

MACIASZEK, L.A. (2005):
Requirements Analysis and System Design, 2nd ed.
Addison Wesley, Harlow England, 504p.
ISBN 0 321 20464 6

Chapter 8
Persistence and Database Design

© Pearson Education Limited 2005

Topics

- *Business objects and persistence*
- *Relational database model*
- *Object-relational mapping*
- *Patterns for managing persistent objects*
- *Implementing database access*
- *Designing business transactions*

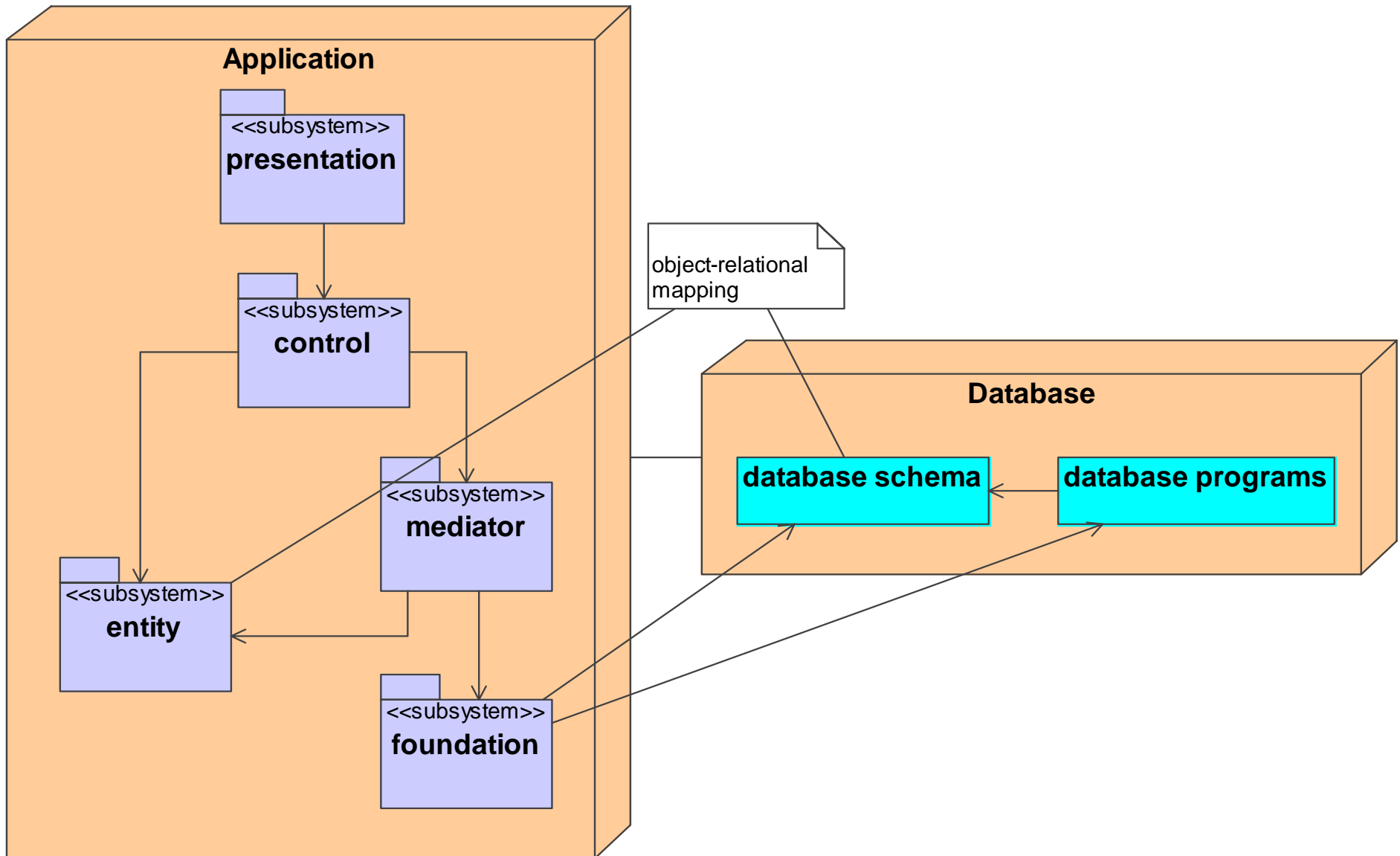
About persistence and databases

- **Database management systems (DBMSs)** provide the technology to support concurrent access by large number of users and application programs to the same data store
- Class diagrams define the data structures required by an application
 - data structures that have **persistent** presence in the database are modeled as the entity classes (“business objects”)
 - **entity classes** correspond to the “E” letter in the PCMEF framework
- Entity classes need to be mapped to data structures in the database
- Data structures in the DB conform to a **database model**
 - object-oriented
 - object-relational
 - relational
- **Relational model** dominates in business information systems

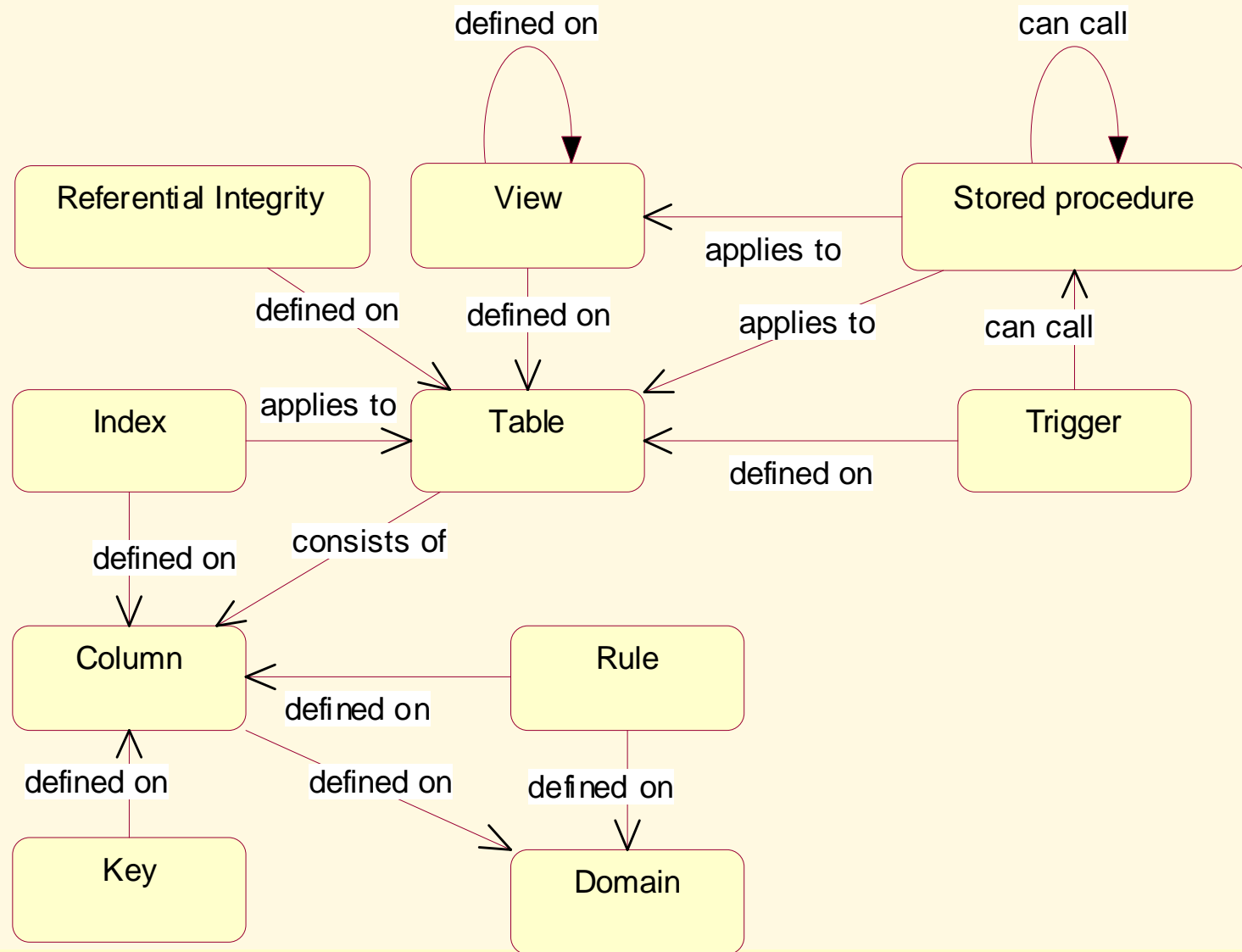
Levels of data models

- **Data model (database schema)** is an abstraction that presents the database structures in more understandable terms than as raw bits and bytes
 - **External (conceptual) data model**
 - as required by a single application
 - **entity-relationship (ER) diagrams**
 - **Logical data model (global conceptual schema)**
 - reflects logical storage structures (tables, etc.) of the database model to be used for the system implementation
 - a global integrated model to support any current and expected applications that need accessing information stored in the database
 - **Physical data model**
 - specific to a particular DBMS (such as Oracle10g).
 - defines how data is actually stored on persistent storage devices, typically disks (defines indexes, clustering of data, etc.)

