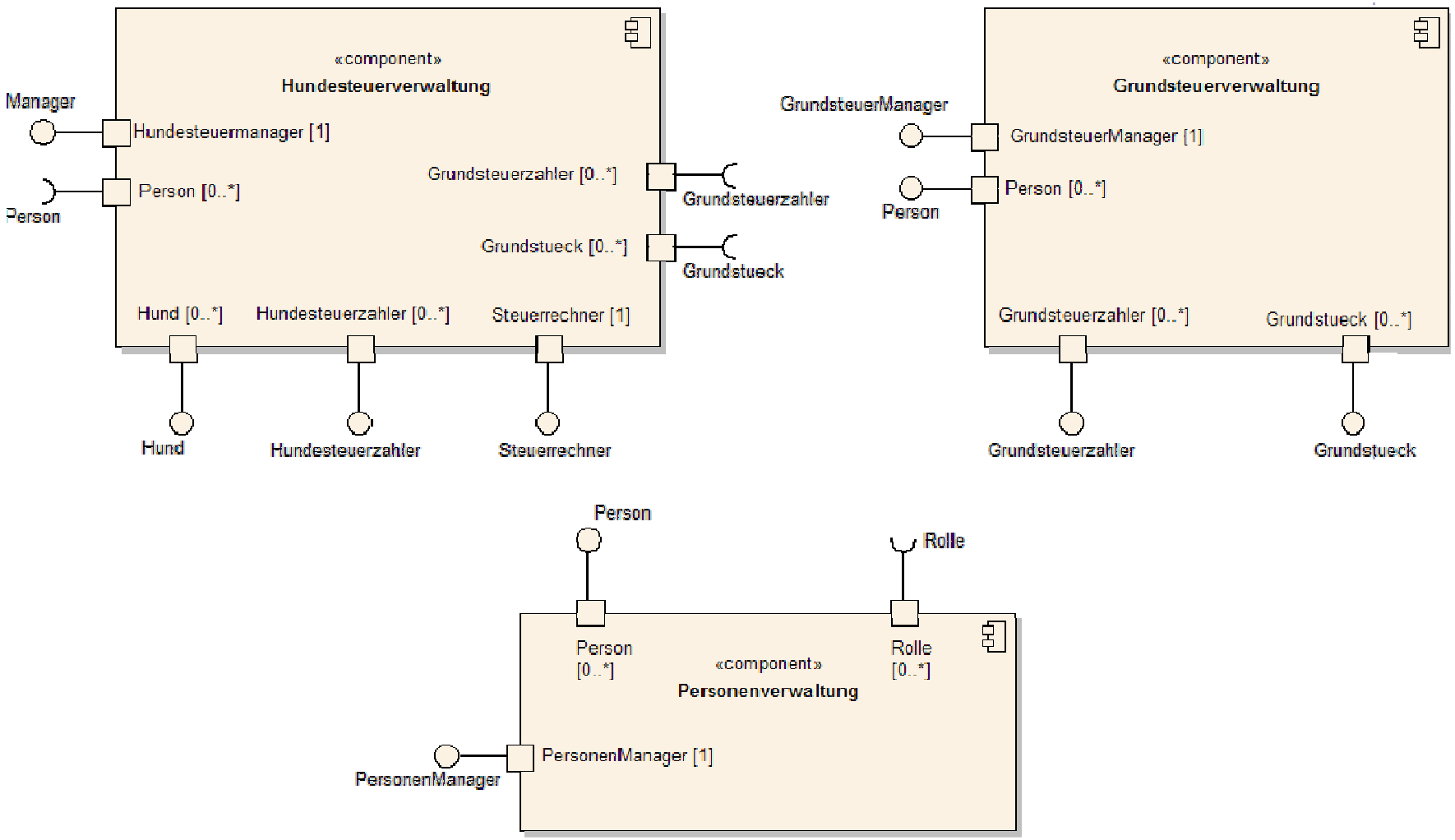
- **Anwendungsschicht** kümmert sich um technische Belange mit Hilfe dedizierter Manager für technische Dienste.
- **Anwendungsschnittstelle** stellt uniforme Sicht auf fachliche und technische Komponenten dar.



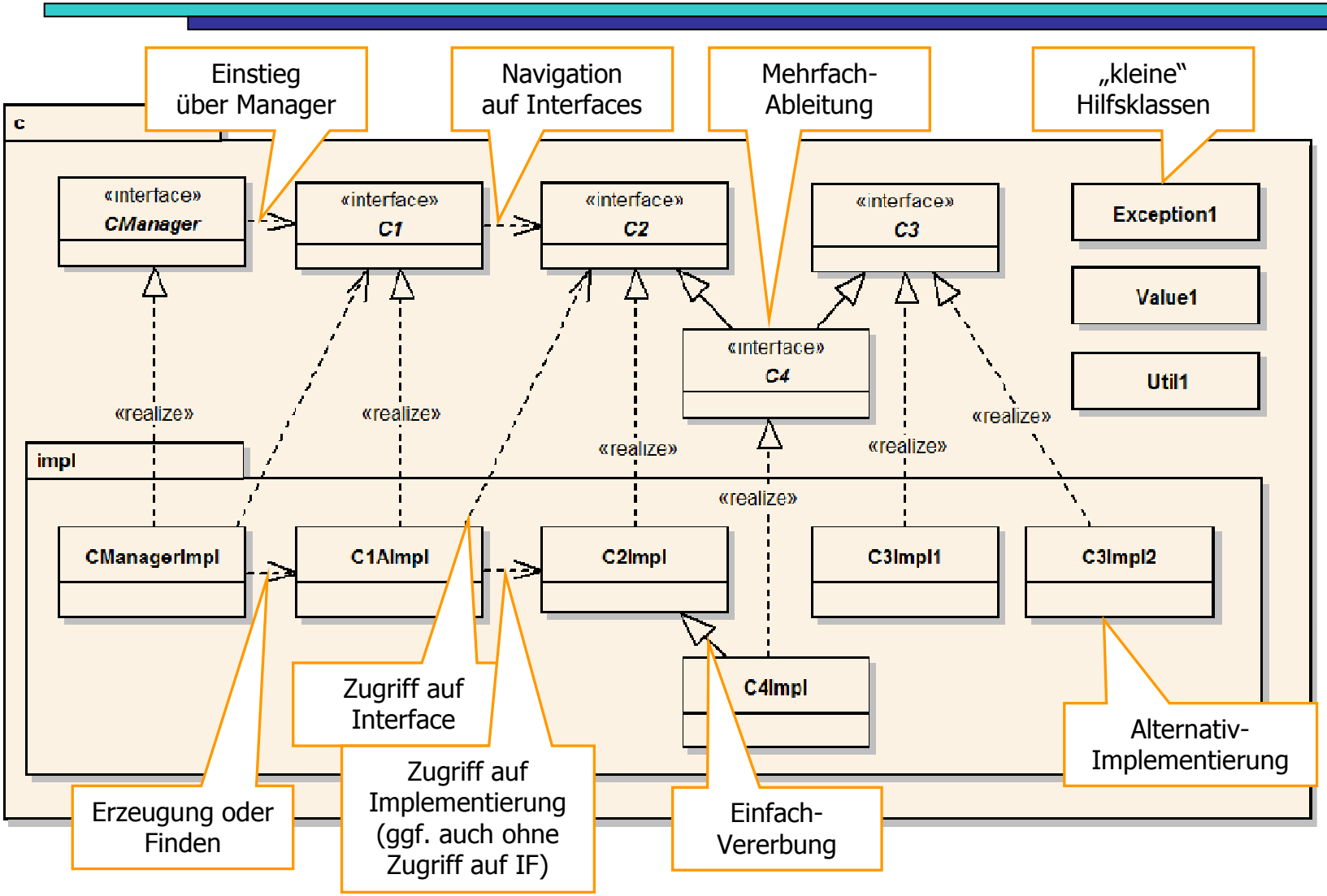
Fragen bei der Realisierung der Anwendungsschicht

1. Wie gestaltet man die Schnittstellen und Implementierung?
   - Interface-Oriented Design von Komponenten und Abhängigkeiten
   - Objektorientierte vs. Service-Orientierte Schnittstellen
2. Wie findet und verknüpft man Komponenten?
   - Service Locator, Komponenten-Infrastrukturen
   - Middleware für Naming etc.
3. Wie schreibt man transaktionalen Anwendungscode?
   - Schreiben von Transaktionen
   - Transaktionsmanager (z.B. JTA, CORBA Transaction Service)
4. Wie kommunzieren entfernte Komponenten miteinander?
   - Synchrone Kommunikation (z.B. RPC, CORBA, RMI)
   - Asynchrone Kommunikation über Messaging (z.B. JNDI)
   - Transaktionales Messaging
5. Welche andere Dienste kann die Anwendungsschicht bieten?
   - Beispiel CORBA Services

Komponenten und Schnittstellen aus Facharchitektur



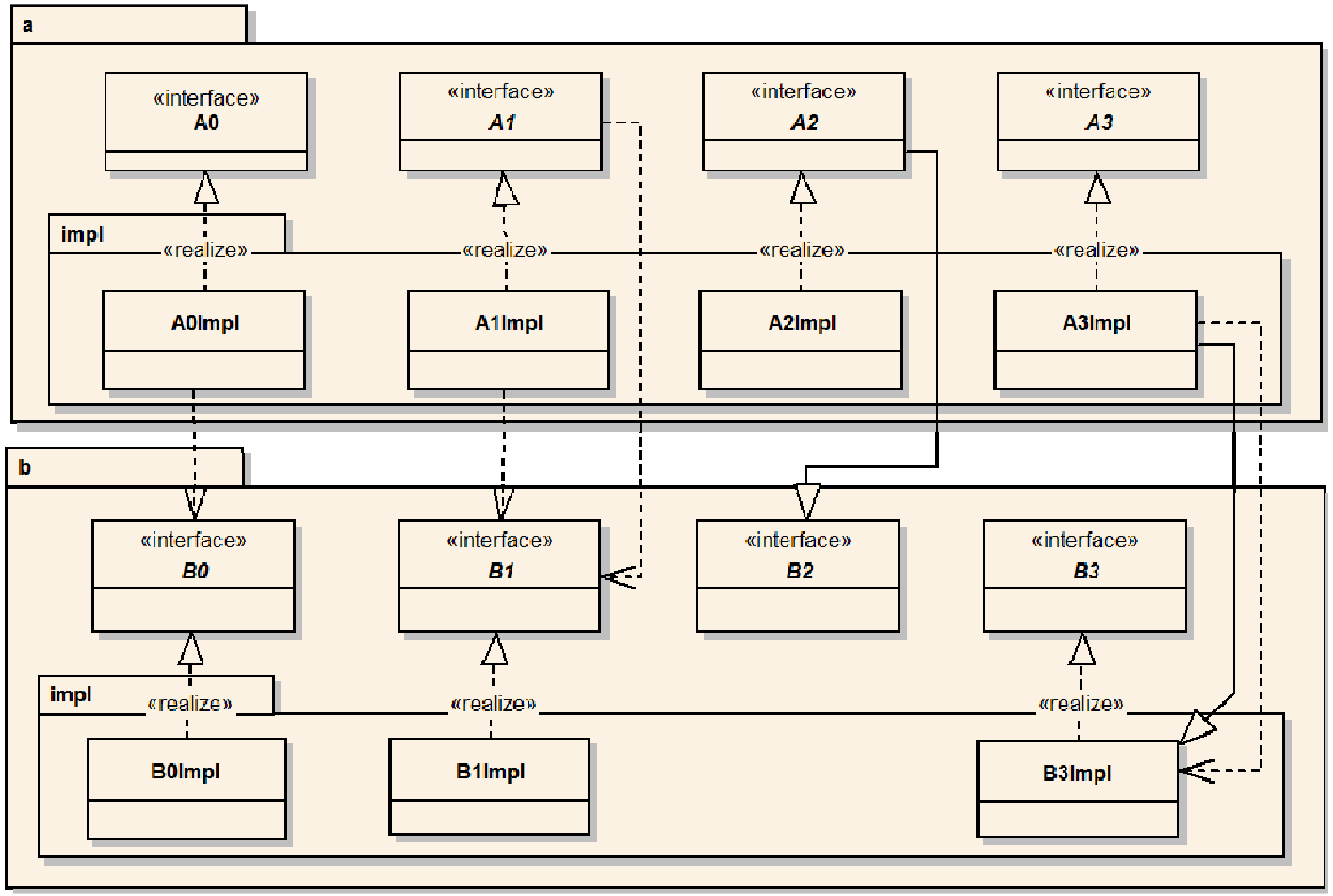
Interface-Oriented Design: Schnittstelle und Implementierung



Interface-Oriented Design: Abhängigkeiten zwischen Komponenten

abhängige Komponente

Basis-komponente



„Normale" Abhängigkeit

Schnittstellen-abhängigkeit

Schnittstellen-Ableitung

Implementierungs-Vererbung/Abhggkt.

