CHAPTER

2

Mapping UML Diagrams to Object-Relational Schemas in Oracle 8

The recent emergence of object-oriented design tools and object-relational database technology into the commercial market has caused new challenges for the implementation of conceptual database designs. The process of database design typically involves the development of a conceptual schema that uses a high-level conceptual modeling language, such as the Extended Entity-Relationship (EER) model [Elmasri 2000]. The conceptual design is then mapped to the model of the target database system that is to be used for implementation of the application. Although well-defined mapping procedures exist for mapping EER and EER-like schemas into relational database designs, very little work has been done to define the mapping of object-oriented schemas, such as class diagrams in UML [Fowler 1997], into object-relational database schemas.

The Unified Modeling Language (UML) is a standard modeling language that provides a graphical notation to represent the design of object-oriented software. UML is the integration of popular object-oriented analysis and design methods, resulting from a wealth of research in this area in the late 1980s and early 1990s. The Object Management Group (OMG) is the organization that is leading the standardization process [OMG 1999]. Object-relational technology [Stonebraker 1996], on the other hand, extends the traditional notion of the relational table with the notion of object tables, where the tuples of an object table generate object identifiers as in the instances of classes in object-oriented database systems. Relationships between tables are then established through the use of object references. Path expressions can be also incorporated into SQL queries to provide a more concise, object-oriented approach to query expression. Furthermore, object-relational models incorporate the notion of user-defined types, or abstract data types, in addition to providing built-in support for large objects and aggregate types such as sets, arrays, and nested tables. As a result, attributes

within an object table or traditional relational table are no longer limited to atomic values. The presence of these features requires a new look at how to use object-relational concepts effectively in the implementation of object-oriented conceptual designs.

This chapter presents our experience with using the Oracle 8 object-relational model [Oracle 1999] in the implementation of UML class diagrams. The results presented are based on our initial work with using object-relational database concepts within an engineering application [Tjahjadi 1997]; [Urban 2000]. In particular, the work in "The Design and Implementation of an EXPRESS to Oracle 8 Mapping" and "A Case Study in Mapping Conceptual Designs to Object-Relational Schemas" provides a detailed description of mapping EXPRESS conceptual schemas into object-relational schemas in Oracle 8, where EXPRESS is a standard for conceptual modeling within the engineering design domain [Schenck 1994]. Since UML has become an industry standard for expressing object-oriented designs, this chapter revisits many of these mapping issues in the context of UML to address a broader audience of database designers. Although there are differences between the object-relational models that are currently available by different database vendors, the mappings should provide a model of mapping techniques that can be modified for use in systems other than Oracle 8. We are currently teaching these mapping issues in an advanced undergraduate database course at Arizona State University that covers object-relational database technology [CSE494DB 2000].

The work presented in this chapter, together with the original work in "The Design and Implementation of an EXPRESS to Oracle 8 Mapping" on which it is based, provides one of the first descriptions of conceptual-to-object-relational mapping procedures. In "Object Database Development: Concepts and Principles" [Embley 1997], Embley provides a brief overview of a case study that maps a database application designed using the Object-Oriented Systems Model into the UniSQL object-relational system. Specific details of how to use object-relational features, however, are not provided. Grimes [Grimes 1998] also provides an overview of commercial tools that support the generation of object-relational schemas for systems such as Oracle 8, the IBM DB2 Universal Database, and the Informix-Dynamic Server. As described in "Modeling Object/Relational Databases," however, most of these tools are still immature and functionally incomplete. Furthermore, a significant problem with these existing tools is that they do not provide sufficient methodological support to help the user understand how to use object-relational features. Rational Software Corporation, for example, provides a tool for generating Oracle 8 schemas from UML diagrams [Rational 1998]. The user, however, must already understand how object-oriented modeling concepts map to object-relational features in order to construct UML diagrams with appropriate Oracle 8 class stereotypes. The stereotypes provide Rational Rose with instructions on how to generate the Oracle 8 schema. In fact, Grimes states [Grimes 1998] that object-relational database systems have not yet entered the mainstream and, at this point, are more likely to be used by advanced users that are willing to skip conceptual modeling. The mapping procedures described in this chapter represent an initial step towards defining a methodology for generating object-relational schemas from object-oriented designs such as those provided by UML.

related work on mapping to object-relational

¿cuál es la situación en el 2004?

The following sections provide an overview of UML notation as well as a comparison of UML to EER notation. A sample application is also presented to support examples of mapping issues. After providing an overview of the object-relational features supported by Oracle 8, we then elaborate on the generation of object-relational

schemas from UML class diagrams. The chapter concludes with a summary and discussion of future work.

Overview of UML

UML has become an industry standard for expressing object-oriented designs. Modeling object-oriented software is complex and UML provides the capability to express various relationships within an object-oriented software system. The *conceptual class diagrams* (also called static view diagrams) of UML describe the types of objects and the relationships between objects. Although the UML methodology was developed for software designs, UML conceptual class diagrams are similar to that of EER diagrams [Elmasri 2000], which are used for conceptual database design. Table 2.1 compares the terminology of the EER and the UML graphical models.

To support the comparison, Figure 2.1 presents an EER diagram of a School enterprise application [Tapia 1999], and Figure 2.2 presents the same enterprise in UML notation. The School enterprise provides an example that will be used throughout the rest of this chapter to illustrate UML-to-object-relational mapping issues. As shown in Figure 2.1, the School database has five entities that are enclosed in rectangles: Person, Faculty, Student, Department, and Campus Club. Attributes in the EER diagram are enclosed in ovals and attached to the entities that they describe. Since Student and Faculty are disjoint specializations of Person, as indicated by the letter "d" in the specialization circle, Student and Faculty inherit the attributes of the class Person. The double edge of the specialization of Person into its subclasses represents a total specialization constraint, which requires a Person to belong to one of its subclasses. The diamonds in an EER diagram represent relationships between entities. The numeric annotations on the edges linking relationships to the entities to which they relate are cardinality ratios, indicating the number of entities that can be involved in the relationship. For example, the Clubs relationship specifies that a Student can be a member of several Campus Clubs and a Campus Club can have several student members. The Advises relationship indicates that a Campus Club can have a Faculty member as an advisor and a Faculty member can advise many Campus Clubs. A double edge linking an entity to a relationship indicates a total participation constraint and requires that an entity be

Table 2.1 Comparison of EER and UML Terminology

EER	UML
Entity	Class
Relationship	Association
Attributes	Attributes
Roles	Roles
Structural Constraints	Multiplicity
Generalization & Specialization	Generalization & Classification