Integrating application and database modeling



Relational database model



Columns, domains and rules

- **Columns** have atomic domains (data types)
- **Domain** defines the legal set of values that a column can take; it can be
 - anonymous (e.g. gender char(1)) or
 - named (e.g. gender Gender)
- Columns and domains can have **business rules** which constrain them:
 - default value (e.g. if no value is provided for city, assume 'Sydney')
 - range of values (e.g. the allowed age is in the range 18 to 80)
 - list of values (e.g. the allowed color is 'green', 'yellow' or 'red')
 - case of value (e.g. the value must be in uppercase or lowercase)
 - format of value (e.g. the value must start with letter 'K')

Tables

■ Relational table

- *fixed set of columns*
- *any number of rows (records)*
 - *no duplicate rows in a table → primary key*
- *foreign keys to link to other tables → referential integrity*
- *NULL values allowed*
- *atomic and non-repeating types only*

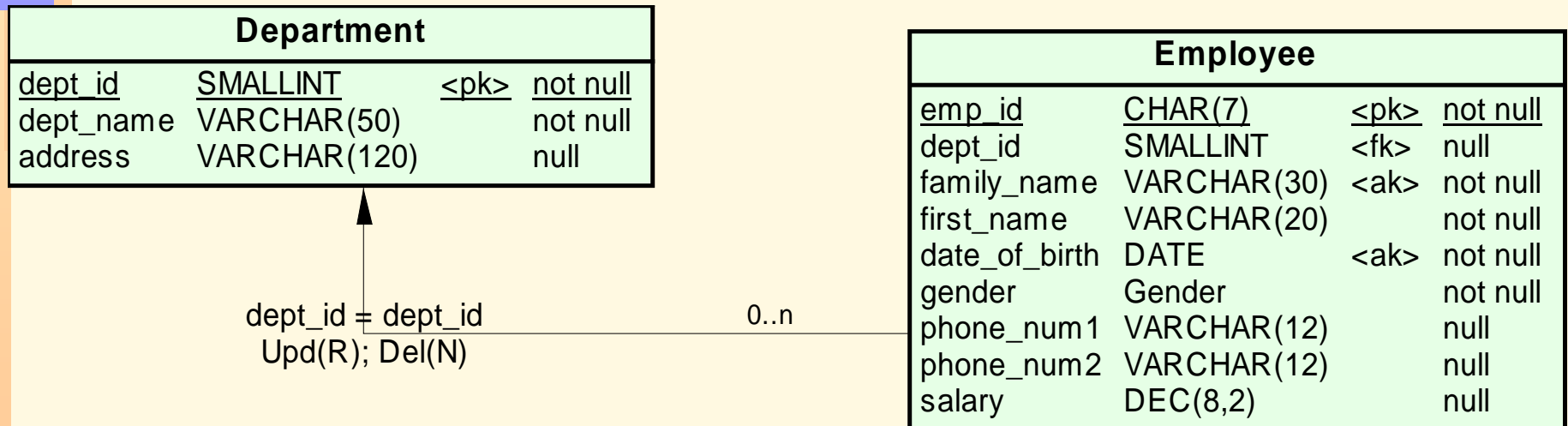
Employee			
<u>emp_id</u>	<u>CHAR(7)</u>	<u><pk></u>	<u>not null</u>
dept_id	SMALLINT	<fk>	null
family_name	VARCHAR(30)	<ak>	not null
first_name	VARCHAR(20)		not null
date_of_birth	DATE	<ak>	not null
gender	Gender		not null
phone_num1	VARCHAR(12)		null
phone_num2	VARCHAR(12)		null
salary	DEC(8,2)		null

SQL for table definition

```
-----  
-- Domain: "Gender"  
-----  
create distinct type "Gender" as CHAR(1) with comparisons;  
-----  
-- Table: "Employee"  
-----  
create table "Employee" (  
    "emp_id"          CHAR(7)          not null,  
    "dept_id"        SMALLINT,  
    "family_name"    VARCHAR(30)       not null,  
    "first_name"     VARCHAR(20)      not null,  
    "date_of_birth"  DATE              not null,  
    "gender"         "Gender"         not null  
        constraint "C_gender" check ("gender" in ('F','M','f','m')),  
    "phone_num1"     VARCHAR(12),  
    "phone_num2"     VARCHAR(12),  
    "salary"         DEC(8,2),  
    primary key ("emp_id"),  
    unique ("date_of_birth", "family_name")  
);
```

Referential integrity

- **Referential integrity** constraints to maintain relationships between tables
 - primary-to-foreign key correspondence
- **Foreign key** – a set of columns in one table whose values are either NULL or are required to match the values of the primary key in the same or another table



Declarative referential integrity constraints

- Associated with **delete** and **update** operations
- What to do with *Employee* rows if a *Department* row is deleted or updated (i.e. when *dept_id* gets updated)?
 - $Upd(R); Del(R)$ – restrict the update or delete operation (i.e. do not allow the operation to go ahead if there are still *Employee* rows linked to that *Department*).
 - $Upd(C); Del(C)$ – cascade the operation (i.e. delete all linked *Employee* rows).
 - $Upd(N); Del(N)$ – set null (i.e. update or delete the *Department* row and set *dept_id* of the linked *Employee* rows to NULL).
 - $Upd(D); Del(D)$ – set default (i.e. update or delete the *Department* row and set *dept_id* of the linked *Employee* rows to the default value).
- **change parent allowed (cpa)** constraint could also be defined for a referential integrity
 - **cpa** states that records in a child (foreign) table can be re-assigned to a different record in a parent table

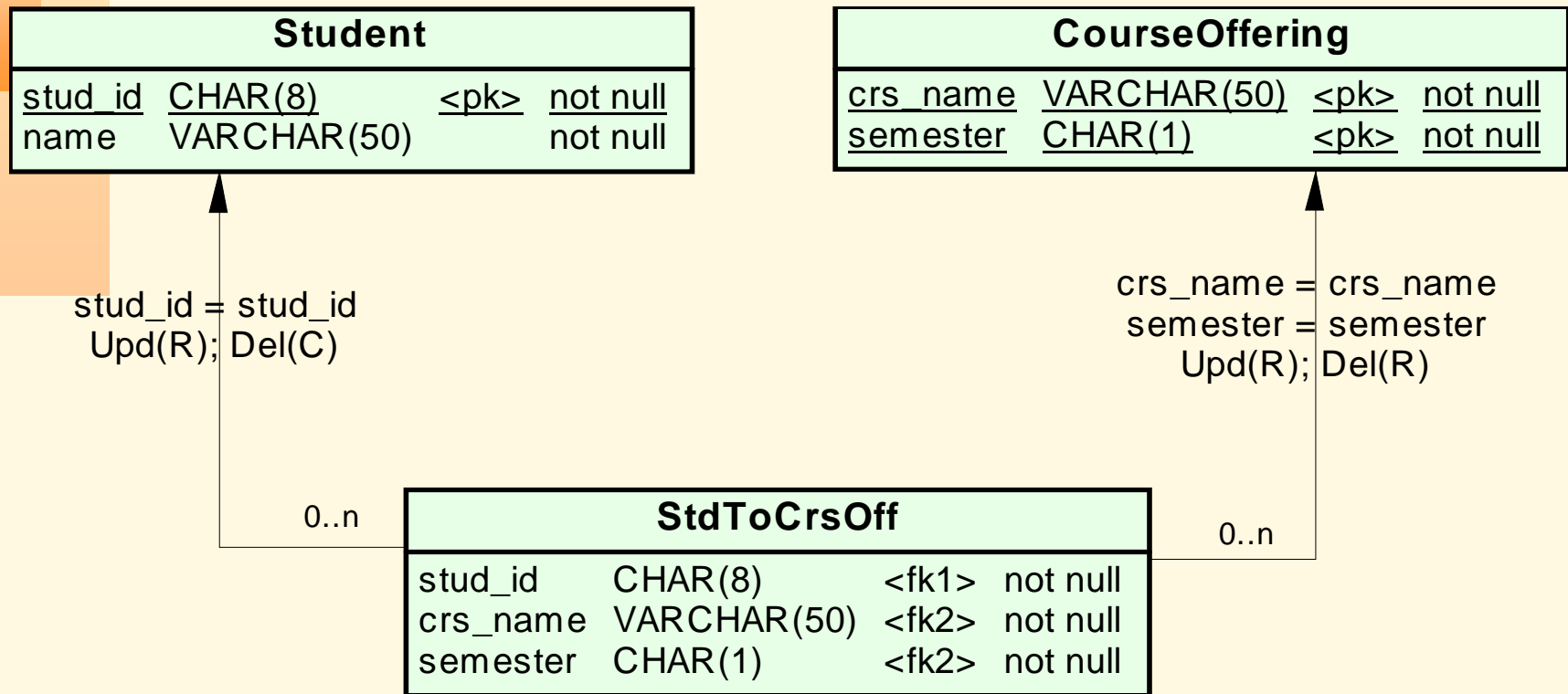
SQL for referential integrity

```
alter table "Employee"  
  drop foreign key "RefToDepartment";
```

```
alter table "Employee"  
  add foreign key "RefToDepartment" ("dept_id")  
  references "Department" ("dept_id")  
  on delete set null;
```

- *referential integrity is specified for the delete operations only*
- *in Oracle, restrict is the only declarative constraint allowed for the update operations, so restrict is implicitly assumed*