OSGi als Komponenten-Infrastruktur

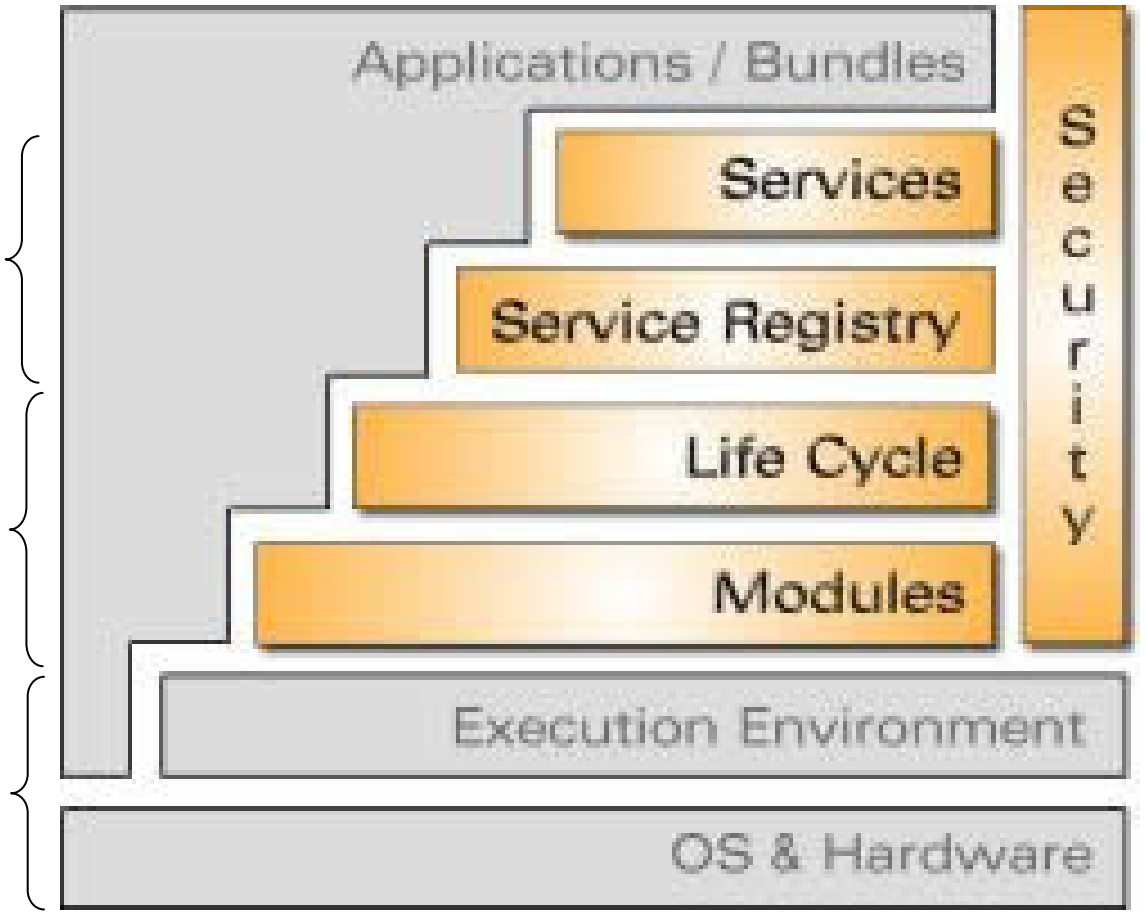
- OSGi – Open Services Gateways Initiative
- OSGi Alliance wurde 1999 gegründet
  - Ziel: Spezifizierung einer offenen Laufzeitumgebung für Services
- Mitglieder sind unter anderem BMW, Telekom, Siemens, Sun Microsystems und andere.
- Standardisierte Laufzeitumgebung für Services
  - Installation, Start, Stop, Update, Deinstallation
  - Berücksichtigung von Versionen und Abhängigkeiten der Services

Überblick über die OSGi-Architektur

Dienste bereitstellen und verschalten

Komponenten verwalten und konfigurieren

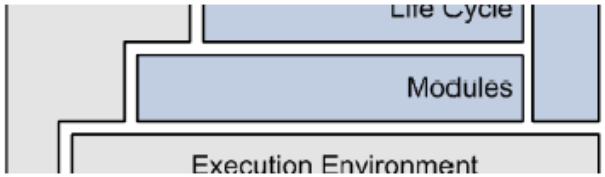
Basisplattform kapseln



Verwaltung von Komponenten in OSGi–Bundles

Class Loading

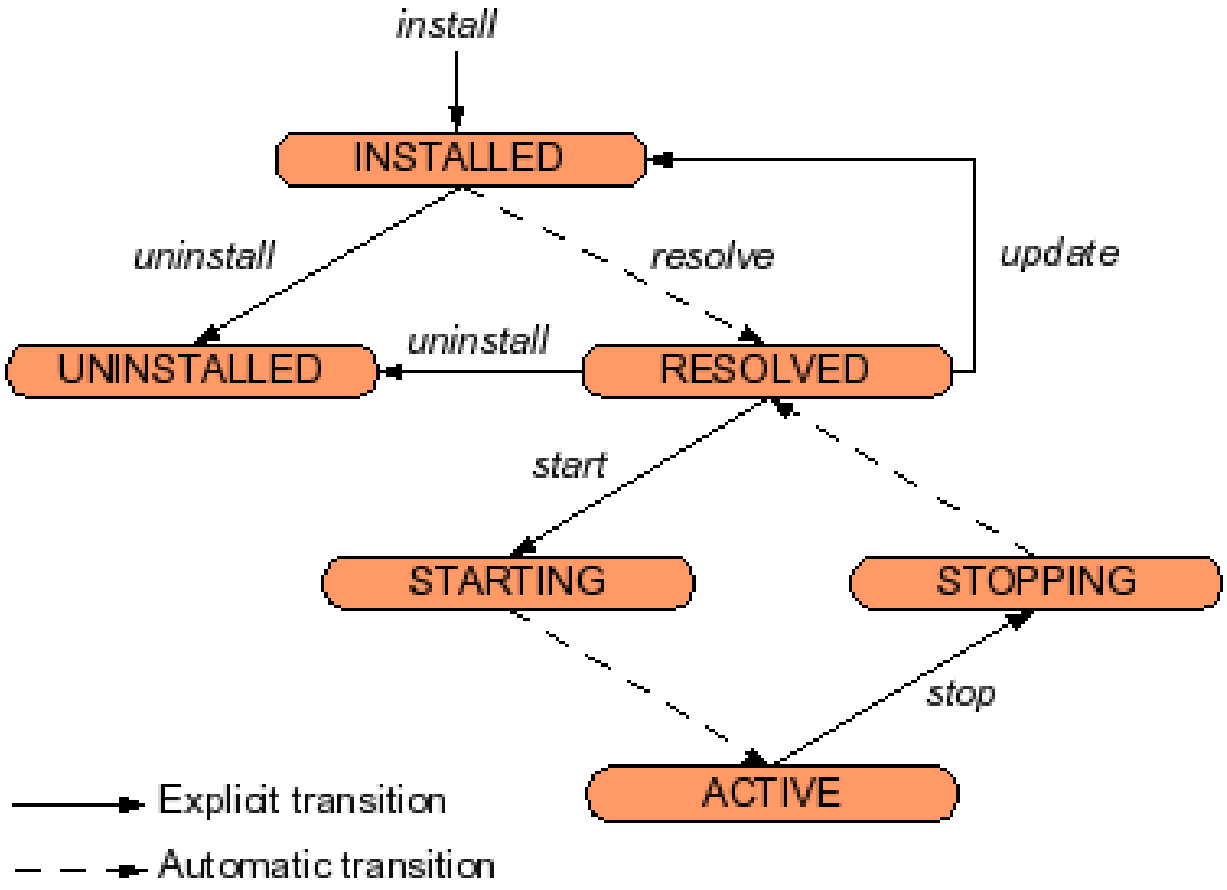
- Vollständige Trennung der Module
- Kontrollierte Zugriffe auf Ressourcen ermöglichen



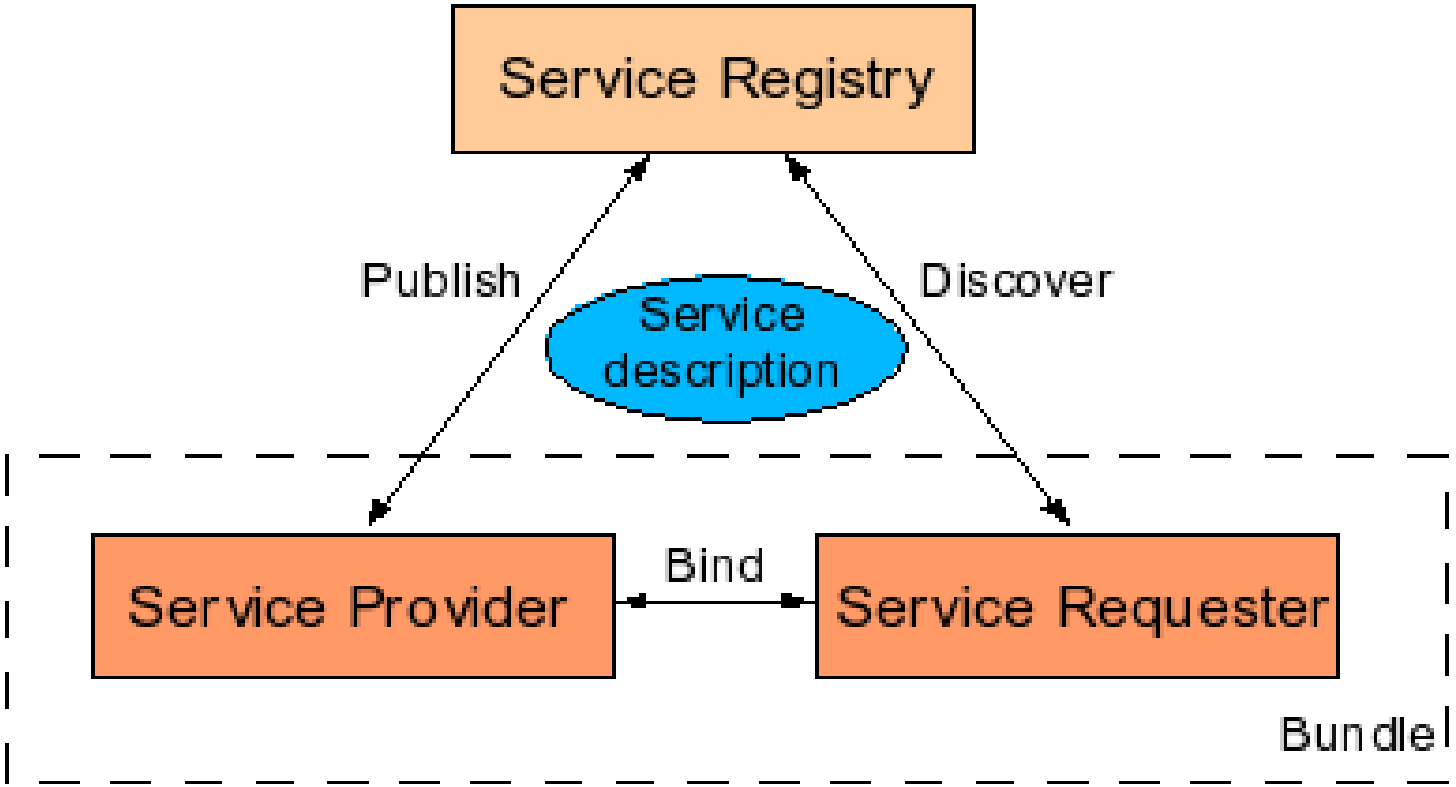
→ Jedes Bundle erhält eine Manifestdatei mit *Import-Package* und *Export-Package* Angaben (Beispiel)