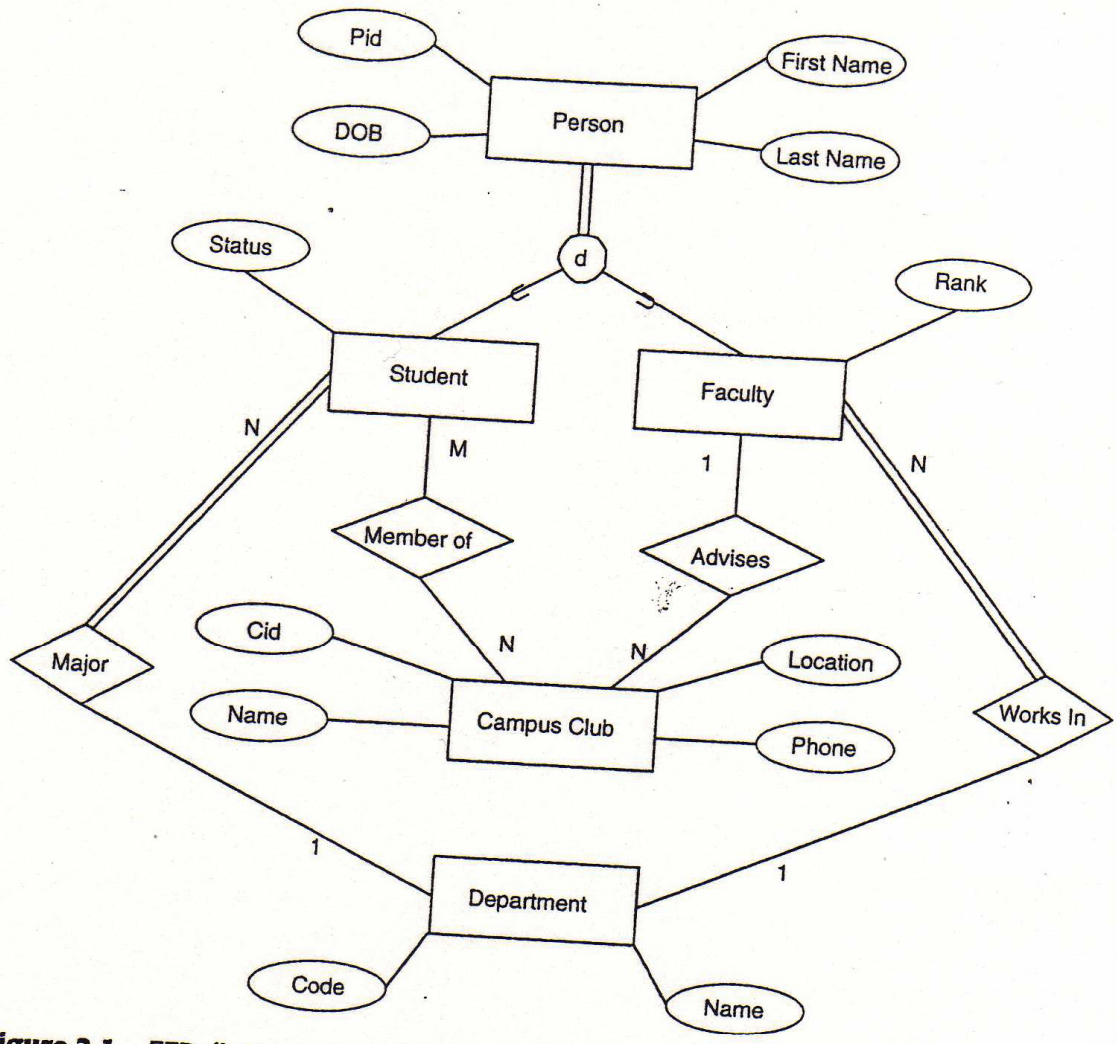

Figure 2.1 EER diagram of the School Database.

involved in at least one relationship instance. For example, each Student and Faculty member must be associated with a Department through its Major and WorksIn relationships, respectively. These relationships also relate a Department to the Students majoring in the Department and the Faculty working in the Department.

In UML, a class is an entity that is described by its attributes and operations. Graphically, a class is enclosed in a rectangle and its attributes and operations are shown in compartments listed after the name of the class, as shown in Figure 2.2. An attribute definition optionally can include a type and default value, in the format *attributeName:type=defaultValue*. Operations on a UML class are listed in a compartment following the attribute specifications. For brevity of presentation, Figure 2.2 shows the names of two operations for the Department class: GetStudents and GetFaculty. These operations return the students majoring in the department and the faculty working in the department, respectively. The operations for the other classes are omitted. An operation optionally can be specified in the format *methodName (parameterList) : returnTypeExpression*. From a database perspective, the operations on a conceptual class diagram should correspond to the public interface of the class to facilitate schema definition.

An association in UML is a binary relationship. Graphically, an association is indicated by an undirected edge between the two classes that it associates. Figure 2.2 provides the Advises association name for the relationship between Faculty and CampusClub. In a similar manner, the relationship between Student and CampusClub is labeled with the Club's association name. Association names correspond to relationship names in Figure 2.1. Both the EER model and UML provide for naming the role of an entity or class in the association. For example, in Figure 2.2, Faculty has the role of advisor in the Advises association and CampusClub has the role of advisorOf. Although not shown in Figure 2.1, the same role names can be added to the EER diagram to describe the role of the entities in the Advises relationship. For example, the edge between Faculty and Advises can be labeled with the role advisor, and the edge between CampusClub and Advises can be labeled with the role advisorOf.

A UML diagram, unlike an EER diagram, includes the concept of navigability. An undirected edge is a bidirectional association, which assumes that an inverse binary relationship is maintained between the classes. Navigability is denoted by a directed edge, which indicates a unidirectional association. Navigability is therefore a reference between two classes that is only maintained in one direction. In Figure 2.2, the Clubs and Advises associations are bidirectional, whereas the associations between Faculty and Department and between Student and Department are unidirectional. As a result, since the association between Faculty and Department is indicated by an association arrow that points to Department and is labeled with the worksIn role name, a Faculty object will maintain information about the Department in which the faculty member works. The inverse direction is not automatically maintained but can be derived by the GetFaculty method of Department.

