06. Technische Grundlagen betrieblicher IS

SEBA Bachelor, Florian Matthes, WS17/18

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Vorlesungsgliederung

1. IT-Unterstützung betrieblicher Anwendungen
2. Requirements Engineering (Vertiefung)
3. Praktische Übung konzeptionelle Modellierung mit UML
4. Aufwandschätzung
5. Konfigurationsmanagement
6. Technische Grundlagen betrieblicher IS
7. Verteilung
8. Persistenz
9. Betrieb und Wartung
10. Gastvortrag Spezialthema
11. Unterstützung von Geschäftsprozessen

6 – Grundlegende Architekturkonzepte für betriebliche Informationssysteme

6.1 Schichtenarchitekturen betrieblicher Anwendungen

- Motivation und Grundlagen
- Verteilung und Skalierung
- Elemente der JEE Schichten-Architektur

6.2 Bibliotheken und Frameworks

6.3. Aspektorientierung und Java - Annotationen

Wiederholung: Schichtenarchitektur

Schichten partitionieren die Komponenten einer Software und verringern dadurch deren Komplexität.

Innerhalb der Schicht herrscht **hohe Kohäsion**

Zwischen den Schichten soll die **Kopplung gering sein**

Zwei Arten von Schichtenarchitekturen

Strikt	Offen
Eine Schicht darf nur auf die direkt unter ihr liegende Schicht zugreifen	Eine Schicht darf auf alle unter ihr liegenden Schichten zugreifen
Vorteil: Einfachere Wartung	Vorteil: Höhere Performanz

→ Eine Schicht darf niemals auf eine über ihr liegende zugreifen!

Drei typische Schichten betrieblicher Anwendungen

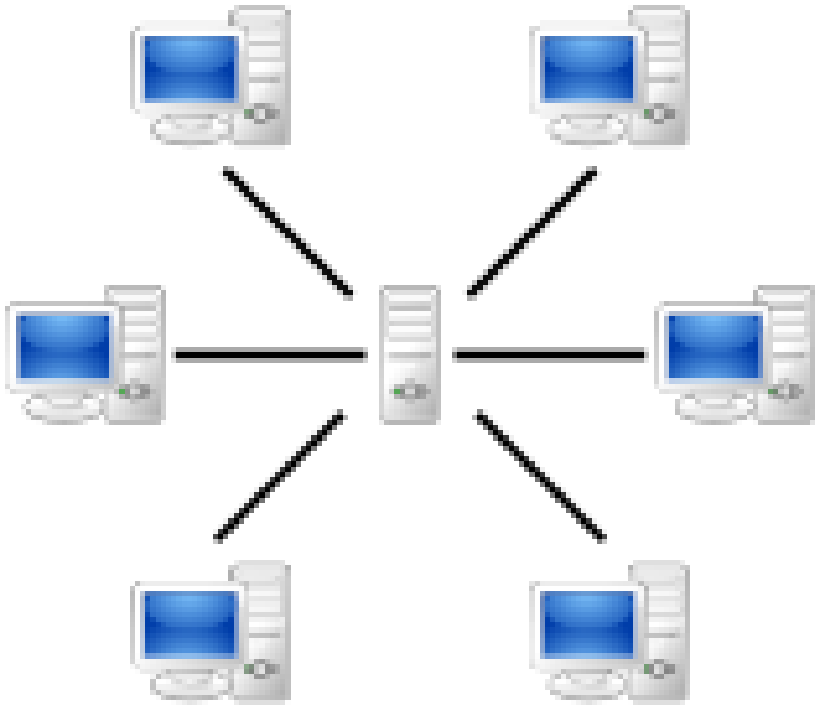
Die folgenden drei Schichten lassen sich praktisch allen betrieblichen Anwendungen finden

Beispiel	Schicht	Technologie
Tabelle Diagramm Menü	**Präsentations-Schicht**	 
Währungsrechner Kontoübersicht Login	**Geschäftslogik-Schicht**	EJB 3 Spring
Kunde Konto	**Datenbank-Schicht**	HIBERNATE MySQL

Verteilte Softwaresysteme - Grundbegriffe

Client-Server-Architektur

- Client-Server ist eine Architektur für verteilte Softwaresysteme
- Darin gibt es zwei Komponenten (Rollen): Client und Server
- Der Client fragt einen Dienst über ein Netzwerkprotokoll beim Server an und erhält eine Antwort
- Ein Server bedient mehrere Clients
- Ein Server kann selbst ein Client anderer Server sein
- Zwischen Client und Server können Vermittlungsdienste (Broker) bestehen.



Web Server

- Stellt seinen Clients (Web Browser) statische oder dynamisch generierte Inhalte bereit (HTML, CSS, Dateien, Bilder)
- Verwendete Protokolle: HTTP, HTTPS, FTP, …
- Weitere Aufgaben:
  - Ressource-Management
  - Zugriffsbeschränkung
  - Cookie-Verwaltung
  - Skriptausführung (z.B. PHP oder Servlets)
  - Caching

Implementierungen

- Apache Tomcat
- Jetty
- Internet Information Services (Microsoft)

→ Moderne Web Server (z.B. Tomcat) können auch Java Code ausführen.

Anwendungsserver (Application Server)

- Unterscheiden sich nach Typ
  - Java EE, .NET, SAP Web Application Server

- Verwendete Protokolle: beliebige, ermöglicht auch Methodenaufrufe
- Weitere Aufgaben:
  - Nachrichtenversand (Messaging)
  - Authentifizierung
  - Asynchrone Kommunikation
  - Kapselung von Datenbanken (Programmierer muss DB nicht kennen)

Implementierungen

- WebSphere
- Jboss Wildfly
- GlassFish

[JW02] Sintes, T: App server, Web server: What's the difference? http://www.javaworld.com/javaqa/2002-08/01-qa-0823-appvswebserver.html (zuletzt aufgerufen am 18.10.2012).

