10. THE OBJECT-ORIENTED DATA MODELS

During the past several years the computer research community has undertaken a number of efforts to produce tools and methodologies that are needed to overcome the software crisis. The combined outcome of a number of such efforts is the object-oriented programming paradigm.

The concepts embodied in the object-oriented programming paradigm have received tremendous attention from many disciplines of computer science. In particular, the database research community has concentrated its efforts on extending database systems with object-oriented capabilities. A number of prototype and commercial systems have been built as a result of this activity (Orion, Gemstone, etc). These systems provide all the traditional DBMS services, but the data models that they support are based on object-oriented concepts. Such database systems have been termed object-oriented database systems (henceforth OODBS).

Despite the popularity that OODBSs have attained, no standard has yet been established for the object-oriented data model — as it is the case with previous data models (e.g., the network data model). Usually, the term object-oriented is used to refer collectively to a set of semantic modeling mechanisms for capturing the information of an application’s real world. However, there appears to emerge a consensus among database researchers regarding a minimal set of features that a database model must exhibit if it is to be termed object-oriented. These features are the following:

- Object and object identity
- Semantic primitives for the modeling of the structure of information
- Object behavior and encapsulation

The first two features can be ascribed to the semantic database models which in general exhibit an orientation toward modeling the structure of the information in an application’s real world. They suggest that an object-oriented database model should provide explicit semantic abstraction primitives for direct object representation, object categorization, integrity constraints, etc. Although a plethora of semantic modeling primitives can be found among various database systems, most of the object-oriented data models adopt only a handful of them that best fit the needs of the application domains for which they are intended. Two of the most common semantic modeling primitives encountered in object-oriented data models are:

- **Classification** of objects by categories.
- **Inheritance** — if an object is a member of a category (e.g., STUDENT), then it is automatically a member of its supercategories (e.g., PERSON). Therefore, the properties of the supercategories (e.g., the attribute last LAST-NAME) automatically apply to the subcategories. Note, however, that supercategory-subcategory is not a
relation between objects of the two categories; rather objects of a category are simultaneously members of all of its supercategories. In some data models, due to implementational restrictions, categories must be disjoint. In those models, inheritance can be introduced indirectly by establishing *is-a* relations among objects and “*rules of inheritance.*”

The last feature, **object behavior**, means that an object-oriented model should provide abstraction mechanisms for modeling the behavior of objects in an application’s real world.

---

**Example 10-1.**

When a student is registered for courses, a certain series of steps is to be followed, such as checking if a course is offered, adding the student into the list of students taking a course, etc. The same series of steps is followed every time that a student is registered for a new term. Such steps constitute a student’s behavior during registration.

---

Various approaches can be found in the OODBSs for modeling object behavior. The most common of these are *methods* which are discussed later in this chapter.

Finally, **encapsulation** means that the values associated with objects in an object-oriented database can only be accessed via specific predefined operations which implement their behavioral characteristics (i.e., methods). Consequently, users should not be concerned about the implementation details of such operations but rather with their functional specification. Applications which access the values associated with certain objects do not have to be rewritten every time that the programs which implement the behavioral characteristics of these objects are modified. This increases the degree of data independence of applications and enforces a near uniform treatment of objects throughout applications.

---

**Example 10-2.**

Consider the schema for the University database. In an object-oriented system one could define a program $P$ whose input is an object in *PERSON* and whose output is a printout of all the concrete values associated with that object. Any modifications in the implementation of $P$ should not have any effect on the applications which use $P$ as long as $P$ conforms to its original input and output specifications.

---

An object-oriented data model, therefore, is a semantic model which provides mechanisms for modeling not only the structural but also the behavioral characteristics of an application’s real world.

In this chapter we describe the object-oriented paradigm for database modeling from the perspective of the Semantic Binary Model. For the purposes of this presentation we use our example of an object-oriented data model. We call this model the *Semantic Binary Object-
10.1. The Semantic Binary Object-Oriented Data Model

This section introduces SBOODM, which is an example of an object-oriented data model. SBOODM has been defined with the intention to expose the basics of the object-oriented paradigm to the readers already familiar with semantic modeling. It is a semantic binary model augmented with an abstraction mechanism to model the behavior of objects.