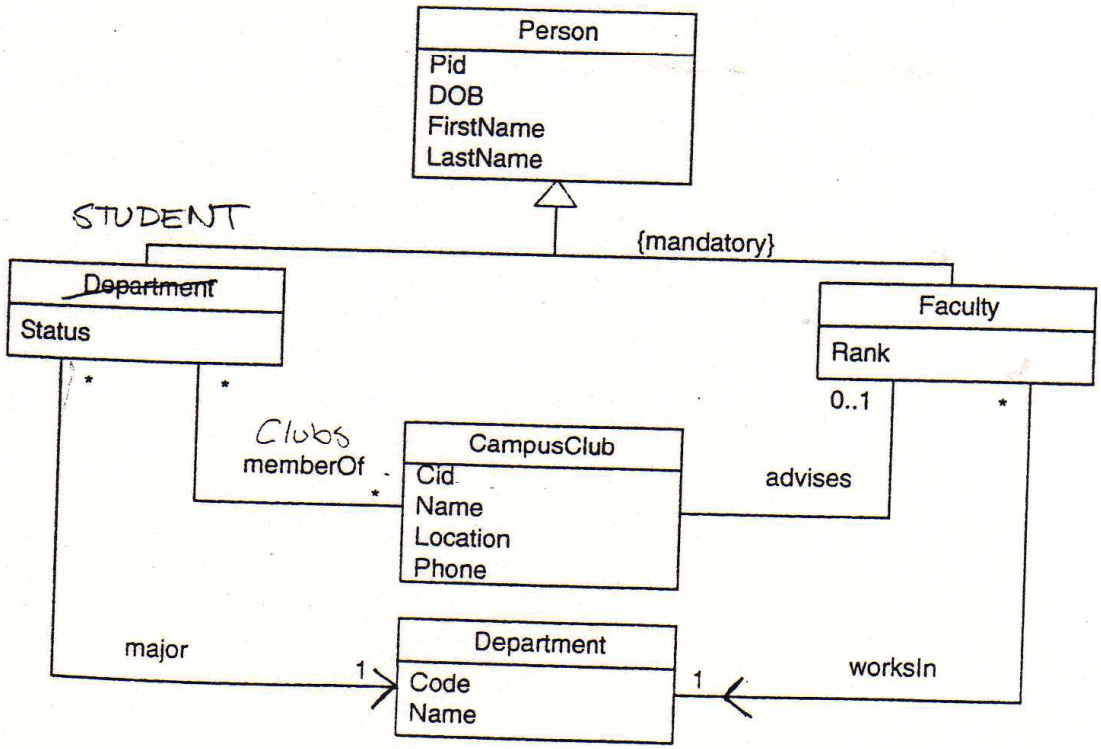

Figure 2.2 UML diagram of the School Database.

valued roles, the default collection type is assumed to be a set. A designer can use the constraint {ordered} to indicate a list or {bag} to indicate an unordered collection possibly containing duplicates. Since a constraint rule does not have a strict syntax, the constraint specification can range from an informal English statement to a formal statement in predicate calculus.

An important feature of UML is its support of generalization and classification. This concept is similar to generalization and specialization in the EER conceptual data model. Diagrammatically, a classification is specified by linking the subclasses of a classification to one triangle, which points to the superclass. In Figure 2.2, Person is classified into the subclasses of Student and Faculty. This notation is similar to the EER model where the subclasses of the specialization are linked to a small circle, which is linked to the superclass. Recall that an "o" inside a circle within an EER diagram indicates possibly overlapping subclasses, and a "d" indicates a disjoint specialization. In UML, a generalization link may be labeled with a discriminator. A discriminator is an attribute of the superclass that indicates the basis for the subtyping. The subtypes that have the same discriminator are assumed to be disjoint. A constraint rule of {mandatory} can also be used to indicate a total specialization. As shown in Figure 2.2, the Person hierarchy has the {mandatory} constraint, indicating that a Person must be either a Student or a Faculty. Although not shown in Figures 2.1 or 2.2, both the EER and UML allow for multiple classifications, such as an additional specialization of Person into Male and Female subclasses.

Figure 2.3 demonstrates the use of attributes on relationships in the EER and UML notations. To illustrate this feature, assume that the example in Figure 2.1 is extended to capture the date that a student joins a club. In the EER notation of Figure 2.3, the attribute is attached to the diamond that represents the relationship. In UML, this feature is supported through the use of association classes. As shown in Figure 2.3, Clubs-Information is an association class that lists the attribute of the association. The association class is connected to the association with a dashed line.

UML class diagrams provide additional advanced concepts that are beyond the scope of this overview. For example, UML allows for the specification of aggregation, commonly referred to as the part-of relationship, which is not inherently supported by the EER conceptual model. The mappings presented in the following sections describe the fundamental concepts needed to support class definitions, class hierarchies, and relationships between objects. Advanced concepts of UML, such as aggregation and other features that are not described here, build on these fundamental mapping procedures through the implementation of constraints in application code. (Refer to *UML Toolkit* by Hans-Erik Eriksson and Magnus Penker [Wiley] for information on more complex UML features.)

Overview of Oracle 8

The Oracle 8.03 (O8) DBMS represents an object-relational extension of its predecessor Oracle 7 [Oracle 1999]. In O8, *object types* (also known as abstract data types, or ADTs) can be used as abstractions of real-world objects. Object types consist of two main components:

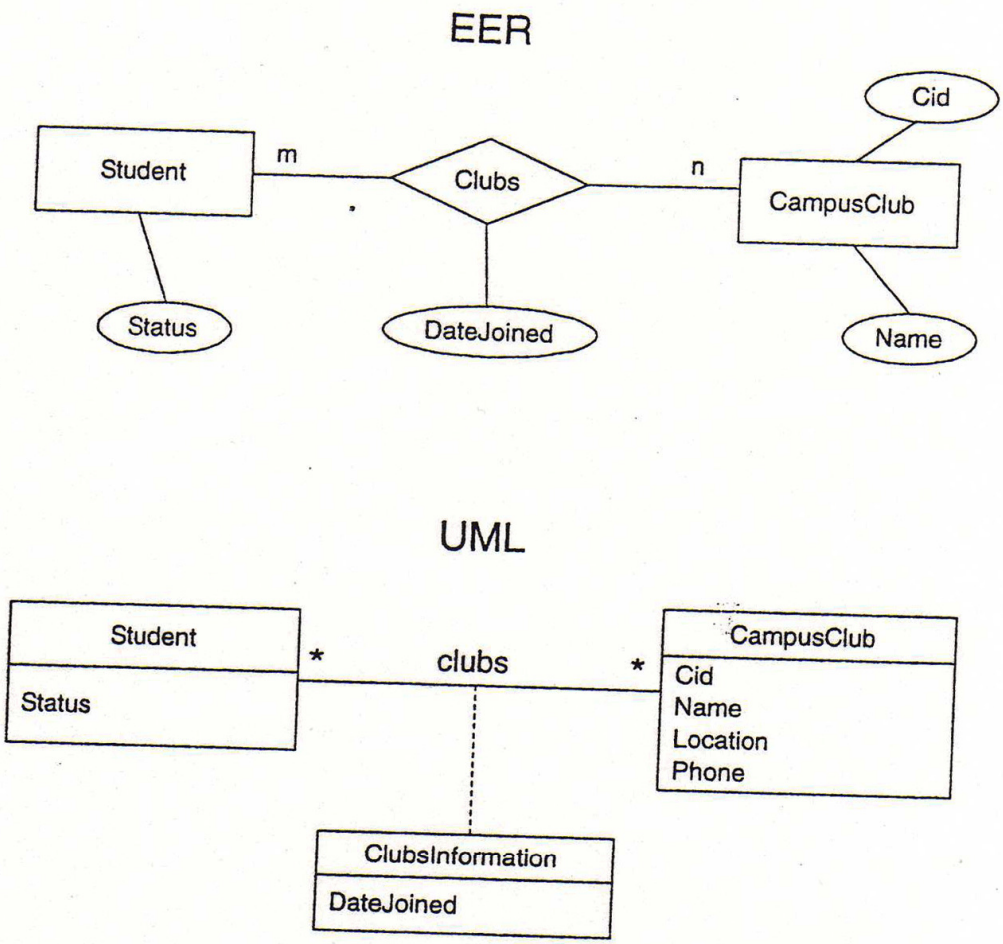

Figure 2.3 EER and UML notations for attributes on relationships.