Datenbankserver

Datenbankserver (Software)

- Beinhaltet eine Datenbanksoftware
- Kann verschiedenen Kategorien angehören: Relational, XML, NoSQL
- Bietet Tools zum Administrieren
- Im betrieblichen Umfeld wird meistens auf relationale Datenbanken gesetzt

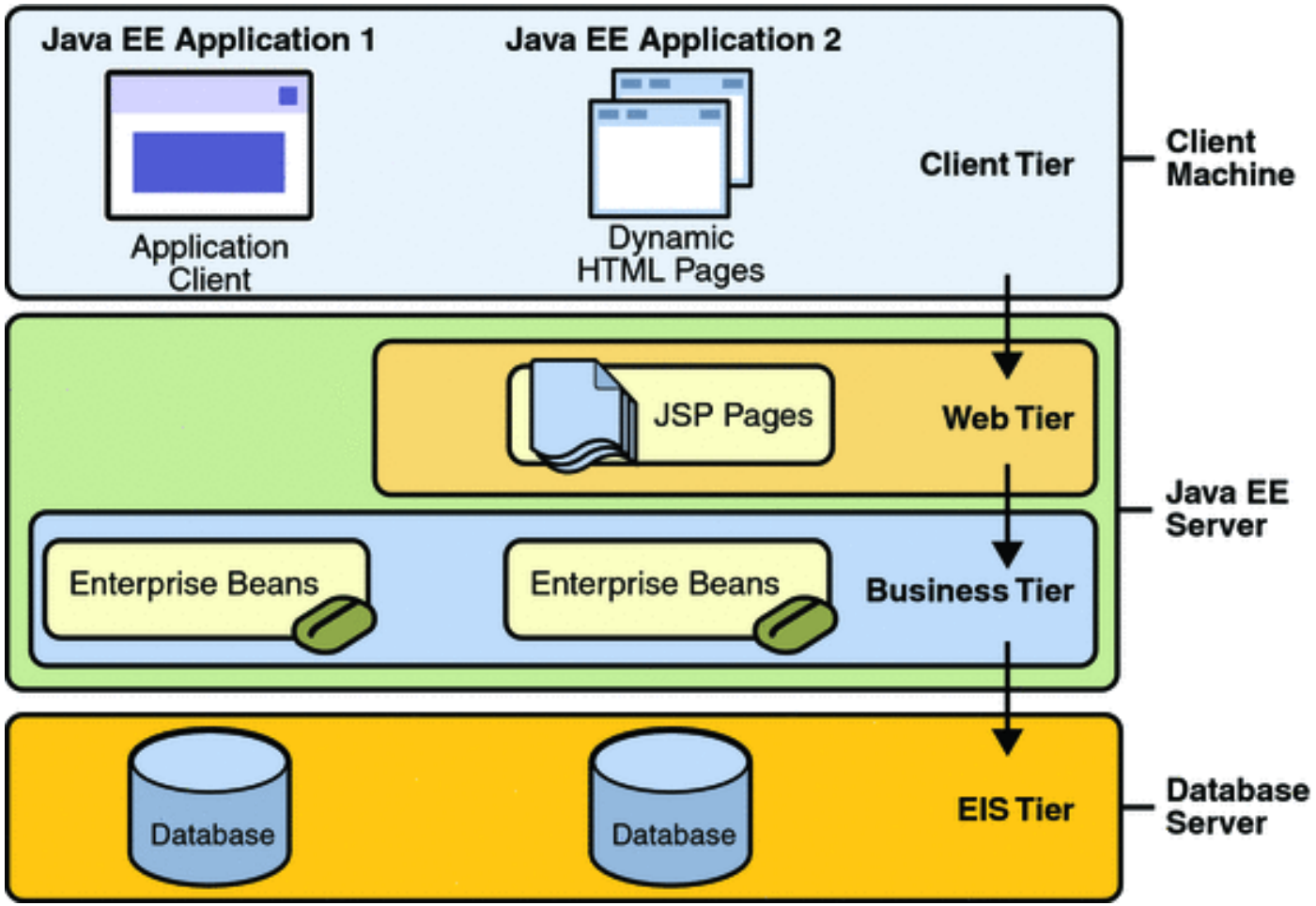
Datenbankserver (Hardware)

- Datenbankserver laufen meist auf einer eigenen Maschine
- Sie nehmen daher die Rolle des Servers im Client-Server Modell ein

Übungsbetrieb

- Wildfly built-in H2 database (relational)
- Kein separater Datenbank-Server
- Nicht für den Produktiv-Betrieb geeignet

JEE Architekturüberblick



[Quelle: Oracle]

