HY 351: Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
CS 351: Information Systems Analysis and Design

Data Management Layer Design (II)

Lecture : 16
Date : 8-12-2005

Yannis Tzitzikas
University of Crete, Fall 2005

Outline

Relational, Object-Oriented and Object-Relational Databases

- Relational Databases
  - Based on the relational model (tables, 1NF, primary keys, foreign keys, relational algebra, SLQ, views, normalization)
  - Examples of relational DBMSs: *Sybase, DB2, Oracle, MySQL, MS Access (end-user DBMS)*
- Object-Relational Databases
  - Extend the relational model to include useful features from object-orientation, e.g. complex types.
  - Add constructs to relational query languages, e.g. SQL, to deal with these extensions
  - Example of ORDBMSs: *PostgreSQL*, *UniSQL, Oracle8,*
- Object-Oriented Databases
  - Extend OO programming to include features required for database system, e.g. persistent objects.
  - Examples of OODBMSs: *ObjectStore, Versant, Objectivity, O2,* Gemstone

Object-Relational Databases

Object-Relational Databases

Object-Relational Databases
- Extend the relational model to include useful features from object-orientation, e.g. complex types.
- Add constructs to relational query languages, e.g. SQL, to deal with these extensions.

- The ORDB model was standardized in 1999, also known as SQL:1999
  - upwards compatible with SQL'92
- Commercial DBMSs
  - DB2, Informix and Oracle have extensions that provide some level of support for objects
  - Many ORDBMSs still do not support inheritance
    - so again a mapping from UML class diagrams to a schema without inheritance is required

About SQL:1999

- The object-relational features of SQL:1999 include:
  - object tables,
  - references between object tables to represent object relationships (these are called REFs), and
  - arrays to represent multi-valued associations.

- Object tables are created by first creating an object type.
- Object types are user defined types that establish the attributes, object relationships, and methods of a class.
- A type is then used to create a table.
- Instances of the table will have object identifiers as in the oo model.
- Object types can be formed into hierarchies that support inheritance ("UNDER").

Object-Relational Databases: Attributes

In contrast to the relational model, here attributes can be sets, arrays and composite.

```
create type Publisher as
(name varchar(20),
 branch varchar(20))

create type Book as
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))

create table books of Book
```

Defining a table without first defining its type.

```
create table books
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))
```


Object-Relational Databases: Inheritance over Types

Inheritance can be at the level of types or at the level of tables.

Inheritance at the level of **types**

```
create type Person as
(name varchar(20),
 address varchar(20))
```

```
create type Student under Person
(degree varchar(20),
 department varchar(20))
```

```
create type Teacher under Person
(rank varchar(20),
 department varchar(20))
```


Object-Relational Databases: Inheritance over Types: Multiple Inheritance

```
create type Person as
(name varchar(20),
 address varchar(20))
```

```
create type Student under Person
(degree varchar(20),
 department varchar(20))
```

```
create type Teacher under Person
(rank varchar(20),
 department varchar(20))
```

```
create type TeachingAssistant under Student, Teacher
```

Department is inherited twice.
Does this attribute have the same semantics?

Renaming can resolve the ambiguity

```
create type TeachingAssistant under
    Student  with (department as student-dept ),
    Teacher  with (department as teacher-dept)
```


Object-Relational Databases: Inheritance

Inheritance at the level of **tables**

```
create table people of Person
```

```
create table students of Student under people
```

```
create table teachers of Teacher under people
```

```
create table teaching-assistants of TeachingAssistant under students, teachers
```


Object-Relational Databases: Querying with Complex Types

```
create type Publisher as
(name varchar(20),
 branch varchar(20))

create type Book as
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))
```

- Composite attributes
  - E.g. "*find the title and name of publisher of each book*"

```
select title, publisher.name
from books
```

- Set-valued attributes
  - E.g. "*find all the books that have "database" as one of their keywords*"

```
select title
from books
where 'database' in (unnest(keyword-set))
```


Object-Relational Databases: Querying with Complex Types (II)

- Arrays
  - E.g. "*find the three authors of the Database System Concepts book*"

```
select author-array[1], author-array[2], author-array[3]
from books
where title = 'Database System Concepts'
```

  - E.g. "*find title-author pairs for each book*"

```
select B.title, A
from books as B, unnest(B.author-array) as A
```

```
create type Publisher as
(name varchar(20),
 branch varchar(20))

create type Book as
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))
```

Object-Relational Databases: Nesting and Unnesting

- Transforming a nested relation into 1NF

```
select name, A as author, date.day, date.month, date.year, K as keyword
from doc as B, B.author-list as A, B.keyword-list as K
```

- Transforming 1NF relation into nested relation

  Example: suppose that flat-books is the 1NF version of the table

```
select title, set(author) as author-set,
       Publisher(pub-name, pub-branch) as publisher,
       set(keyword) as keyword-set
from flat-books
group by title, publisher
```

```
create type Publisher as
(name varchar(20),
 branch varchar(20))

create type Book as
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))
```


Object-Relational Databases: Functions

Functions can be defined by users

- By using a PL or a DML (e.g. SQL). For example, an SQL (extended) function that given a document returns the number of authors:

```
create function author-count(title varchar(20))
    returns integer
    begin
    declare a-count integer;
          select count(author) into a-count
          from authors
          where authors.title = title
       return a-count
    end
```

Q: "*find the name of all documents that have more than one author*"

```
select title
from books
where author-count(title) > 1
```


Object-Relational Databases: Creating Objects and Complex Values

- Inserting a tuple into relation book
  - composite attributes: use parenthesis
  - set valued attributes: use keyword **set** and ()
  - arrays: use keyword **array** and []

```
insert into books values
('Compilers', array['Smith', 'Jones'], Publisher('McGraw-Hill', 'NY'),
   set('parsing', 'analysis'))
```

```
create type Publisher as
(name varchar(20),
 branch varchar(20))

create type Book as
(title varchar(20),
 author-array varchar(20) array[10],
 pub-date date,
 publisher Publisher,
 keyword-set setof(varchar(20)))
```


Object-Oriented Databases

Object-Oriented Databases

Is the result of combining o-o programming principles with database management principles

- OO concepts such as *encapsulation*, *polymorphism* and *inheritance* are enforced as well as database management concepts such the *ACID properties* (Atomicity, Consistency, Isolation and Durability) which lead to system integrity, support for an ad hoc query language and secondary storage management systems which allow managing very large amount of data.