Attributes. Contain built-in types or other object types as values. Attributes model the structure of the real world entity.

Methods. Functions or procedures written in PL/SQL or an external language like C that are stored in the database. Methods implement the operations the application can perform on the real world entity.

Object types can be used to create *object tables* and/or to define *embedded objects*. In object tables, the tuple instances have object identifiers. Object types that are used to create object tables can be used as the attribute types of attributes in other relational or object tables to create object references between tables. Embedded objects do not have object identifiers, and are used to define complex record structures as attribute values within relational or object tables. The current release of O8 does not support object type hierarchies. O8 does, however, support a feature that allows traditional relational schemas to be transformed into *object views*. By using object views, a user can retrieve, update, insert, and delete relational data as if they were stored as object entities.

O8 supports a number of new types in addition to the scalar types already supported by Oracle 7:

- *VARRAYs* and *nested tables* are two new types that allow a structured collection of data to be the type of a table column.
- REFs (or object references) are used to store logical pointers to objects.
- *LOBs* are large arrays of bytes with additional system-defined properties.

The following section elaborates on the use of the object-relational features of O8 in the generation of O8 schemas from UML conceptual class diagrams.

Mapping Issues

At a high level, mapping UML conceptual diagrams to object-relational schemas in O8 can be summarized as follows:

- Classes map to object tables, where the type of an object table must first be defined as an object type.
- Single-valued relationships between objects (i.e., the 1 side of a 1:1 or 1:M relationship) map to either:
  - References between objects using REFs.
  - Derived attributes that are calculated through the use of functions. A derived attribute can be used only on one side of the relationship. depende del tipo
- Multivalued relationships between objects (i.e., the M side of a 1:M relationship or either side of a N:M relationship) map to the use of either:
  - Structured collections such as VARRAYS or nested tables.
  - Derived attributes that are calculated through the use of functions. A derived attribute can be used only on one side of the relationship. pero en una n:m no hay un lado 1
- Superclass/subclass relationships can be mapped to the use of either:
  - A superclass variable that appears in subclasses to establish class hierarchy relationships.
  - An object type that represents the flatten class hierarchy with relational views established to simulate the subclass structure.
- If object reference information is explicitly maintained on both sides of a relationship, triggers can be generated to maintain inverse relationships automatically.
- Object types can be created to represent user-defined structures, enumeration types, or other complex types that may be needed within an application. Such types can then be used to create embedded objects within relational and object tables.

The following subsections present examples of these mapping procedures and address some of the mapping issues of which a designer must be aware in the mapping process. In the text that follows, Oracle schema components are shown in italics, and UML schema components are shown in normal text, with classes and attributes capitalized and relationships in lowercase.

Objects and Relationships between Objects

To illustrate the mapping of classes to object tables, consider the Department class in Figure 2.2. Listing 2.1 illustrates that the object type *department* is created and then used to create the *departments* object table. Each row in the *departments* object table is referred to as a row object and will have an object identifier (OID). Columns in the table correspond to the attributes of the object type definition, which in this case are *Code* and *Name*.

The navigability notation in Figure 2.2 defines that *department* objects do not maintain attributes that contain the faculty working in the department or the students majoring in the department. Instead, the GetFaculty and GetStudents operations are specified in Figure 2.2. These operations are implemented as functions in Listing 2.1. Each function returns a *persons_array* type, which is a VARRAY of references to *person* objects. We will discuss the *persons_array* definition in more detail in a later section.

The type definition for *department* includes a type body, which defines the code for each of the member functions defined in Listing 2.1. Since we are interested primarily in structural mapping issues in this chapter, the code for the member functions is not shown. This feature of O8, however, readily supports the implementation of methods that are associated with UML class definitions. Since each UML class corresponds to an O8 object type, each method signature in a UML class should have a corresponding function or procedure specification in the object type definition, with an implementa-

```
CREATE OR REPLACE TYPE department AS OBJECT (
    Code      VARCHAR(20),
    Name      VARCHAR(40),
    MEMBER FUNCTION get_students     RETURN persons_array,
    MEMBER FUNCTION get_faculty      RETURN persons_array,
);
/
CREATE OR REPLACE TYPE BODY department IS
    MEMBER FUNCTION get_students RETURN persons_array IS
        -- declarations go here
    BEGIN
        -- function definition goes here
    END;

    MEMBER FUNCTION get_faculty RETURN persons_array IS
        -- declarations go here
    BEGIN
        -- function definition goes here
    END;

    END;
/
CREATE TABLE departments OF department;
```

Listing 2.1 Department object type and object table definition.

tion in the type body. Although not shown here, an object type in O8 also requires the definition of a *map member function*, which defines the relative position of a given instance in the ordering of all instances of the object type. Map member functions are used implicitly by O8 in object type comparisons.

As an example of a more complex object table definition that contains attributes that define relationships with other objects, consider the CampusClub class from Figure 2.2. The CampusClub class is implemented in O8 using the *campus_club* object type and the *clubs* object table shown in Listing 2.2. Attributes such as *Cid*, *Name*, *Location*, and *Phone* are implemented as in traditional relational mappings. Since each Club has one advisor through a relationship in the UML diagram to Faculty objects, the *Advisor* attribute is implemented as a REF to a *person* object. In Listing 2.2, *person* is assumed to be an object type associated with the Person hierarchy from the UML diagram. The complete definition of the *person* object follows in the next section, in which we address mappings for superclass/subclass hierarchies. By defining *Advisor* as a REF of type *person*, the *Advisor* attribute will contain references to OIDs of *person* objects. This representation is similar to object references used in object-oriented databases and replaces the use of foreign keys in the pure relational model. As we will illustrate later, SQL queries can navigate through object relationships that are represented as REFs. In addition to the *Advisor* attribute, the *Members* attribute, representing the membership relationship between Campus Clubs and Students, is represented as a VARRAY that contains REFs of *person* objects. Additional functions and procedures associated with the CampusClub UML class definition can be defined within the *campus_club* object type as needed.

Object Hierarchies

Given that O8 does not support the concept of object hierarchies, a hierarchical structure can be simulated through the use of hierarchy flattening. As discussed in "The

```
CREATE OR REPLACE TYPE campus_club AS OBJECT (
    Cid          INTEGER(2, 10),
    Name         VARCHAR(25),
    Location     VARCHAR(25),
    Phone        VARCHAR(25),
    Advisor      REF person,          -- RELATION Advised by
    Members      persons_array,       -- RELATION Member of
    MEMBER FUNCTION get_advisor_fname    RETURN VARCHAR,
    MEMBER FUNCTION get_advisor_lname    RETURN VARCHAR,
    MEMBER PROCEDURE print_members,
    MEMBER PROCEDURE add_member(member_to_add IN REF person),
    MEMBER PROCEDURE delete_member(member_to_delete IN REF person));
CREATE OR REPLACE TYPE BODY campus_clubs IS
    -- code for the type body goes here
/
CREATE TABLE clubs OF campus_club
```

Listing 2.2 CampusClub object type and object table definition.

