

Object-Oriented DBMS Concepts

Topics

- Background
- Basic OO concepts
 - object, attribute, OID, class, method, encapsulation, class hierarchy, single/multiple inheritance, extensibility, complex object, overloading, overriding, polymorphism, user-defined type
- Query language in Object-Relational DBMS
- OO data model vs other data models
- Some problems in OO data model
- Inheritance conflicts in OO systems
- OO schema design
- Some reading materials (optional)

Background

- **Relational DBMSs** support a small, fixed collection of data types (e.g. integer, dates, string, etc.) which has proven adequate for traditional application domains such as administrative and business data processing. RDBMSs support very high-level queries, query optimization, transactions, backup and crash recovery, etc.
- However, many other application domains need complex kinds of data such as CAD/CAM, multimedia repositories, and document management. To support such applications, DBMSs must support **complex data types**.
- **Object-oriented** strongly influenced efforts to enhance database support for complex data and led to the development of **object-database systems**.

Object-database systems have developed along **two distinct paths**:

(1) Object-Oriented Database Systems. The approach is heavily influenced by OO programming languages and can be understood as an attempt to add DBMS functionality to a programming language environment.

- The Object Database Management Group (**ODMG**) has developed a standard Object Data Model (**ODM**) and Object Query Language (**OQL**), which are the equivalent of the SQL standard for relational database systems.

(2) Object-Relational Database Systems. ORDB systems can be thought of as an attempt to extend relational database systems with the functionality necessary to support a broader class of application domains, provide a bridge between the relational and object-oriented paradigms. This approach attempts to get the best of both.

- The **SQL:1999** (also known as **SQL3**) standard extends SQL to incorporate support for ORDB systems
- RDDMS vendors, such as IBM, Informix, ORACLE have added ORDBMS functionality to their products.

- Object-oriented DBMS's failed because they did not offer the efficiencies of well-entrenched relational DBMS's.
- Object-relational extensions to relational DBMS's capture much of the advantages of OO, yet retain the relation as the fundamental attraction.

Basic OO Concepts

Object and Class

- A **conceptual entity** is anything that exists and can be distinctly identified.
E.g. a person, an employee, a car, a part
- In an OO system, all conceptual entities are modeled as **objects**.
- An object has **structural properties** defined by a finite set of **attributes** and **behavioural properties** defined by a finite set of **methods**.
- Each object is associated with a logical **non-reusable** and unique **object identifier (OID)**.
The OID of an object is **independent of the values** of its attributes.
- All objects with the same set of attributes and methods are grouped into a **class**, and form **instances** of that class.

- Classes are classified as **lexical classes** and **non-lexical classes**.

- A **lexical class** contains objects that can be directly represented by their values.

E.g. integer, string.

- A **non-lexical class** contains objects, each of which is represented by a set of attributes and methods.

- Instances of a non-lexical class are referred to by their **OIDs**.

E.g. PERSON, EMPLOYEE, PART are non-lexical classes.

- In some OO systems, a class is treated as an object also, and therefore processes its own attributes and methods. These properties are called **class attributes** and **class methods**.

(Similar to **static fields** or **class variables** in Java)

E.g. A class EMPLOYEE can have **class attributes** called NO_of_EMPLOYEES which holds a count of the number of employee instances in the class, and NEXT_ENO which holds the employee number of the next new employee.

The class EMPLOYEE can have a **class method** called **NEW** which is used to construct new instances of the class.

Attribute

The **domain** of an attribute of a non-lexical class A can be one of the following:

Case (a) a **lexical class** such as integer, string.
An attribute with this domain is called a **data-valued attribute**.

Case (b) a **non-lexical class** B. An attribute with this domain is called an **entity-valued attribute**.

- * Note the **recursive** nature of this definition.
- * There is an implicit binary relationship between attributes A and B.
- * The value of the attribute A is the OID of an instance of B, which must exist before it can be assigned to the attribute. This provides **referential integrity**.

- * A special case exists in which the class B is in fact A. This represents a **cyclic definition** in the OO model.

E.g. PART-SUBPART
COURSE-PREREQUISITE

- * In **ORION** (from **MCC** - Microelectronics and Computer Technology Corporation - http://en.wikipedia.org/wiki/Microelectronics_and_Computer_Technology_Corporation), the relationship between A and B can be given semantics such as **IS-PART-OF** in which case, A is a composite object comprising B.

ORION also supports the concept of an **existentially-dependent object**, in which the existence of the object depends on the existence of its parent object.
(Similar to **EX** and **ID** relationships in ER approach).

Case (c) a **set**, **set(E)**, where E is either a lexical class or a non-lexical class. An attribute with this domain is called a **set-valued attribute**.

- * If E is lexical, values from E are stored in the set.
- * If E is non-lexical, members of the set can either be an **instance of E or its subclasses**. In this case, the set comprises instances from possibly heterogeneous classes. Only **OID** of each instance is stored in the set.
- * In **O2** (from **O2 Technology**) both **sets** and **lists** are supported. Note that a set has no duplicates, but members of a list may be duplicated.

Case (d) a **query type** whose values range over the set of possible queries coded in a query language. An attribute with this domain is called a **query-valued attribute**.

- * The value of a query-valued attribute is the result of the query, which is a set of objects satisfying the query.
- * **POSTGRES** (from UC Berkeley, or call PostgreSQL - <http://en.wikipedia.org/wiki/PostgreSQL>) allows query-valued attributes.

Case (e) a **tuple type**. An attribute with this domain is called a **tuple-valued attribute**.

- * This represents an **aggregation of attributes** of the tuple type, which is treated as a composite attribute of A.
- * An attribute of the tuple type can be a data valued, entity-valued, set-valued, query-valued, or tuple-valued attribute.
- * The definition of attributes of non-lexical classes is **recursive**.

Users can define their own complex data types using the mentioned attribute types.

Method

- A method of an object is invoked by sending a **message** (which is normally the method name) to the object. Such a **message-passing** mechanism represents a **binary interaction** between the sender of the message and the recipient.
- A method's specification is represented by a **method signature**, which provides the method name and information on the types of the method's input parameters and its results.

The implementation of the method is separated from the specification. This provides some degrees of **data independence**.

- Methods play an important role in defining object semantics.

E.g. When an employee is fired, we need to delete the employee information from the employee file, delete the employee from the employee-project file, and insert the employee information into a history file, etc.

One method called “**Fire-employee**” can be defined that incorporates this sequence of actions.

E.g. CPF (Central Provident Fund) method for employees.

- In OO systems that support **strong encapsulation** (e.g. **ORION**, **SMALLTALK** - <http://en.wikipedia.org/wiki/Smalltalk>), the only interaction with an object is through the object’s methods.

The attributes are **not** directly accessible, but are instead retrieved/updated through respective **get/set** methods.

- In OO systems that support a **relaxed form of encapsulation**, attributes may be accessed directly. Some protection mechanisms are provided to restrict access to sensitive data such as “salary”.

E.g. O2 provides a mechanism to partition an object’s properties into **public** and **private**.

Example 1 This example provides definition of non-lexical classes **EMPLOYEE** and **DEPARTMENT** (using an O2-like notation).

```
add class DEPARTMENT
    type tuple(D#: string, Dname: string, Mgr: EMPLOYEE)

add class EMPLOYEE
    type tuple (E#: integer, Name: string, Salary: integer,
                Dept: DEPARTMENT,
                Address: tuple (street: string, city: string),
                Supervisor: EMPLOYEE,
                Supervisees: set (EMPLOYEE) )

add method compute_tax( ): integer in class EMPLOYEE

add method fire_employee( ): Boolean in class EMPLOYEE
```

Note: A **cyclic definition** exists in the “Supervisor” and “Supervisees” attributes. Redundancy exists. Data consistency checking is required when one of them is updated.

Class Hierarchy

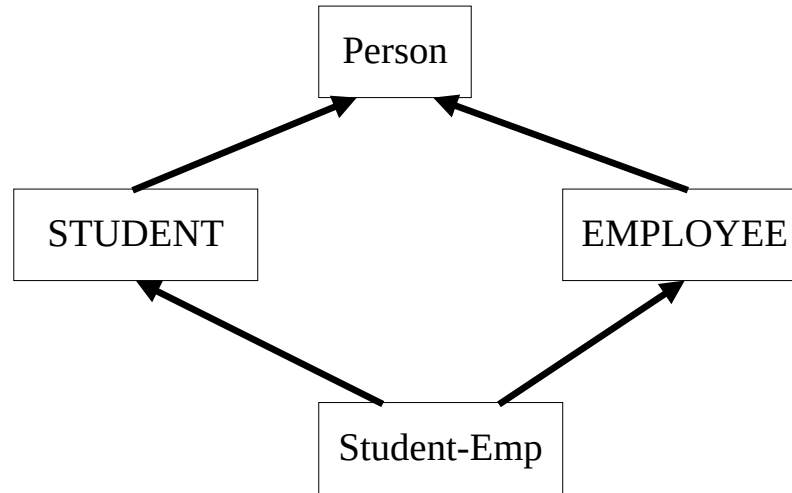
- Given 2 classes X and Y, **X ISA Y** means that each instance of X is also an instance of Y. We call X a **subclass** of Y and Y a **superclass** of X.

