Journal of Object-Oriented Programming, March 1996, pp. 61 - 64.

OODB Design with EER

Byung S. Lee
Graduate Programs in Software
University of St. Thomas

ABSTRACT

In contrast to the conventional methodology of object-oriented program design focused on the interaction of objects, object-oriented database design should be based on the *representation* of objects. We put more emphasis in the application semantics pertinent to the structures of, relationships between, and constraints on objects than operations on the objects. Enhanced Entity-Relationship (EER) model is a convenient tool for representing these semantics. In this paper, I address the concept and methodology of using the EER model to design an object-oriented database schema. The EER model facilitates the design of a logical schema that can be mapped to an object-oriented schema straightforward. An EER schema diagram is also a useful document that describes the logical database schema to other designers and users.

INTRODUCTION

Designing an object-oriented database (OODB) should be portrayed as a process distinct from designing an object-oriented program. Most popular object-oriented design methods are based on how objects 'interact', that is, how methods are invoked among objects. From a database viewpoint, however, the primary concern for a design is to represent the structures and relationships of data items that are to be stored in a database. Interactions among objects are to be cared when we write application queries on the designed database. In this article, I will address using the Enhanced Entity-Relationship (EER) model to design an OODB schema, based on the experience of teaching OODB design to graduate students.

EER model is an appropriate tool for designing a logical database schema which can be mapped to a system data model of your choice. Provided with the constructs for representing entities, relationships, and constraints explicitly, EER model is also a convenient tool for documenting the design and facilitating communications among human users. When designing an OODB schema, it is easier and more accurate to design an EER diagram first and then implement it in an OODB. Basically, entity types are mapped to OODB classes and relationships are mapped to object references in the OODB classes. Constraints that cannot be specified declaratively in the schema are hard-coded in methods. Additional methods are identified and included in the OODB classes at the time of designing queries and application programs.

DATA MODEL AND DATABASE DESIGN

Designing a database is to create a schema by defining the structures of data entities and their relationships. A schema can also include constraints on entities and relationships. Different data models provide different notions of data structures, relationships, and constraints. In the relational data model, a data structure is defined as a table. The attributes of a table are specified by atomic, system-defined data types. A relationships is represented by a foreign key in one relation that refers to the primary key of the same or another relation. Because each entity is identified uniquely by the value of the primary key of a table, relational data model requires to preserve the entity integrity on the primary key and referential integrity on each foreign key of a table. On the other hand, in an object-oriented data model, a data structure is defined as a class. The attributes of a class are allowed to be of non-atomic (collection) or user-defined types (complex attributes). Relationships are represented by complex attributes, which are materialized to be the references to the linked objects. Unlike relational, entities are identified uniquely by their own identities, and hence O-O data model has no need for the primary or foreign key constraints.

Once you buy a DBMS, whether relational or object-oriented, you have to understand the data model -- how to represent structures, relationships, and constraints -- supported by the system and learn how to design a database using its data definition language (DDL). If you have experiences in either one, you will agree that there are lots of semantics you wish to represent in your schema but cannot because the system data model lacks the necessary features. Further, once you create your schema in your system's DDL, you find it difficult to transfer the design to someone else without spending numerous hours to explain what are represented in your design. We need a data model which is semantically more expressive than relational or object-oriented. EER was chosen as the one that can satisfy these needs.

THE EER MODEL

EER model is an enhanced version of the Entity-Relationship (ER) model -- one of the most popularly used semantic data models. A semantic data model refers to a data model that supports a richer set of modeling constructs for representing the semantics of entities, their relationships, and constraints. EER model is used conveniently as a tool for logical data modeling and design documentation. A database schema designed in EER is independent of the specific data model supported by your DBMS -- whether it is relational, object-oriented, network, or hierarchical. Being a higher-level data model, it can be implemented into your system's data model with little difficulty (Figure 1). You will find that the system data model is not powerful enough to represent all the semantics you were able to in the EER, and hence give up some of the higher level semantics. EER in turn will be a good reference to retrieve these missing semantics from.

