

Regular paper

A Methodological Approach for Object-Relational Database Design using UML

Esperanza Marcos, Belén Vela, José María Cavero

Kybele Research Group, Rey Juan Carlos University, 28933 Madrid, Spain; E-mail: {cuca,b.vela,j.m.cavero}@escet.urjc.es

Initial submission: 22 January 2002/Revised submission: 10 June 2002

Published online: 7 January 2003 – © Springer-Verlag 2003

Abstract. The most common way of designing databases is by means of a conceptual model, such as E/R, without taking into account other views of the system. New object-oriented design languages, such as UML (Unified Modelling Language), allow the whole system, including the database schema, to be modelled in a uniform way. Moreover, as UML is an extendable language, it allows for any necessary introduction of new stereotypes for specific applications. Proposals exist to extend UML with stereotypes for database design but, unfortunately, they are focused on relational databases. However, new applications require complex objects to be represented in complex relationships, object-relational databases being more appropriate for these requirements. The framework of this paper is an Object-Relational Database Design Methodology, which defines new UML stereotypes for Object-Relational Database Design and proposes some guidelines to translate a UML conceptual schema into an object-relational schema. The guidelines are based on the SQL:1999 object-relational model and on Oracle8i as a product example.

Keywords: UML extensions – Stereotypes – Object-Relational Databases – Database Design – Object Persistence – Design Methodology – UML – SQL:1999 – Oracle

1 Introduction

In spite of the impact of relational databases over the last few decades, databases of this kind have some limitations for supporting the data persistence required

by present day applications. Owing to recent hardware improvements, more sophisticated applications have emerged, such as CAD/CAM (Computer-Aided Design/Computer-Aided Manufacturing), CASE (Computer-Aided Software Engineering), GIS (Geographic Information System), etc. They may be characterized as consisting of complex objects with complex relationships. Representing such objects and relationships in the relational model implies that the objects must be broken down into a large number of tuples. Thus, a considerable number of joins are necessary to retrieve an object and, when tables are too deeply nested, performance is significantly reduced [3]. A new generation of databases has appeared to solve these problems: the object-oriented database generation, which includes object-relational [24] and object databases [4]. This new technology is well suited for storing and retrieving complex data because it supports complex data types and relationships, multimedia data, inheritance, etc.

Nonetheless, good technology is not enough to support complex objects and applications. It is necessary to define methodologies that guide designers in the object database design task, in the same way as has traditionally been done with relational databases. In recent years some approaches to object-oriented database design have appeared [5, 11, 18, 23, 24]. Unfortunately, none of these proposals can be considered “the method”, either for object-relational or for object databases. On the one hand, they do not consider the latest versions of the representative standards for the two technologies: ODMG 3.0 for object databases [8] and SQL:1999 for object-relational databases [10], while, on the other hand, some of them are based on former techniques such as OMT [5] or even on the E/R model [24]. They therefore have to be updated with UML, SQL:1999 and ODMG 3.0 as their reference models.

This paper is a revised and extended version of Extending UML for Object-Relational Database Design, presented in the UML'2001 conference [17].

In this paper we outline a methodology for object-relational database design. The methodology is based on UML extended with the required stereotypes. We will focus on object-relational databases, although the proposed UML extensions could also be adapted to object database design.

UML [6], as the Unified Modelling Language, is becoming increasingly more accepted. It also has the advantage of being able to model the full system, including the database, in a uniform way. Besides, as UML is an extendable language, it is possible to define the required stereotypes, tagged values and/or constraints, for each specific application. Therefore, in the methodology suggested we propose to use the UML class diagram as the conceptual modelling notation. We also propose some guidelines for translating a conceptual schema (in UML notation) into a logical schema. The logical model used is the SQL:1999 object-relational model so that the guidelines were not dependent on the different implementations of object-relational products. We use Oracle8i as an implementation example. Moreover, comments are also offered here on every issue affected by improvements introduced in the new version, Oracle9i.

Traditionally, methodologies provided graphical techniques to represent a relational schema, such as Data Structured Diagrams (DSD), or some other graphical representation (see, for example, [2]), etc. In the same way, an object-relational schema can be represented either in SQL (SQL:1999 or Oracle8i) or by means of some graphical notation. As the graphical notation to represent the logical schema we also propose to use the UML class diagram extended with the required stereotypes, tagged values and/or constraints.

Both the methodology and the UML extensions have been applied to several cases of study. In this paper, we present as an example a project for the management of reservations for the University's computer classrooms via the web.

We would like to remark on the importance of providing methodological guidelines for database design using UML for data intensive applications. "Generic lifecycle methodologies for object-oriented development have, to date, not given serious consideration to the need for persistence; either in terms of storage of objects in a relational database or in an objectbase" [7]. Information systems have to manage very many data that need to be persistent. Good persistent data design will improve use and maintenance. Today, the main mechanism of persistence is still the database. As is stated in [12], the IDC market research firm reported global sales revenue for 1999 of \$ 11.1 billion for relational and object-relational databases and \$ 211 million for object databases. Up to 2004, IDC predicts annual growth rates of 18.2 percent for relational and object-relational databases and 12.5 percent for object ones. Therefore, databases will still be the kernel of almost every information system for a long time.

The rest of the paper is organized as follows: Sect. 2 summarizes the current UML extensions for database design. Section 3 proposes new UML extensions for object-relational database design based on the SQL:1999 and Oracle8i object-relational models. Section 4 sums up, by means of an example, the methodology, including some transformation guidelines, using the proposed extensions. Finally, Sect. 5 summarizes the main conclusions and future work.

2 Previous Work




The UML extension mechanism allows for the controlled extension of the language by means of stereotypes, tagged values and constraints [6].

- **Stereotype:** "a stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. This new block has its own special properties (each stereotype may provide its own set of tagged values), semantics (each stereotype may provide its own constraints), and notation (each stereotype may provide its own icon). A stereotype is rendered as a name enclosed by guillemets (for example, << name >>) and placed above the name of another element. As a visual cue, you may define an icon for the stereotype."
- **Tagged value:** "a tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties. A tagged value is rendered as a string enclosed by brackets and placed below the name of another element."
- **Constraint:** "a constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. A constraint specifies conditions that must be held true for the model to be well-formed. A constraint is rendered as a string enclosed by brackets and placed near the associated element."

For very common applications, such as Web applications, database applications, etc., it would be desirable to provide a standard extension that could be used by every developer. So, Conallen has proposed a UML extension for Web applications [9] and there are also some extensions for database design [1, 6, 19]. Taking into account the previous extensions for database design we propose some specific stereotypes for each of the phases of the development of a relational database (see Table 1).