E.g. Manger isa Employee

- A class hierarchy provides an **inheritance mechanism** which allows a class to inherit properties (attributes and methods) from its superclasses.
- In **single inheritance** systems, a class can have **at most one** direct superclass and therefore can only inherit from that superclass.
The class hierarchy forms a **tree**.
- In **multiple inheritance** systems, a class can have **more than one** direct superclass.
The class hierarchy is a **lattice**.

Note: In multiple inheritance systems, a class may inherit properties and methods from different super classes and therefore may have **inheritance conflicts**.

Example 2. A multiple inheritance example.



Person is the **root** of the class hierarchy.

Student_Emp has **2** superclasses, STUDENT and EMPLOYEE.

Q: Does Java allow multiple inheritance?

Extensibility

- **Extensibility** is another important feature of the OO paradigm. It allows the creation of new data types, i.e. **user-defined types**, and operations on these new data types from built-in **atomic data types** and user defined data types using the **type constructor**.
- A **type constructor** is a mechanism for building new domains.
A **complex object** is built using type constructors such as **sets**, **tuples**, **lists** and nested combinations.
- A combination of an user-defined type and its associated methods is called an **abstract data type (ADT)**.

Extensibility (cont.)

- Hiding of ADT internals (implementation) is called **encapsulation**.
- Most OODBMSs, e.g., ORION, O2, IRIS support **data type extensibility**.

E.g. A new data type called POLYGON can be added to handle geometric objects. The user can define an operator AE (Area Equal) which allows two polygons to be compared for area equality. A method “draw” allows a polygon to be plotted.

E.g. One might define operations on an image data type (jpeg_image) such as compress, rotate, shrink, crop on an image, and overlay two images.

- In the OO paradigm, different classes may have methods with the same name.

E.g. Consider 3 classes, PERSON, EMPLOYEE and SECRETARY, and the ISA relationships:

```
EMPLOYEE  ISA  PERSON
SECRETARY ISA  EMPLOYEE
```

All 3 classes have a method called “Print”.

The implementation of “Print” in SECRETARY **redefines** and **overrides** the “Print” method in EMPLOYEE, which in turn, redefines and overrides the “Print” method in PERSON. All these 3 “Print” methods are different.

An **overloading** of the “Print” method has occurred.

- A feature related to the use of overloaded methods is **polymorphism**. Polymorphism is the ability of different objects to respond differently to the same message.

Example 3 Consider a linked list comprising objects from PERSON, EMPLOYEE and SECRETARY classes. A traversal of the linked list can be done so that each node in the list a ‘Print’ method is invoked. This C++ like program provides a piece of polymorphic code to perform the traversal.

```
void Print (Person *p) {  
    for (Person * ptr = p; ptr; ptr = ptr -> next)  
        ptr -> Print( );  
}
```

The type of the pointer ptr is resolved during runtime to be one of PERSON, EMPLOYEE or SECRETARY.

The appropriate “Print” method is then invoked depending on the type.

Resolution of the type of an object during runtime is referred to as **late binding**.

Query Language in Object-Relational DBMS

Example. from **SQL/X** of **UniSQL, Inc.** UniSQL is the earliest proposal to add object-oriented features into relational DBMS.

```
create class PERSON
```

```
    ( Name    CHAR(20),  
      Sex     CHAR(1),  
      BirthDay DATE )
```

```
METHOD
```

```
    Age ( ) INTEGER;
```

```
create class EMPLOYEE
```

```
    ( Job      CHAR(20),  
      Salary   FLOAT,  
      Hobby    SET-OF ACTIVITY,  
      WorksFor COMPANY )
```

```
METHOD
```

```
    CPF-CONTRIBUTION ( ) INTEGER
```

```
AS SUBCLASS OF PERSON;
```


create class **ACTIVITY**

```
( Name    CHAR(20),  
  NumPlayers  INTEGER);
```

create class **COMPANY**

```
( Name    CHAR(30),  
  Location CHAR(20),  
  Budget  FLOAT);
```

Query: Single class query

```
select  Name, Salary  
from    EMPLOYEE  
where   Job = "Engineer";
```

Note: Name is inherited from the **superclass** PERSON

Query: Two-class Join Query

```
select  EMPLOYEE.Name, Job, WorksFor.Name  
from    EMPLOYEE, COMPANY  
where   Employee.Name = Company.Name;
```

Note: Both Employee and COMPANY have a property called "Name".

Q: What is the meaning of this query?

Query: Path Query

```
select   Name, Job
from     EMPLOYEE
where    “Tennis” IN Hobby.Name AND
          WorksFor.Name = “NUS”;
```

Query: Query with Group By

```
select   Job, AVG(Salary)
from     EMPLOYEE
where    “Tennis” IN Hobby.Name
group by Job;
```

Query: Query with a nested subquery.

```
select   Name, Salary
from     EMPLOYEE
where    Salary >
          0.01 * ( select MIN (Budget)
                   from   COMPANY
                   where  Location = “Jurong” );
```

Query: Query against a class and all its subclasses

```
select  Name, BirthDay
from    ALL PERSON
where   Sex = "M";
```

The keyword **ALL** is used in order to find **all** subclasses of person.

Query: Query with method

```
select  Name, Hobby, Age
from    EMPLOYEE
where   Job = "Sales"  AND
        CPF-contribution > 500;
```

Note: Queries cannot involve **methods with side effects**. **Why?**

OO data model vs hierarchical data model

- The nested structure of **objects** and the nested structure of **records** in hierarchical databases are similar.
- The essential difference is that the OO data model uses **logical** and **non-reusable OIDs** to link related objects while the hierarchical model uses **physical reusable pointers** to physically link related records. Hierarchical model has no object and OID concepts.
- Another difference is that the OO data model allows **cyclic definition within object structures**.

E.g. a course can refer to other courses as its pre-requisite courses.

To support cyclic definition in the hierarchical data model, **dummy record types** (e.g. prerequisite record) are needed.

OO Data Model vs Nested Relations

- In the nested relation approach, an attribute of a relation can itself be a relation.

The nested relation **is stored physically** within the base relation.

This approach does **not** allow the nested relation to be **shared** among relations.

There may be a **redundant** storage of data which can lead to updating anomalies.

- In the OO approach, nested relations are simulated by using the **OIDs** of tuples of a relation that are to be nested within a base relation.

Because OIDs are used, **sharing** of tuples of nested relation is possible. There is **less redundancy**.

Question: Any redundancy? Yes! Why?

OODBMS vs OOPL

- There is a strong parallel between developments in OODBMS's and OO programming languages (OOPL's).
- Developments in OOPL's have taken one of the two approaches:
 - (a) Take an **existing PL** and **extends** it with OO constructs.
E.g. C++ and objective-C extend C
CLOS and LOOPS extend LISP
 - (b) Develops a **new OOPL**
E.g. Java, Smalltalk, Eiffel
- In the OODBMS community, 2 similar approaches:
 - (a) **Extend the relational DBMS** to incorporate OO concepts (i.e. object relational model).
E.g. POSTGRES, UNISQL, DB2, ORACLE, Informix, Microsoft SQL Server
 - (b) Develop a DBMS **around an OO data model**.
E.g. ORION, IRIS, O2

- Differences between OOPL's and OODBMS's
 - (a) OOPL's do not have the amenities of databases such as **data persistency** and **concurrency**.
E.g. an OOPL does not have inherent data persistence and cannot share data across multiple sessions, except through a programmer-manipulated file system.
 - (a) The type systems of OOPL's and OODBMS's differ. Database calls are **declarative** and operate on **a set at a time** basis, while an OOPL is **imperative** and suited for handling **a record at a time** processing.

Some problems and proposed solutions in Object-Oriented Data Models

1. General disagreement on OO concepts

- Several OO data models have been proposed that offer somewhat different interpretation of OO concepts. No common agreement for a long period.
- Wide diversity in implementation of the data models.

E.g. Gemstone (from Servio Logic) adopts the **object/message paradigm**, and Vbase (from Ontologic) uses an **abstract data type paradigm** to encapsulate data and operation

E.g. IRIS (from HP) uses a **functional approach** in which methods and attributes are modeled by mathematical functions and POSTGRES (from UC Berkeley) extends the relational model to support OO concepts.

- Despite this diversity, a core set of OO concepts is common across these data models, such as **object**, **attribute**, **method**, **class**, **class hierarchy**, **encapsulation**, and **polymorphism**.

2. Navigational Model of Computation

- The value of an attribute of an object may be an **OID (object identifier)** of another object, which in turn may reference another object, leading to a complex and nested structure. Wide diversity in implementation of the data models.
- The use of **explicit reference** is similar to the CODASYL approach - network model, which uses pointers.
- This **navigational component** causes several problems:

(a) Consider the schema and below 2 person objects:

(Ob1, < name: “John”, spouse: Ob2>)

(Ob2, < name: “Mary”, spouse: Ob1>)

where Ob1 and Ob2 are OIDs.

This example uses an **inverse relationship reference**. This causes the update problem, contradicts to the easy maintainability objective of the OO paradigm.

- (b) For a navigational interface, access to data is **hard-coded** and therefore does not enjoy the benefit of a **query optimizer**.
- (c) The navigational approach does not preserve **data independency** any better than the hierarchical or network model.
- An OODBMS can be augmented with a **declarative query language** to complement the navigational access.

3. Issues in Use of Methods

The message passing mechanism that is used to trigger methods in OO systems presents some problems.

(a) Message passing is **binary**.

n-ary ($n > 2$) relationships need to be redefined as binary relationships. This will result in information loss.