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.2
Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
Bundle-Name: Log Service
Bundle-SymbolicName: org.knopflerfish:Log Service:1.0.0
Bundle-Version: 1.0.0
Bundle-Description: The Knopflerfish OSGi log service
Bundle-Vendor: Knopflerfish
Export-Package: org.osgi.service.log; specification-version=1.2,
org.knopflerfish.service.log; specification-version=1.1
Import-Package: org.knopflerfish.service.console,
org.knopflerfish.service.log,org.osgi.framework, org.osgi.service.cm,org.osgi.service.log
Bundle-UUID: org.knopflerfish:Log Service:1.0.0:all
```

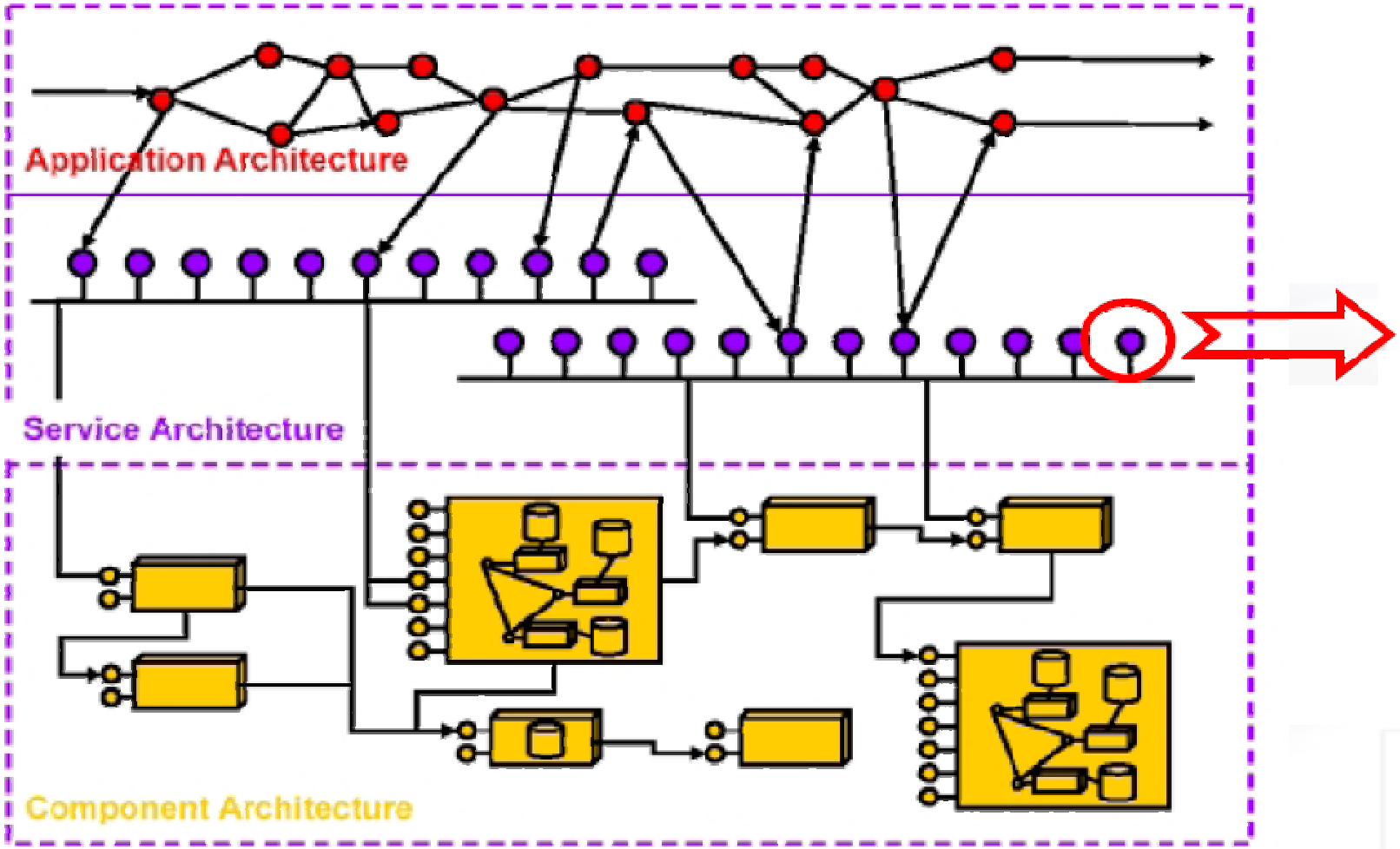
Zustand von OSGi-Bundles



Nutzung von Bundles innerhalb von OSGi



Services als Anwendungs-Bausteine



Objektorientierte vs. Service-Orientierte Schnittstellen

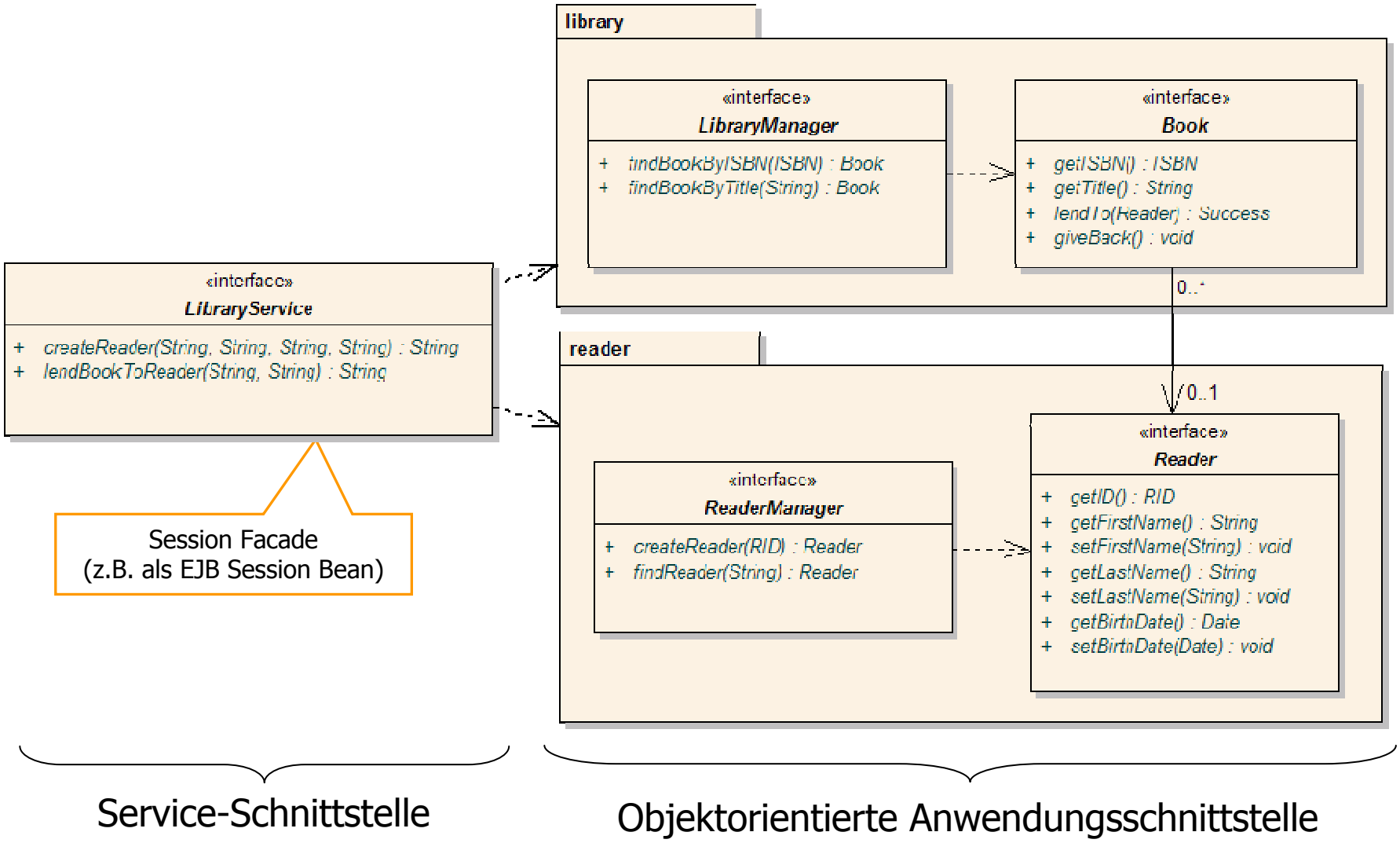
Objekt-Orientierung

- Ziele
  - Geschäftsobjekte und fachliche Funktionalität umsetzen
  - Einfache und effiziente Implementierung
- Eigenschaften
  - Objektidentität
  - Eigens definierte Typen
  - Feingranulare Schnittstelle
    - Attributzugriff und Management einzelner Assoziationen
    - Einfache, beliebig kombinierbare fachliche Operationen

Service-Orientierung

- Ziele
  - Kontextlose, für sich stehende Geschäftstransaktionen umsetzen