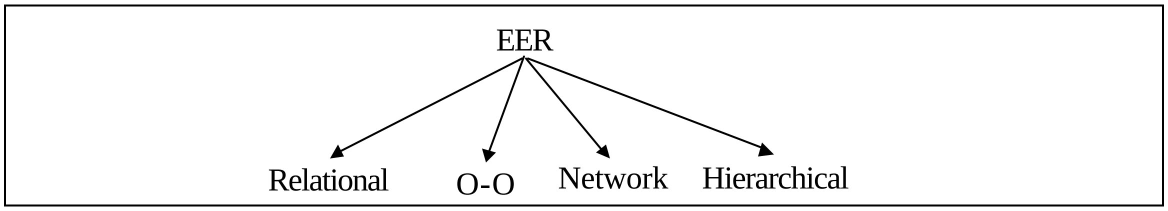


Figure 1. Mapping EER to system data models

EER SCHEMA DESIGN

Figure 2 shows the diagram of an example EER schema. Entity types are shown in boxes, attributes in ovals, and relationship types in diamonds. The numbers within parentheses on each side of a relationship type denote the minimum and maximum numbers of entities linked by the relationship. For the Work_for relationship, for example, an employee can work for one and only one department and a department can hire minimum 2 and maximum 30 employees. A dependency constraint on a relationship is shown by an arrow. For example, a Child entity can exist only if they exists an Employee entity linked through the Raise relationship. An attribute may be composite (e.g., name of Employee) or multi-valued (e.g., locations of Department). Multi-valued attributes are shown in double-lined ovals. Underlined attributes are key attributes, whose values are distinct over all entities of a given type. If an entity type does not have a complete key attribute, like the Child in Figure 2 (because two children raised by two different employees may have an identical name), the entity type is called a weak entity type and shown in a double-lined box. The attribute 'name' of Child is only a partial key, as denoted by a dotted underline

Entity types can be related by the IS-A relationship resulting from a specialization or generalization. These entity types, connected by the dense lines denoting the IS-A relationship, configure a hierarchy (or lattice). SalesEngineer is particularly called a shared subclass entity type. Subclass entities may be overlapping (the letter *o* in a circle) or disjoint (*d* in a circle). If managers, sales representatives, and engineers are the only types of employees, we would have used a double line from Employee to the circle to indicate such a constraint. All the attributes, relationships, and constraints on an entity type are inherited to its subclass entity types.

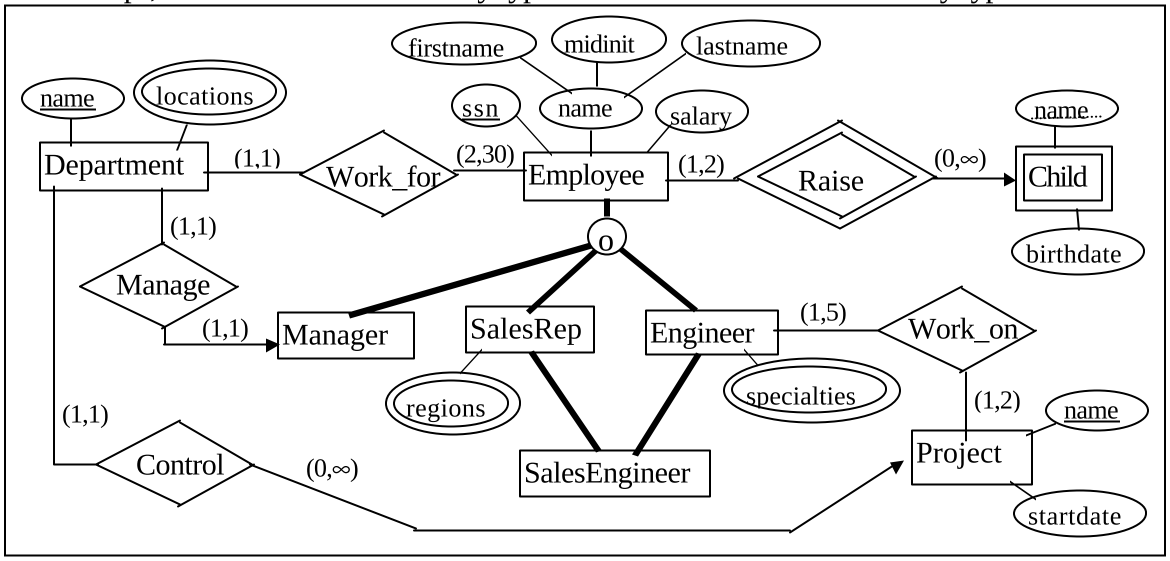


Figure 2. An example EER schema

EER VERSUS OO

There are a couple of distinctions to be highlighted here between EER and O-O before we discuss the mapping between them.

