Chapter 15

Object-Oriented Database Development

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **atomic literal, collection literal, set, bag, list, array, dictionary, structured literal,** and **extent**.
- Create logical object-oriented database schemas using the object definition language (ODL).
- Transform conceptual UML class diagrams to logical ODL schemas by mapping classes (abstract and concrete), attributes, operations (abstract and concrete), association relationships (one-to-one, one-to-many, and many-to-many), and generalization relationships.
- Identify the type specifications for attributes, operation arguments, and operation returns.
- Create objects and specify attribute values for those objects.
- Understand the steps involved in implementing object-oriented databases.
- Understand the syntax and semantics of the object query language (OQL).
- Use OQL commands to formulate various types of queries.
- Gain an understanding of the types of applications to which object-oriented databases have been applied.

INTRODUCTION

In Chapter 14, we introduced you to object-oriented data modeling. You learned how to conceptually model a database using UML class diagrams. In this chapter, we will describe how such conceptual object-oriented models can be transformed into logical schemas that can be directly implemented using an object database management system (ODBMS).

As you will learn later, although relational databases are effective for traditional business applications, they have severe limitations (in the amount of programming required and DBMS performance) when it comes to storing and manipulating complex data and relationships. In this chapter, we will show how to implement applications within an object-oriented database environment. In Appendix D, you will learn about object-relational databases, which are the most popular way object-oriented principles are implemented in DBMSs.

In this chapter, we will adopt the Object Model proposed by the Object Database Management Group (ODMG) (see **www.odmg.org**) for defining and querying an object-oriented database (OODB). For developing logical schemas, we will specifically use the object definition language (ODL), a data definition language for OODBs specified in the ODMG 3.0 standard (Cattell et al., 2000). The ODMG was formed in 1991 by OODB vendors to create standards, and hence make OODBs more viable. Just as an SQL data definition language (DDL) schema can be implemented in a SQL-compliant relational DBMS (see Chapters 7 and 8), a logical schema created using ODL can be implemented in an ODMG-compliant ODBMS.

We will use examples similar to the ones you saw earlier in Chapter 14 to show how to map conceptual UML class diagrams into logical ODL schemas. You will learn how to map classes (abstract and concrete), attributes, operations (abstract and concrete), association relationships (unary and binary), and generalization relationships from a UML class diagram into corresponding ODL constructs.

Recall from Chapter 7 that, in addition to the DDL component, SQL has a data manipulation language (DML) component that allows users to query or manipulate the data inside a relational database. A similar language, called object query language (OQL), has been specified in the ODMG 3.0 standard to query object databases (Cattell et al., 2000). We will describe how various types of queries can be formulated using OQL. The ODMG has prescribed versions of ODL and OQL for C++, Smalltalk, and Java. We use a generic version in our examples.

In Chapter 14, we showed a conceptual object-oriented model for the Pine Valley Furniture Company. In this chapter, we will transform this conceptual model into a logical ODL schema. Finally, we will discuss the types of applications for which ODBMSs are well suited and describe briefly some of the applications developed using existing ODBMS products.

OBJECT DEFINITION LANGUAGE

In Chapter 7, you learned how to use the SQL DDL to specify a logical schema for a relational database. Similarly, the ODL allows you to specify a logical schema for an object-oriented database. ODL is a programming-language-independent specification language for defining OODB schemas. Just as an SQL DDL schema is portable across SQL-compliant relational DBMSs, an ODL schema is portable across ODMG-compliant ODBMSs.

Defining a Class

Figure 15-1 shows a conceptual UML class diagram (see Chapter 14 for an explanation of the notation for a class diagram) for a university database, an example with which all readers are familiar. For the time being, we will focus on the Student and Course classes and their attribute properties. In ODL, a class is specified using the class keyword, and an attribute is specified using the *attribute* keyword. The Student and Course classes are defined as follows:

```
class Student {
   attribute string name;
   attribute Date dateOfBirth;
   attribute string address;
   attribute string phone;
// plus relationship and operations . . .
};
```

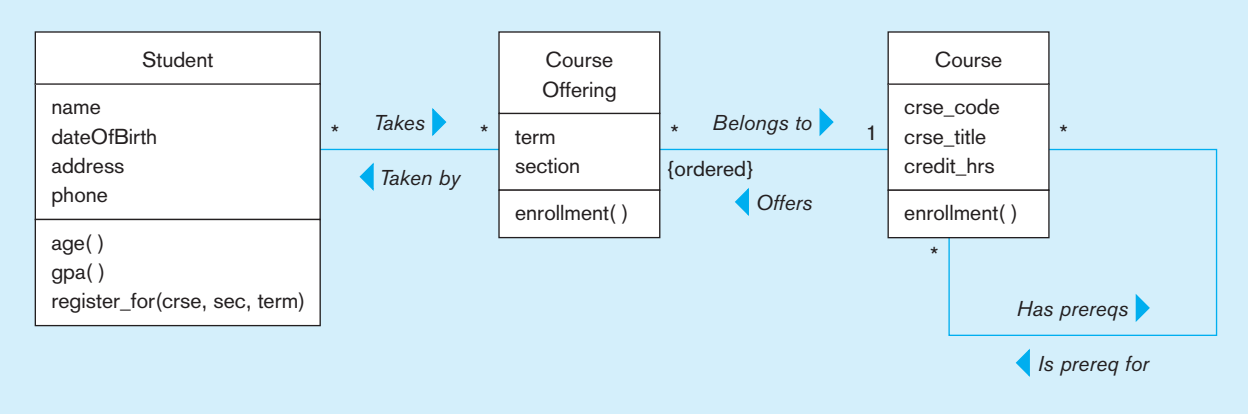


Figure 15-1
UML class diagram for a university database

```
class Course {
   attribute string crse_code;
   attribute string crse_title;
   attribute short credit_hrs;
// plus relationships and operation . . .
};
```

We have highlighted the ODL keywords in bold. We have added comments (preceded by the "//" sign) to indicate that we need to add the relationships and operations (see Figure 15-1) to the classes at a later stage. Note that next to an attribute keyword, we have specified the type of the attribute followed by the attribute's name.

Defining an Attribute

An attribute's value is either a literal or an *object identifier*. As we discussed in Chapter 14, each object has a unique identifier. Because an object retains its identifier over its lifetime, the object remains the same despite changes in its state. In contrast, literals do not have identifiers and, therefore, cannot be individually referenced like objects. Literals are embedded inside objects. You can think of literal values as constants. For example, the string Mary Jones, the character C, and the integer 20 are all literal values.

The Object Model supports different literal types, including atomic literals, collection literals, and structured literals. Examples of **atomic literal** types are string, char (character), boolean (true or false), float (real number), short (short integer), and long (long integer).

A **collection literal** is a collection of elements, which themselves could be of any literal or object type. The collection literal types supported by the ODMG Object Model include *set*, *bag*, *list*, *array*, and *dictionary*. A **set** is an unordered collection of elements of the same type without any duplicates. A **bag** is an unordered collection of elements that may contain duplicates. In contrast to sets and bags, a **list** is an ordered collection of elements of the same type. An **array** is a dynamically sized ordered collection of elements that can be located by position. A **dictionary** is an unordered sequence of key-value pairs without any duplicates.

A **structured literal**, also known as a *structure*, consists of a fixed number of named elements, each of which could be of literal or object type. The Object Model supports the following predefined structures: Date, Interval, Time, and Timestamp. In addition, it supports user-defined structures, examples of which will be given later.

Let us now go back to the ODL schema for the university database. The attributes name, address, and phone of Student and crse_code and crse_title of Course are all of

Atomic literal: A constant that cannot be decomposed into any further components.