  - Optimierte Kommunikation, möglichst wenige Aufrufe
- Eigenschaften
  - Fachliche Identifikatoren
  - Verwendung von Standardtypen
  - Grobgranulare Schnittstelle
    - Übergabe von Objektgeflechten als Structures / Value Objects
    - Wenige, komplexe Operationen (entsprechend Anwendungsfällen)

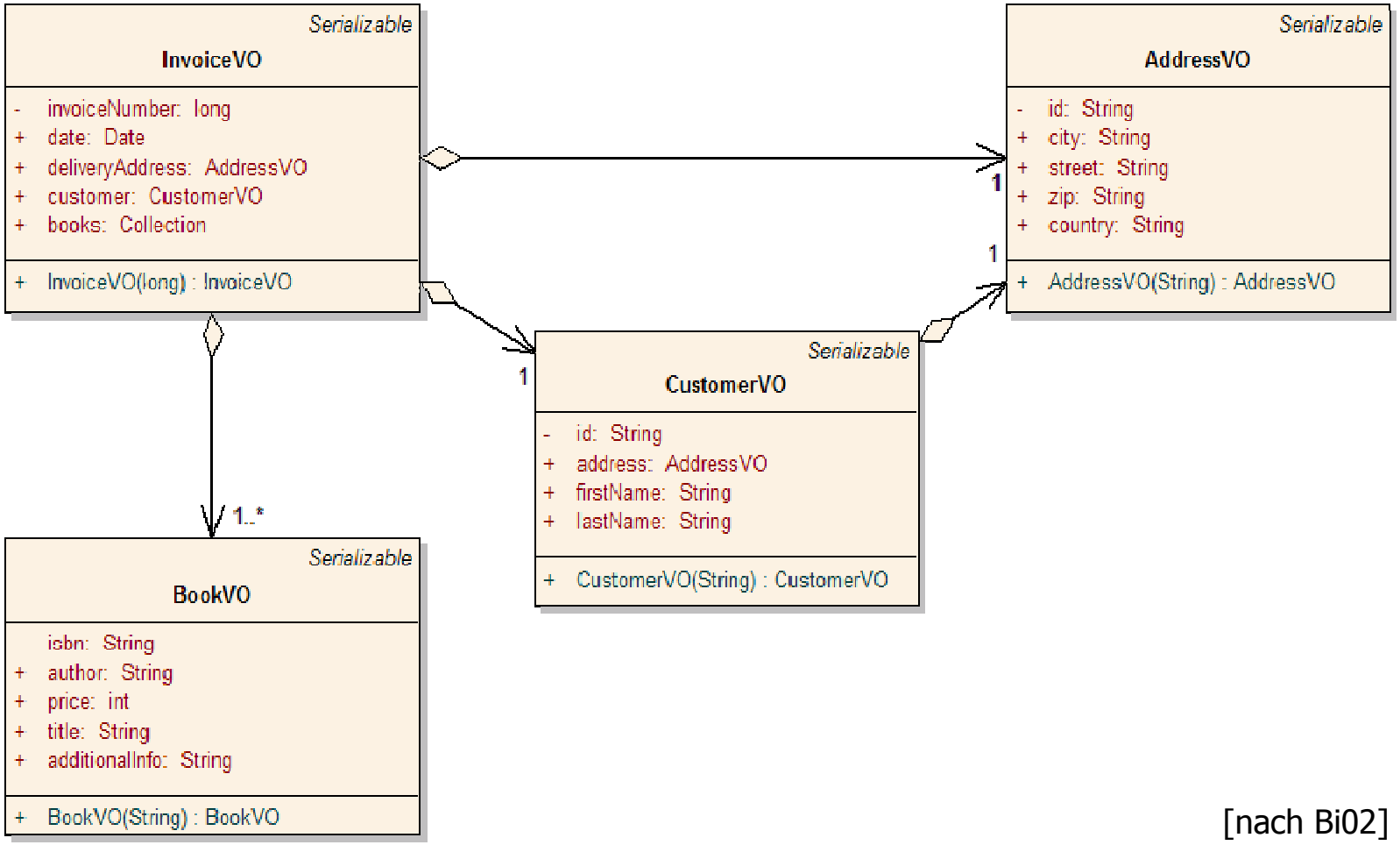
Objektorientierung vs. Service-Orientierung - Beispiel



Value Objects

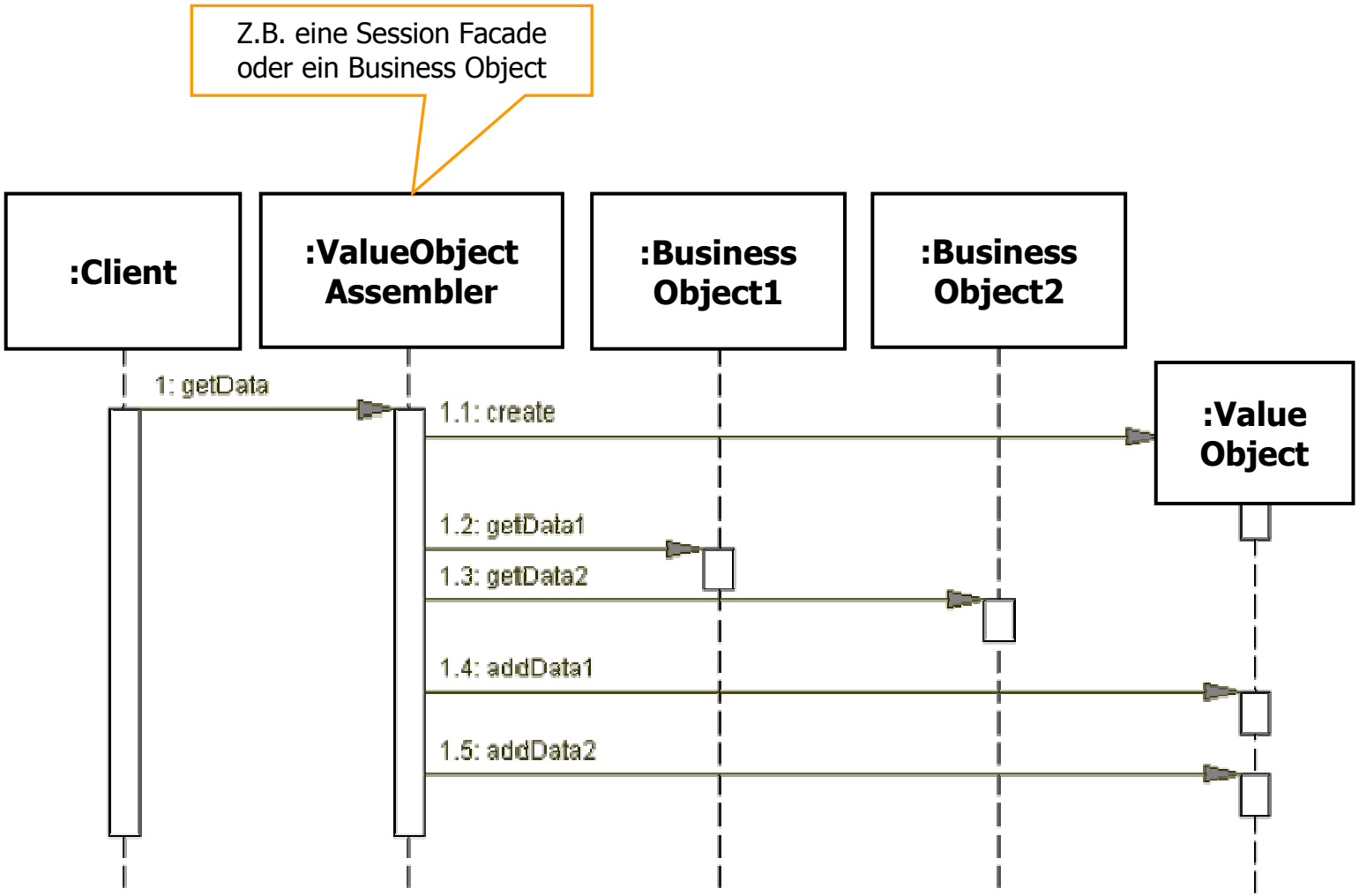
- Serialisierbare Daten-Objekte ohne weitere Funktionalität
  - nur Attribute und ggf. get/set-Methoden
  - Spiegeln Ausschnitte der Struktur der Geschäftsobjekte
- Ziele und Eigenschaften
  - \+ Minimierung der Anzahl der Übertragungen über Netzwerke
  - \+ Abkürzung langer Parameterlisten
  - \+ Leichte Abbildung auf unterschiedliche Programmiersprachen und Formate (z.B. als CORBA-Structs oder XML-Strukturen)
  - \- Zusätzliche Komplexität durch zusätzliche Klassen (→ Generierung)
- Implementierung
  - Zusammenfassung von Objekten anhand von Aggregationsbeziehungen
  - Value Objects enthalten typischerweise (nicht änderbare) ID zur eindeutigen Abbildung auf Geschäftsobjekte

Value Object - Beispiel



[nach Bi02]

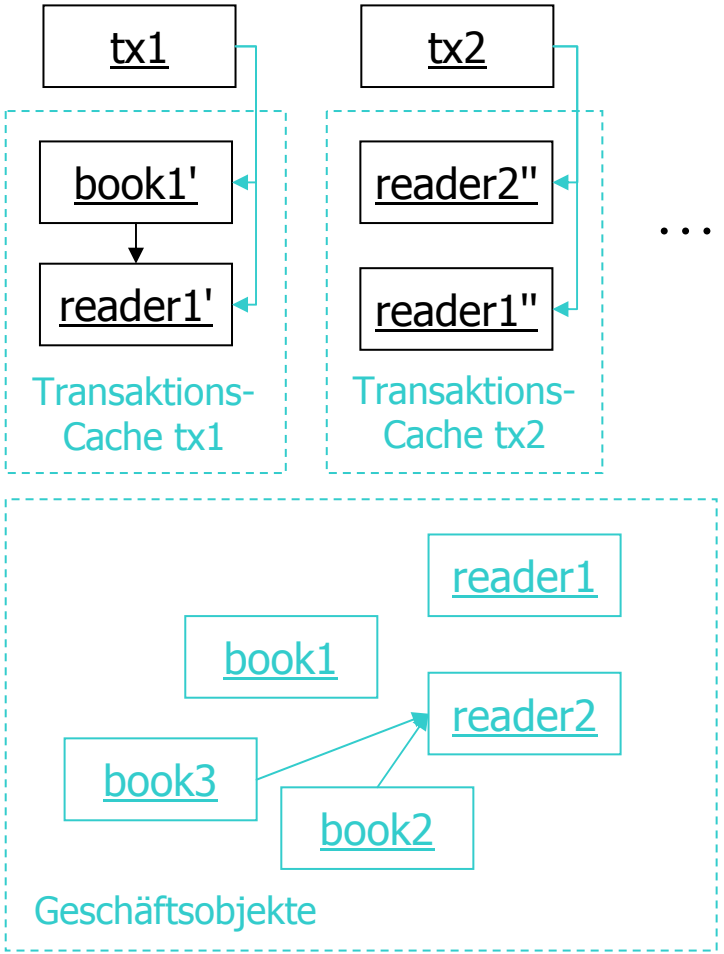
Value Object Assembler



Transaktions-Management

