Part 7: Object Oriented Databases

Object Oriented Database Systems

Database Model:

Data Model = Schema + Constraints + Relationships (Operations)

• A logical organization of the real world objects, constraints, and relationships among the objects.

Object Oriented Data Model:

• It captures object-oriented concepts.

Object Oriented Databases:

• A collection of objects whose behavior and states, and the relationships are defined in accordance with an object oriented data model.

Object Oriented Database Management System:

• A database system which allows the definition and manipulation of object oriented databases.
## Structural Object Orientation

- It relates the notion of *complex objects*.
- The data model allows defining data structures for entities of any complexity.
- That is, objects whose attribute values are themselves objects rather than simple types, such as integers and strings, the is-a-part hierarchy.

**Example:**
- The \(x\) and \(y\) coordinates represent a point.
- Two diagonal points represent a rectangle.
- The layout mask for an integrated circuit can be represented by a collection of rectangles.

## Operationally Object Orientation

- The data model includes operators to deal with complex objects in their entirety.
- Operationally object-orientation implies structural object-orientation.
Behaviorally Object Orientation

- It relates the notation of data encapsulation, the concepts of methods, and the notations of the *is-a* type hierarchy type and its associated concept of inheritance.
- The data model incorporates features for users to define object types and operators associated with the types.
- In this case, the instance can be used only by these calling operators.
- In an example, if a rectangle is an encapsulated object, the user can not modify its coordinates without going through predefined operators.

Basic Concepts of Object Oriented Data Model

- Each set real world entity instances is modeled as a set of objects with associated class definition.
- Each object is associated with a unique object identifier.
- Each object has a set of instance attribute (instance variables, or properties), and methods (operators).
- The value of an attribute (property) can be an object, or a set of objects.
- An object can be a simple or atomic object, or a complex object.
- The concept allows arbitrarily complex object to be defined as an aggregation of other objects.
- The set of attribute values in an object represents the object status, the set of methods represent the object behavior(s).
Objects and Literals

- Objects and literals are basic building blocks of the object oriented data model.
- The main difference between the two is that: an object has both an object identifier and a state (or current value), whereas a literal has only a value but no object identifier.
- In either case, the value have a complex structure.
- The object state can change over time by modifying the object value.
- A literal is basically a constant value, possibly having a complex structure.
- An object is described by four characteristics:
  1. identifier, 2. name, 3. lifetime, and 4. structure.

Classes

Class:

- A class is a specification of both the abstract behavior and abstract state of an object.
- All objects that share the same set of attributes (properties) or methods may be grouped into a class.
- A class represents a template for a set of similar objects.
- An object belongs to only one class as an instance of that class.
- A class is a also an object, and is an instance of a metaclass.
- A class can be defined as a specialization of one or more classes.
- A class defined as a specialization is called a subclass, and inherits attributes and methods from its superclass(es).
- A class can also overwrite or redefine the definitions of attributes and methods from its superclass(es).
- Class contains two aspects: an object factory and an object warehouse. The object factory can be used as a constructor to create new objects. The object warehouse allows the class to be attached to its extension, i.e., the set of objects that are instances of that class.
Complex Objects and Object Structures

Simple Objects (Literals):
- Integers, characters, byte strings of any length, Boolean, and floats.

Complex Object:
- It is build up from simpler object by applying type constructors to them.

Type Constructors:
- They are used to build the values of complex objects from other objects.

Complex Object Constructors:
- Atom - Simple object
- Tuples - A natural way of representing properties of entities.
  - It is analogous to relational database tuple but not equivalent.

Collection Types:
- Sets - A natural way of representing collections from the real world.
- Bag - A bag is similar to a set, except that it allows duplicate values to exist.
- List & Arrays - It captures orders of data in matrices and time-series data
  - It is mainly for many scientific applications.

Complex Object Type

It is recursively defined as follows:

1. Let D be a name for a domain, then D is a (domain) type.

2. If T₁, ..., Tₙ are (name of ) types and A₁, ..., Aᵣ are distinct attributes, then
   - T = [A₁, ..., Aᵣ] is a tuple type, where T ≠ Tᵢ for 1 ≤ i ≤ n.