JEE ist eine Sammlung vieler Spezifikationen

Web Application Technologies

- Java API for JSON Processing
- Java Servlets 3.1
- Java Server Faces 2.2

Enterprise Application Technologies

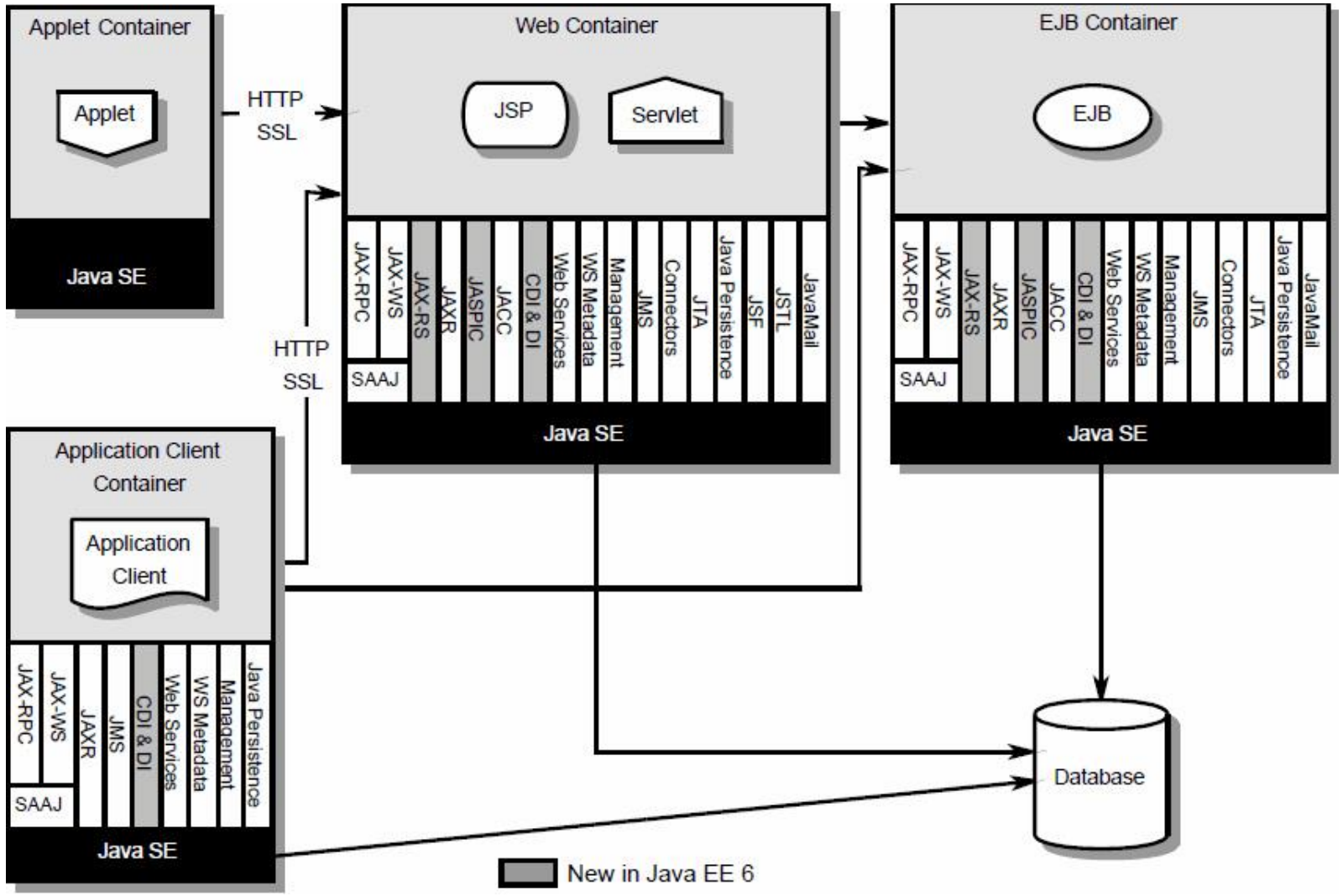
- Dependency Injection for Java 1.0
- Enterprise Java Beans 3.2
- Java Persistence 2.1
- Java Message Service API 2.0

Web Service Technologies

- Java API for RESTful Web Services 1.1
- Java APIs for XML-based RPC

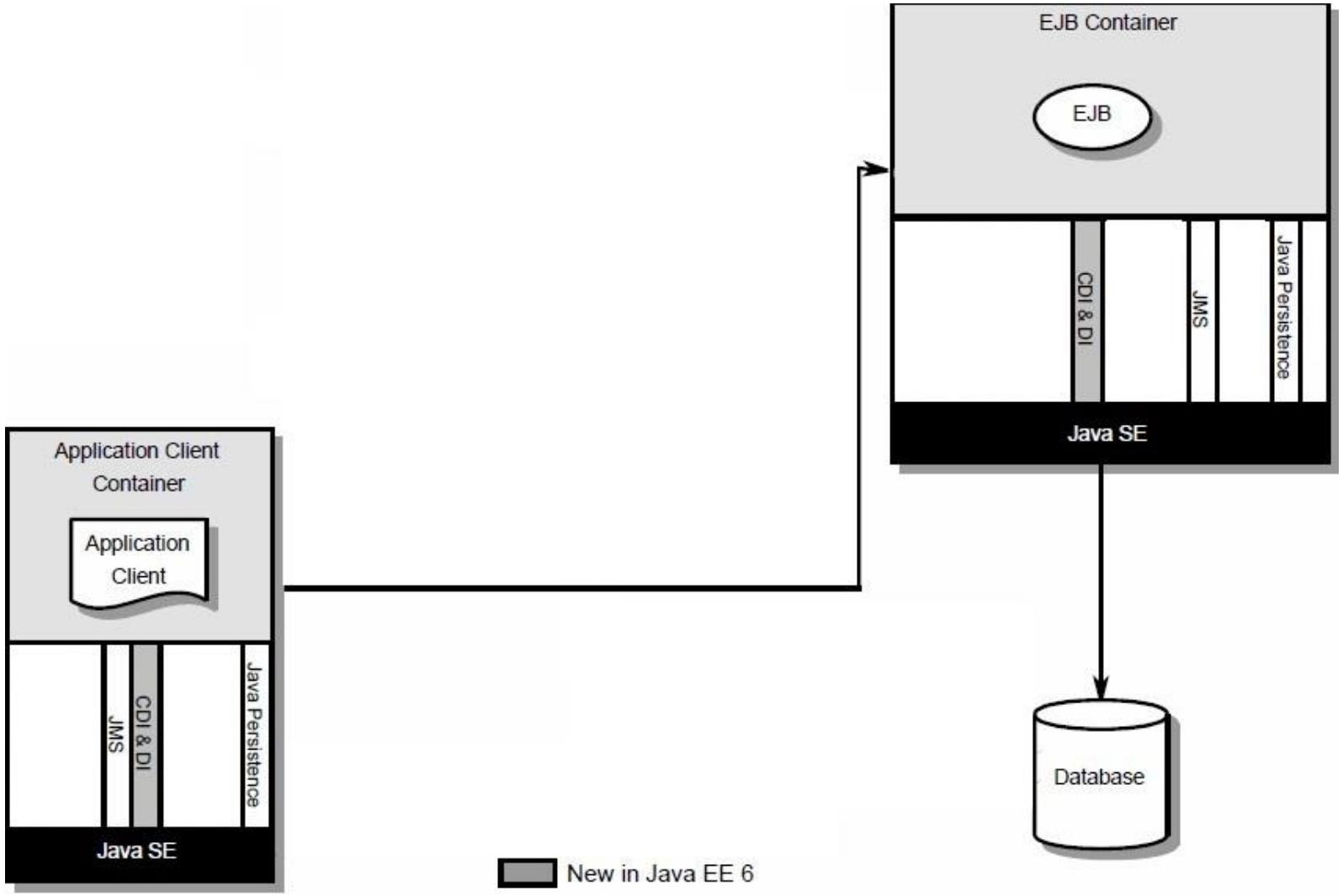
→ Fokus der Vorlesung und Einsatz in der Übung: DI, EJB, JPA, JMS

