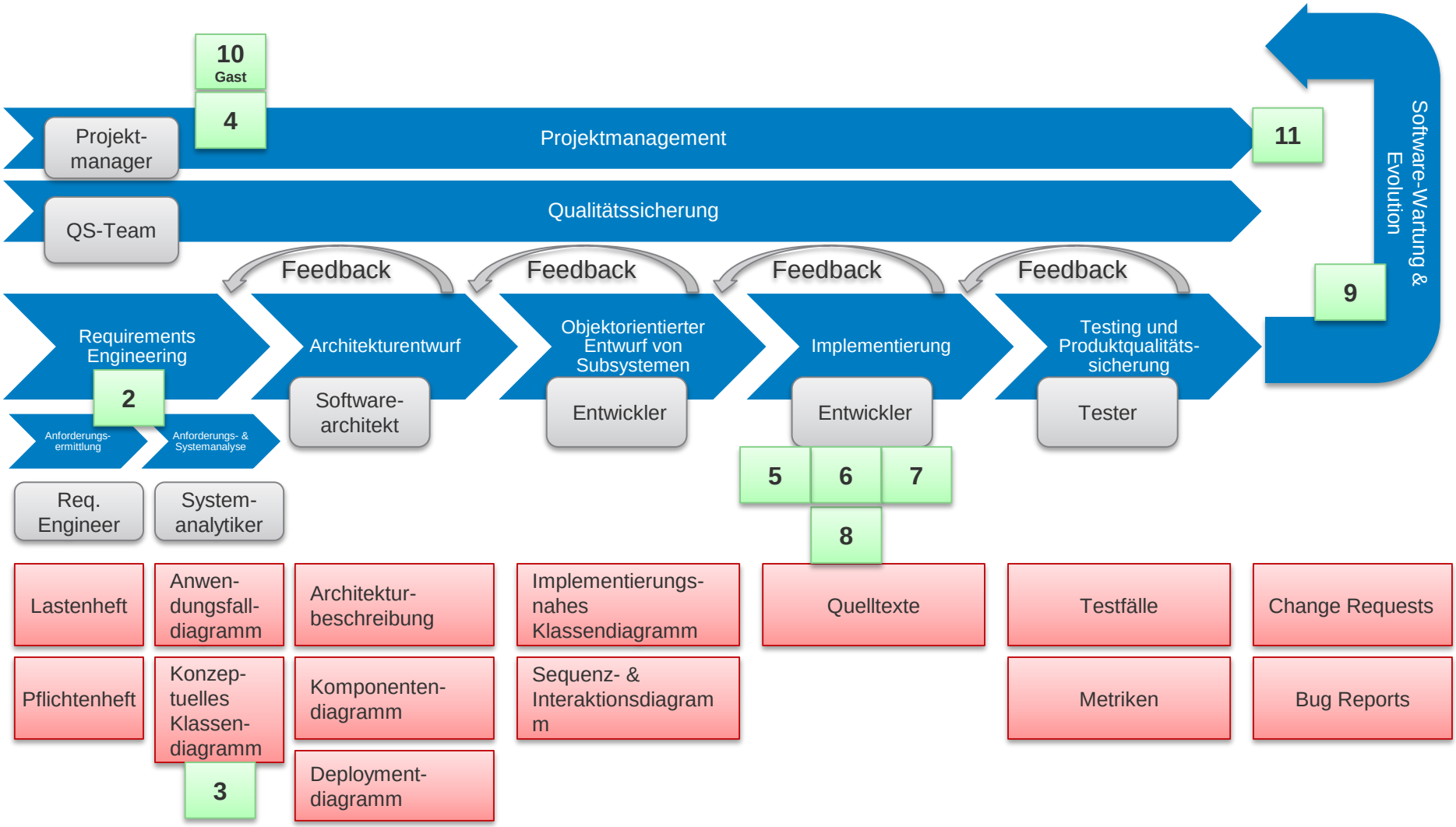
Vorlesungsgliederung

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 (z.B. Aufwandsschätzung, Software-Renovierung, …)
11. Unterstützung von Geschäftsprozessen

Themen SEBA Bachelor und Gliederung EIST



8 - Persistenz

8.1 Motivation

8.2 Persistente Datenspeicher: Stärken und Schwächen

8.3 Zugriff auf persistente Datenspeicher

8.4 Persistent Entities

8.5 Criteria API

Management persistenter Daten in betrieblichen Anwendungen

Persistente Daten: Daten werden über die Lebensdauer der Sitzung eines Benutzers oder der Betriebssystem-Prozesse, die die Daten verarbeiten, hinaus gespeichert (z.B. Stammdaten, Bewegungsdaten, Historisierte Daten, …).

Datenhaltung / Datenverwaltung: Mechanismen und Strategien zur Speicherung und Bearbeitung von Daten.

Zur Erinnerung:

1. Spezialisierte Datenstrukturen (Einführung in die Informatik 1) für spezifische Zugriffsmuster
2. Relationale Datenspeicherung (Grundlagen Datenbanken) für Speicherung von Massendaten

→ Zugriffsmuster aus 1. passt nicht zu Speicherungsparadigma aus 2. (*Impedance Mismatch)*

→ Problemstellung, welche in betrieblichen Anwendungen angegangen werden muss

7 - Persistenz

7.1 Motivation

7.2 Persistente Datenspeicher: Stärken und Schwächen

7.3 Zugriff auf persistente Datenspeicher

7.4 Persistent Entities

7.5 Criteria API

Zur Implementierung stehen verschiedene persistente Datenspeicher zur Verfügung

sebis

- Dateisystem (Textdateien, XML-Dateien, Proprietäre Dateiformate, …)
- Relationale Datenbank
  - MySQL , Oracle DB, IBM DB2, MS SQL Server
- Native XML Datenbank
  - Tamino XML Server, Apache Xindice
- Objektorientierte Datenbank
  - GemStone/S, Ozone database
- Content Management System
  - CoreMedia CMS, Jackrabbit (Open Source)
- NoSQL Datenbanken
  - db4o, Neo4j, mongoDB, couchDB, Berkeley DB…

Dateisystem (1)

Mittels einer I/O API können Daten unter verschiedenen Betriebssystemen (DOS, Windows, Unix etc.) in Dateien geschrieben werden. In den meisten Fällen ist die I/O API transparent bezüglich verschiedener Datenträger (Festplatte, Diskette, CD-ROM, Netzlaufwerk, SIM Kerte etc.)

