

Medii de proiectare și programare

2024-2025

Curs 9

Conținut

- Object-Relational Mapping (ORM)
 - Hibernate
 - Entity Framework
- Servicii Web

ORM Impedance Mismatch

- Paradigma orientată obiect promovează dezvoltarea aplicațiilor folosind obiecte care păstrează date, dar conțin și logica aplicației.
- Bazele de date relaționale stochează datele în tabele și manipulează datele folosind proceduri stocate și interogări SQL.
- Diferențele dintre cele două abordări au fost numite: *object-relational impedance mismatch* sau doar *impedance mismatch*.
- Ex. în paradigma orientată obiect obiectele sunt traversate folosind relațiile dintre ele, în paradigma relațională se folosește operația de join.
- Tipurile de date diferite în limbajele orientate obiect și bazele de date relaționale:
 - Java: string și int - Oracle: varchar și smallint.
 - Java: colecții - Oracle: tabele
 - Java: obiecte - Oracle: blobs

Strategii pentru Impedance Mismatch

- Maparea moștenirii
- Maparea relațiilor dintre obiecte
- Maparea proprietăților statice

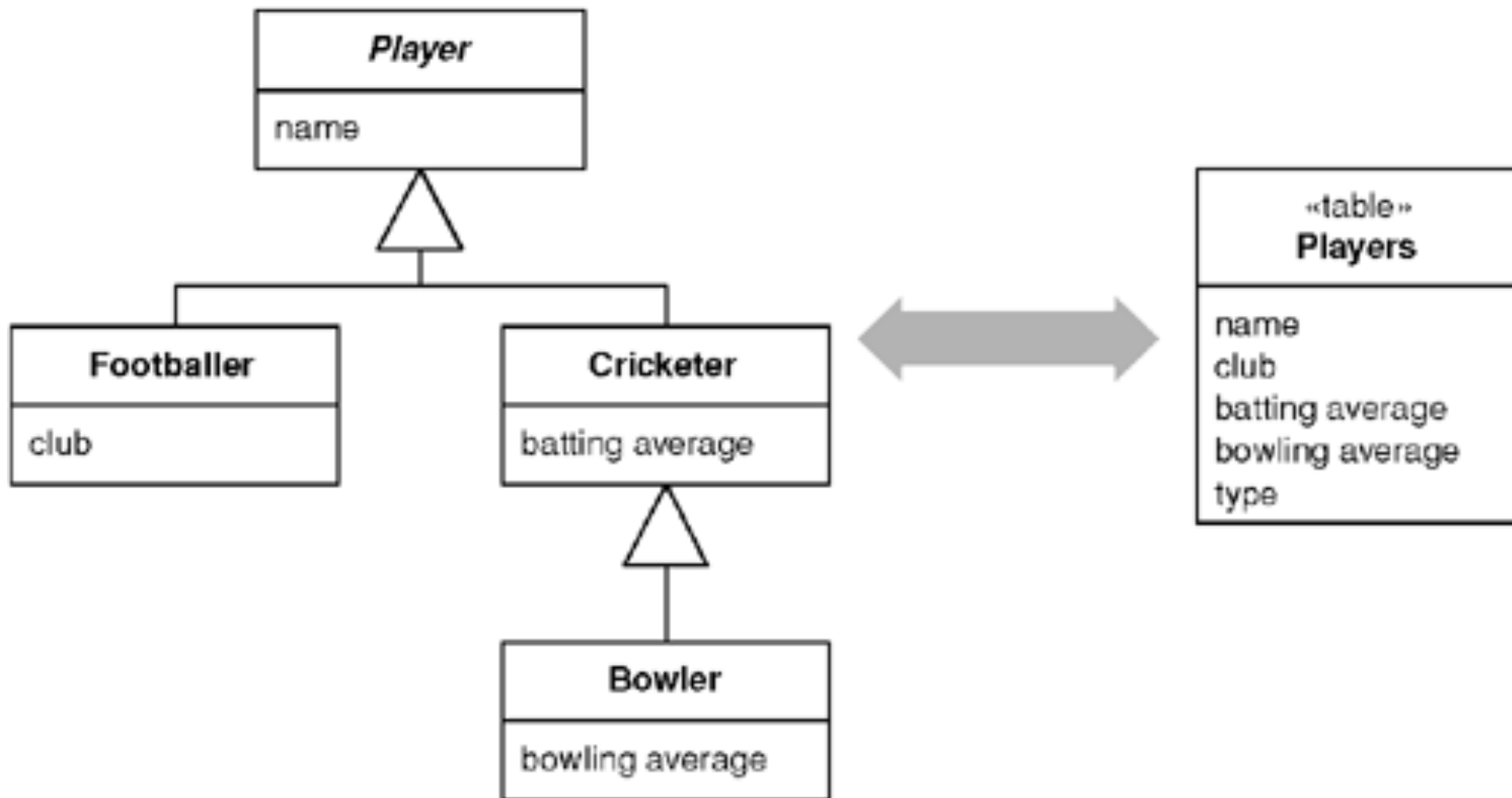
Maparea moștenirii

- Bazele de date relaționale nu suportă moștenirea.
- Programatorul trebuie să mapeze moștenirea dintre entitățile din modelul orientat obiect într-o bază de date relațională.
- Tehnici:
 - Maparea ierarhiei de clase într-o singură tabelă.
 - Maparea fiecărei clase într-o tabelă.
 - Maparea fiecărei clase concrete în tabela ei.
 - Maparea claselor într-o structura de tabele generică.

Moștenirea - O singură tabelă

- Reprezentarea unei ierarhii de clase (moștenire) **ca și o singură tabelă** cu coloane pentru toate atributele din toate clasele din ierarhie.
- Fiecare clasă păstrează informațiile relevante pentru ea într-o înregistrare din tabelă. Coloanele care nu sunt relevante rămân goale.
- Când se încarcă un obiect din tabelă, instrumentul ORM trebuie să știe ce clasă să instanțieze.
- În tabelă se adaugă o coloană care indică ce clasă ar trebui instanțiată (numele clasei sau un cod):
 - Codul trebuie interpretat în codul sursă pentru a putea face maparea cu clasa corespunzătoare.
 - Numele clasei poate fi folosit direct pentru instanțiere (folosind reflecție).