Design and Implementation of an EXPRESS to Oracle 8 Mapping" and "A Case Study in Mapping Conceptual Designs to Object-Relational Schemas" [Tjahjadi 1997]; [Urban 2000], the flattening approach implements the superclass at the top of the hierarchy together with all of the attributes of its subclasses. Views for each of the subclasses are then constructed to show the attributes that belong to each subclass. In Listing 2.3, the flattened object type is *person*, which contains attributes of the Person class as well as attributes of the Student and Faculty subclasses. The attributes *Is_person, Is_student,* and *Is_faculty* are attributes included in the flattened object type to identify the subclass membership of a *person* object instance.

Relational views allow the flattened hierarchical structure to simulate the class hierarchy from Figure 2.2. Listing 2.4 shows derived views of the Student and Faculty classes. The information returned for the *student* and *faculty* views includes the REF of the *person* object, the inherited attributes from the Person class, and the immediate attributes of either the Student or Faculty class, respectively. O8 *instead of* triggers can then be used to allow the user to modify the relational views and redirect the changes to the flattened *persons* object table. Relational views are used instead of using the object view feature of O8. Object views allow a user to build virtual objects from relational data. If used on top of object tables, however, object views do not return the same object identifier as in the original object table, thus generating multiple OIDs for the same object.

Pero esta gen. es mandatoria y disjunta, de modo que bastaría con un atributo sub_class_is con valores posibles S o F, o en todo caso 2: is_student, is_faculty, pero no se necesitan los 3, pues Is_person no puede ser 0 para un objeto de persona

A

ojo

```
CREATE OR REPLACE TYPE clubs_table AS TABLE OF REF campus_club;
/
CREATE OR REPLACE TYPE person AS OBJECT (
    DOB          date_t,
    Pid          VARCHAR(20),
    FirstName    VARCHAR(20),
    LastName     VARCHAR(20),
    Status       current_year,
    Member_of    clubs_array,          -- RELATION Member of
    Advisor_of   clubs_table,
    Major        REF department,       -- RELATION Major
    Works_in     REF department,       -- RELATION Works_in
    Is_person    NUMBER(1),
    Is_student   NUMBER(1),
    Is_faculty   NUMBER(1),
    MEMBER FUNCTION ivalidation                RETURN BOOLEAN,
    MEMBER FUNCTION get_department_name        RETURN VARCHAR,
    MEMBER FUNCTION is_associated_club
        (candidate_club IN REF campus_club)RETURN BOOLEAN,
    MEMBER PROCEDURE add_club(club_to_add IN REF campus_club),
    MEMBER PROCEDURE delete_club(club_to_delete IN REF campus_club)
/
CREATE TABLE persons of person
    NESTED TABLE "ADVISOR_OF" STORE AS advised_clubs
```

Listing 2.3 Person object type and object table definition.

Listing 2.5 provides an example of an *instead of* trigger that is used to insert objects into the *persons* object table through the use of the Faculty view. The trigger first verifies that the department *Code* is valid and that there is no other person in the database with the same identifier. A tuple is then inserted into the *persons* object table. Note that the attributes of Person and Faculty are set to appropriate values and attributes associated with Student are set to null. In addition, the *Is_person* and *Is_faculty* flags are set to one and the *Is_student* flag is set to zero. Similar triggers can be created to allow modification and deletion of objects from the views in Listing 2.4. These triggers should be developed to enforce all class hierarchy and membership constraints as specified in the UML schema. For example, in Figure 2.2, a Person is required to be either a Student or a Faculty object, and Student and Faculty objects must be disjoint.

O9i si tiene inheritance

~~Object-relational systems other than O8 may directly support class hierarchies, and as a result will not require the~~ approach described in this section. In such systems, mapping superclasses and subclasses to object definitions will more closely parallel class definitions that typically are found in object-oriented database systems, where support for inherited attributes and methods is provided automatically. The obvious ~~disadvantage~~ of the hierarchy flattening approach in O8 is that it requires significant overhead in the specification of class views and in the coding of *instead of* triggers associated with each view. ~~On the other hand, hierarchy flattening can be a flexible approach for dealing with conceptual models such as UML, which allow objects to be instances of multiple subclasses and support constraints on subclass membership~~. The work in "The Design and Implementation of an EXPRESS to Oracle 8 Mapping" [Tjahjadi 1997] compares the flexibility of the hierarchy flattening approach to class hierarchies in object-oriented database systems, where objects are only allowed to be an instance of one class, and membership constraints on subclasses are not supported.

B ~~An alternative translation option for class hierarchies in O8 is to create a separate object type and table for each class in the hierarchy.~~ Object references can then be used to link together the classes that belong to the hierarchy. The problem with this approach is that an object will have multiple OIDs, one for each class to which it belongs in the hierarchy. Inherited attributes must also be referenced explicitly through the use of the REF attributes.

Is_person nunca puede ser 0

```
CREATE OR REPLACE VIEW student (obj_ref, DOB, Pid, FirstName,
        LastName, Status, Member_of, Major) AS
    SELECT REF(p), p.DOB, p.Pid, p.FirstName, p.LastName, p.Status,
           p.Member_of, p.Major
    FROM   persons p
    WHERE  p.Is_person = 1 AND p.Is_student = 1;

CREATE OR REPLACE VIEW faculty (obj_ref, DOB, Pid, FirstName,
        LastName, AdvisorRnk, Advisor_of, Works_In) AS
    SELECT REF(p), p.DOB, p.Pid, p.FirstName, p.LastName,
           p.AdvisorRnk, p.Advisor_of, p.Works_in
    FROM   persons p
    WHERE  p.Is_person = 1 AND p.Is_faculty = 1;
```

Listing 2.4 Student and Faculty views derived from the person object table.

VARRAYS and Nested Tables

In the previous sections, Listings 2.1 and 2.2 introduced the use of the *persons_array* VARRAY. Listing 2.3 also introduces the use of the *clubs_array* VARRAY and the nested table *clubs_table*. This section elaborates on some of the issues associated with the use of the VARRAY and nested table collection types for representing multivalued attributes and multivalued relationships between objects.

The VARRAY and nested table collection types are similar in that:

- The data types of all elements in each collection must be the same (i.e., each collection must be of a homogeneous type).
- A VARRAY or nested table can be used as the data type of a column in a relational or object table, or as a PL/SQL variable, parameter, or function return type.

```
CREATE OR REPLACE TRIGGER faculty_insert
    INSTEAD OF INSERT ON faculty
    FOR EACH ROW
DECLARE
    p_ref       REF person;
    counter     INTEGER;
    depts       INTEGER;
BEGIN
    SELECT COUNT(*) INTO depts
    FROM departments d
    WHERE ref(d) = :new.Works_In;
    SELECT COUNT(*) INTO counter
    FROM persons p
    WHERE p.Pid = :new.Pid;
    IF (counter = 0) THEN
        IF (depts = 1) THEN
            INSERT INTO persons p VALUES
                (:new.DOB, :new.Pid, :new.FirstName, :new.LastName,
                    NULL, :new.AdvisorRnk, NULL, NULL, NULL,
                    :new.Works_In, 1, 0, 1)
                RETURNING REF(p) INTO p_ref;
        ELSE
            RAISE_APPLICATION_ERROR(-20001,
                'Error - not a valid department code');
        END IF;
    ELSE
        RAISE_APPLICATION_ERROR(-20001,
            'Error - a person already exists with this Pid');
    END IF;
END;
```

Listing 2.5 INSTEAD OF trigger for the faculty view.