JEE Architekturüberblick im Detail



[Quelle: Oracle]

JEE Architekturüberblick – Fokus SEBA

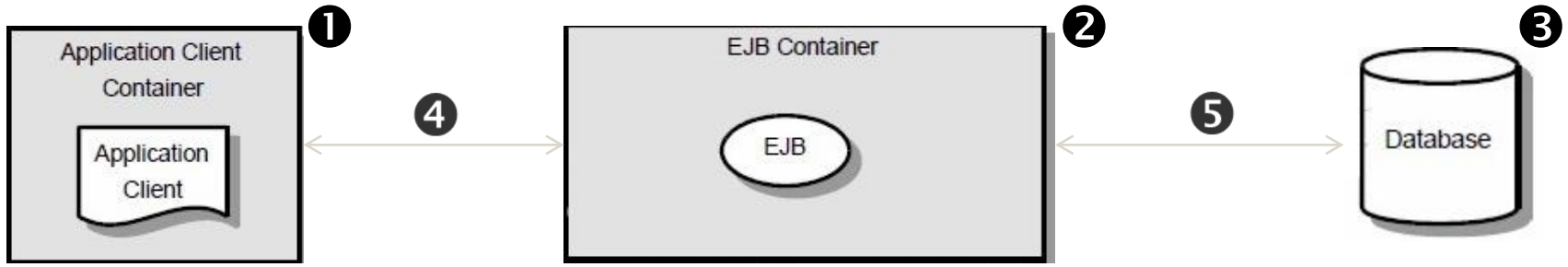


JEE Architekturüberblick – Beispiel

Es soll eine Java EE Anwendung entwickelt werden, die es Bankangestellten einer international agierenden Bank mit mehr als 20.000 Mitarbeitern ermöglicht Kontobewegungen in einer zentralen Datenbank zu erfassen.

Komponenten und ihre Implementierung

1. **Client:** Java Fat Client auf Kommandozeilen-Basis
2. **Server:** EJBs und JMS
3. **Datenbank:** Oracle
4. **Verbindung zwischen Client und Server:** wird vom App.Server bereit gestellt
5. **Verbindung zwischen Server und Datenbank:** JPA



→ Ohne JEE müsste alles immer wieder selbst implementiert werden!

6 – Grundlegende Architekturkonzepte für betriebliche Informationssysteme

6.1 Schichtenarchitekturen betrieblicher Anwendungen

6.2 Bibliotheken und Frameworks

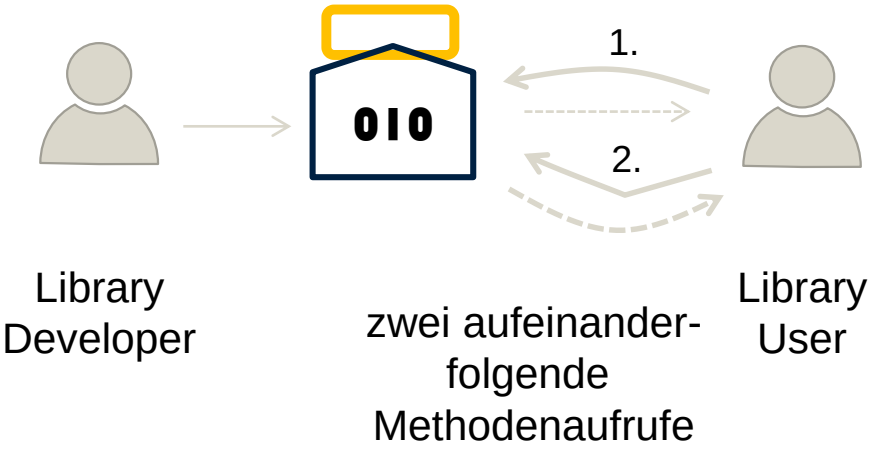
- Bibliotheken
- Frameworks
- Inversion of Control

6.3. Aspektorientierung und Java - Annotationen

Definition Bibliothek (Library)