Collection literal: A collection of literals or object types.

Set: An unordered collection of elements without any duplicates.

Bag: An unordered collection of elements that may contain duplicates.

List: An ordered collection of elements of the same type.

Array: A dynamically sized ordered collection of elements that can be located by position.

Dictionary: An unordered sequence of key-value pairs without any duplicates.

Structured literal: A fixed number of named elements, each of which could be of literal or object type.

string type. A string attribute can store a string of alphanumeric characters enclosed within double quotes. The attribute credit_hrs of Course is short because its value is always an integer less than 2^{16}. In addition to these atomic literal types, the schema also specifies the structured literal type Date for the dateOfBirth attribute of Student.

Defining User Structures

In addition to the standard data types provided by ODL, you can define structures yourself by using the *struct* keyword. For example, you can define a structure called Address that consists of four components—street_address, city, state, and zip—all of which are string attributes.

```
struct Address {
   string street_address;
   string city;
   string state;
   string zip;
};
```

Similarly, you can define Phone as a structure consisting of an area code and a personal_number. Note that the latter is specified as a long integer, because it is more than 2^{16}.

```
struct Phone {
   short area_code;
   long personal_number;
};
```

Their structures can now be used as the structure for elements of other structures. For example, if a student can have more than one phone number, the phone attribute could be defined as follows:

```
attribute set < phone > phones;
```

Defining Operations

We can also define the operations for the two classes. In ODL, you specify an operation using parentheses after its name. The ODL definition for Student is now as follows:

```
class Student {
   attribute string name;
   attribute Date dateOfBirth;
//user-defined structured attributes
   attribute Address address;
   attribute Phone phone;
//plus relationship
//operations
   short age( );
   float gpa( );
   boolean register_for(string crse, short sec, string term);
};
```

We have defined all three operations shown in Figure 15-1: age, gpa, and register_for. The first two are query operations. The register_for operation is an update operation that registers a student for a section (sec) of a course (crse) in a given term. Each of these arguments, shown within parentheses, is preceded by its type.[1] We also have

[1] If the strict ODL syntax is followed, the argument type has to be preceded by the keyword "in," "out," or "inout," specifying the argument as an input, output, or input/output, respectively. We have chosen not to show this specification for the sake of simplicity.

to specify the return type for each operation. For example, the return types for age and gpa are short (short integer) and float (real number), respectively. The return type for the register_for operation is boolean (true or false), indicating if the registration was successfully completed or not. If the operation does not return any value, the return type is declared as void.

Each type of object has certain predefined operations. For example, a set object has a predefined "is_subset_of" operation, and a data object (attribute) has a predefined boolean operation "days_in_year." See Cattell et al. (2000) for a thorough coverage of predefined object operations.

Defining a Range for an Attribute

If you know all the possible values that an attribute can have, you can enumerate those values in ODL. For example, if you know that the maximum number of sections for a course is eight, you can use the keyword enum before the attribute name (section) within the CourseOffering class and the possible values after the name, as shown in the following:

```
class CourseOffering {
   attribute string term;
   attribute enum section {1, 2, 3, 4, 5, 6, 7, 8};
//operation
   short enrollment( )
};
```

Defining Relationships

Finally, we will add the relationships shown in Figure 15-1 to the ODL schema. The ODMG Object Model supports only unary and binary relationships. There are two binary relationships and one unary relationship in Figure 15-1. As discussed in Chapter 14, a relationship is inherently bidirectional. In Figure 15-1, we have provided names for both directions of a relationship. For example, we have named the relationship between Student and Course Offering *Takes* when traversing from the former to the latter and *Taken by* when traversing in the reverse direction. We use the ODL keyword relationship to specify a relationship.

```
class Student {
   attribute string name;
   attribute Date dateOfBirth;
   attribute Address address;
   attribute Phone phone;
// relationship between Student and CourseOffering
   relationship set < CourseOffering> takes inverse CourseOffering::taken_by;
// operations
   short age( );
   float gpa( );
   boolean register_for(string crse, short sec, string term);
};
```

Within the Student class, we have defined the "takes" relationship, using the relationship keyword. The name of the relationship is preceded by the class the relationship targets: CourseOffering. Because a student can take multiple course offerings, we have used the keyword "set" to indicate that a Student object is related to a set of CourseOffering objects (and the set is unordered). This relationship specification represents the traversal path from Student to CourseOffering.

The ODMG Object Model requires that a relationship be specified in both directions. In ODL, the *inverse* keyword is used to specify the relationship in the reverse direction. The inverse of "takes" is "taken_by" from CourseOffering to Student. In the

class definition for Student, we have named this traversal path (taken_by), preceded by the name of the class from where the path originates (CourseOffering), along with a double colon (::). In the class definition for CourseOffering shown below, the relationship is specified as "taken_by," with the inverse being "takes" from Student. Because a course offering can be taken by many students, the relationship links a set of Student objects to a given CourseOffering object. For a many-to-many relationship such as this, therefore, you must specify a collection (set, list, bag, or array) of objects on both sides.

```
class CourseOffering {
   attribute string term;
   attribute enum section {1, 2, 3, 4, 5, 6, 7, 8};
// many-to-many relationship between CourseOffering and Student
   relationship set <Student> taken_by inverse Student::takes;
// one-to-many relationship between CourseOffering and Course
   relationship Course belongs_to inverse Course::offers;
//operation
   short enrollment( );
};
```

The ODBMS would automatically enforce the *referential integrity* of the relationships you specify in an ODL schema (Bertino and Martino, 1993; Cattell et al., 2000). For instance, if you delete a Student object, the ODBMS will automatically dereference its links to all CourseOffering objects. It will also dereference links from all CourseOffering objects back to that Student object. If you link a Student object to a set of CourseOffering objects, the ODBMS will automatically create inverse links. That is, it will create a link from each of the CourseOffering objects back to the Student object in question.

We have specified another relationship, "belongs_to," for the CourseOffering class, with the inverse being "offers" from Course. Because a course offering belongs to exactly one course, the destination of the belongs_to traversal path is Course, implying that a CourseOffering object can be linked to only one Course object. In specifying the "one" side of an association relationship, therefore, you simply specify the destination object type, not a collection (e.g., set) of an object type.

We show below how to specify the relationships and operation for Course. We have specified the offers relationship within Course, with the inverse being belongs_to. But because the course offerings are ordered within a course (according to section number within a given term), we have used *list*, as opposed to *set*, to denote the ordering within the collection. Both directions of the unary relationship shown in Figure 15-1, has_prereqs and is_prereq_for, begin and terminate in the Course class, as specified in the following Course definition.

```
class Course {
   attribute string crse_code;
   attribute string crse_title;
   attribute short credit_hrs;
// unary relationship for prerequisite courses
   relationship set <Course> has_prereqs inverse Course::is_prereq_for;
   relationship set <Course> is_prereq_for inverse Course::has_prereqs;
// binary relationship between Course and CourseOffering
   relationship list <CourseOffering> offers inverse CourseOffering::belongs_to;
//operation
   short enrollment( );
};
```

Extent: The set of all instances of a class within the database.