The differences between the two are primarily due to the manner in which they are internally managed by O8. A VARRAY must always have a specific size limit. If the size of the ARRAY is smaller than 4000 bytes, O8 stores the VARRAY directly inside the table in which it is defined. Otherwise, O8 will create a binary large object for separate storage of the VARRAY. VARRAYs are also ordered and the contents are dense. This means that all elements from the beginning to the end of the VARRAY must be filled. Individual elements of a VARRAY cannot be deleted. To update a VARRAY, the entire collection must be replaced with a new collection.

Nested tables do not have an upper bound on the size limit. Nested tables are sparse, allowing individual elements to be deleted and inserted at any point in the table. A nested table is always stored as a table separate from the table in which it is defined. Unlike VARRAYS, a nested table can be indexed. The choice of using a VARRAY or nested table therefore depends on application needs. VARRAYS are better for smaller, ordered collections that do not require frequent changes. Nested tables are better for larger collections that are more volatile in nature. In the School Database example, VARRAYS and nested tables are used to illustrate the way in which each collection type is defined and accessed.

To illustrate the use of a nested table, return to the definition of the *persons* table in Listing 2.3. A nested table is used in this example to store the clubs that a faculty member advises. Initially, *clubs_table* is defined as a table of REFs to *campus_club* objects. The *clubs_table* type is then used in the *person* type definition to define the type of the *Advisor_of* attribute. In the table definition statement for the *persons* table, the phrase *"NESTED TABLE "ADVISOR_OF" STORE AS advised_clubs"* is included to create the internal name of the table space that will be used to store the actual nested table.

Definition of the *persons_array* and *clubs_array* VARRAYS is complicated by the fact that recursive references exist. For example, Figure 2.2 shows that Students are members of CampusClubs and CampusClubs have Students as members. Since we want a *person* type to have a VARRAY of REFs to *campus_club* and we want *campus_club* to have a VARRAY of REFs to the *person* type, the types have to be incrementally introduced and refined. In particular, we must create the types *campus_club* and *person* as incomplete object types, allowing the creation of the VARRAYs of REFs for each type. The sequence of definitions is shown in Listing 2.6.

As shown previously in Listing 2.3, the full definition of the object type *person* includes the *Member_of* attribute of type *clubs_array*. The full definition of the object type *campus_club* in Listing 2.2 includes the *Members* attribute of type *persons_array*. Because of the VARRAY definitions, there can be at most 50 members in a given club, and there can be

```
CREATE OR REPLACE TYPE campus_club
/
CREATE OR REPLACE TYPE clubs_array as VARRAY(15) of REF campus_club
/
CREATE OR REPLACE TYPE person
/
CREATE OR REPLACE TYPE persons_array as VARRAY(50) of REF person
```

Listing 2.6 Recursive VARRAY definitions.

at most 15 clubs to which a person belongs. After the recursive definitions in Listing 2.6, *clubs_array* and *persons_array* are holding types that have not yet been declared as objects. After the necessary array types have been created as in Listing 2.6, we can define the *campus_club* and *person* object types as shown in Listings 2.1 and 2.2, respectively.

To illustrate the use of VARRAYS and nested tables, Listing 2.7 shows a function that, given a specific person and club object, determines if the person is associated with the club. The use of DEREF transforms the *to_person* object reference into a tuple of type *person*, thus allowing access to the attributes of a *person* object rather than just the object identifier. If the *person* is a Student, then the code iterates through the *Member_of* VARRAY to determine if the student is a member of the club. If the *person* is a Faculty member, then the code iterates through the *Advisor_of* nested table to determine if the faculty member is an advisor of the club. O8 provides features such as COUNT, FIRST, and LAST for accessing the elements of VARRAYS and nested tables.

Triggers for Maintaining Inverses

We have already illustrated the use of triggers for creating views of an object hierarchy. Triggers can also be used to maintain inverses. As an example, there is an inverse rela-

```
CREATE OR REPLACE FUNCTION is_associated_with_club
    (candidate_club IN REF campus_club,to_person IN REF person)
    RETURN BOOLEAN IS
        is_associated    BOOLEAN;
        prsn             person;
BEGIN
    select DEREF(to_person) into prsn from dual;
    is_associated := false;
    IF (prsnIis_student = 1) THEN
        FOR i IN 1..prsn.Member_of.COUNT LOOP
            IF (candidate_club = prsn.Member_of(i)) THEN
                is_associated:= TRUE;
            END IF;
        END LOOP;
    END IF;
    IF (prsn.Is_faculty = 1) THEN
        FOR i IN prsn.Advisor_of.FIRST..prsn.Advisor_of.LAST LOOP
            IF (candidate_club = prsn.Advisor_of(i)) THEN
                is_associated:= TRUE;
            END IF;
        END LOOP;
    END IF;
    RETURN is_associated;
END;
```

Listing 2.7 Traversal through VARRAYS and nested tables.

tionship between Faculty and CampusClubs through the Advises relationship. The application has been designed so that triggers are used to maintain this relationship. In particular, the advisor of a CampusClub is specified through modification of the *Advisor* attribute of a *campus_clubs* object. The list of clubs that a Faculty member advises is updated automatically through the use of a trigger associated with modification of a *campus_clubs* object. In a similar manner, modification of the relationship from the Faculty side will cause the appropriate *campus_clubs* objects to be modified. Listing 2.8 illustrates a trigger defined for deletion of a *faculty* object. For all clubs advised by the deleted *faculty* object, the trigger sets the *Advisor* attribute to null. Application code for maintaining inverses should enforce all minimum and maximum bounds on the relationship as specified in the UML diagram.

As with the hierarchy flattening approach, the specification of triggers for maintaining inverses represents additional overhead in the schema generation process. Most object-oriented database systems automatically maintain inverse relationships. If the designer of an object-relational schema chooses to explicitly maintain inverse relationships between objects, the designer must be prepared to fill in the appropriate trigger code, potentially iterating through VARRAYS and nested tables in the case of 1:N and N:M relationships.