The ***Object Oriented Database Manifesto*** [Atkinson et al. 1989] lists the following features as mandatory for a system to support before it can be called OODBMS:

- Complex Objects, Object Identity, Types and Classes, Class or Type Hierarchies, Overriding, Overloading and late binding, Computational Completeness, Extensibility, Persistence, Secondary Storage management, Concurrency, Recovery and Ad Hoc Query Facility

http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

Object-Oriented Databases: The distinctive features

- Each object has an OID that is permanent and system generated
- Accessing objects in the database is done in a transparent manner
  - the interaction of persistent objects is no different from the interaction of in-memory objects.
- So you are not obliged to use a query language (like SQL in relational DBMS)
  - however Object SQL can be used for making queries outside of a programming environment
- When a client requests an object from the database, the object is transferred from the database into the applications' memory where it can be used either as
  - a transient value (i.e. disconnected from its representation in the database), or as
  - a mirror of the version in the database
    - updates to the object are reflected in the database
    - changes to objects in the database require that the object is refetched from the database.
- Database operations typically involve obtaining a database root from the OODBMS which is usually a data structure (like a graph, vector, hash table, or set) and traversing it to obtain objects to create, update or delete from the db.

Object-Oriented notions vs Relational databases notions

class ⟷ table

extent: instances of a class ⟷ tuples of the table

instance ⟷ tuple

methods ⟷ No support

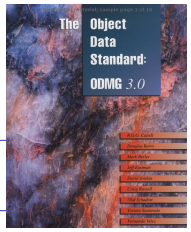
•… ~ stored procedures

methods are computationally complete (general purpose control and computational structures are provided)

ODL - OML - OQL

- The Object Database Standard: ODMG 3.0 (www.odmg.org)
  - ODMG: a consortium of object-oriented DBMS vendors and users
  - *Standard for object-relational mapping products as well as object DBMSs*
  - Free sample from the ODMG 2.0 book are available at
    - http://www.odmg.org/odmgbookextract.htm

- ODL: Object Definition Language
- OML: Object Manipulation Language
- OQL: Object Query Language



The ODMG is now an Object Storage API standard that can work with any DBMS or tool.
See also: JDO (Java Data Object) API.

Object Oriented DBMSs and Applications

Examples of pure OODBMSs:
- Gemstone, Jasmine
- O2, Objectivity
- Object-Store, POET
- Versant, Ontos, Poet, EyeDB

Open source OODBMSs:
- Ozone
- Zope
- FramerD
- XL2

Mainly used for multimedia applications (that involve complex data: graphics, video, sound).

Some systems (that handle mission critical data) that are based on OODBMSs:

- Chicago Stock Exchange: uses Versant
- CERN (Large Hadron Collider): uses Objectivity
- Stanford Linear Accelerator Center (SLAC): uses Objectivity
  - 169 terabytes (Nov 2000)
- SouthWest Airline's Home Gate: uses ObjectStore
- Iridium System (by Motorola): Data repository for system component naming, satellite mission planning data and orbital management data : uses Objectivity

Advantages of OODBMSs (versus RDBMSs)

Advantages of OODBMSs
- Avoid the "impedance mismatch"
- They are better in handling complex data (allow storing Composite Objects)
  - in a RDBMS we would either have to define a table with many columns and a lot of null values, or a set of tables linked via foreign keys (querying would have joins)
- Class Hierarchy
- A QL is not necessary to access the db
- No need to define primary keys (ODBMS does this behind the scenes via OIDs)

Disadvantages of OODBMSs (versus RDBMSs)

Disadvantages of OODBMSs
- Schema changes are … hard
  - In a RDBMS changes at the schema are independent from the application programs
  - in an OODBMS changes at the schema require recompiling the application program
- They are language dependent (usually tied to an o-o PL)
  - OODBMSs are typically accessible from a specific PL using a specific API
- Lack of Ad-Hoc Queries
  - in RDBMS and with SQL we can define queries that create new tuples from joining existing tables and querying them. It is not possible to duplicate the semantics of joining 2 tables by joining two classes (so OODBMSs are less flexible than RDBMSs from that perspective)

A four step design approach:

(A) Select the format of the storage
- *Files*
- *Relational Databases*
- *Object-Relational Databases*
- *Object-Oriented Databases*

(B) Map problem domain objects to object-persistence formats

(C) Optimizing the object-persistence formats

(D) Design data access and manipulation classes

Selecting an Object Persistence Format

	Sequential and Random Access Files	Relational DBMS	Object Relational DBMS	Object-Oriented DBMS
Major Strengths	Usually part of an object-oriented programming language Files can be designed for fast performance Good for short-term data storage	Leader in the database market Can handle diverse data needs	Based on established, proven technology (e.g., SQL) Able to handle complex data	Able to handle complex data Direct support for object-orientation
Major Weaknesses	Redundant data Data must be updated using programs (i.e., no manipulation or query language) No access control	Cannot handle complex data No support for object-orientation Impedance mismatch between tables and objects	Limited support for object-orientation Impedance mismatch between tables and objects	Technology is still maturing Skills are hard to find
Data Types Supported	Simple and Complex	Simple	Simple and Complex	Simple and Complex
Types of Application Systems Supported	Transaction processing	Transaction processing and decision making	Transaction processing and decision making	Transaction processing and decision making
Existing Storage Formats	Organization dependent	Organization dependent	Organization dependent	Organization dependent
Future Needs	Poor future prospects	Good future prospects	Good future prospects	Good future prospects

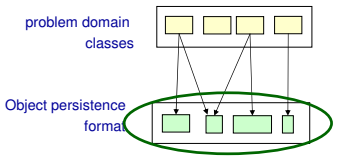
Dennis et al. 2005

A four step design approach:

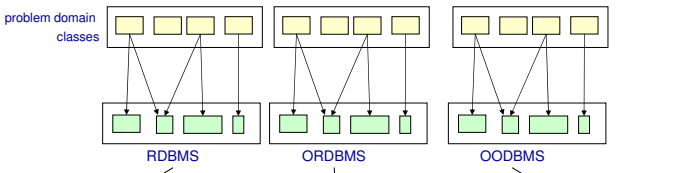
(A) Select the format of the storage

(B) Map problem domain objects to object-persistence formats

we have to map the structures defined by the UML class diagrams to data structures that are recognized by the database model selected



Map problem domain objects to object-persistence formats



"Impedance mismatch": caused by having to map objects to tables and vice versa is a performance penalty.

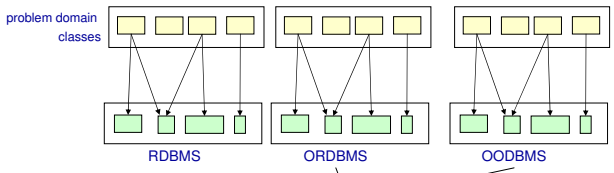
Conversion method: See Lecture 15.

Less "impedance mismatch":
A conversion is still needed depending on the support of inheritance (some ORDBMSs do not support inheritance).