The complete schema for the university database is shown in Figure 15-2. Notice that we have introduced the ODL keyword *extent* to specify the extents of the classes. The **extent** of a class is the set of all instances of the class within the database (Cattell

```
class Student {
(     extent students)
      attribute string name;
      attribute Date dateOfBirth;
      attribute Address address;
      attribute Phone phone;
      relationship set ⟨CourseOffering⟩ takes inverse CourseOffering::taken_by;
      short age( );
      float gpa( );
      boolean register_for(string crse, short sec, string term);
};

class CourseOffering {
(     extent courseofferings)
      attribute string term;
      attribute enum section {1, 2, 3, 4, 5, 6, 7, 8};
      relationship set ⟨Student⟩ taken_by inverse Student::takes;
      relationship Course belongs_to inverse Course::offers;
      short enrollment( );
};

class Course {
(     extent courses)
      attribute string crse_code;
      attribute string crse_title;
      attribute short credit_hrs;
      relationship set ⟨Course⟩ has_prereqs inverse Course::is_prereq_for;
      relationship set ⟨Course⟩ is_prereq_for inverse Course::has_prereqs;
      relationship list ⟨CourseOffering⟩ offers inverse CourseOffering::belongs_to;
      short enrollment( );
};
```

Figure 15-2
ODL schema for university database

et al., 2000). For example, the extent called "students" refers to all the Student instances in the database.

Defining an Attribute with an Object Identifier as Its Value

In all the examples that you have seen so far, an attribute's value is always a literal. Recall that we said that it could also be an object identifier. For instance, there could be an attribute called "dept" within Course that represents the department offering a course. Instead of storing the department's name, the attribute could store the identifier for a Department object. We need to make the following changes:

```
class Course {
// the dept attribute's value is an object identifier
   attribute Department dept;
// other attributes, operations, and relationships . . .
};
class Department {
   attribute short dept_number;
   attribute string dept_name;
   attribute string office_address;
};
```

The type of the dept attribute is Department, implying that the attribute's value is an object identifier for an instance of Department. This is akin to representing a unidirectional relationship, from Course to Department. The ODBMS, however,

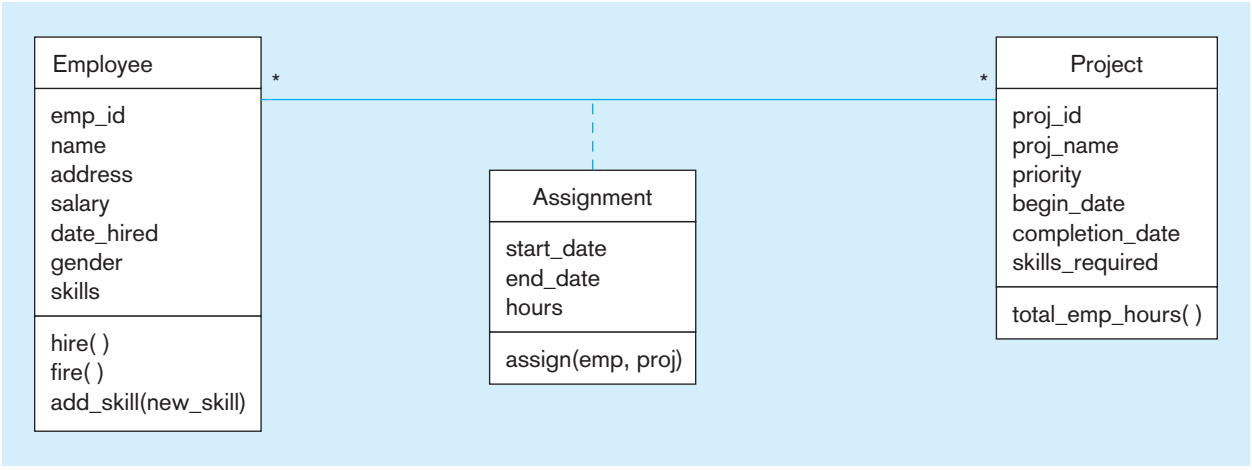
does not automatically maintain the referential integrity of such unidirectional relationships specified through attributes. If most of the references in user queries are from Course to Department (e.g., finding the name of the department offering a given course), not vice versa, and referential integrity is not an issue, then representing such a relationship in one direction using an attribute reference provides a more efficient alternative.

DEFINING MANY-TO-MANY RELATIONSHIPS, KEYS, AND MULTIVALUED ATTRIBUTES

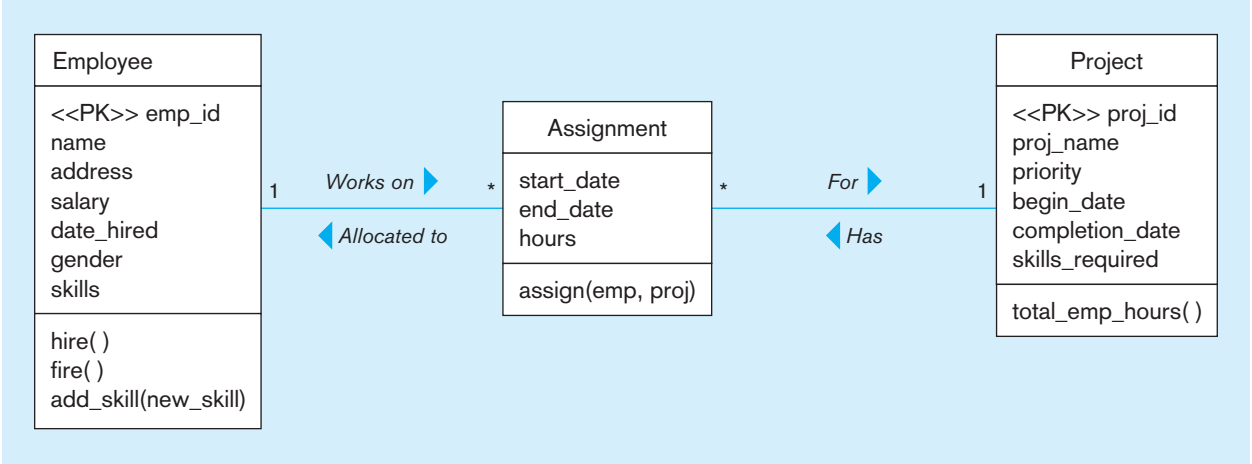
Figure 15-3a shows a many-to-many relationship between Employee and Project. To make an assignment, you need an Employee object, as well as a Project object. Hence, we have modeled Assignment as an association class with features of its own. The features include the start_date, end_date, and hours attributes and the assign operation. In Figure 15-3b, we have broken the many-to-many relationship between Employee and Project into two one-to-many relationships, one from Employee to Assignment and the other from Project to Assignment. Although we could have specified the many-to-many relationship directly in ODL, we could not have captured the features special to Assignment unless we decomposed the relationship into two one-to-many relationships.

Figure 15-3
UML class diagram for an employee project database

(a) Many-to-many relationship with an association class



(b) Many-to-many relationship broken into two one-to-many relationships



The class diagram shown in Figure 15-3b can now be transformed into the following ODL schema:

```
class Employee {
( extent employees
// emp_id is the primary key for Employee
   key emp_id)
   attribute short emp_id;
   attribute string name;
   attribute Address address;
   attribute float salary;
   attribute Date date_hired;
   attribute enum gender {male, female};
// multivalued attribute
   attribute set <string> skills;
   relationship set <Assignment> works_on inverse Assignment::allocated_to;
// the following operations don't return any values
   void hire( );
   void fire( );
   void add_skill(string new_skill);
};
class Assignment {
( extent assignments)
   attribute Date start_date;
   attribute Date end_date;
   attribute short hours;
   relationship Employee allocated_to inverse Employee::works_on;
   relationship Project for inverse Project::has;
// the following operation assigns an employee to a project
void assign (short emp, string proj);
};
class Project {
( extent projects
// proj_id is the primary key for Project
   key proj_id);
   attribute string proj_id;
   attribute string proj_name;
   attribute enum priority {low, medium, high};
   attribute Date begin_date;
   attribute Date completion_date;
//multivalued attribute
   attribute set <string> skills_required;
   relationship set <Assignment> has inverse Assignment::for;
   long total_emp_hours( );
};
```