Derived Attributes

An alternative to explicitly storing both sides of an inverse relationship is to use a function to derive one side of the inverse relationship. In the School Database application, the major and worksIn relationships are expressed using navigability in Figure 2.2, which indicates that only one side of the relationship should be explicitly maintained. As a result, the *person* type in Listing 2.3 was defined to support the *Major* and *WorksIn* attributes, containing REFs to the department that a Student or Faculty member majors in or works in, respectively. As specified in the department class of Figure 2.2, the *department* object type in Listing 2.1 defines the *get_students* function to return the students majoring in a department and the *get_faculty* function to return the faculty that work in a department.

```
CREATE OR REPLACE TRIGGER faculty_delete
    INSTEAD OF DELETE ON faculty
    FOR EACH ROW
DECLARE
    p_ref     REF person;
BEGIN
    SELECT REF(p) into p_ref FROM persons p WHERE p.PID = :old.Pid;
    UPDATE clubs c SET c.Advisor = NULL WHERE p_ref = c.Advisor;
    DELETE FROM persons p WHERE p.PID = :old.Pid;
END;
```

Listing 2.8 Trigger for updating the clubs advised by a faculty member.

los atributos derivados en estos ejemplos son las listas de personas que son _faculty o students de un depto en particular

```
MEMBER FUNCTION get_faculty RETURN persons_array IS
    temp_array persons_array;
BEGIN
    SELECT CAST(MULTISET
                (SELECT REF(p)
                 FROM persons p
                 WHERE p.Is_faculty = 1 and deref(p.Works_in) = SELF)
            AS persons_array)
    INTO temp_array
    FROM dual;
    RETURN(temp_array);
END;
```

Listing 2.9 Function for deriving the faculty members of a department. → ver reverso

The *get_faculty* function is shown in Listing 2.9, which returns a *persons_array* of *faculty* objects (i.e., *person* objects with *Is_faculty* = 1) that have a *WorksIn* attribute set to a specific *department* object. The DEREF function is used to transform the *department* object reference within the *works_in* attribute into a tuple of type *department* for comparison with SELF, where SELF is the specific department object that is executing the *get_faculty* function.

Embedded Objects

Tipos usados para definir columnas de tablas

In addition to supporting the creation of object tables, object types can also be used to create *embedded objects*. Embedded objects are used to support user-defined types that do not need to be represented as independent objects with object identifiers. Embedded objects are stored directly inside of other objects and are created by using the object type definition as an attribute type rather than using a REF to an object type. In the definition of the *person* object type in Listing 2.3, the type *date_t* is used as an embedded object to define the *DOB* (date of birth) attribute of a *person* object. The definition of *date_t* is shown in Listing 2.10, where a date is defined to consist of *day*, *month*, and *year* attributes together with functions for validating a date value. The *date_t* object type therefore encapsulates the semantics of date values. A *person* object *p* can then refer to *p.DOB.year*, for example, to return the year in which the person was born. Using this approach, a date does not have an object identifier but exists as a complex object inside of another object type. Embedded objects can also be used as attribute values in relational tables.

In the School Database example, we have also used user-defined object types to implement enumeration types that serve as embedded objects. For example, the *month* attribute in Listing 2.10 is defined using another object type *months_t*, where *months_t* is used to define valid month values. The *current_year* and *persons_rank* types are also used as embedded objects for enumeration values in the *person* object type definition of Listing 2.3. The *current_year* type defines the enumeration values of freshman, sophomore,

```
CREATE OR REPLACE TYPE date_t AS OBJECT (
    day          INTEGER,
    month        months_t,
    year         INTEGER,
    MEMBER FUNCTION ivalidation                          RETURN BOOLEAN,
    MEMBER FUNCTION valid_date (par IN date_t)           RETURN BOOLEAN,
    MEMBER FUNCTION leap_year (year IN INTEGER)          RETURN BOOLEAN,);
```

Listing 2.10 The date_t object type definition for use as an embedded object.

junior, senior, and graduate. The *persons_rank* type defines enumeration values that are used to identify a faculty member as being either an assistant, associate, or full professor.

Listing 2.11 shows the object type definition for *months_t*, together with the table definition that enumerates the valid month values and also associates each value with an internal numeric index. The implementations of the functions and procedures associated

```
CREATE OR REPLACE TYPE months_t AS OBJECT (
    current_choice INTEGER,
    MAP MEMBER FUNCTION type_value RETURN INTEGER,
    MEMBER FUNCTION ivalidation RETURN BOOLEAN,
    MEMBER FUNCTION ivalidation(choice IN INTEGER) RETURN BOOLEAN,
    MEMBER FUNCTION ivalidation(choice IN VARCHAR2) RETURN BOOLEAN,
    MEMBER FUNCTION get_value RETURN INTEGER,
    MEMBER FUNCTION get_enumerated RETURN VARCHAR2,
    MEMBER PROCEDURE set_value (new_choice IN INTEGER),
    MEMBER PROCEDURE set_enumerated (new_choice IN VARCHAR2));

CREATE TABLE months_choices (
    elem_id INTEGER PRIMARY KEY,
    element VARCHAR2(200) UNIQUE);

INSERT INTO months_choices VALUES ( 0, 'January');
INSERT INTO months_choices VALUES ( 1, 'February');
INSERT INTO months_choices VALUES ( 2, 'March');
INSERT INTO months_choices VALUES ( 3, 'April');
INSERT INTO months_choices VALUES ( 4, 'May');
INSERT INTO months_choices VALUES ( 5, 'June');
INSERT INTO months_choices VALUES ( 6, 'July');
INSERT INTO months_choices VALUES ( 7, 'August');
INSERT INTO months_choices VALUES ( 8, 'September');
INSERT INTO months_choices VALUES ( 9, 'October');
INSERT INTO months_choices VALUES (10, 'November');
INSERT INTO months_choices VALUES (11, 'December');
```

Listing 2.11 The month_t object type definition for defining enumeration types.

with *months_t* (not shown in Listing 2.11) use the table *months_choices* to ensure that only valid month values are assigned to the *current_choice* attribute of *months_t*. The technique for defining enumeration types used in the School Database example is based on the technique described by Tjahjadi in [Tjahjadi 1997].

Embedded objects are especially useful in support of mappings from UML schemas. UML class diagrams provide the user with flexibility in specifying the types of attribute values. Using embedded objects in O8, a developer can implement user-defined types that correspond to the types used in the UML specification. Unlike other object-relational systems, object types in O8 fully support encapsulation, where function and procedure definitions can be associated explicitly with type definitions.

Queries over Objects

One of the advantages of using object REFs and collections of object REFs in an object-relational schema design is that REFs allow a more direct representation of object-oriented conceptual schemas. As a result, queries can make use of REFs to navigate through object relationships instead of explicitly specifying join conditions. As an example, suppose that we want to return the name of the department of a club's advisor. The following select statement uses object references to traverse through the single-valued relationships that exist from a club to its advisor and from an advisor to the department in which the advisor works:

```
select   c.Advisor.WorksIn.Name
from     clubs c
where    c.name = "Epsilon Sigma";
```

This query returns a string value representing the name of the appropriate department.

Queries that return objects must be written either to return the object REF or to return the tuple that represents the object type. The DEREF function is provided to access the actual type of an object reference. As an example, the following query retrieves the code and name of the department in which Pablo is a major:

```
DECLARE
    d department;
BEGIN
    select DEREF(p.Major) into d
    from persons p
    where p.FirstName = "Pablo";
END;
```

In this query, *p.Major* returns a REF to a *department* object. *DEREF(p.Major)* returns a tuple of type *department*, which contains the *Code* and *Name* attributes as defined in Listing 2.1.