EER relationships

Relationships are represented explicitly in EER but only implicitly through object references in OO. An EER relationship enables us to express the semantics, cardinalities, and dependencies of how two or more entity types are related. This is the primary source of the elaborate expressive power of EER. Besides, we can represent more sophisticated constraints or comments in the form of text on the EER diagram. Note that EER is a logical data model and used for a human interpretation.

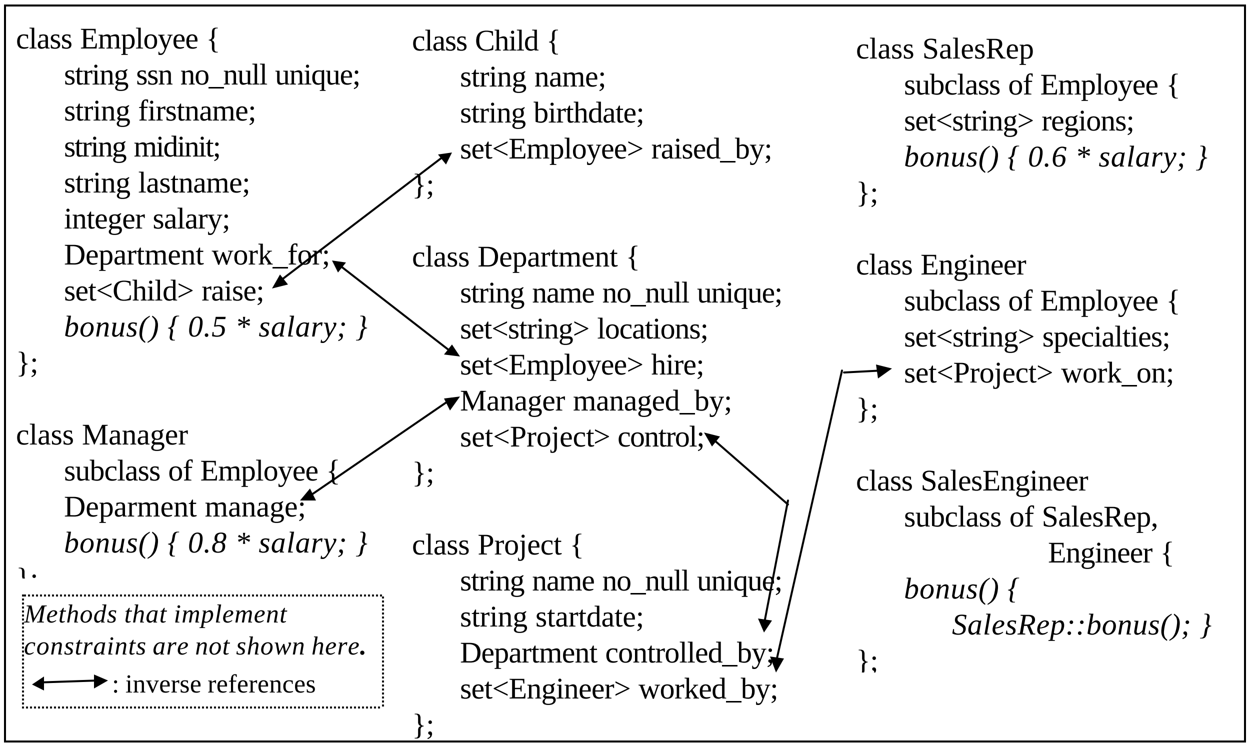
EER key attributes

Key attributes that are used to distinguish entities uniquely in EER are not used for the same purpose in an O-O data model because objects are distinguished by object identifiers that are assigned by the system. However, these key attributes can still be useful criteria for searching objects, particularly with indexes created on them.