Eine Library ist

> „*eine **wiederverwendbare Softwarekomponente**, welche aus einer Vielzahl von Klassen bestehen kann. Die **Funktionen** der Bibliothek können über deren Application Programming Interface (API) benutzt werden“.*



API (Anwendungsprogrammierungsschnittstelle)

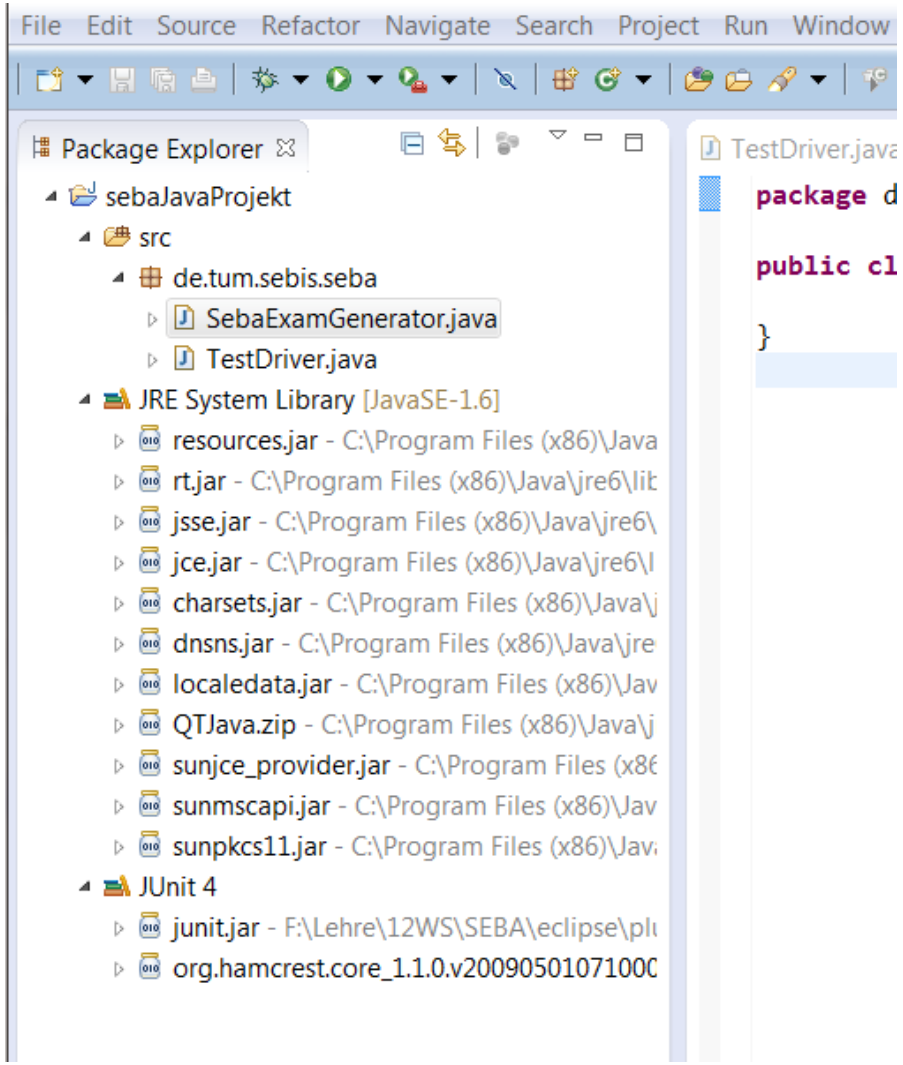
- Stellt eine Programmierschnittstelle dar
- Die Reihenfolge, in der die bereitgestellten Funktionen aufgerufen werden, bestimmt der Benutzer

Beispiele

- Log4J (Protokollierung), JDBC (Datenbankzugriff)
- dom4j (Navigation durch XML Dateien in Java)

Realisierung von Bibliotheken in Java

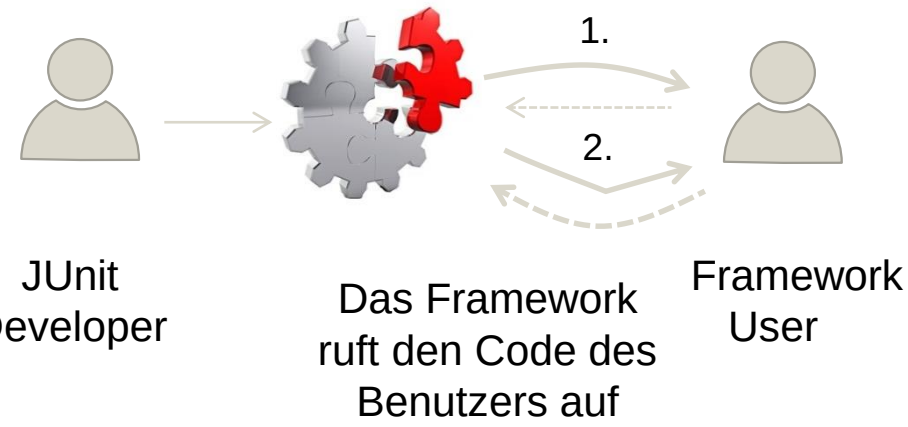
- In Java liegen Bibliotheken in Form von .jar Dateien vor. Dies sind im Wesentlichen .zip Dateien, die jedoch eine vorgegebene Ordnerstruktur besitzen (z.B. META-INF Ordner).
- Damit Bibliotheken in einem Java Projekt verwendet werden können, müssen diese dem Class Path (Klassenpfad) hinzugefügt werden (siehe Bild rechts).



Definition Framework

Eine Framework ist

„*ein **halbfertiges Softwaresystem**, welches aus einer Vielzahl von aufeinander abgestimmten Softwarekomponenten besteht, aus denen mit **relativ geringem Aufwand** ein **angepasstes Softwaresystem** erstellt werden kann*“.