Üblicherweise wird die Standard I/O API der jeweiligen Programmiersprache verwendet.

Beispiel 1: Lesen von Zeilen aus einer Textdatei in Java

```
try {
   BufferedReader in = new BufferedReader(new FileReader("file.txt"));
   String line = in.readLine();
   while (line != null) {
      System.out.println(line);
      line = in.readLine();
   }
   in.close();
} catch (IOException ioe) {
   //Ausnahmebehandlung
}
```

[Da04]

Dateisystem (2)

Beispiel 2: Schreiben von Text in eine Datei in Java

```
try {
   PrintWriter out = new PrintWriter(new FileWriter("file.txt"));
   out.println("Hello, World!");
   out.close();
} catch (IOException ioe) {
   //Ausnahmebehandlung
}
```

Vorteile

- Überall verfügbar selbst auf SIM-Karten
- Einfachste Form von Zugriff; Benutzung

Nachteile

- Fehlende Erweiterbarkeit
- Mehrbenutzerzugriff
- Skalierbarkeit
- Kodierung indifferent, z.B. Little vs. Big Endian
- Updates problematisch

[Da04]

XML-Dateien

eXtensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<trial attempt="1">
  <timestamp>10/15/2012</timestamp>
  <request>This is a request.</request>
</trial>
```

Vorteile

- plattformunabhängig
- erweiterbar
- flexibel
- menschenlesbar, selbstbeschreibend
- eignet sich zur Kapselung von Legacy-Anwendungen
- wird gut von Datenbankmanagementsystemen unterstützt
- trennt den Inhalt von seiner Präsentation (im Gegensatz zu HTML)
- wird durch APIs zur XML-Bearbeitung unterstützt
- verwendet anwendungsspezifische Tags

Nachteile

- im Detail nicht trivial
- stellt keine einfache Unterstützung für Referenzen zur Verfügung
- beschränkt auf hierarchische Schachtelung (wie Dateisystem)
- unterstützt Aktualisierung nur über Laden & Speichern

Technologien zur Verarbeitung von XML

XML wird von der Industrie und von Communities weitreichend unterstützt

- Programmiermodelle
  - Simple API for XML (SAX) - Strom und Ereignisbasierte Verarbeitung
  - Document Object Model (DOM) - Hauptspeicherbasierte Baumstruktur
- Bibliotheken zur Bearbeitung
  - JDOM, DOM4J
  - Apache Xerces, Xalan
- Auf XML aufbauende und unterstützende Standards
  - DTD, XML Schema
  - XPath, XLink, XPointer, XQuery
  - XSL, XSLT
- Forschungsthema: XML & Programmiersprachen
  - Typisierung
  - Literale
  - Anfragen

Datenbankmanagementsystem

Die Gesamtheit der Programme zum Zugriff auf die Datenbasis, zur Kontrolle der Konsistenz und zur Modifikation der Daten wird als Datenbankverwaltungssystem (bzw. Datenbankmanagementsystem) bezeichnet. [KE06]

Persistenzbezogene Funktionalitäten:

Persistente Haltung von Daten, Änderung gespeicherter Daten, parallele Datenänderungen, Umgang mit Massendaten, Einhaltung von Integritätsbedingungen sicherstellen, Recovery im Fehlerfall.