Less "impedance mismatch":
A conversion is still needed depending on the support of inheritance (e.g. some do not support multiple inheritance)

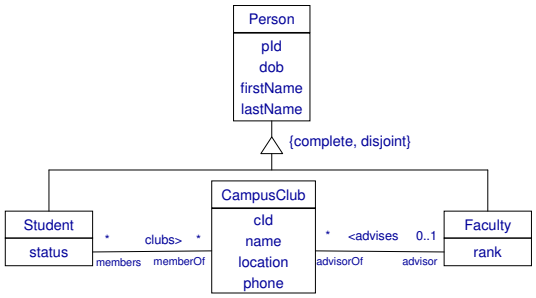
We can apply the rules described in the lecture about "Class and Method Design"

Map problem domain objects to object-persistence formats

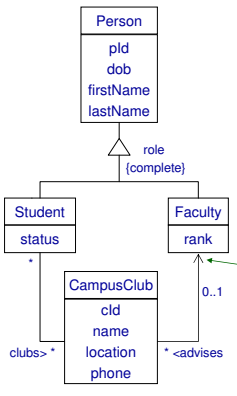


- Even in these cases it is suggested to create one object persistence class in the ORDBMS/OODBMS for every concrete problem domain class that needs persistent storage.
- Later on we will define data access and manipulation (DAM) classes that contain the functionality required to manage the interaction between each such pair of classes. This design increases portability.

Example: A class diagram



Its translation to the Relational model

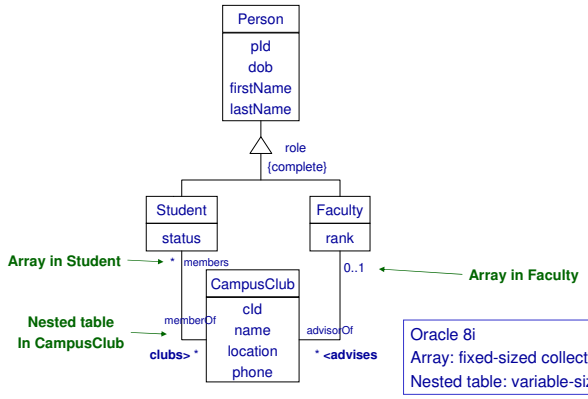


Person(pId, dob, firstName, lastName)
Student(pId, status)
foreign key (pId) references Person(pId)
Faculty (pId, rank)
foreign key (pId) references Person(pId)
CampusClud(cId, name, phone, location, advisor)
foreign key(advisor) references Faculty(pID)
Clubs(pId,cId)
foreign key (pId) references Person(pId)
foreign key (cId) references CampusClub(cId)

The unidirectional association here just indicates what is possible in the above relational schema (the reverse direction can be supported by posing a query).

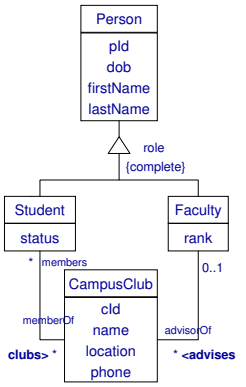
An alternative approach would be to flatten the isA hierarchy (see Lectures 13,15)

Translating to an Object-Relational DBMS



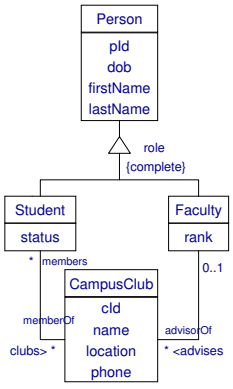
Oracle 8i
Array: fixed-sized collection
Nested table: variable-sized collection

Its translation to an ORDBMS: SQL:1999 (1/2)




```
CREATE TYPE person_udt AS (
  pID    VARCHAR(11),
  dob    DATE,
  firstName   VARCHAR(20),
  lastName   VARCHAR(20))
NOT FINAL
REF IS SYSTEM GENERATED;

CREATE TABLE person OF person_udt (
     CONSTRAINT person_pk PRIMARY_KEY(PID),
     REF IS oid SYSTEM GENERATED);

CREATE TYPE faculty_udt UNDER person_udt AS (
  rank    VARCHAR(20),
  advisorOf  REF(campusClub_udt) SCOPE campusClub
     ARRAY[20])
NOT FINAL
CREATE TABLE faculty OF faculty_udt under person;
```


Its translation to an ORDBMS: SQL:1999 (2/2)




```
CREATE TYPE student_udt UNDER person_udt AS (
  status    VARCHAR(20),
  clubs     REF(campusClub_udt) SCOPE campusClub ARRAY[20])
NOT FINAL
CREATE TABLE student OF student_udt under person;

CREATE TYPE campusClub_udt UNDER person_udt AS (
  cID       VARCHAR(11),
  name      VARCHAR(25),
  location  VARCHAR(25),
  phone     VARCHAR(25),
  advisor     REF(faculty_udt) SCOPE faculty,
  members  REF(students_udt) SCOPE student  ARRAY[100])
NOT FINAL
REF IS SYSTEM GENERATED;
CREATE TABLE campusClub OF campusClub_udt (
     CONSTRAINT campusClub_pk PRIMARY KEY (cID),
     REF IS oid SYSTEM GENERATED);
```

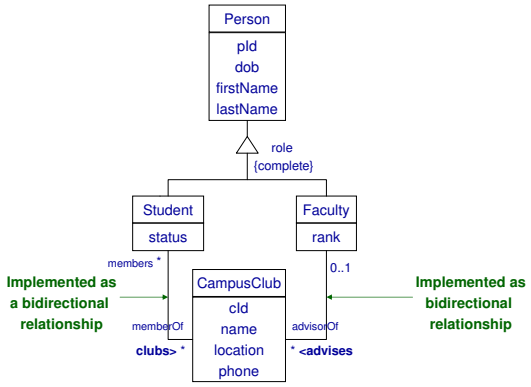

SQL:1999

- FINAL types may not have subtypes
- The table of a type has one column for each attribute of its type plus one column to define REF value for the row (object id).