(b) The OID of the receiving object must be **known before** a message can be sent. Such OID may sometimes not be available.

E.g. Find the names of Employees who are younger than 30 and are male.

However, it is useful to have methods that provide better semantics for object behaviour,

E.g. fire-employee method

E.g. CPF computation for employee objects

Unlike attributes, it may not be possible to index methods.

Why?

4. Standard Declarative Query Language

- Unlike relational DBMS's which have adopted SQL as the de facto standard, no generally agreed upon standard declarative query language was available for OODBMS's for a long period.
- Object Data Management Group (ODMG)
<http://www.odbms.org/odmg> designed a query language **Object Query Language (OQL)** modeled after SQL as a query language standard for OODBMS. Because of its overall complexity no vendor has ever fully implemented the complete OQL.

- In OODBMS's that lack query language, methods need to be defined to handle queries. It faces at least 2 problems:
 - (a) It is impossible to pre-empt *all possible queries* and provide methods for them.
 - (b) Consider the following query on SPJ db:
Find all suppliers who supply at least one red part to more than one project.

This is not a trivial method to write.

This method cannot be defined as a method for supplier object. A more appropriate level is either declare it as a class method, or as a method at meta class level.

- A query language is needed to handle **complex queries**.
- **Optimizing queries** in the presence of arbitrary methods is a difficult issue.
- IRIS only allows methods without **side-effects** to participate in queries.

POSTGRES restricts methods to contain only data manipulation commands that can be optimized.

Other systems do **not** permit **methods** in queries.

5. Access Methods and Data Type Extensibility

- Most OODBMS's such as ORION, O2, IRIS allow **new data types** (i.e. **user-defined types**) to be added.

E.g. Add a new type POLYGON together with an operator called AE (Area Equal) and a method 'draw'.

- Conventional DBMS's already provide standard access methods (e.g. **B-trees**, **hash tables**, etc.) to support an efficient database access. To provide efficient access instances of new, specialized user-defined types, access methods beyond those provided by the DBMS's are required.
- Some proposed that users provide their own access methods to support their new data types. However, supporting these user-defined access methods is difficult. Query optimizers have problem to use user defined access methods for query optimization processing.

6. Support for complex objects

- Many applications need to define and manipulate a set of objects as a single logical complex entity.
- Complex objects can be built using **list**, **set**, **record** and **nested combinations** of these.
- Most OODBMS's e.g. O2, ORION, support complex objects.

In ORION, semantic relationships such as **IS-PART-OF** are assigned to inter-object references within complex objects.

- ORION also supports the concept of an **existentially-dependent object** (weak entity in ER approach), in which the existence of the object depends on the existence of its parent object.

The **deletion** of an object triggers a **cascading delete** of all objects that are existentially dependent on the deleted object. This adds to the integrity features of the ORION data model.

- While the use of complex objects has an important semantic value, the efficient retrieval of complex object (and its components) is still a difficult issue.
- Several techniques e.g. **clustering** & **indexing**, have been proposed to improve the performance of complex object retrieval for navigational based or query-based retrieval.
- **Clustering** is suitable when an object is navigated using inter-object references. In clustering, components of a complex objects are stored together on a physical transfer unit (e.g. page), and hence they can be retrieved efficiently.

However, any clustering of objects is **optimal** for one type of access to the objects, but **sub-optimal** for most other types of access. It is left to the users to specify a preferred clustering strategy.

- The idea of using **indexes** (of RDBMS's) has been extended to OODBMS's.

The notion of a **class hierarchy index** and **nested attribute index** have been proposed.

- A **class hierarchy index** is defined on an attribute of a class and instances that are indexed belong to the class and its subclasses, if any (e.g. ORION).

E.g. A query “Find all students who are 21 years old” on the class hierarchy in Example 2 (page 20) has the search condition that he or she should be a Student as well as the predicate that he or she should be 21 years old. Here, all TAs can also be considered Students by the ISA relationship imposed on the class hierarchy. Therefore, objects of all classes in the hierarchy rooted at Student should be searched for this example query. How to index the age attribute of the classes?

- A **nested attribute** is an attribute of a nested component object of a complex object. Queries on a complex object can be predicated on a nested attribute.

By defining an index on the nested attribute, the queries can be more efficiently supported. How?

7. Object Identity

There have been debates on the relative merit of supporting **OIDs** and user-defined **keys** (as in RDBMS's)

Some proposed that OIDs should be assigned if keys are not available.

Some believed that **OIDs are unnecessary** and undesirable because:

- (1) All keys e.g. SSN, E#, PART#, etc. are actually user created. In fact, all attributes are artificially created by users. Therefore, a key can always be artificially created by the user for an object class that does not possess one.
- (2) Keys are more natural and **human readable** comparable to OIDs, which are implementation specific (e.g. pointer-based OIDs).

(3) **Overhead** incurred by OIDs.

4 common implementation techniques of OIDs:

1. **Physical Address** (memory address or disk address)
2. **Structured Address** (logical page no. + record no. in the page)
3. **Untyped Surrogate key** (positive integer value)
4. **Typed surrogate key** (record type + positive integer value)

If **physical addresses** are used for implementing OIDs, reorganization of disk storage (e.g. remove deleted objects in order to improve disk usage and performance) may not be possible.

On the other hand, if **logical pointers** are used for implementing OIDs, one more table lookup access is required to access an object.

- (4) While values of keys can change because of changing conditions, such changes represent a conscious effort on the part of the user, and can be done in a controlled environment.

E.g. Change all 7 digit house phone numbers to 8 digit phone numbers by adding a leading digit 6. The changes can be done offline.

- (5) **Multi-databases**. In order to identify or find a real world object, **key value** (or some attribute value) is needed. Also a real world object which appears in two different databases, sure have different (system generated) OID values. To determine whether two objects from 2 different databases are referred to the same real world object, **key values** are needed.
- (6) **Object migration** problem. When an object moves to its superclass or subclass (e.g. an employee is promoted to manager position) whether the object's OID should be changed or not? How to implement OIDs in order to avoid changing of OIDs when objects migrate?

One approach: Implement OIDs by **untyped surrogate keys**. **Why?**

Q: Does Java allow object migration such as promotion of employees?

- (7) **Weak entity.** The semantics of a **weak entity** requires that it should be accessed in conjunction with its parent entity. However, the use of OIDs allows the weak entity (i.e. existentially dependent object) to be directly accessed. This weakens the semantics of weak entities.
- (8) **View object.** For RDB, we can create view relations (external relations). For example, we want to create a view relation called GoodStudent, to store all the good students who have CAP ≥ 4.50 . If we can create view objects, then how to design/create the OIDs for the view objects?

8. Should attributes be directly accessible?

There were debates on whether attributes of an object should be directly accessible.

- **Approach 1:** access to an object's attribute (e.g. Sex, DOB of a person) should be through the object's **(public) methods**. This approach shields applications from changes in the implementation of attributes and provide data independence. However, it appears trivial and redundant to generate public access methods (e.g. **get/set**) for attributes.
- **Approach 2:** subject to a separate authorization scheme, attributes of objects should be **directly accessible**. This is because a query language optimizer needs to access the object's values directly.

- Use methods to access object's attribute values generate unnecessary overhead. Approach one will have difficulty to write methods to answer queries such as:

E.g. Find all male employees who are older than 30 but younger than 50 and work for Sales or Personnel Department .
- However, e.g. the **CPF contribution** of an employee should be implemented as a method. Whenever the CPF contribution computation formula changes, only the method's implementation needs to be changed; applications that use this method are not affected.

Note that CPF contribution of an employee is not an attribute. CPF contribution rates can be found on

<http://mycpf.cpf.gov.sg/Members/Gen-Info/Con-Rates/ContriRA>

Note: Java provides different types of access control using field (attributes) modifiers such as **public**, **protected**, **no modifier**, and **private**.

The following slides discuss some **OO data modeling issues** which can be resolved by applying concepts and techniques from ER data modeling.

9. Everything as Objects?

- SMALLTALK has “successfully” demonstrated the usefulness of a consistent treatment of everything as objects in a programming environment.
- It may be less useful to treat everything as objects in a database environment.
- In database design, it is important to **distinguish** among **attributes**, **entities**, and **relationships**.

10. Formal Foundation for OO paradigm

- RDBS have the relational model, which has a mathematical basis in **first order logic**. **Normalization** and **data dependency** theories can be applied to a RDB schema to determine its quality.
- Initially, no equivalent theories (e.g. FD, MVD theories and normalization) were available for OO database design, it is difficult to judge whether an OO schema is ‘good’.
- [1] extended ER Diagram with methods, called **OOER diagram**, to represent OO schemas. **Normal form OOER diagram** was proposed to determine quality of an OO schema.

[1] Tok Wang Ling, Pit Koon Teo: A Normal Form Object-Oriented Entity Relationship Diagram. ER 1994: 241-258, 1993

11. Lack of support for explicit relationships

- Most OO data models (e.g. O2, ORION) use **inter-object references** (using OIDs) and the **class hierarchy** to support relationship among objects.
Inter-object references provide only **implicit binary relationship** between 2 objects.
- Using this approach, the modeling of **m:m**, **n-ary** and **recursive relationships** are problematic and introduce problems similar to those faced by hierarchical and network models.

Note: Object Definition Language (**ODL**) from Object Data Management Group (**ODMG**) allows user to define the **reverse relationship** of a m:m binary relationship in a class.

