Systems for Design and Implementation

2015-2016

Course 7

Contents

- ORM
- Hibernate/NHibernate

Object/Relational Mapping (ORM)

- Object-relational mapping (ORM, O/RM, and O/R mapping) is a programming technique for converting data between object oriented type systems and relational databases.
- The principle of object-relational mapping is to delegate to tools the management of persistence, and to work at code-level with objects representing a domain model, and not with data structures in the same format as the relational database.
- Object-relational mapping tools establish a bidirectional link with data in a relational database and objects in code, based on a configuration and by executing SQL queries (dynamic most of the time) on the database.

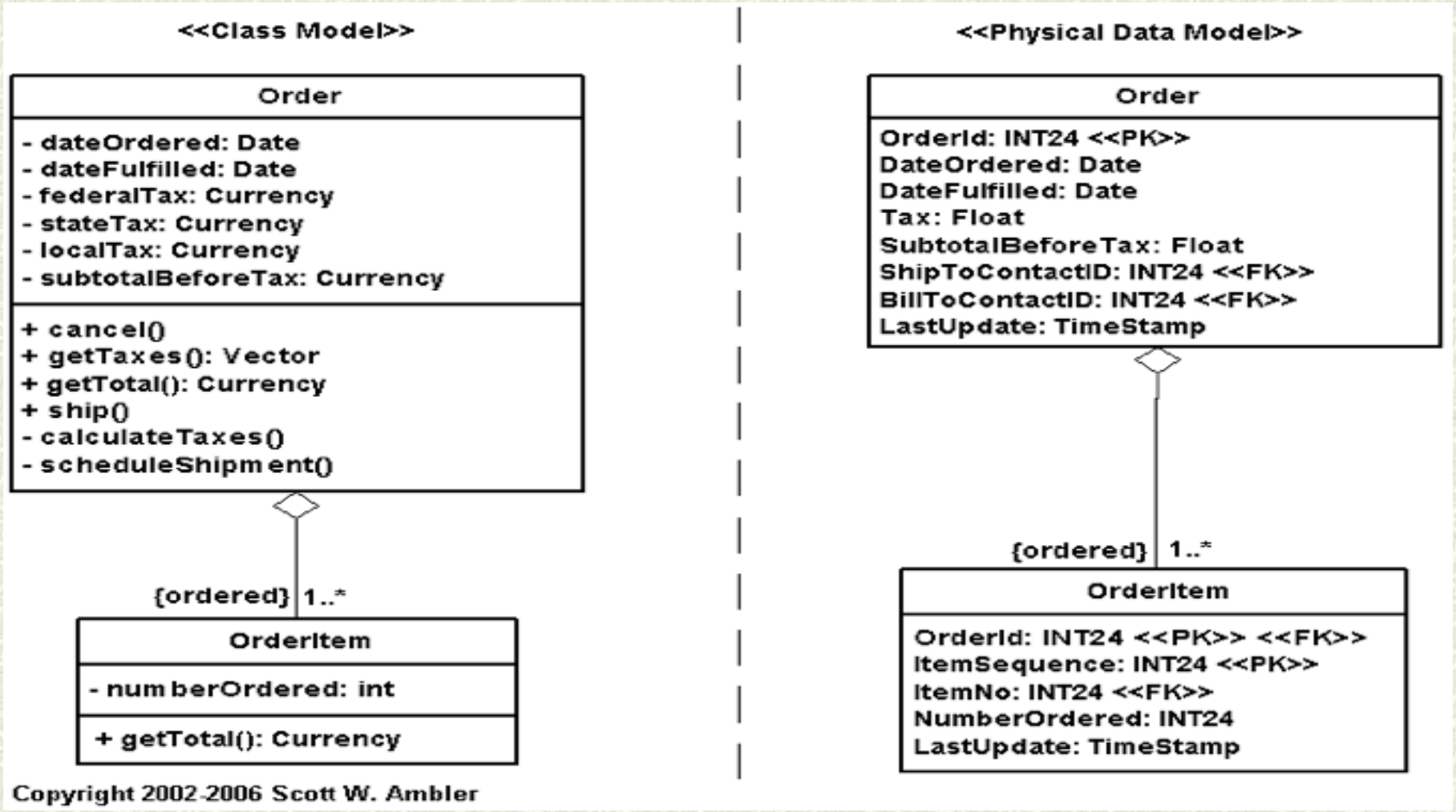
Mapping Terminology

- *Mapping*. The act of determining how objects and their relationships are persisted in permanent data storage, in this case relational databases.
- *Property*. A data attribute, either implemented as a physical attribute such as the `string firstName` or as a virtual attribute implemented via an operation such as `getTotal()` which returns the total of an order.
- *Property mapping*. A mapping that describes how to persist an object's property.
- *Relationship mapping*. A mapping that describes how to persist a relationship (association, aggregation, composition, inheritance) between two or more objects.

Simple Mappings

- The simplest case is the one in which a class maps to a table.
- However, except for very simple domain models, a one-to-one mapping of classes to tables is very rare.
- The easiest mapping is a property mapping of a single attribute to a single column. It is even simpler when both (the attribute and the column) have the same basic types:
  - they are both dates,
  - the attribute is a string and the column is a char,
  - the attribute is a number and the column is a float.

Simple Mappings



Differences

- There are several attributes for **`tax`** in the object schema yet only one in the data schema. The three attributes for tax in the **`Order`** class presumably should be added up and stored in the tax column of the **`Order`** table when the object is saved.
- The data schema indicates keys whereas the object schema does not. Rows in tables are uniquely identified by primary keys and relationships between rows are maintained through the use of foreign keys.
- Relationships to objects, on the other hand, are implemented via references to those objects not through foreign keys. The implication is that in order to fully persist the objects and their relationships, the objects need to know about the key values used in the database to identify them. This additional information is called "shadow information".
- Different types are used in each schema. The **`subTotalBeforeTax`** attribute of **`Order`** is of the type **`Currency`** whereas the **`SubTotalBeforeTax`** column of the **`Order`** table is a float. When you implement this mapping you will need to be able to convert back and forth between these two representations without loss of information.

Shadow Information

- Shadow information is any data that objects need to maintain, above and beyond their normal domain data, to persist themselves.
- It typically includes:
  - *primary key* information, particularly when the primary key is a surrogate key that has no business meaning,
  - *concurrency control* markings such as timestamps or incremental counters,
  - *versioning* numbers.
- The `Order` table has an `OrderID` column used as a primary key and a `LastUpdate` column that is used for optimistic concurrency control that the `Order` class does not have.

Metadata Mapping

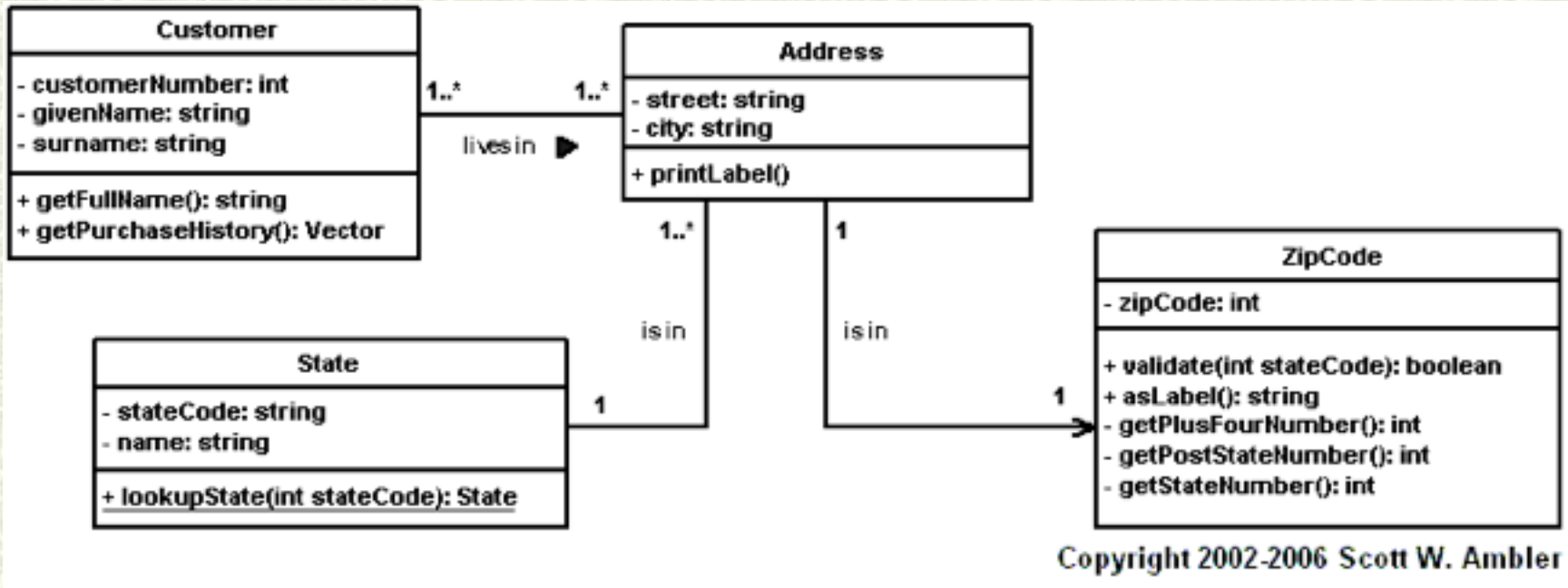
- Metadata is information about data. Metadata mapping describes the way the meta data representing the properties are mapped to the meta data corresponding to tables.