- REF
  - User generated (REF USING <predefined type>)
  - System generated (REF IS SYSTEM GENERATED)
  - Derived from a list of attributes (REF (<list of attributes>)
    - Default is system generated

```
CREATE TYPE real_estate AS (owner REF (person), ...)
NOT FINAL REF USING INTEGER

CREATE TYPE person AS (ssn INTEGER, name CHAR(30),...)
NOT FINAL REF (ssn)
```


Translating to an Object-Oriented DBMS

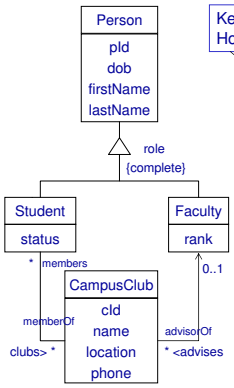


Target OODBMS
Translating associations: *Attributes or Relationships?*

An association (of the class diagram) can be represented as
- an attribute (at one side) // if of course it is not N-M
- two attributes (one at each side)
  - In this case the application programmer is responsible for maintaining the consistency in both sides
- as a bidirectional relationship
  - These are stored at both sides and their consistency is automatically maintained by the DBMS
    - A modification to one side results in the automatic maintenance of the data on the other side of the relationship
    - These relationships should be explicitly defined in ODL (direct and inverse direction)

In ORDBMS this is not possible. We have to write triggers for maintaining consistency

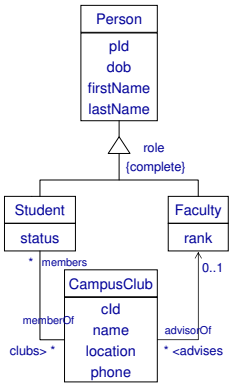
Its translation to an OODBMS (e.g. Objectivity): using ODL (1/2)



Keys are optional in ODL since each object has a unique OID. However we can declare keys using the keyword "key".

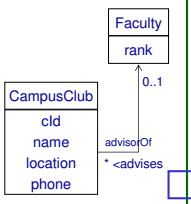
```
class Person
(extent people
key pID)
{attribute string pID;
 attribute date dob;
 attribute string firstName;
 attribute string lastName;
}

class Student extends Person
(extent students)
{ attribute string status;
  relationship set <CampusClub> memberOf
                inverse CampusClub::members;
}
```


Its translation to an OODBMS (e.g. Objectivity): using ODL (2/2)




```
class Faculty extends Person
(extent facultyMembers)
{attribute string rank;
 relationship set <CampusClub> advisorOf  inverse
     CampusClub::advisor;
}

class CampusClub
(extent campusClubs
  key cID)
{attribute string cID;
 attribute string name;
 attribute string location;
 attribute string phone;
 relationship set <Student> members  inverse
     Student::memberOf;
 relationship Faculty advisor inverse Faculty::advisor;
}
```

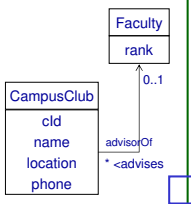

Its translation to an OODBMS (e.g. Objectivity) Remark




```
class Faculty extends Person
(extent facultyMembers)
{attribute string rank;
 relationship set <CampusClub> advisorOf  inverse CampusClub::advisor;
}

class CampusClub
(extent campusClubs
  key cID)
{attribute string cID;
 attribute string name;
 attribute string location;
 attribute string phone;
 relationship set <Student> members  inverse Student::memberOf;
 relationship Faculty advisor inverse Faculty::advisor;
}
```

Here we translated this association as a relationship

Its translation to an OODBMS (e.g. Objectivity): Remark (II)




```
class Faculty extends Person
(extent facultyMembers)
{attribute string rank;
 attribute  set <CampusClub> advisorOf;
}

class CampusClub
(extent campusClubs
  key cID)
{attribute string cID;
 attribute string name;
 attribute string location;
 attribute string phone;
 relationship set <Student> members  inverse Student::memberOf;
 attribute Faculty advisor;
}
```

Here we translated this association using attributes

A four step design approach:
(A) Select the format of the storage
(B) Map problem domain objects to object-persistence formats
(C) Optimizing the object-persistence formats
(D) Accessing the database from the code
Design data access and manipulation classes

(C) Optimizing the object-persistence formats

Dimensions of optimization:
- Storage efficiency (minimizing storage space)
- Speed of access (minimizing time to retrieve desired information)

This task is often called Physical Database Design

As the objective of normalization in relational databases (recall lecture 15) aims at reducing redundancies, we can consider that the normalization falls into this category.

Estimating Data Storage Size and Number of Accesses

- Estimating data storage size
  - For each table
    - calculate the tuple size
    - estimate the number of tuples at the beginning and its growth rate (e.g. in a per year basis)
  - Estimate the storage size of the entire database in 1 year, 2 years, …
- Estimating workload and response times
  - Estimate the frequency of each use case scenario
  - For each use case identify the operations that need to access the database
  - For each operation see which tables need to be accessed, the type of access (read/write) and count the average number of tuples that need to be accessed.
  - From the above we can estimate the response time of an operation

Estimating Data Storage Size and Number of Accesses and Non-Functional Requirements

- Estimating data storage size
- Estimating response time

compare ↔ Non functional requirements

↓

Identify problematic cases

Storage:
Revisit the definition of attribute (field sizes).
Employ coding and compression techniques.

Response time:
Identify potential efficiency problems and investigate whether redundancy (at the data storage level) can alleviate the problem.

Denormalization

Denormalization is the process of spliting or combining normalized relations into physical tables based on afinity of use of rows and fields.
- Denormalization by columns
- Denormalization by rows

- By placing data used together close to one another on disk, the number of I/O operations needed to retrieve all the data needed by a program is minimized.
- Denormalization is best suited for data that are accessed very frequently and rarely updated

The capability to split a table into separate sections, often called partitioning, is possible in most commercial DBMSs. For example Oracle 9i supports:
- range partitioning: partitions are defined by non-overlapping ranges of values from a specified attribute
- hash partitioning: a table row is assigned to a partition by an algorithm and then maps the specified attribute value to a partition
- composite partitioning: combines range and hash partitioning by first segregating data by ranges on the designated attribute, and then within each of these partitions it further partitions by hashing on the designated attribute

– each partition is stored in a separate contiguous section of disk space, which Oracle calls a tablespace.

Guidelines for Creating Indexes

- There is a trade-off between improved performance on retrievals and degrading performance for inserting, deleting and updating rows.
- So indexes should be used generously for databases intended primarily to support data retrievals (e.g. decision support applications).
- Use indexes sparingly for transaction systems and applications with heavy updating requirements.
- Typically, for each each table we usually create
  - a unique index based on the primary key
  - an index based on the foreign key
- Create an index for fields used frequently for grouping or sorting.

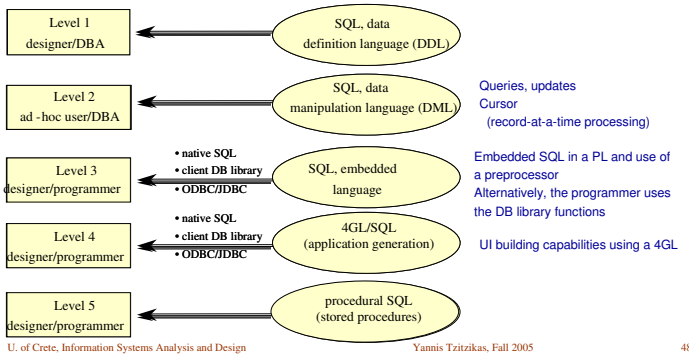
(D) Designing database management classes

Levels of accessing a database

DAM (data access and manipulation) classes

Levels of Accessing a Database Case: RDB

Concerning how a client program communicates with a database server note that SQL comes in different dialects that can be used at different levels of programming abstraction.