- The class hierarchy allows object classes that are related by ISA relationship to be organized into a hierarchy. However, special relationship types such as **UNION**, **INTERSECTION**, **DECOMPOSE**, etc. are not supported.

Several problems arise from this.

(a) *Nested relations*

Consider the **nested relation**

DEPT (D#, Dname, **EMP** (E#, Name, Sex))

in which EMP attribute is itself a relation.

Such a nested relation imposes a strictly hierarchical structure which does **not** facilitate **symmetric queries**.

- * In OODBMS's, there are at least 2 approaches to support the DEPT nested relation.
 - (i) **Approach 1:** Treat the EMP attribute in DEPT as a **multivalued attribute**, as its value, a list of OIDs that identifies the employee working in the department. This approach is adopted in O2 and allows object sharing. However, it is difficult to handle symmetric queries.

(ii) **Approach 2:** This approach is adopted in POSTGRES. It allows the **value of an attribute** in a relation to be a **relational query**.

It assume that there exist 2 physical tables:

EMP(E#, Name, Sex, D#)

DEPT(D#, Dname, EMPS)

The **EMPS** attribute in DEPT can be defined to hold a **query** such as:

```
select  E#, Name, Sex
from    EMP
where   EMP.D# = D#;
```

The problem with this approach is that **update** on the EMPS attribute (which is a view) of DEPT must be translated to updates on the base EMP tables. This may not always be possible in general.

The performance may not be good.

Question: Where to store the query?

(b) *M:M, N-ary and Recursive Relationships*

Consider the SP database, in which S and P are related by an m:m relationship type SP.

- * One can store S, P, and SP as objects with (logical) pointers linking them.

ORION adopted this approach, which provides a navigational component that may be hard to maintain.

- * An alternative is to store P within S. This impose a hierarchical structure which cannot handle symmetric queries effectively.

It also introduces redundancy and leads to updating anomalies.

Recall: Object Definition Language (ODL) from Object Data Management Group (ODMG) allows user to define the reverse relationship of an m:m binary relationship in a class.

- * The above modeling problems are amplified when n-ary ($n > 2$) relationships are considered, e.g. SPJ database.
- * There is no feasible solution for modeling an n-ary relationship using inter-object binary references in the OO paradigm.

- * Some suggested that new “**relationship**” object classes must be created to represent ternary (and higher degree) relationships.
- * Similar problems occur for modeling **recursive relationships** such as course-prerequisite, part-subpart, etc.

One way to represent this in ORION and O2 is to define a set valued attribute called **pre-requisite** in a course class, with data type course also.

Deeper levels and **transitive closures** must be computed. The recursive nature is lost in this representation.

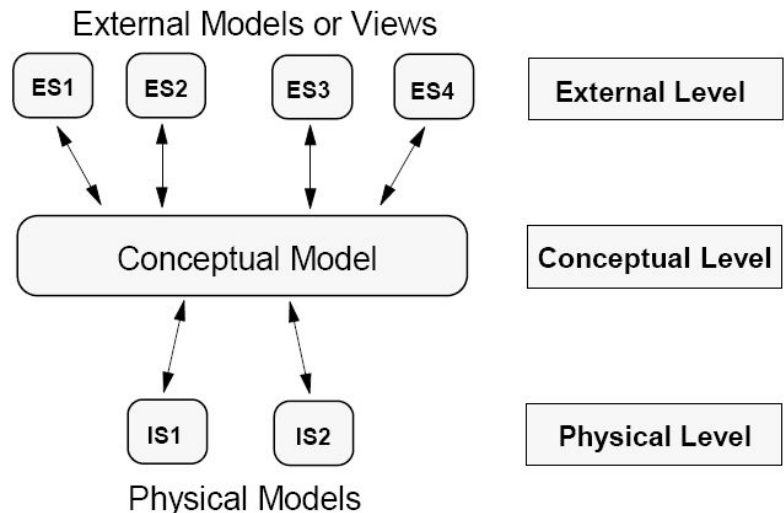
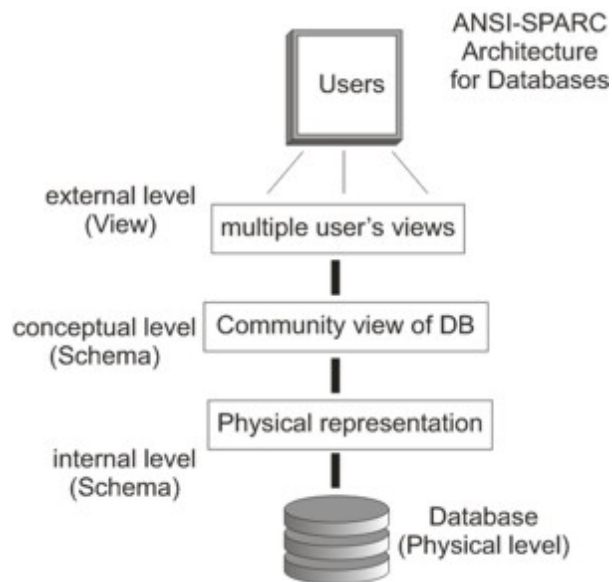
In some OODBMS's, the query language is enhanced with syntactic constructs to support the computation of the transitive closure of a recursive relation.

E.g. POSTGRES has a transitive operator “*****”.

12. Lack of General View Support

- Except for those OODBMS's that are based on the extended relational model (e.g. POSTGRES), most OODBMS's do not fit into the **3-level schema architecture framework** as spelled out in **ANSI/X3/SPARC proposal** (American National Standards Institute, Standards Planning And Requirements Committee) in 1975.

Note: The **ANSI-SPARC model** however never became a formal standard.



The 3 level of schemas are:

- **External Level (User Views)** : A user's view of the database describes a part of the database that is relevant to a particular user. It excludes irrelevant data as well as data which the user is not authorised to access.
- **Conceptual Level** : The conceptual level is a way of describing what data is stored within the whole database and how the data is inter-related. The conceptual level does not specify how the data is physically stored.
- **Internal Level** : The internal level involves how the database is physically represented on the computer system. It describes how the data is actually stored in the database and on the computer hardware.
- Users of most OODBMS's are often presented with a large-grained conceptual schema, with little or no facility for defining views.

- Several proposals have been made to incorporate **views** in OODBMS's, but most of the proposals do not provide the same generality and flexibility of a declarative relational view mechanism.
 - (i) **Approach 1:** Some only allows to define **multiple views** to a class.

Joins of classes and selections on classes **are not allowed** for defining views. A view of a class contains a **subset** of methods and attributes of the class.

Question: What is the value of a view object's OID?
 - (ii) **Approach 2:** Use a **query based view mechanism** to derive subclasses from superclass.

Such views are not updatable or updates apply only to non-recursive views that are based on a join of the primary key of the base tables.

They cannot handle other kinds of relationships, such as m:m, n-ary relationships.

(iii) **Approach 3:** An OO schema is represented by an OOER (schema) diagram. Mapping rules are proposed to generate external schemas (i.e. views) from the OOER diagram. Views of the OO schema are represented as views of the OOER diagram.

13. Conflicts in Class Hierarchy and Multiple Inheritance

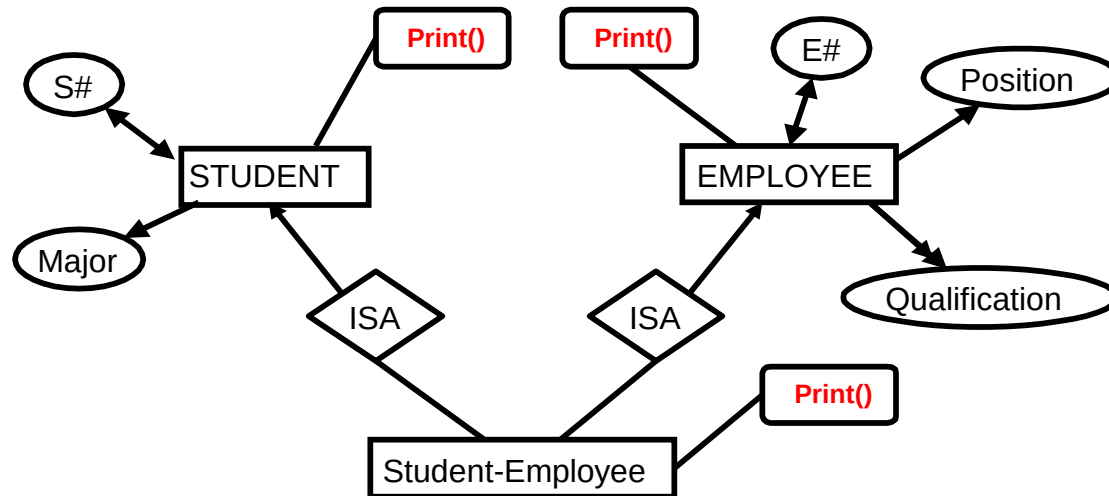
- There may have attribute and/or method **name conflicts** among a class and its superclasses.
- Details in the following slides.

Inheritance Conflicts in OO Systems

- In the OO paradigm, classes related through the ISA relationship are organized into a **class hierarchy**.
- There may have attribute and/or method **name conflicts** among a class and its superclasses.
- A class **inherits properties** (attributes and methods) from its superclasses in the class hierarchy.
- When a class inherits several commonly named properties of its superclasses, a **conflict situation** occurs which is resolved differently in different OO systems.