3. If T is a type, then T' = {T} is a se of type, where T' ≠ T.

4. Recursion may not be allowed in some systems.

Books = {Book}
Book = {Author: ..., Title: ..., ..., Reference: Books}
Book = {Author: ..., Title: ..., ..., Reference: {Book}}
Example

Example 1:

```
CHILD = [CNAME:string, BIRTHDATE:date, SEX:string]
Children = {CHILD}
```

```
SKILL = [SKTYPE:string, EXAMDATE:date]
Skills = {SKILL}
```

```
EMPLOYEE = [ID:integer, NAME:string, ADDRESS:string, CHILDREN:Children, SKILLS:Skills]
```

Example 2:

```
PERSON = [PNAME:string, HOME_ADDR:ADDRESS, SPEAKS:LANGUAGES, WORKS_FOR:COMPANY]
ADDR = [STREET:string, CITY:string, ZIP:integer]
LANGUAGES = {string}
COMPANY = {CNAME:string, LOCATION:ADDR}
```

Structure of EMPLOYEE Format

```
EMPLOYEE = (ID, NAME, ADDR, CHILDREN, SKILLS)
EMPLOYEE = (ID, NAME, ADDR, (CNAME, BIRTHDATE, SEX), (TYPE, EXAMDATE))
```

```
EMPLOYEE

/---

/ | 

/ | 

/ | 

/ | 

/ | 

/ | 

/ | 

/ | 

CNAME BIRTHDATE SEX TYPE EXAMDATE

ID NAME ADDR CHILDREN* SKILLS

Junping Sun                                        Database Systems 7-11

Junping Sun                                        Database Systems 7-12
Complex Objects

Complex Objects Are Recursively Defined as Follows:

1. If D is a domain type and dom(D) is a set of distinct values associated with D, then every o \( \in \) dom(D) is an (atomic) object of type D.

2. If T = \([A_1:T_1, \ldots, A_n:T_n]\) is a tuple and o_i is an object of type T_i, 1 \( \leq \) i \( \leq \) n, then \([A_1:o_1, \ldots, A_n:o_n]\) is a (tuple) object (of type T).

3. If T' = \([T]\) is a set type and o_1, ..., o_n are objects of type T, then \{o_1, ..., o_n\} is a set of object (of type T').

Example:

Definition

```plaintext
class Person
    superclass: none
    attribute name: string
    attribute address: string
    attribute soc-id: string

class Student
    superclass: Person
    attribute transcript: set of CourseHistory

class CourseHistory
    superclass: none
    attribute class: Course
    attribute when: date
    attribute grade: character

class Faculty
    superclass: Person
    attribute salary: integer
    attribute works-for: set of Position

class Position
    superclass: none
    attribute place: Department
    attribute rank: string
    attribute hired: date