Varianten:

- Relationale Datenbank
- Objektorientierte Datenbank → hat sich nicht durchgesetzt
- (Native) XML Datenbank → hat sich nicht durchgesetzt
- Historisch: hierarchische Datenbank
- NoSQL Datenbanken
- …

Grundlagen relationaler Datenbanken

- Eine relationale Datenbank ist eine Menge von benannten Relationen.
- Die Anzahl der Zeilen der Tabelle ist offen und wird Kardinalität der Relation genannt.
- Die Anzahl der Spalten der Tabelle ist fest und wird Stelligkeit (engl.: arity) der Relation genannt.
- Jede Relation besitzt einen Primärschlüssel (Primary Key), dieser kann ein einzelnes Attribut sein oder sich aus mehreren Attributen zusammensetzen. Der Primärschlüssel identifiziert jedes Tupel in der Relation eindeutig.
- Beziehungen zwischen Tupeln werden durch Referenzierung des entsprechenden Tupels über dessen Primärschlüssel ausgedrückt.
- Wird ein Primärschlüssel in Relation A verwendet um dort auf ein Tupel in Relation B zu referenzieren, so heißt dieser Schlüssel in Relation A Fremdschlüssel (Foreign Key).

Definition Contentmanagementsystem (CMS)

Ein Contentmanagementsystem (CMS) ist ein System, das es mehreren Benutzern (Programmen) ermöglicht Informationen (Content) gemeinsam zu bearbeiten und zu speichern.

→ **Wichtig: Entkopplung von Information und Repräsentation**

Standard für den Zugriff auf CMS: Java Content Repository (JCR) API

- Unabhängig von allen darunter liegenden Architekturen, Datenquellen oder Protokollen
- Einfach zu verwenden (programmieren)
- Einfache Unterstützung gängiger CMS
- Standardisierung komplexer Funktionalitäten (für fortgeschrittene *content-related* applications)

Zwischen den letzten beiden Zielen bestehen Spannungen → daher wurde die Spezifikation in zwei compliance levels unterteilt.

NoSQL Datenbanken - Definition

Unter NoSQL wird eine neue Generation von Datenbanksystemen verstanden, die meistens einige der nachfolgenden Punkte berücksichtigt:

1. Das zugehörige Datenmodell ist nicht relational.

2. Die System sind von Anbeginn an auf eine verteilte und horizontale Skalierbarkeit ausgerichtet.

3. Das NoSQL-System ist Open Source

4. Das System ist schemafrei oder hat nur schwächere Schemarestriktion

5. Aufgrund der verteilten Architektur unterstützt das System eine einfache Datenreplikation

6. Das System bietet eine einfache API

7. Dem System liegt mindestens auch ein anderes Konsistenzmodell zugrunde:
   z.B. *Eventually Consistent* aber nicht *ACID*.

[Ed10]

Wann sollten NoSQL Datenbanken anstelle von relationalen Datenbanken eingesetzt werden (1)