Accessing the database from a PL
Object-Oriented vs Relational

Relational vs OO DBMS
Example Java code for an instant messaging appl.

1. Validating a user

ObjectStore (OODBMS)

```
import COM.odi.*;
import COM.odi.util.query.*;
import COM.odi.util.*;
import java.util;
try {
  // start database session
  Session session = Session.create(null,null);
  session.join()
  // open database and start transaction
  Database db =
          Database.open("Imdatabase", ObjectStore.UPDATE);
Transaction tr = Transaction.begin(ObjectStore.READONLY);
//get hashtable of user objects from DB
  OSHashMap users = (OSHaspMap)
                                  db.getRoot("IMusers");
// get password and username from user
String username = getUserNameFromUser();
String passwd    = getPasswordFromUser();
```

IBM's DB2 (RDBMS)

```
import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;
import java.util;

try {
  // launch instance of database driver

Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance
();
  // create database connection
  Connection con =
DriverManager.getConnection("jdbc:db2:Imdatabase");
 // get password and username from user
String username = getUserNameFromUser();
String passwd    = getPasswordFromUser();
```


Relational vs OO DBMS
Example Java code for an instant messaging appl. (2)

1. Validating a user (cont)

ObjectStore

```
// get user object from db and see if it exists
UserObject user = (UserObject) users.get(username);

if (user == null)
   System.out.println("Non-existent user");
else
  if (user.getPassword().equals(passwd)
    System.out.println("Successful login");
  else
     System.out.println("Invalid Password");
//end transaction, close db and terminate session
tr.commit();
db.close();
session.terminate();
}
// exception handling ...
```

IBM's DB2

```
// perform SQL query
Statement sqlQry = conn.createStatement();
ResultSet rset = sqlQry.executeQuery("SELECT password
from user_table WHERE username='" + username +"'");
if (rset.next()){
  if (rset.getString(1).equals(passwd))
    System.out.println("Successful login");
     else
       System.out.println("Invalid Password")
 } else System.out.println("Non-existent user");
// close database connection
sqlQry.close()
conn.close();
}
// exception handling ...
```

Remark: It is more .. "clean" to perform operations on a UserObject than on a ResultSet

Relational vs OO DBMS
Example Java code for an instant messaging appl. (3)

2. Getting user's contact lists

ObjectStore

```
import COM.odi.*;
import COM.odi.util.query.*;
import COM.odi.util.*;
import java.util;
try {
  /* start session and open db, as before */
  //get hashtable of user objects from DB
  OSHashMap users = (OSHaspMap) db.getRoot("IMusers");
UserObject u = (UserObject) users.get("MARIA");
UserObject[] contactList = u.getContactList();

System.out.println("These are persons of the contact list");
for (int i=0; i< contactList.length; i++)
   System.out.println(contactList[i].toString());

/* close session as before */
```

IBM's DB2

```
import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;
import java.util;
try {
  // launch instance of database driver
Statement sqlQry = conn.createStatement();
ResultSet rset = sqlQry.executeQuery("SELECT fname,
lname, user_name, online_status, webpage FROM
contact_list, user_table WHERE
contact_list.owner_name='MARIA' and
contact_list.buddy_name=user_table.user_name");
System.out.println("These are persons of the contact list");
while (rset.next())
   System.out.println("Full Name:" + rset.getString(1) + " "
+ rset.getString(2) + ".....

/* close session and db as before */
```


Relational vs OO DBMS
Example Java code for an instant messaging appl. (3)

3. Get all on-line users

ObjectStore

```
Query q = new Query = (UserObject.class,
                        "onlineStatus.equals(\"online\"");

Collection users = db.getRoot("Imusers");
Set onlineUsers = q.select(users);

Iterator iter = onlineUsers.iterator();

while (iter.hasNext())
{
   UserObject user = (UserObject) iter.next();
            < do something >
}
```

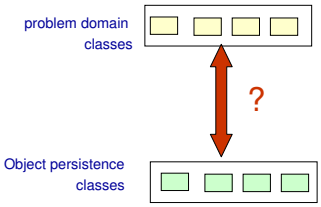
IBM's DB2

```
Statement sqlQry = conn.createStatement();
ResultSet rset = sqlQry.executeQuery("SELECT fname,
laname, user_name, online_status,wepage FROM
user_table WHERE online_status='online'");

while (rset.next())
{
  UserObject user = new UserObject(rset.getString(1),
rset.getString(2), rset.getString(3) ....)

                < do something>
}
```


(D) Designing database management classes

DAM (data access and manipulation) classes

Designing database management classes



Classes are less reusable if they are tightly coupled to the mechanisms by which instances are stored in some kind of file system or database

Designing database management classes Available Options

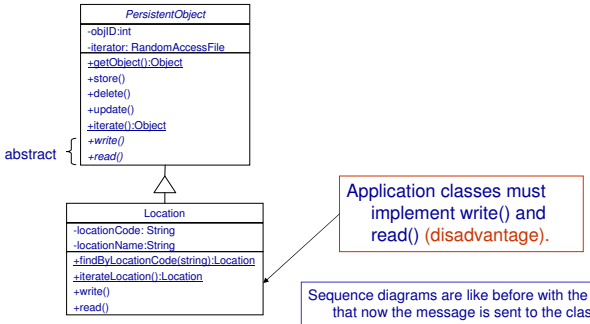
[i] Add operations to each class to enable objects to save and store themselves
- This reduces the reusability of the problem domain classes (low class cohesion)
- If an object is not currently instantiated how we can send it a message to invoke an operation to load itself?

[ii] We can bypass the last problem by making the storage and retrieval operations class-scope methods rather than instance-scope methods
- Ok, but this still reduces the reusability of the problem domain classes (low cohesion again)

[iii] All persistent objects could inherit methods for storage from an abstract superclass *PersistentObject*.
- It couples all problem domain classes that need persistent storage to the superclass PersistentObject, as they all inherit from this superclass (low inheritance cohesion)

Designing database management classes Option [iii] in more detail

[iii] All persistent objects could inherit methods for storage from an abstract superclass *PersistentObject*.



Application classes must implement write() and read() (disadvantage).

Sequence diagrams are like before with the exception that now the message is sent to the class and the class then instantiates the object ….

Designing database management classes Available Options (II)

[iv] Introduce separate classes into the system to deal with storage and retrieval. This is the "database broker" approach.
- The advantage is that problem domain classes contain nothing that indicates how they are to be stored, so they can be reused unchanged with different storage mechanisms.