Frameworks bieten also

- eine Basisarchitektur für ein Softwaresystem,
- einen hohen Grad der Wiederverwendung und
- eine gegebene Reihenfolge von Funktionen die der Benutzer erweitern kann,
- wodurch die grobe Verarbeitungslogik vorgegeben wird.

Frameworks für verschiedene Verwendungszwecke

- Grafische Oberflächen: WPF (Windows Presentation Foundation), Swing
- Web-Entwicklung: Apache Wicket, ASP.NET

[HN09] Hansen; Neumann (2009): Wirtschaftsinformatik 1, UTB Verlag, 10. Auflage.

Framework Beispiel: JUnit 4.x

- JUnit ist ein Framework zur Unterstützung bei Softwaretests
- Das Ausführen von Tests erfolgt durch einen Methodenaufruf in der Klasse *TestRunner*
- JUnit führt daraufhin alle Testklassen aus, die der Framework-Benutzer entwickelt hat
- Auf Basis der entstehenden Testresultate wird eine Übersicht generiert

→ **Das Framework bestimmt, zu welchem Zeitpunkt Code des Benutzers ausgeführt wird.**

Inversion of Control (IoC)

Auch bekannt als Hollywood Prinzip: **"Don't call us, we'll call you!“**

IoC unterscheidet ein **Framework** von einer **Library**

- Der Programmierer registriert seinen Code beim Framework (Klassen-Konfiguration, Subklassendefinition, Methoden-Annotationen, Datenbanktabellen, …)
- Das Framework ruft den registrierten Code des Programmierers auf, und kontrolliert die Lebenszyklen der angelegten Instanzen (Create, Activate, Passivate, Delete).

Vorteile:

- Keine hart verdrahteten Abhängigkeiten im Code
- Dadurch Austausch von Komponenten zur compile sowie run time möglich
- IoC bzw. dependency injection wird meist von Containern übernommen

Vor- und Nachteile von Frameworks

Vorteile

- Wiederverwendung von Design & Implementierungen
- Schnellere Entwicklung möglich
- Weniger Fehler durch erprobte Mechanismen
- Förderung technischer Standardisierung

Nachteile

- Hoher Einarbeitungsaufwand für den Programmierer
- Programmiersprache und Umgebung strikt vorgegeben
- Nur für einen bestimmten Problembereich anwendbar
- Hoher Entwicklungsaufwand zur Framework-Entwicklung
- Kontrolle liegt beim Framework
- Frameworks lassen sich in der Praxis schlecht kombinieren.

6 – Grundlegende Architekturkonzepte für betriebliche Informationssysteme

6.1 Schichtenarchitekturen betrieblicher Anwendungen

6.2 Bibliotheken und Frameworks

6.3. Aspektorientierung und Java - Annotationen

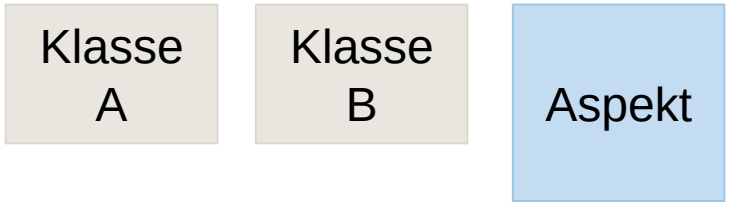
- AOP und AspectJ
- Annotationen mit Beispielen
- Reflexion
- Serialisierbarkeit

Aspektorientierte Programmierung (AOP)

Aspektorientierte Programmierung ist:

> *„ein **Programmierparadigma** im Kontext der objektorientierten Programmierung, um **generische Funktionalität** über mehrere Klassen hinweg zu verwenden (Querschnittsaspekt) und **nur einmal zu implementieren**. Logische Aspekte eines Programms werden dabei von der eigentlichen Geschäftslogik getrennt“.*

Objektorientierte Programmierung



Objektorientierte und aspektorientierte Programmierung

Aspektorientierte Programmierung (AOP)
Bedarf für ein Framework

Beispiel: Transaktionsmanager

- Ziel: Bestimmte Methoden sollen als Transaktion ausgeführt werden
- Transaktion bedeutet, alle Operationen sollen ausgeführt werden. Im Fehlerfall müssen bereits ausgeführte Operationen rückgängig gemacht werden
- Anfallende Daten sollen in einer Datenbank gespeichert werden
- Ablauf: Verbindungsaufbau → Datenzugriff → Verbindungsende

Kann ein Aspekt nicht als normale Java Klasse implementiert werden und per Methodenaufruf ausgeführt werden?

- Ein Aspekt kann die Ausführung von Code sowohl zu Beginn als auch am Ende einer Methode erfordern (z.B. Verbindungsaufbau/-abbau)
- Ein Aspekt kann auf Exceptions reagieren (z.B. Fehler in Datenbank)
- Ein Aspekt kann Attribute und Methoden zu Klassen hinzufügen

→ Die Java-Sprache reicht nicht aus, um die genannten Anforderungen zu erfüllen
→ Lösung: AOP Frameworks wie z.B. AspectJ

Grundlegende AOP Begriffe

- **Interceptors:** Unterbrechen den Programmablauf. Können *before*, *after* oder *around* Methoden zulassen.
- **Joinpoints:** Mögliche Stellen, an denen man **Interceptors** ausführen kann, z.B. Methodenausführung, Objektinitialisierung oder Exceptions.
- **Pointcut:** Ein Pointcut kann mehrere **Joinpoints** beinhalten, z.B. Ausführung jeder Methode der Klasse Kunde, deren Name mit „set” anfängt.
- **Advice:** Ein Advice beinhaltet den auszuführenden Code.
- **Aspect:** Ein Aspekt fasst **Pointcuts** und **Advices** zusammen.