O-O methods

EER does not have a notion of methods. Does this mean that EER lacks modeling power compared to an OODB? No, it does not. To a database designer, methods are not an essential component of a database schema but a new programming technique for encapsulating the data structure. When there are constraints that cannot be specified declaratively in an O-O class, we are forced to verify the constraints procedurally in our application programs. It is by virtue of the O-O programming technique that the constraint verifications can be implemented in methods and included as part of the O-O classes. Of course, we do not have to bother to write these methods if we have any declarative constraint specifications available from the OODB system. Later on, when queries are written on the designed OODB, additional methods are defined as necessary and included as part of the O-O classes as well. There is even a negative consensus among OODB designers against the support for a 'strong' encapsulation in an OODB design. To them, it is rather a nuisance to encapsulate every attribute accessed by an application query by defining a corresponding 'get' method (e.g., 'get_X() { return X; }'). Besides, how can we create an index on an encapsulated attribute? Indexes are created on attribute values, not on the functions that return attribute values.

MAPPING EER TO OO



```
class Employee {
    string ssn no_null unique;
    string firstname;
    string midinit;
    string lastname;
    integer salary;
    Department work_for;
    set<Child> raise;
    bonus() { 0.5 * salary; }
};

class Manager
    subclass of Employee {
    Deparment manage;
    bonus() { 0.8 * salary; }
};

class Child {
    string name;
    string birthdate;
    set<Employee> raised_by;
};

class Department {
    string name no_null unique;
    set<string> locations;
    set<Employee> hire;
    Manager managed_by;
    set<Project> control;
};

class Project {
    string name no_null unique;
    string startdate;
    Department controlled_by;
    set<Engineer> worked_by;
};

class SalesRep
    subclass of Employee {
    set<string> regions;
    bonus() { 0.6 * salary; }
};

class Engineer
    subclass of Employee {
    set<string> specialties;
    set<Project> work_on;
};

class SalesEngineer
    subclass of SalesRep,
                Engineer {
    bonus() {
        SalesRep::bonus(); }
};
```

Methods that implement constraints are not shown here.

⟷ : inverse references

Figure 3. O-O classes mapped from the EER schema of Figure 2

We can implement an EER schema to an OODB schema in three steps:

1) Map EER entity types and relation types to O-O classes. Implement in methods the constraints that cannot be specified declaratively.
2) Add additional methods to each O-O class. The necessary methods are identified as the result of analyzing application queries.
3) Extend the O-O classes to turn them into database classes, that is, add the system-provided statements for handling persistent objects.

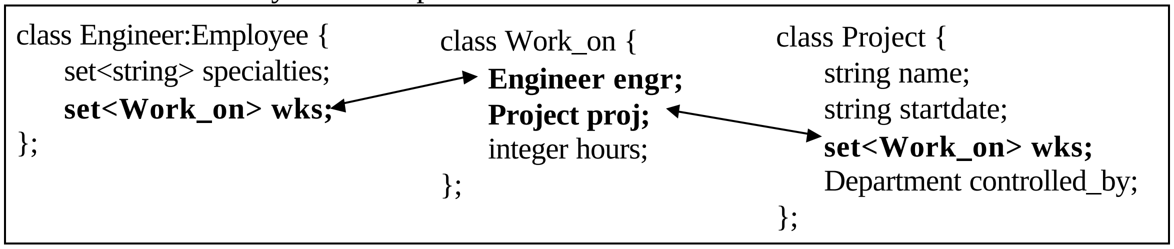
Figure 3 shows the result of mapping the EER schema of Figure 2 to an OODB schema after steps 1 and 2. It was assumed that there is an application query which calculates the bonus differently for Employee and its subclasses.

Rules of thumb

Shown below are the rules of thumb that can come in handy for carrying out the step 1.