Comparison to Other Systems

As part of our study, we performed comparisons of the mappings described in the previous sections to:

- The relational model
- A different object-relational model
- An object-oriented database model

Due to its object-oriented nature, UML obviously provides a smoother mapping to an object-relational model than to a traditional relational model. Object tables, nested tables, VARRAYS, and object references have provided a more natural representation of UML data than relational tables. The object extensions to SQL also provided a more concise expression of queries compared to traditional relational queries. Class definitions and user-defined types, however, require the generation of the appropriate methods and triggers that subsequently can be used to access objects and complex types as well as to check the appropriate constraints associated with such types. As a result, the translation of conceptual schemas from a model such as UML to an object-relational schema is more complex than traditional ER-to-relational mapping techniques. The mappings that we have defined not only identify the object types, relational tables, and object tables of the target schema, but also define the manner in which the methods and triggers associated with types and tables should be used.

PostgreSQL was the first database management system that utilized object-relational database concepts. The system is public-domain software that supports a subset of SQL92 and some extended features of object-relational systems. Illustra Information Technology (a wholly owned subsidiary of Informix Software, Inc.) also provides a commercial object-relational database system called *Data Blade*, which was originally based on Postgres (Stonebraker 1996). Since *Data Blade* was not freely available at the time of this study, PostgreSQL is used for comparison in this subsection.

Although PostgreSQL also has a *CREATE TYPE* statement, the way it is used is different from the Oracle 8 *CREATE TYPE* statement. The approach in Oracle 8 for defining object types is to create a composite type with its associated member functions (like creating a *class* in C++) that can be used as attribute types or to create object tables. In PostgreSQL, the *CREATE TYPE* statement will create a new base (primitive) type that is stored as a TEXT type internally, except for the base type that is used as an array. The new base type, which is referred to as a *user-defined type*, can have a name at most 15 characters long. A PostgreSQL user-defined type must have *input* and *output* functions. These functions are used to determine how the new type appears in string form for input by and output to the user, and how the new type is stored in memory. Contrary to Oracle 8, the PostgreSQL user-defined type cannot specify any encapsulated member functions. Global functions are used to access and manipulate the type. PostgreSQL also does not have strong procedural language support as with Oracle's PL/SQL. As a result, PostgreSQL requires the use of *native calls implementation* for most of the PostgreSQL functions.

A table in PostgreSQL is the same as an object table in Oracle 8. Every instance in PostgreSQL has a unique object identifier, which is stored implicitly in a special attribute called *oid*. A PostgreSQL table can also be used as a user-defined type for arguments in PostgreSQL functions or as an attribute type. Because of the PostgreSQL table properties, a UML class would be mapped using a PostgreSQL table. Forward relationships between objects are implemented using the OID built-in type (or an array of OIDs for multivalued relationships). A PostgreSQL OID can be viewed as a generic OID that stores only the persistent object logical address. To retrieve the object instance that owns the OID, an SQL join operation has to be performed.

UML inverse relationship mappings in PostgreSQL use the same UML-to-O8 mapping principles. The first alternative is to use a function to determine the inverse relationship. This approach could face function name conflicts caused by the same inverse attribute names for different tables since all PostgreSQL functions have to be defined globally and are not encapsulated as in Oracle 8 member functions. The other approach is to have an attribute of OID type to store the inverse relationship information. This approach needs the PostgreSQL rule system to maintain the integrity of the OID. UML derived attributes would also be mapped using PostgreSQL functions, which can cause name conflicts between derived attributes with the same name but used for different tables. The advantage of PostgreSQL over Oracle 8, however, is its support for inheritance. In PostgreSQL, there is no need to create multiple table views as in the O8 mapping scheme, since object tables in PostgreSQL can be organized into inheritance hierarchies.

For mapping comparisons to object-oriented database technology, we used Objectivity/DB [Objectivity 1995], a commercial object-oriented database system that is compliant with the ODMG-93 standard [ODMG 1996]. A UML class is represented in Objectivity/DB using a class that inherits from *ooObj* class. UML inheritance hierarchies can be mapped directly to Objectivity inheritance hierarchies. To support persistent inheritance, only the supertype of the type hierarchy needs to be inherited from the *ooObj* class. Creation of Objectivity classes that represent multiple inheritance are required to enforce overlapping subclasses in UML. The ability to redefine the default constructor for Objectivity classes can be utilized to enforce UML subclass constraints. By default, every Objectivity persistent class inherits a virtual function called *ooValidate* from *ooObj* class. The *ooValidate* function can be overridden to do any necessary constraint checking that can be called from the persistent class constructors.

Object instances that inherit from *ooObj* can have *to-one* or *to-many* relationships to other *ooObj* instances. Objectivity supports both unidirectional relationships and bidirectional relationships that can be maintained automatically by the database system. Bidirectional relationships can be viewed as inverse relationships. These *ooObj* features can be used directly for the UML forward and inverse relationships without any triggers or member function simulations as in O8 and PostgreSQL. Expressing a derived attribute in Objectivity is implemented simply by using the class public member function. Collection types can be represented using the Objectivity array called *ooVArray*, a generic variable array that can contain elements of the same type.

The primary advantage of an object-relational system such as Oracle 8 over the use of an object-oriented database is found in the use of SQL. Most object-oriented databases do not yet support the advanced query facilities found in object-relational systems.

Conclusion

This chapter has described the fundamental issues associated with mapping UML class diagrams to object-relational schemas using the Oracle 8 object-relational system as a framework to define the basic mapping issues. The mapping defined in this research has been targeted to utilize most of the object features within Oracle 8. In particular:

- Object types support the creation of abstract data types that correspond to class definitions in UML and to other, user-defined complex types that are needed for attribute definitions.
- Object tables are created from object types and used to hold persistent instances with object identifiers that correspond to UML class objects.
- Embedded objects are used to define complex attribute values from user-defined object types.
- Object references (REFs) are used to define relationships between objects.
- VARRAYS and nested tables are used to implement multivalued attributes and multivalued relationships between objects.
- Triggers are used to maintain inverse relationships between objects.
- Relational views and instead of triggers are used to simulate the functionality of UML class hierarchies.

Our work with mapping UML to an object-relational model demonstrates several important results about the use of object-relational concepts. As with the relational model, designing an application directly through the creation of an object-relational schema can be a difficult task; it is still more desirable to design an application at a higher level using a modeling language such as UML. The generation of the underlying implementation schema, however, is complicated by the advanced features of the object-relational model. More alternatives must be considered in the translation process to exploit fully the use of object-relational features. As a result, the designer may need to be more involved in decisions about the final mapping solutions and in filling in the details associated with views, triggers, and user-defined types. Ideally, automatic schema translation tools are needed that understand the different mapping options and interact with the designer in the effective use of object-relational technology.