It will be noticed that Table 1 considers the relational model including stereotypes to represent primary keys, foreign keys, etc. Nevertheless, it does not provide specific stereotypes for object or object-relational models such

Table 1. Stereotypes for database design

	Database Element	UML Element	Stereotype	Icon
Architectural	Database	Component	<<Database>>	
	Schema	Package	<<Schema>>	
Conceptual	Persistent class	Class	<<Persistent>>	
	Multivalued Attribute	Attribute	<<MA>>	
	Calculated Attribute	Attribute	<<DA>>	
	Composed Attribute	Attribute	<<CA>>	
	Identifier	Attribute	<<ID>>	
Logical	Table	Class	<<Table>>	
	View	Class	<<View>>	
	Column	Attributes	<<Column>>	
	Primary Key	Attributes	<<PK>>	
	Foreign Key	Attributes	<<FK>>	
	NOT NULL Constraint	Attributes	<<NOT NULL>>	
	Unique Constraint	Attributes	<<Unique>>	
	Trigger	Constraint	<<Trigger>>	
	CHECK Constraint	Constraint	<<Check>>	
	Stored Procedure	Class	<<Stored Procedure>>	
Physical	Tablespace	Component	<<Tablespace>>	
	Index	Class	<<Index>>	

as stereotypes for collection types, REF types, methods, etc. The next section introduces the new required extensions (stereotypes, tagged values and/or constraints) for object-relational database design.

3 UML Extensions for Object-Relational Database Design

Before explaining the UML extensions for object-relational database design we have to introduce the main constructors of the object-relational model. As we pointed out in Sect. 1, we have taken SQL:1999 and Oracle8i as our reference models, the former because it is the current standard for object-relational databases and the latter as an example of product implementation. The SQL:1999 data model extends the relational data model with some new constructors to support objects. Most of the latest versions of relational products include some object extensions, but because they generally came onto the market before standard approval, current versions of object-relational products usually do not totally match the SQL:1999 model.

3.1 The Object-Relational Model: SQL:1999 and Oracle8i
SQL:1999 Object-Relational model. SQL:1999 [10, 13] is the current standard for object-relational databases. Its data model seeks to integrate the relational model with the object model. In addition to the object extensions, SQL:1999 provides other extensions to the SQL-92, such as

triggers, OLAP extensions, new data types for multimedia data storage, etc. One of the main differences between the relational and the object-relational model is that the First Normal Form (1NF), the basic rule of a relational schema, has been removed from the object-relational model. A column of an object table can therefore contain a collection data type.

SQL:1999 allows the user to define new structured data types according to the required data types for each application. Structured data types provide SQL:1999 the main characteristics of the object model. It supports the concept of strongly typed language, behaviour, encapsulation, substitutability, polymorphism and dynamic binding.

Structured types can be used as the type of a table or as the type of a column. A structured type used as the base type in the definition of a column permits the representation of complex attributes; in this case, structured types represent value types. A structured type used as the base type in the definition of a table corresponds to the definition of an object type (or a class); the table being the extension of the type. In SQL:1999 such tables are called typed tables. An object in SQL:1999 is a row of a typed table. When a typed table is defined, the system adds a new column representing the OID (Object Identifier) of each object of the table. The value of this attribute is generated by the system, it is unique for each object and cannot be modified by the user. Figure 1 shows an example of a structured type defined in SQL:1999; in a the structured type is used as a value type (as the type of

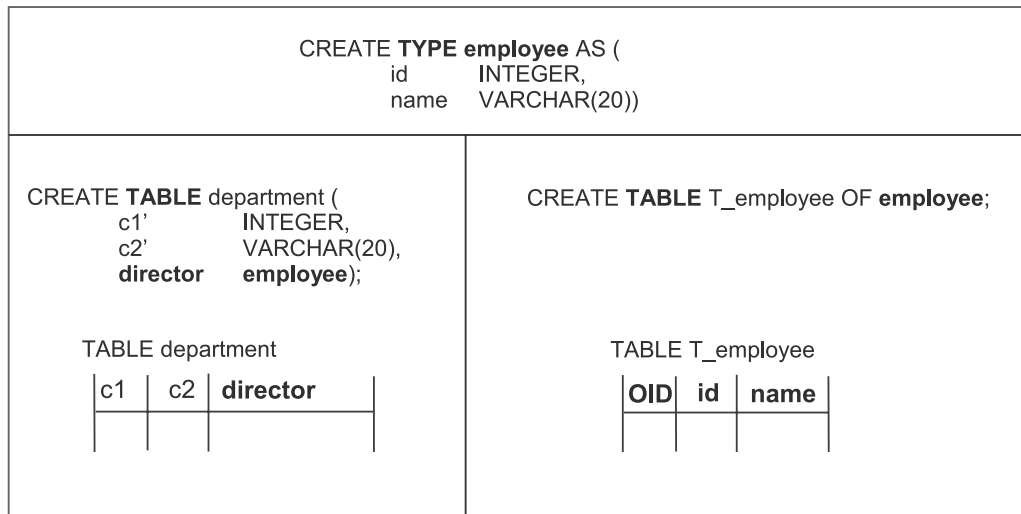


Fig. 1. Structured types used as value and object types in SQL:1999 [13]

a column of a table) whereas in b it is used as an object type (as the type of a table).

A structured type can include associated methods representing its behaviour. A method is an SQL function, whose signature is defined next to the definition of the structured type. The body specification is defined separately from the signature of the method.

SQL:1999 supports simple inheritance for structured types and for typed tables. A subtype inherits the attributes and the behaviour of the supertype. A subtable inherits the columns, constraints, triggers and methods of the supertable.

A row of a typed table is an object and differs from the rest of objects by its OID. The value of the OID is generated by the system when a new object is inserted in the table. The type of this column is a reference type (REF). Therefore, each typed table has a column that contains the OID value. There are different REF types, one for each object type; that is, the REF type is a constructor of types rather than a type itself. An attribute defined as a reference type holds the OID of the referred object. So, the REF type permits the implementation of relationships without using foreign keys.

SQL:1999 supports another structured type: the ROW type. The ROW type is a structured type defined

by the user. It has neither extension nor OID; so, it cannot be used as an object type.

SQL:1999 only supports one collection type: ARRAY, which can be used whenever another type can be placed (as the type of an attribute of a structured type, as the type of a column, etc.). The ARRAY type allows for the representation of multivalued attributes without requiring tables to be in 1NF.

Oracle8i Object-Relational Model. As well as SQL:1999, Oracle8i [20–22] supports structured data types that can be defined by the user (but with a different syntax, as can be seen in Fig. 2). A structured type can be used, as in SQL:1999, as a column type or as a type of a table. A structured type used as a column type represents a value type and a structured type used as the type of a table represents an object type, where the table is the extension of the type. Each row of such a table is an object and, in the same way as in SQL:1999, the tables have a special column of reference type (REF) that allows for the identification of each object (OID). It is also possible to define an attribute as a reference to an object type. Oracle8i allows adding behaviour to object types, defining the signature of the methods as part of the type definition. The body of the method is specified separately.