- Entity types: Each entity type, primary or weak, is mapped one-to-one to an O-O class. A weak entity type does not make any difference from a primary entity type because key attributes are not used to identity objects in an OODB. An entity type hierarchy is mapped to an O-O class hierarchy. Shared subclass entity type is mapped to a subclass with multiple inheritance. A composite attribute is mapped to its component attributes. A multi-valued attribute is mapped to a collection (e.g., set, bag, or list) attribute.
- Relationship types: Each binary relationship type is mapped to an object reference in an O-O class. Use a bi-directional reference to facilitate a bi-directional navigation if the system

supports the implementation of inverse references. If the cardinality of a relationship is greater than one, use a collection of object references (e.g., set<Project>). If a binary relationship type has one or more attributes of its own, then introduce a third O-O class for storing those attributes and make object references to the two entity types. An example is shown in Figure 4 for the case in which the Work_on relationship type of Figure 2 has an attribute 'hours'. An N-ary (N>2) relationship type is mapped to an O-O class in the same manner as the binary relationship with attributes.



```
class Engineer:Employee {
    set<string> specialties;
    set<Work_on> wks;
};

class Work_on {
    Engineer engr;
    Project proj;
    integer hours;
};

class Project {
    string name;
    string startdate;
    set<Work_on> wks;
    Department controlled_by;
};
```

Figure 4. Mapping a many-to-many relationship <u>with</u> attributes to O-O classes

- <u>Constraints</u>: If the system provides a facility for specifying a constraint declaratively, use it. Otherwise, we have to implement the constraint procedurally in a method.
  - <u>Constraints on entity types</u>: If available in your system, specify the keywords such as 'no_null' and 'unique' on an attribute mapped from the key of an entity type. There is hardly a simple way of enforcing the constraint of overlapping or disjoint entities in an entity type hierarchy. In the case of a multiple inheritance, like SalesEngineer, overlapping objects are created naturally along the hierarchy on constructing a SalesEngineer object. In a more general case such as an overlap between Manager and SalesRep, constructing a Manager object does not necessarily create a SalesRep object as well. We need to implement separate constructors for an overlapping case and disjoint case, and use them accordingly in each case.
  - <u>Constraints on relationship types</u>: Cardinality constraints on a relationship type should be implemented in the methods for constructing or destroying objects and the methods for inserting/deleting members into/from a collection of object references. An example is shown in Figure 5 for the '(2, 30)' constraint on the cardinality of the Work_for relationship type that is mapped to the attribute 'hire' of the Department class. If there exists a dependency constraint on a relationship type, this constraint should be implemented in the methods for constructing and destroying objects as well.

```
Department(set<Employee> emps) {
    if (cardinality(emps) < 2 or
        cardinality(emps) > 30) {
        error ("Constraint violation");
        exit;
    }
    // Construct the Department object.
}
```

a. In a constructor

```
add_employee(Employee emp) {
    if (cardinality(hire) >= 30) {
        error ("Constraint violation."):
        exit;
    }
    // Insert emp into hire.
}
```

b. In a set insertion method

Figure 5. Implementation of a cardinality constraint checking in a method

Special case: multi-dimensional subclasses

It is worthwhile to give a special consideration to the case of a multi-dimensional subclasses, such as the one shown in Figure 6. The Employee entity type is specialized into subclass entity types along two orthogonal dimensions -- one into Engineer and SalesRep based on their job titles, and the other into Staff and Manager based on their ranks. The SalesManager is in turn a shared subclass entity type of SalesRep and Manager, and TechnicalStaff is that of Engineer and Staff. Note that the two superclass entity types of SalesManager and TechnicalStaff, respectively, are those that belong to separate hierarchies under Employee. If we map the four subclass entity types to O-O classes one to one (Figure 7a) by following the rules of thumb, we lose the semantics of two orthogonal dimensions. Consequently, it complicates the implementation of OODB classes. For one thing, whenever an Engineer object is created, its rank must be checked to see if it is a Manager or Staff object as well. If it turns out to be a Staff object, it should be created as a Staff object and also a TechnicalStaff object. Besides, since both Engineer and Staff objects are created, it may create duplicate Employee objects along the two hierarchies. (Some systems may be able to handle this problem, others may not.) My recommendation is to use a 'relational' approach in the Employee O-O class (Figure 7b), that is, to use *attributes* to distinguish between multi-dimensional subclass entities. It will eliminate the need for all subclasses, which is less object-oriented but easier to implement.