- Relationale Datenbanken weisen i.d.R. Leistungsprobleme bei datenintensiven Applikationen auf, z.B.:
  - Indexierung von großen Dokumentmengen,
  - Webseiten mit hohen Lastaufkommen,
  - Streaming-Media-Applikationen.

- Relationale Datenbanken sind nur dann effizient, wenn sie für **häufige aber kleine Transaktionen** oder für **große Batch-Transaktionen mit seltenen Schreibzugriffen optimiert sind**.

- Relationale Datenbanken können i.d.R. schlecht mit gleichzeitig hohen Datenanforderungen und häufigen Datenänderungen umgehen.

Beispiel
Im Juni 2011 wurden täglich 200 Millionen Tweets gesendet. Genau ein Jahr zuvor waren es täglich *nur* 65 Millionen Tweets.

Wann sollten NoSQL Datenbanken anstelle von relationalen Datenbanken eingesetzt werden (2)

sebis

- NoSQL Datenbanken können mit vielen Schreib- und Leseanfragen umgehen.
  → Digg, Facebook, Twitter, eBay, Amazon, etc.

- NoSQL-Architekturen bieten dafür aber i.d.R. nur schwache Garantien hinsichtlich Konsistenz (*eventual consistency*)

- Einige NoSQL Systeme unterstützen ACID, beispielsweise durch Hinzufügung spezieller Middleware wie CloudTPS

- Viele NoSQL-Implementierungen unterstützen verteilte Datenbanken mit redundanter Datenhaltung auf vielen Servern (Knoten auf großen DB-Clustern), beispielsweise unter Nutzung einer verteilten *hash table* (können darurch Ausfälle leichter überstehen)

- Einer der Vorreiter in diesem Gebiet war Google mit dem dem BigTable-Datenbanksystem (2004) auf dem eigenen Fileserver (GFS). Diese Technologie hat die Verbreitung des Google‘s Map/Reduce Verfahrens sehr begünstigt.

Kategorisierung von NoSQL-Systemen

NoSQL-Kernsysteme

- Wide Column Stores / Column Families
- Document Stores
- Key/Value/Tuple Stores
- Graphdatenbanken

Nachgelagerte NoSQL-Systeme

- Objektdatenbanken
- XML-Datenbanken
- Grid-Datenbanken
- und viele weitere nichtrelationale Systeme…

[Ed10]

7 - Persistenz

Zugriff auf persistente, relationale Datenspeicher

- **Ziel**
  → Quellcode der Geschäftslogik greift auf persistent gespeicherte Daten zu
- **Übliches Szenario**
  - Geschäftslogik ist in objektorientierter Programmiersprache entwickelt
  - Relationales Datenbanksystem wird als persistenter Datenspeicher verwendet
- **Anforderungen**:
  - Erfüllung der ACID-Bedingungen für Transaktionen (*Atomicity*, *Consistency*, *Isolation*, *Durability*)
  - Geschäftslogik unabhängig vom Datenzugriff (z.B. bei Änderungen der Persistenzstrategie dürfen keine Änderungen an der Geschäftslogik nötig werden)

Verschiedene Umsetzungsstrategien

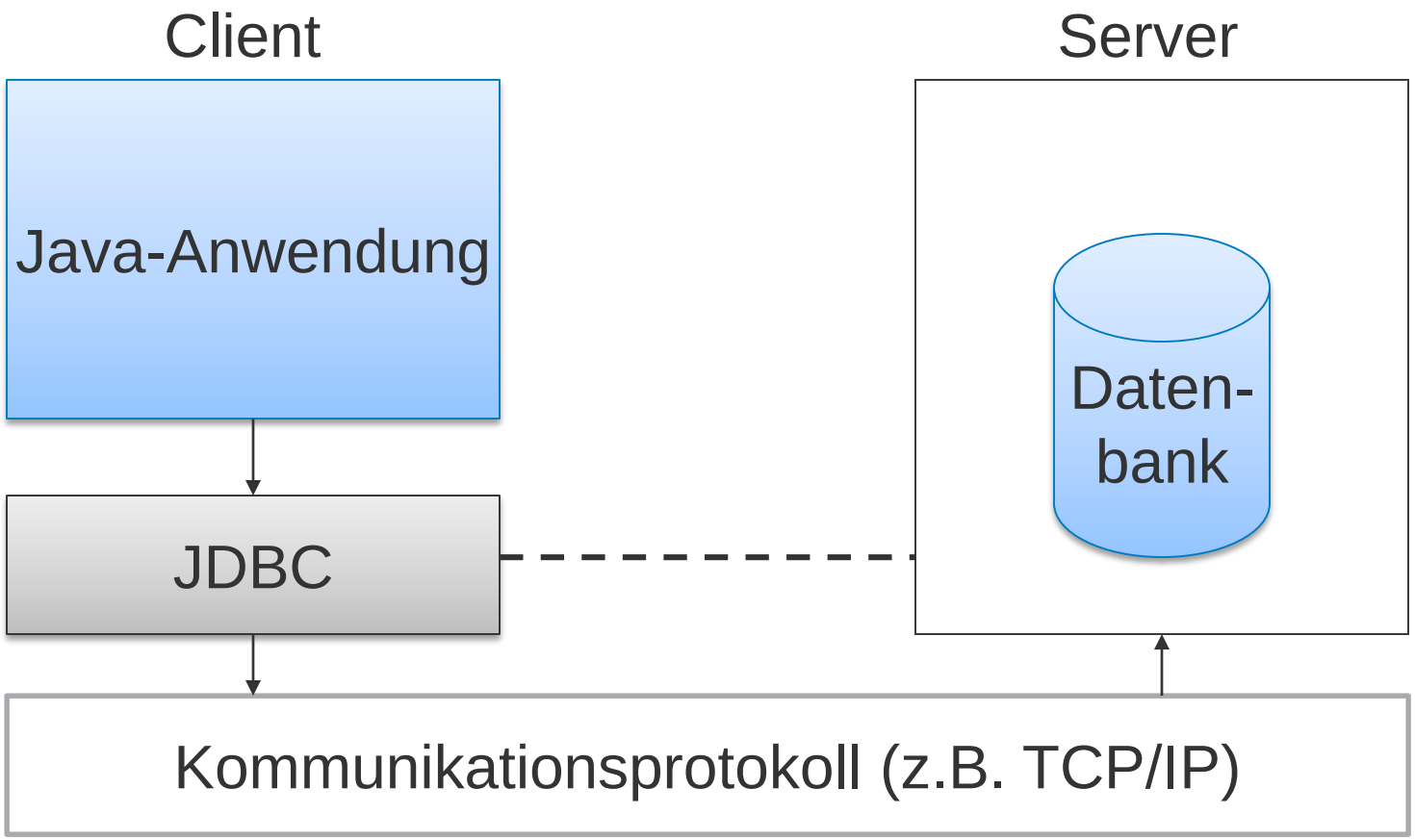
Direkte SQL-Aufrufe

- Mittels einer entsprechenden Technologie aus der Programmiersprache heraus