Moștenirea - O singură tabelă



Moștenirea - O singură tabelă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B", 21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H", 12, 23);
```

Players

PK	Name	Club	BattlingAvg	BowlingAvg	Type
1	A A	ABC			Footballer
2	C C		23		Cricketer
3	B B		21	47	Bowler
4	D D	BGD			Footballer
5	H H		12	23	Bowler

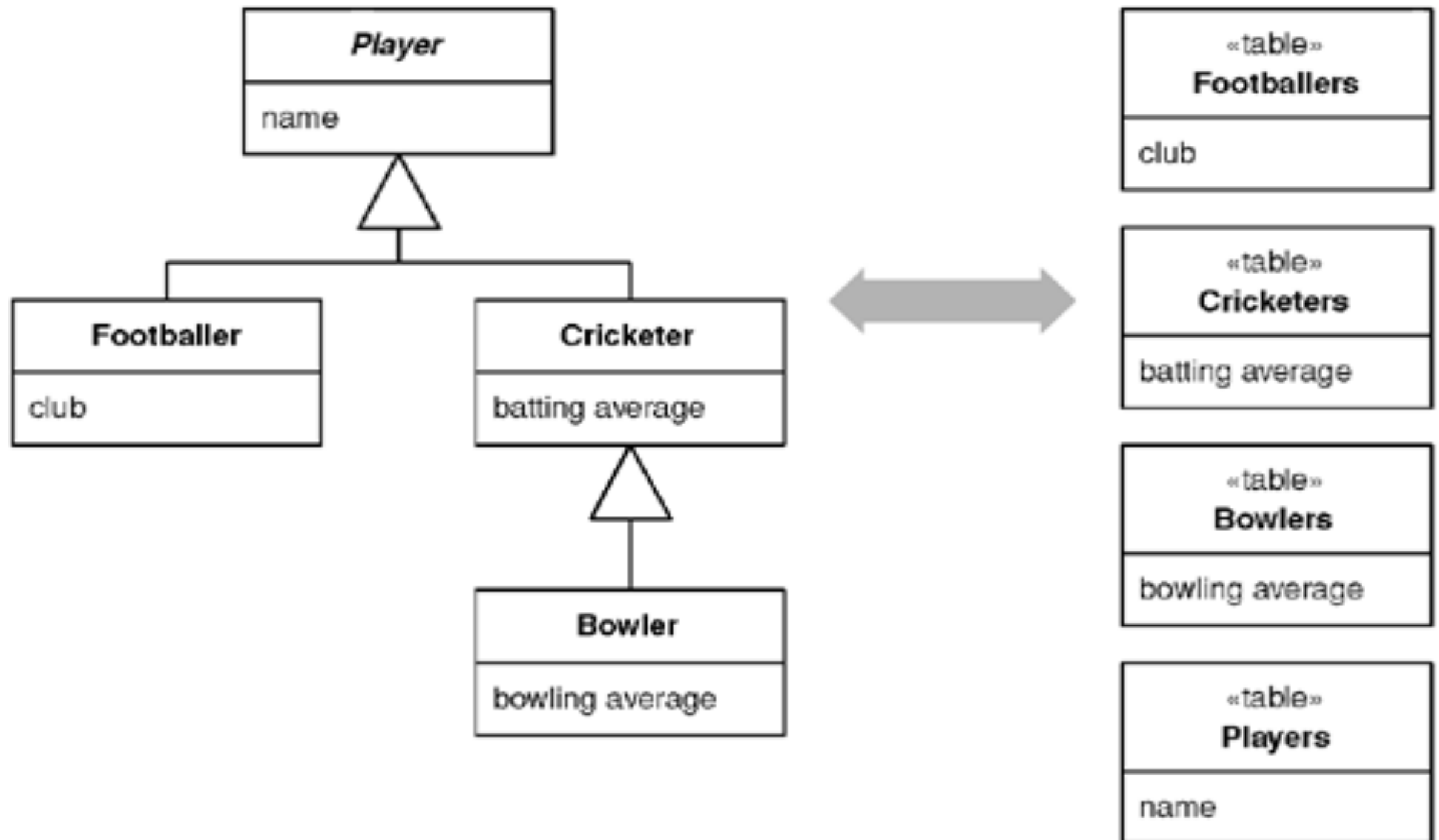
Moștenirea - O singură tabelă

- Avantaje:
 - Există doar o singură tabelă în baza de date.
 - Nu e nevoie de operații join pentru regăsirea informației.
 - Orice refactorizare care mută attributele în ierarhie nu necesită modificarea bazei de date.
- Dezavantaje:
 - Nu toate câmpurile din tabelă sunt relevante (depinde de tipul clasei). Este confuz pentru cei care folosesc tabelele direct (fără instrumentul ORM).
 - Coloanele folosite doar de subclase duc la spațiu nefolosit (multe coloane goale).
 - Tabela poate deveni prea mare, cu mulți indecși și blocări dese ale tablei. Poate afecta performanța.
 - Există un singur spațiu de nume pentru câmpuri, dezvoltatorul trebuie să se asigure că se vor folosi nume diferite în tabelă.
 - Adăugarea numelui clasei (prefix, postfix) poate ajuta. (ex. NumeClasă_NumeProprietate)

Tabelă pentru fiecare clasă

- Fiecare clasă din ierarhie are tabela ei.
- Atributele din clasă se mapează direct la coloanele corespunzătoare din tabelă.
- *Problemă: Cum se leagă înregistrările din tabele?*
 - *Soluția A:* folosirea cheii primare atât în tabela corespunzătoare clasei de bază cât și în clasa derivată. Deoarece clasa de bază are câte o înregistrare pentru fiecare înregistrare din clasele derivate, cheia primară va fi unica între toate tabele.
 - *Soluția B.* Fiecare tabelă să aibă cheia primară proprie, și folosirea cheii străine pentru a păstra legătura cu tabela corespunzătoare clasei de bază.
- *Provocare: încărcarea/regăsirea informațiilor din mai multe tabele în mod eficient.*
 - Operații de join între diferite tabele
 - Operațiile de join între mai mult de 3 sau 4 tabele sunt lente din cauza modului în care bazele de date optimizează operațiile interne.
- *Interogările asupra bazei de date sunt dificile.*

Tabelă pentru fiecare clasă



Tabelă pentru fiecare clasă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B",21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H",12, 23);
```

Players

PK	Name
1	A A
2	C C
3	B B
4	D D
5	H H

Footballers

PK	Club
1	ABC
4	BGD

Cricketers

PK	BattlingAvg
2	23
3	21
5	12

Bowlers

PK	BowlingAvg
3	47
5	23

Tabelă pentru fiecare clasă

- Avantaje:

- Toate coloanele sunt relevante pentru fiecare înregistrare, tabelele sunt mai ușor de înțeles și nu se folosește spațiu în mod ineficient.
- Relația dintre entitățile din modelul orientat obiect și baza de date relațională este ușor de înțeles.

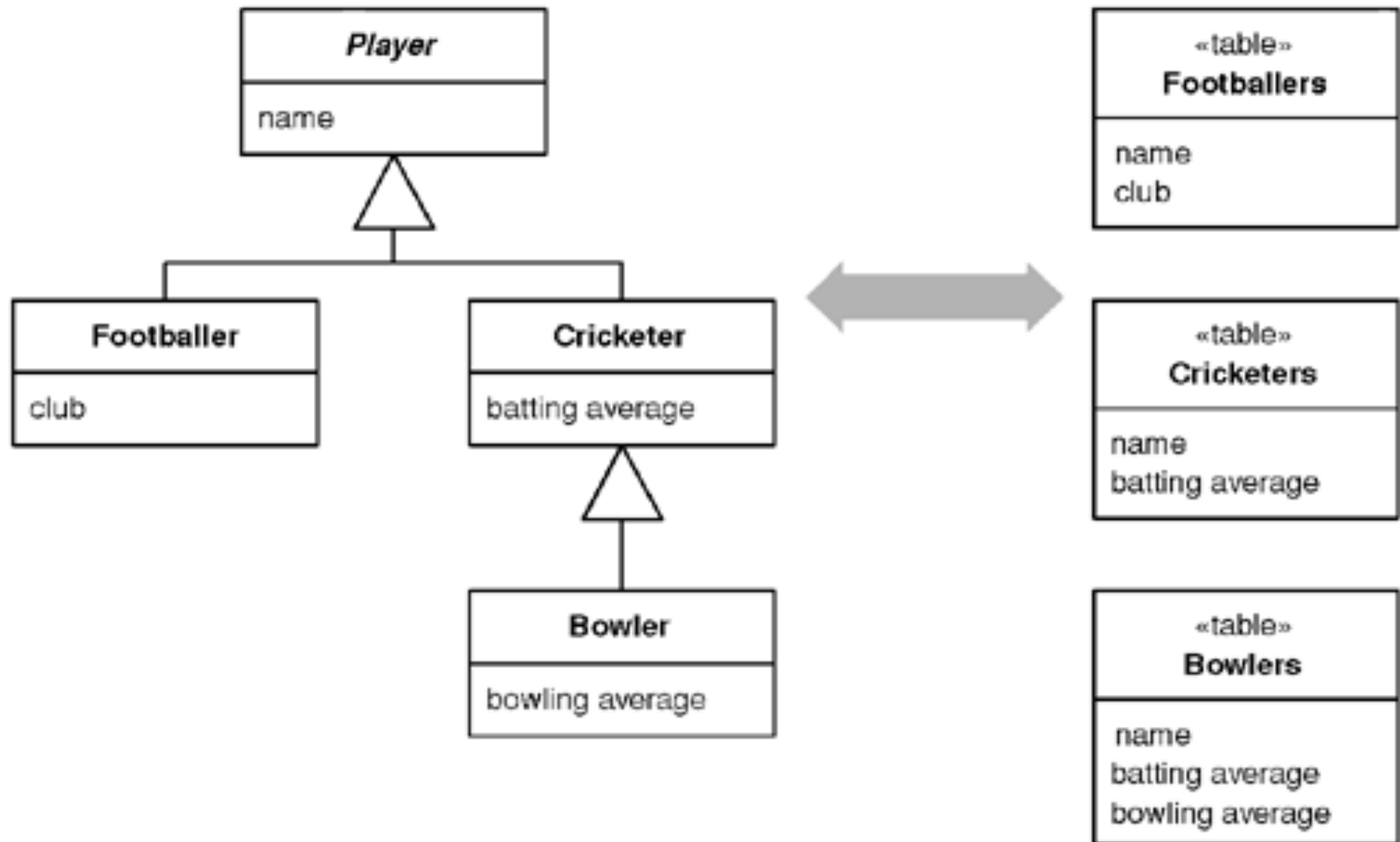
- Dezavantaje:

- Este necesară folosirea mai multor tabele pentru a încărca un obiect din mediu **persistent** (operație join sau mai multe interogări și folosirea memoriei).
- Orice refactorizare (mutarea câmpurilor în ierarhia de clase) cauzează modificarea structurii bazei de date.
- Tabelele corespunzătoare claselor de bază pot cauza probleme de performanță din cauza accesării dese.
- Normalizarea poate duce la înțelegerea dificilă a interogărilor ad-hoc.

Tabelă pentru fiecare clasă concretă

- Fiecare clasă concretă (non-abstract) din ierarhie are tabela ei.
- Fiecare tabelă conține coloane pentru toate proprietățile din ierarhie până la ea. ***Atributele din clasa de bază sunt duplicate în tabelele corespunzătoare subclaselor.***
- Este responsabilitatea programatorului de a se asigura că cheile sunt unice nu doar în tabela corespunzătoare clasei dar și între toate tabelele asociate ierarhiei.