In the ODL schema, we have specified the candidate keys for Employee and Project using the keyword called *key*. Note that each Employee or Project instance in an object database is inherently unique; that is, you do not require an explicit identifier to enforce the uniqueness of objects. However, specifying a key ensures that no two objects belonging to a class have the same value for the key attribute(s). The scope of uniqueness is confined to the extent of the class. Hence, before specifying a key for a class, you must specify its extent. The emp_id attribute must have a unique value for each Employee object; the same applies to proj_id of Project. ODL also supports compound keys, that is, keys consisting of more than one attribute. For example, if an employee is uniquely identified by name and address, then you could specify the key as follows:

```
keys {name, address}
```

This schema also illustrates how to define a multivalued attribute, an attribute that may have multiple values at a given point in time. The skills attribute of

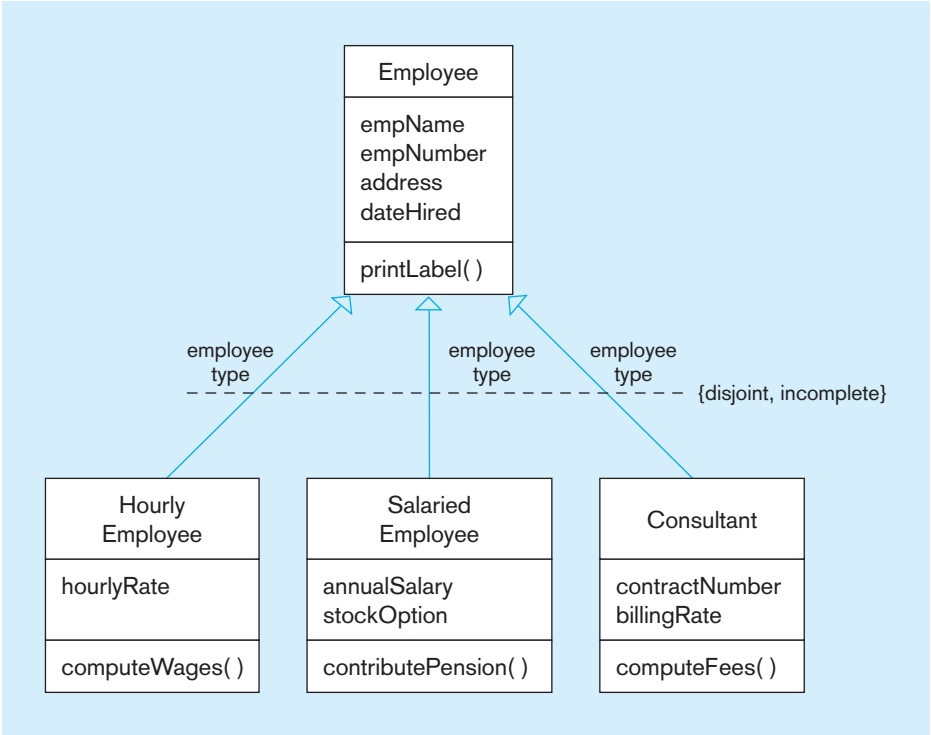


Figure 15-4
UML class diagram showing employee generalization

Employee and the skills_required attribute of Project are each specified as a set of string values.

Defining Generalization

ODL supports unary and binary association relationships, but not relationships of higher degree. It allows you to represent generalization relationships using the *extends* keyword. In Figure 15-4, we show a UML class diagram that you first saw in Chapter 14. Three subclasses—Hourly Employee, Salaried Employee, and Consultant—are generalized into a superclass called Employee. The ODL schema corresponding to the class diagram is given below:

```
class Employee {
( extent employees)
   attribute short empName;
   attribute string empNumber;
   attribute Address address;
   attribute Date dateHired;
   void printLabel( );
};
class HourlyEmployee extends Employee {
( extent hrly_emps)
   attribute float hourlyRate;
   float computeWages( );
};
class SalariedEmployee extends Employee {
( extent salaried_emps)
   attribute float annualSalary;
   attribute boolean stockOptions;
   void contributePension( );
};
```

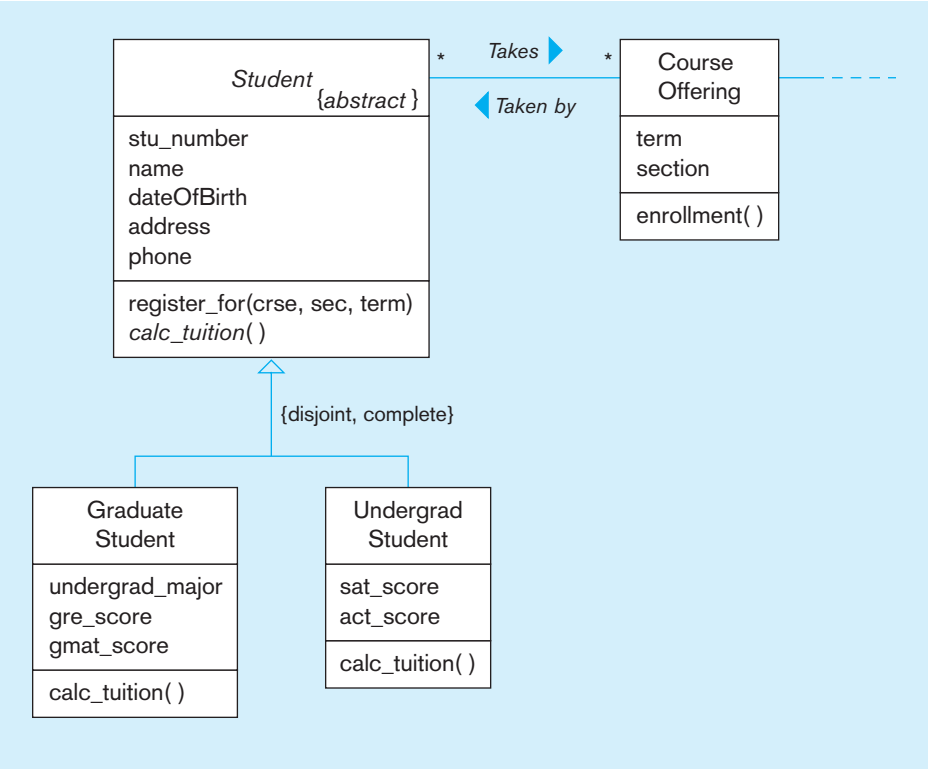


Figure 15-5
UML class diagram showing student generalization

```
class Consultant extends Employee {
( extent consultants)
   attribute short contractNumber;
   attribute float billingRate;
   float computeFees( );
};
```

The subclasses HourlyEmployee, SalariedEmployee, and Consultant extend the more general Employee class by introducing new features. For example, HourlyEmployee has two special features, hourlyRate and computeWages, in addition to the common set of features inherited from Employee. All the classes, including Employee, are concrete, implying that they can have direct instances. Employee is a concrete class because the subclasses are incomplete.