- Beispiele: JDBC, ODBC

Software für objektrelationales Mapping

- Automatisiert Aspekte des Zugriffs auf den persistenten Datenspeicher
- Beispiel: Persistent Entities

JDBC – Ein Überblick



Vor- und Nachteile von JDBC

Vorteile

JDBC erweist sich als günstig

- falls Stored Procedures aufgerufen werden sollen
- spezielle Abfragen ausgeführt werden müssen
- auf proprietäre Datenbank-Funktionalität zugegriffen werden soll

Nachteile

- Während der Entwicklung oft fehleranfällig, die Handhabung erweist sich i. A. als zu komplex für Entwickler
  - z.B. sind proprietäre Error-Codes der Hersteller in SQL-Exceptions wenig aussagekräftig und aufwändig zu interpretieren.
- Erfordert die Festlegung auf eine bestimmte Persistenzstrategie
  - z.B. ist die Tabellenstruktur nur schwer zu ändern, da dies umfangreiche Änderungen am JDBC-basierten Code nach sich zieht
  - Wird z.B. das Datenbankschema durch Software zum objektrelationalen Mapping generiert, sind strukturelle Änderungen einfacher abzubilden
- JDBC-Aufrufe dürfen nicht direkt in Geschäftslogik-Quellcode integriert werden

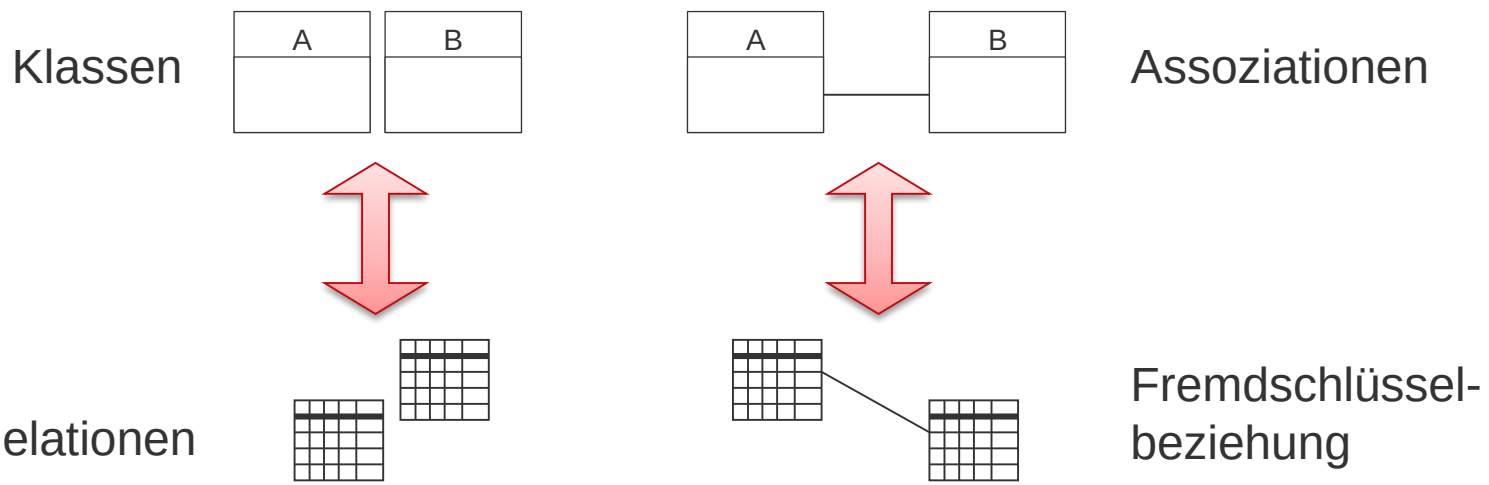
Objektrelationales Mapping (OR-Mapping)

Motivation:

- RDBMS stellen eine reife Technologie dar (>25 Jahre)
- Objektorientierung ist ein verbreitetes und ausdrucksstarkes Entwicklungsparadigma

➔ Allerdings: Probleme von z.B. JDBC-basiertem Datenzugriff

Objektrelationales Mapping versucht, den Zustand von (Java-)Objekten auf Daten in einer relationalen Datenbank abzubilden, um transparent persistente Datenhaltung bereitzustellen.

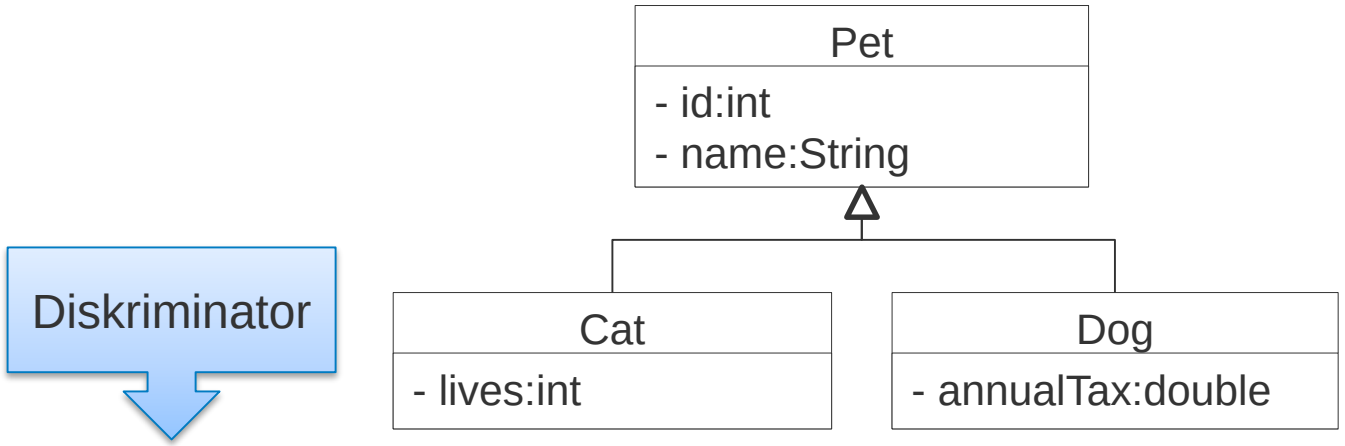


Relationale Abbildung von Vererbungshierarchien – „Single Table Strategy“

- Alle Klassen der Subklassenhierarchie werden auf eine Tabelle abgebildet
- Alle Attribute aller (Sub-)Klassen werden auf Spalten abgebildet

Vorteil: Einfach PK-Behandlung, Suche nach Instanzen von Pet ohne JOIN

Nachteil: Viele NULL-Werte



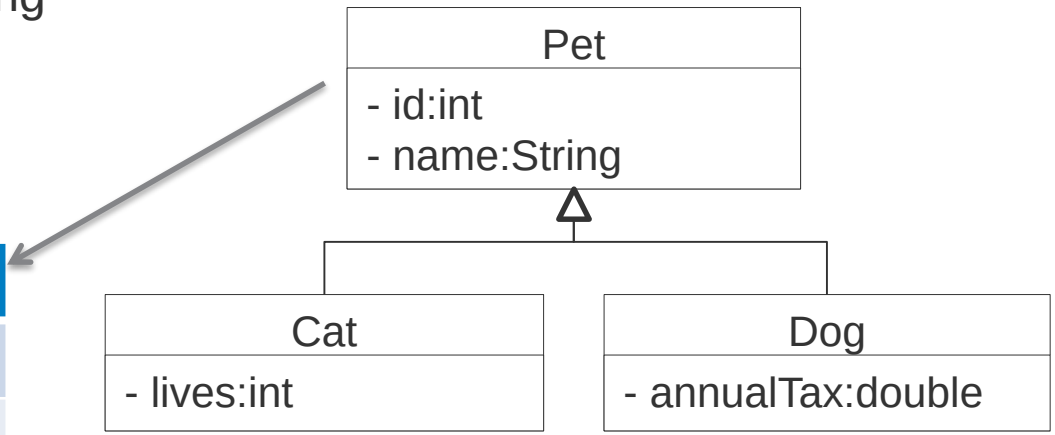
Pets				
id	**type**	**name**	**lives**	**annualTax**
1	Pet	Karl	NULL	NULL
2	Cat	Schnurri	7	NULL
3	Dog	Hasso	NULL	270,30

Relationale Abbildung von Vererbungshierarchien – „Table per Class Strategy“

- Alle Klassen der Subklassenhierarchie werden jeweils auf eine Tabelle abgebildet
- Die entsprechenden Attribute der (Sub-)Klassen werden auf Spalten abgebildet

Vorteil: Keine NULL-Werte, Suche nach Instanzen von Pet ohne JOIN

Nachteil: Komplexe PK-Behandlung



Pets	
id	**name**
1	Karl

Cats		
id	**name**	**lives**
2	Schnurri	7

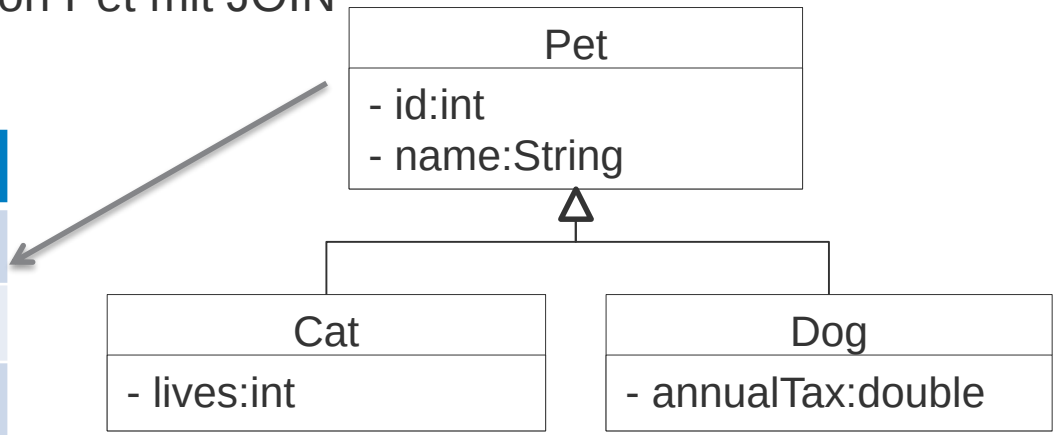
Dogs		
id	**name**	**annualTax**
3	Hasso	270,30

Relationale Abbildung von Vererbungshierarchien – „Joined Table Strategy“