class Course
    superclass: none
    attribute name: string
    attribute number: integer
    attribute home: Department
```
Definition

class Department
  superclass: none
  attribute name: string
  attribute college: string
  attribute chairs: set of Regime

class Regime
  superclass: none
  attribute person: Faculty
  attribute date-range: TimePeriod

class TimePeriod
  superclass: none
  attribute being: date
  attribute end: date
  method length: begin \times end \rightarrow integer

Nested Relation in Non-First Normal Form

(a) DEPT (schema)

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MANAGER</th>
<th>EMPLOYEES</th>
<th>PROJECTS</th>
<th>LOCATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ENAME</td>
<td>DEPENDENTS</td>
<td>PNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DNAME</td>
<td>AGE</td>
<td></td>
</tr>
</tbody>
</table>

(b) DEPT (example of a nested tuple)

<table>
<thead>
<tr>
<th>4</th>
<th>Administration</th>
<th>Wallace</th>
<th>Zaila</th>
<th>Thomas</th>
<th>8</th>
<th>New benefits</th>
<th>Stafford</th>
<th>Stafford</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jennifer</td>
<td>6</td>
<td>Computerization</td>
<td>Stafford</td>
<td>Greenway</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wallace</td>
<td>18</td>
<td>Phone System</td>
<td>Greenway</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Robert</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mary</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jabbar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Object Representation

Representation of Objects:

- Each object is a triple (i,c,v)
  - i is unique object identifier (the OID)
  - c is a constructor
  - v is the object value or state.
- An object value v is interpreted on the basis of the value of the constructor c in the tuple (i,c,v) that represents the object.
- If c = atom, the value v is an atomic value from domain D of basic values supported by the system.
- If c = set, the value v is a set of object identifiers \( \{i_1, i_2, \ldots, i_n\} \), which are the identifiers (OIDs) for a set of objects that are typically of the same type.
- If c = tuple, the value v is a tuple of the form \(<a_1:i_1, a_2:i_2, \ldots, a_n:i_n>\), where \( a_i \) is an attribute name (instance variable name) and each \( i_i \) is an object identifier (OID).
- If c = list, the value v is an ordered list of object identifiers \([i_1, i_2, \ldots, i_n]\) of the same type.
- If c = array, the value is an array of object identifiers.
- Note the object identifier \( i_i \) means the OID for any type of object.

\[O_1 = (i_1, \text{atom, Houston})\]
\[O_2 = (i_2, \text{atom, Bellaire})\]
\[O_3 = (i_3, \text{atom, Sugarland})\]
\[O_4 = (i_4, \text{atom, 5})\]
\[O_5 = (i_5, \text{atom, Research})\]
\[O_6 = (i_6, \text{atom, 22-May-78})\]
\[O_7 = (i_7, \text{set, } \{i_1, i_2, i_3\})\]
\[O_8 = (i_8, \text{tuple, } <\text{DNAME : } i_5, \text{ DNUMBER : } i_4, \text{ MGR : } i_3, \text{ LOCATION : } i_7, \text{ EMPLOYEE : } i_9, \text{ PROJECT : } i_{11}> )\]
\[O_9 = (i_9, \text{tuple, } <\text{MANAGER : } i_{12}, \text{ MANAGERSTARTDATE : } i_6>)\]
\[O_{10} = (i_{10}, \text{set, } \{i_{12}, i_{13}, i_{14}\})\]
\[O_{11} = (i_{11}, \text{set, } \{i_{15}, i_{16}, i_{17}\})\]
The Tree Representation of the Schema
Object Identity

Object Identity:

- Every object created has an identity unique to itself for its entire lifetime in the database system, whether or not there are ever other objects with similar or identical state or properties.

- An object has an existence that is independent of its value.
  With the concept of object identity:
  Two objects can identical (the same object).
  Two objects can be equal (same value or content).

- It supports object sharing and object updating.

Example:
A component shared by two objects can be updated without inconsistencies.
Object Identity and Object Identifier

Object Identity:
- An OO database system provides a unique identity to each independent object.

Object Identifier:
- The unique identity to each independent object is typically implemented via a unique, system generated object identifier, or OID.
- The value of an OID is not visible to the external user, but it is used internally by the system to identify each object uniquely and to create and manage inter-object references.

Properties of OID:
- The main property required of an OID is that it be immutable, that is, the value of an OID for a particular object should not be changed. This preserves the identity of the real-world object being represented.
- It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be reassigned to another object.
- This two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed.

Comparisons of OID and Key

- A key is unique within a relation in relational databases, whereas the object identifier is unique within entire system.