Referential integrity for many-to-many relationship

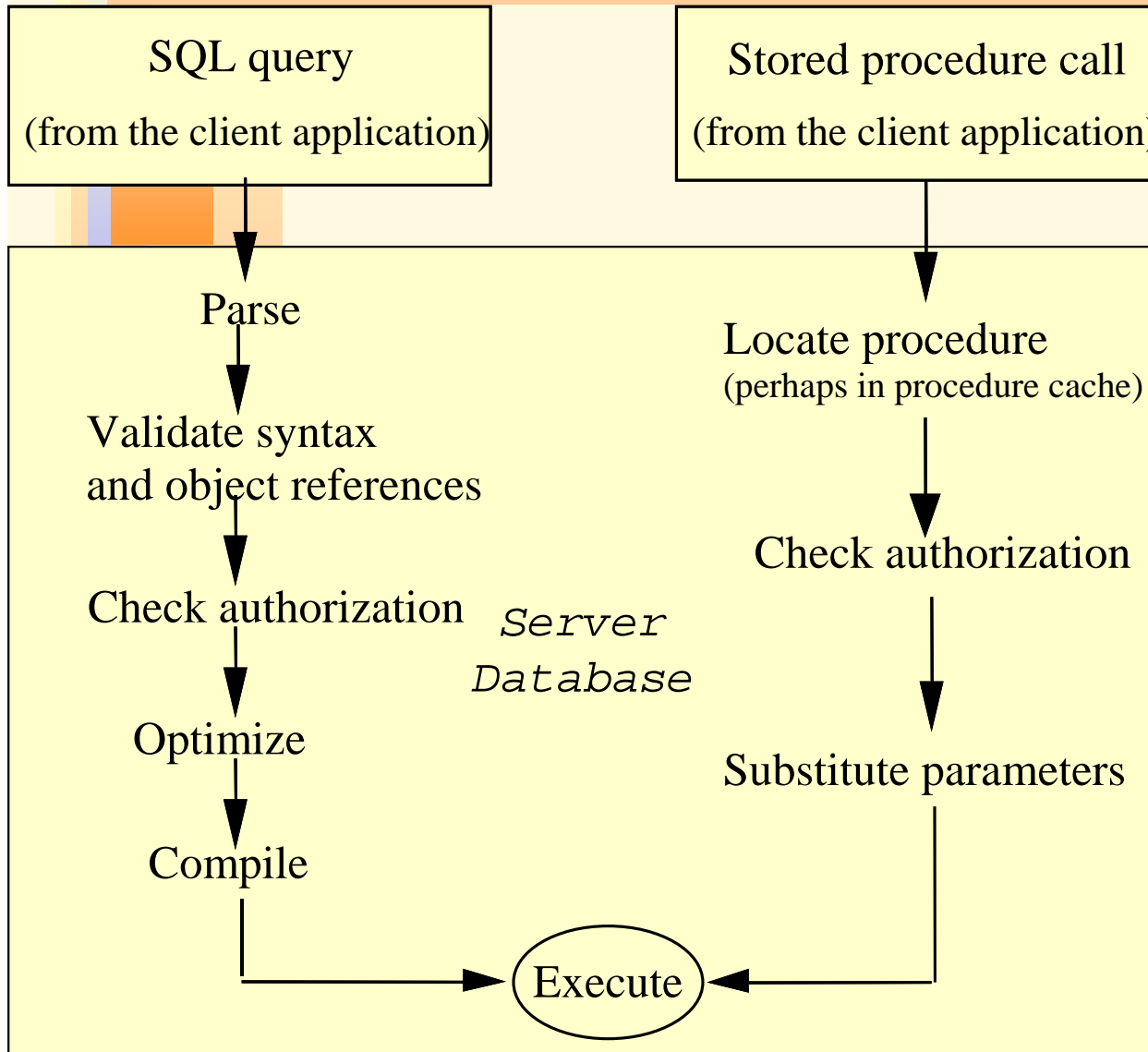


Triggers

- **DB programs**, written in an extended SQL, executed automatically (triggered) as a result of a modification operation (*insert*, *update* or *delete*) on a table on which the trigger has been defined
- To **procedurally** enforce business rules (more complex referential integrity constraints)

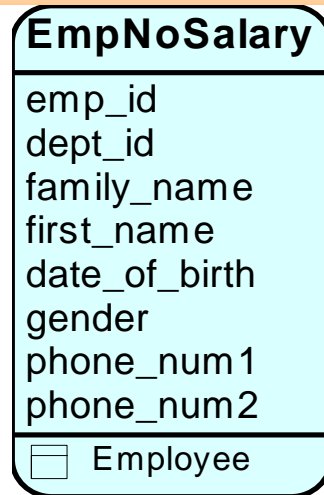
```
create trigger keepdpt
on Department for delete
as
if @@rowcount = 0
    return /* avoid firing trigger if no rows affected */
if exists
    (select * from Employee, deleted
     where Employee.dept_id = deleted.dept_id)
begin
    print 'Test for RESTRICT DELETE failed. No deletion'
    rollback transaction
    return
end
return
go
```

Stored procedures



- **DB programs**, written in an extended SQL, that can be called from a client program as required
- A **stored procedure** is given a name, can take input and output parameters, it is compiled and stored in the database
 - Triggers are a special kind of stored procedures

Views



```
--  
=====
```

```
-- View: "EmpNoSalary"  
--  
=====
```

```
=====
```

```
create view "EmpNoSalary" as  
    select Employee.emp_id, Employee.dept_id,  
           Employee.family_name, Employee.first_name,  
           Employee.date_of_birth, Employee.gender,  
           Employee.phone_num1, Employee.phone_num2  
from Employee;
```

- *stored and named **SQL query***
- *because a result of any SQL query is a **transient table**, a view can be used in place of a table in other SQL operations*
- *can be derived from one or more tables and/or one or more other views*

Normal forms for tables

■ 1st NF

- *no structured or multi-valued columns*
- *can exhibit **update anomalies** → need for higher NFs*
- *a table can be brought to a higher NF by splitting it vertically along columns into two or more smaller tables*

■ 2nd NF

■ 3rd NF

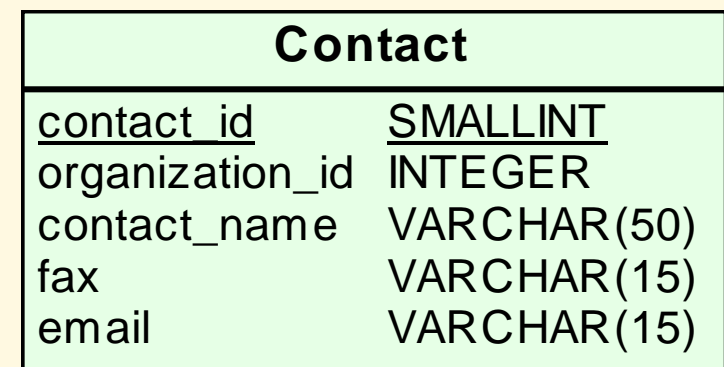
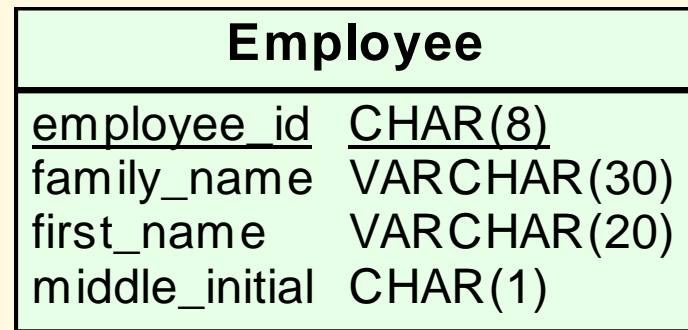
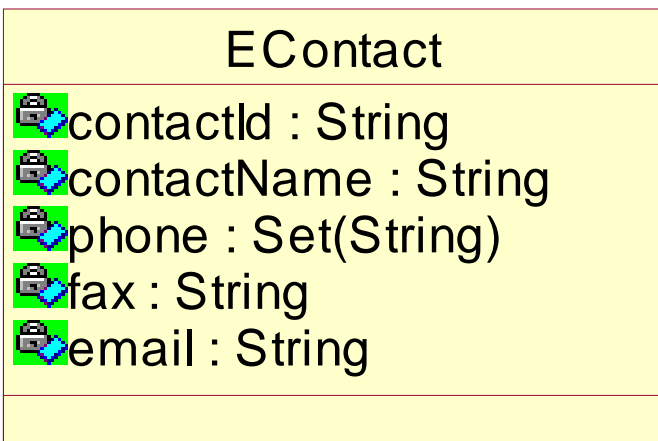
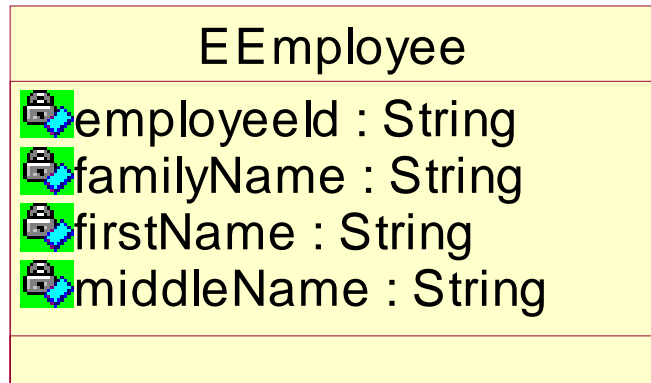
■ BCNF (Boyce–Codd NF)

■ 4th NF

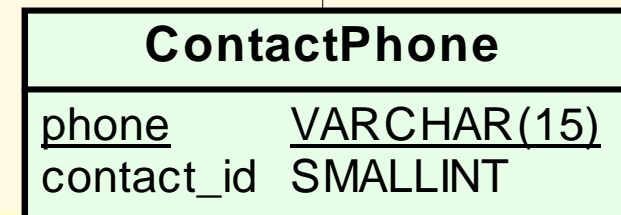
■ 5th NF

- *eliminates all update anomalies but may badly impact on performance of retrieval operations*