Property	**Column**
Order.orderID	Order.OrderID
Order.dateOrdered	Order.DateOrdered
Order.dateFulfilled	Order.DateFulfilled
Order.getTotalTax()	Order.Tax
Order.subtotalBeforeTax	Order.SubtotalBeforeTax
Order.shipTo.personID	Order.ShipToContactID
Order.billTo.personID	Order.BillToContactID
Order.lastUpdate	Order.LastUpdate
OrderItem.ordered	OrderItem.OrderID
Order.orderItems.position(orderItem)	OrderItem.ItemSequence
OrderItem.item.number	OrderItem.ItemNo
OrderItem.numberOrdered	OrderItem.NumberOrdered

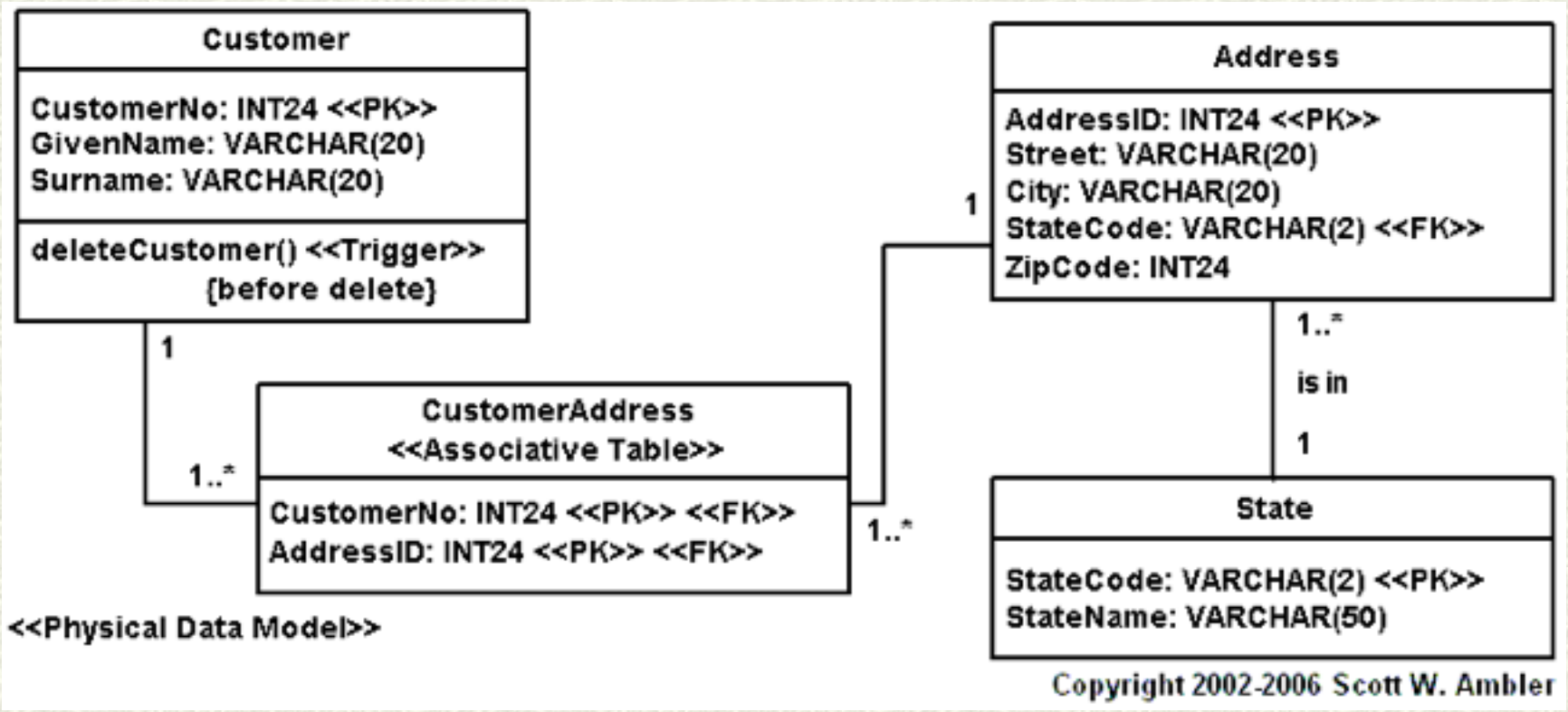
ORM Impedance Mismatch

- Object-oriented technology supports the building of applications out of objects that have both data and behavior.
- Relational technologies support the storage of data in tables and manipulation of that data via data manipulation language (DML) internally within the database via stored procedures and externally via SQL calls.
- The differences between the two approaches were labeled the *object-relational impedance mismatch* or simply *impedance mismatch*.
- Eg. with the object paradigm you traverse objects via their relationships whereas with the relational paradigm you join the data rows of tables.
- The easiest similarity/difference to observe is the different types in object languages and in relational databases.
  - Java has a string and an int - Oracle has a varchar and a smallint.
  - Java has collections - Oracle has tables
  - Java has objects - Oracle has blobs

ORM Impedance Mismatch



ORM Impedance Mismatch



Strategies for Overcoming the Object-Relational Impedance Mismatch

- Mapping inheritance structures
- Mapping object relations
- Mapping class-scope properties