Tabelă pentru fiecare clasă concretă



Tabelă pentru fiecare clasă concretă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B",21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H",12, 23);
```

Footballers

PK	Name	Club
1	A A	ABC
4	D D	BGD

Cricketers

PK	Name	BattlingAvg
2	C C	23

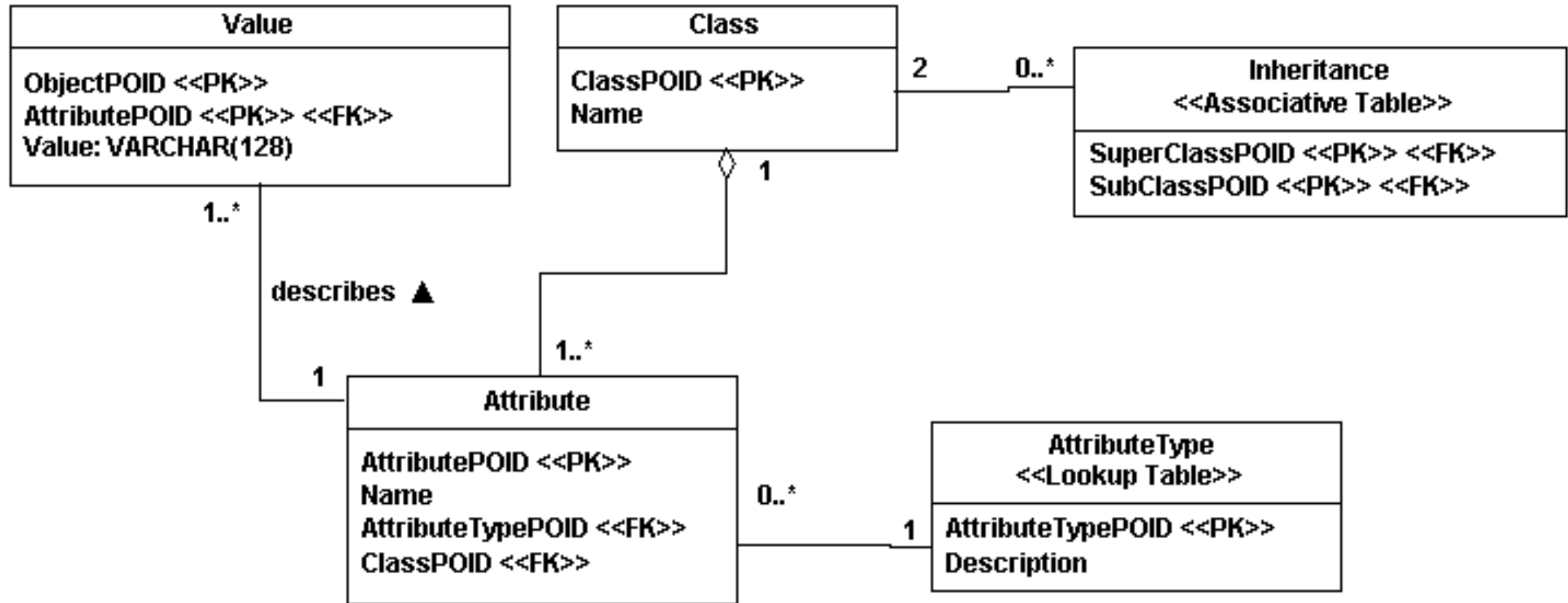
Bowlers

PK	Name	BattlingAvg	BowlingAvg
3	B B	21	47
5	H H	12	23

Tabelă pentru fiecare clasă concretă

- **Avantaje:**
 - Fiecare tabelă păstrează toate informațiile relevante și nu are câmpuri irelevante. Este ușor de înțeles și de alte aplicații care nu folosesc obiecte.
 - Nu este nevoie de operații join pentru citirea datelor.
 - Fiecare tabelă este accesată doar când clasa respectivă este accesată. Performanța este mai bună.
- **Dezavantaje:**
 - Gestiunea dificilă a cheilor primare.
 - Nu pot fi constrânse relațiile către clasele abstracte.
 - Dacă câmpurile din modelul obiectual sunt mutate în ierarhie, trebuie modificate definițiile tabelelor.
 - Dacă se modifică un câmp dintr-o clasă de bază, trebuie modificate toate tabelele corespunzătoare subclaselor, pentru că aceste câmpuri sunt duplicate.
 - O operație de căutare folosind clasa de bază, necesită căutări în toate **tabelele** (accesări multiple ale bazei de date sau o operație de join complicată).

Tabelle generiche



Tabele generice

- Avantaje:
 - Poate fi extinsă pentru a oferi suport pentru o gamă largă de mapări, inclusiv maparea relațiilor.
 - Este flexibilă, permite modificarea ușoară a modului în care sunt păstrate obiectele (trebuie modificate doar metadatele din tabelele *Class*, *Inheritance*, *Attribute* și *AttributeType*).
- Dezavantaje:
 - Este fezabilă doar pentru date de dimensiuni mici, deoarece necesită accesări dese ale bazei de date doar pentru reconstruirea unui singur obiect).
 - Interogările pot fi dificile deoarece necesită accesarea mai multor înregistrări pentru un singur obiect.

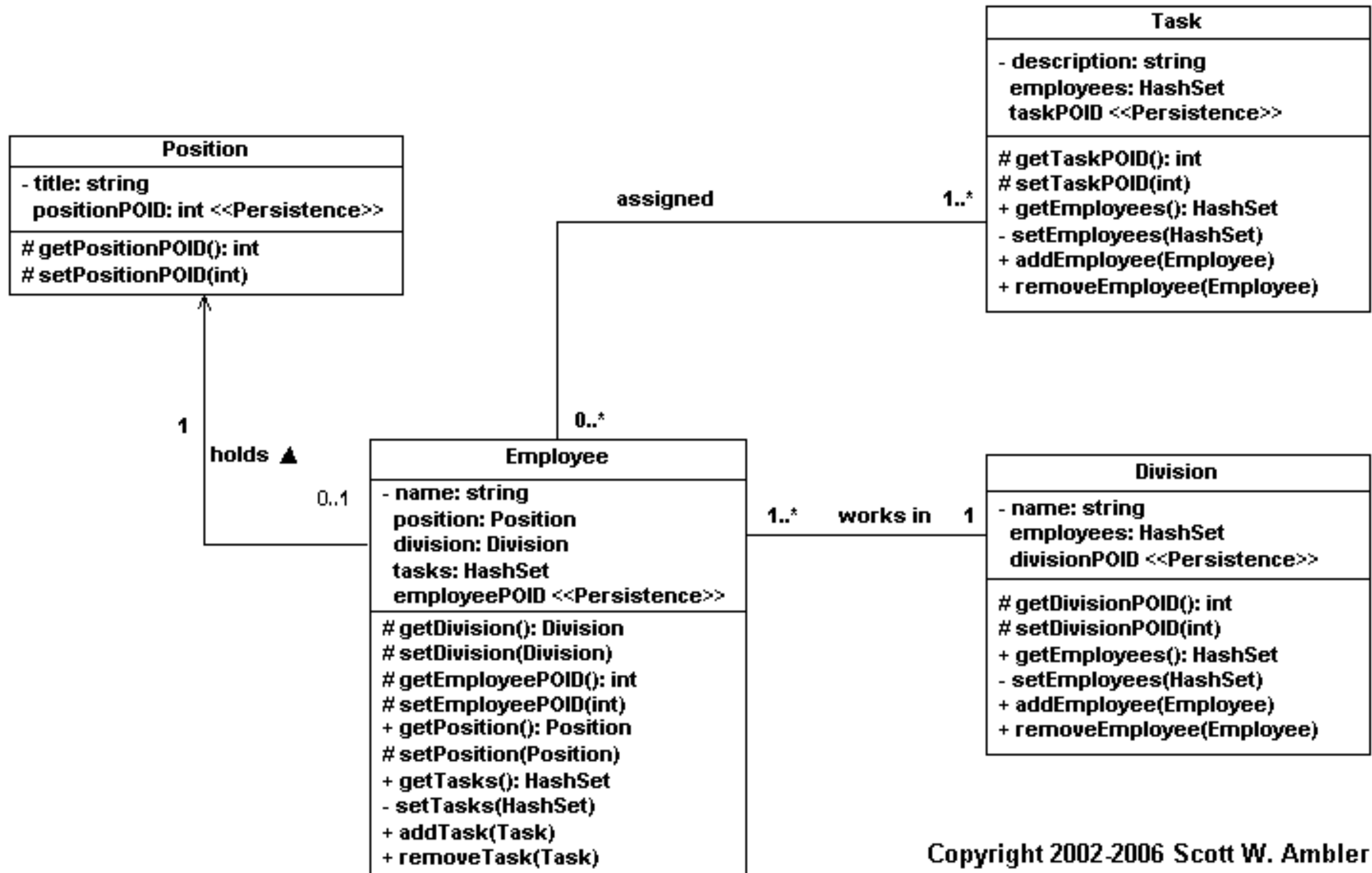
Maparea relațiilor dintre obiecte

- *Problema*: modul în care paradigma orientată obiect și modelul relațional tratează legăturile dintre obiecte/înregistrari, care cauzează două situații.
 1. *Diferența reprezentării*. Obiectele tratează legăturile prin păstrarea referințelor care există în timpul execuției (**referințele sunt temporare**). Bazele de date relaționale tratează legăturile prin păstrarea cheilor în tabele (**cheile sunt permanente**).
 2. *Obiectele pot folosi colecții pentru a gestiona mai multe referințe într-un singur atribut*. Normalizarea obligă ca toate legăturile/valorile să nu fie multiple. Se inversează structura de date dintre obiecte și tabele.