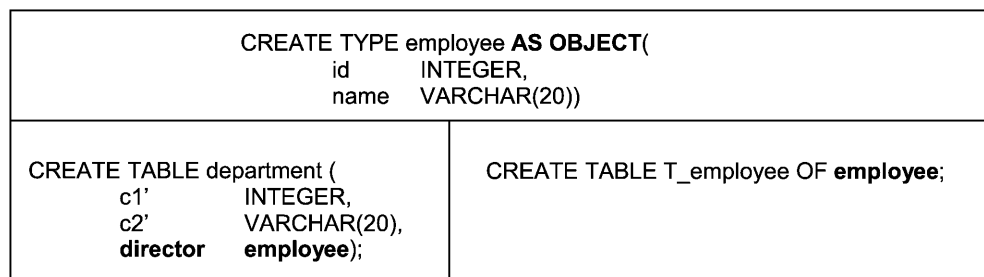


Fig. 2. Structured types used as value and object types in Oracle8i

Oracle8i supports two kinds of collections: VARRAYS, equivalent to the SQL:1999 ARRAY and the Nested Table. A Nested Table is a table that is embedded in another table. It is possible to define a table data type and to use this type as a column type in a table. Therefore, this column contains a table (called a nested table) with a collection of values, objects or references. In Oracle8i, the elements of a collection type (VARRAYS or nested tables) can be of any data type except another collection type. In Oracle9i, this restriction disappears, allowing to define multilevel collections [22].

One of the main differences between the Oracle8i and SQL:1999 object-relational models is that Oracle8i does not support inheritance, whether of types or tables. However, Oracle9i supports single type inheritance. A subtype can be created under a non-final type. It inherits all attributes and methods from its supertype. It can add new attributes and methods and/or override inherited methods [22].

3.2 Object-Relational Extension for UML

If we want UML to be used as the standard modelling language for designing databases, we have to modify it for it to allow us to represent object-relational schemas. In this way, it would be able to represent the constructors defined above, such as the ARRAY, the nested table or the REF type.

One possible solution could be to modify the UML meta-model, which may seem rather drastic. However, UML has been designed to be extended in a controllable way. To accommodate this need for flexibility, UML provides the extension mechanism already explained in Sect. 2. This mechanism enables us to create new types of building blocks by means of stereotypes, tagged values and constraints.

According to [9], a UML extension should contain: a brief description of the extension; the list and description of the stereotypes, tagged values and constraints; and a set of rules of well-formedness to determine whether a model is semantically consistent. For each stereotype we have to specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype [6]. If we want these stereotype elements to have distinctive visual cues, we have to define new icons for them [6]. Table 2 summarizes the UML extension proposed for object-relational database design.

4 Methodological Framework

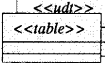



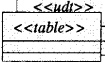
This section summarizes the main steps of a methodology for object-relational database design. The methodology utilizes UML extended with the stereotypes mentioned in the previous section, and is based on the proposals of Bertino and Marcos for object-oriented database design [3] and those of Marcos and Cáceres [14]. Guidelines for the transformation from conceptual into logical schemas will also be offered in this section. Finally, a brief example will illustrate the main steps of the methodology.




As Fig. 3 shows, the methodology proposes three phases: analysis, design and implementation. Nonetheless, differences between analysis, design and implementation phases are not as strong as in structured design.

At the **analysis** phase, the UML class diagram is used to design the conceptual schema instead of the Extended E/R Model (commonly used for relational databases), because UML is the standard language for object-oriented system design. Unlike E/R, UML has the advantage that

Table 2. UML Extension for object-relational database design

<p>Description</p> <p>This extension to UML defines a set of stereotypes, tagged values and constraints that enable us to model applications that work with object-relational databases. The extensions are defined for certain components that are specific to object-relational models allowing us to represent them in the same diagram that describes the rest of the system. The extension is based on SQL:1999 and Oracle8i object-relational models. Each element of the object-relational model has to have a UML representation. To choose the stereotyped element we used the following criteria:</p> <ul style="list-style-type: none"> • For SQL:1999 we have considered structured types and typed tables as stereotyped classes because they are explicitly defined in the SQL schema. The rest of the types (REF, ROW and ARRAY) are considered as stereotyped attributes. • For Oracle8i we have considered object types, object tables and nested tables as stereotyped classes for the same reason as in SQL:1999. The REF type has been considered as a stereotyped attribute because it cannot be defined explicitly in the SQL schema. As the VARRAY may or may not be defined explicitly in the schema we shall allow for both possibilities: defining it as stereotyped attributes or as stereotyped classes. We shall use the stereotyped class when the VARRAY type was defined explicitly in the schema. In this way, the UML schema with stereotypes for Oracle8i will help the developer in the compilation process (compiling these schemas in Oracle8i is tedious because types have to be recompiled, so this technique can be helpful for the user).
<p>Extensions Prerequisites</p> <p>We consider that the required extension for the relational mode has already been defined (see Table 1).</p>

SQL:1999 Stereotypes	
<p>Structured Type Metamodel class: Class. Description: A <code><<UDT>></code> allows the representation of new User defined Data Types.</p> <p>Icon: None. Constraints: Can only be used to define value types.</p> <p>Tagged values: None.</p>	<p>Typed Table Metamodel class: Class. Description: It is defined as <code><<Object Type>></code>. It represents a class of the database schema, that should be defined as a table of a structured data type.</p> <p>Icon: </p> <p>Constraints: A typed table implies the definition of a structured type, which is the type of the table.</p> <p>Tagged values: None.</p>
<p>Knows Description: A <code><<Knows>></code> association is a special relationship that joins a class with a user defined data type <code><<UDT>></code> that is used with the class. It is a uni-directional relationship. The direction of the association is represented by an arrow at the end of the user defined type used by the class.</p> <p>Icon: None. Constraints: Can only be used to join a <code><<Object Type>></code> class with a <code><<UDT>></code> class.</p> <p>Tagged values: None.</p>	<p>REF Type Description: A <code><<REF>></code> represents a link to some <code><<Object Type>></code> class.</p> <p>Icon: </p> <p>Constraints: A <code><<REF>></code> attribute can only refer to a <code><<Object Type>></code> class.</p> <p>Tagged values: Tagged values: The <code><<Object Type>></code> class to which it refers.</p>
<p>ARRAY Metamodel class: Attribute. Description: An <code><<Array>></code> represents an indexed and bounded collection type.</p> <p>Icon: </p> <p>Constraints: The elements of an <code><<Array>></code> can be of any data type except the <code><<Array>></code> type.</p> <p>Tagged values: The basic types of the array. The number of elements.</p>	<p>ROW Type Metamodel class: Attribute. Description: A <code><<row>></code> type represents a composed attribute with a fixed number of elements, each of them can be of different data type.</p> <p>Icon: </p> <p>Constraints: Has no methods.</p> <p>Tagged values: The name for each element and its data type.</p>
<p>Redefined Method Metamodel class: Method. Description: A <code><<redef>></code> method is an inherited method that is implemented again by the child class.</p> <p>Icon: None. Constraints: None.</p> <p>Tagged values: The list of parameters of the method with their data types. The data type returned by the method.</p>	<p>Deferred Method Metamodel class: Method. Description: A <code><<def>></code> method is a method that defers its implementation to its child classes.</p> <p>Icon: None. Constraints: It has to be defined in a class with children</p> <p>Tagged values: The list of parameters of the method with their data types. The data type returned by the method.</p>
Oracle8i Stereotypes	
<p>Object Type Metamodel class: Class. Description: A <code><<UDT>></code> allows the representation of new user defined data types. It corresponds to the structured type in SQL:1999.</p> <p>Icon: None. Constraints: Can only be used to define value types.</p> <p>Tagged values: None.</p>	<p>Object Table Metamodel class: Class. Description: It is defined as <code><<Object Type></code>. It represents a class of the database schema, that should be defined as a table of an object type. It corresponds to the typed table in SQL:1999.</p> <p>Icon: </p> <p>Constraints: A typed table implies the definition of a table of a structured type.</p> <p>Tagged values: None.</p>