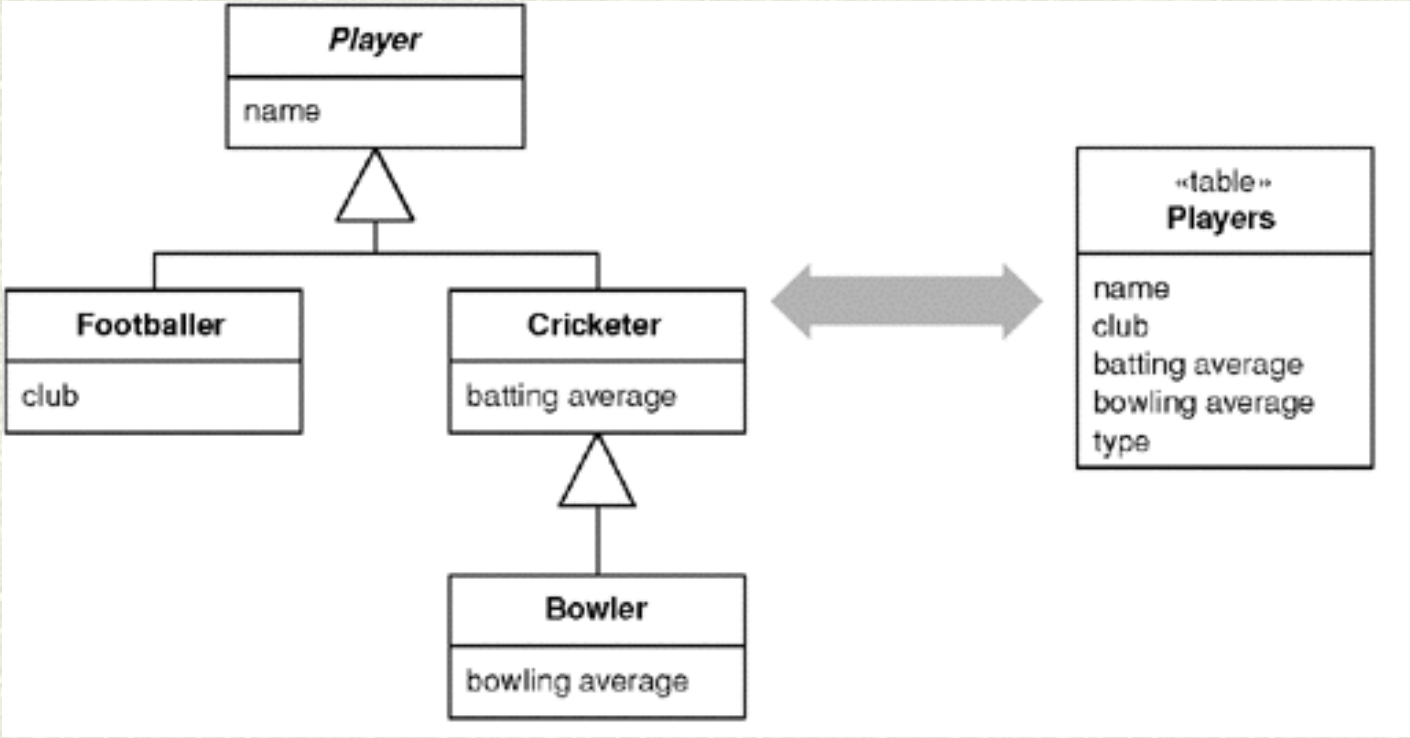
Mapping Inheritance Structures

- Relational databases do not support inheritance.
- The developer has to map the inheritance structures within the domain model to the relational database.
- Techniques:
  - Map the entire class hierarchy to a single table
  - Map each concrete class to its own table
  - Map each class to its own table
  - Map the classes into a generic table structure

Single Table Inheritance

- Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.
- Each class stores the data that is relevant to it in one table row. Any columns in the database that are not relevant are left empty.
- When loading an object into memory you need to know which class to instantiate.
- For this you have a field in the table that indicates which class should be used. This can be the name of the class or a code field.
  - A code field needs to be interpreted by some code to map it to the relevant class. This code needs to be extended when a class is added to the hierarchy.
  - If you embed the class name in the table you can just use it directly to instantiate an instance.

Single Table Inheritance



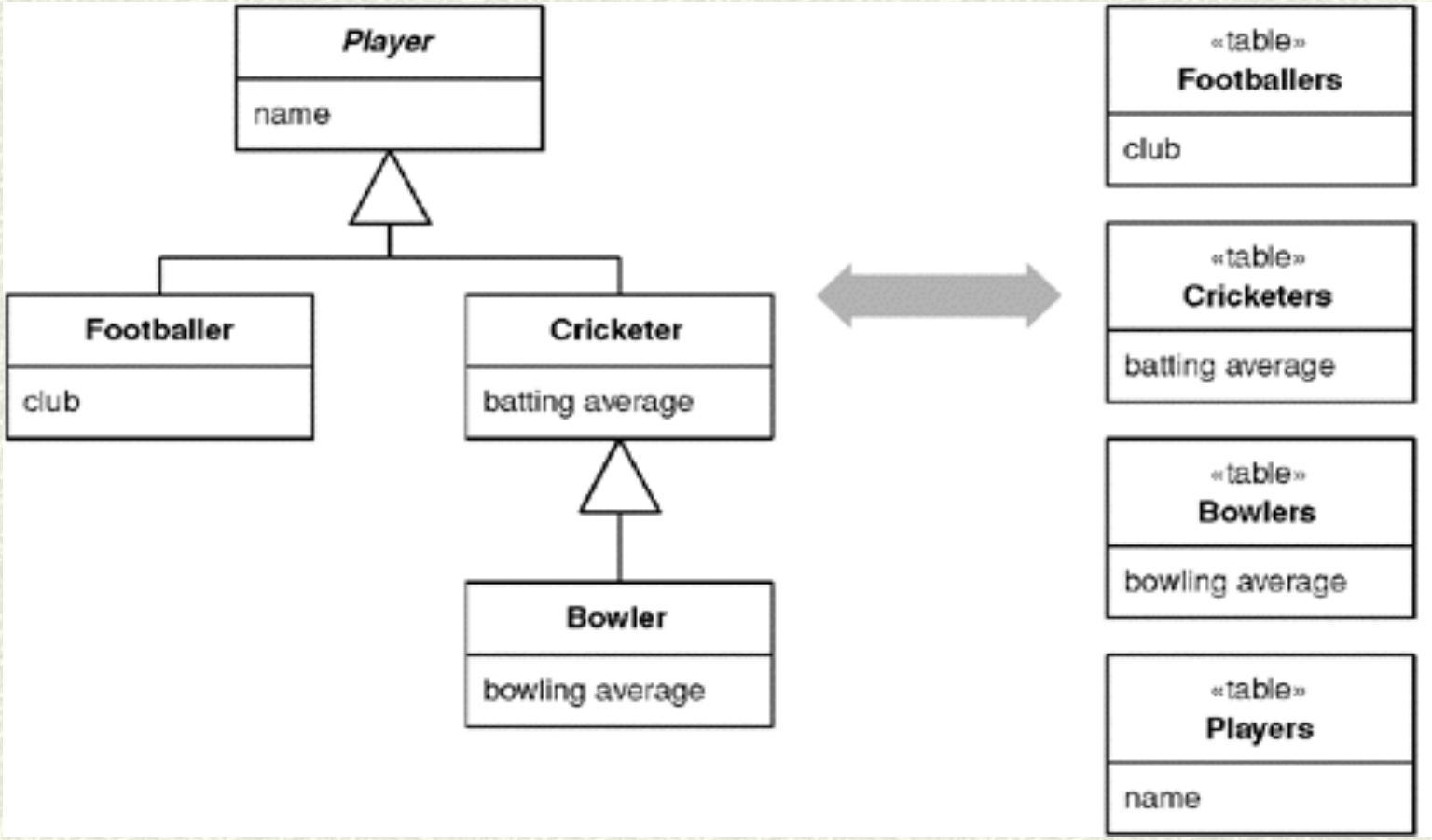
Single Table Inheritance

- Advantages:
  - There is only a single table to worry about on the database.
  - There are no joins in retrieving data.
  - Any refactoring that pushes fields up or down the hierarchy doesn't require you to change the database.
- Disadvantages:
  - Fields are sometimes relevant and sometimes not, which can be confusing to people using the tables directly.
  - Columns used only by some subclasses lead to wasted space in the database.
  - The single table may end up being too large, with many indexes and frequent locking, which may hurt performance.
  - There is a single namespace for fields, so you have to be sure that you do not use the same name for different fields. Compound names with the name of the class as a prefix or suffix help here.

Class Table Inheritance

- Represents an inheritance hierarchy of classes with one table for each class.
- The fields in the domain class map directly to fields in the corresponding tables.
- One issue is how to link the corresponding rows of the database tables.
  - A possible solution is to use a common primary key value in both the base class table and the derived table. Since the superclass table has a row for each row in the other tables, the primary keys are going to be unique across the tables.
  - An alternative is to let each table have its own primary keys and use foreign keys into the superclass table to tie the rows together.
- The biggest implementation issue is how to bring the data back from multiple tables in an efficient manner:
  - Joins across the various component tables,
  - Joins for more than three or four tables tend to be slow because of the way databases do their optimizations.
- Queries are also difficult.