10.1.1. Object-oriented schemas

An object-oriented schema must capture not only the structural properties of an application’s real world but also its behavioral properties. In the SBOODM, a schema lists all aspects of an application’s real world in graphical form and in the appendix. More formally a SBOODM schema is defined as follows:

**SBOODM schema** — a semantic binary schema whose appendix lists the methods defined for each of the categories and the interface to each category. The concepts of method and category interface are defined in the sections that follow.

10.1.2. Methods

In the object-oriented paradigm it is possible to predefine a library of operations (procedures and functions) for a given schema. These operations are called methods and are the primary means for modeling the behavioral characteristics of an application’s real world. Methods can be constructed by data manipulation primitives. Of special interest are the object methods and category methods.

In the SBOODM methods are implemented via Extended Pascal procedures or functions and are defined as follows:

**Object method** — an extended Pascal function or procedure which satisfies the following:

- The first formal parameter of the method is of type ABSTRACT and must be a value parameter. When a method $m$ is invoked and an object $o$ is assigned to its first parameter, then we say that $m$ is invoked on $o$.

- The method restricts the first argument to belong to a particular category. When that is a category $C$, we say that the method is defined on $C$.

We allow for the following convenient syntactic abbreviation for the header of a procedure method $m$ defined on a category $C$:

```
procedure m(first-argument:C; other-arguments);
```

(and similarly for function methods.)
Example 10-3.

The method

```
procedure print-name(i:INSTRUCTOR);
begin
    writeln(i.LAST-NAME, i.FIRST-NAME)
end
```
can be regarded as a syntactic abbreviation for

```
procedure print-name(i:ABSTRACT);
begin
    if not (i is an INSTRUCTOR) then
        writeln(‘Error: The input object is not of the category INSTRUCTOR’)
    else writeln(i.LAST-NAME, i.FIRST-NAME)
end
```

Example 10-4.

An object method which computes and returns the age of a person.

```
function get-approximate-age(person: PERSON): INTEGER;
var current-year: INTEGER;
begin
    read(current-year);
    get-age := current-year - person.BIRTH-YEAR;
end;
```

Example 10-5.

An object method which updates the major of a student.

```
procedure change-major(student-to-change: STUDENT; new-major: STRING);
```
var dept: ABSTRACT;
begin
  transaction
    for dept in DEPARTMENT where dept NAME new-major do
      student-to-change.MAJOR := dept;
  end;

Example 10-6.
An object method which enrolls a student in a course.

procedure enroll(student-to-enroll: STUDENT; course-name: STRING;
                   quarter: STRING; year: INTEGER);
var offer, enrollment : ABSTRACT;
begin
  transaction
    for offer in COURSE-OFFERING where offer.THE-COURSE.NAME = course-name
            and offer.THE-QUARTER.YEAR = year
            and offer.THE-QUARTER.SEASON = quarter do
      begin
        create new enrollment in COURSE-ENROLLMENT;
        enrollment.THE-STUDENT := student-to-enroll;
        enrollment.THE-OFFER := offer;
      end
  end
end;

Example 10-7.
An example of an object method which interactively registers a student into up to
five courses.

procedure register(student-to-register: STUDENT);
var enrollment: ABSTRACT;
    course-name, quarter: STRING;
begin
  transaction
    begin
      nbr-courses := 0;
      writeln(`Please enter the current year and quarter');
      readln(year, quarter);
      repeat
        writeln(`Please enter the course name');
        readln(course-name);
        enroll(student-to-enroll, course-name, quarter, year);
        nbr-courses := nbr-courses + 1;
        writeln(`Register for another course (y/n) ?');
        readln(answer);
      until (answer = 'n' or nbr-courses >= 5) ;
    end;
end;

Example 10-8.

An example of an object method which gets all the information pertaining to a particular person from the standard input and then it inserts it into the database.

procedure get-person-data(person-object: PERSON);
  var first-name, last-name, address: STRING;
    birth-year: 1870..1990;
begin
  transaction
    begin
      writeln(`Please enter the first and last name and the birth year');
      readln(first-name, last-name, birth-year);
      person-object.FIRST-NAME := first-name;
  end;
end;
Example 10-9.