[v] Use only one data storage class. Different instances of this class will be created with attributes to hold the names of tables/files that are to be used to store and retrieve instances of their associated class.
- more difficult to set up and implement

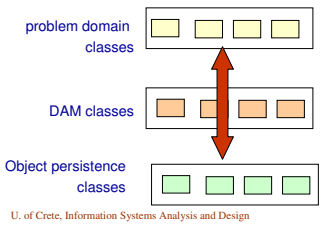
More OO developers prefer option [iv].

Option [iv] involves a number of patterns.

Designing database management classes Options [iv] in more detail

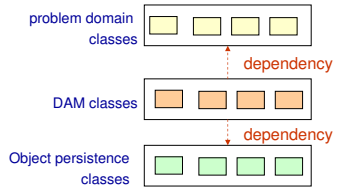
[iv] Introduce separate classes into the system to deal with storage and retrieval. This is the "database broker" approach.
- The advantage is that problem domain classes contain nothing that indicates how they are to be stored, so they can be reused unchanged with different storage mechanisms.

These classes are called broker classes or DAM (Data Access and Manipulation) classes



- The **DAM (Data Access and Manipulation) classes** act as "translators" between the object persistence and the problem domain objects.
- They should be able to read and write both object persistence and the problem domain objects (so they provide mechanisms to materialize objects from the db and to dematerialize them back to the db).

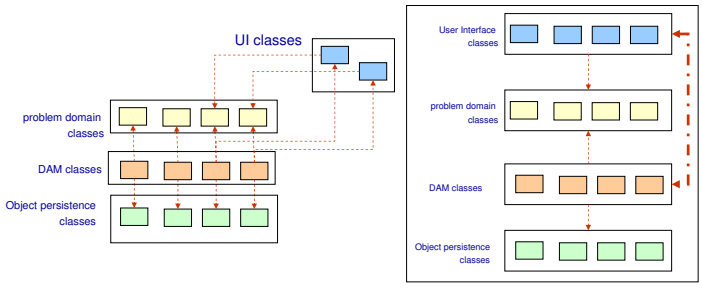
DAM (data access and manipulation) classes



- Object persistence classes are created for the concrete problem domain classes
- DAM classes depend on both problem domain and object persistence classes

DAM classes and UI classes

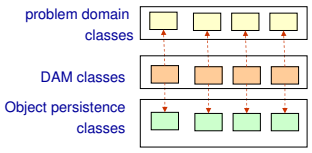
Sometimes DAM classes are associated to UI classes.
This however adds extra dependencies (which in general is undesirable).



DAM classes and problem domain classes

The DAM classes will be placed in a separate package.

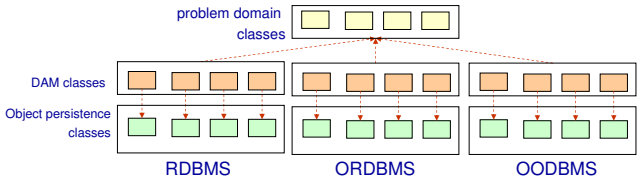
Rule: one DAM class per problem domain class



Remarks:

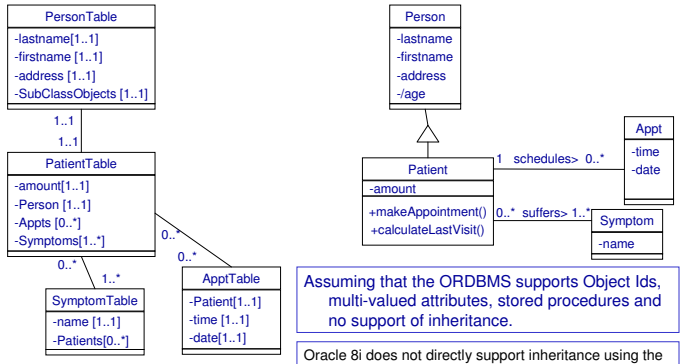
- Each DAM class is responsible for a one-to-one translation

Important Remark



- *Notice that in this way the problem domain classes remain unchanged*
- *We have kept them independent from the underlying database management system.*
- *Changing DBMS requires changing only the DAM classes*

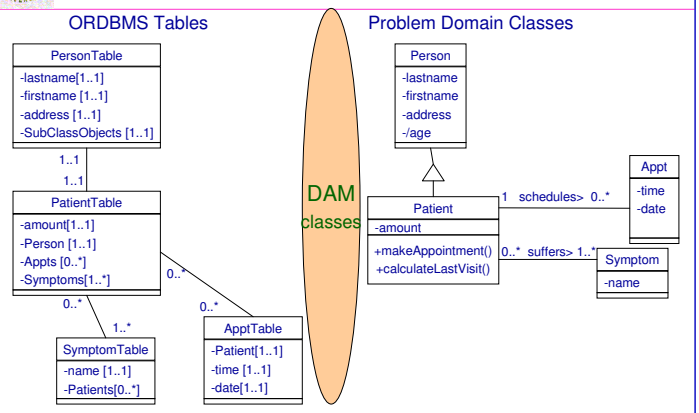
Example: Mapping problem domain objects to ORDBMS



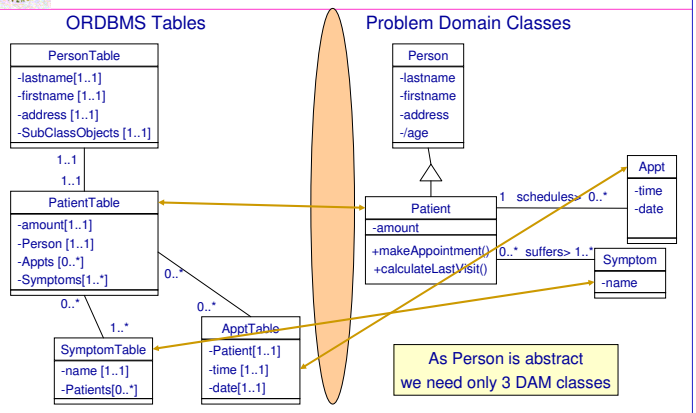
Assuming that the ORDBMS supports Object Ids, multi-valued attributes, stored procedures and no support of inheritance.