Ref: Tok Wang Ling, Pit Koon Teo: Inheritance Conflicts in Object-Oriented Systems.
DEXA 1993: 189-200

Example



- “**Print**” is a method in **EMPLOYEE** that display information such as **E#**, **Position**, and **Qualifications** of an **EMPLOYEE** object. **STUDENT** has a similarly named method “**Print**” which displays information such as **S#** and **Major** of a **STUDENT** object.

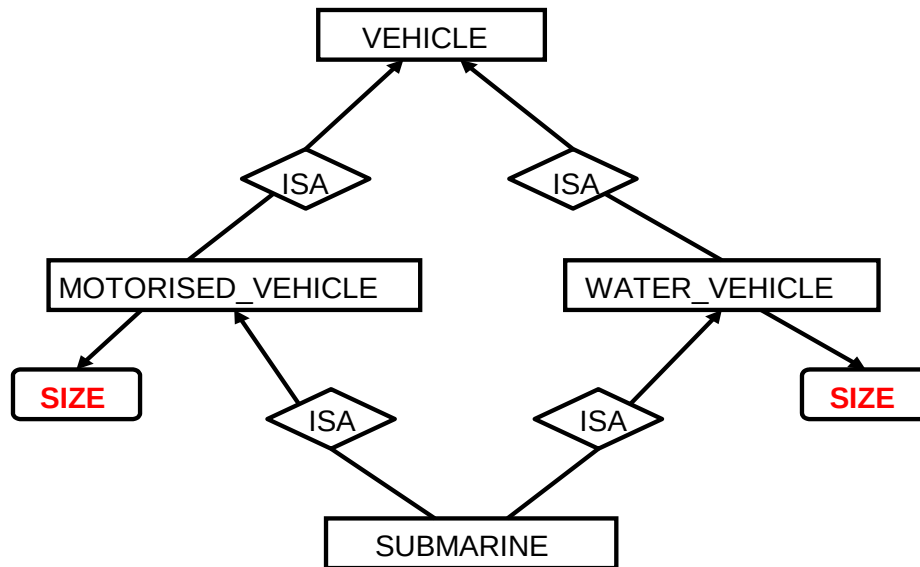
Note: We append “()” to a string to denote the string is the name of a method. Method is represented by a round rectangle in the OOER diagram.

- The semantics of “**Print**” in these 2 classes are different.

- The subclass Student-Employee can define a similarly named method “Print” which has a different semantics from the 2 “Print” methods of its superclasses.
“Print” is an **overloaded method**.
- The use of a common method name in a class hierarchy allows the exploitation of the notion of **polymorphism**, i.e. the ability of different objects to response differently to the same message (method name).
- There is **no conflict**.

1. Motivating Example

- In Figure 1, the class SUBMARINE needs to determine “**SIZE**” attribute to inherit from its 2 direct superclasses, i.e..
MOTORISED_VEHICLE and WATER_VEHICLE.



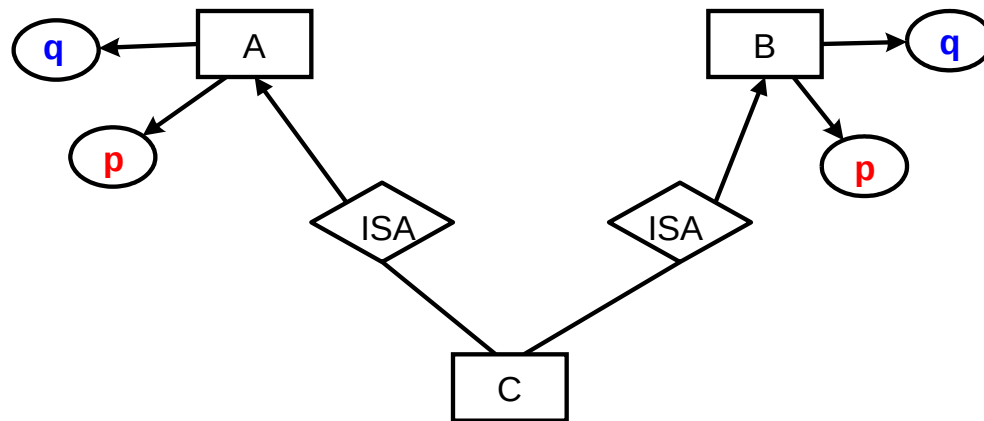
Q: Which **SIZE** to be inherited by SUBMARINE?

Fig. 1. Motivating Example

- Several resolution techniques have been proposed for OODBMS's to handle conflicts in **multiple inheritance** situations.

(i) The method used in **ORION** is to choose the **first** in the list of superclasses.

* This approach is somewhat arbitrary and may not yield the required semantics



Q: We want C to inherit p from A and q from B. How to express these 2 requirements in ORION?

(ii) **POSTGRES** does **not** allow the creation of a subclass that inherits conflicting attributes.

* This approach is not flexible.

(iii) **O2** allows the explicit selection of the properties to inherit by specifying the **inheritance path**.

(iv) **IRIS**: the property of the **most specific class** is chosen. If a single most specific property cannot be found, user specified rules will apply.

2. A Model of Inheritance

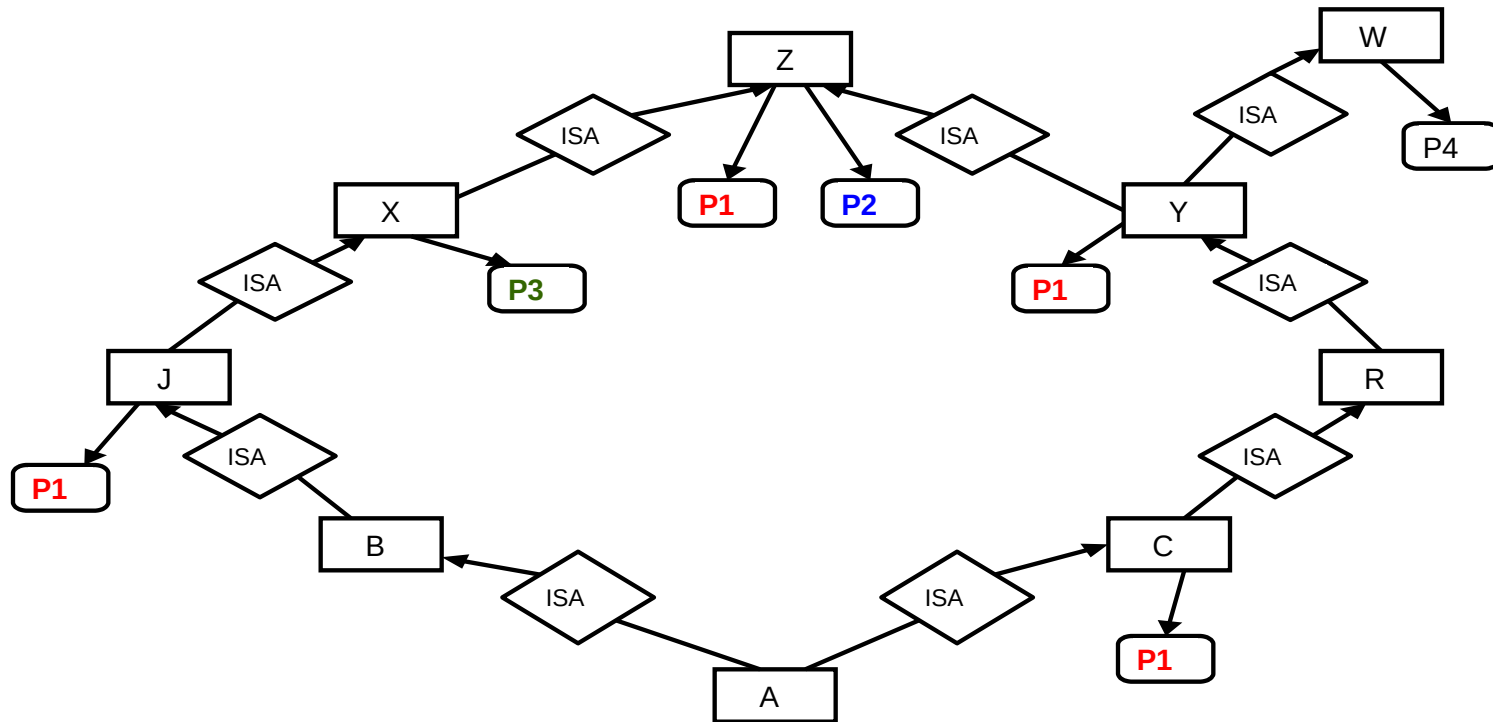


Fig. 2. An Inheritance Diagram

- A property is **specified** in a class if it is either **defined** or **redefined** for the class.
- A redefined property overloads a similar property in some superclass(es) of the class.
- An inherited property is **well defined** if it is specified in one and only one superclass, possibly indirect.
- A **conflict** situation exists when an inherited property is not well-defined, i.e., 2 or more superclasses specify the same property.

E.g. In Fig. 2,

- * Property p1 is **redefined** in classes Y, J, and C.
- * Class B **inherits** p1 from class J, and p2, p3 from classes Z, X.
- * P1 contributes to a **conflict situation** in class A, but p2 is **well-defined** in class A.

Most OODBMS's consider p2 in class A as a conflict situation.