A new student has arrived at the Computer Science department and he or she must be registered in several courses. The following program will request all the information pertaining to the new student and the courses that he or she would like to be registered for, and then it will insert them into the University database.

```
program New-Student-Registration(Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var new-student, student-category: ABSTRACT;
begin
  (* Create a new student *)
  create new new-student in STUDENT;
  (* Get all the personal information about the new student *)
  get-person-data(new-student);
  (* Update the student’s major *)
  update-major(new-student, 'Computer Science');
  (* Get the course names for the courses that the student is interested in taking and register the student in these courses *)
  register-student(new-student);
end.
```

Sometimes, it is of interest to the users of a database to retrieve or store global information about a category rather than specific information about its objects. Such information about categories can be manipulated by methods which are called category methods, and are a
special case of object methods in that the objects upon which they are invoked are categories themselves. Category methods are defined within the SBOODM by considering a special category CATEGORY whose objects are the categories of the schema. The category CATEGORY is an example of a special type of categories called meta-categories. Meta-categories are used together with meta-relations to manipulate the concepts of a schema. Formally, category methods are defined as follows:

**Category method** — an object method such that the category for which it is defined is the meta-category CATEGORY.

```
Example 10-10.

The following is an example of a category method which counts the number of objects contained in a given category.

```
function category-size(category-object: CATEGORY): INTEGER;

var object: ABSTRACT;

begin
  count := 0;
  for object in category-object do count := count + 1;
  category-size := count;
end;
```

In the example above, the method category-size had to repeatedly read the database in order to determine the number of objects that are members of some particular category. Obviously, category-size is an expensive operation in terms of number of times that the database must be accessed, especially if the size of the database is large. An alternative to this implementation is to have all global information that is of concern to the users of a category (e.g., the number of objects which are members of a category) associated with the category object itself via some relation.

```
Example 10-11.

Suppose that for some users of the University database it is necessary to know the number of objects that are recorded in the instantaneous database for each category. Instead of computing this number by the method category-size, it is possible to alter the schema so that this information is directly recorded in the instantaneous database and it does not have to be recomputed every time that it is needed. By adding the relation number-objects from the category CATEGORY to
```

Rishe-92-DDS p. 10.8
the category of integer numbers, it is possible to record the number of objects which are members of a category in the instantaneous database. Each time that a new object is inserted into the database, the integer value associated with number-objects must be incremented by 1. The following is an example of a category method to increment the number-objects counter.

\begin{verbatim}
procedure increment-object-counter(category-object: CATEGORY);
begin
    category-object.NUMBER-OBJECTS := category-object.NUMBER-OBJECTS + 1;
end;
\end{verbatim}

Example 10-12.

The following is an example of a category method which returns the number of objects which are members of a given category in the instantaneous database.

\begin{verbatim}
function get-number-objects(category-object: CATEGORY): INTEGER;
begin
    get-number-objects := category-object.NUMBER-OBJECTS;
end;
\end{verbatim}

The database designer can provide the users with a library of methods attached to the schema. The bodies of the methods are not visible to the users, only the headers of the methods are visible. For some methods which are used only as subroutines in other methods, neither the headers nor bodies are visible to the users. The schema’s interface is the the headers of the methods that may be used by the users. For any category \( C \), the user-visible headers of the schema object methods defined on \( C \) are called the category interface of \( C \).

In SBOODM, a method defined for a category can invoke any of the other methods defined for the categories in the schema. However, it should be noted that in some object-oriented models, the methods defined for a specific category cannot indiscriminately invoke any other methods defined in the schema. In these models various mechanisms are provided to restrict the visibility of the methods defined for a particular category to only a subset of the categories defined in the schema.
10.1.2.1. Overloading and late binding

It is often convenient to use the same name for different methods. Such a need arises primarily when the functional specification of a method resembles that of other methods. Consider the following example, which illustrates this point.

Example 10-13.

A method needs to be defined for the University schema which given an object in \texttt{PERSON} will display its name and address. The following procedure may be used to implement this method.

\begin{verbatim}
procedure display(person: PERSON);
begin
    writeln(`Name:', person.FIRST-NAME, ` ', person.LAST-NAME);
    writeln(`Address:', person.ADDRESS);
end;
\end{verbatim}

Similarly, for persons who are students another method is to be defined which given an object in \texttt{PERSON} it displays its name, address, and major. A convenient choice for the name of this method is also \textit{display}. The following is a procedure to implement this method:

\begin{verbatim}
procedure display(student-object: STUDENT);
begin
    writeln(`Name:', student-object.FIRST-NAME, student-object.LAST-NAME);
    writeln(`Address:', student-object.ADDRESS);
    display-dept-name(student-object.MAJOR);
end;
\end{verbatim}

Here, \textit{display-dept-name} is a method defined on \texttt{DEPARTMENT}: given an object in \texttt{DEPARTMENT} it displays the department’s name.

In the above example two methods are defined and both are to display the values associated with abstract objects which are members of the category \texttt{PERSON}. Their functional specifications present a lot of similarities, which justifies the choice for sharing the same name. A method name shared by more than one method is called an \textbf{overloaded method name}.

Example 10-14.

A user’s program can be as follows:
for p in PERSON do display(p);

For persons that happen to be students the second method will be automatically used.

A reference to an overloaded method name cannot be bound to a specific function or procedure body at compile time. Therefore, the code that is to be bound to an overloaded method name is determined during execution time (this practice is called late binding). If late binding were not available, then a programmer would have to use different names for the display methods above, such as display-person and display-student, and then test for the category where each object belongs before referencing any of the method names.

Example 10-15.
An equivalent user program without overloading would be:

```plaintext
for p in PERSON do
    if p is a STUDENT then display-student(p)
    else display-person(p);
```

However, with late binding the programmer is free from such constraints and he or she can simply reference the name display where needed.

In SBOODM the following algorithm is used to determine the procedure or function body which is to be bound to an overloaded method name reference.

### 10.1.2.2. A late binding algorithm

Let $C_1, C_2, \ldots, C_n$ be categories in a schema and let $m$ be a method name which is shared by methods defined for each of the categories $C_1, C_2, \ldots, C_n$ (i.e., $m$ is an overloaded method name). Let $b_1, b_2, \ldots, b_n$ be the method bodies where $b_i$ is the body of a method with name $m$ defined for category $C_i$ (for $i = 1, 2, \ldots, n$). Finally, let $o$ be the object identifier that is assigned to the first parameter of $m$ during some invocation of $m$.

The algorithm will test the object $o$ for membership in each of the categories $C_1, C_2, \ldots, C_n$. If $o$ belongs to the category $C_i$, then $m$ is bound to the body $b_i$ (where $1 \leq i \leq n$). However, the following two special cases must be considered if the algorithm is to work properly:

1. If a category $C_i$ is a subcategory of some other category $C_j$, where $1 \leq i, j \leq n$, then if the object $o$ is in $C_i$ then the object’s membership in $C_j$ is ignored.

2. It is possible that $o$ belongs to the intersection of two or more categories so that all have a method named $m$ and none of them is a subcategory of any of the others. If this is the case, the late binding algorithm cannot unambiguously determine which procedure or

Rishe-92-DDS p. 10.11
function body should be bound to the name $m$. In order to avoid such problems the data
definition language processor must warn users of such ambiguities if they are detected
within a schema being defined.

The following two examples illustrate the late binding algorithm.

---

**Example 10-16.**

Assume that some application on the University database has referenced the
method name `display`. Since `display` is an overloaded method name, it must be
determined at run time which of the two method bodies defined in the schema is
to be invoked. The algorithm to determine this is the following:

```java
if (o is a STUDENT) then
  (* Bind display to the body of the method display defined on STUDENT *)
else if (o is a PERSON) then
  (* Bind display to the body of the method display defined on PERSON *)
else (* Error: display can not be bound to any method body. *)
```

where $o$ is the object assigned to the first parameter of `display`. The algorithm
first tests $o$ for membership in `STUDENT` and then in `PERSON`. If the test were
performed in the reverse order and $o$ is in `STUDENT`, then the major associated
with $o$ would not be printed.

---

**Example 10-17.**

Assume that a new method has been added to the University application schema.
This new method is also named `display` and it is defined on the category
`INSTRUCTOR`. Given an object in `INSTRUCTOR`, `display` will print the name,
address, and the name of the department where the instructor works.

Consider now a reference to the name `display` by some application program. It is
possible that the object assigned to the first parameter of `display` is a member of
both `STUDENT` and `INSTRUCTOR` (i.e., it lies in the intersection of categories
`STUDENT` and `INSTRUCTOR`). Hence, there are two possible procedure bodies
that can be bound to the name `display`, one defined on `STUDENT` and another
defined on `INSTRUCTOR`. Obviously, the system cannot unambiguously bind
any method body to `display` and it should generate a run-time error message
indicating that.

---

Rishe-92-DDS p. 10.12
10.2. Object-Oriented Terminology

The following are definitions of terms which are common to many object-oriented models.

Attribute — synonymous with the SBOODM term binary relation.

In some object-oriented models (e.g., the $O_2$ model) an object is not only represented by its object identifier but as a grouping of all the facts in which the object is the domain value. We call this an OO-object and it is defined as follows:

**OO-Object** — is a pair $(I, V)$ where $I$ is an object identifier and $V$ is set of pairs of the form $(A:v)$. $A$ is an attribute and $v$ can be a concrete value or an object identifier or the symbol nil. $(A:v)$ means that there exists a fact "$I A v$" in the database.

**Example 10-18.**
The following is an example of an OO-object in the category STUDENT whose object identifier is $o1$.

```
(o1,
  { (FIRST-NAME : 'Roberta'),
    (LAST-NAME : 'Jackson'),
    (BIRTH-YEAR : 1950),
    (ADDRESS : '111 Park Ave., New York, NY, 22233, U.S.A.'),
    (MAJOR : o2),
    (MINOR : o3) }
  )
```

$o2$ and $o3$ are object identifiers for the departments which constitute the major and the minor of the student named Roberta Jackson.

Class — synonymous with the term category.

Class instance — an object is an instance of class C if it is a member of $C$.

Message — a method invocation on some object. In some object-oriented data models, the invocation of a method $m$ on some object $o$ (i.e., ‘‘a message $m$ sent to $o$’’) is denoted as:

```
o.m(parameter-list)
```

Method signature — the specification of a method header.

Class method — synonymous with the term category method.

Instance method — synonymous with the term object method.

Public method — any method which appears in the interface of a category.

Private method — any method which is defined for a category and does not appear in the interface of that category. That is, the method is used by the database designer only as a
subroutine in defining other methods, but the method is not made available to the other users.

**Class type** — is a triple \((N,T,M)\) where

- \(N\) is the name of the class for which the type is defined.
- \(T\) is a set of pairs \((A:C)\) where \(A\) is an attribute and \(C\) is a class name. \((A:C)\) means that objects in class \(N\) can relate through \(A\) with objects in class \(C\).
- \(M\) is a set of methods defined for the class whose name is \(C\).

**is-a relationship** — equivalent to the subcategory-supercategory relationship which may exist between two categories.

**Inheritance** — all the properties, such as methods and attributes, of a category \(C\) automatically apply to all of its subcategories. We say that all the subcategories of \(C\) inherit its properties.

**Class lattice** — a graph showing the inclusion of categories. It is a directed acyclic graph \(H = (V, A)\) where

- \(V\) is a set of nodes, and each node represents a class in a database schema.
- \(A\) is a set of arcs representing is-a relationships between classes. An arc from \(C_1\) to \(C_2\) means that \(C_1\) is a subclass of \(C_2\).

In some object-oriented models, a class cannot be an immediate subclass of more than one class. In such models, \(H\) is a set of trees.

**Reference Schemas**

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
Figure Ref-2. A relational schema for the university application.