- Exemplu: Un obiect *Order* are o colecție de obiecte de tip *line item* care nu păstrează o referință către obiectul de tip order.
 - ★ Structura tablei este inversă, înregistrările *line item* includ o cheie străină către înregistrarea *order* corespunzătoare (câmpurile dintr-o înregistrare nu pot fi multivaloare).

Tipuri de relații

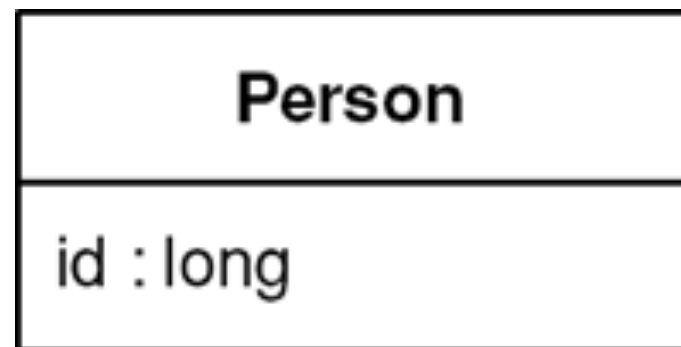
- Două categorii de relații între obiecte importante la mapare:
- **Multiplicitatea**. Include 3 tipuri:
 - Relații *unu-la-unu*. Maximul multiplicității la ambele capete este 1.
 - Relații *unu-la-n (sau n-la-unu)*. Multiplicitatea la unul dintre capete este 1, la celălalt capăt este n.
 - Relații *n-la-n*. Maximul multiplicității la ambele capete este mai mare decât 1.
- **Direcția**. Include 2 tipuri:
 - Relații *unidirecționale*. Un obiect știe de obiectul (obiectele) cu care are o legătură, dar celălalt obiect (celelalte obiecte) nu știe (nu știu) de el.
 - Relații *bidirecționale*. Ambele obiecte aflate într-o relație știu unul de celălalt.

Tipuri de relații



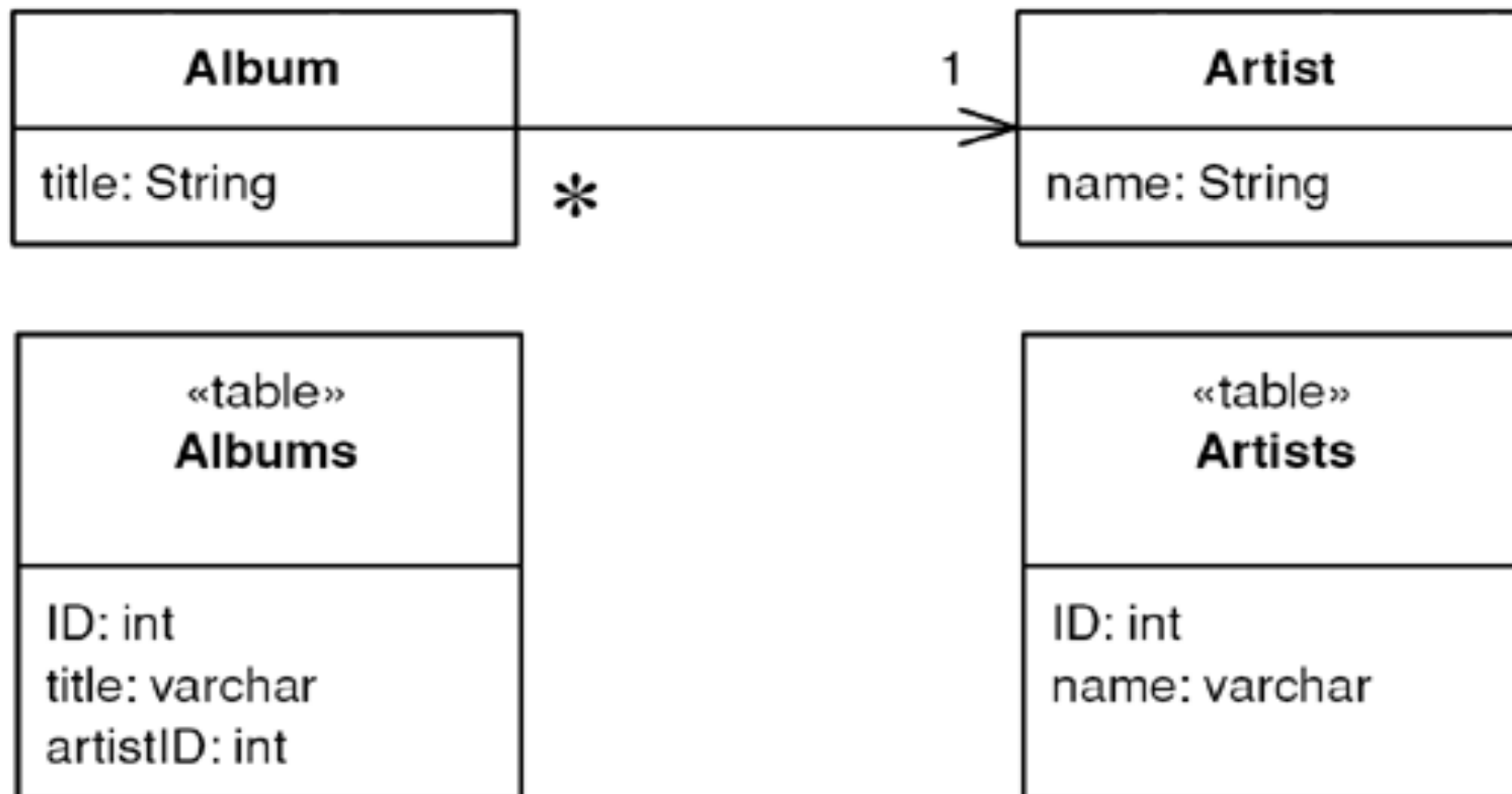
Șablonul *Identity*

- Salvează un identificator (ID) din baza de date într-un obiect pentru a păstra legătura dintre un obiect în memorie și o înregistrare din baza de date.
- Cheia primară dintr-o bază de date relațională este păstrată printre attributele obiectului.
- Șablonul ar trebui folosit când există o mapare între obiectele din memorie și înregistrările din baza de date (cheia primară este diferită de attributele din obiect).



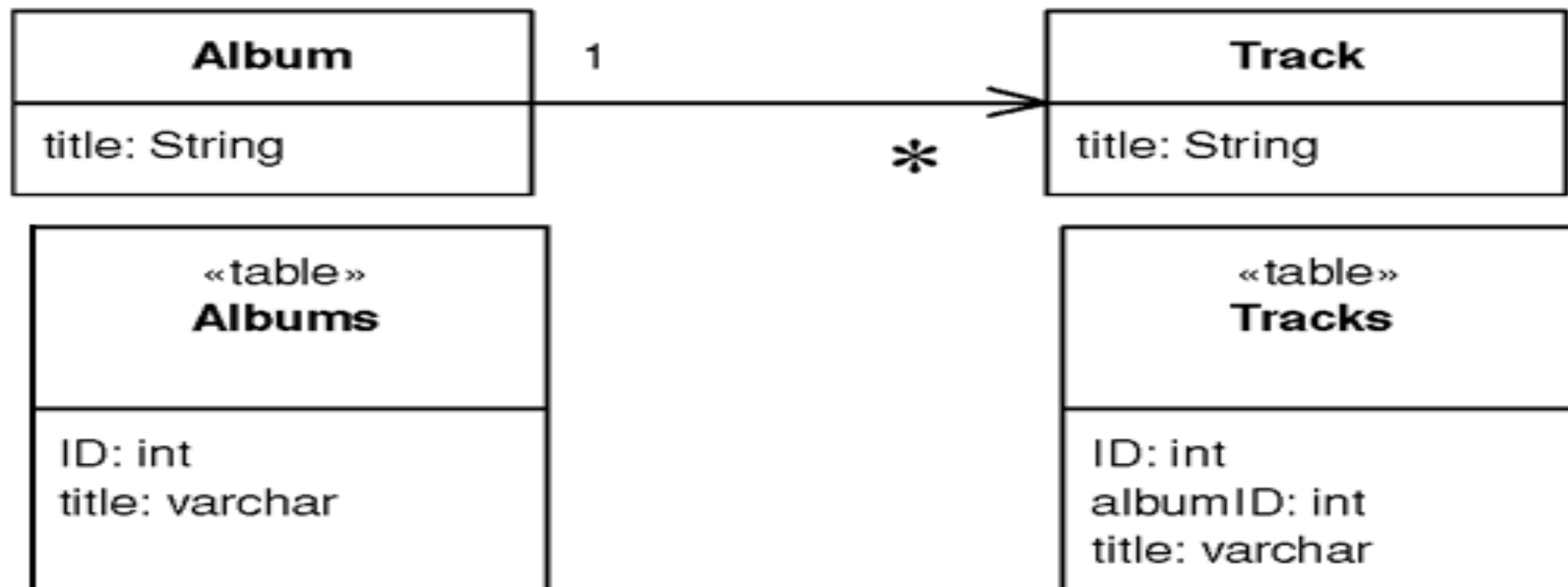
Maparea cheii străine

- Mapează o asociere dintre obiecte ca și cheie străină între tabelele dintr-o bază de date relațională.
- Fiecare obiect conține cheia din tabela corespunzătoare.
- Dacă două obiecte sunt legate printr-o relație de asociere, relația poate fi înlocuită printr-o cheie străină în baza de date.



Maparea cheii străine

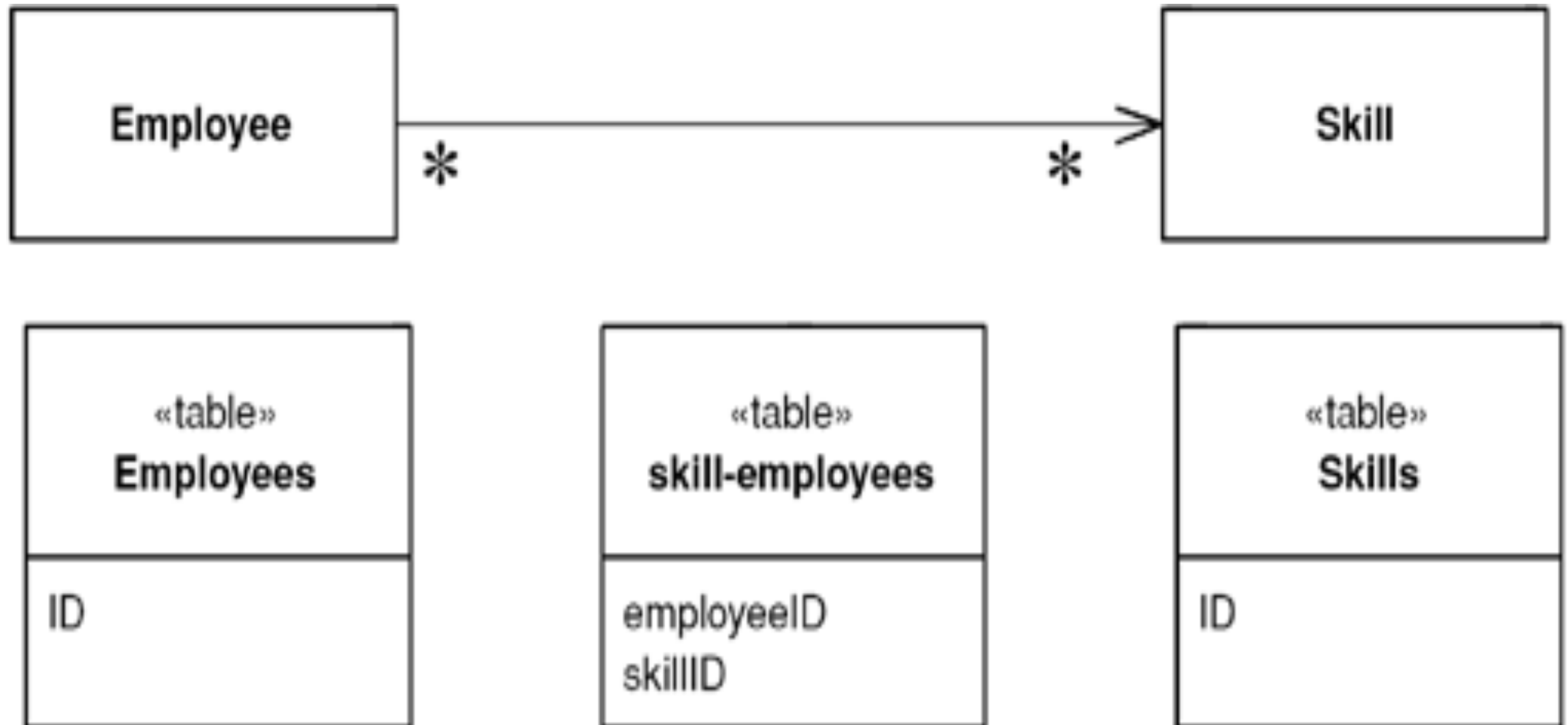
- Maparea unei colecții de obiecte.
- Într-o bază de date relațională nu se poate salva o colecție, trebuie inversată direcția referinței.
- Maparea cheii străine poate fi folosită pentru aproape toate asocierile dintre clase. Nu poate fi folosită pentru asocierea *n-la-n*.



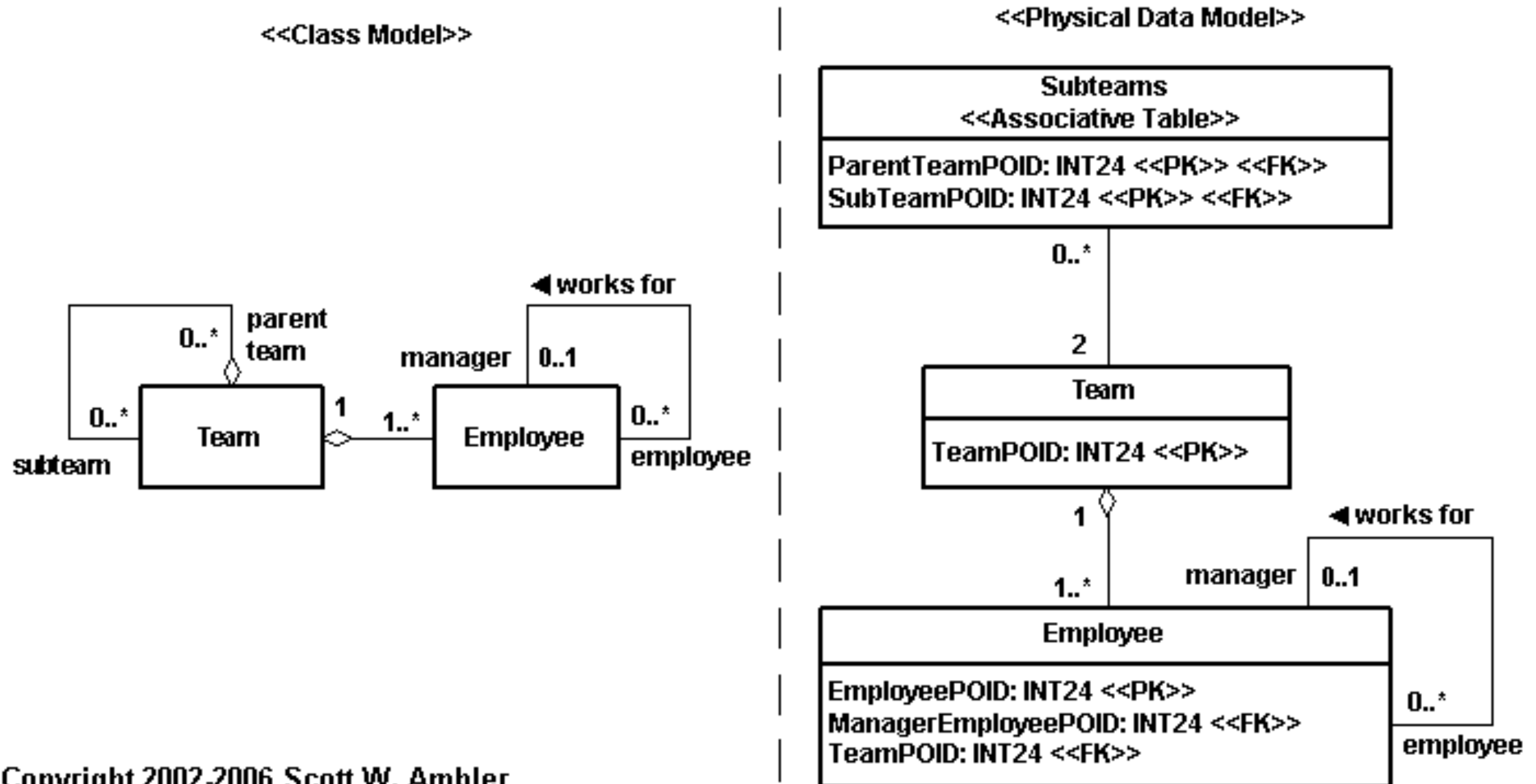
Maparea folosind o tabelă de asociere

- Salvează o asociere ca o tabelă cu chei străine către tabelele legate prin asociere.
- Obiectele pot păstra mulțimi de valori folosind colecții. Bazele de date relaționale nu au această caracteristică și sunt restrictionate la câmpuri cu o singură valoare.