3. Conflict Resolution Algorithm

Given an OO schema with ISA hierarchies

FOR each conflict situation in the hierarchy DO

IF it is a single-inheritance situation THEN /* **Case I:** SI (section 5.1) */
adopt precedence rule that prefers subclass properties, and ensure semantics is understood

IF it is a multiple-inheritance situation THEN

/* Check for ISA redundancy arising from ISA transitivity property */

IF conflicts arises because of ISA redundancy THEN

/* **Case II:** MI with ISA Redundancy (Section 5.2) */

resolve conflict by removing ISA redundancy

ELSE

BEGIN

Let the MI conflict situation be classes A, B1, ..., Bn ($n > 1$) where B1, ..., Bn are the nearest superclasses of A that specify a property p.

/* Note that a superclass of some Bi may itself specify a property p. */

/* Check the semantics of p in B1, ..., Bn */

IF semantics of p is the same in B1, ..., Bn THEN

BEGIN

IF intersection of B1, ..., Bn is empty THEN

/* **Case III:** MI-same semantics (Empty Subclass) (Section 5.3) */

Design error, since class A (which is, in fact, the intersection of B1, ..., Bn) is empty

```

ELSE /* Case IV: MI-same semantics (Factoring) (Section 5.4) */
  IF there exists a more general class K which is UNION of B1, ..., Bn THEN
    Factor p to class K /* see section 5.4 for explanation */
  ELSE
    Resolve the conflict by either:

    (a) creating a general class K that is the UNION of B1, ..., Bn and factoring p to
        K. Add new ISA relationships Bi ISA K for i = 1, ..., n. For each maximal
        superclass Ci of Bi such that K is a superset of Ci, add the ISA relationship
        Ci ISA K and remove the redundant ISA relationship Bi ISA K.
        IF there exists a class Y such that Y is a minimal superset of K THEN
          Insert new ISA relationship K ISA Y.
        /* Option (a) removes data redundancy but may create some ISA
        redundancies which will be removed by applying Case II */

    OR

    (b) Explicitly choosing one superclass to inherit the property.
        /* data redundancy exists which must be managed */

  END

```

```

ELSE
  BEGIN      /* Case V: MI-properties with different semantics (Section 5.5) */
    Let G1, G2, ..., Gm be sets of mutually exclusive classes from B1, ..., Bn such
    that classes in a group share the same semantics for p. Resolve the conflict in A
    by adopting one of the following:
    (a) redefine p in class A, /* not a good solution: see Section 5.5 */ or
    (b) Rename p in Gj to, say, p_Gj for j = 1, ..., m to reflect their different
        semantics. To conform to the unique name assumption. Each p in the schema
        that has the same semantics as P_Gj must be renamed to p_Gj.
        FOR each group Gj (j = 1, ..., m) with 2 or more classes having property
        p_Gj DO
          /* An MI situation exists between class A and the classes in Gj; */
          /* p_Gj has the same semantics in the classes of Gj */
          Resolve the conflict in class A using the method described in class III and
          IV.
        ENDFOR
    END
  END
END
ENDFOR

```

Case 1 Single Inheritance Situation.

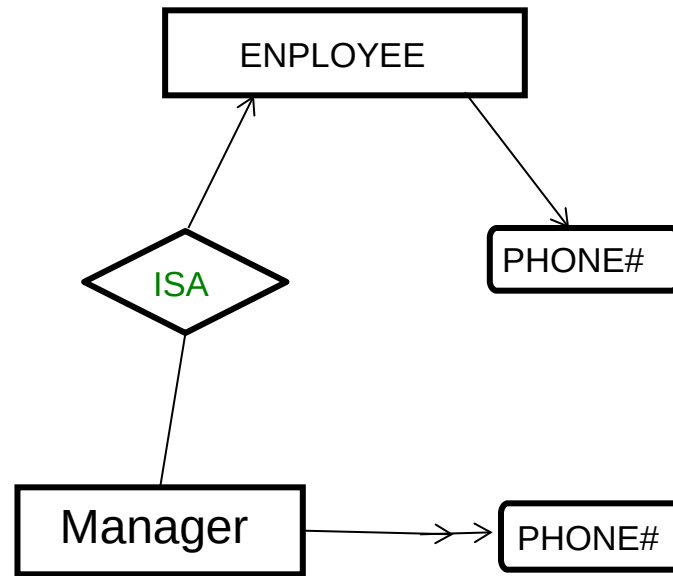


Fig. 3. PHONE# is overridden in MANGER and treated as multi-valued

- * From a conventional database design viewpoint, Fig. 3 is **erroneous**. However, OO approach allows this. Here MANAGER **overrides** the PHONE# of its superclass EMPLOYEE and **redefines** PHONE# as a multivalued attribute in MANAGER.

Case 2 Multiple inheritance with **ISA Redundancy**

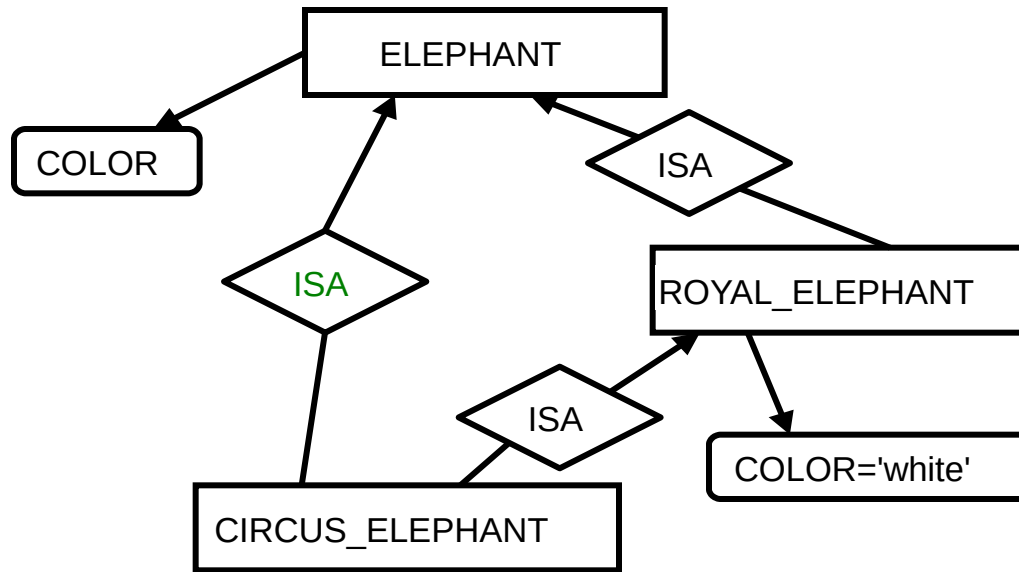


Fig 4. Removing Redundant ISA Relationship

- * The ISA link between ELEPHANT and CIRCUS_ELEPHANT is **redundant** and can be removed.

Case 3 Multiple Inheritance - Same Semantics (Empty Subclass)