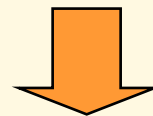
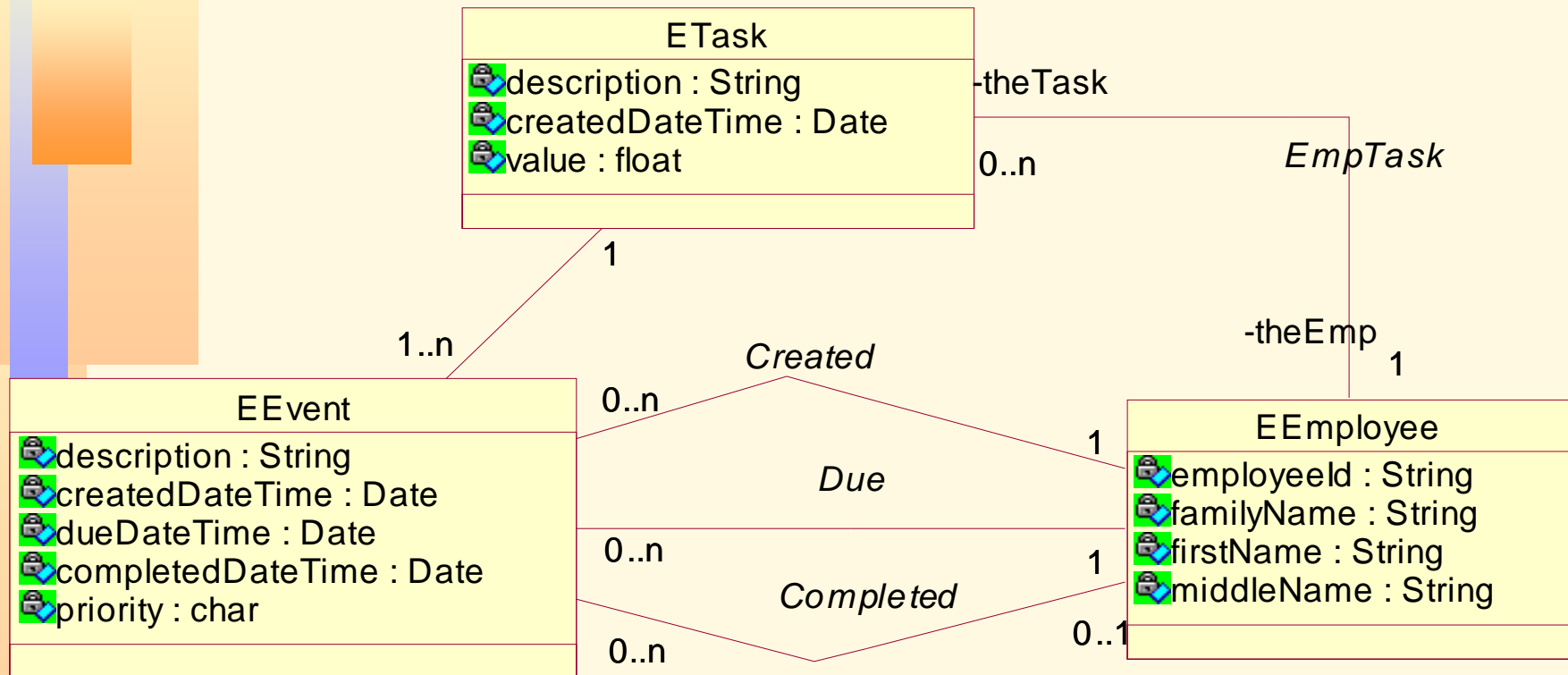
Object-relational mapping – entity classes