Defining an Abstract Class

Figure 15-5 shows an example of an abstract class called Student, which cannot have any direct instances. That is, a student has to be an instance of Graduate Student or of Undergraduate Student (note the "complete" constraint among the subclasses). In the logical schema, we specify Student as an abstract class as follows:

```
abstract class Student {
( extent students
   key stu_number)
   attribute long stu_number;
   attribute string name;
   attribute Date dateOfBirth;
   attribute Address address;
   attribute Phone phone;
   relationship set <CourseOffering> takes inverse CourseOffering::taken_by;
   boolean register_for(string crse, short section, string term);
```

```
// abstract operation
   abstract float calc_tuition( );
};
```

Notice that the calc_tuition operation of Student is abstract, implying that, at this level, the operation's form is specified but not its implementation. We have used the *abstract* keyword to specify both the abstract class and the abstract operation.[2] The subclasses are defined as follows:

```
class GraduateStudent extends Student {
( extent grads)
   attribute char undergrad_major;
   attribute GRE gre_score;
   attribute GMAT gmat_score;
   float calc_tuition( );
};
class UndergradStudent extends Student {
( extent undergrads)
   attribute SAT sat_score;
   attribute ACT act_score;
   float calc_tuition( );
};
```

Because both of the subclasses are concrete, the calc_tuition operations within them must also be concrete. Therefore, although each subclass inherits the form of the operation from Employee, it still has to provide the method. The calc_tuition operation is specified separately within each subclass, thereby illustrating polymorphism. The fact that it is concrete is indicated by the absence of the *abstract* keyword.

Defining Other User Structures

The schema definition contains user-defined structures such as GRE, GMAT, SAT, and ACT, which specify the types for the various test scores. Although a predefined structure, such as Date or Time, allows you to use it readily for attribute specification, you can define additional structures that could be used for the same purpose. We define the structures for the test scores as follows:

```
struct GRE {
   Score verbal_score;
   Score quant_score;
   Score analytical_score;
};
struct GMAT {
   Score verbal_score;
   Score quant_score;
};
struct SCORE {
   short scaled_score;
   short percentile_score;
};
struct SAT { . . . };
struct ACT { . . . };
```

The GRE structure consists of a verbal score, a quantitative score, and an analytical score, whereas the GMAT structure does not have an analytical score. The type of

[2]ODL does not currently support the abstract keyword for classes and operations. The syntax we have used parallels that of Java, which clearly differentiates an abstract class/operation from a concrete class/operation.

each of these scores, in turn, is a structure called SCORE, which consists of a scaled score (e.g., 680) and a percentile score (e.g., 95 percent).

OODB DESIGN FOR PINE VALLEY FURNITURE COMPANY

In Chapter 14, we developed a conceptual object-oriented model for the Pine Valley Furniture Company in the form of a class diagram (see Figure 14-18). We will now transform this conceptual model into a logical ODL schema, which may be used to implement an object-oriented database system for the company.

The ODL schema is shown in Figure 15-6. By now you should be able to understand clearly how each class, attribute, operation, and relationship in the class diagram has been mapped to an equivalent ODL construct in the logical schema. A few points need some mention, however. The definitions of the Address and Phone structures are not shown in the figure because they were given previously. Notice that the type for the salespersonFax attribute of Salesperson is Phone, indicating that the attribute shares the same type as salespersonTelephone. Also, although the class diagram does not specify any of the collections as ordered, we ordered some of them in the ODL schema. For example, the collection of order lines contained within an Order object is specified as a list, implying that the order line objects are ordered or sorted.

Another thing to note is how we mapped the two many-to-many relationships, each with an association class, into two one-to-many relationships. The first one is between Order and Product with an association class called OrderLine. In the logical schema, we defined a class called OrderLine that participates in two one-to-many relationships, one with Order and the other with Product. The other many-to-many relationship with an association class is Supplies, which was similarly mapped into two one-to-many relationships, one between Supplier and Supplies and the other between RawMaterial and Supplies.

Figure 15-6 (Continues)
ODL schema for
Pine Valley Furniture Company database

```
class Salesperson {
(   extent salespersons
    key salespersonID)
    attribute string salespersonID;
    attribute string salespersonName;
    attribute Phone salespersonTelephone;
    attribute Phone salespersonFax;
    relationship SalesTerritory serves inverse
        SalesTerritory::represented_by;
    float totalCommission( );
};

class SalesTerritory {
(   extent salesterritories
    key territoryID)
    attribute char territoryID;
    attribute char territoryName;
    relationship set⟨Salesperson⟩ represented_by
        inverse Salesperson::serves;
    relationship set⟨Regular Customer⟩ consists_of inverse
        Regular Customer::does_business_in;
};

class Order {
(   extent orders
    key orderID)
    attribute string orderID;
    attribute Date orderDate;
    relationship Customer submitted_by inverse
        Customer::submits;
    relationship list⟨OrderLine⟩ contains inverse
        OrderLine::contained_in;
    float orderTotal( );
};

class OrderLine {
(   extent orderlines)
    attribute short orderedQuantity;
    relationship Order contained_in inverse
        Order::contains;
    relationship Product specifies inverse
        Product::specified_in;
    long orderlineTotal( );
};
```

```
abstract class Customer {
(   extent customers
    key customerID)
    attribute string customerID;
    attribute string customerName;
    attribute Address customerAddress;
    attribute short postalCode;
    attribute float balance;
    attribute enum customerType {National, Regular};
    attribute boolean National;
    attribute boolean Regular;
    relationship list⟨Order⟩ submits inverse
        Order::submitted_by;
    void mailInvoice(float amount);
    void receivePaymt(float amount);
};

class Regular Customer extends Customer {
(   extent reg_custs)
    relationship set⟨Sales Territory⟩
        does_business_in inverse
        Sales Territory::consists of;
};

class National Customer extends Customer {
(   extent nat_custs)
    attribute string acctManager;
};

class WorkCenter {
(   extent materialID
    key workCenterID)
    attribute char workCenterID;
    attribute string workCenterLocation;
    relationship set⟨Product⟩ produces inverse
        Product::produced_in;
    relationship list⟨Union Employee⟩ employs inverse
        Employee::works_in;
};

class RawMaterial {
(   extent rawmaterials
    key materialID)
    attribute string materialID;
    attribute string materialName;
    attribute enum unitOfMeasure {piece, box,
        carton, lb, oz, gallon, litre};
    attribute float standardCost;
    relationship set⟨Product⟩ used_in inverse Product::uses;
    relationship set⟨Supplies⟩ listed_in inverse Supply::lists;
};
```

```
class Product {
(   extent products
    key productID)
    attribute string productID;
    attribute string productDescription;
    attribute char productFinish;
    attribute float standardPrice;
    relationship ProductLine belongs_to inverse
        ProductLine::includes;
    relationship set⟨OrderLine⟩ specified_in
        inverse OrderLine::specifies;
    relationship set⟨WorkCenter⟩ produced_in
        inverse WorkCenter::produces;
    relationship set⟨RawMaterial⟩ uses inverse
        RawMaterial::used_in;
    float totalSales( );
    boolean assignProd(string line);
};

class ProductLine {
(   extent productlines)
    attribute string productLineName;
    relationship list⟨Product⟩ includes inverse
        Product::belongs_to;
    float totalSales( );
};

class Vendor {
(   extent vendors)
    attribute string vendorName;
    attribute Address vendorAddress;
};

class Employee {
(   extent employees
    key employeeID)
    attribute string employeeID;
    attribute string employeeName;
    attribute Address employeeAddress;
    attribute enum employeeType {Management, Union};
    relationship set⟨Skill⟩ has inverse Skill::possessed_by;
    boolean checkSkills(string product);
};

class Skill {
(   extent skills);
    attribute string skillName;
    relationship set⟨Employee⟩ possessed_by inverse
        Employee::has;
};
```

Figure 15-6 (Continues)

```
class Supplies {
(   extent supplies)
    attribute float supply UnitPrice;
    relationship RawMaterial lists inverse
        RawMaterial::listed_in;
    relationship Supplier provided_by inverse
        Supplier::provides;
};

class Supplier extends Vendor {
(   extent Suppliers)
    attribute short contractNumber
    relationship set⟨Supplies⟩ provides inverse
    Supplies::provided_by;
}
```

```
class Union Employee extends Employee {
(   extent union_emps)
    relationship set⟨WorkCenter⟩ works_in inverse
        Work Center::employs;
    relationship Management Employee supervised_by inverse
        Management Employee::supervises;
};

class Management Employee extends Employee {
(   extent mgmt_emps)
    relationship set⟨Union Employee⟩ supervise inverse
        Union Employee::supervised_by;
};
```

Figure 15-6 (Continued)

CREATING OBJECT INSTANCES

When a new instance of a class is created, a unique object identifier is assigned. You may specify an object identifier with one or more unique tag names. For example, we can create a new course object called MBA 669 as follows:

MBA669 **course** ( );

This creates a new instance of Course. The object tag name, MBA699, can be used to reference this object. We have not specified the attribute values for this object at this point. Suppose you want to create a new student object and initialize some of its attributes.