- Ideea este de a crea o tabelă de legătură/asociere pentru a stoca asocierea.
- Tabela are doar două coloane corespunzătoare cheilor străine, conține câte o înregistrare pentru fiecare pereche de obiecte asociate.
- Tabela de legătura nu are echivalentul unui obiect în memorie (nu are ID). Cheia primară este compusă din cheile primare ale celor două tabele asociate.
- Tabela de asociere este folosită cel mai des pentru maparea asocierii n-la-n, dar poate fi folosită și pentru alte tipuri de asocieri (mai dificil, complex).

Tabela de asociere



Maparea asocierilor recursive



Maparea proprietăților statice

TableA

A
<u>StaticAttributeA</u> name

PK	Name	StaticAttributeA
	AA	1
	BB	2
	CC	1

TableA

PK	Name
	AA
	BB
	CC

StaticTableA

StaticAttributeA
1

Maparea proprietăților statice - Strategii (1)

- O tabelă cu o singură înregistrare, o singură coloană pentru fiecare proprietate statică:
 - Pro: Simplu, acces rapid
 - Con: multe tabele mici

A
<u>StaticAttributeA</u> name

B
<u>StaticAttrB1</u>
<u>StaticAttrB2</u>

C
<u>StaticAttrC1</u>

StaticTableA

StaticAttributeA
1

StaticTableB_1

StaticAttrB1
23

StaticTable_B2

StaticAttrB2
ValueA

StaticTableC

StaticAttrC1
23.56

Maparea proprietăților statice - Strategii (2)

- O tabelă cu mai multe coloane, o singură înregistrare pentru fiecare clasă:
 - Pro: Simplu, acces rapid
 - Con: multe tabele mici, dar mai puține decât la strategia precedentă

A
<u>StaticAttributeA</u> name

StaticTableA

StaticAttributeA
1

B
<u>StaticAttrB1</u> <u>StaticAttrB2</u>

StaticTableB

StaticAttrB1	StaticAttrB2
23	ValueA

C
<u>StaticAttrC1</u>

StaticTableC

StaticAttrC1
23.56

Maparea proprietăților statice - Strategii (3)

- O singură tabelă cu mai multe coloane - o singură înregistrare pentru toate clasele:
 - Pro: număr minim de tabele introdus
 - Con: potențiale probleme de concurență dacă mai multe clase trebuie să acceseze datele în același timp.

A
<u>StaticAttributeA</u> name

B
<u>StaticAttrB1</u> <u>StaticAttrB2</u>

C
<u>StaticAttrC1</u>

StaticAttributesTable

A.StaticAttributeA	B.StaticAttrB1	B.StaticAttrB2	C.StaticAttrC1
1	23	ValueA	23.56

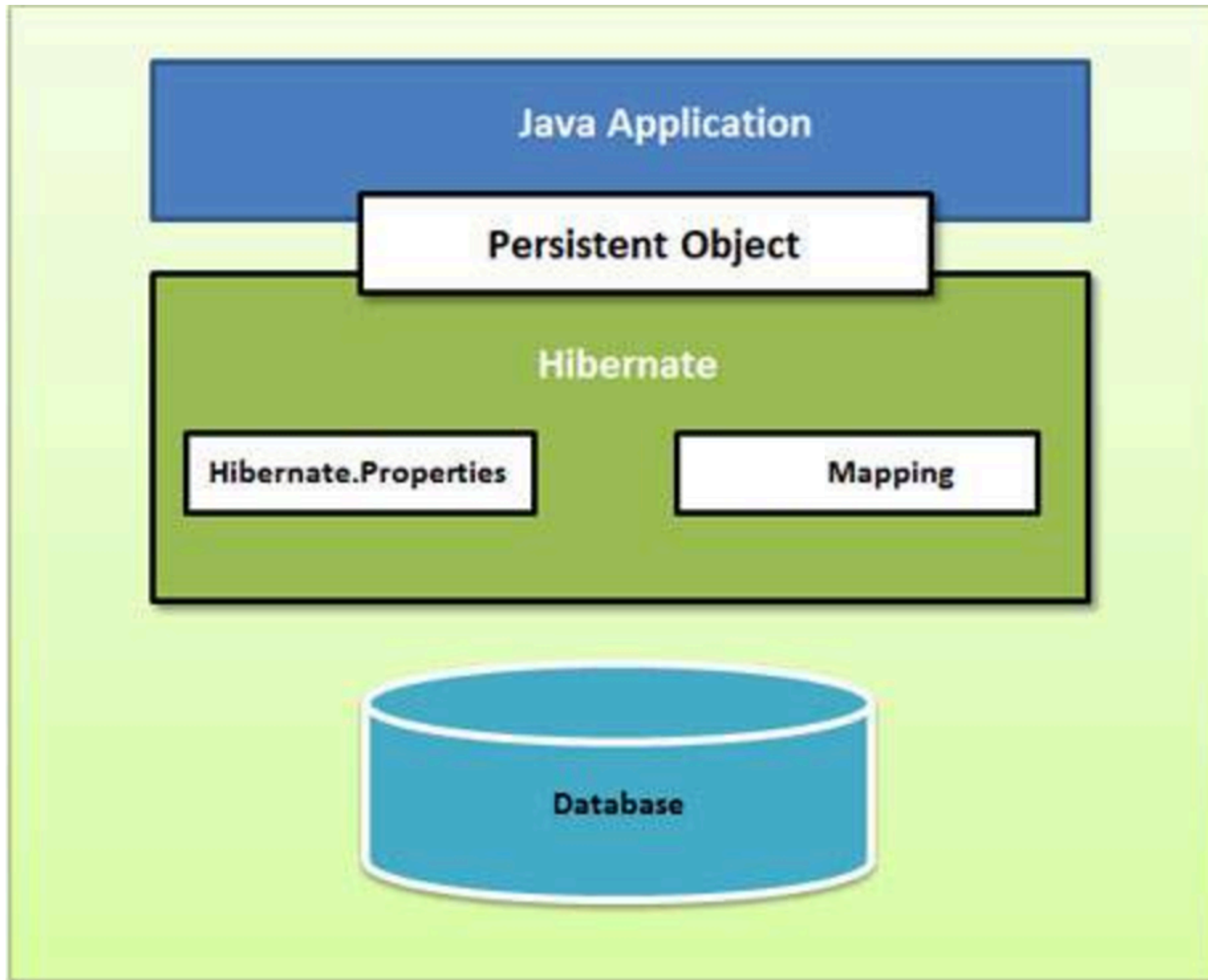
Maparea proprietăților statice

- Strategii (cont):
 - Schemă generică cu mai multe înregistrări pentru toate clasele:
 - Pro: număr minim de tabele introdus. Reduce problemele legate de concurență.
 - Con: Necesitatea convertirii între tipuri de date. Schema este asociată cu numele claselor și a proprietăților statice.

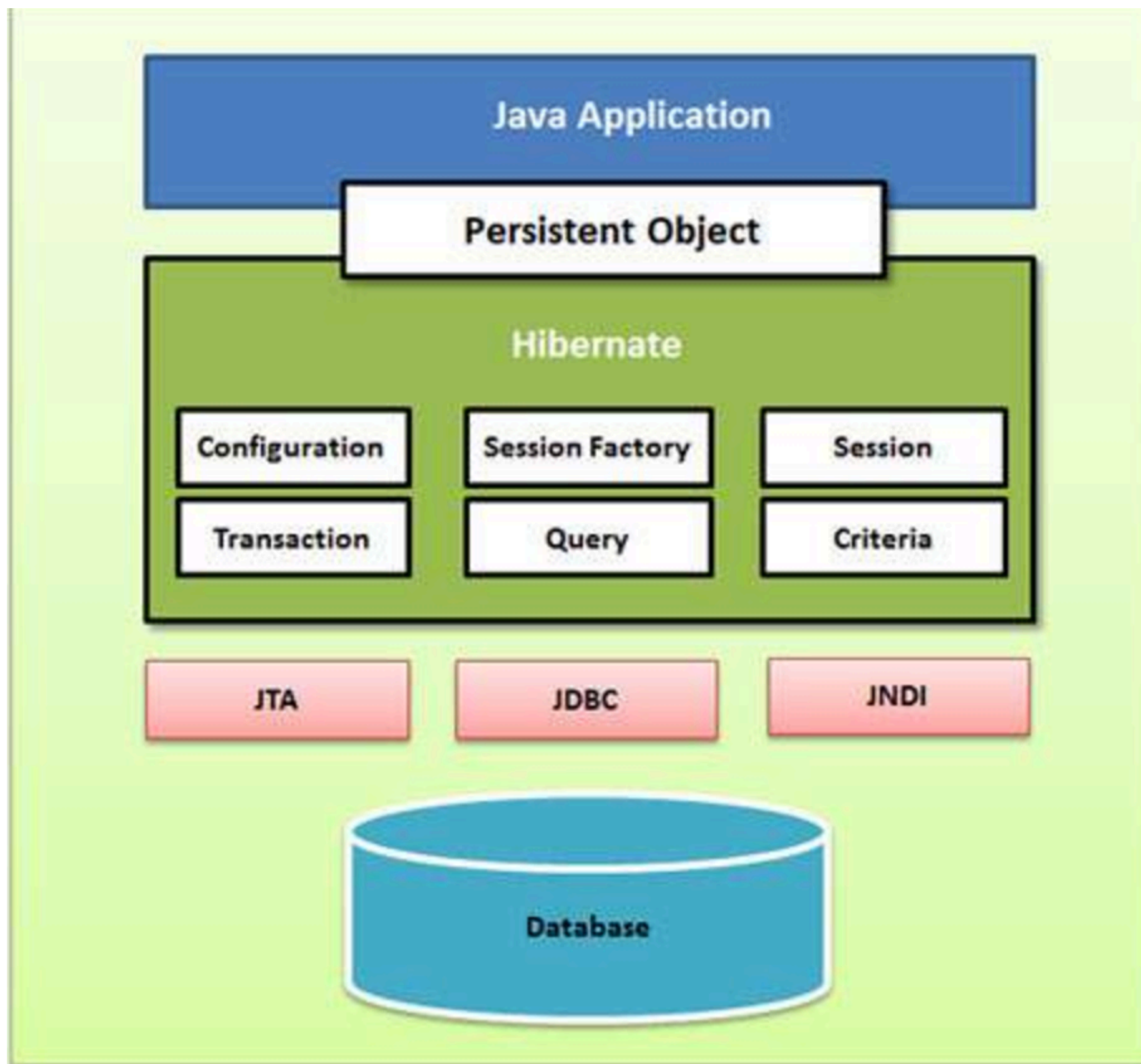
Hibernate

- Instrument open source pentru ORM.
- Aplicațiile care folosesc Hibernate definesc/specifică clasele persistente ce vor fi mapate în tabele într-o bază de date relațională.