contact_id = contact_id

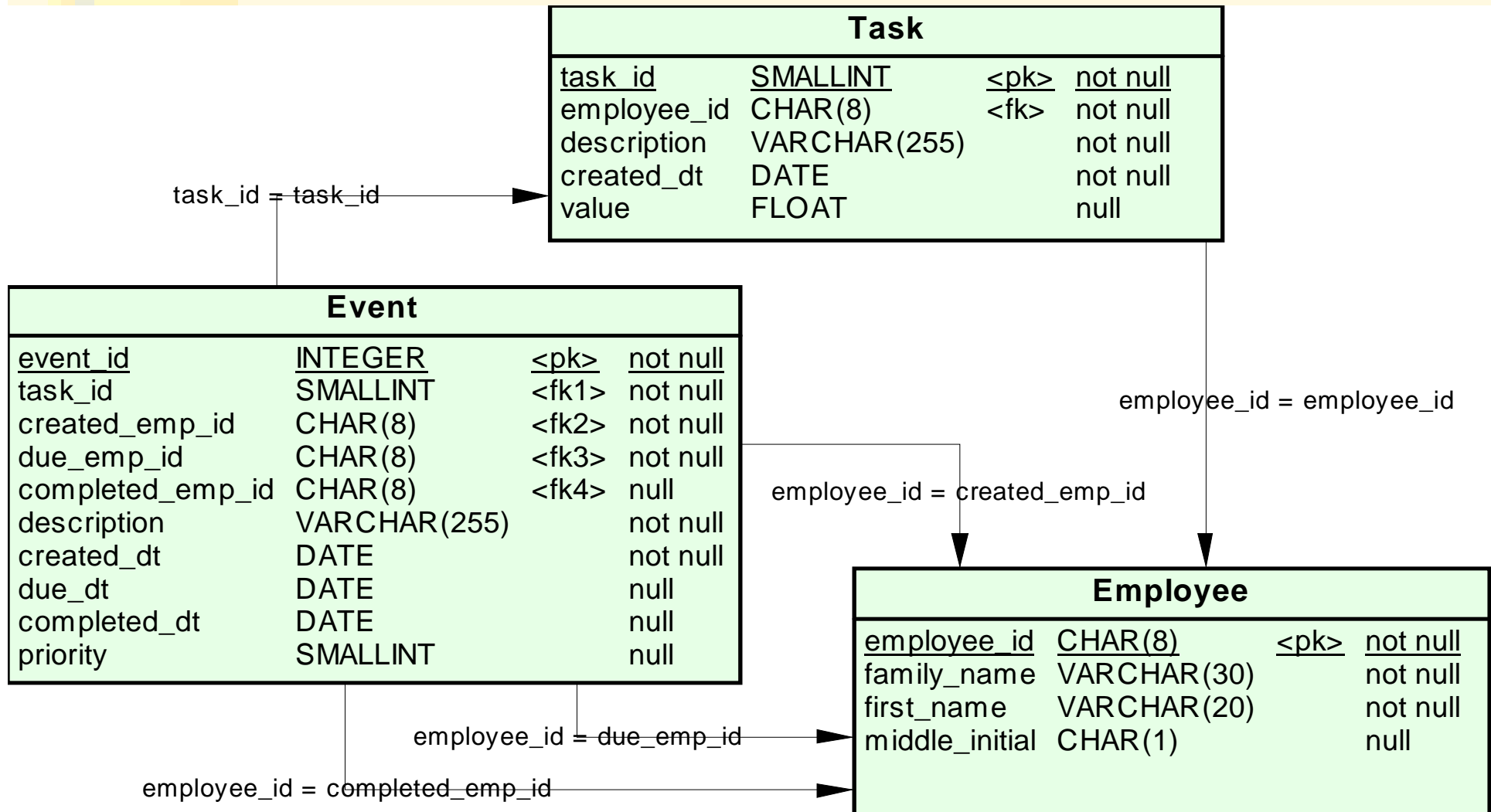


Object-relational mapping – associations

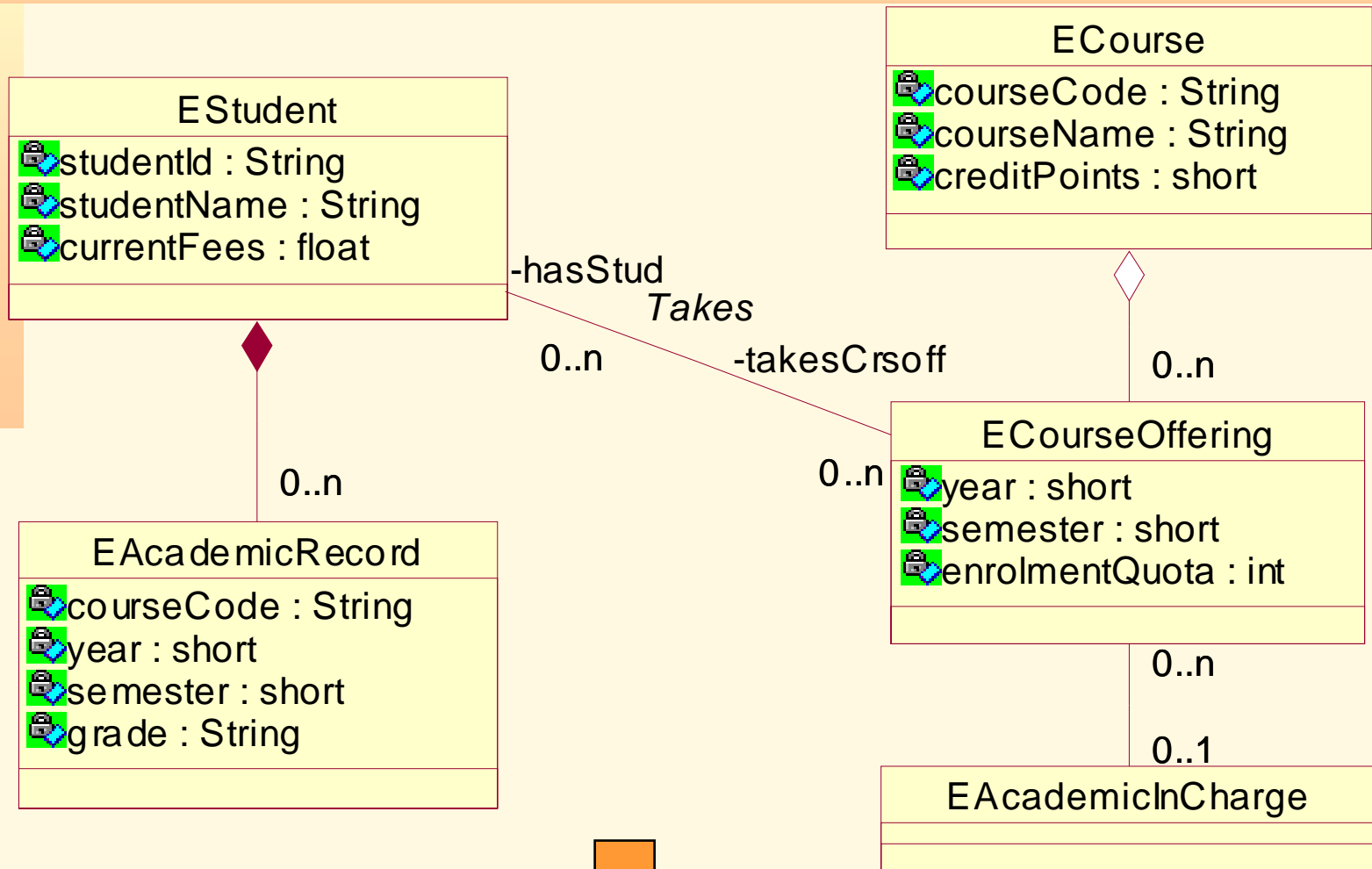


next slide

Object-relational mapping – associations

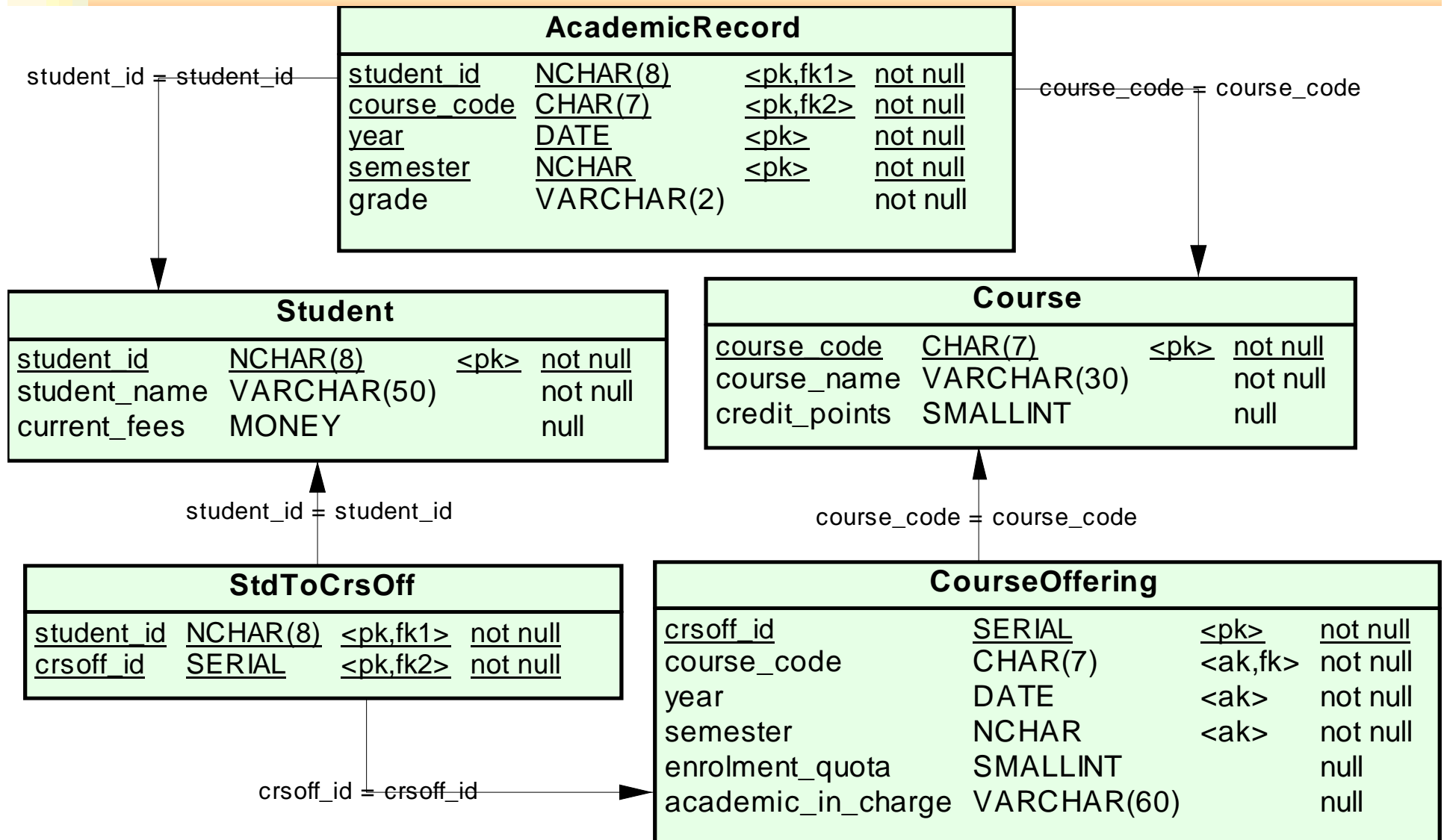


Object-relational mapping – aggregations

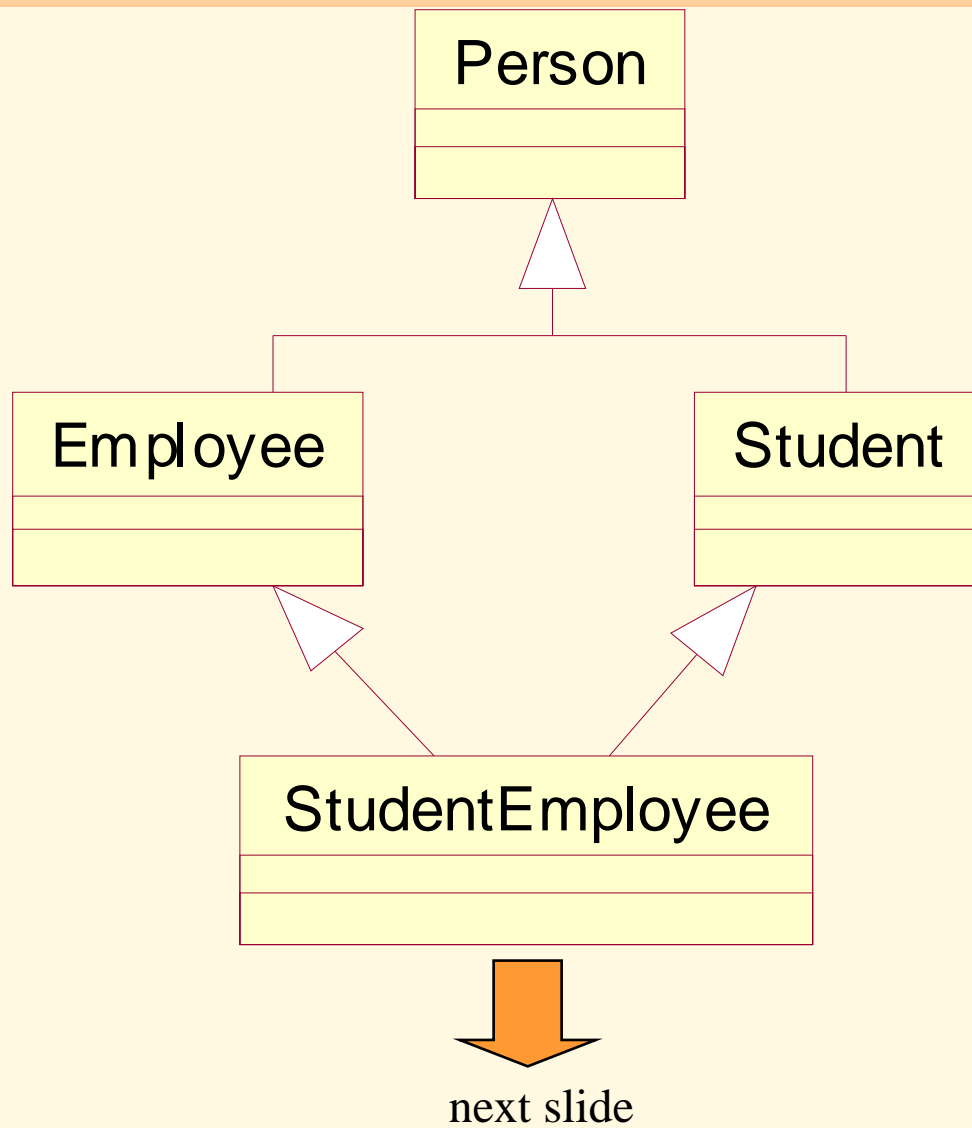


next slide

Object-relational mapping – aggregations

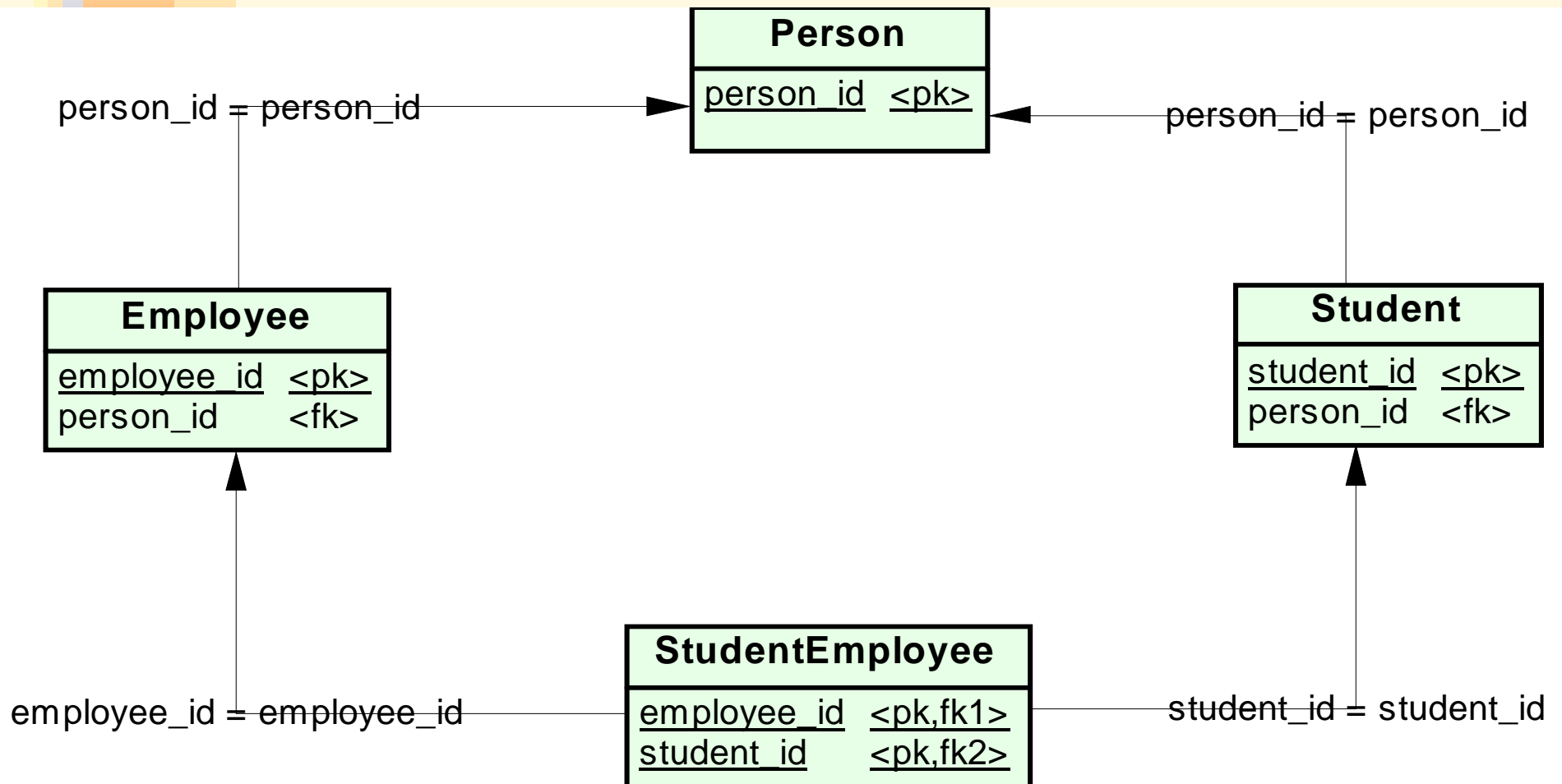


Object-relational mapping – generalizations



Object-relational mapping – generalizations

- Mapping each class to a table



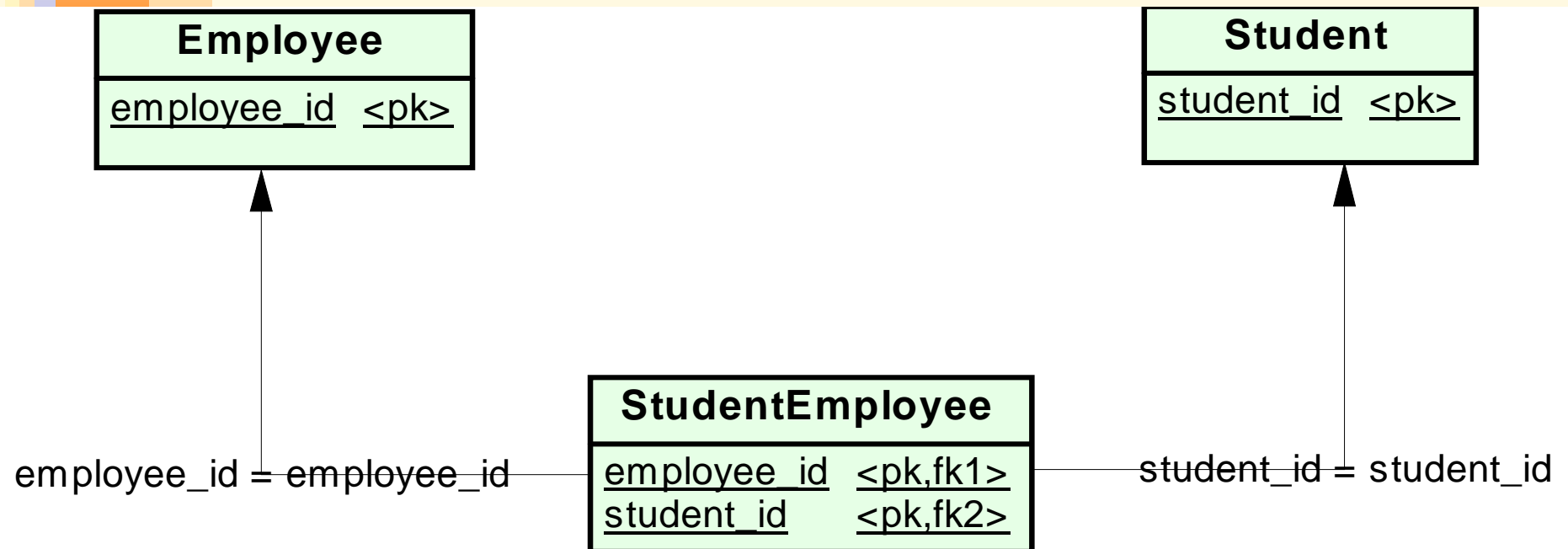
Object-relational mapping – generalizations

- *Mapping the class hierarchy to a table*

Person			
<u>person_id</u>	<u>uniqueidentifier</u>	<u><pk></u>	<u>not null</u>
is_employee	char(1)		null
is_student	char(1)		null

Object-relational mapping – generalizations

- Mapping each concrete class to a table



Object-relational mapping – generalizations