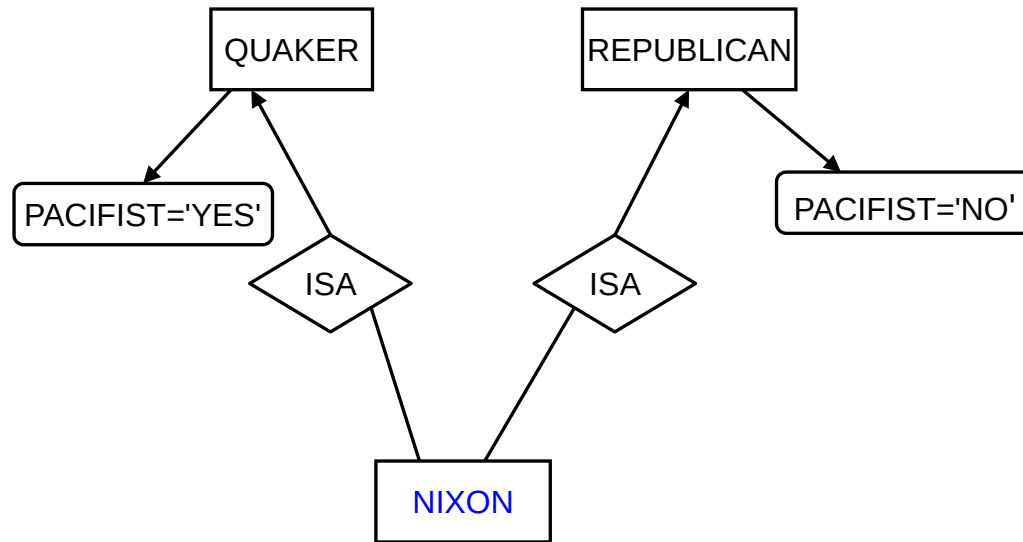


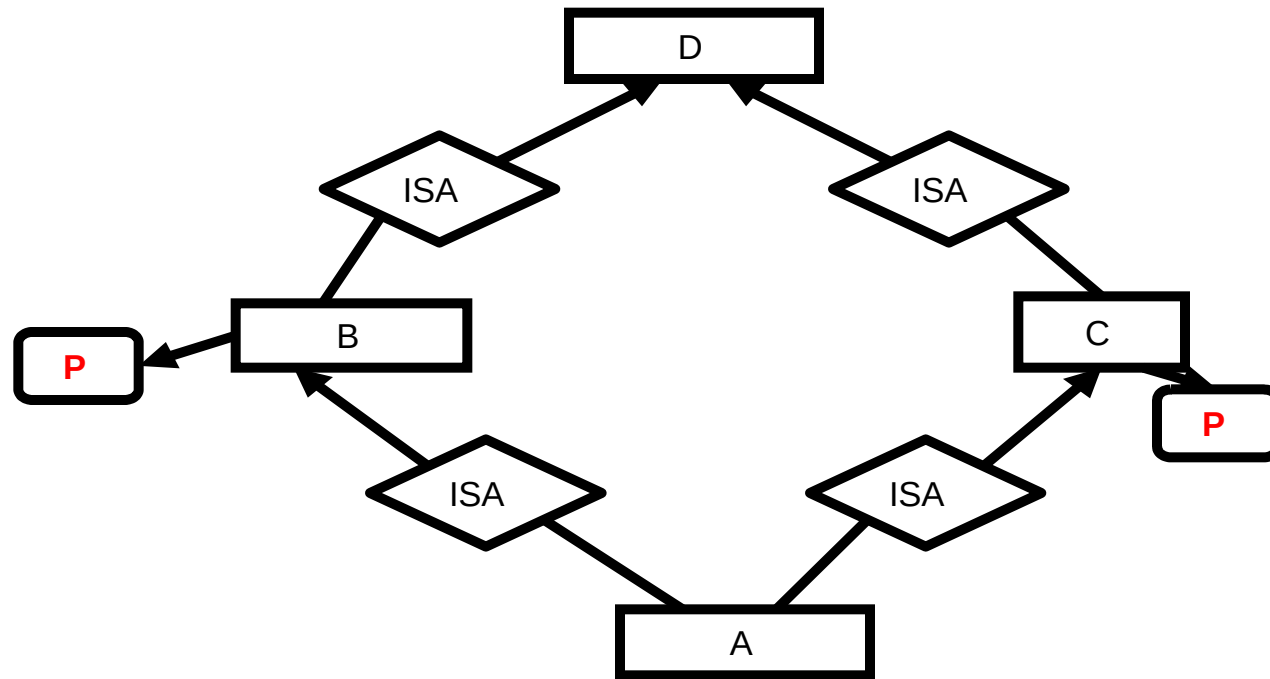
Fig. 5. ‘PACIFIST’ is overloaded and should be renamed to resolve conflict

- We assume that NIXON in Fig. 5 refers to a class of NIXON-like people.
- If the property PACIFIST has the **same semantics** in both QUAKER and REPUBLICAN, then there is clearly a **design error**.

A quaker is a member of the Society of Friends, a Christian religious group that meets without any formal ceremony or priests and that is opposed to violence.

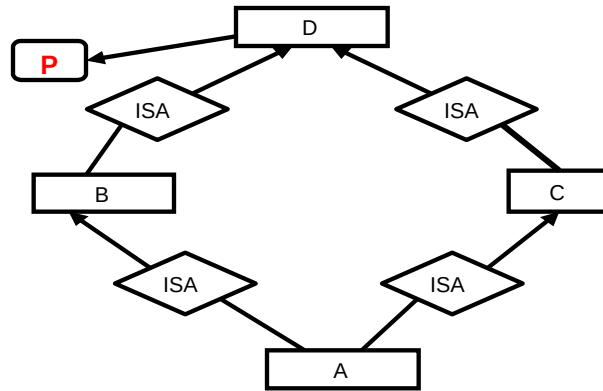
A pacifist is someone who believes that wars are wrong and who refuses to use violence.

Case 4 Multiple Inheritance - Same Semantic (Factoring)

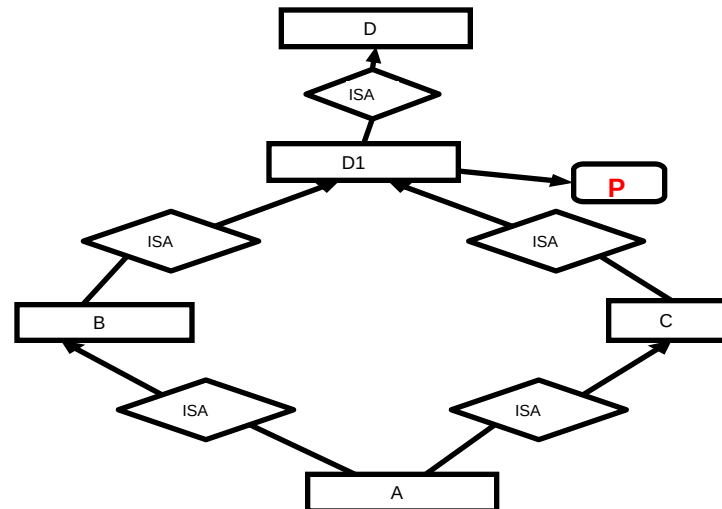


Both **P** in classes B and C are of same semantics

Case (4.a) If $D = B \cup C$, then factor P to D.



Case (4.b) IF $D \supset B \cup C$, then create D1 such that $D1 = B \cup C$, and factor P to D1.



Case 5 Multiple inheritance - Properties with different semantics

E.g. In Fig. 1, if the 2 **SIZE** are of different semantics, there are 2 options:

- (1) user can **redefine** or **overload** the property “**SIZE**” in SUBMARINE, e.g., explicitly mention

“SIZE” in SUBMARINE is “SIZE” in WATER_VEHICLE.

Problem: SUBMARINE can't inherit SIZE of MOTORISED_VEHICLE.

- (2) **Rename** the property “**SIZE**” in either MOTORISED_VEHICLE or WATER_VEHICLE or both.

Summary on inheritance conflict resolution approaches:

- renaming properties
- redefining (or overriding) an overloaded property
- removing redundant ISA relationship
- explicitly selecting an inheritance class
- redesigning the schema (e.g. factoring)

OO Schema Design

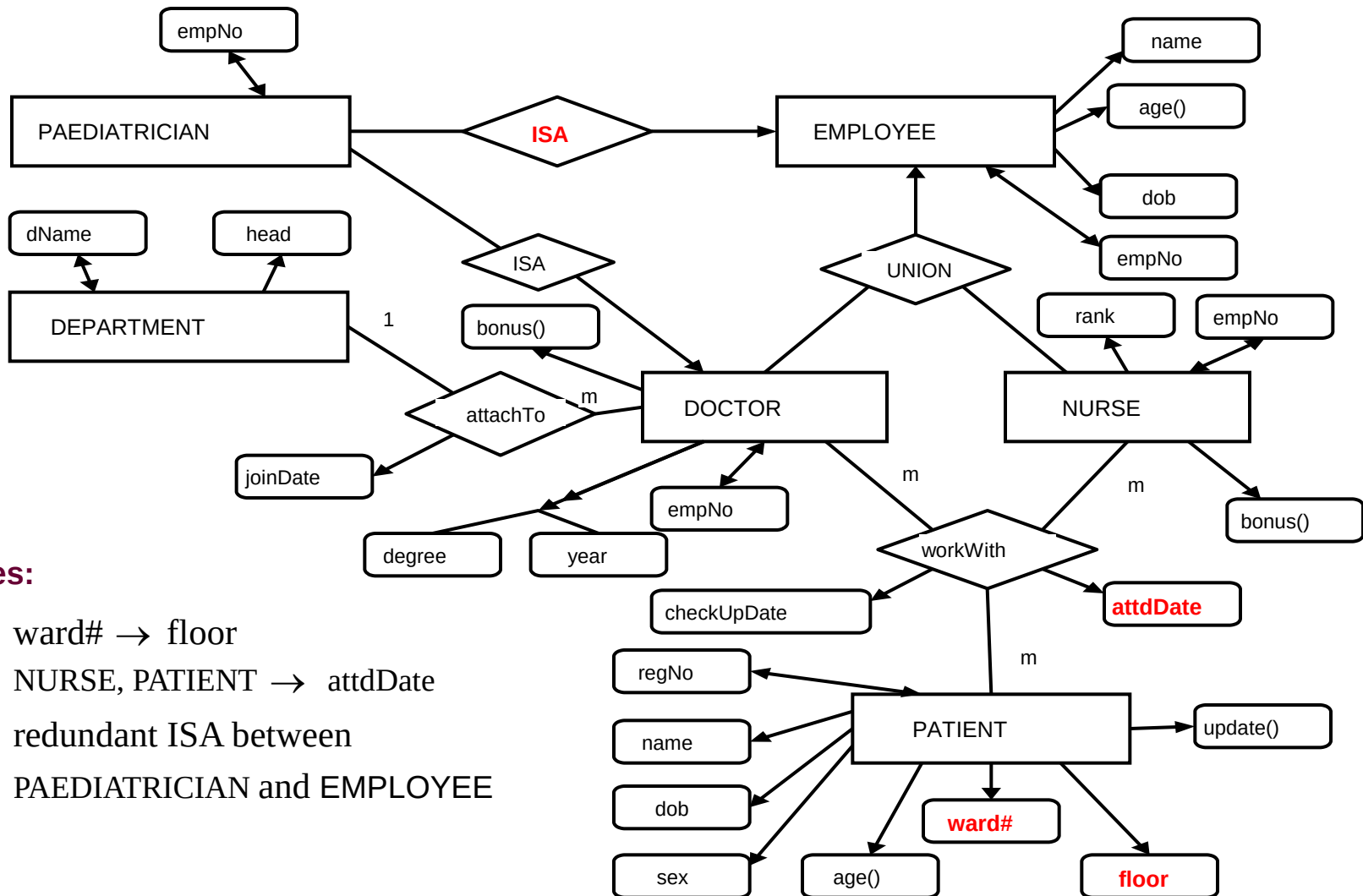
- Entity-Relationship Diagrams can be extended to support OO schema design.
- All the **structural properties** of the OO approach can be expressed in or derived from an ER diagram.