<p>Knows Metamodel class: Association. Description: A << <i>Knows</i> >> association is a special relationship that joins a class with a user-defined data type (<< <i>UDT</i> >>, << <i>Array</i> >> or << <i>NT</i> >>) that is used with the class. It is a unidirectional relationship. The direction of the association is represented by an arrow at the end of the user defined type used by the class. Icon: None. Constraints: Can only be used to join a << <i>Object Type</i> >> class with a << <i>UDT</i> >> class. Tagged values: None.</p>	<p>REF Type Metamodel class: Attribute. Description: A << <i>REF</i> >> represents a link to some << <i>Object Type</i> >> class. Icon:  Constraints: A << <i>REF</i> >> attribute can only refer to a << <i>Object Type</i> >> class. Tagged values: The << <i>Object Type</i> >> class to which it refers.</p>
<p>VARRAY Metamodel class: Attribute/Class. Description: A << <i>Array</i> >> represents an indexed and bounded collection type. It corresponds to the array type in SQL:1999. Icon:  Constraints: The elements of an << <i>Array</i> >> can be of any data type, except another collection type (<< <i>Array</i> >> or << <i>NT</i> >>) (see Comments below). Tagged values: The basic types of the array. The number of elements.</p>	<p>Nested Table Metamodel class: Class. Description: A << <i>NT</i> >> represents a non-indexed and unbounded collection type. Icon:  Constraints: The elements of a << <i>NT</i> >> can be of any data type except another collection type (<< <i>Array</i> >> or << <i>NT</i> >>) (see Comments below). Tagged values: The basic type of the nested table.</p>
<p>Rules of Well-Formedness Each << <i>UDT</i> >>, << <i>Array</i> >> or << <i>NT</i> >> class has to be joined by a << <i>Knows</i> >> association with any class. A << <i>REF</i> >> attribute in a << <i>Object Type</i> >> class implies an association with another class. A << <i>Object Type</i> >> class that contains a collection attribute whose elements were objects of a << <i>Object Type</i> >> class of << <i>REF</i> >> to these objects implies an association between the two classes. Each << <i>Object Type</i> >> class corresponds in SQL:1999 to a structured type with its corresponding extension. The extension is the typed table. Each << <i>Object Type</i> >> class in Oracle8i corresponds to an object type with its corresponding extension. The extension is the table of object type. That is to say: the object type and its extension are represented in the UML extension just as a << <i>Object Type</i> >> class.</p>	
<p>Comments This extension considers the object-relational model of SQL:1999 and Oracle8i. It should be modified according to the new versions of both object-relational models. For example, in Oracle9i [21], multilevel collections are allowed, so any collection type (<< <i>Array</i> >> or << <i>NT</i> >>) can be used as a type of any collection type. Moreover, if we wish to use other products this extension has to be adapted.</p>	

it permits the design of the entire system, facilitating integration between different system views.

The **design** phase is divided into two steps:

- Standard design, that is, a logical design independent of any product.
- Specific design, that is, the design for a specific product (for example, Oracle8i) without considering tuning or optimization tasks.

Standard design is especially important in object-relational database design because each product implements a different object-relational model. This phase provides an object-relational specification independent of the product improving the database maintainability as well as making migration between products easier. As is shown in Fig. 3, we propose two alternative techniques for this phase: defining the schema in SQL:1999,

because it does not depend on any specific product; and/or using a graphical notation describing a standard object-relational model (the SQL:1999 model). This graphical notation corresponds with the one that represents the logical design of a relational database [2]. As graphical notation we propose to use the UML extensions defined in Sect. 3 for the SQL:1999 object-relational model.

For the specific design (intermediate stage between design and implementation), we have to specify the schema in the SQL (language) of the chosen product. We use, as an example, Oracle8i. Moreover, we also propose to make optional use of a graphical technique to improve the documentation and the understandability of the SQL code generated. Moreover, this technique makes the compilation of the database schema easier by showing the correct order in which we have to compile each new user-

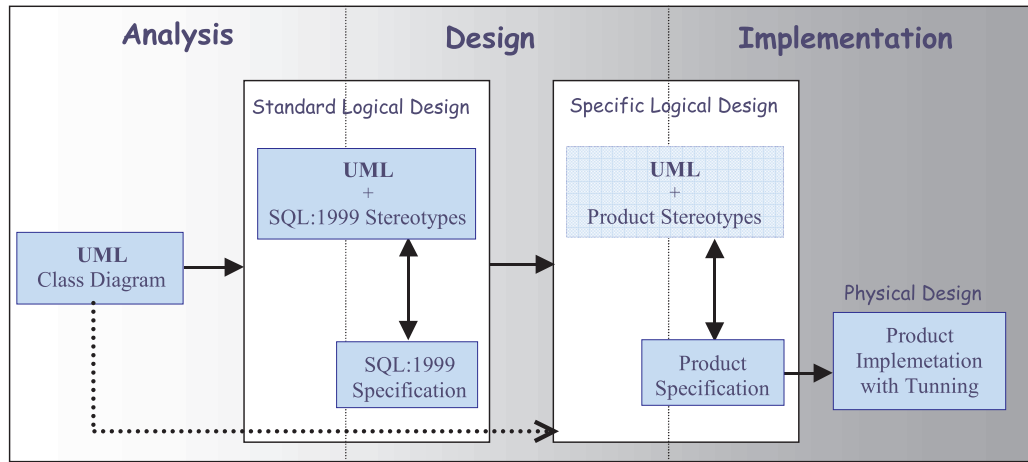


Fig. 3. Object-Relational Database Design Methodology

defined type and each table. The graphical notation is also UML replacing the SQL:1999 stereotypes with the specific stereotypes for the selected product.

Finally, the **implementation** phase includes some physical design tasks. In this phase the schema obtained in the previous one should be fine-tuned to improve the response time and storage space according to the specific needs of the application.

Relational database methodologies propose some rules for transforming a conceptual schema into a standard logical schema. In the same way, our methodology also proposes a technique that permits the transformation of a schema from one phase to the next. The technique suggests guidelines which are detailed with an example in the next subsection.

4.1 Transformation Guidelines

Table 3 summarizes some transformation guidelines that could be used in the methodology [15]. As we have al-

ready said, Oracle9i supports the Generalization concept, so the corresponding rules should be modified [22].

These rules are explained below through an application example developed for a Web environment using XML and Oracle8i to resolve the problem of computer reservations in the University, enabling lecturers to reserve computer classrooms for their classes. A classroom is composed of a set of computers, and students can only reserve them if no teacher or other student has previously reserved them.