Cheryl **student** (**name:** "Cheryl Davis", **dateOfBirth:** 4/5/77);

This creates a new student object with a tag name of Cheryl and initializes the values of two attributes. You can also specify the values for the attributes within a structure, as in the following example:

Jack **student** (
name: "Jack Warner", **dateOfBirth:** 2/12/74,
address: {street_address "310 College Rd", city "Dayton", state "Ohio", zip 45468},
phone: {area_code 937, personal_number 228–2252});

For a multivalued attribute, you can specify a set of values. For example, you can specify the skills for an employee called Dan Bellon as follows:

Dan **employee** (**emp_id:** 3678, **name:** "Dan Bellon",
skills: {"Database design", "OO Modeling" });

Establishing links between objects for a given relationship is also easy. Suppose you want to store the fact that Cheryl took three courses in fall 1999. You can write:

Cheryl **student** (**takes:** {OOAD99F, Telecom99F, Java99F });

where OOAD99F, Telecom99F, and Java99F are tag names for three course-offering objects. This definition creates three links for the "takes" relationship, from the object tagged Cheryl to each of the course offering objects.

Consider another example. To assign Dan to the TQM project, we write:

assignment (**start_date:** 2/15/2001, **allocated_to:** Dan, for TQM);

Notice that we have not specified a tag name for the assignment object. Such objects will be identified by the system-generated object identifiers. The assignment

object has a link to an employee object (Dan) and another link to a project object (TQM).

When an object is created, it is assigned a lifetime, either transient or persistent. A transient object exists only while some program or session is in operation. A persistent object exists until it is explicitly deleted. Database objects are almost always persistent.

OBJECT QUERY LANGUAGE

We will now describe the Object Query Language (OQL), which is similar to SQL-92 and has been set forth as an ODMG standard for querying OODBs. OQL allows you a lot of flexibility in formulating queries. You can write a simple query such as

Jack.dateOfBirth

which returns Jack's birth date, a literal value, or

Jack.address

. . . which returns a structure with values for street address, city, state, and zip. If instead we want to simply find in which city Jack resides, we can write

Jack.address.city

Like SQL, OQL uses a select-from-where structure to write more complex queries. Consider, for example, the ODL schema for the university database given in Figure 15-2 . We will see how to formulate OQL queries for this database. Because of the strong similarities between SQL and OQL, the explanations in the following sections are quite brief. For further explanations, you may want to review Chapters 7 and 8 on SQL and Chapter 14 on object modeling. The more interested reader is referred to the chapter on OQL in Cattell et al. (2000).

Basic Retrieval Command

Suppose we want to find the title and credit hours for MBA 664. Parallel to SQL, those attributes are specified in the select clause, and the *extent* of the class that has those attributes is specified in the from clause. In the where clause, we specify the condition that has to be satisfied. In the query shown below, we have specified the extent courses of the Course class and bound the extent to a variable called c in the from clause. We have specified the attributes crse_title and credit_hrs for the extent (i.e., set of all Course instances in the database) in the select clause, and stated the condition c.crse_code = "MBA 664" in the where clause.

```
select c.crse_title, c.credit_hrs
from courses c
where c.crse_code = "MBA 664"
```

Because we are dealing with only one extent, we could have left out the variable c without any loss in clarity. However, as with SQL, if you are dealing with multiple classes that have common attributes, you must bind the extents to variables so that the system can unambiguously identify the classes for the selected attributes. The result of this query is a bag with two attributes.

Including Operations in Select Clause

We can invoke operations in an OQL query similar to the way we specify attributes. For example, to find the age of John Marsh, a student, we invoke the age operation in the select clause.

```
select s.age
from students s
where s.name = "John Marsh"
```

The query returns an integer value, assuming that there is only one student with that name. In addition to literal values, a query can also return objects with identity. For example, the query

```
select s
from students s
where s.gpa >= 3.0
```

returns a collection (bag) of student objects for which the gpa is greater than or equal to 3.0. Notice that we have used the gpa operation in the where clause.

If we want to formulate the same query, but only for those students who do not reside in Dayton, we can use the not operator as in SQL:

```
select s
from students s
where s.gpa >= 3.0
and not (s.address.city = "Dayton")
```

Instead of using "not," we could have specified the new condition as follows:

```
s.address.city ! = "Dayton"
```

where ! is the inequality operator.

Now suppose that we want to find the ages of all students whose gpa is less than 3.0. This query is

```
select s.age
from students s
where s.gpa < 3.0
```

Finding Distinct Values

The preceding query will return a collection of integers. It is possible that there is more than one student with the same age. If you want to eliminate duplicates, you can reformulate the query using the distinct keyword as shown below:

```
select distinct s.age
from students s
where s.gpa < 3.0
```

Querying Multiple Classes

In an OQL query, you can join classes in the where clause as in SQL. This is necessary when the relationship that is the basis for the join has not been defined in the object data model. When the relationship has been defined, then you can traverse the paths for the relationships defined in the schema. The following query finds the course codes of all courses that were offered in fall 2005.

```
select distinct y.crse_code
from courseofferings x,
   x.belongs_to y
where x.term = "Fall 2005"
```

We have used *distinct* in the select clause because a course may have had multiple offerings in the given term. In the from clause, we have specified a path from a CourseOffering object to a Course object using the belongs_to relationship between them. The variable y gets bound to the Course object where the path represented by x.belongs_to terminates.

Suppose that we want to find the number of students enrolled in section 1 of the MBA 664 course. The enrollment operation is available in CourseOffering, but the course code is available in Course. The query given below traverses from CourseOffering to Course using the belongs_to relationship. The variable y represents the destination object for the x.belongs_to path.

```
select x.enrollment
from courseofferings x,
   x.belongs_to y
where y.crse_code = "MBA 664"
and x.section = 1
```

The following query traverses two paths, one using the takes relationship and the other using the belongs_to relationship, to find the codes and titles of all courses taken by Mary Jones.

```
select c.crse_code, c.crse_title
from students s
s.takes x,
   x.belongs_to c
where s.name = "Mary Jones"
```

We can also select a structure consisting of multiple components. For example, the following query returns a structure with age and gpa as its attributes.

```
select distinct struct(name: s.name, gpa: s.gpa)
from students s
where s.name = "Mary Jones"
```

Writing Subqueries

You can use a select statement within a select statement. To select course codes, course titles, and course offerings for which the enrollment is less than twenty, you can write the following OQL command. (See Figure 15-2 for the design of the database.)

```
select distinct struct (code: c.crse_code, title: c_crse_title,
(select x
from c.offers x
where x.enrollment < 20 ))
from courses c
```

Recall that enrollment is an operation of a CourseOffering object and Course has a *1:M* relationship offers with CourseOffering. This query returns a collection of distinct structures, each of which contains string values for course code and course title, and an object identifier for a CourseOffering object that has enrollment below twenty.