Oracle 8i does not directly support inheritance using the UNDER clause as in SQL:1999

Example: Mapping problem domain objects to ORDBMS> DAM Classes

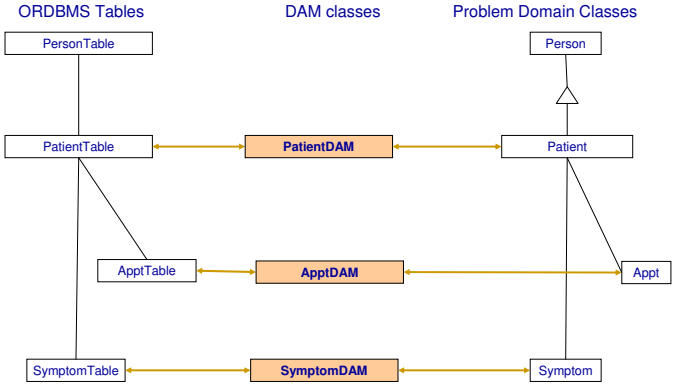


3 DAM classes

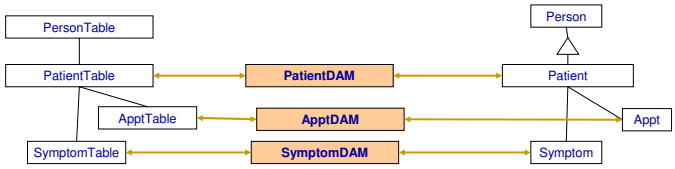


As Person is abstract we need only 3 DAM classes

3 DAM classes



PatientDAM



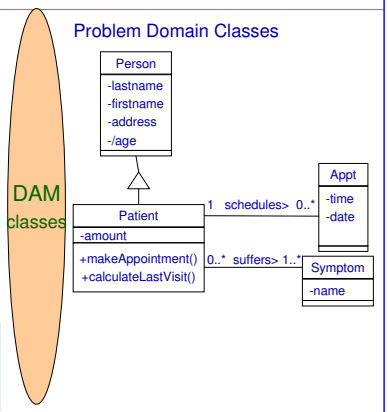
- The process to create an instance of the Patient problem domain class can be complicated:
- the PatientDAM may have to be able to retrieve information from all four ORDBMS tables.
  - Patient-DAM class retrieves the information from the Patient table.
  - Using the OIDs stored in the attribute values associated with Person, Appts and Symptoms attributes, its retrieves the the remaining information required for creating an instance of Patient

Case: Mapping problem domain objects to RDBMS

- Here the DAM classes are more complicated
- The relational schema comprises 5 tables.
- So the number of dependencies between from DAM classes to the databases tables has increased.

PersonTable(pId,lastname,firstname,address)
Patient(pid,amount)
ApptTable(apId, pId, time,date)
Symptom(sId,name)
Suffers(pid,sid)

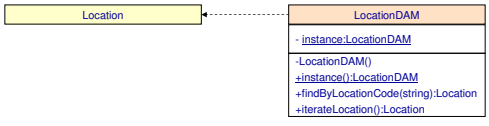
To create an instance of the Patient problem domain class, the PatientDAM must query (join) all tables.



Designing DAM classes in more detail

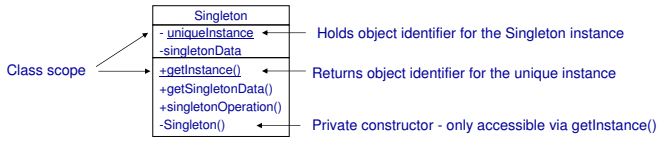
Designing the LocationDAM class

- In order to ensure that for each DAM class there will be only one instance, we can use the Singleton pattern.
- This means that we use a class-scope operation but only to obtain an instance of the DAM class that can be used to subsequently access the database.

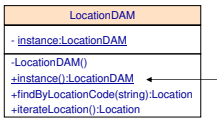


(Pattern: Singleton)

Singleton Pattern: One and only one instance of the class can exist.



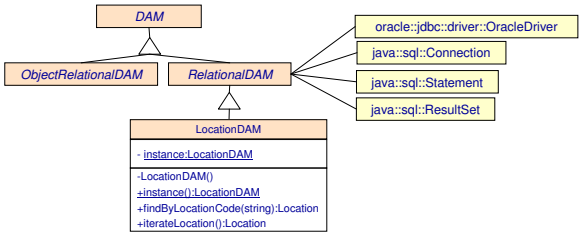
In our case:




```
public LocationDAM instance() {
  if (instance==null) {
        instance = new LocationDAM()
  }
 return instance;
}
```

Designing the `LocationDAM` class

- As each persistent class in the system will require a DAM class, it makes sense to create a superclass that provides the services required by all the DAM classes.
- We could have 2 levels of generalization
  - At the top is an abstract class "DAM" that provides the operation to materialize an object using its object identifier.
  - This class is then specialized to provide different abstract classes of brokers for different kinds of storage.



Proxies and Caches

Problems that remain to be resolved

- *what happens if a loaded object has to send a message that has not been retrieved from the database?*
- *How to handle transactions where a number of objects are created, retrieved from the database, updated and deleted?*

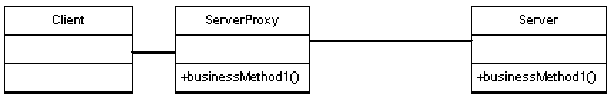
We can handle these problems by extending the previous design with

- **proxies**
  - use proxy objects for those that have not yet been retrieved from the database
  - The Proxy Pattern provides a proxy object as a placeholder for another object until it is required.
- **caches**
  - caches hold objects in memory and keep track of which has been created, updated or deleted.

(Structural Patterns: Proxy)

A proxy object acts as a substitute for the actual object.

- Example: Remote object interaction protocols. When an object needs to interact with a remote object (across a network) the most preferred way of encapsulating and hiding the interaction mechanism is by using a proxy object that mediates communication between the requesting object and the remote object.

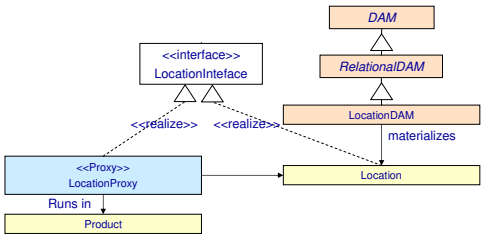


Proxies: In our case

- Suppose that an object of the class Product is already loaded in memory.
- If no message is sent to the associated location objects, then the proxy does nothing
- If a message is sent (e.g. myProduct.locations[1].printAddress()) , then the proxy asks the relevant DAM class to retrieve the object from the db (the proxy just knowns the oid of the real object), and once it has been materialized, the proxy can pass the message directly to it. Subsequently messages can be sent directly to the object by the proxy.

Proxies

- The proxy class must also implement the same interface as the real class so that it appears to other objects as if it is the real thing.