To develop the application we have applied the methodology described following the phases shown in Fig. 3. The whole example is worked through in Sect. 4.2.

4.1.1 Transformation of Classes, Attributes and Methods

Only persistent classes have to be transformed into a class of the database schema. A persistent class in UML is marked with the stereotype `<< persistent >>` (or, in Rational Rose notation, with `<< schema >>`). To transform

Table 3. Guidelines for object-relational database design

UML	SQL:1999	Oracle8i
Class	Structured Type	Object Type
Class Extension	Typed Table	Table of Object Type
Attribute	Attribute	Attribute
Multivalued	ARRAY	VARRAY/Nested Table
Composed	ROW / Structured Type in column	Object Type in column
Calculated	Trigger/Method	Trigger/Method
Association		
One-To-One	REF/[REF]	REF/[REF]
One-To-Many	[REF]/[ARRAY]	[REF]/[Nested Table/VARRAY]
Many-To-Many	ARRAY/ARRAY	Nested Table/Nested Table VARRAY/VARRAY
Aggregation	ARRAY	Nested Table/VARRAY of References
Composition	ARRAY	Nested Table/VARRAY of Objects
Generalization	Types/Typed Tables	Oracle8i cannot directly represent the generalization concept

a UML persistent class into a SQL:1999 or Oracle8i class, it is necessary to define the object type as well as its extension. An object type in SQL:1999 is defined as a structured type, and its extension is defined as a table of that object type. A UML persistent class is translated into Oracle8i in the same way as into SQL:1999. They only differ in the syntax of the structured type (specified in Oracle8i by “AS OBJECT”).

Each attribute of a UML class is transformed into an attribute of the type. Neither SQL:1999 nor Oracle8i supports visibility levels, so in design and implementation phases they disappear. Visibility levels should be implemented by defining views or privileges, etc.

Multivalued attributes are represented in SQL:1999 and Oracle8i with a collection type. The only collection type supported by the standard SQL:1999 is the ARRAY type, whereas in Oracle8i it is possible to choose between the VARRAY and the nested table types. Use of VARRAY is recommended if the maximum number of elements is known and small, and if entire collections are to be retrieved; if the number of values is unknown, or very uncertain or if efficient queries are to be made, a nested table is recommended.

Composed attributes can be transformed into an SQL:1999 ROW type and into an Oracle8i object type without extension (that is, by defining the object type and not specifying the associated table).

Derived attributes can be implemented by means of a trigger or a method in both models.

Each UML class method is transformed into SQL:1999 and Oracle8i by specifying the signature of the method in the definition of the object type. In this way the method is joined to the type to which it belongs. The body of the method is defined separately.

Figure 4 shows an example of class transformation into SQL:1999 and Oracle8i. Figure 4a shows a class representing a teacher. Figure 4b and c are the corresponding standard and specific schemas, which, in this case, are identical.

4.1.2 Association Transformation

UML associations can be represented in an object-relational schema either as unidirectional relationships or as bidirectional relationships. If we know that queries require data in both directions of the association, then their implementation as bidirectional relationships would be recommended, thus improving the response time. However, we have to take into account that bidirectional relationships are not maintained by the system, so consistency has to be guaranteed by means of triggers or methods. Therefore two-way relationships, despite sometimes improving response times, have a higher maintenance cost. The navigability (if it is represented) in a UML diagram shows the direction in which the association should be implemented.

Depending on the maximum multiplicity of the two classes involved in an association (*One* or *Many*), we propose the following transformation rules (considering bidirectional associations): Maximum *One* multiplicity would be implemented through an REF type attribute in the object type involved in the association. If the minimum multiplicity were also one, it would be necessary to impose the NOT NULL restriction to the REF attribute in the corresponding typed table (because the restrictions have to be defined in the table rather than in the object type). Maximum *Many* multiplicity would be transformed including an attribute of collection type in the

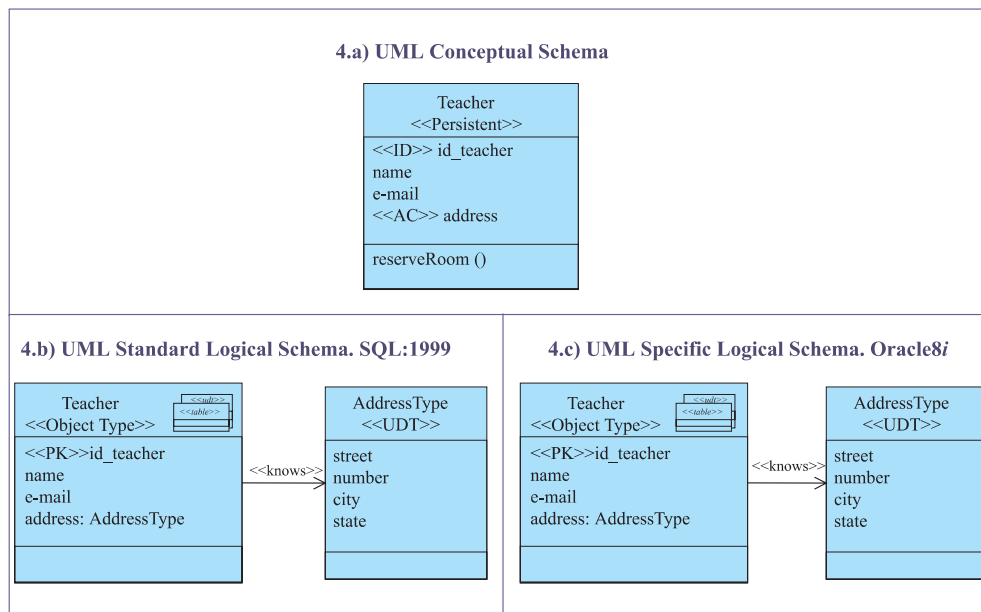


Fig. 4. Transformation of classes, attributes and methods

object type that participates in the association. The collection types contain references (REF type) to the other object type involved in the association. In SQL:1999 the collection type is an ARRAY (because it is the only collection type supported by the standard). However, in Oracle8i it is possible to use nested tables instead of VARRAYS because this constructor permits the maintenance of unbounded collections of elements. If the maximum cardinality were known (for example, suppose that a classroom could not contain more than 50 computers) then it would be more advisable to use a VARRAY. Other possible transformations may be advisable in some cases. For example, in the case One-to-Many, a REF type could be used in the object type that participates with multiplicity *Many* (as usual), and, instead of using a collection type in the object type with multiplicity *One*, a method could maintain this direction of the association.

Another possibility, of course, is to use traditional relational foreign keys.

Figure 5 shows an example of a one-to-many association transformation as a bidirectional association.

4.1.3 Generalizations Transformation

SQL:1999 supports generalization of types and generalization of typed tables. The former allows implementing the inheritance concept associated to a generalization; the latter allows implementing the subtype concept (every object that belongs to the subclass also belongs to

the superclass) that is also associated to a generalization. The definition is made including the UNDER clause in the specification of each subtype indicating its supertype (only simple inheritance is allowed). It is also necessary to specify the corresponding hierarchy of tables by means of the UNDER clauses in the corresponding subtables.