```
Transaction tx1 =
   taMgr.newTransaction();
…
try {
  tx1.begin();
  Reader reader1 = readerManager.
    findReader("Huber");
  reader1.setFirstName("Otto");
  Book book1 = libraryManager.
    findBookByISBN("1-2345-6789-0");
  book1.borrow(reader1);
  tx1.commit();
} catch Exception e {
  …
}
```

Commit schlägt fehl insbesondere bei Konflikt mit anderer Transaktion.

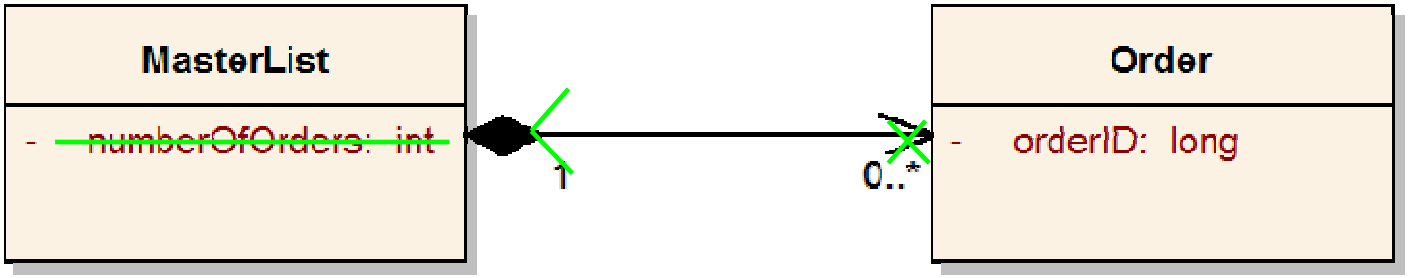


Transaktionsmanagement – keine heile Welt

- Meist können Architektur und verwendete Middleware die ACID-Prinzipien nicht vollständig garantieren.
  - Transaktionsmodi verletzen Isolation
    - **Serializable:** vollständige Isolation, geringste Performance
    - **Repeatable Read:** Anzahl der Objekte kann sich ändern, neue Objekte aus anderen TA-Commits kommen dazu (Phantom Reads)
    - **Read Committed:** mehrfaches Lesen eines Objekts kann unterschiedliche Werte ergeben (Nonrepeatable Reads)
    - **Read Uncommitted:** keine Isolation, keine TA-Caches, höchste Performance; inkonsistente Zwischenstände sichtbar (Dirty Reads)
  - Keine Atomarität bei verteilten Transaktionen
    - Siehe letzte Vorlesung.
- Anwendungscode muss auf Eigenheiten beim Transaktions-management Rücksicht nehmen
    - Transaktionskontext möglich klein halten …

Anti-Beispiel: MasterList

- Transaktionsgesteuertes eCommerce-System
- Fachliche Funktionalität
  - Bearbeiter erzeugen und bearbeiten Aufträge
  - Aufträge werden in eine MasterList eingetragen
  - Transaktion für Erstellen eines neuen Auftrags und Einfügen in MasterList
- Problem
  - Viele parallele Transaktionen gehen schief



- Lösung
  - Verhindern, dass immer auch MasterList in Transaktionskontext gerät
  - Attribut numberOfOrders streichen
  - Richtung der Verweise umkehren

Middleware zum Transaktionsmanagement

- Stand-Alone-Systeme
  - Eigene Schnittstellen zum TA-Management
  - Keine Teilnahme an verteilten Transaktionen
  - Beispiele: einfache RDBMS, ODBMS

- Standardisierte Transaktions-Infrastrukturen
  - Implementieren Standard-Schnittstellen wie JTA, CORBA OTS
  - Kommunizieren über Standard-Protokolle wie XA
  - Teilnahme an verteilten Transaktionen in den Rollen:
    - Transaktionskoordinator
    - Transaktionale Ressource
  - Beispiele: RDBMS, ODBMS, Message Services, Application Server

Client-Managed vs. Container-Managed Transactions

Client-Managed

- Transaktionsklammern werden explizit gesetzt
- Transaktionen über Factory erzeugen oder bekommen
- begin, commit, abort
- Vor-/Nachteile
  - \+ Feingranulare Steuerung möglich
  - \- Mehr Möglichkeiten für Fehler

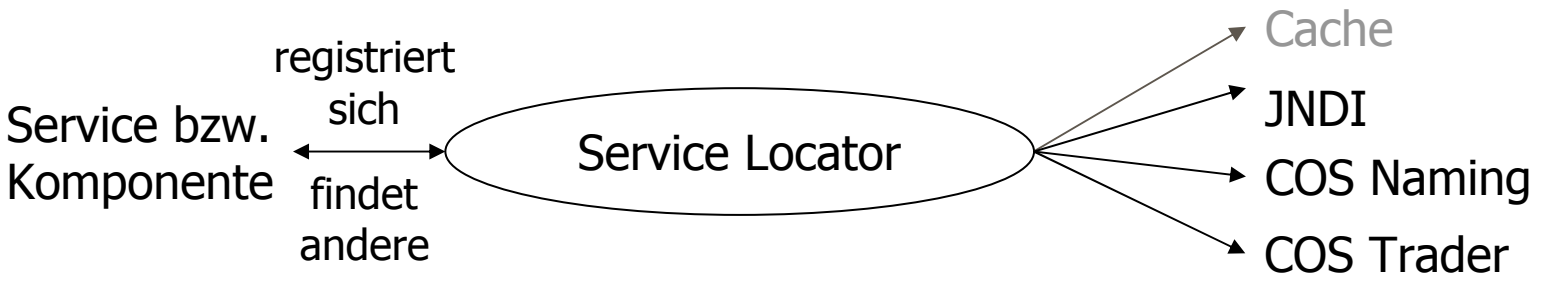
Container-Managed

- Transaktionsklammern werden konfigurationsgesteuert gesetzt
- Transaktionen werden durch Container / AOP injiziert
- Geeignet insbesondere für Service-Schnittstellen
- Vor-/Nachteile
  - \+ Einheitlichkeit, Einfachheit
  - \- Starres Schema

Insgesamt gilt: Ein System sollte möglichst nur eine einzige, einheitliche Strategie für das Transaktionsmanagement haben.

Finden von Komponenten und Diensten

- **Service Locator**
  - Zentrale Anlaufstelle, um Manager oder Dienste zu finden
  - Kapselt Middleware für Naming, Trading oder Verzeichnisdienste
  - Kann zusätzliche Aufgaben übernehmen (z.B. Caching)



- **Verschalten durch Komponenten selbst**
  - Nur erforderlich, wenn Anwendungswissen erforderlich ist
- **Verschalten durch eigenen Service oder Container**
  - Dependency Injection stellt Unabhängigkeit der Komponenten sicher
  - Container kann Zugriff auf Verzeichnisdienste übernehmen

Middleware für Naming, Directory, Trading

- **Naming Service**
  - Binden von Namen an Objekte
  - Finden von benannten Objekten
  - Beispiel: COS Naming Service
- **Directory Service**
  - Zusätzlich hierarchische Kontexte (Directories) und Namens-Pfade
  - Zusätzlich Attribute zu Objekten
  - Beispiel: JNDI-Schnittstelle, ActiveDirectory, LDAP-Server
- **Trading Service**
  - Erlaubt es, Kriterien für Objekte und Dienste anzugeben
  - Beispiel: Gib mir Service, der
    - Interface BankingService implementiert
    - Verfügbarkeit 95% hat
  - Beispiele:
    - COS Trader für CORBA Objects
    - UDDI für Web-Services im Kontext von Unternehmensdiensten

Beispiel: Schnittstelle Naming Service

```
interface NamingContext {

   // Definiere neues Binding zwischen n und obj.
   void bind (in Name n, in Object obj);

   // Binde existierenden Namen n an anderes Objekt obj.
   void rebind (in Name n, in Object obj);

   // Lösche Binding von n.
   void unbind (in Name n);

   // Finde an Namen n gebundenes Objekt.
   Object resolve (in Name n);

   // Erzeuge neuen (leeren) Namenskontext.
   NamingContext new_context ();

   // Zerstöre Namenskontext.
   void destroy ();