Caches

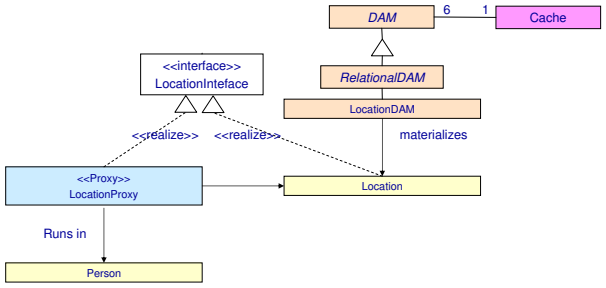
The DAM superclass can maintain one or more caches of objects that have been retrieved from the database. Each cache can be implemented as a hashtable, using the object identifier as the key.

- We can use 1 cache or 6 caches:
  - new clean cache: newly created objects,
  - new dirty cache: newly created objects that have been modified,
  - new deleted cache: newly created objects that have been deleted,
  - old clean cache: objects retrieved from the database,
  - old dirty cache: retrieved objects that have been modified,
  - old delete cache: retrieved objects that have been deleted.
- As objects are changed, the DAM superclass must be notified so that it can move them from one cache to the other.
  - This can be achieved using the *Observer-Observable pattern*: the object implements Observable, and the broker inherits from Observer.

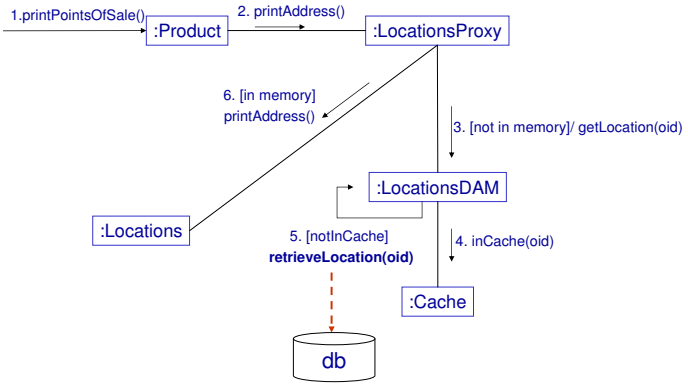
Caches (2)

- When the transaction is complete, the DAM superclass can be notified. If the transaction is to be committed, the DAM superclass can process each object according to which cache it is in:
  - new clean cache: newly created objects => write to the db
  - new dirty cache: newly created objects that have been modified => write to the db
  - new deleted cache: newly created objects that have been deleted => delete from the cache
  - old clean cache: objects retrieved from the database, => delete from the cache
  - old dirty cache: retrieved objects that have been modified, => write to the db
  - old delete cache: retrieved objects that have been deleted. => delete from the db
- The cache or caches can be used by the proxy object to check whether an object is already available in memory. When it receives a message, the proxy can ask the DAM superclass for the object, if it is in a cache, the DAM superclass will return a reference to it directly, if it is not in cache, the DAM superclass will retrieve it.

Proxies and Caches



Sketch of the communication diagram for loading an object (assuming proxies and caches)



Using a Data Management Product or Framework

There are products frameworks that provide a persistence mechanism

- Webgain Toplink Foundation Library jor Java is a product that will take classes and map their attributes to columns in relational database tables. It can either map attributes to columns in existing tables, or it can generate the schema for the necessary tables from a class definition. It also provides Java classes to provide the persistence mechanism. There are versions that work with applications servers
- CocoBase (from Thought Inc)
- JDO (Java Data Objects)
- Application servers

Designing Business Transactions (outline)

- Short Transactions
  - Pessimistic concurrency control
    - Levels of isolation
    - Automatic recovery
  - Programmable recovery
    - Savepoint
    - Trigger rollback
  - Designing stored procedures and triggers
- Long transactions

Designing Business Transactions

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

Pessimistic concurrency control

- **Locks** are acquired on every persistent object that a transaction processes.
- Types of locks:
  - **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**).

Levels of Isolation (II)

- The level of isolation may differ between transactions in the same application.
  - The SQL statement `set transaction` can be used for that purpose
- Increasing the level of isolation reduces the overall concurrency of the system
- In every case, the beginning of the transaction must always be delayed to the last second.
  - E.g. suppose a form that allow users to register to a service. They should first fill in all the fields of the form. We should begin the transaction at the end (when the user presses the submit button).

Automatic recovery

Many things may go wrong

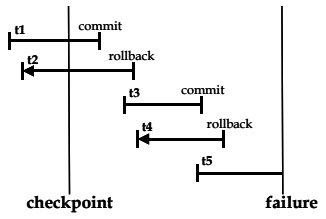
- power supply shutdown
- disk head crash
- running processes can hang or be aborted

- DBMSs provide automatic recovery for most situations.
- Depending on the state of the transaction at failure point, a DBMS will automatically perform a **rollback** or **rollforward** of the transaction as soon as the problem has been eliminated.

- The database administrator can control the amount of recovery time by setting the frequency of **checkpoint**.
- A checkpoint forces the DBMS to stop all transactions temporarily and write all the transactional changes (made since the previous checkpoint) to the database.

Automatic recovery

- Depending on the state of the transaction at failure point, a DBMS will automatically perform a **rollback** or **rollforward** of the transaction as soon as the problem has been eliminated.



Recovery after failure:

t1 - rollforward (redo)

t2 - rollback

t3 - rollforward

t4 - rollback

t5 - no action

Short and Long Transactions

- Most conventional IS applications require **short transactions**.
  - A short transaction contains one or more SQL statements and must be completed as quickly as possible so that other transactions are not held up.
  - Example application: airline reservation systems
- Some new classes of IS applications encourage cooperation between users and require **long transactions**
  - A **long transaction** can span computer sessions (users can take breaks then continue working in the same long transaction after returning)
  - Example applications: CSCW (computer-supported cooperative work)
  - Users work in their own **workspaces** using personal databases of data **checked-out** (copied) from the common workgroup database
  - A **long transaction** is not allowed to be automatically rolled back (the users would lost their work)
  - **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

Reading and References

- **Systems Analysis and Design with UML Version 2.0** (2nd edition) by A. Dennis, B. Haley Wixom, D. Tegarden, Wiley, 2005. Chapter 11
- **Requirements Analysis and System Design** (2nd edition) by Leszek A. Maciaszek, Addison Wesley, 2005, Chapter 8
- Dare Obasanjo, An Exploration of Object Oriented Database Management System, 2001
- S. D. Urban et al. , "Using UML Class Diagrams for a Comparative Analysis of Relational, Object-Oriented, and Object-Relational Database Mappings", SIGCSE'2003
- **Patterns of Enterprise Application Architecture**, Martin Fowler, Addison-Wesley, 2003
- **Modern Systems Analysis & Design** (4th Edition) by Jeffrey A. Hoffer, Joef F. George, Joseph S. Valacich, Prentice Hall, 2005, Chapter 10
- **Object-Oriented Systems Analysis and Design Using UML** (2nd edition) by S. Bennett, S. McRobb, R. Farmer, McGraw Hil, 2002, Chapter 18