- By using OID one can define heterogeneous collections of objects which identify the objects belonging to different classes.
- A collection consists of a set of OIDs are independent from the class to which the objects belong.
- OIDs are implemented by the system, the applications programmers do not have to concern themselves with selecting appropriate keys for the various classes of objects.
- Better performance can be achieved, OIDs are implemented at lower level by the system.
- OID can not be modified and reused, but key in relational database can.
Identical Equality and Value Equality

Two objects can be identical (the same object).
Two objects can be value equal (same value or content).

\[ O_1 = \langle i_1, \text{tuple}, \{ a_1 : i_4, a_2 : i_6 \} \rangle \]
\[ O_2 = \langle i_2, \text{tuple}, \{ a_1 : i_5, a_2 : i_6 \} \rangle \]
\[ O_3 = \langle i_4, \text{tuple}, \{ a_1 : i_4, a_2 : i_6 \} \rangle \]
\[ O_4 = \langle i_4, \text{atom}, 10 \rangle \]
\[ O_5 = \langle i_5, \text{atom}, 10 \rangle \]
\[ O_6 = \langle i_6, \text{atom}, 20 \rangle \]

- Object \( O_1 \) and \( O_2 \) are value equal.
- Object \( O_1 \) and \( O_3 \) are identical equal.
- Object \( O_4 \) and \( O_5 \) are value equal.

Identity Equality

- Two objects \( O_1 \) and \( O_2 \) are identical, if they have the same identifiers, it corresponds to the equality of references or pointers of two objects (\( O_1 = O_2 \)).

\[ O_1: i_1 \quad \text{Name} \quad \text{Age} \quad \text{Address} \]
\[ O_2: i_{101} \quad \text{Name} \quad \text{Age} \quad \text{Address} \]

\[ \text{John} \quad 32 \quad \text{1212 Main, Walnut Creek, CA 94596} \]
\[ \text{Mary} \quad 34 \quad \text{i_4} \]

\[ \text{Streets} \quad \text{Streetname} \quad \text{City} \quad \text{State} \quad \text{Zip} \]
\[ \text{1212 Main, Walnut Creek, CA 94596} \]

\( O_1.\text{Address} \) and \( O_2.\text{Address} \) are identity equal or identical.
Object Representation

O1 = (i1, tuple, <Name : i2, Age : i3, Address : i4>)
O2 = (i2, atom, "John")
O3 = (i3, atom, 32)
O4 = (i4, tuple, <Street # : i5, Streetname : i6, City : i7, State : i8, Zip : i9>)
O5 = (i5, atom, "1212")
O6 = (i6, atom, "Main")
O7 = (i7, atom, "Walnut Creek")
O8 = (i8, atom, "CA")
O9 = (i9, atom, "94596")

O101 = (i101, tuple, <Name : i102, Age : i103, Address : i4>)

Shallow Equality (Simple Equality)

- Two objects O1 and O2 are shallow equal if their states or contents are identical, i.e. value(O1) = value(O2).
- This definition is not recursive (holds only for the first level of objects).

O1.Children and O2.Children are shallow equal.
Deep Equality

- Two objects \(O_1\) and \(O_2\) are **deep equal** if the correspondent atomic object values are equal. (replacing the identifiers with values)

\[
\begin{align*}
O_1 &: \text{i1} \\
& \quad \text{a} \quad \text{b} \\
& \quad 1 \quad \text{i2} \\
& \quad \text{c} \quad \text{d} \\
& \quad 2 \quad 3 \\
\text{a} \quad \text{i3} \\
& \quad \text{a} \quad \text{b} \\
& \quad 1 \quad \text{i5} \\
\text{a} \quad \text{i4} \\
& \quad \text{c} \quad \text{d} \\
& \quad 2 \quad 3
\end{align*}
\]

- Objects \(O_1\) and \(O_2\) are deep equal, and also \(O_1\) and \(O_2\), and \(O_2\) and \(O_3\).
- Objects \(O_1.b\) and \(O_2.b\) are identical equal
- Objects \(O_1\) and \(O_2\) are shallow equal.

Discussions on Object Equality

**Identical equality** checks whether two objects are the same.  
**Shallow equality** which goes one level deep, checks the corresponding identifiers of the values of elements of the objects.  
**Deep equality** checks the contents of the corresponding base or atomic objects.

Each object is (identical/shallow/deep) equal to itself: **reflexive**.
- If \(O_1\) is (identical/shallow/deep) equal to \(O_2\), then \(O_2\) is (identical/shallow/deep) equal to \(O_1\): **symmetric**.
- If \(O_1\) is (identical/shallow/deep) equal to \(O_2\), and \(O_2\) is (identical/shallow/deep) equal to \(O_3\), then \(O_1\) is (identical/shallow/deep) equal to \(O_3\): **transitive**.

- The strongest equality is identical, and shallow is stronger than deep.
- Two identical objects are always shallow equal and deep equal.
- Two shallow objects are always deep equal.
Composite Objects

Composite Object:
- It is in general a heterogeneous collection of objects, that is, the components objects from a number of different classes.

References:
- If the domain of an attribute is a non-primitive class, the value stored for the attribute is the object identifier(s) of the instance(s) of the domain.
  In this case, we say the object reference other objects.

Weak Reference:
- It is the object reference under the object data model without the part-of semantics.

Composite Reference:
- It is a weak reference augmented with the part-of relationship.

Exclusive Composite Reference:
- It is a reference from an object X to another object Y is a part of only X;