[LR09] Lahres, Bernhard; Rayman Gregor (2009): Objektorientierte Programmierung, 2. Auflage, Galileo Computing, ISBN 978-3-8362-1401-8.
[Pa03] Paragi (2003): Aspect Oriented Programming, Technical Report, Palo Alto Research Center, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA417906 (zuletzt aufgerufen am 18.10.2012).

Aspektorientierte Programmierung (AOP)
Beispiel

Wann immer eine public-Methode einer Klasse aus dem Package *de.tum.sebis* aufgerufen wird, soll dieser Aufruf in der Konsole ausgegeben werden.

1. Pointcut definieren, der alle relevanten Joinpoints beinhaltet
2. Advice definieren, der dem Pointcut den Interceptor *before* zuordnet und den auszuführenden Code beinhaltet
3. Pointcut und Advice zu einem Aspect zusammenführen

execution (private int datumsdatenbank(..));

aspectj

```
public aspect Logging {
    pointcut toBeLogged() :
        execution (public * de.tum.sebis.*.*(..));

    before(): toBeLogged() {
        System.out.println("Before another Method
                                execution");
    }
}
```

1
2
3

[LR09] Lahres, Bernhard; Rayman Gregor (2009): Objektorientierte Programmierung, 2. Auflage, Galileo Computing, ISBN 978-3-8362-1401-8.
[Pa03] Paragi (2003): Aspect Oriented Programming, Technical Report, Palo Alto Research Center, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA417906 (zuletzt aufgerufen am 18.10.2012).

Aspektorientierte Programmierung (AOP)
Joinpoints

Java bzw. AspectJ bietet z.B. folgende Joinpoints

- Ausführen einer Methode
- Ausführen eines Konstruktors
- Zugriff auf Datenelemente (Field Access)
- Auftreten einer Exception

Da selten alle **Joinpoints** einer Klasse (z.B. alle Methodenaufrufe) mit einem Aspekt versehen werden sollen, kann ein **Pointcut** via Pattern Matching auf Klassen- und Methodennamen eingeschränkt werden.

Die Verwendung des Pattern Matchings setzt jedoch voraus, dass alle Methoden die von einem Aspekt betroffen sind, gleich heißen.

→ **Annotationen** ermöglichen die Benennung einer beliebigen Menge von **Joinpoints**, sodass ein Aspekt genau auf diese wirken kann.

Annotationen (Beispiel)

Sowohl die Methode *ueberweiseGeld(int Betrag)* der Klasse Konto als auch die Methode *login()* der Klasse Kontoinhaber sollen bei ihrem Aufruf eine Ausgabe in die Konsole erstellen.

Da reines Pattern Matching nicht zur Selektion der **Joinpoints** ausreicht, müssen eigene **Joinpoints** definiert werden.

In Java geschieht dies über **Annotationen**:

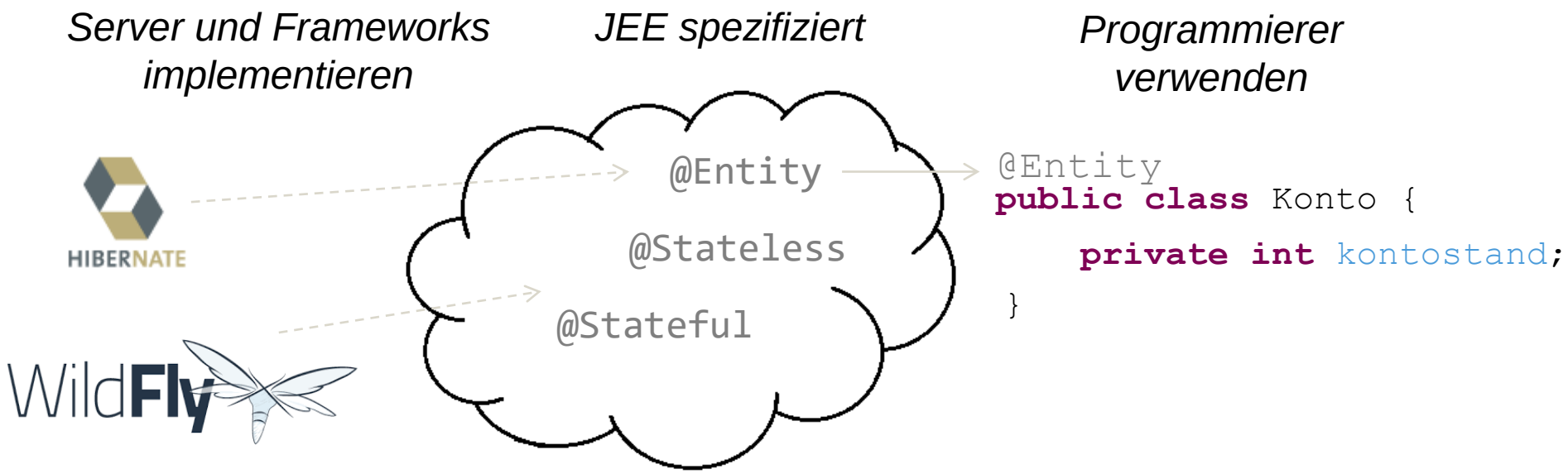
```
public class Konto {

    private int kontostand;

    @PrintLog
    public void ueberweiseGeld(int Betrag) {…}

    public void getKontostand(){…};
}
```

→ Nun kann jeder beliebige **Aspekt** über den Namen „PrintLog“ auf diesen **Joinpoint** zugreifen um dort Code auszuführen (**Advice**)

Verwendung von Annotationen in JEE

JEE spezifiziert eine Reihe von Annotationen, wie z.B. @Entity für Klassen die in einer Datenbank persistiert werden sollen.