You can also use a select statement within the from clause. In the example below, we have written a query that retrieves the names, addresses, and gpas for those students over age thirty with a gpa greater than or equal to 3.0.

```
select x.name, x.address, x.gpa
from (select s from students s where s.gpa > = 3.0) as x
where x.age > 30
```

Here x is the alias for the extent created by the select statement within the from clause.

Calculating Summary Values

OQL supports all the aggregate operators that SQL does: count, sum, avg, max, and min. For example, we can find the number of students in the university by using the count operator as follows:

```
count(students)
```

We could have also written this query as

```
select count (*)
from students s
```

Let us now consider the schema for the employee-project database that we saw earlier (see Figure 15-3). Suppose we want to find the average salary of female employees in the company. We use the avg function to do that in the following query:

```
select avg_salary_female: avg (e.salary)
from employees e
where e.gender = female
```

To find the maximum salary paid to an employee, we use the max function:

```
max (select salary from employees)
```

To find the total of all employee salaries, we use the sum function:

```
sum (select salary from employees)
```

Calculating Group Summary Values

As in SQL, you can partition a query response into different groups. In the following query, we have used the group command to form two groups based on gender: male and female. The query calculates the minimum salary for each of the two groups.

```
select min (e.salary)
from employees e
group by e.gender
```

If we want to group the projects based on their priority levels, we can write the following query:

```
select *
from projects p
group by
   low:      priority = low,
   medium:   priority = medium,
   high:     priority = high
```

This query returns three groups of project objects, labeled by the priority of the group: low, medium, and high.

Qualifying Groups As with SQL, we can use the having command to impose a condition or filter on each group as a whole. For example, in the following query, we filter only those groups for which a total of more than 50 hours have been logged in.

```
select *
from projects p
group by
   low:      priority = low,
   medium:   priority = medium,
   high:     priority = high
having sum(select x.hours from p.has x) > 50
```

Using a Set in a Query

Sometimes you will have to find whether an element belongs to some set. To do that, you should use the *in* keyword. Suppose we want to find the IDs and names of those employees who are skilled in database design or object-oriented modeling. Note that skills is a multivalued attribute, implying that it stores a set of values. In the where clause,

we have used "in" to determine whether database design or object-oriented modeling is one of the elements in an employee's skill set.

```
select emp_id, name
from employees
where "Database Design" in skills
or "OO Modeling" in skills
```

Similarly, we can find those employees who have worked in a project whose ID is TQM9.

```
select e.emp_id, e.name
from employees e,
      e.works_on a,
      a.for p
where "TQM9" in p.proj_id
```

To find those projects that do not require C ++ programming skills, we can write

```
select *
from projects p
where not ("C ++ Programming" in p.skills_required)
```

Finally, you can use the existential quantifier *exists* and the universal quantifier *for all*. The following query finds those employees who have been assigned to at least one project.

```
select e.emp_id, e.name
from employees e
where exists e in (select x from assignments y
      y.allocated_to x)
```

The select statement inside the where clause returns a set of employee objects (i.e., their identifiers) allocated to all the assignments. The exists quantifier then checks whether an employee object bound in the from clause is in that set. If so, that employee's ID and name are included in the response, otherwise not.

If we want to find the employees who have worked only on projects starting since the beginning of 2005, we can use the *for all* quantifier as follows:

```
select e.emp_id, e.name
from employees e,
      e.works_on a
where for all a: a.start_date >= 1/1/2005
```

In the from clause, the query finds a set of assignment objects that an employee object is linked to through the works_on relationship. In the where clause, it applies the condition (start_date > = 1/1/2001) to all the objects in the set using the for all quantifier. Only if the condition is satisfied by all those objects are the employee's ID and name included in the query response.

Summary of OQL

We have illustrated in this section only a subset of the capabilities of OQL. See Cattell et al. (2000) and Chaudhri and Zicari (2001) for more standard OQL features and how OQL is implemented in various ODBMSs.

CURRENT ODBMS PRODUCTS AND THEIR APPLICATIONS

With the growing need in organizations to store and manipulate complex data (e.g., image, audio, and video) and relationships, in applications ranging from computer-aided design and manufacturing (CAD/CAM) to geographic information systems to multimedia, ODBMS products are gaining popularity. But more than

anything else, industry analysts believe that Internet and Web-based applications are responsible for the sudden renewed interest in ODBMSs (King, 1997; Watterson, 1998). ODBMSs are certainly not overtaking RDBMSs, but they are viable products for selected applications.

ODBMSs allow organizations to store diverse components (objects) associated with their Web sites (Watterson, 1998). The proliferation of complex data types on the Web and the need to store, index, search, and manipulate such data have provided ODBMS technology an edge over other database technologies. To counter this emerging technology, major relational database vendors such as Oracle, Informix, IBM, and Sybase have come up with universal databases, also known as object-relational DBMSs (ORDBMSs), as a possible alternative. An ORDBMS is a hybrid relational DBMS that somehow incorporates complex data as objects (King, 1997; also see Appendix D). However, these systems raise concerns relating to performance and scalability. Moreover, the fundamental mismatch between relational and object technology may induce many firms to adopt the pure ODBMS option.

The types of applications for which ODBMSs are particularly useful include bill-of-materials data (see Figure 14-16), telecommunications data supporting navigational access, health care, engineering design, finance and trading, multimedia, and geographic information systems. Several commercial ODBMS products are currently available. Examples include ObjectStore, Versant ODBMS, GemStone, Objectivity, POET Object Server, and NeoAccess (see Table 15-1).

Watterson (1998) and Barry & Associates (**http://www.service-architecture.com/object-oriented-databases/**) provide several examples of real-world applications of ODBMSs. Lucent Technologies' Customer Support Division used GemStone to share information globally on customers' switches. Motorola used Objectivity to store complex celestial information for one of its satellite networks. Groupe Paradis, a retirement-plan management company in France, uses O_2 to access over 100 gigabytes of data spread across several databases. And companies such as GTE, Southwest Airlines, and Time Warner have used ObjectStore to develop dynamic Web applications that require integrating pieces of information from various sources on the fly. The Chicago Stock Exchange uses Versant ODBMS for its Internet-based trading system.

Industry experts predict that ODBMSs represent the most promising of the emerging database systems. While for traditional business applications, relational DBMSs are expected to maintain their hold on the market, the data for many applications, such as the ones just described, cannot be easily flattened to two-dimensional database tables. Also, accessing the data from various tables requires you to perform joins, which could become very costly.

Table 15-1 ODBMS Products

Company	*Product*	*Web site*
GemStone Systems	GemFire	**www.gemstone.com**
Objectivity	Objectivity/DB	**www.objectivity.com**
Versant	Versant Object Database	**www.versant.com**
Other Links Related to ODBMS Products		
Barry & Associates		**www.odbmsfacts.com**
Doug Barry's *The Object Database Handbook*		**wiley.com**
Object database newsgroup		**news://comp.databases.object**
Rick Cattell's *The Object Database Standard ODMG 3.0*		**www.mkp.com**
Object Database Management Group		**www.odmg.org**
Chaudhri and and Zicarl's *Succeeding with Object Databases*		**www.wiley.com/compbooks/chaudhri**

Summary

In this chapter, you learned how to implement an object-oriented database system using the object definition language. We introduced you to the syntax and semantics of ODL. You learned how to transform a conceptual schema, represented in the form of a UML class diagram, to a logical schema, defined using ODL constructs. You also learned how to populate an OODB by creating new instances and specifying attribute values for those instances. We also introduced you to OQL, a language designed for querying OODBs. Using OQL, we showed you how to write various types of OODB queries.