Class Table Inheritance



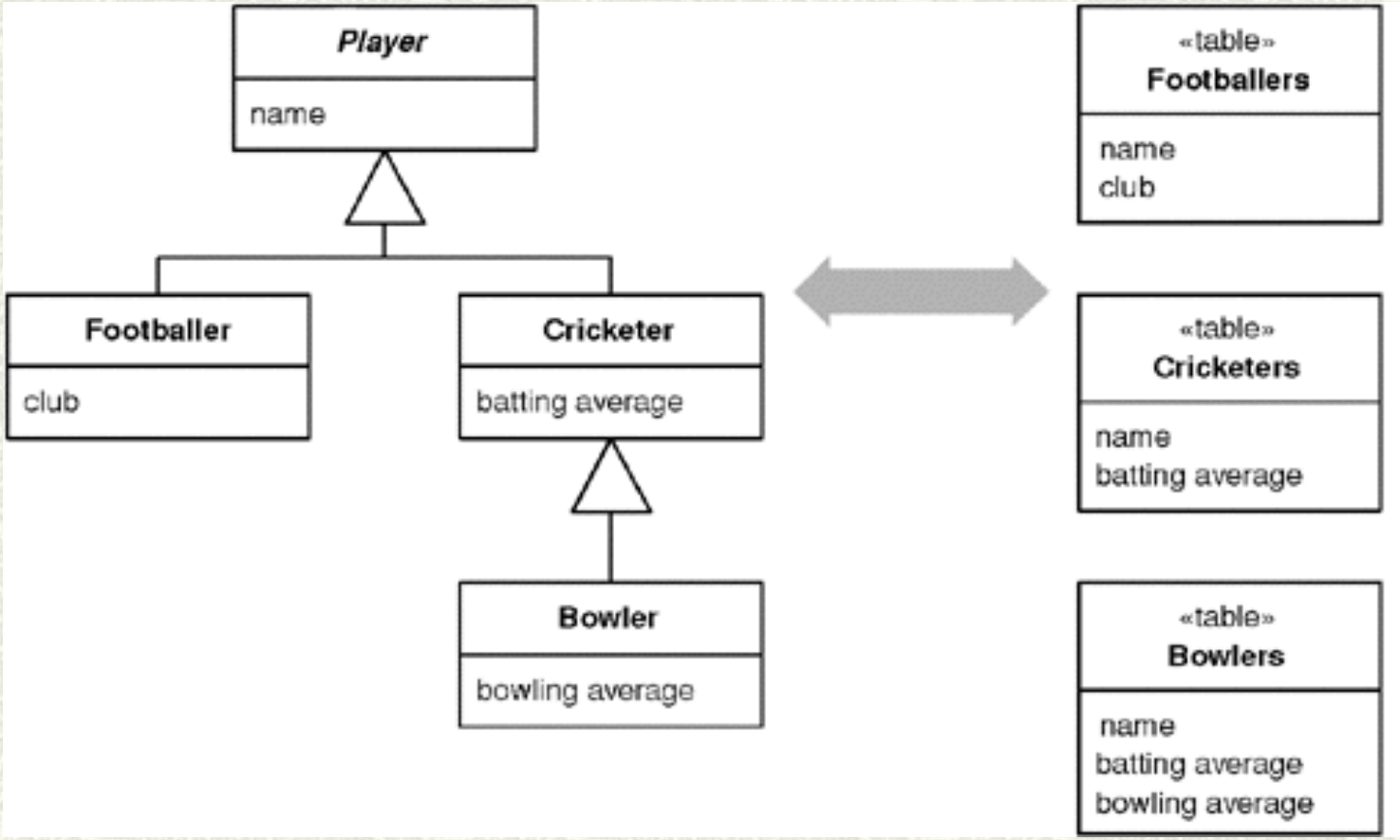
Class Table Inheritance

- Advantages:
  - All columns are relevant for every row so tables are easier to understand and do not waste space.
  - The relationship between the domain model and the database is very straightforward.
- Disadvantages:
  - You need to use multiple tables to load an object, which means a join or multiple queries and memory usage.
  - Any refactoring of fields up or down the hierarchy causes database changes.
  - The supertype tables may become a bottleneck because they have to be accessed frequently.
  - The high normalization may make it hard to understand for ad hoc queries.

Concrete Table Inheritance

- Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.
- Each table contains columns for the concrete class and all its ancestors, so any fields in a superclass are duplicated across the tables of the subclasses.
- The developer has to ensure that keys are unique not just to a table but to all the tables from a hierarchy.

Concrete Table Inheritance

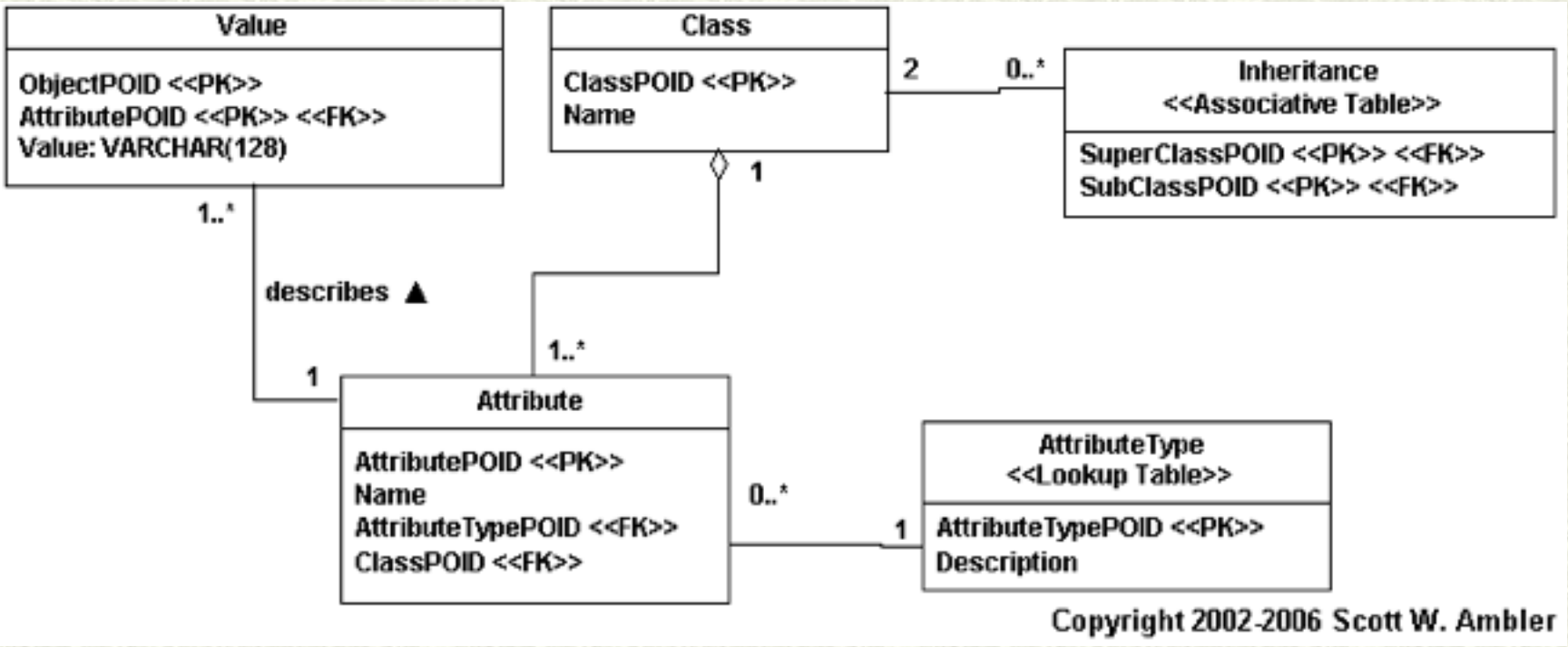


Concrete Table Inheritance

- Advantages:
  - Each table is self-contained and has no irrelevant fields. As a result it makes good sense when used by other applications that aren't using the objects.
  - There are no joins to do when reading the data.
  - Each table is accessed only when that class is accessed, which can spread the access load.
- Disadvantages:
  - Primary keys can be difficult to handle.
  - Database relationships to abstract classes cannot be enforced.
  - If the fields on the domain classes are pushed up or down the hierarchy, you have to alter the table definitions.
  - If a superclass field changes, you need to change each table that has this field because the superclass fields are duplicated across the tables.
  - A find on the superclass forces you to check all the tables, which leads to multiple database accesses (or a weird join).

Generic Table

- This approach is not specific to inheritance structures, it supports all forms of mapping.
- It constructs a generic metadata database for each attribute in the class hierarchy.



Generic Table

- Advantages:
  - It can be extended to provide meta data to support a wide range of mappings, including relationship mappings.
  - It is flexible, enabling you to quickly change the way that you store objects because you merely need to update the meta data stored in the *Class*, *Inheritance*, *Attribute*, and *AttributeType* tables accordingly.
- Disadvantages:
  - It only works for small amounts of data because you need to access many database rows to build a single object.
  - Querying against this data can be very difficult due to the need to access several rows to obtain the data for a single object.

Mapping object relationships

- The central issue here is the different way in which objects and relations handle links, which leads to two problems.
- First, there is a difference in representation. Objects handle links by storing references that are held by the runtime of either memory-managed environments or memory addresses. Relational databases handle links by forming a key into another table.
- Second, objects can easily use collections to handle multiple references from a single field, while normalization forces all relation links to be single valued. This leads to reversals of the data structure between objects and tables.
- An *order* object naturally has a collection of *line item* objects that do not need any reference back to the order. However, the table structure is the other way around—the line item must include a foreign key reference to the order since the order cannot have a multivalued field.