   // Weitere Methoden (z.B. zum Auflisten) nicht gezeigt...
}
```

Konfigurationsgesteuerte Verschaltung durch Container

Konfigurationsdateien dieser Art definieren in einem XML-Format deklarativ die Verschaltung der diversen Komponenten (also den „Glue“)

Komponenten werden über Factories erzeugt (hier nun endlich die Komp.-implementierung)

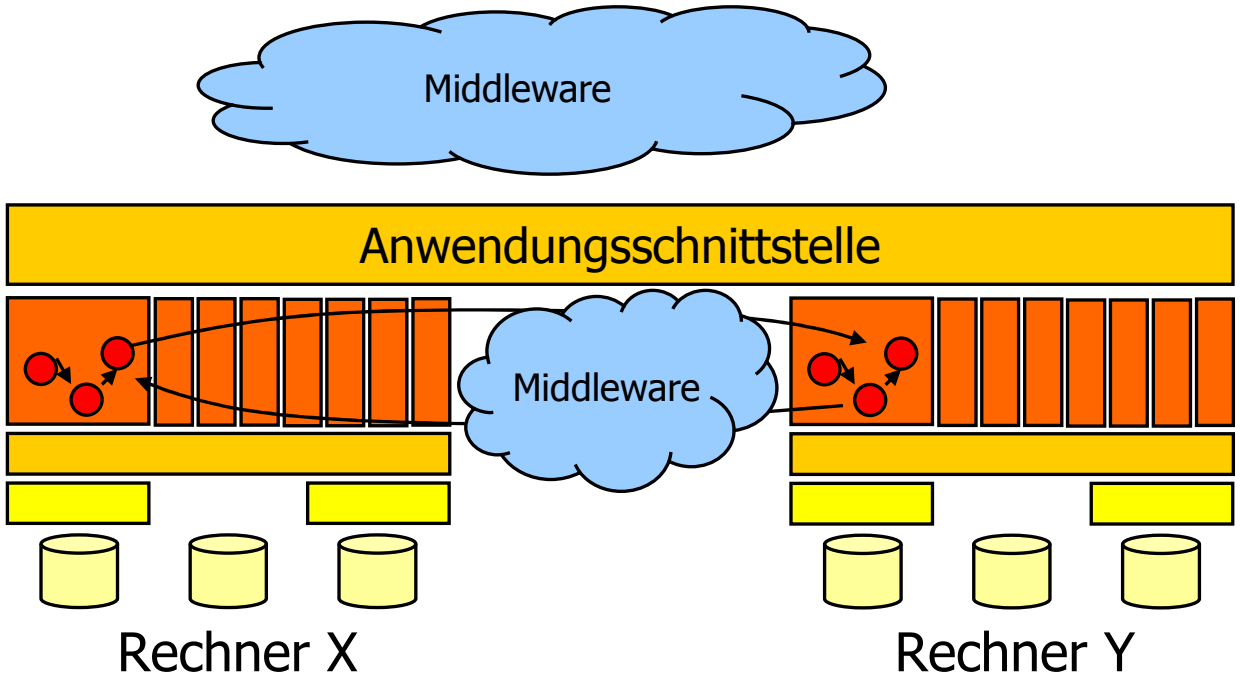
Erst nach Berücksichtigung aller Abhängigkeiten entscheidet Spring über die Reihenfolge der Initialisierung aller Komponenten

Spring stellt weiterhin einen „Application Context“ zur Verfügung, über den nach Komponenten gesucht werden kann („*dependency lookup*“)

```
<bean id="logfactory"
      class="org.apache.commons.logging.LogFactory"
      factory-method="getFactory">
</bean>
<bean id="objectmodelmanager">
    ...
</bean>
<bean id="xmlfileiolog" factory-bean="logfactory"
    factory-method="getInstance">
    <constructor-arg>
     <value>com.foursoft.fourever.xmlfileio</value>
    </constructor-arg>
</bean>
<bean id="xmlfileiomanager"
    class="com.foursoft.fourever.xmlfileio.impl.XML
           FileIOManagerImpl"
    factory-method="createInstance"
    destroy-method="destroy">
    <constructor-arg>
          <ref bean="xmlfileiolog"/>
    </constructor-arg>
    <constructor-arg>
          <ref bean="objectmodelmanager"/>
    </constructor-arg>
</bean>
```

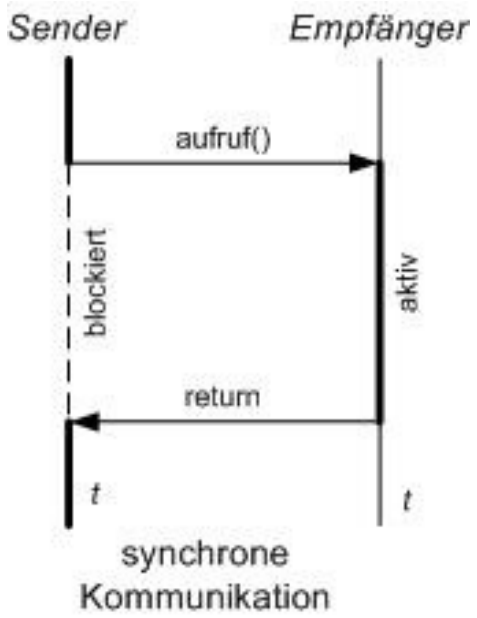
Verteilungssicht der Anwendungsschicht

- Unter einer gemeinsamen Anwendungsschnittstelle ist die räumliche Verteilung der Anwendungsobjekte transparent.
- Technische Mechanismen müssen daher in geeigneter Form kommunizieren (z.B. Transaction Manager, Security Manager, Query Manager).



Synchrone Kommunikation

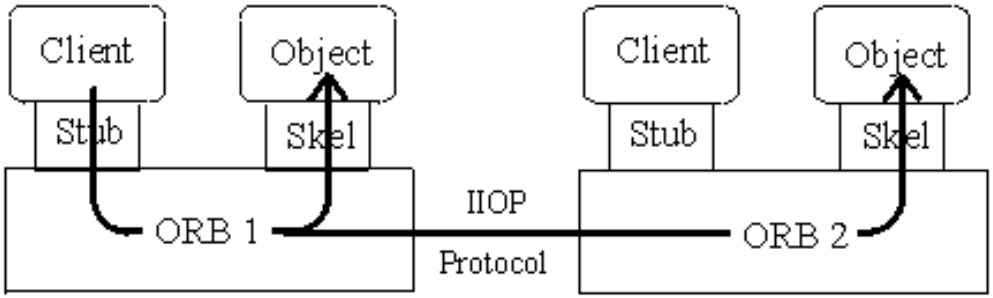
- Sender und Empfänger blockieren beim Ausführen der Operationen zum Senden bzw. Empfangen.
- Eigenschaften:
  - Enge Kopplung von Sender und Empfänger.
  - Empfänger muss bereitstehen: Wenig robust, nicht gut skalierbar.
  - Einfaches Programmiermodell für Anwendungs-programmierer (keine parallele Ausführung).
  - Hohe Abhängigkeit insbesondere im Fehlerfall.
- Voraussetzung:
  - Sichere und schnelle Netzverbindungen sind verfügbar.
  - Empfangender Prozess ist verfügbar.



synchrone Kommunikation

Synchrone entfernte Kommunikation

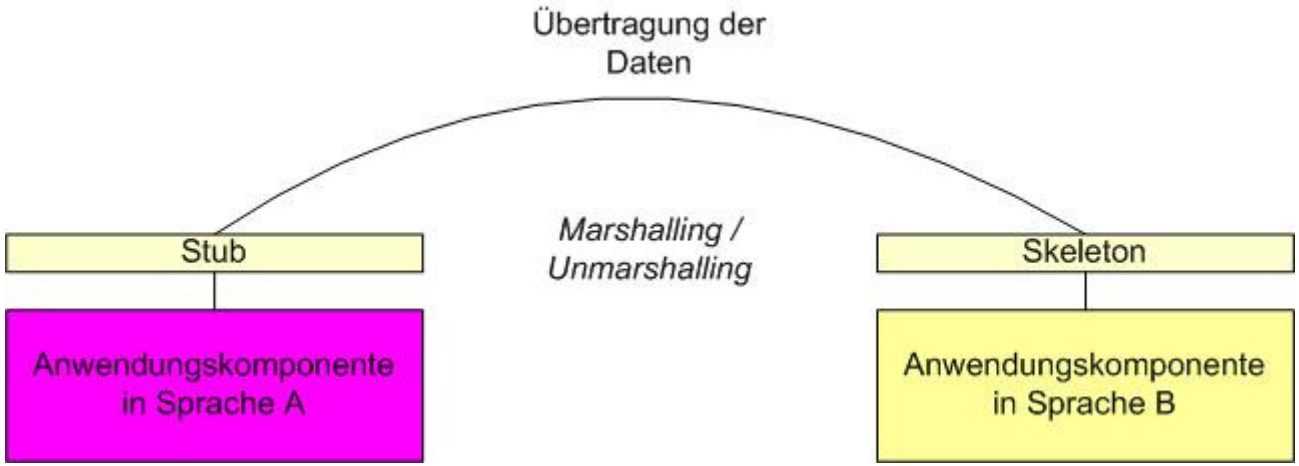
- Entfernte Methodenaufrufe über Stubs und Skeletons



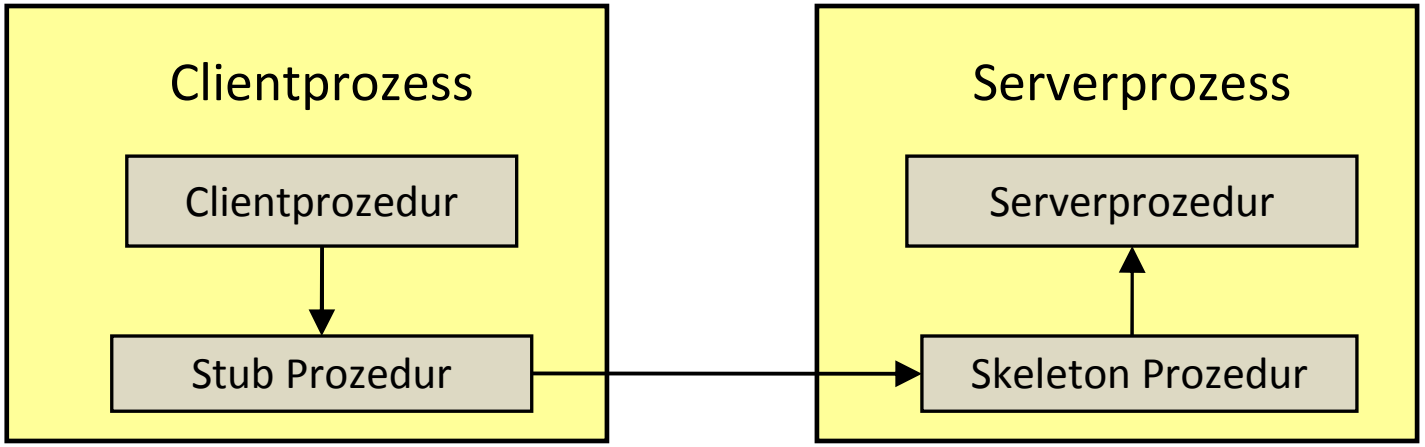
- Middleware
  - Remote Procedure Calls (RPC) für prozedurale Sprachen
  - CORBA ORB / IIOP als objektorientierte Variante
  - RMI und andere proprietäre Verfahren
  - SOAP/WSDL als standardisierte XML-basierte Variante
- Implementierung
  - Stubs und Skeletons explizit codieren und generieren
  - Container-gesteuert hinzukonfigurieren

Marshalling und Unmarshalling

- Unter Marshalling versteht man die Transformation von Daten in ein Übertragungsformat.
- Unter Unmarshalling versteht man die Rücktransformation eines Zeichen- oder Bytestroms in Daten einer konkreten Programmiersprache.
- Verantwortlich für die Aufgabe sind Stubs und Skeletons der Middleware.
- Ebenfalls möglich in speziellen Infrastrukturen: Verschicken von Objekten mit Methoden (erfordert gleichartige Laufzeitumgebung).

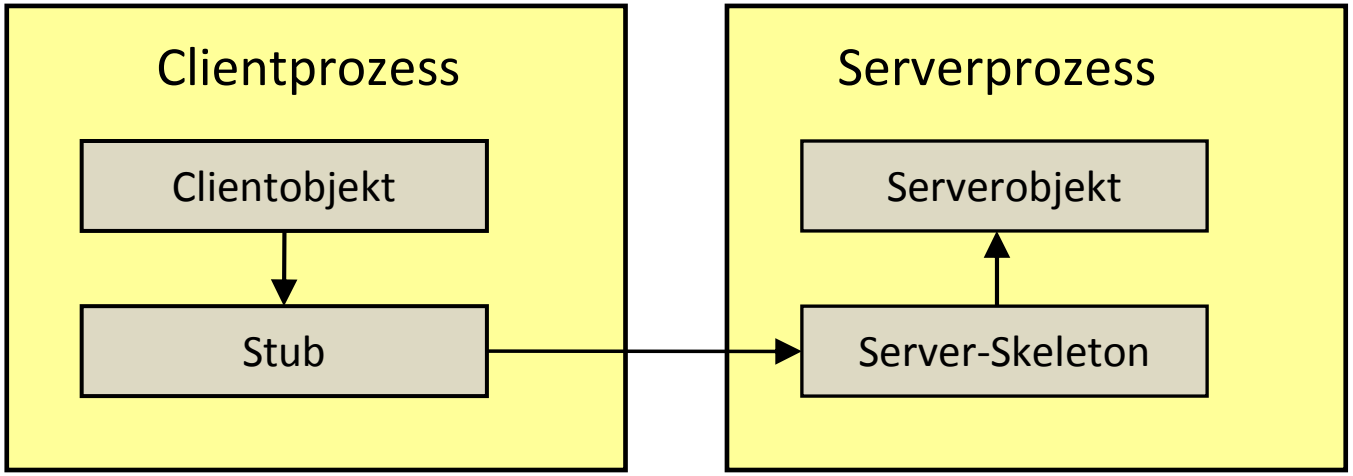


Der entfernte Prozeduraufruf (RPC)



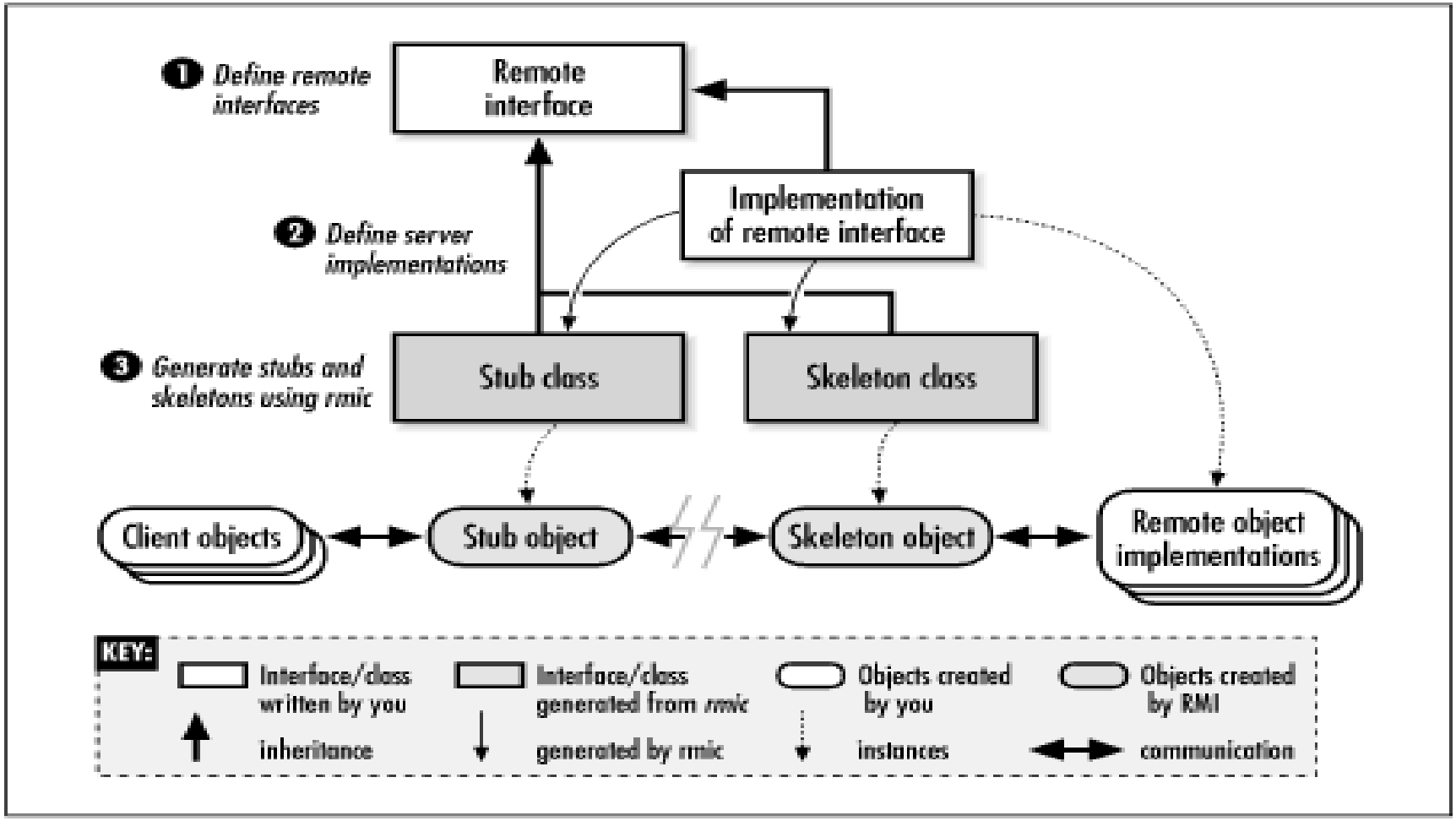
- Ermöglicht einem Clientprozess den entfernten Aufruf einer Prozedur auf einem Serverprozess.
- Die Kommunikation erfolgt nach dem Anfrage-/Antwort-Prinzip.
- Das Modell setzt auf verschiedenen Hilfsprozeduren auf.

Der entfernte Methodenaufruf (RMI)



- Ermöglicht es einem Objekt, Methoden auf einem entfernten Objekt aufzurufen.
- Die Kommunikation erfolgt nach dem Anfrage-/Antwort-Prinzip.
- Modell setzt auf dem Proxy Pattern auf.

Entwicklungsprozess bei „manuellem“ RMI



Konfiguration eines RMI-Servers mit Spring-Container

Lokaler Service als POJO

Lokaler Service