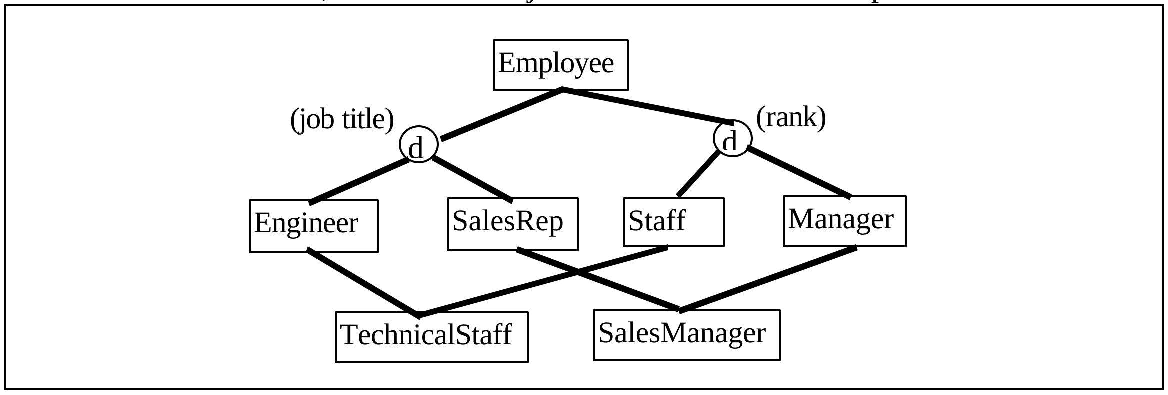


Figure 6. Multidimensional subclass entity types in EER

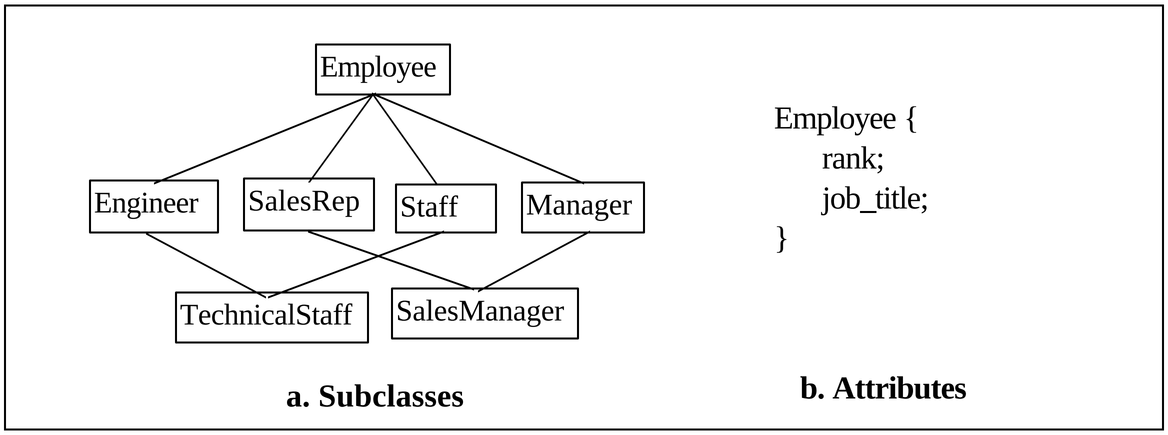


Figure 7. Subclasses versus attributes for mapping from Figure 6 to O-O classes

SUMMARY

In this article, I presented the ideas of using the EER model for designing an OODB schema. EER is a semantic data model for designing a logical database schema and can be implemented to a system-specific database schema, such as relational or object-oriented. Using the EER facilitated the OODB design by dichotomizing the design process in two steps -- first focusing on the structural design at a semantic level and focusing on the behavioral design at the time of designing application queries. With its rich expressiveness, an EER diagram is used as an important document that describes the logical database schema for human interpretations. I was very pleased with the process of designing an OODB starting with the EER model.

References

1. Elmasri, R. and Navathe, S. Fundamentals of Database Systems (2nd ed.), Benjamin Cummings Publishing Company, Inc., 1994.
2. Chen, P. The Entity Relationship Model -- Toward Unified View of Data, ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976.
3. Special Report SR-OPT-001826: Information Modeling Concepts and Guidelines (Issue 1), Information Exchange Management, Bellcore, Morristown, New Jersey, January 1991.
4. Kemper, A. and Moerkotte, G. Object-Oriented Database Management, Prentice Hall, Englewood Cliffs, New Jersey, 1994.