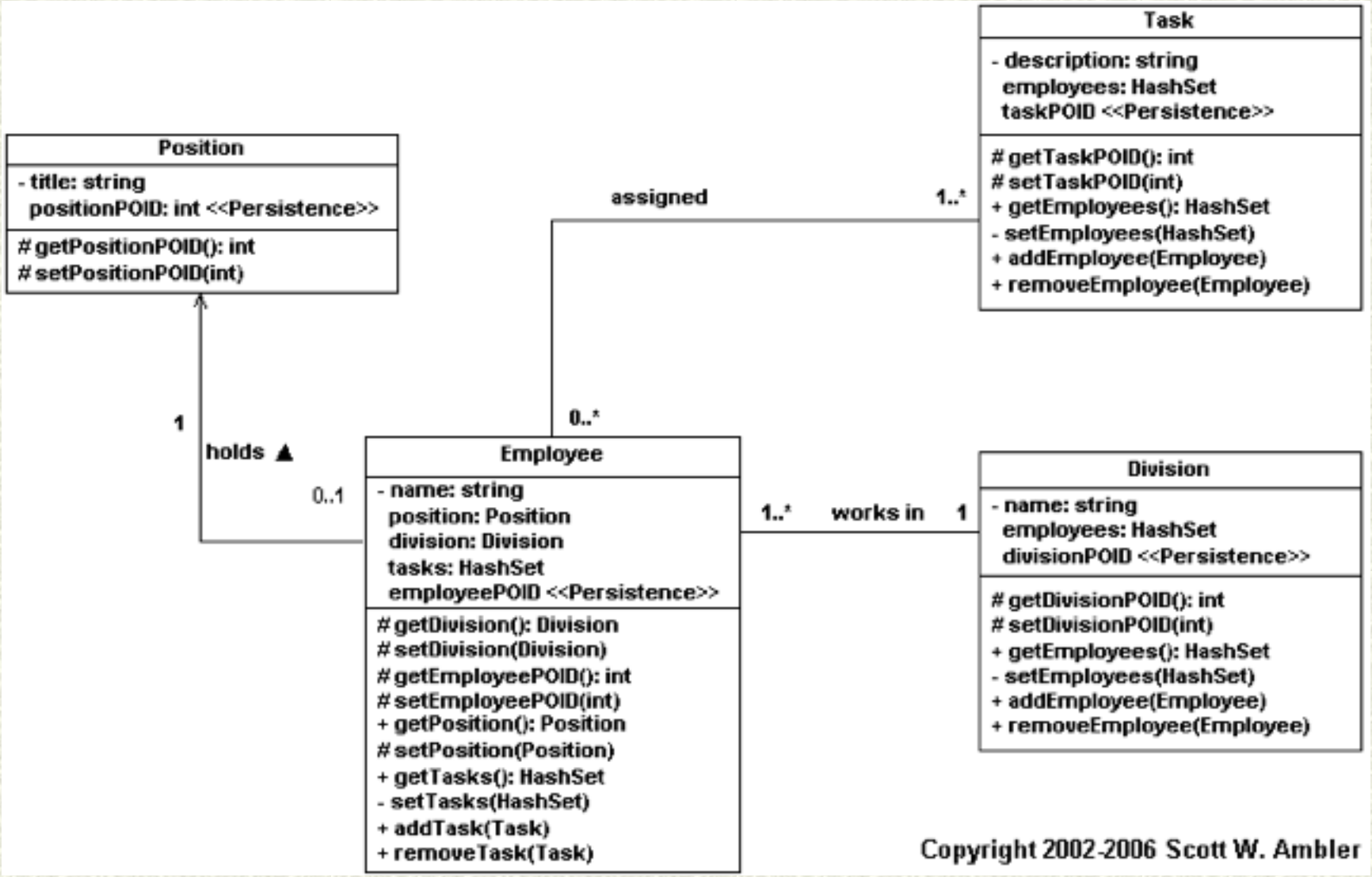
Types of relationships

- There are two categories of object relationships that you need to be concerned with when mapping:

- ***Multiplicity***. It includes three types:
  - *One-to-one* relationships. This is a relationship where the maximums of each of its multiplicities is one.
  - *One-to-many* relationships. Also known as a many-to-one relationship, this occurs when the maximum of one multiplicity is one and the other is greater than one.
  - *Many-to-many* relationships. This is a relationship where the maximum of both multiplicities is greater than one.

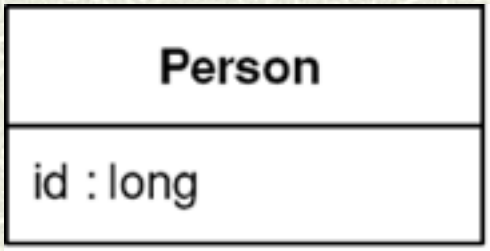
- ***Directionality***. It contains two types:
  - *Uni-directional* relationships. A uni-directional relationship exists when an object knows about the object(s) it is related to but the other object(s) do not know of the original object.
  - *Bi-directional* relationships. A bi-directional relationship exists when the objects on both end of the relationship know of each other.

Types of relationships



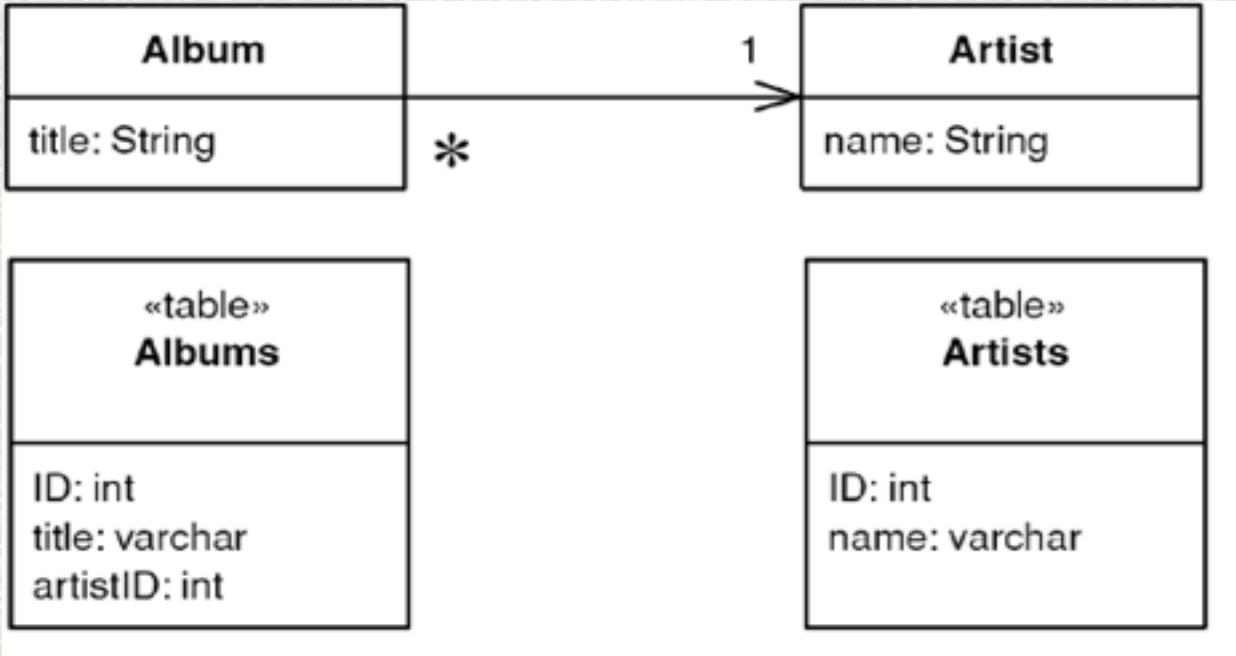
Identity Field

- Saves a database ID field in an object to maintain identity between an in-memory object and a database row.
- The primary key of the relational database table is stored in the object's fields.
- Identity Field should be used when there is a mapping between objects in memory and rows in a database.