The chapter also discussed the types of applications for which ODBMSs are well suited. It briefly described some of the applications for which current ODBMSs have been used.

While Chapter 14 provided you with the conceptual underpinnings of OODB design, this chapter provides you with the knowledge required to actually implement an object-oriented database system using an ODMG-compliant ODBMS.

Chapter Review

Key Terms

Array
Atomic literal
Bag
Collection literal
Dictionary
Extent
List
Set
Structured literal

Review Questions

1. Define each of the following terms:
   a. object class
   b. atomic literal
   c. relationship
   d. structured literal
   e. extent
2. Contrast the following terms:
   a. list; bag; dictionary
   b. set; array
   c. collection literal; structured literal
3. Explain the concept of an object identifier. How is an object identifier different from a primary key in a relational system?
4. What is the purpose of the struct keyword in ODL?
5. What is the purpose of the enum keyword in ODL?
6. Explain the meaning of the term relationship set for many of the relationships in Figure 15-2.
7. Explain the hazards of representing a relationship in ODL by implying that an attribute's value is an object identifier rather by using the relationship clause.
8. Explain the meaning of the extends keyword in ODL.
9. Explain the parallels and differences between SQL and OQL.
10. Explain how a many-to-many relationship is represented using an ODL schema.
11. Explain how multivalued attributes are handled in ODL.
12. Explain how to define an abstract class in ODL.
13. When is it necessary to join classes in the where clause of OQL when querying multiple classes?
14. What can be done in an OQL group by clause that cannot be done in SQL?
15. Explain how a unary relationship is represented using ODL.
16. What does the following OQL query return?

```
select distinct struct (custid: x.cid, balance: x.bal)
from customers x
where x.state = "MA"
```

17. What type of object does the following OQL query return?

```
select a
from items a
where price >= 19.99
```

18. When creating an object instance in ODL, how do you specify multivalued attributes?
19. Explain how to write ODL commands to establish links between objects (i.e., object instances) for a given relationship.
20. Perform some research on the Internet on OODBMS products. A good starting place is the links provided at the end of the chapter. Compare various OODBMSs currently on the market in terms of features, capacity, and scalability. How do they compare with RBMS products?

Problems and Exercises

1. Develop an ODL schema for the following problem situation. A student, whose attributes include studentName, Address, phone, and age, may engage in multiple campus-based activities. The university keeps track of the number of years a given student has participated in a specific activity and, at the end of each academic year, mails an activity report to the student showing his participation in various activities.
2. Develop an ODL schema for a real estate firm that lists property for sale. The following describes this organization:
   - The firm has a number of sales offices in several states; location is an attribute of sales office.
   - Each sales office is assigned one or more employees. Attributes of employee include employeeID and employeeName. An employee must be assigned to only one sales office.
   - For each sales office, there is always one employee assigned to manage that office. An employee may manage only the sales office to which he or she is assigned.
   - The firm lists property for sale. Attributes of property include propertyName and location.
   - Each unit of property must be listed with one (and only one) of the sales offices.
   - A sales office may have any number of properties listed, or may have no properties listed.
   - Each unit of property has one or more owners. Attributes of owner are ownerName and address. An owner may own one or more units of property. For each property that an owner owns, an attribute called percentOwned indicates what percentage of the property is owned by the owner.
3. Develop an ODL schema for some organization that you are familiar with—Boy Scouts/Girl Scouts, sports team, and so on. Include at least four association relationships.
4. Develop an ODL schema for the following situation (state any assumptions you believe you have to make in order to develop the schema): Stillwater Antiques buys and sells one-of-a-kind antiques of all kinds (e.g., furniture, jewelry, china, and clothing). Each item is uniquely identified by an item number and is also characterized by a description, asking price, condition, and open-ended comments. Stillwater works with many different individuals, called clients, who sell items to and buy items from the store. Some clients only sell items to Stillwater, some only buy items, and some others both sell and buy. A client is identified by a client number and is also described by a client name and client address. When Stillwater sells an item in stock to a client, the owners want to record the commission paid, the actual selling price, sales tax (tax of zero indicates a tax exempt sale), and date sold. When Stillwater buys an item from a client, the owners want to record the purchase cost, date purchased, and condition at time of purchase.

Problems and Exercises 5 through 13 all pertain to the ODL schema in Figure 15-2. Write OQL queries for these exercises.

5. Find the names and phone numbers of all those students who took only one course in fall 2003.
6. Find the code and titles of all courses that were offered in both the winter 2003 and fall 2003 terms.
7. Find the total enrollment for all sections of the MBA 664 course being offered in winter 2004.
8. Find the prerequisite courses (code and title) for the MIS 385 course.
9. Find all those students who reside in Cincinnati and who took the MBA 665 course in fall 2001.
10. Find the total credit hours for all prerequisite courses for the MIS 465 course.
11. Find the average age and gpa of students, grouped by the city they live in.
12. Find the names and phone numbers of all students who are taking sections 1 and 2 of the MBA 665 course in winter 2004.
13. Find the minimum enrollment among all the sections of each three-credit-hour course being offered in the winter 2004 term. The response should by grouped by course, and the group label should be crse_code.

Problems and Exercises 14 through 21 all deal with the Pine Valley Furniture Company ODL schema in Figure 15-6. Write OQL queries for each of these exercises.

14. List all products with an ash finish.
15. List the customers who live in California or Washington. Order them by postalCode, from high to low.
16. Determine the average standard price of each product line.
17. Which employees were hired during 1999?
18. For every product that has been ordered, determine the total quantity that has been ordered. List the most popular product first and the least popular last.
19. Produce a list of customers for each sales territory.
20. List the total sales of each product.
21. List the total sales for each work center.

Field Exercises

1. Interview a database administrator in a company with which you are familiar. Ask this person to explain the potential benefits of an ODBMS for that organization. Are they planning to use an ODBMS? Why or why not?
2. Using Table 15-1 and your own Internet searches, visit several sites for vendors of ODBMSs. Prepare a summary of one of the products you find. How does its DDL compare to the ODL in this chapter? How does its query language compare to the OQL explained in this chapter? What claims does the vendor make about the relative advantages of its product versus other ODBMS or relational products?

References

Bertino, E., and L. Martino. 1993. *Object-Oriented Database Systems: Concepts and Architectures.* Wokingham, England: Addison-Wesley.

Cattell, R. G. G., D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. (Eds.) 2000. *The Object Database Standard: ODMG 3.0.* San Francisco: Morgan Kaufmann.

Chaudhri, A. B., and R. Zicari. 2001. *Succeeding with Object Databases.* New York: Wiley.

King, N. H. 1997. "Object DBMSs: Now or Never." *DBMS* 10,7 (July): 42–99.

Watterson, K. 1998. "When It Comes to Choosing a Database, the Object Is Value." *Datamation* 44,1 (December–January): 100–107.

Further Reading

Atkinson, M, F. Bacnilhon, D. DeWitt, K. Dittich, D. Maier, and S. Zdonik. 1995. "The Object-Oriented Database System Manifesto." available at **http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html.**

Leavitt Communications. 2003. "Whatever Happened to Object-Oriented Databases." available at **www.leavcom.com/db_08_00 .htm.**

Web Resources

www.cai.com/products/jasmine/analyst/idc/14821Eat.htm This bulletin discusses the changes and innovations currently shaping database technology and related products. It includes a summary of 1996 when a trend toward multimedia-type database product rollouts and a new extended version of relational database technology emerged that was dubbed the "object relational" database management system (ORDBMS). The ORDBMS technology (see Appendix D) is compared with the relational databases from which ORDBMS is evolving and with the pure object databases that they will never replace. What challenges did you face in completing this task?