- Alle Klassen der Subklassenhierarchie werden jeweils auf eine Tabelle abgebildet
- Nur die (Sub-)Klassen-spezifischen Attribute werden auf Spalten abgebildet

Vorteil: Kein NULL-Werte, Einfache PK-Behandlung

Nachteil: Suche nach Instanzen von Pet mit JOIN



Pets	
id	**name**
1	Karl
2	Schnurri
3	Hasso

Cats	
id	**lives**
2	7

Dogs	
id	**annualTax**
3	270,30

Software für objektrelationales Mapping

- Einfach strukturierte Daten können in relationale DB ausgelagert werden und von deren Fähigkeiten (Indizierung, Konsistenzsicherung, hohe Verfügbarkeit, Mehrbenutzerbetrieb) profitieren.
- Komplex strukturierte Daten werden (soweit möglich) transparent in einfacher strukturierte, dem RDBMS zugängliche Strukturen zerlegt.
- Der Benutzer „sieht“ ausschließlich Strukturen der objektorientierten Programmiersprache, die Umsetzung von OO-Operationen auf relationale Operationen erfolgt transparent.
- Historisch gewachsene und in relationalen Datenbanken gespeicherte Daten können mit den Mitteln des objektorientierten Systems weiterverwendet werden.
- Dienstleistungen, die bereits von der relationalen Datenbank erbracht werden, brauchen in der objektorientierten Anwendung nicht erneut realisiert zu werden.
  - z.B. Sicherstellung von Datenintegrität
  - …

7 - Persistenz

Grundlagen

- Persistent Entities bieten objektorientierter Zugriff auf die persistenten Informationen in der Datenbank (z.B. Kunde oder Artikel).
  - z.B. „Mapping“ einer Tabelle auf eine Entity-Klasse
  - Jede Zeile der Tabelle wird durch eine Instanz einer Entity repräsentiert

- Persistenzfunktionen für Entities können entfernt genutzt werden und sind mehrbenutzerfähig, d.h. mehrere Clients können Entity-Instanzen benutzen, welche dieselben Daten repräsentieren.

- Jede Entity-Instanz besitzt einen eindeutigen Primary Key (analog zum Datenbankeintrag).

- Entities existieren solange die zugehörige Datenbank existiert (überleben also auch Serverausfälle) oder sie explizit gelöscht werden

Spezifikation/Entwicklung einer Entity

Eine Entity gehorcht der JavaBean Spezifikation, d.h.

- Attribute haben „getter“- und „setter“-Methoden
- Die Klasse hat einen Default-Konstruktor (ohne Parameter)
- Die Klasse muss `java.io.Serializable` implementieren

Darüber hinaus ist sie für ein O/R-Mapping annotiert :

- Zur Markierung einer Entity: `javax.persistence.Entity`
- Zur Auswahl des Primärschlüssels: `javax.persistence.Id`

Optional:

- Zur Generierung von Primärschlüsseln: `javax.persistence.GeneratedValue`
- Zur Auswahl der Tabelle: `javax.persistence.Table`
- Zur Auswahl der Tabellenspalte: `javax.persistence.Column`

Beispiel einer Entity