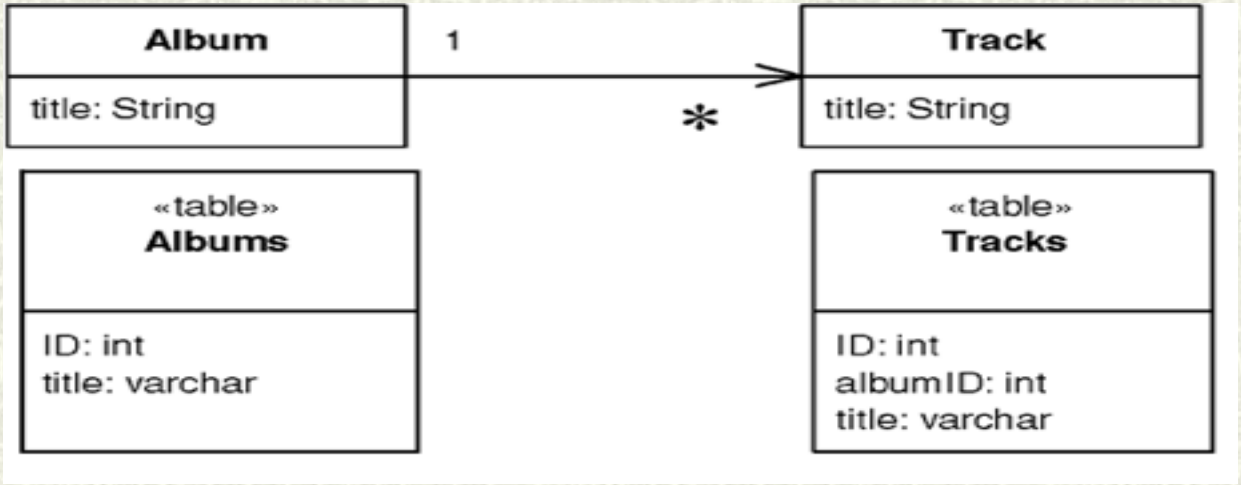
Foreign Key Mapping

- Maps an association between objects to a foreign key reference between tables.
- Each object contains the database key from the appropriate database table.
- If two objects are linked together with an association, this association can be replaced by a foreign key in the database.



Foreign Key Mapping

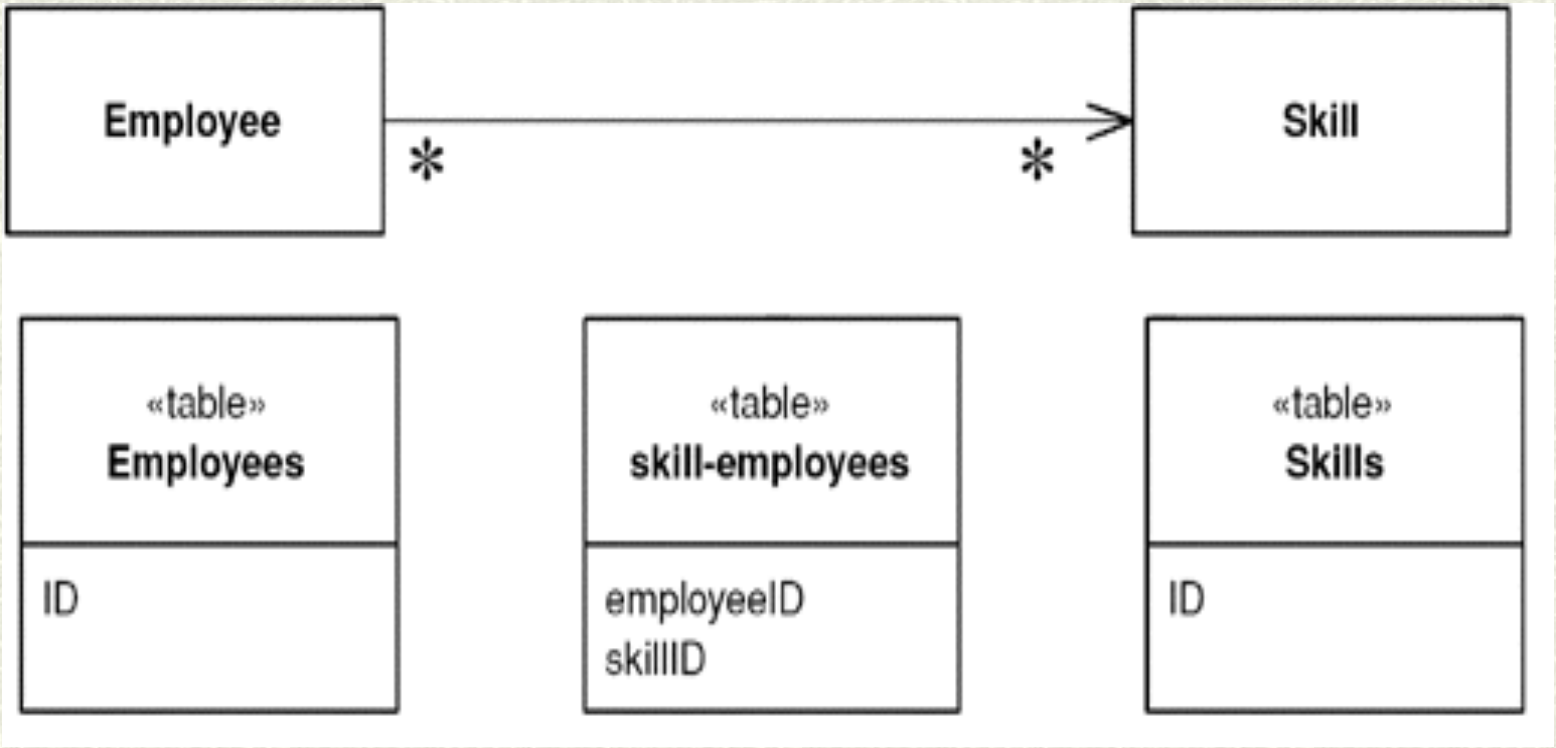
- A more complicated case is when we have a collection of objects.
- We cannot save a collection in the database, so we have to reverse the direction of the reference.
- A Foreign Key Mapping can be used for almost all associations between classes.
- It is not possible with many-to-many associations.