Oracle8i does not support inheritance. Therefore, generalizations are implemented by means of foreign keys, as in the relational model, or REF types. Besides, it is necessary to specify restrictions (CHECK, assertions and triggers) or operations that permit the simulation of their semantics.

Oracle9i seems to solve this issue. As we have explained in Sect. 3.1, it supports inheritance of types and inheritance of views [22].

4.1.4 Aggregation (and Composition) Transformation

To represent this kind of aggregation in an object-relational model we propose the inclusion of an attribute of the collection type in the definition of the *whole* type. This collection contains references to its *parts*, that is, references to the objects that compose the *whole*. In SQL:1999 the collection is defined by means of an ARRAY of references. In Oracle8i the collection is defined by means of a VARRAY. Besides, in Oracle8i it could also be defined as a nested table. If the maximum number of components (maximum cardinality) is known (for example, suppose that a classroom could not have more

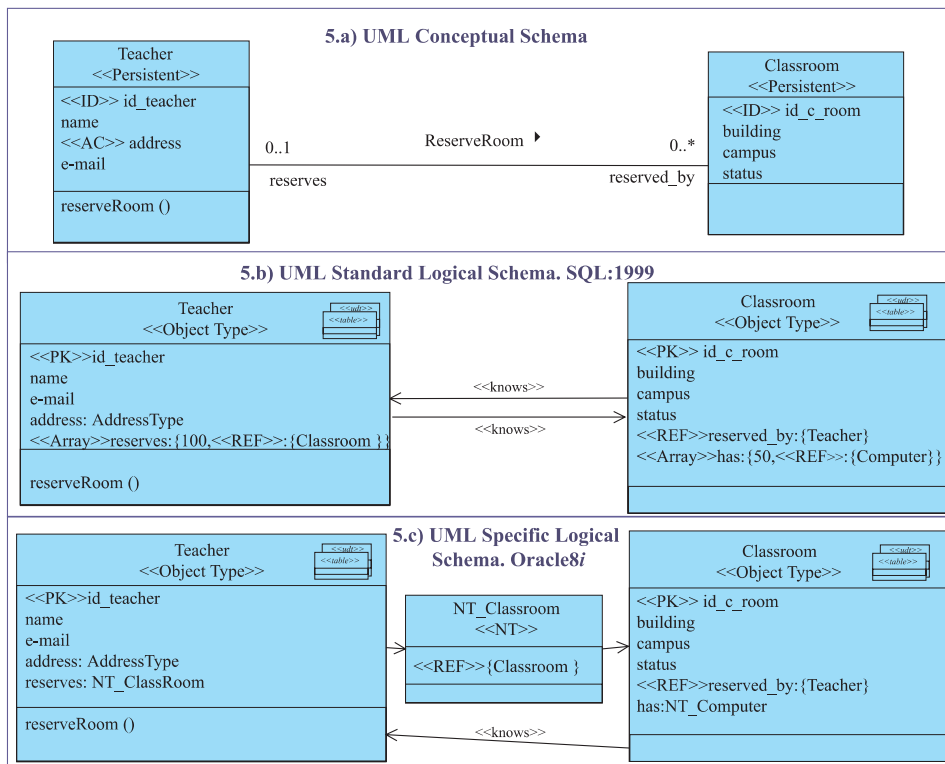


Fig. 5. Association transformation

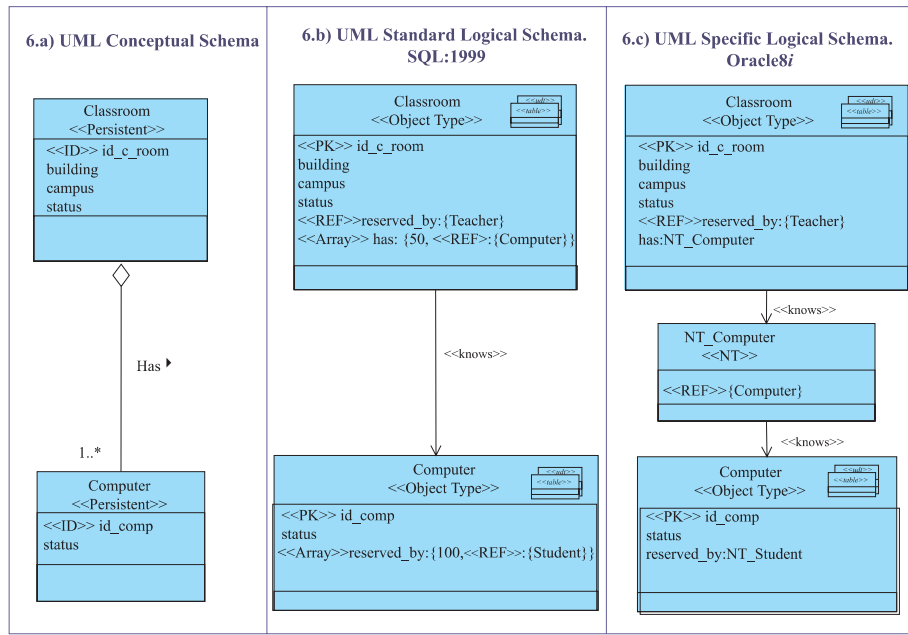


Fig. 6. Aggregation transformation

than 10 computers) then it would be more advisable to use a VARRAY.

Regarding compositions (a special kind of aggregation in which the *parts* are physically linked to the *whole*), these do not differ from aggregation implementations in the case of SQL:1999, because it only provides the ARRAY collection type. The special restrictions of being a composition (a part cannot belong to more than one whole; if a whole is removed, all its parts are removed, too, etc.) must be controlled by means of checks, assertions and/or triggers. However, in Oracle8i it is possible to implement directly the concept of composition maintaining the differences with

regard to the aggregation concept [15]. This is because Oracle8i also provides the nested table. A nested table is a collection type but it is also a table and, as such, can be defined as an object table. So, the nested table can contain the *parts* as objects rather than as references. At the same time the nested table is embedded in a column of another object table (the *whole*). However, in this case, component objects are not real objects (they have no OID and can not be referenced). Therefore, we propose to implement the composition in Oracle8i in the same way as the aggregation, that is, with a nested table of references, in spite of losing semantics.