E.g. subclass-superclass relationship: ISA, UNION

composite object: IS-PART-OF

existentially dependent object: EX and ID dependent relationships

- **Methods** and **derived attributes** can be defined for both entity types and relationship types
- An ER diagram **augmented with methods** is called an **OOER diagram**.
- An OOER diagram is a **normal form OOER** diagram if its corresponding ER diagram is a **NF-ER diagram**, and there are **no inheritance conflicts** in its ISA hierarchies.

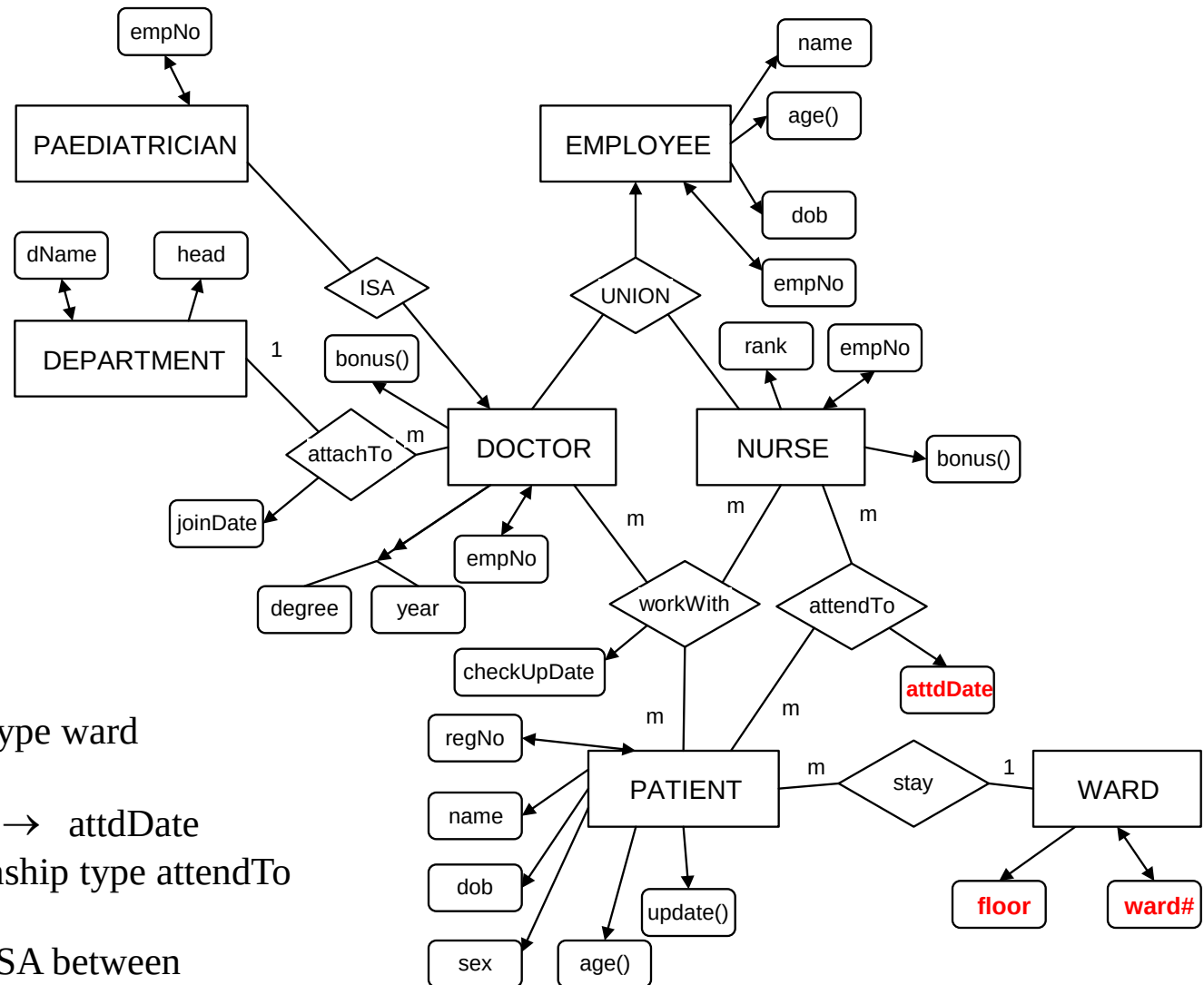


Notes:

- (1) ward# → floor
- (2) NURSE, PATIENT → attdDate
- (3) redundant ISA between PAEDIATRICIAN and EMPLOYEE

floor

Fig. 1. An OOER diagram



- (1) ward# → floor
 - create a new entity type ward
- (2) NURSE, PATIENT → attdDate
 - create a new relationship type attendTo
- (3) Remove redundant ISA between PAEDIATRICIAN and EMPLOYEE

Fig. 2. Normal Form OOER Diagram

Deriving Normal Form OOER Diagrams

Steps to convert an OOER diagram to a normal form OOER diagram:

Step 1 Ensure all property names within each entity type and relationship type are distinct and of different semantics. Ensure all key attributes are unique.

Step 2 Convert the ER diagram to normal form ER diagram.

Step 3 Remove any inheritance conflicts from ISA hierarchies.

Note: In step 1, we adopt the **relaxed universal relation assumption** mentioned earlier.

Generating OO Schemas

- Three approaches can be adopted.
- (1) **Approach 1.** The underlying OO data model **supports** the notion of **relationship directly**.
- Each entity type, m:m, n-ary or recursive **relationship type** can be mapped directly into a **class** in the OO schema.

E.g. The attendTo relationship type of Fig. 2.

```
class NURSE inherits EMPLOYEE type tuple
    (rank: string)
    method bonus(): integer;
end;
class attendTo type tuple
    (nurse: NURSE,
     patient: PATIENT,
     attdDate: integer)
end;
```

(2) **Approach 2.** The underlying OO data model **does not** support **relationship**.

- Each entity type is mapped into a class.
- Each relationship type is mapped into each of its participating entity type's object class using **inter-object references**.

Problem. Redundancies may occur.

However, **these redundancies are known and can be controlled**.

E.g. The relationship type **attendTo** and **workWith** in Fig. 2.

```
class NURSE inherits EMPLOYEE type tuple
  (rank: string,
   attendTo: set(tuple(patient: PATIENT, attDate: string)),
   workWith : set(tuple(doc : DOCTOR, patient : PATIENT,
                        checkUpDate : string)))
  method bonus() : integer
end;

class PATIENT type tuple
  (regNo : string,
   name : string,
   dob : string,
   sex : char,
   attendTo : set(tuple(nurse : NURSE, attDate : string)),
   workWith : set(tuple(doc : DOCTOR, nurse : NURSE,
                        checkUpDate : string)))
  method age() : integer,
        update()
end;
```

Note: FDs such as NURSE, PATIENT --> attDate, are not captured.

- (3) **Approach 3.** Treat each OO schema as a **view** of a normal form OOER diagram
- **Rules** for generating OO external views are needed.
Updatability of view objects needed to be determined.
 - Any **redundancies** in the external view is **virtual**.

E.g. An external schema of Fig. 2

```
class EMPLOYEE type tuple
    (empNo : string,
     name : string,
     dob : integer)
    method age() : integer
end;
class DOCTOR inherits EMPLOYEE type tuple
    (qual : set(tuple(year : string, degree : string)),
     Doc-Pat : set(PATIENT))
    method bonus() : integer
end;
class NURSE inherits EMPLOYEE type tuple
    (rank: string,
     Nurse_Pat : set(tuple(patient : PATIENT,
                           attdDate : integer))) /* via attendTo */
    method bonus() : integer
end;
class PATIENT type tuple
    (regNo : string,
     name : string,
     dob : string,
     sex : char,
     Pat-Doc : set(DOCTOR),
     Pat-Nurse : set(NURSE)) /* via workWith */
    method age() : integer,
          update()
end;
```

Note: Some information may be dropped,

E.g. workWith relationship type is not included in NURSE.

Some references:

- Tok Wang Ling and Pit Koon Teo, Toward Resolving Inadequacies in Object-Oriented Data Models. Information and Software Technology, vol 35, no 5, 1993.
<http://www.comp.nus.edu.sg/~lingtw/papers/IST93.teopk.pdf>
- Tok Wang Ling and Pit Koon Teo, Inheritance conflicts in object-oriented systems. Proceedings of DEXA'93, Springer-Verlag, 1993.
<http://www.comp.nus.edu.sg/~lingtw/papers/dexa93.teopk.pdf>
- Tok Wang Ling and Pit Koon Teo, A Normal Form Object-Oriented Entity-Relationship Diagram. Proceedings of ER'94, Springer-Verlag, 1994.
<http://www.comp.nus.edu.sg/~lingtw/papers/er94.teopk.pdf>
- Tok Wang Ling, Pit Koon Teo, Ling-Ling Yan: Generating Object-Oriented Views from an ER-Based Conceptual Schema. DASFAA 1993: pp 148-155.
<http://www.comp.nus.edu.sg/~lingtw/papers/dasfaa93.teopk.pdf>.