Shared Composite Reference:
- It is a reference from an object X to another object Y is a part of X and possibly other objects.

Dependent v.s. Independent Composite Reference

Dependent Composite Reference:
- It is a reference from X to Y such that the existence of Y depends on the existence of X.
- Deletion of an object will trigger recursive deletion of all objects referenced by the object through dependent composite reference (both exclusive and shared).

Independent Composite Reference:
- It is a reference from X to Y such that the existence of Y does not depend on the existence of X.

Summary of Four Types of Composite Reference:

(1) Exclusive dependent composite reference
(2) exclusive independent composite reference
(3) shared dependent composite reference
(4) shared independent composite reference
Structure for Object Instances

OID:
- It is system-generated identifier for object, which is globally unique in the lifetime of the OODBMS.

Object Size:
- the size of the object in bytes.

Number of Attributes:
- give the attribute count.

Identifiers of Attributes:
- System-defined attributes include the version number of and update timestep for versioned objects.
- User-defined attributes include the attributes for which the object has explicitly specified values.

Offsets:
- It consists of the offsets $O_i$ in the value part of the object storage structure, of the values of the attributes.

Values:
- The values can be the simple object values, or a reference to another object, the OID of the referenced object.

Object Instance Representation

<table>
<thead>
<tr>
<th>OID</th>
<th>Object Size</th>
<th>Number of Attributes</th>
<th>Offsets</th>
<th>Simple values or OIDs of Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Object Identifier

System-defined attributes

User-defined attributes

Identifiers of Attributes
Objects on a Disk Page

Structure of Class Objects

- In most object-oriented database systems, classes are regarded as objects, just as the instance of a class is an object, or it is an instance of a metaclass.

- A class object maintains the following information:

1. Class Attributes (shared by all instances of the class)
2. Class Methods (e.g., new)
3. Instance Attributes (names and structures)
4. Instance Methods (names and codes)
5. A reference to superclass
6. References to all instances

- All the information related to a class stored as a class object.
## Schema of an Object-Oriented Database

The Schema of an Object-Oriented Database Consists of:
- System-defined classes and User-defined classes

### System-Defined Classes:

**The class Class:**
- It contains attributes ClassName, Attributes, Superclasses, Subclasses, and Methods.
- ClassName is the name of the class
- Attributes is the set of attributes defined for or inherited into the class.
- The attributes Superclasses and Subclasses are the sets of superclasses and subclasses of the class.
- Methods is the set of methods defined for or inherited into the class.

**The class Attribute:**
- The class Attribute has attributes, Class, Domain, and InheritedFrom.
- The attribute Class references the class to which the attribute belongs.
- Domain specifies the class to which the value of the attribute is bound.
- InheritedForm indicates the attribute of the superclass from which the attribute is inherited.

### Classes for Schema

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>Attributes</td>
<td>Domain</td>
<td>MethodName</td>
</tr>
<tr>
<td>Superclasses</td>
<td>InheritedFrom</td>
<td>Code</td>
</tr>
<tr>
<td>Subclasses</td>
<td></td>
<td>InheritedFrom</td>
</tr>
<tr>
<td>Methods</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

OODB Schema in Possible Object Definition Language

class Person (extent persons) key ssn) {
  attribute struct Pname (string fname, string mname, string lname) name;
  attribute string ssn;
  attribute date birthdate;
  attribute enum Gender {M, F} sex;
  attribute struct Address {short no, string street, short aptno, string city, string state, short zip} address;
  short age();
}
Definition of Faculty Class in Possible ODL

class Faculty extends Person
( extent faculty )
{
    attribute string rank;
    attribute float salary;
    attribute string office;
    attribute string phone;
    relationship Department worksin inverse Department::has_faculty;
    relationship set<GradStudent> advises inverse GradStudent::advisor;
    relationship set<GradStudent> on_committee_of inverse GradStudent::committee;
    void give_raise ( in float raise )
    void promote ( in string new_rank );
};

Definition of Grade Class in Possible ODL

class Grade
( extent grades )
{
    attribute enum GradeValues { A, B, C, D, F, I, P } grade;
    relationship Section section inverse Section::students;
    relationship Student student inverse Student::Completed_sections;
};
## Definition of Student Class in Possible ODL