```java
import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name="Pet")
public class Pet implements Serializable {
    private int id;
    private String name;

    @Id
    @GeneratedValue
    public int getId() {
        return this.id;
    }

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

    @Column(name="name")
    public String getName() {
        return this.name;
    }

    public void setName(String name) { this.name = name; }
}
```

Objektrelationale Abbildung bei Vererbung

- Single Table Strategy

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Type")
@DiscriminatorValue("Pet")
public class Pet implements Serializable {
    /* ... */
}

@Entity
@DiscriminatorValue("Cat")
public class Cat extends Pet {
     /* … */
}
```

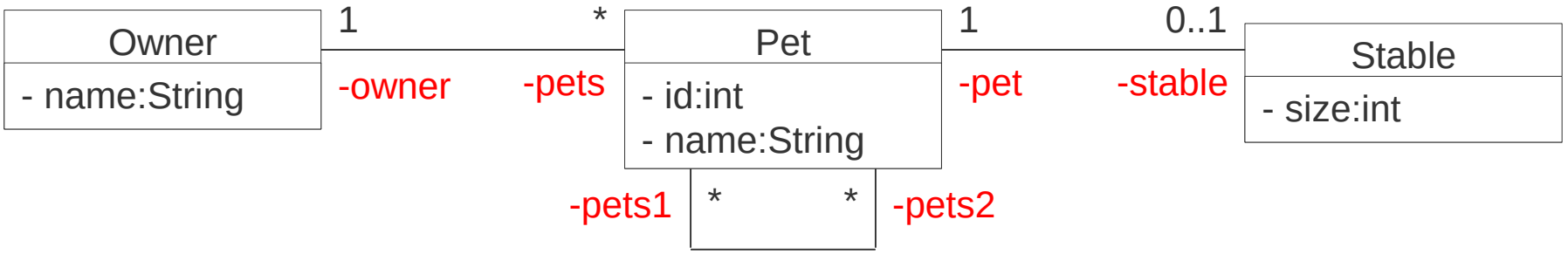
- Table Per Class Strategy: `@Inheritance(strategy=TABLE_PER_CLASS)`
- Joined Table Strategy: `@Inheritance(strategy=JOINED)`

➔ In beiden Fällen ist kein Diskriminator nötig.

Objektrelationale Abbildung bei Assoziationen zwischen Objekten

- JPA unterstützt die Abbildung von Assoziationen folgender Typen:
  - One-To-One
  - One-To-Many
  - Many-To-Many
- JPA bietet Parameter zur Festlegung der Lösch- oder Änderungsweitergabe, z.B. `cascade = DELETE`
- JPA bietet **verschiedene Ladestrategien**:
  - Referenzierte Objekte sofort laden `fetchType = EAGER`
  - Referenzierte Objekte später nachladen `fetchType = LAZY`

Problem
Der Entwickler der Entity spezifiziert das Ladeverhalten, evtl. ohne zu wissen, wie die Entity später (z.B. in Session-Beans) genutzt wird.



Objektrelationale Abbildung: One-To-One Assoziation

```java
@Entity
public class Pet implements Serializable {
    private Stable stable;

    @OneToOne(optional=true)
    public Stable getStable() { return this.stable; }

    public void setStable(Stable stable) { this.stable = stable; }

    /* … */
}
```

```java
@Entity
public class Stable implements Serializable {
    private Pet pet;

    @OneToOne(mappedBy="stable")
    public Pet getPet() { return this.pet; }

    public void setPet(Pet pet) { this.pet = pet; }

    /* … */
}
```

Objektrelationale Abbildung: One-To-Many Assoziation

```java
@Entity
public class Pet implements Serializable {
    private Owner owner;

    @ManyToOne
    public Owner getOwner() { return this.owner; }

    public void setOwner(Owner owner) { this.owner = owner; }

    /* … */
}
```

```java
@Entity
public class Owner implements Serializable {
    private Collection<Pet> pets;

    @OneToMany(mappedBy="owner",cascade=CascadeType.REMOVE, fetch=FetchType.EAGER)
    public Collection<Pet> getPets() { return this.pets; }

    public void setPets(Collection<Pet> pets) { this.pets = pets; }

    /* … */
}
```

Objektrelationale Abbildung: Many-To-Many Assoziation

```java
@Entity
public class Pet implements Serializable {
    private Collection<Pet> pets1;
    private Collection<Pet> pets2;

    @ManyToMany
    public Collection<Pet> getPets1() { return this.pets1; }

    public void setPets1(Collection<Pet> pets1) { this.pets1 = pets1; }


    @ManyToMany(mappedBy="pets1")
    public Collection<Pet> getPets2() { return this.pets2; }

    public void setPets2(Collection<Pet> pets2) { this.pets2 = pets2; }

    /* … */
}
```

- Achtung: `@ManyToMany(mappedBy="pets1“, fetchType=EAGER)` würde bedeuten, dass alle `pets1` und deren `pets1` und … (transitive Hülle) geladen werden!

Nutzer-Sicht auf ein Persistent Entity

Der Nutzer (Programmierer) benötigt Methoden zwei verschiedener Arten:

- Methoden zum Erstellen, Finden und Löschen von Entity-Instanzen

  ➔ EntityManager (wird für alle Entity-Typen verwendet)

- Methoden für den lesenden und schreibenden Zugriff auf Attributwerte einer Entity-Instanz

  ➔ get- und set-Methoden (pro Entity-Typ)

Entity Manager

`javax.persistence.EntityManager` erlaubt den Zugriff auf den Datenspeicher.

Er bietet folgende Methoden:

- `void persist(Object o)` speichert das Objekt in die Datenbank
- `void remove(Object o)` löscht das Objekt aus der Datenbank
- `Object find(Class clazz, Object primaryKey)` findet das Objekt mit dem zugehörigen PrimaryKey in der Datenbank
- `void merge(Object o)` verändert das Objekt in der Datenbank

Wichtig: Die Methoden des EntityManagers unterstützen Transaktionen, wie sie z.B. von Session-Beans erzeugt werden.