- Toate instrucțiunile SQL sunt generate automat în timpul execuției.
- Informațiile despre mapare sunt transmise prin adnotări (vers<6, ca și un document de mapare în format XML).
- Adnotările specifică:
 - cum vor fi mapate proprietățile la coloanele din tabelă (tabele).
 - strategia aleasă de dezvoltator pentru mapare (unde este cazul).

Arhitectura



Arhitectura



Arhitectura

- Interfețele/Clasele definite de Hibernate pot fi clasificate astfel:
 - Interfețe folosite de aplicații pentru a efectua operații CRUD și interogări. Logica aplicației corespunzătoare nivelului de persistență va fi strâns cuplată de acestea: *Session*, *Transaction* și *Query*.
 - Clase apelate de aplicație pentru configurarea instrumentului Hibernate: clasa *Configuration*.
 - Interfețe callback care permit aplicațiilor să reacționeze la evenimente ce apar în timpul execuției instrumentului Hibernate: *Interceptor*, *Lifecycle* și *Validatable*.
 - Interfețe care permit extinderea Hibernate: *UserType*, *CompositeUserType* și *IdentifierGenerator*.
- Hibernate folosește Java API existent: JDBC, Java Transaction API (JTA) și Java Naming and Directory Interface (JNDI).

Interfețe de bază

- Interfața *Session*: este cea mai folosită interfață de către aplicațiile care folosesc Hibernate. O instanță de tip Session **nu consumă multe resurse și poate fi ușor creată/distrusă**.
- Interfața *SessionFactory*. Aplicațiile obțin instanțe de tip Session folosind un obiect de tip SessionFactory. **Păstrează în cache instrucțiunile SQL generate dinamic și alte metadate legate de mapări folosite în timpul execuției.**
- Clasa *Configuration*. Un obiect de tip Configuration este **folosit pentru configurarea și pornirea Hibernate**. Aplicațiile pot folosi acest obiect pentru a **specifica locația bazei de date și a altor proprietăți specifice Hibernate**.
- Interfața *Transaction*. **Abstractizează codul corespunzător unei tranzacții de implementarea specifică folosită**: o tranzacție JDBC, o tranzacție JTA UserTransaction, etc.
- Interfața *Query*. **Permite efectuarea de interogări asupra bazei de date și controlează modul în care este executată interogarea**. Interogările pot fi scrise în HQL sau folosind direct limbajul SQL corespunzător bazei de date.

Exemplu

- Etape:
 - Crearea claselor Java corespunzătoare entităților
 - Adăugarea informațiilor pentru mapare (adnotări)
 - Crearea fișierului de configurare Hibernate
 - Implementarea nivelului de persistență folosind funcțiile corespunzătoare
 - Testarea claselor

Exemplu - Entitatea

@Entity