```
<bean id=„libraryService“
   class=„LibraryServiceImpl"> … </bean>
```

Erstellung RMI-Server durch Konfiguration

Server-Mechanismus

Name des Remote Service

Zugrundeliegender lokaler Service

Zugrundeliegendes lokales Interface

RMI-technische Konfiguration (Port)

```
<bean class="org.springframework.
   remoting.rmi.RmiServiceExporter">
   <property name="serviceName„
        value=„RMILibraryService"/>
   <property name="service„
        ref=„libraryService"/>
   <property name="serviceInterface"
        value="example.LibraryService"/>
   <property name="registryPort„
        value="1199"/>
</bean>
```

Konfiguration des zugehörigen Clients mit Spring

Client erhält LibraryService …

```
public class Client {
  private LibraryService l;
  public void setLibraryService(
    LibraryService ls) { l = ls; } …
```

… via Dependency Injection von Container

```
<bean class="example.Client">
  <property name="libraryService"
    ref="libraryService"/>
</bean>
```

Client-Mechanismus

Zugrundeliegendes Interface

RMI-technische Konfiguration (Server)

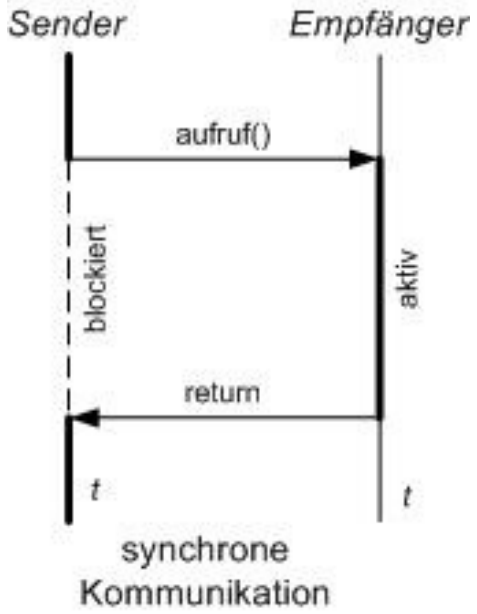
```
<bean id="libraryService"
  class="org.springframework.
    remoting.rmi.RmiProxyFactoryBean">
 <property name="serviceInterface"
    value="example.LibraryService"/>
 <property name="serviceUrl" value=
    "rmi://HOST:1199/RMILibraryService"/>
</bean>
```

CORBA ORB als weiteres Beispiel für Middleware

- Ortstransparenz
  - Entfernte Objekte über netz-eindeutige Identifikatoren bereitstellen
  - Transparente Vermittelung von Aufrufen zwischen Objekten in verschiedenen Prozessen an unterschiedlichen Orten
  - Kontexte weiterreichen (Transaktionen, Sicherheit)
- Statische und dynamische Methodenaufrufe:
  - Methodenaufrufe statisch beim Kompilieren definieren oder dynamisch zur Laufzeit „entdecken"
- Freie Wahl von Programmiersprachen:
  - Sprache der Aufrufe unabhängig von Implementationssprache des Servers durch Trennung von Schnittstelle und Implementierung
  - Sprachenunabhängige Datentypen
- Selbstbeschreibendes System:
  - Interface Repository als Verzeichnis der Schnittstellen, die von Servern angebotene Methoden und deren Parameter beschreiben (in IDL)

Synchrone Kommunikation

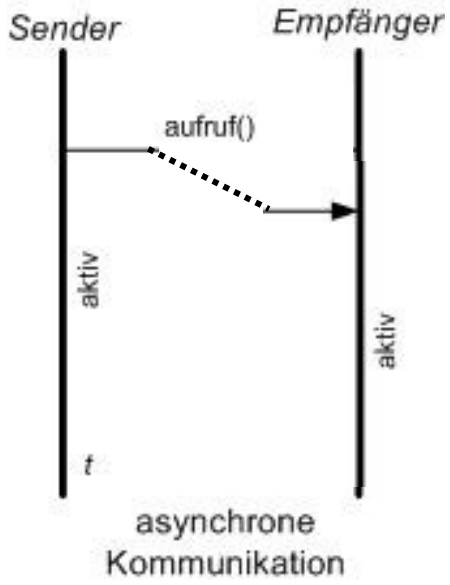
- Sender und Empfänger blockieren beim Ausführen der Operationen zum Senden bzw. Empfangen.
- Eigenschaften:
  - Enge Kopplung von Sender und Empfänger.
  - Empfänger muss bereitstehen: Wenig robust, nicht gut skalierbar.
  - Einfaches Programmiermodell für Anwendungs-programmierer (keine parallele Ausführung).
  - Hohe Abhängigkeit insbesondere im Fehlerfall.
- Voraussetzung:
  - Sichere und schnelle Netzverbindungen sind verfügbar.
  - Empfangender Prozess ist verfügbar.



synchrone Kommunikation

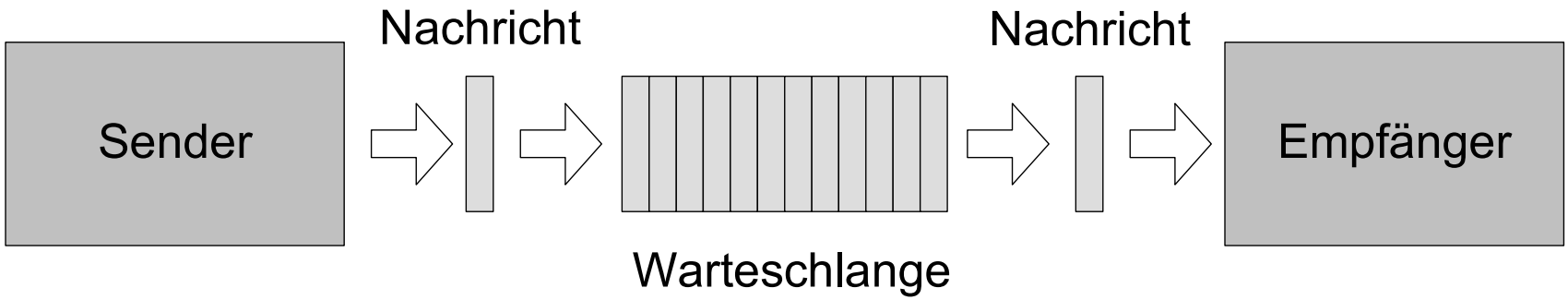
Asynchrone Kommunikation

- Sender ist nicht blockiert; der Prozess kann nach dem Senden der Nachricht sofort weiterarbeiten.
- Antworten sind optional:
  - Der Sender erhält bei Gelegenheit das Ergebnis asynchron.
  - Der Sender holt sich bei Gelegenheit aktiv das Ergebnis.
- Eigenschaften:
  - Lose Koppelung von Prozessen.
  - Empfänger muss beim Senden nicht empfangsbereit sein.
  - Komplizierter zu implementieren, dafür robuster und skalierbarer.
- Realisierung über Warteschlangen (Message Queuing).
- Es gibt unterschiedliche MEPs (Message Exchange Patterns).



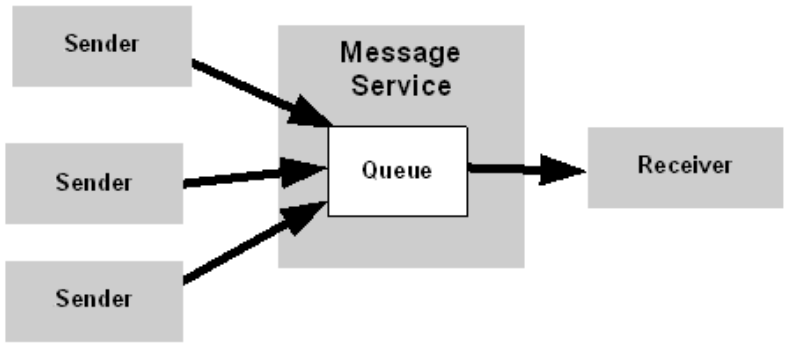
Message Queuing, Message Passing, Messaging

- Message Queuing ist ein Programmiermodell zum asynchronen Transport von Daten zwischen Prozessen.
- Die Daten werden innerhalb von Nachrichten (Messages) übertragen.
- Message Passing arbeitet auf der Basis von Warteschlangen:
  - Nachrichten werden vom Sender in eine Warteschlange gelegt
  - und werden an den Empfänger weitergeleitet.
- Die Warteschlange arbeitet nach dem FIFO-Prinzip.

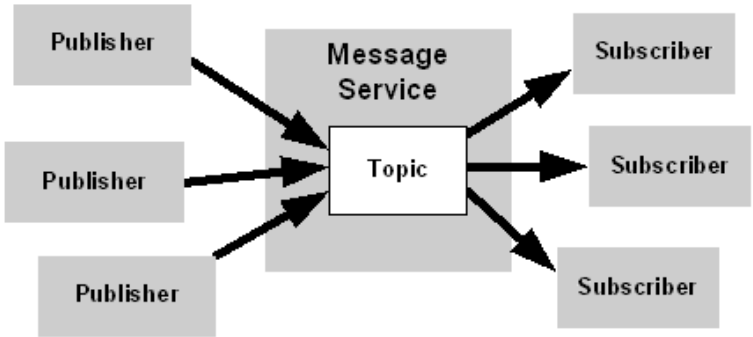


Message-Queuing-Varianten

- Point-to-point: Die Nachrichtenübertragung findet zwischen zwei festgelegten Servern statt. Message Exchange Patterns sind
  - One-Way-Pattern: Sender schickt Nachricht an Empfänger.
  - Request-Reply-Pattern: Analog zu synchroner Kommunikation (Sender wartet nach Senden auf Antwort), über zwei One-Way-Nachrichten realisierbar.
  - Request-Callback-Pattern: Sender spezifiziert Callback-Funktion, die zum Empfang aufgerufen wird und arbeitet erstmal weiter.
- Broadcasting: Eine Nachricht wird an viele Server versendet.
  - Publish-Subscribe-Pattern: Empfänger melden sich explizit an.



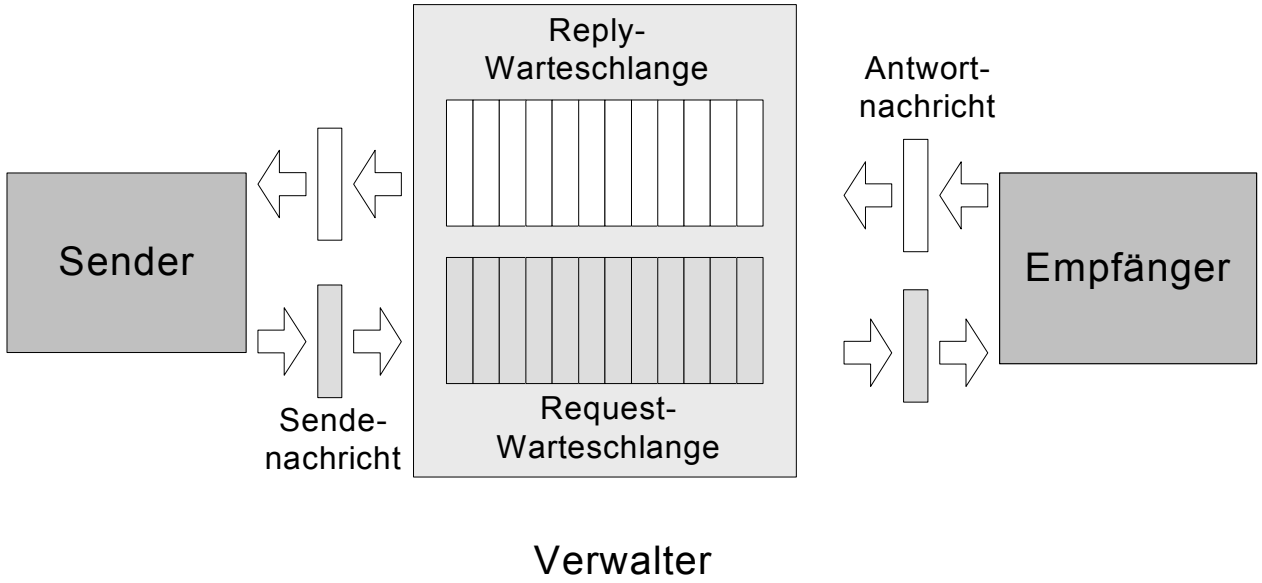
Point to Point



Publish/Subscribe

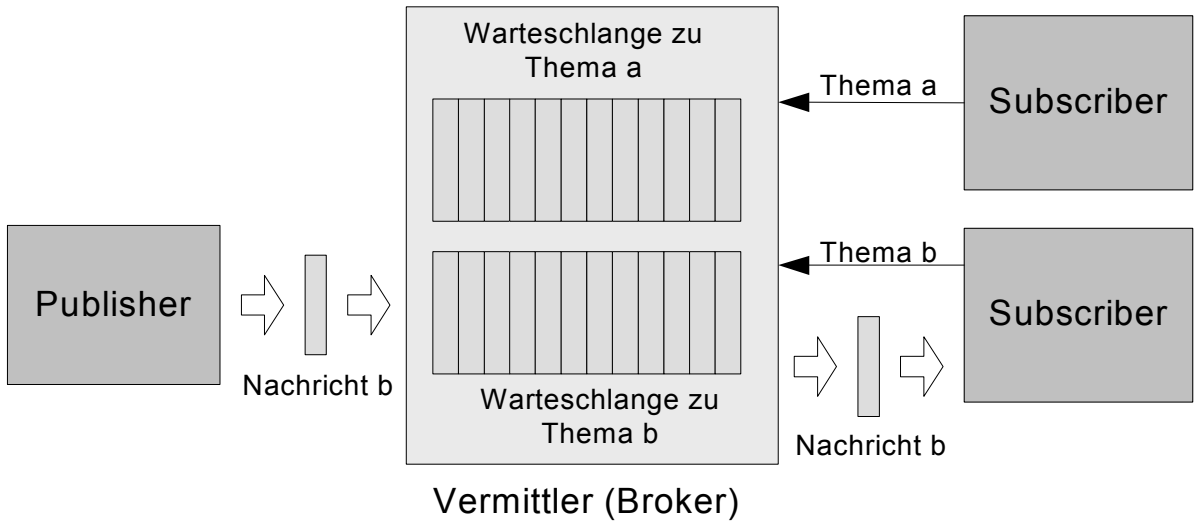
Request-Reply-Modell

- Das Verschicken einer Nachricht (Request) und das Empfangen einer Antwortnachricht (Reply) erfolgen als eine Einheit: Nur wenn beide erfolgreich ablaufen, war auch die Kommunikation erfolgreich.
- Ermöglicht eine (fachlich gesehen) synchrone Kommunikation über eine (technisch gesehen) asynchrone Middleware.
- Eine mögliche Umsetzung erfolgt über unterschiedliche Warteschlangen für Request- und Reply-Nachrichten.



Publish-Subscribe-Modell

- Prozesse erhalten Rollen als Publisher bzw. Subscriber.
- Subscriber abonnieren Nachrichten zu einem Thema.
- Publisher veröffentlichen Nachrichten zu einem Thema.
- Broker übernehmen die Vermittlung.
- Eine mögliche Realisierung erfolgt durch themenspezifische Warteschlangen.



Beispiel für Empfangen einer asynchronen Nachricht mit JMS

Client registriert sich als Empfänger …

… und spezifiziert Callback-Objekt

asynchron (vom Thread des JMS-Servers) gerufene Callback-Methode

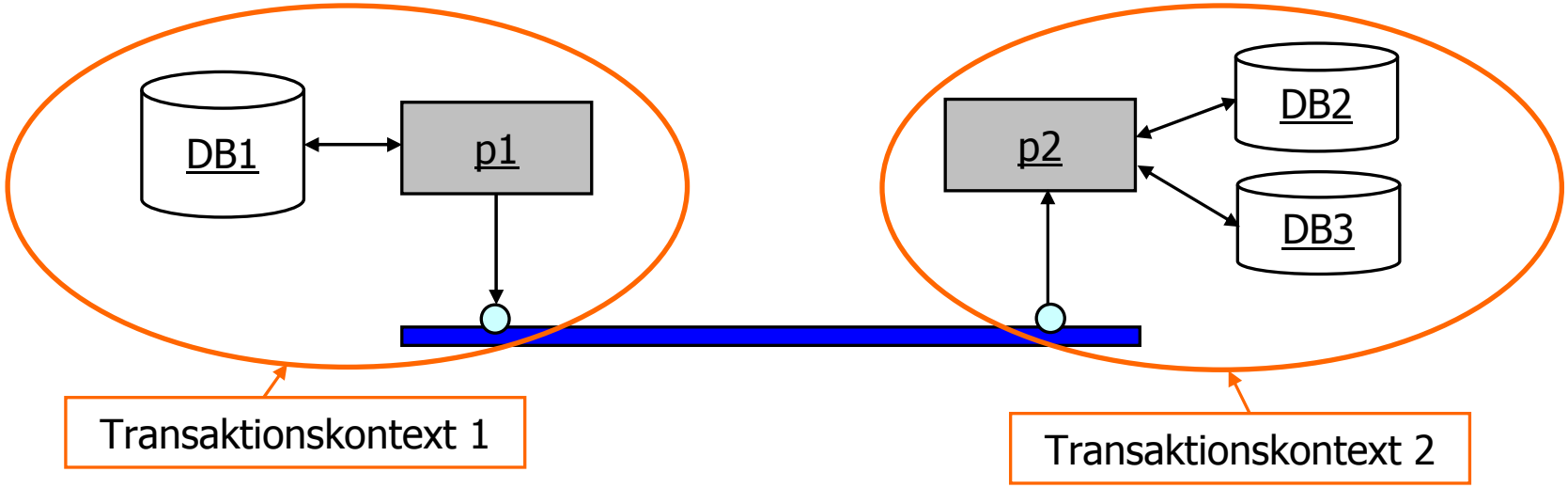
```
Class Client {
   …
   subscriber = jmsSession.
     createTopicSubscriber(topic);
   subscriber.setMessageListener(
     new Listener());
   …
}