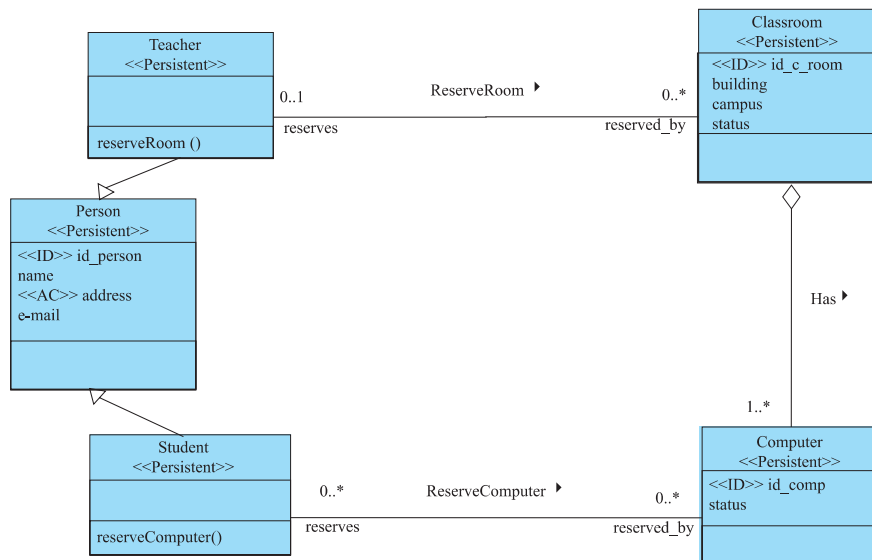


Fig. 7. Conceptual Design in UML

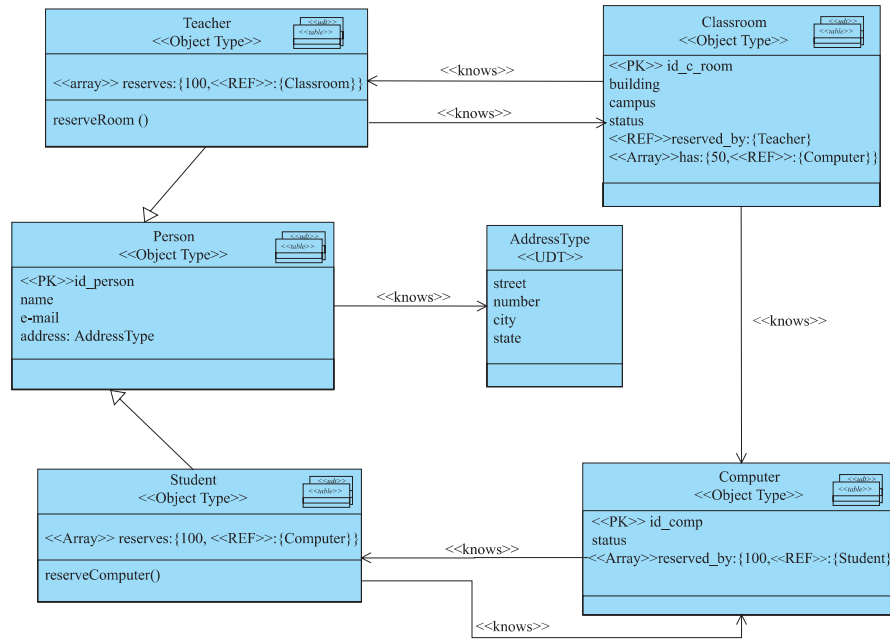


Fig. 8. Logical Design in SQL:1999

Figure 6 shows an aggregation between a classroom and its computers. It has been defined in SQL:1999 (a) by means of a collection of references to computers, and in Oracle8i (c) by means of a nested table.

4.2 The Computer Reservation Application

In this section we describe the entire example of the case for study described in Sect. 4.1, according to the

phases outlined in Fig. 3. **Analysis phase:** Fig. 7 shows the UML class diagram used to design the conceptual schema.

Analysis-Design phase: Figure 8 shows the standard design with the proposed graphical notation, consisting in the defined UML extensions for the SQL:1999 object-relational model.

Design-Implementation phase: Figure 9 shows the specific design (intermediate stage between design

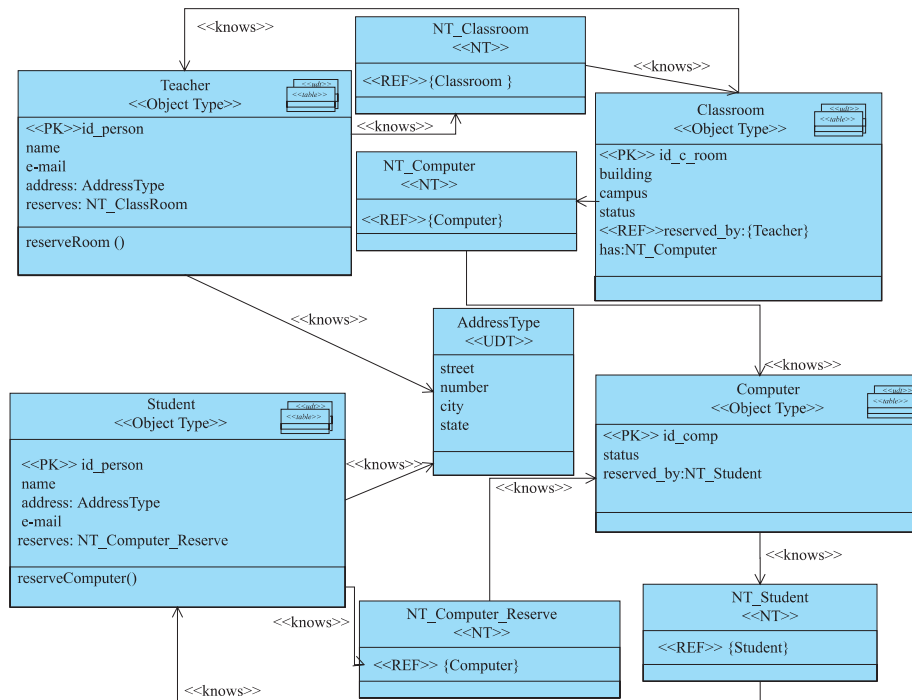


Fig. 9. Logical Design in Oracle8i

```

CREATE OR REPLACE TYPE AddressType AS OBJECT
( street VARCHAR(20)
, num NUMBER
, city VARCHAR(20)
, state VARCHAR(10))
/
CREATE OR REPLACE TYPE Computer AS OBJECT
(id_comp VARCHAR(5)
, status VARCHAR(2)
, reserved_by NT_Student)
/
CREATE OR REPLACE TYPE NT_Computer
AS TABLE OF REF Computer
/
CREATE OR REPLACE TYPE NT_Computer_Reserve
AS TABLE OF REF Computer
/
CREATE OR REPLACE TYPE Student AS OBJECT
( id_person VARCHAR(5)
, name VARCHAR(50)
, address AddressType
, e_mail VARCHAR(100)
, reserves NT_Computer_Reserve)
/
CREATE OR REPLACE TYPE NT_Student
AS TABLE OF REF Student
/
CREATE OR REPLACE TYPE Teacher AS OBJECT
( id_person VARCHAR(5)
, name VARCHAR(50)
, e_mail VARCHAR(100)
, address AddressType
, reserves NT_ClassRoom)
/
CREATE OR REPLACE TYPE Classroom AS OBJECT
(id_c_room VARCHAR(5)
, building VARCHAR(50)
, campus VARCHAR(10)
, status VARCHAR(2)
, reserved_by REF Teacher
, has NT_Computer)
/
CREATE OR REPLACE TYPE NT_ClassRoom
AS TABLE OF Ref Classroom
/
/* Recompile */
CREATE OR REPLACE TYPE Computer AS OBJECT
(id_comp VARCHAR(5)
, status VARCHAR(2)
, reserved_by NT_Student)
/
CREATE OR REPLACE TYPE Teacher AS OBJECT
( id_person VARCHAR(5)
, name VARCHAR(50)
, e_mail VARCHAR(100)
, address AddressType
, reserves NT_ClassRoom)
/
/* Table Creation*/
CREATE TABLE TClassroom OF Classroom
(PRIMARY KEY (id_c_room))
NESTED TABLE has STORE AS table_computer_cr;
CREATE TABLE TComputer OF Computer
(PRIMARY KEY (id_comp))
NESTED TABLE reserved_by STORE AS table_student;
CREATE TABLE TStudent OF Student
(PRIMARY KEY (id_person))
NESTED TABLE reserves STORE AS table_computer_s;
CREATE TABLE TTeacher OF Teacher
(PRIMARY KEY (id_person))
NESTED TABLE reserves STORE AS table_classroom;