```
public class Book {  
    private Long id;  
    private String title;  
  
    public Book() {} //obligatoriu  
    public Book(String title){ this.title=title;}  
  
    @Id  
    @GeneratedValue(strategy=IDENTITY)  
    public Long getId() { return id;}  
  
    public void setId(Long id) {this.id = id;}  
  
    @NotNull  
    public String getTitle() { return title;}  
  
    public void setTitle(String title) {this.title = title;}  
}
```


Exemplu - Entitatea

```
@Entity
@Table( name = "Messages")
public class Message {
    @Id
    @Column(name = "idMessage")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "Message_Text")
    private String text;

    public Message() {} //obligatoriu
    public Message(String text) { this.text = text; }

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

    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
}
```

Fișierul de configurare *hibernate.properties*

```
#hibernate.properties -numele implicit căutat de Hibernate
```

```
# URL-ul de conectare la baza de date (JDBC)
```

```
jakarta.persistence.jdbc.url=jdbc:sqlite:/Users/grigo/HibernateExample/hibernate6.db
```

```
# Credentials
```

```
#jakarta.persistence.jdbc.user=
```

```
#jakarta.persistence.jdbc.password=
```

```
jakarta.persistence.schema-generation.database.action=update
```

```
#none/create/etc
```

```
# SQL statement logging
```

```
hibernate.show_sql=true
```

```
hibernate.format_sql=true
```

```
hibernate.highlight_sql=true
```

Exemplu – Salvarea unei entități

```
//INSERT
```

```
void addBook(Book b) {  
    sessionFactory.inTransaction(session -> {  
        session.persist(b);  
        System.out.println("Book added "+b);  
    });  
}
```

```
//Book b=new Book("Test Book ");
```

```
//dupa rulare
```

```
//Book added Book{id='8', title='Test Book '}
```

Exemplu – Modificarea unei entități

```
//FindOne
Book findBook(int id){
    try(Session session=sessionFactory.openSession()){
        return session.find(Book.class, id);
    } //se apelează session.close()
    catch (Exception e){
        System.err.println("Eroare la findBook " +e);
    }
    return null;
}

//update
Book update(int id){
    Book b=findBook(id);
    b.setTitle("New Title");
    sessionFactory.inTransaction(session->{
        session.merge(b);
        //session.flush()
    });
    return b;
}
```

Exemplu – ștergerea unei entități

```
//DELETE  
void delete(Book book) {  
    sessionFactory.inTransaction(session -> {  
        session.remove(book) ;  
    }) ;  
}
```

Exemplu – selectarea unei entități

```
//SELECT
```

```
List<Book> getBooks() {  
    try( Session session= sessionFactory.openSession()) {  
        return session.createQuery("from Book where title like 'Test %'",  
Book.class).setFirstResult(0).  
            setMaxResults(5).list();  
    } //se apelează session.close()  
}
```

```
//Book - numele entității, NU a tabelii
```

```
//title - proprietate a clasei Book, NU numele coloanei din tabela
```

Exemplu – MainBook

```
public class MainBook {
    static SessionFactory sessionFactory;
    public static void main(String[] args) {
        // A SessionFactory is set up once for an application!
        sessionFactory = new Configuration() //cauta fisierul hibernate.properties
            .addAnnotatedClass(Book.class)
            .buildSessionFactory();

        Book b=new Book("Test Book ");
        // persist an entity
        addBook(b);

        System.out.println("Searching book having id 1 "+findBook(1));
        update(6);

        // query data using HQL
        for(Book bb:getBooks())
            System.out.println(bb);

        //delete an entity
        delete(b);