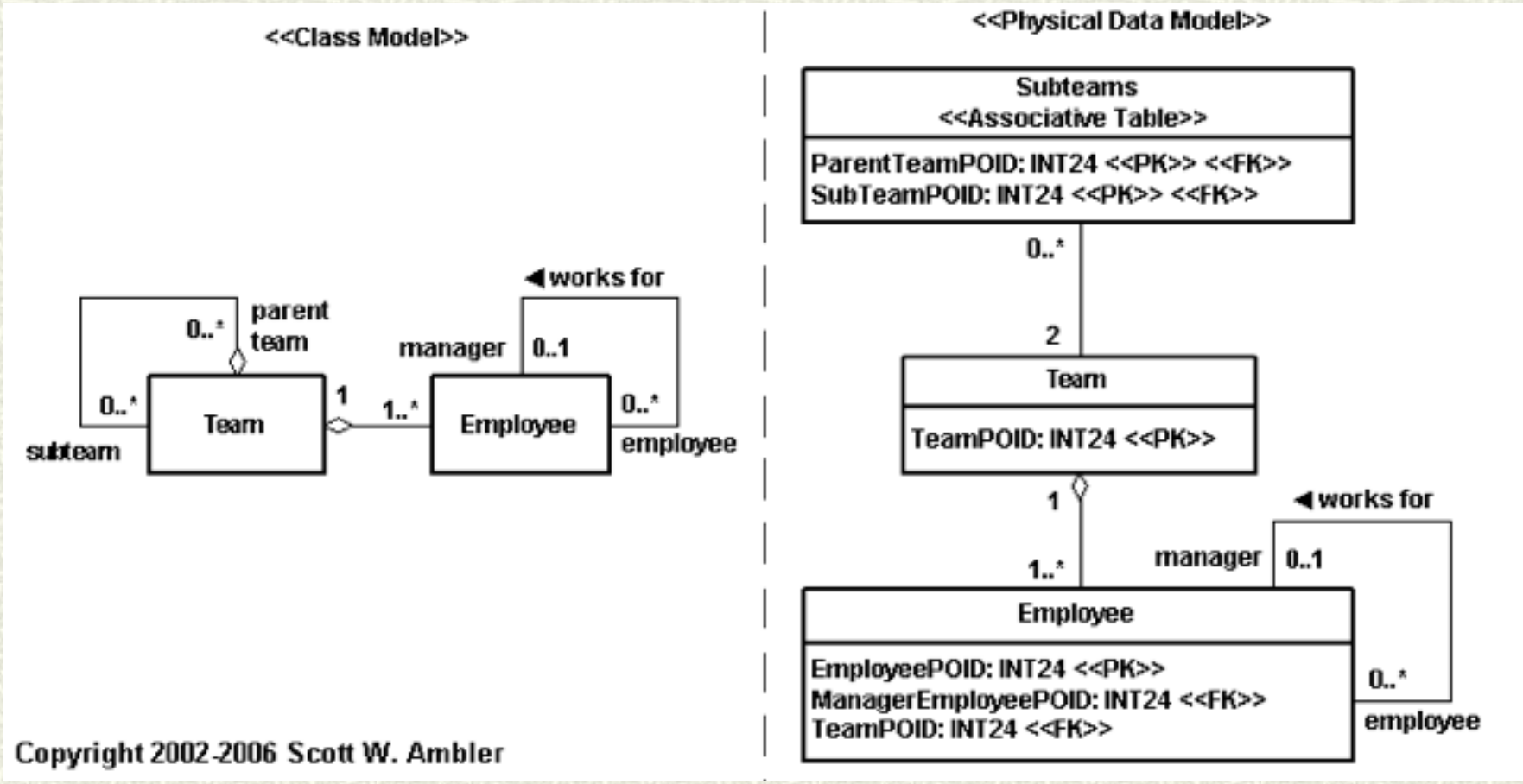
Association Table Mapping

- Saves an association as a table with foreign keys to the tables that are linked by the association.
- Objects can handle multivalued fields easily by using collections as field values. Relational databases do not have this feature and are constrained to single-valued fields only.
- The basic idea is to use a link table to store an association. This table has only the foreign key IDs for the two tables that are linked together, it has one row for each pair of associated objects.
- The link table has no corresponding in-memory object. As a result it has no ID. Its primary key is the compound of the two primary keys of the tables that are associated.
- The Association Table Mapping is most often used to map a many-to-many association. However, it can be used for other types of associations, but it is more complex and difficult.

Association Table Mapping



Mapping recursive associations



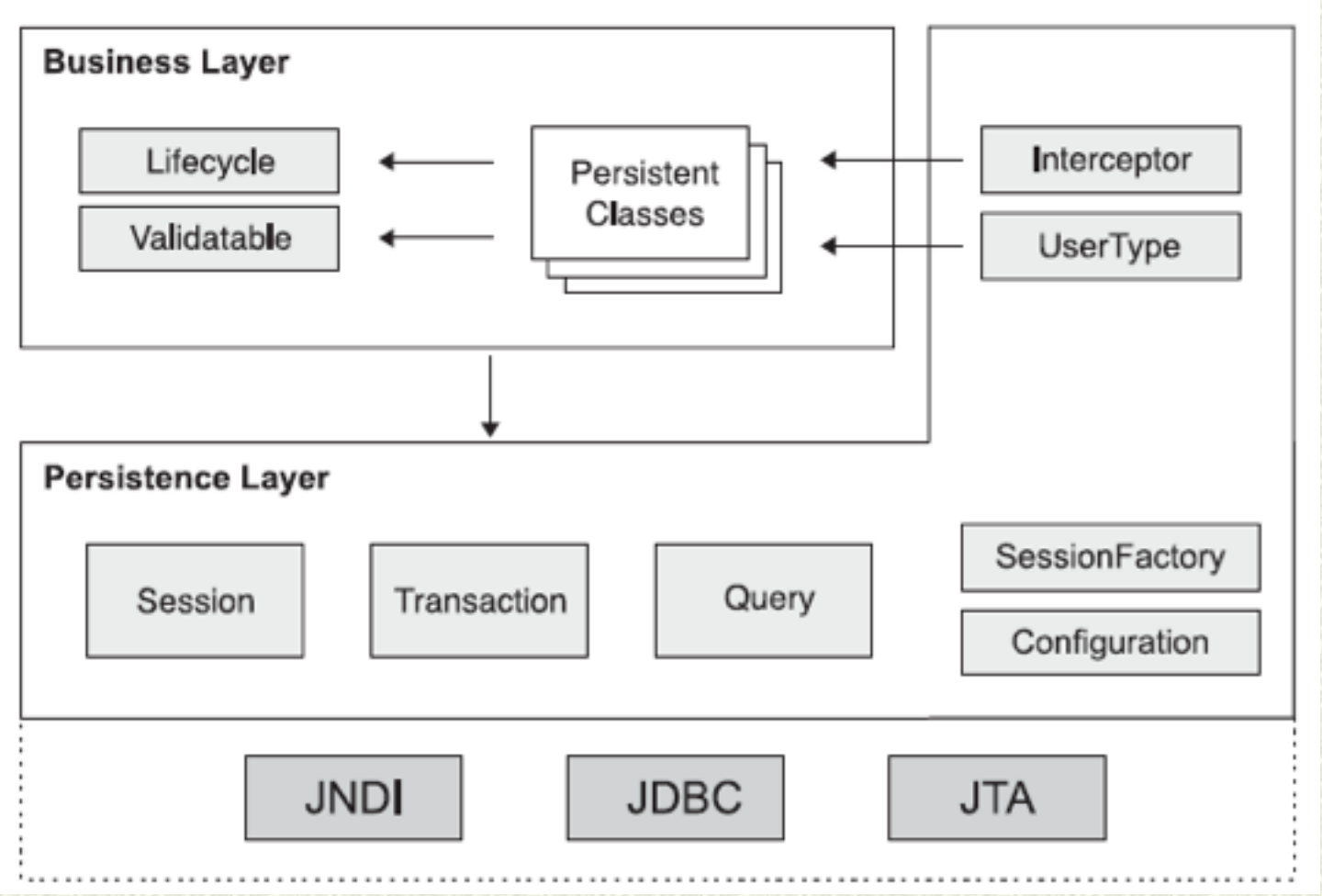
Mapping class scope properties

- Strategies:
  - Single Column, Single-Row Table
    - Pros:Simple, Fast access
    - Cons: Could result in many small tables
  - Multi-Column, Single-Row Table for a Single Class
    - Pros:Simple, Fast access
    - Cons:Could result in many small tables, although fewer than the single column approach
  - Multi-Column, Single-Row Table for all Classes
    - Pros: Minimal number of tables introduced.
    - Cons: Potential for concurrency problems if many classes need to access the data at once.
  - Multi-Row Generic Schema for all Classes
    - Pros: Minimal number of tables introduced. Reduces concurrency problems.
    - Cons: Need to convert between types. The data schema is coupled to the names of your classes and their class scope properties.

Hibernate

- Open source Object Relational Mapping tool.
- Hibernate applications define persistent classes that are “mapped” to database tables.
- All SQL statements are automatically generated at runtime.
- The information about how a class should be persisted is given as an XML mapping document.
- The mapping document defines, among other things, how the properties of a class map to columns of table(s).
- The strategies used for object relational mapping are chosen by the developer in the XML mapping document.

Architecture



Architecture

- The Hibernate interfaces may be classified as follows:
  - Interfaces called by applications to perform basic CRUD and querying operations. These interfaces are the main point of dependency of application business/control logic on Hibernate. They include Session, Transaction, and Query.
  - Interfaces called by application infrastructure code to configure Hibernate, most importantly the Configuration class.
  - Callback interfaces that allow the application to react to events occurring inside Hibernate, such as Interceptor, Lifecycle, and Validatable.
  - Interfaces that allow extension of Hibernate, such as UserType, CompositeUserType, and IdentifierGenerator.These interfaces are implemented by application infrastructure code (if necessary).
- Hibernate uses existing Java APIs, including JDBC, Java Transaction API (JTA), and Java Naming and Directory Interface (JNDI).

Core Interfaces

- *Session* interface: is the primary interface used by Hibernate applications. An instance of Session is lightweight and is inexpensive to create and destroy.
- *SessionFactory* interface. The application obtains Session instances from a SessionFactory. It caches generated SQL statements and other mapping metadata that Hibernate uses at runtime.
- *Configuration* interface. The Configuration object is used to configure and start Hibernate. The application uses a Configuration instance to specify the location of mapping documents and Hibernate-specific properties and then create the SessionFactory.
- *Transaction* interface. A Transaction abstracts application code from the underlying transaction implementation (a JDBC transaction, a JTA UserTransaction, etc.).
- *Query* and *Criteria* interfaces. The Query interface allows performing queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of the database. The Criteria interface is very similar; it allows you to create and execute object oriented criteria queries.