- Mapping each disjoint concrete class to a table

Employee			
<u>employee_id</u>	<u>NCHAR(8)</u>	<u><pk></u>	<u>not null</u>
is_student	BOOLEAN		not null

Student			
<u>student_id</u>	<u>NCHAR(10)</u>	<u><pk></u>	<u>not null</u>
is_employee	BOOLEAN		not null

Patterns for managing persistent objects

■ **Identity Map**

- assigning object identifiers (OIDs) to persistent objects held in memory
- mapping these OIDs to memory addresses of these objects
- mapping other identifying attributes of objects to their OIDs
- providing a single registry of object identifiers that other objects in the program can use to access objects by their OIDs

■ **Data Mapper**

- so that the program knows if a required object is in memory cache or it has to be retrieved from the database
- also knowing if an object in memory is clean or dirty

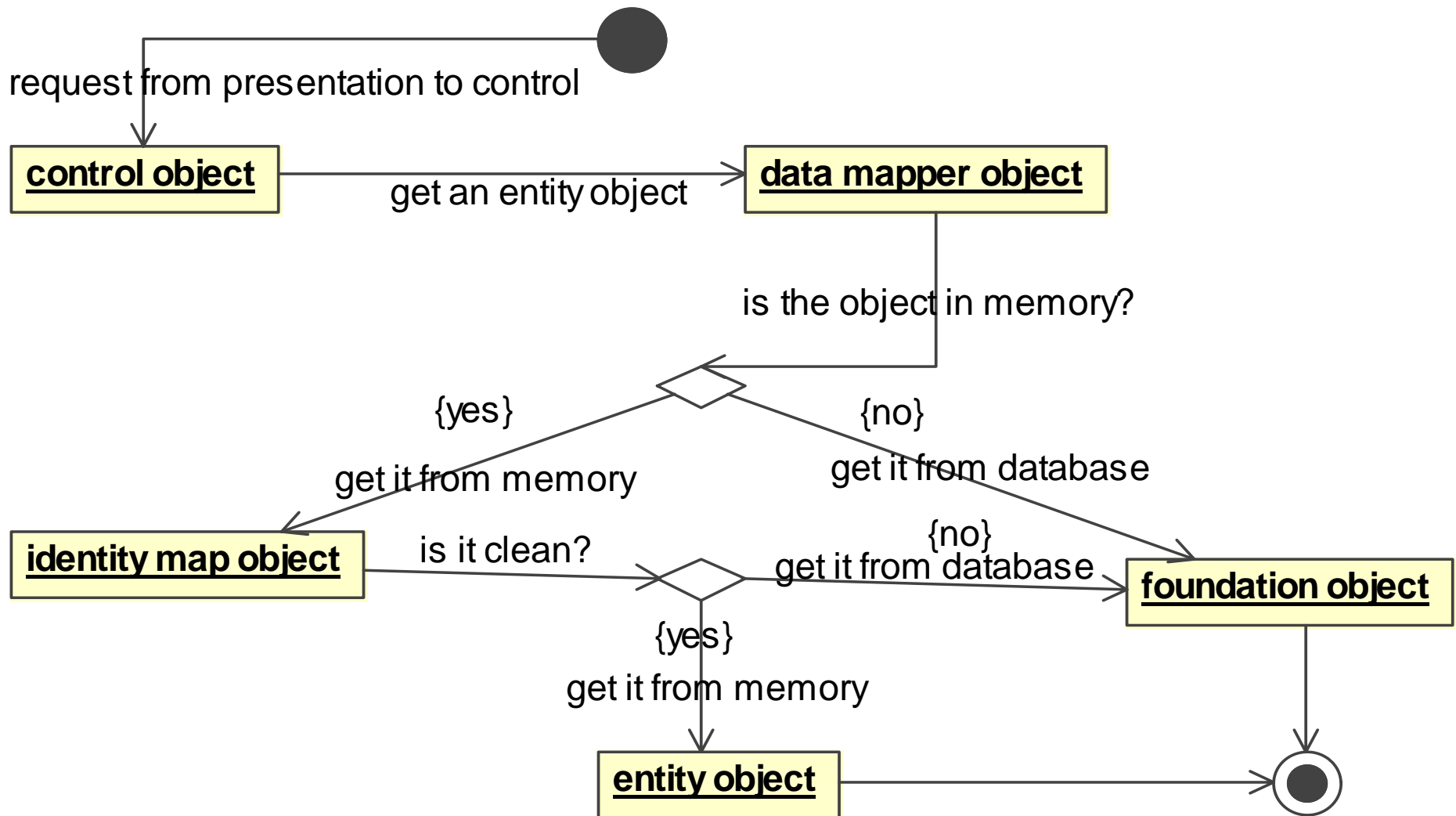
■ **Lazy Load**

- “an object that doesn’t contain all of the data you need but knows how to get it”