class Listener implements MessageListener {
   void onMessage(Message msg){
     if(msg instanceof TextMessage) {
        … handle TextMessage …
     }
}
```

Unterschiedliche Nachrichtenarten möglich z.B. Text, XML, Objekt

Asynchrone entfernte Kommunikation über Nachrichten

- Queues und Topics können auf Wunsch persistent gehalten werden
- Transaktionales Messaging stellt sicher, dass Nachrichten ankommen



- Wann asynchrone Kommunikation in betrieblichen Infosystemen?
  - Unidirektionale Nachrichten → Notifikation von Clients
  - Erhöhte Parallelität → Sender muss nicht warten
  - Erhöhte Performanz → Tuning von Message Queues einfach, Vermeidung des schwergewichtigen Two-Phase-Commit

Zwischenzustände und Kompensation

- Nachteile des asynchronen transaktionalen Messaging: **ACID** verletzt
  - Nicht **A**tomar und nicht **I**soliert: Fachlich relevante „Zwischenzustände" können entstehen und müssen beim Anwendungsentwurf berücksichtigt werden.
  - Rollback von Geschäftsvorfällen durch Entwickler zu lösen: Kompensations-Aktionen müssen entworfen werden.
  - Beispiel: Überweisung eines Geldbetrags von Quellkonto zu Zielkonto, Quell- und Zielkonto werden auf zwei Systemen verwaltet.

- Möglichkeit 1:
  - TA1: Vorfall anstoßen
    - Auf Zielsystem Betrag auf Zielkonto verbuchen.
    - Abbuchungsnachricht für Quellkonto verschicken.
  - TA2: Nachricht bearbeiten
    - Auf Quellsystem Abbuchungs-nachricht empfangen.
    - Betrag von Quellkonto abbuchen.

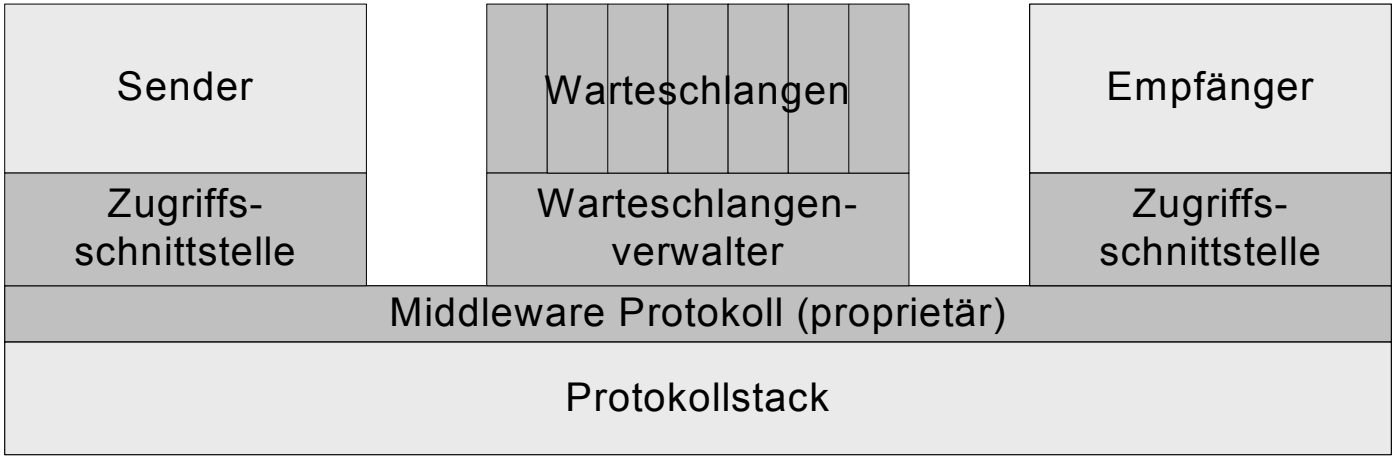
- Möglichkeit 2:
  - TA1: Vorfall anstoßen
    - Auf Quellsystem Betrag X von Quellkonto abbuchen.
    - Verbuchungsnachricht für Zielkonto verschicken.
  - TA2: Nachricht bearbeiten
    - Auf Zielsystem Verbuchungs-nachricht empfangen.
    - Betrag X auf Zielkonto verbuchen.

Middleware zum Messaging

- Oft als Integrations-Middleware verwendet
  - Systeme bieten Adapter oder folgen Standards
  - Persistente Speicherung der Nachrichten in Standard-Datenbanken
- Implementierungsvarianten
  - als Dienst in Application Server integriert
  - als Standalone-System
  - als Enterprise Service Bus (ESB)
- Bekannteste Standards und Vertreter
  - CORBA Event Service, CORBA Notification Service
  - Java Messaging Service (JMS)
  - Microsoft Message Queue
  - IBM MQSeries
  - Sopera ESB

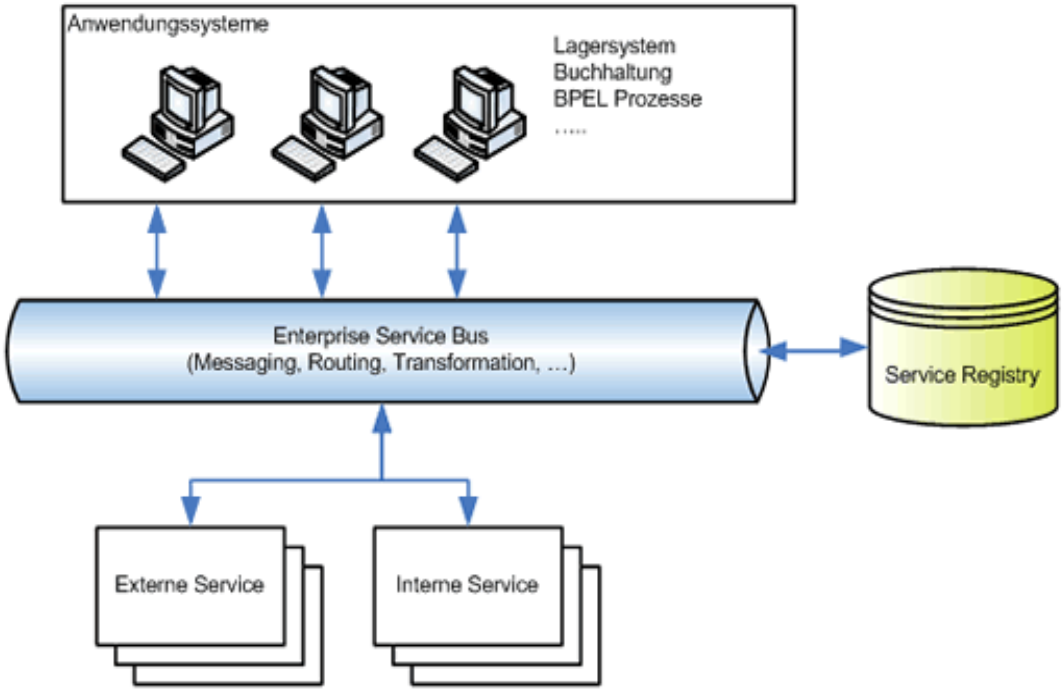
Message Oriented Middleware (MOM)

- Ist die Middleware-Technologie zum Message-Queuing-Programmiermodell.
- MOM bietet neben der rein asynchronen Kommunikation weitere Mechanismen und Dienste:
  - Unterstützung der verschiedenen Messaging-Modelle
  - Warteschlangenverwaltung, Verbindungsmanagement
  - Quality-of-Service Zusicherungen (QoS)

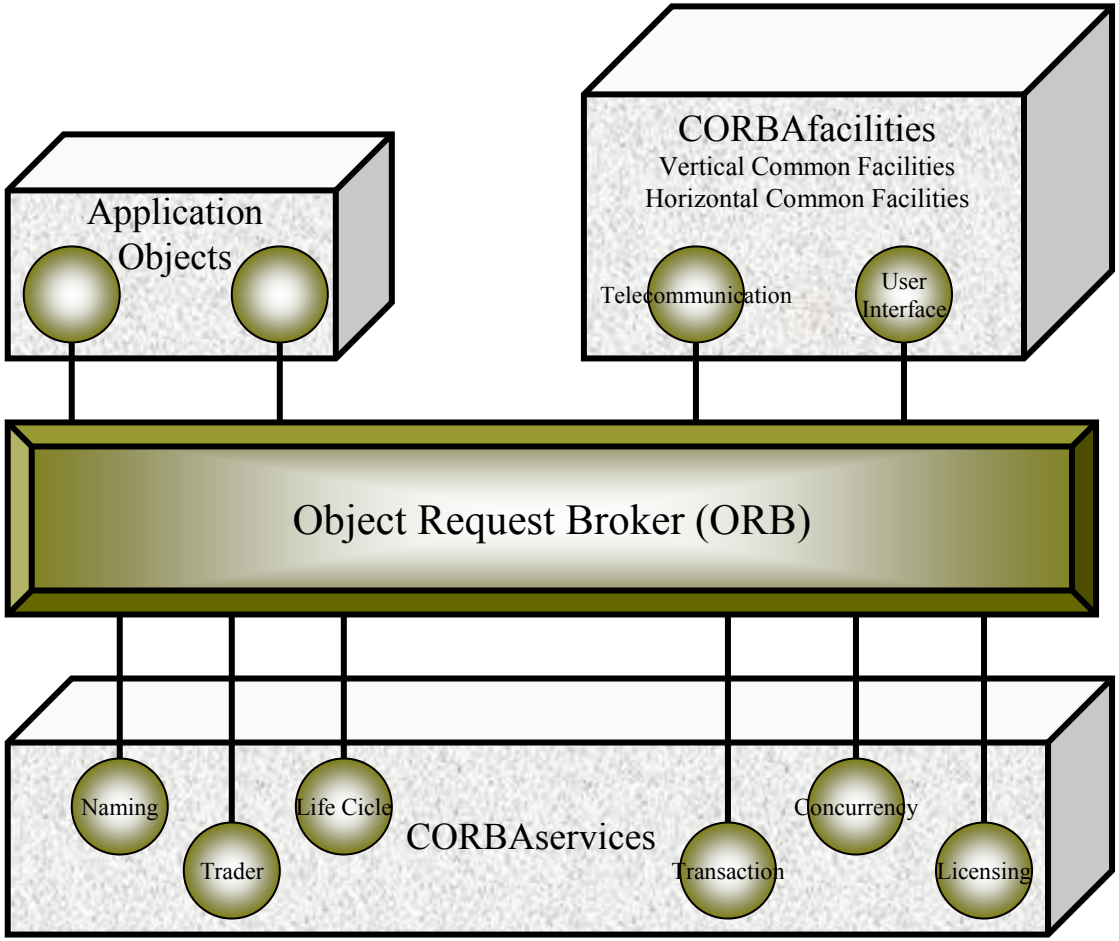


ESBs als allgemeine Kommunikationsinfrastruktur

- ESBs sind als grundlegende Integrationsinfrastruktur für die Verbindung von Anwendungen und Services gedacht
- Unterstützung unterschiedlicher Protokolle und Schnittstellen
  - synchrone Service-Aufrufe
  - asynchrones Messaging
- Spezifikation von Regeln für Routing und Daten-transformation
- Verwaltung der Services und Regeln in einem Repository
- Überwachung, Proto-kollierung, Debugging
- Zusatzdienste wie Workflow-Management



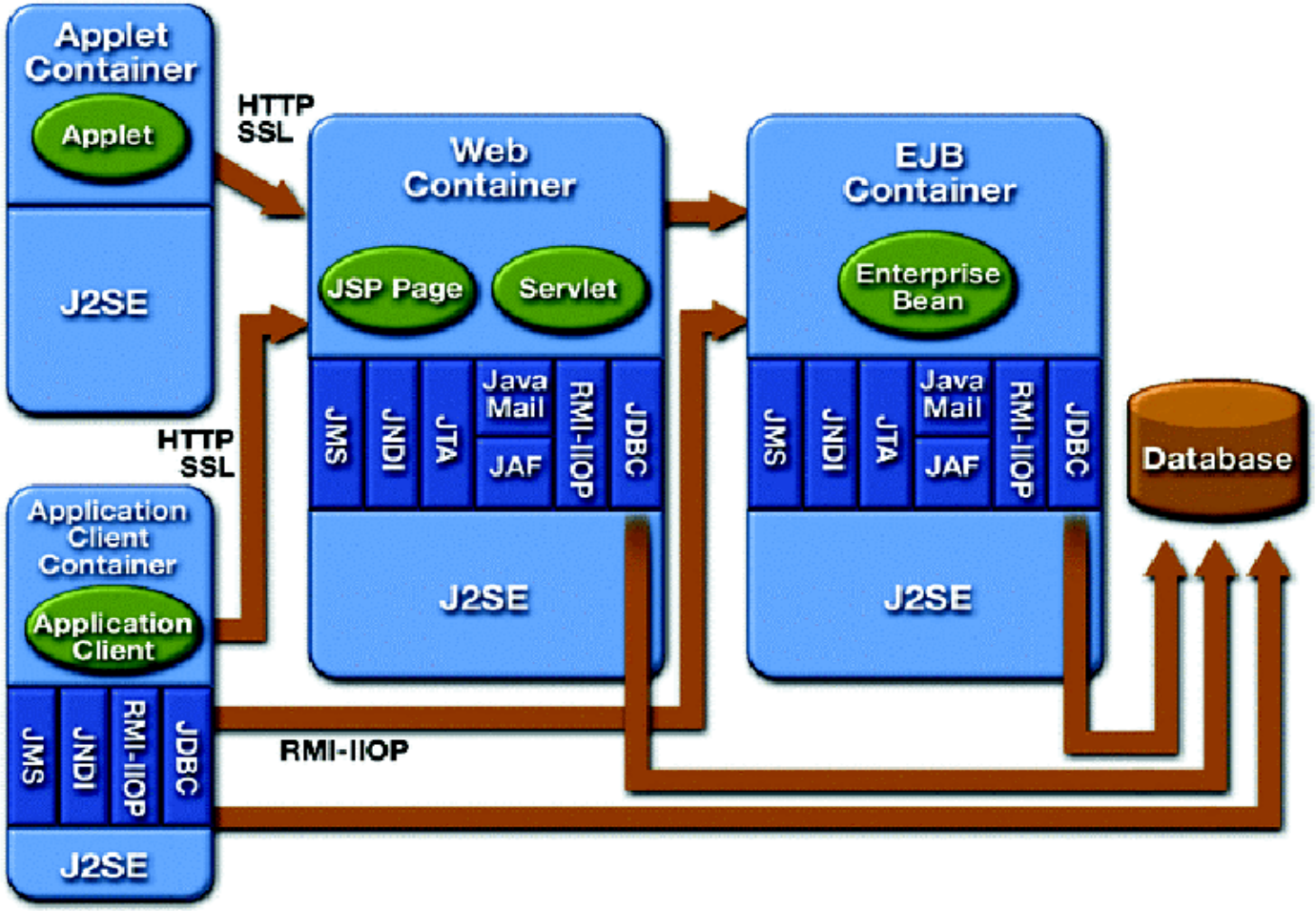
Object Management Architecture und CORBA Services



Basisdienste bei CORBA

- **Life Cycle Service** stellt Operationen zur Erzeugung, zum Kopieren, Verschieben und Löschen von Objekten bereit.
- **Collection Service** ermöglicht es, Objekte in Gruppen zu manipulieren. (Arrays, Bäume, Stacks, ...)
- **Relationship Service** ermöglicht die Definition von Beziehungen zwischen Objekten.
- **Property Service** erlaubt es, einem Objekt dynamisch eine Eigenschaft zuzuweisen (z. B. einen Titel oder ein Datum).
- **Concurrency Control Service** ermöglicht Objekten den Zugriff auf gemeinschaftlich genutzte Ressourcen mit Hilfe von Sperren (Locks).
- **Security Service** stellt Sicherheitsmechanismen bereit. Er unterstützt z. B. Authentifizierung und Zugriffskontrolle.
- **Time Service** stellt z. B. Interfaces zur Verfügung, um sich in einer verteilten Umgebung miteinander zu synchronisieren oder zeitabhängige Ereignisse zu definieren.
- **Licensing Service** erlaubt es, die Nutzungsdauer von Komponenten zu messen und entsprechend abzurechnen.

Basisdienste bei J2EE



Literaturhinweise

- [Bi02] Adam Bien: *J2EE Patterns*, Addison-Wesley, 2002.
- [EJB3] Sun Microsystems: *J2EE Website*. http://java.sun.com/j2ee, 2005.
- [jBPM] *jBPM WfE Homepage*, http://www.jboss.com/products/jbpm
- [Sie02] J. Siegel, *An Overview Of CORBA 3.0*, Object Management Group, 2002.
- [Spr05] Rob Harrop, Jan Machacek: *Pro Spring*, apress, 2005.
- [Spring] *Spring Framework Homepage*, http://www.springframe-work.org
- [WfMC] *Workflow Management Coalition Homepage*, http://www.wfmc.org