```java
class Student extends Person
( extent students )
{
    attribute string class;
    attribute Department minors_in;
    relationship Department majors_in inverse Department::has_majors;
    relationship set<Grade> completed_sections inverse Grade::student;
    relationship set<CurrSection> registered_in inverse
        Curr_Section::registered_students;
    void change_major ( in string dname )  raises ( dname_not_valid );
    float gpa();
    void register ( in short secno) raises ( section_not_valid );
    void assign_grade ( in short secno; in GradeValue grade )
        raises (section_not_valid, grade_not_valid );
}
```

## Definitions of Degree and GradStudent Classes in ODL

```java
class Degree
{
    attribute string college;
    attribute string degree;
    attribute string year;
};

class GradStudent extends Student
( extents grad_students )
{
    attribute set<Degree> degrees;
    relationship Faculty advisor inverse Faculty::advises;
    relationship set<Faculty> committee inverse Faculty::on_committee_of;
    void assign_advisor ( in string lname; in string fname )
        raises ( faculty_not_valid );
    void assign_committee_member ( in string lname; in string fname )
        raises ( faculty_not_valid );
};
```
Definition of Department Class in Possible ODL

class Department
( extent departments key dname )
{
    attribute string dname;
    attribute string dphone;
    attribute string doffice;
    attribute string college;
    attribute Faculty chair;
    relationship set<Faculty> has_faculty inverse Faculty::works_in;
    relationship set<Student> has_majors inverse Student::majors_in;
    relationship set<Course> offers inverse Course::offered_by;
};

Definition of Course Class in Possible ODL

class Course
( extent courses key cno )
{
    attribute string cname;
    attribute string cno;
    attribute string description;
    relationship set<Section> has_sections inverse Section::of_course;
    relationship Department offered_by inverse Department::offers;
};
Definitions of Section and CurrSection Classes in ODL

class Section
( extent sections )
{
    attribute short secno;
    attribute string year;
    attribute enum Quarter {Fall, Winter, Spring, Summer} qtr;
    relationship set<Grade> students inverse Grade::section;
    relationship Course of_course inverse Course:: has_sections;
};

class CurrSection extends Section
( extent current_sections )
{
    relationship set<Student> registered_student inverse Student::registered_in;
    void register_student ( in string ssn )
        raises ( student_not_valid, section_not_valid, section_full );
}
Step 1:

- Create an ODL class for each EER entity type or subclass (subtype). The defined class type should include all the attributes defined as atomic data type or complex type predefined in terms of “tuple constructor.”

- Define multivalued attributes by using the set, bag, or list constructors.

- Define composite attributes by tuple constructor (struct declaration in ODL).

- Declare an extent for each class, and specify any key attributes as keys of the extent if the facility in ODL is available.

Step 2:

- Add relationship properties or reference attributes for each binary relationship in to the defined class(es) that participate in the relationship.

- These may be created in either one or both directions. If a binary relationship is represented by references in both directions, declare the references to be properties relationship properties that are inverses of one another if the facility in ODL is available.

- Declare the relationship as single-valued on the side such that the cardinality ratio of the relationship is 1:1 or N:1 direction, and as collection type (set, bag, list, ..) on the side 1:N or M:N direction.

- For attribute belonging to a relationship, use tuple constructor as the following format:
  <reference as relationship property, relationship attribute>
Mapping an EER Schema to an ODB Schema

Step 3:
- Include appropriate operations for each class.
- Constraint checking should be incorporated in both object constructor, destructor, and others.

Step 4:
- An defined class that corresponds to a subclass in the EER schema inherits (via extend) the type and methods of its superclass in the schema in ODL.

Step 5:
- Weak entity can be mapped in the same way as regular entity types.
- Alternatively, weak entity can be included as composite multivalued attributes in the owner entity if the weak entity does not participate in any relationships except the identifying relationship.
- The multivalued composite attribute for weak entity can be defined by collection types such as: set, bag, list, ...

Step 6:
- An n-ary relationship with degree n > 2 can be mapped into a separated class, with appropriate references to each participating classes.
- These references are based on the mapping a 1:N relationship from each class that participating entity type to the class that represents the n-ary relationship.
- An M:N binary relationship may also use this option, especially, if it contains the relationship attribute.