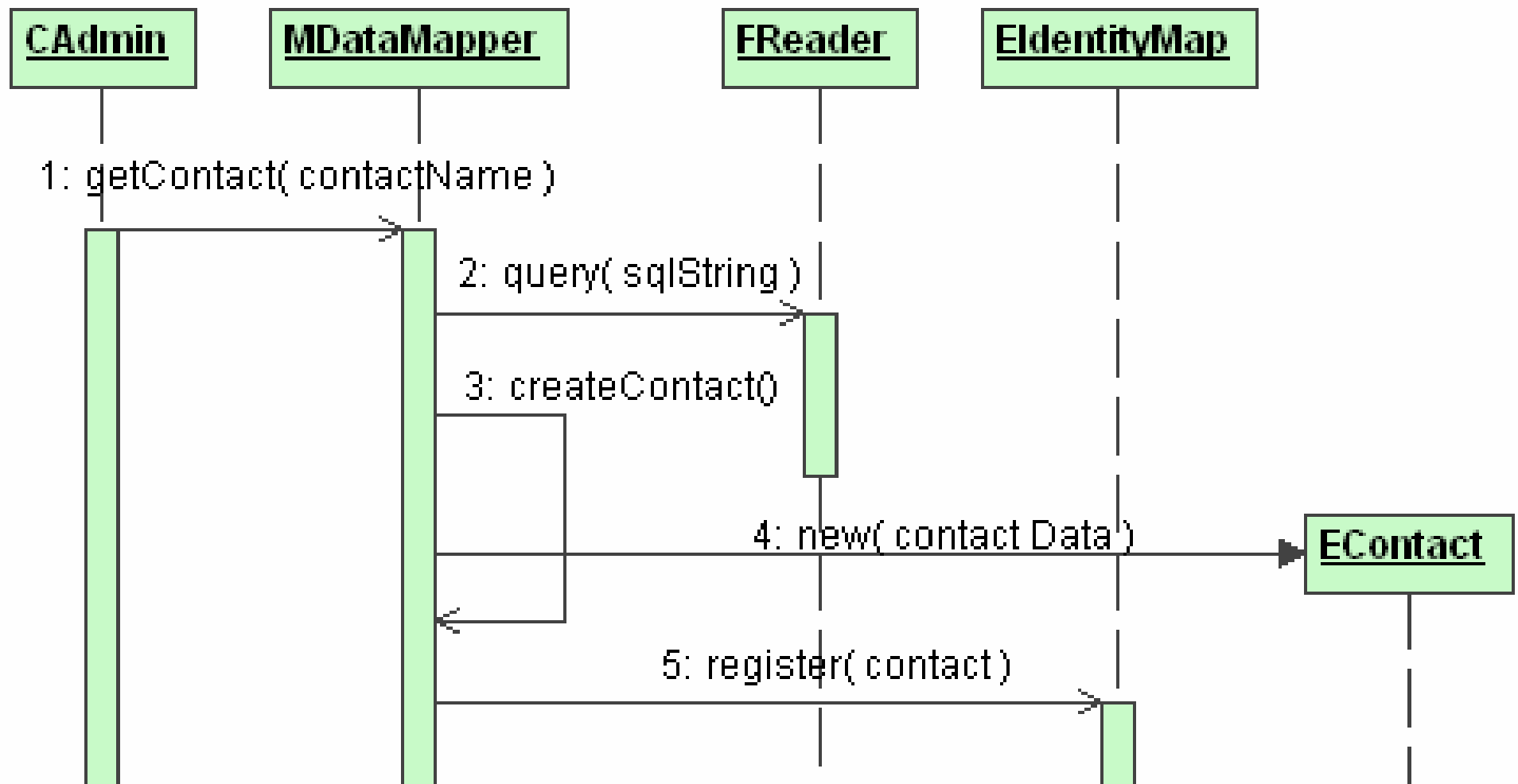
■ **Unit of Work**

- so that the program knows which objects in memory are embraced by a business transaction
- “maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems”