```

Fig. 10. Oracle8i Code

and implementation phases), with the proposed graphical notation. The graphical notation is also UML replacing the SQL:1999 stereotypes with the specific ones for the selected product. We have also specified the obtained schema in the SQL (language) of the chosen product, Oracle8i, as Fig. 10 shows.

5 Conclusions and Future Work

Object-relational databases are going to be increasingly used because they provide a better support than relational technology for complex data and relationships. Consequently, new methodologies for object-relational database design are emerging. In this paper we have summarized a methodology for object-relational database design, which is based on UML extended with the required stereotypes. We have focused on object-relational databases, although the UML proposed extensions could also be adapted for object database design.

The methodology proposes three phases: analysis, design and implementation. As a conceptual modelling technique we have chosen the UML class diagram. As the logical model we have used the SQL:1999 object-relational model, so the guidelines are not dependent on the implementations of object-relational products. As a product ex-

ample we have used Oracle8i, although the main improvements of Oracle9i affecting this study have been explained (fundamentally inheritance and multilevel collections).

Traditionally, methodologies provided graphical techniques to represent a relational schema, such as Data Structured Diagrams (DSD), relational “graph”, etc. In the same way, an object-relational schema can be represented either in SQL (SQL:1999 or Oracle8i) or by means of some graphical notation. As graphical notation to represent the logical schema we have proposed the use of the UML class diagram extended with the required stereotypes, tagged values and/or constraints. This paper has focused on the UML extensions required for object-relational design in both models, SQL:1999 and Oracle8i, and on some guidelines for the transformation from UML conceptual schema into logical schemas. We have summarized the stereotypes, tagged values and constraints. In the case of Oracle8i, the graphical notation for the defined stereotypes helps the developer in the compilation process by showing the right order in which new user defined types and tables must be compiled. We have also explained part of one of the applications developed using the UML extension: the computer classroom reservations for the *Rey Juan Carlos* University.

We are now working on the integration of this work in a methodology for Web Database Design [15].

Acknowledgements. This work is being carried out as part of the MIDAS project. MIDAS is partially financed by the Spanish Government (CICYT) and the European Community (reference number: 2FD97-2163).

References

1. Ambler, S.: *The Object Primer*. Cambridge University Press, 2nd edition, 2001
2. Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R.: *Database Systems. Concepts, Languages and Architectures*. McGraw-Hill, 1999
3. Bertino, E. and Marcos, E. *Object: Oriented Database Systems*. In: O. Díaz and M. Piattini (eds.): *Advanced Databases: Technology and Design*. Artech House, 2000
4. Bertino, E. and Martino, L.: *Object-Oriented Database Systems. Concepts and Architectures*. Addison-Wesley, 1993
5. Blaha, M. and Premerlani W.: *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998
6. Booch, G., Rumbaugh, J., and Jacobson, I.: *The Unified Modeling Language User Guide*. Addison Wesley, 1999
7. Case, T., Henderson-Sellers, B., and Low, G.C.: A generic object-oriented design methodology incorporating database considerations. *Annals of Software Engineering*, 2: 5–24, 1996
8. Cattell, R.G.G. and Barry, D.K.: *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000
9. Conallen, J.: *Building Web Application with UML*. Addison-Wesley, 2000
10. Eisenberg, A. and Melton, J.: *SQL:1999, formerly known as SQL3*. *ACM SIGMOD Record*, 28: 131–138, March 1999
11. Kovács, C. and Van Bommel, P.: Conceptual modeling-based design of object-oriented databases. *Information and Software Technology*, 40: 1–14, 1998
12. Leavit, N.: Whatever Happened to Object-Oriented Databases?. *Computer*, pp. 16–19, August 2000
13. Mattos, N.M.: *SQL:1999, SQL/MM and SQLJ: An Overview of the SQL Standards*. Tutorial, IBM Database Common Technology, 1999
14. Marcos, E. and Cáceres, P.: Object Oriented Database Design. In: Shirley Becker (ed.): *Developing Quality Complex Database Systems: Practices, Techniques, and Technologies*. Idea Group, 2001
15. Marcos, E., Cáceres, P., Vela, B. and Cavero, J.M.: MIDAS/BD: a Methodological Framework for Web Database Design, International Workshop on Data Semantics in Web Information Systems, DASWIS-2001, in conjunction with 20th International Conference on Conceptual Modeling (ER 2001), Yokohama, Japan, November 2001
16. Marcos, E. Vela, B., Cavero, J.M. and Cáceres, P.: Aggregation and composition in object-relational database design. *Advanced Databases and Information Systems. Communications of the 5th East-European Conference ADBIS' 2001*. Vilnius, Lithuania, September 2001
17. Marcos, E. Vela, B. and Cavero, J.M.: *Extending UML for Object-Relational Database Design*. UML 2001. Springer Verlag. *Lectures Notes in Computer Science*. LNCS 2185: 225–239, 2001
18. Muller, R.: *Database Design for Smarties*. Morgan Kaufmann, 1999
19. Naiburg, E.J., Maksimchuk, R.A.: *Uml for Database Design*, Addison-Wesley, 2001
20. Oracle Corporation *Objects and SQL in Oracle8*. Oracle Technical White paper. In *Extended DataBase Technology conference (EDBT'98)*. Valencia (Madrid), 1998
21. Oracle Corporation. *Oracle8i. SQL Reference*. Release 3 (8.1.7). In: www.oracle.com, 2000
22. *Simple Strategies for Complex Data: Oracle9i Object-Relational Technology*. Oracle Technical White Paper. In: http://otn.oracle.com/products/oracle9i/pdf/ort_fo.pdf, 2001
23. Silva, M.J.V. and Carlson, C.R.: MOODD, a method for object-oriented database design. *Data & Knowledge Engineering*, 17: 159–181, 1995
24. Stonebraker, M. and Brown, P.: *Object-Relational DBMSs. Tracking the Next Great Wave*. Morgan Kauffman, 1999
25. Ullman, J. and Widom, J.: *A First Course in Database Systems*. Prentice-Hall, 1997