Beispiel: Verwendung des Entity-Managers

- Annahme: ein EntityManager ist unter dem Namen `sebaEM` registriert

```
@Stateless
@Remote({ TestInterface.class })
public class TestBean implements TestInterface {
    @PersistenceContext(unitName = "sebaEM")
    protected EntityManager em;

    public void createPet(Pet p) {
        this.em.persist(p);
    }

    public Pet findPet(int id) {
        return this.em.find(Pet.class, id);
    }
}
```

Entity Manager: Querying (1)

Anforderungen:

- Abfragefunktionen (Querys) über den Datenbeständen auf Persistent Entities
- Assoziationen sind datenbankunabhängig ausgedrückt

➔ Datenbankunabhängige Sprache (Java Persistence Query Language – JPQL)

➔ natives SQL der genutzten Datenbank wird ebenfalls unterstützt

EntityManager unterstützt Querys dieser Art durch die Methode:

- `Query createQuery(String jpqlStatement)` – erzeugt ein neues Query aus dem angegebenen Statement

Die Klasse `javax.persistency.Query` unterstützt zwei Methoden zum Zugriff auf Ergebnisse:

- `List getResultList()` – gibt eine Ergebnisliste zurück
- `Object getSingleResult()` – gibt das einzige Ergebnis zurück oder wirft eine `NonUniqueResultException`, falls es mehr als ein Ergebnis gibt

Entity Manager: Querying (2)

Beispiel JPQL:

```
@PersistenceContext(unitName="sebaEM")
protected EntityManager em;

public List<Cat> findCatsByLives(int livecount) {
    Query q = em.createQuery("select c from Cat c where c.lives = ?1");
    q.setParameter(1, livecount);
    return q.getResultList();
}
```

- Zur Projektion (SELECT) muss in JPQL immer ein Alias verwendet werden (c)
- Die Selektion (FROM) bezieht sich immer auf eine Persistent Entity Klasse (Cat)
- Mit „?“ können einzelne Bestandteile der Query parametrisiert werden, wodurch SQL Injection verhindert werden kann

Vorteile

- JPQL sehr ähnlich zu SQL
- Query kann einfach als String formuliert werden

Nachteile

- Compiler kann Strings nicht prüfen → Fehleranfällig
- Lange Queries schnell sehr unübersichtlich

7 - Persistenz

Criteria API Motivation

Beispiel JPQL:

```
@PersistenceContext(unitName="sebaEM")
protected EntityManager em;

public List<Cat> findCatsByLives(int livecount) {
    Query q = em.createQuery("select c from Cat c where c.lives = ?1");
    q.setParameter(1, livecount);
    return (List<Cat>) q.getResultList();
}
```

Nachteile von JPQL

- Statt Java wird eine weitere Sprache verwendet (JPQL, SQL)
- Weder Attribute noch Klassen sind typsicher, d.h. der Compiler kann deren Richtigkeit nicht prüfen
- Der Compiler kann weder die richtige Schreibweise von Namen als auch der JPQL Schlüsselwörter (SELECT,…) prüfen → Fehler erst zur Laufzeit
- Dynamisches generieren von Filterns (WHERE) erfordert unschöne String-Konkatenation

Die 6 Schritte des Criteria API

Schritte

1. Erstelle ein CriteriaBuilder und ein CriteriaQuery Objekt
2. Konfiguriere die FROM Klausel
3. Konfiguriere die SELECT Klausel
4. Konfiguriere die Kriterien (Prädikate, WHERE)
5. Erstelle die WHERE Klausel mithilfe der Kriterien
6. Führe die Query aus

Beispiel

```
//Immer gleich
CriteriaBuilder cb = em.getCriteriaBuilder();                    //Schritt 1
CriteriaQuery<Customer> cqry = cb.createQuery(Customer.class);   //Schritt 1

//Interessanter Code ist hier
Root<Customer> root = cqry.from(Customer.class);         //Schritt 2 (FROM Customer c)
cqry.select(root);                                       //Schritt 3 (SELECT *)

//Immer gleich
Query qry = em.createQuery(cqry);                        //Schritt 6 Query erstellen
List<Customer> results = qry.getResultList();            //Schritt 6 Query ausführen
```

Strukturiert Kriterien definieren

Prädikate repräsentieren Einschränkungen auf der Ergebnismenge

```
//Immer gleich
CriteriaBuilder cb = em.getCriteriaBuilder();                    //Schritt 1
CriteriaQuery<Customer> cqry = cb.createQuery(Customer. class);  //Schritt 1

//Interessanter Code ist hier
Root<Customer> root = cqry.from(Customer.class);          //Schritt 2 (FROM Customer c)
cqry.select(root);                                        //Schritt 3 (SELECT *)
//WHERE Klausel
Predicate pGtAge = cb.gt(root.get("age"),10);             //Schritt 4 Prädikat
cqry.where(pGtAge);                                       //Schritt 5 (WHERE age > 10)

//Immer gleich
Query qry = em.createQuery(cqry);                         //Schritt 6 Query erstellen
List<Customer> results = qry.getResultList();             //Schritt 6 Query ausführen
```

Methoden zur Prädikatsdefinition (Auswahl)

- **cb.gt():** größer als
- **cb.equals():** gleich
- **cb.and(Predicate a, Predicate b):** WHERE a AND b

Typsicherheit

Typsicherheit auf Attributen noch nicht gewährleistet

```
Predicate pGtAge = cb.gt(root.get("age"),10);          //Schritt 4 Prädikat
```

Referenz auf Attribut mit dessen Namen als String → Fehlerquelle

Typsicherheit via Meta-Modell

- Zugrundeliegendes Meta-Modell lässt sich mit externen Bibliotheken automatisiert erstellen
- Zugriff auf Attribute des Meta-Modells erfolgt über Klassenreferenz (z.B. Customer_.age)
- Damit ist Typsicherheit gewährleistet
- Meta-Modell muss bei jeder strukturellen Änderung einer Klasse neu erstellt werden → Einbettung in Build-Management nötig
- Meta-Modell wird in der Vorlesung nicht weiter verwendet