        //close sessionFactory
        sessionFactory.close();
    }
    //... codul metodelor addBook, findBook, update, getBooks, delete
}
```

Configurare Gradle

```
plugins {  
    id 'java'  
    id 'application'  
}  
  
group = 'ro.mpp'  
version = '1.0'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
  
    implementation 'org.hibernate.orm:hibernate-core:6.4.4.Final'  
  
    // Hibernate Validator  
    implementation 'org.hibernate.validator:hibernate-validator:8.0.0.Final'  
    implementation 'org.glassfish:jakarta.el:4.0.2'  
  
    //pentru lucrul cu diferite baze de date: Sqlite, MariaDb, etc...  
    implementation 'org.hibernate.orm:hibernate-community-dialects:6.4.4.Final'  
  
    testImplementation platform('org.junit:junit-bom:5.9.1')  
    testImplementation 'org.junit.jupiter:junit-jupiter'  
  
    // driverul pentru conectarea la o baza de date Sqlite  
    runtimeOnly 'org.xerial:sqlite-jdbc:3.45.3.0'  
}
```


Interogări ale bazei de date

- Două posibilități:

- Hibernate Query Language

```
session.createQuery("from Category c where c.name like  
    'Laptop%'");
```

- SQL

```
session.createNativeQuery(  
    "select {c.*} from CATEGORY {c} where NAME like 'Laptop%'",  
    "c", Category.class);
```

Obținerea rezultatelor

- Metodele `list()/getResultList()` execută interogarea și returnează rezultatul ca și o listă:

```
List<User> result = session.createQuery("from User",  
    User.class).getResultList();
```

- Un singur obiect ca și rezultat :

```
Bid maxBid =session.createQuery("from Bid b order by  
    b.amount desc", Bid.class)  
    .setMaxResults(1)  
    .uniqueResult();
```

Interogări cu parametri

- Parametrii cu nume

```
String queryString = "from Item item where item.description  
    like :searchString and item.date > :minDate";  
List result = session.createQuery(queryString)  
    .setParameter("searchString", searchS)  
    .setParameter("minDate", minD).list();
```

- Parametrii cu poziție:

```
String queryString = "from Item item where item.description  
    like ?1 and item.date > ?2";  
List result = session.createQuery(queryString)  
    .setParameter(1, searchString)  
    .setParameter(2, minDate)  
    .list();
```

Hibernate Query Language (HQL)

Suportă aproape toate funcțiile și operațiile SQL:

- from clause: `from Cat as cat`
- select clause: `select foo from Foo foo, Bar bar where foo.startDate = bar.date`
- where clause: `from Cat as cat where cat.name='Fritz'`
- aggregate functions:

```
select cat.color, sum(cat.weight), count(cat) from Cat cat
group by cat.color
```

- order by clause
- group by clause
- expressions
- etc.

Maparea relațiilor

Adnotări pentru maparea relațiilor dintre clase:

- **@ManyToMany**
- **@ManyToOne**
- **@OneToOne**
- **@JoinTable** - tabela de asociere
- **@JoinColumn**

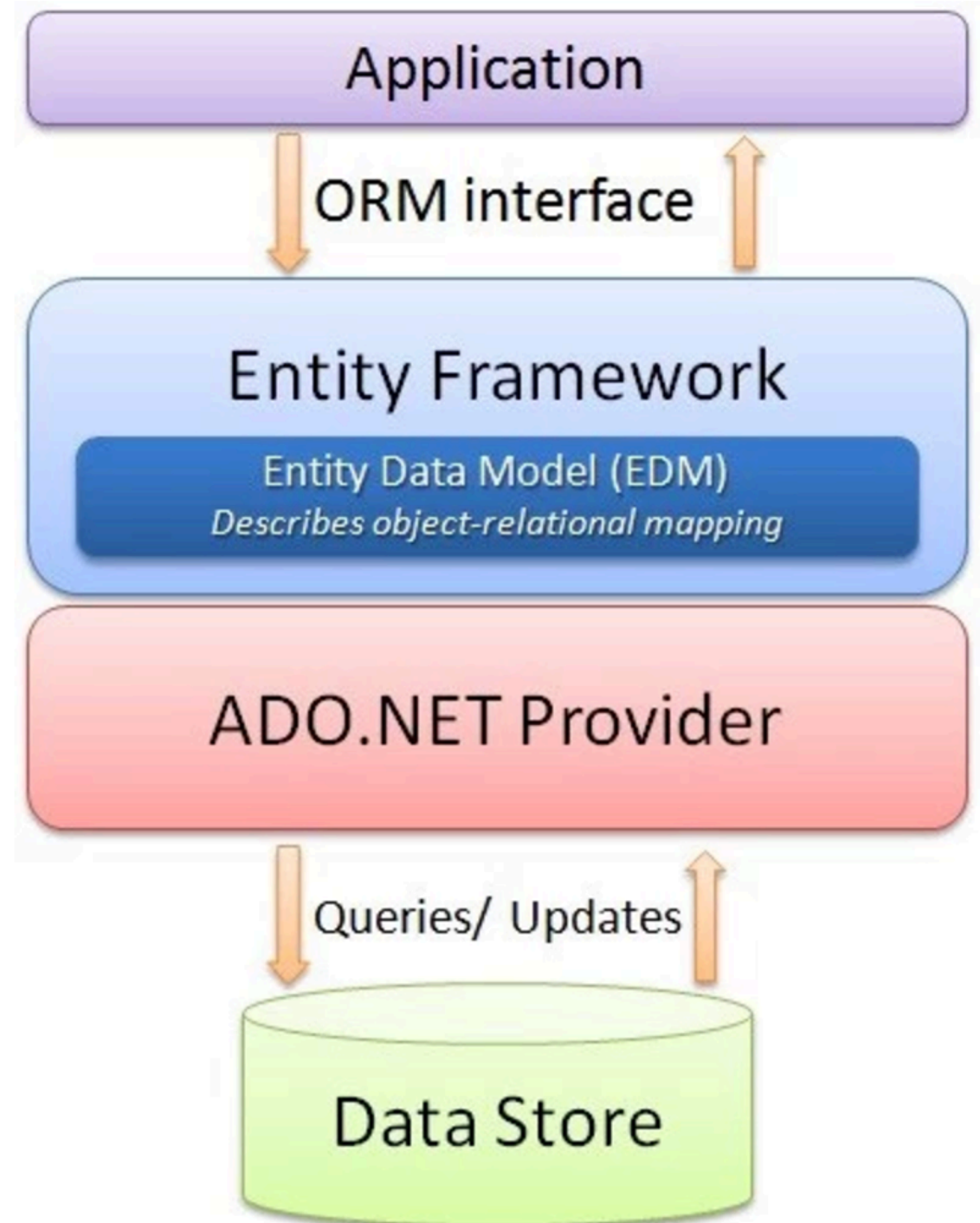
Fetching - **lazy**, **eager**

https://docs.jboss.org/hibernate/orm/6.3/introduction/html_single/Hibernate_Introduction.html

Exemplu Hibernate

.NET ORM

- Instrumente ORM bazate pe LINQ:
 - [LINQ to SQL](#) (L2S)
 - [Entity Framework](#) (EF)
- EF permite o mai bună decuplare a claselor de modelul relațional



.NET L2S

- Decorarea claselor folosind attribute .NET
- Spațiul de nume `System.Data.Linq.Mapping`

```
[Table (Name="Customers")]  
public class Customer  
{  
    [Column(IsPrimaryKey=true)]  
    public int ID {get;set};  
    [Column (Name="FullName")]  
    public string Name {get; set};  
}
```


.NET L2S

```
var context = new DataContext ("database connection string");

Table<Customer> customers = context.GetTable <Customer>();

// numărul de înregistrări din tabelă.
Console.WriteLine (customers.Count());

// Clientul cu Id-ul 2.
Customer cust = customers.Single (c => c.ID == 2);

Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SubmitChanges();
```

.NET EF

- Decorarea claselor folosind attribute .NET
- Referință către [System.Data.Entity.dll](#)

```
[EdmEntityType (NamespaceName = "EFModel", Name = "Customer")]  
public partial class Customer  
{  
    [EdmScalarPropertyAttribute (EntityKeyProperty=true,  
        IsNullable=false)]  
    public int ID { get; set; }  
  
    [EdmScalarProperty (EntityKeyProperty = false, IsNullable = false)]  
        public string Name { get; set; }  
}
```

.NET EF

```
var context = new ObjectContext ("entity connection string");
    context.DefaultContainerName = "EntitiesContainer";
    ObjectSet<Customer> customers =
    context.CreateObjectSet<Customer>();

// numărul de înregistrări din tabelă
    Console.WriteLine (customers.Count());

// Clientul cu ID-ul 2
Customer cust = customers.Single (c => c.ID == 2);

Customer cust = customers.OrderBy (c => c.Name).First();
    cust.Name = "Updated Name";
    context.SaveChanges();
```