- Application Server oder Persistence Frameworks wie Hibernate stellen Aspekte bereit, die die spezifizierte Funktionalität umsetzen
- Der Programmierer muss lediglich **@Entity** an eine Klasse schreiben, um die bereitgestellte Funktionalität zu benutzen



Erstellen eigener Annotationen

Um eine neue Annotation zu definieren, müssen verschiedene Parameter definiert werden.

- Retention (wann wird die **Annotation** ausgewertet)
  - Class: Sie wird in das .class File kompiliert
  - Runtime: Sie steht während der Ausführung zur Verfügung
  - Source: Sie wird vor dem Kompilieren entfernt
- Target (worauf kann die **Annotation** angewendet werden)
  - Method
  - Class
  - Field
  - …

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface PrintLog {
   //keine Methoden nötig
}
```

Reflexion (Reflection)

Definition

> *„Reflexion ist ein Vorgang, bei dem ein Programm auf **Informationen zugreift**, die nicht zu den Daten des Programms, sondern zur **Struktur** des Programms selbst gehören. Diese Strukturen können über Reflexion jedoch **nicht modifiziert** werden“.*

Beispiel: Methodenaufruf über deren Namen

```
public class SebaExamGenerator {
        public String genExcercise() {
                return "Aufgabe 1";
        }
}
public class ReflectionTest {
     public String genExcerciseReflection() {
        SebaExamGenerator obj = new SebaExamGenerator();
        return SebaExamGenerator.class.getMethod("genExcercise").invoke(obj);
          }
}
```

[LR09] Lahres, Bernhard; Rayman Gregor (2009): Objektorientierte Programmierung, 2. Auflage, Galileo Computing, ISBN 978-3-8362-1401-8.

Reflexion zur Auswertung von Annotationen

Damit Klassen auf die Annotationen anderer Klassen zugreifen können, müssen sie diese durch Reflexion auslesen.

Beispiel: Alle mit `@PrintLog` annotierten Methoden der Klasse SebaExamGenerator ausführen.

```
public void callPrintLogs() {
    SebaExamGenerator obj = new SebaExamGenerator();
    // lies alle Methoden aus
    Method[] methods = SebaExamGenerator.class.getMethods();
    // Prüfe Annotation
    for (Method m : methods) {
        if (m.isAnnotationPresent(PrintLog.class)) {
            // Führe Methode aus
            m.invoke(obj);
        }
    }
}
```

Serialisierbarkeit von Objekten

Verteilte Java-Architekturen setzen voraus, dass Objekte „über die Leitung“ übertragen werden können.

Um ein Objekt aus einem Byte-Stream wiederherzustellen werden u.a. benötigt:

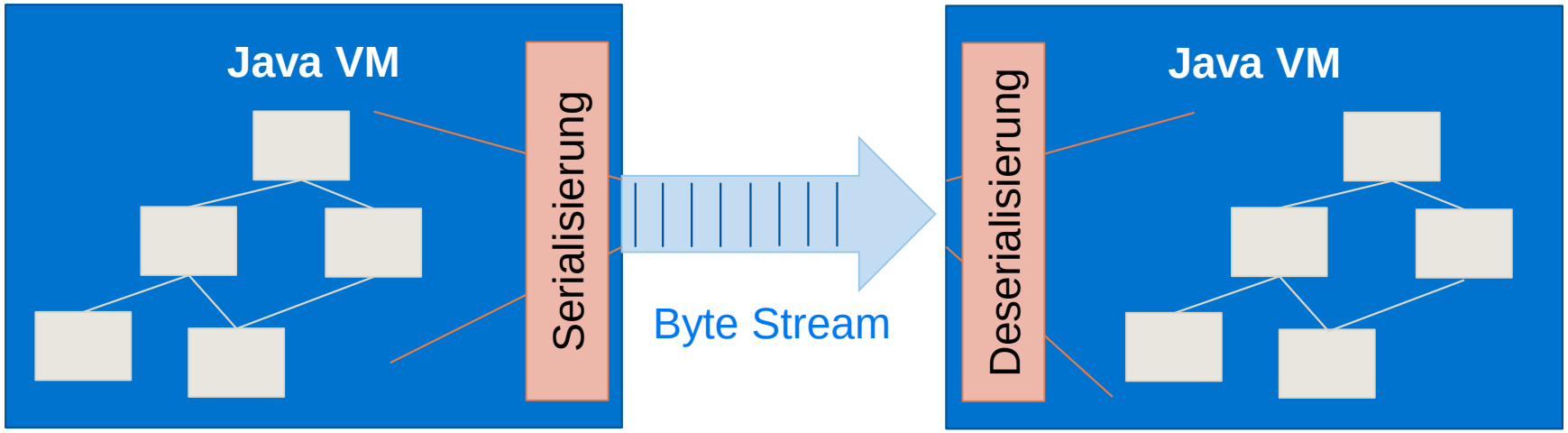
- Attributwerte
- Attributtypen
- Objekttyp
- Alle referenzierten Objekte

Um die Objekte einer Klasse serialisierbar zu machen, muss das **Serializable** Interface implementiert werden (Flag-Interface).

```
public class Konto implements Serializable {
   //more Code here
}
```

→ Serialisieren und Deserialisieren nennt man auch (un)marshalling

Serialisierbarkeit von Objekten



Grenzen: nicht alles kann serialisiert werden

- Threads
- FileInputStreams
- `transient` Attribute, z.B. Passwörter
- Sockets und ServerSockets
- Alle Objekte, die auf Objekte einer solchen Klasse verweisen