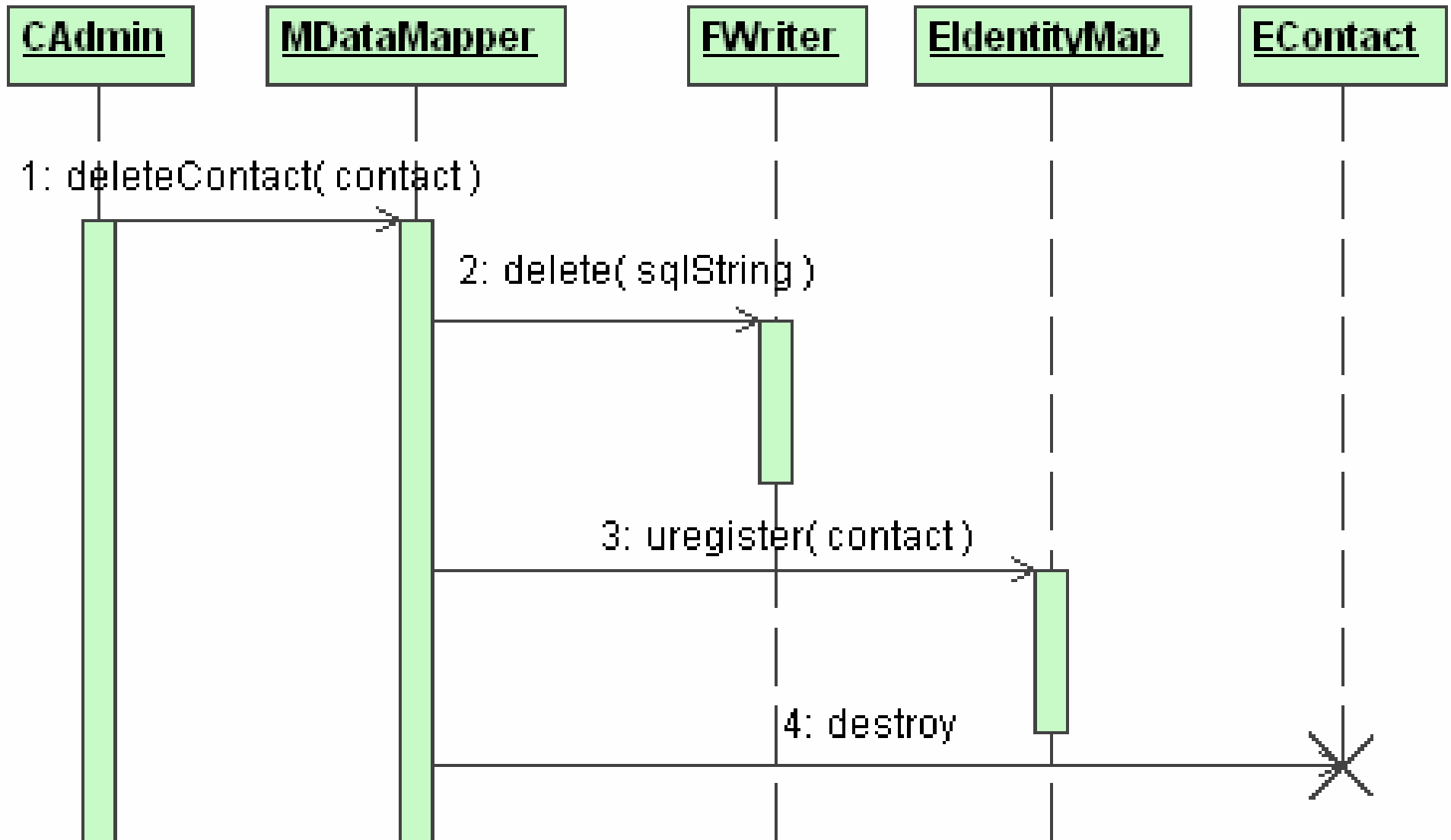
Searching for persistent objects



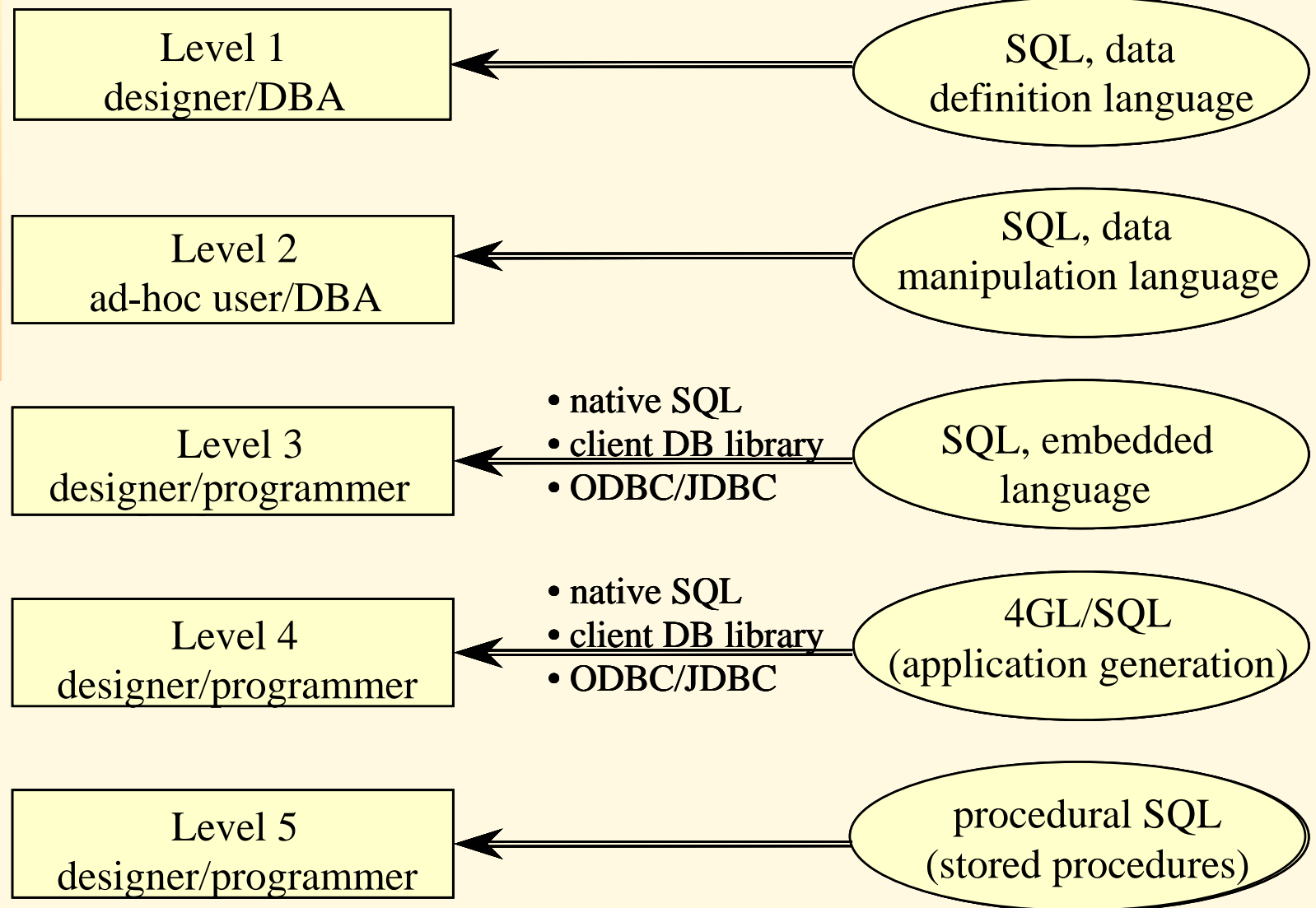
Loading persistent objects



Unloading persistent objects



Implementing database access



Designing business transactions

- **Transaction** is a logical unit of work that comprises one or more SQL statements executed by a user
- **Transaction** is a unit of database consistency – the state of the database is consistent after the transaction completes.
- **Transaction manager** of a DBMS serves two purposes:
 - database recovery and
 - concurrency control
- Transaction is **atomic** – the results of all SQL statements in the transaction are either committed or rolled back
- Concurrency control enables **multi-user concurrent access** to DB while ensuring DB consistency

Pessimistic concurrency control

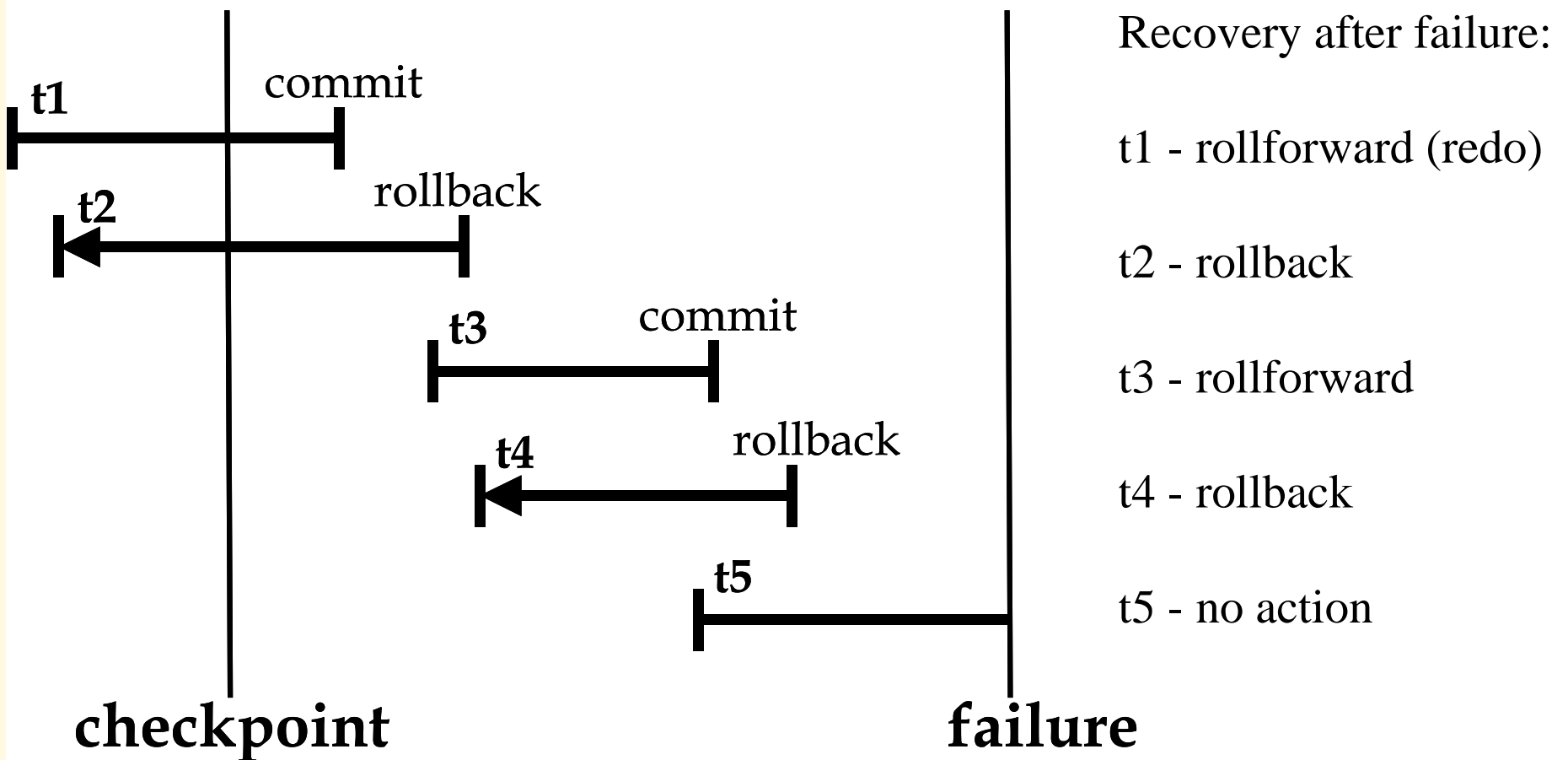
- **Locks** are acquired on every persistent object that a transaction processes:
 - **Exclusive (write) lock** – other transactions must wait until the transaction holding such a lock completes and releases the lock.
 - **Update (write intent) lock** – other transactions can read the object but the transaction holding the lock is guaranteed to be able to upgrade it to the exclusive mode, as soon as it has such a need.
 - **Read (shared) lock** – other transactions can read and possibly obtain an update lock on the object.
 - **No lock** – other transactions can update an object at any time; suitable only for applications that allow ‘dirty reads’ – i.e. a transaction reads data that can be modified or even deleted (by another transaction) before the transaction completes.

Levels of isolation

- Associated with these four kinds of **locks** are the four **levels of isolation** between concurrently executing transactions:
 - **Dirty read possible** – transaction *t1* modified an object but it has not committed yet; transaction *t2* reads the object; if *t1* rolls back the transaction then *t2* obtained an object that in a sense never existed in the database.
 - **Nonrepeatable read possible** – *t1* has read an object; *t2* updates the object; *t1* reads the same object again but this time it will obtain a different value for the same object.
 - **Phantom possible** – *t1* has read a set of objects; *t2* inserts a new object to the set; *t1* repeats the read operation and will see a ‘phantom’ object.
 - **Repeatable read** – *t1* and *t2* can still execute concurrently but the interleaved execution of these two transactions will produce the same results as if the transactions executed one at a time (this is called serializable execution).

Automatic recovery

- For most situations except the physical loss of disk data → recovery from **database backup**
- DBMS can automatically perform a **rollback** or **roll forward**



Programmable recovery

- *If the transaction has committed then the programmer can “undo” only by writing a **compensating transaction***
- *To better handle transactional failures, the programmer should use savepoints and trigger rollbacks*
 - ***Savepoint** is a named statement in a program that divides a longer transaction into smaller parts*
 - *The programmer has then an option of rolling back the work to a named savepoint rather than to the beginning of the transaction*
 - ***Trigger rollback** allows a roll back of a failed execution of a trigger rather than a roll back of the whole transaction*
 - *The program (possibly a stored procedure) can then analyze the problem and decide on further action*

Designing stored procedures and triggers

- ***Program navigation*** models could identify stored procedures and triggers
- ***These stored procedures and triggers need to be designed***

```
BEGIN
INPUT PARAMETERS (@event_id, @user_id)
Select Event (where event_id = @event_id)
IF @user_id = Event.created_emp_id
  THEN
    delete Event (where event_id = @event_id)
    IF no more events for
      Task.task_id = Event.task_id AND Event.event_id = @event_id
    THEN
      delete that Task
    ENDIF
  ELSE
    raise error ('Only the creator of the event can delete that event')
  ENDIF
END
```

Long transaction

- **Workgroup computing** (computer-supported cooperative work (CSCW)) applications require long transactions
- **Long transaction** can span computer sessions (users can take breaks then continue working in the same long transaction after returning)
 - Users work in their own **workspaces** using personal databases of data **checked-out** (copied) from the common workgroup database
 - **Long transaction** is not allowed to be automatically rolled back
 - **Short transactions** are still necessary to guarantee atomicity and isolation during the check-out and check-in operations between the group database and private databases

Summary

- *There are three levels of **data models** – external, logical and physical*
- ***Mapping of objects to databases** is the mapping of a UML class model to a logical data model within a relational database*
- *The communication of application program with a database must adhere to the **architectural framework** → the PCMEF framework*
- *There are various **design patterns** for managing persistent objects in the application code*
- *A consideration needs to be given to the five **levels of SQL** interfaces*
- ***Transaction** is a logical unit of database work that starts in a consistent database state and ensures the next consistent state when finished*
 - *Conventional database applications require **short transactions**, while some new DB applications work in **long transactions***