Example

- Steps to follows:
  - Create POJO Classes
  - Create Mapping Files
  - Hibernate Configuration
  - Make the DAO layer with the required functions
  - Test the DAO classes

Example - Create POJO Classes

```
package hello;
public class Message {
    private Long id;
    private String text;
    private Message nextMessage;
    public Message() {}
    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;  }
    public Message getNextMessage() {  return nextMessage; }
    public void setNextMessage(Message nextMessage) {
        this.nextMessage = nextMessage;
    }
}
```

Example - Create Mapping Files

```
<!—hello/Message.hbm.xml -->
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="hello.Message"
            table="MESSAGES">
        <id name="id‟ column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property
                name="text"
                column="MESSAGE_TEXT"/>
        <many-to-one
                name="nextMessage"
                cascade="all"
                column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Example - Hibernate Configuration

```
<!--hibernate.cfg.xml -->
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.url"> jdbc:mysql://localhost/agenti</property>
        <property name="connection.driver_class">com.mysql.jdbc.Driver </
   property>
        <property name="connection.username">grigo</property>
        <property name="connection.password">grigo</property>
        <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
   <!-- mapping files -->
        <mapping resource="hello/Message.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Example – DAO Layer

```
//INSERT
void addMessage(){
     Session session = getSessionFactory().openSession();
     Transaction tx=null;
     try{
        tx = session.beginTransaction();
        Message message = new Message("Hello World2");
        session.save(message);
        tx.commit();
     }catch(RuntimeException ex){
        if (tx!=null)
               tx.rollBack();
     }finally{
        session.close();
     }
 }
```

Example – DAO Layer

```
    //UPDATE
void updateMessage(){
   Session session = getSessionFactory().openSession();
    Transaction tx=null;
    try{
       tx = session.beginTransaction();
        Message message =
              (Message) session.load( Message.class, new Long(1) );
        message.setText("New Text 2");
        Message nextMessage = new Message("Next message 2");
        message.setNextMessage( nextMessage );
        tx.commit();
    } catch(RuntimeException ex){
        if (tx!=null)
              tx.rollBack();
     }finally{
           session.close();
     }
 }
```

Example – DAO Layer

```
//DELETE
void deleteMessage(){
   Session session = getSessionFactory().openSession();
   Transaction tx=null;
     try{
        tx = session.beginTransaction();
      Message crit=(Message)session.createCriteria(Message.class)
                  .add( Restrictions.like("text", "Ne%"))
                   .setMaxResults(1)
                   .uniqueResult();
       session.delete(crit);
        tx.commit();
     } catch(RuntimeException ex){
        if (tx!=null)
              tx.rollBack();
     }finally{
        session.close();
     }
 }
```

Example – DAO Layer

```
//SELECT
 void getMessages(){
    Session session = getSessionFactory().openSession();
    Transaction tx=null;
     try{
        tx = session.beginTransaction();
    List messages =
       session.createQuery("from Message as m order by m.text asc").list();
        System.out.println( messages.size() + " message(s) found:" );
        for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
            Message message = (Message) iter.next();
            System.out.println( message.getText() );
        }
        tx.commit();
     }catch(RuntimeException ex){
        if (tx!=null)
                tx.rollBack();
     }finally{
         session.close();
     }
 }
```

Example – Test DAO Layer

```
class Test{
   static SessionFactory sessions;
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
       cfg.configure("hibernate.cfg.xml");
       sessions = cfg.buildSessionFactory();
       Test test=new Test();
        test.addMessage();
        test.getMessages();
        test.updateMessage();
        test.getMessages();
    }

    static SessionFactory getSessionFactory(){
        return sessions;
    }
}
```

Executing queries

- Three possibilities:
  - HQL

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

  - Query by Criteria

```
Criteria crit=session.createCriteria(Category.class)
.add( Expression.like("name", "Laptop%") );
```

  - SQL

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

Listing the results

- The `list()` method executes the query and returns the results as a list:

```
List result = session.createQuery("from User").list();
```

- Unique instance:

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

Bid bid = (Bid) session.createCriteria(Bid.class)
.add( Expression.eq("id", id) )
.uniqueResult();
```

Queries with parameters

- Named parameters

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

- Positional parameters:

```
String queryString = "from Item item where
  item.description like ? and item.date > ?";
List result = session.createQuery(queryString)
.setString(0, searchString)
.setDate(1, minDate)
.list();
```

Hibernate Query Language (HQL)

It supports almost all SQL functions and operators:

- 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.

Criteria Queries

Object-oriented like criteria:

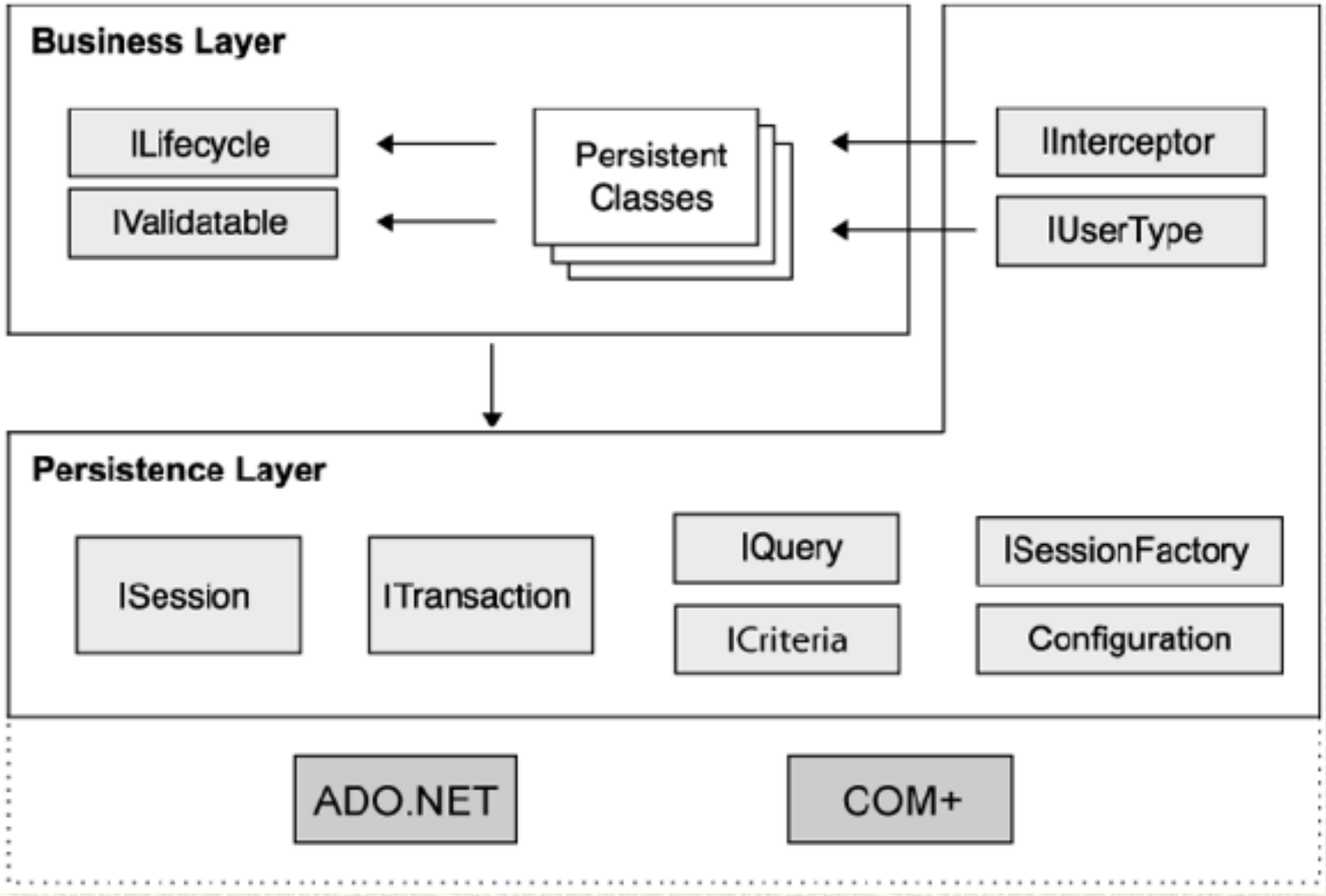
```
User user = (User)
session.createCriteria(User.class).add( Expression.eq("email",
"foo@hibernate.org") ).uniqueResult();

session.createCriteria(User.class).add( Expression.like("firstname",
"G%") ).list();

List results =
session.createCriteria(User.class).addOrder( Order.asc("lastname") )
.addOrder( Order.asc("firstname") ).list();

Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results =
session.createCriteria(Cat.class).add( Example.create(cat) ).list();
```

NHibernate


Business Layer
ILifecycle
IValidatable
Persistent Classes
IInterceptor
IUserType
Persistence Layer
ISession
ITransaction
IQuery
ICriteria
ISessionFactory
Configuration
ADO.NET
COM+