Aggregation and Composition in Object - Relational Database Design

E. Marcos, B. Vela, J. M. Cavero, P. Cáceres

Kybele Research Group
Rey Juan Carlos University
Madrid (Spain)
{cuca, b.vela, j.m.cavero, p.caceres}@escet.urjc.es

Abstract. Recently, there have emerged more sophisticated applications, which have to represent complex objects related with complex relationships, such as aggregations and compositions. New object-relational databases are more appropriated than relational databases to support complex objects. Besides, the most common way of designing databases is using the E/R model, without taking in account the program design. However, new object-oriented design techniques, such as UML (Universal Modelling Language), permit modelling the full system, including the database schema, in a uniform way. Besides, as UML is an extensible language, it allows introducing new stereotypes for specific applications if it is needed. So, new stereotypes for database design can be also defined. The framework of this paper is an Object-Relational Database Design Methodology. The methodology specifies new UML stereotypes for Object-Relational Database Design as well as it proposes some guidelines to translate an UML schema into an object-relational one. The guidelines are based on the SQL:1999 object-relational model and in Oracle8*i* as an example of product. In this paper we focus on the design of two UML constructors: aggregation and composition. We propose its implementation in Oracle8*i*, because unlike SQL:1999, Oracle8*i* supports a data type, the nested table, that is specially appropriated to represent the UML aggregation and composition.

Keywords: Aggregation, Composition, Nested Table, Database Design, Object-Relational Databases, Design Methodology, UML, SQL:1999, Oracle8*i*

1 Introduction

In spite of the impact of relational databases in last decades, this kind of databases has some limitations to support data persistence required by actual applications. Due 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. These applications can be characterised as consisting of complex objects related by complex relationships. Representing such objects and relationships in the relational model implies that the objects must be decomposed into a large number of tuples. Thus, a considerable number of joins is 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 include the object-relational [23] 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 traditionally has been done with relational databases. In last years some approaches to object-oriented database design have appeared [5,13,17,21,24]. Unfortunately, none of these proposals can be considered as “the method”, neither for object-relational nor for object databases. On the one hand, they do not consider last versions of the representative standards for both technologies: ODMG 3.0 for object databases [8] and SQL:1999 for object-relational databases [9]. And, on the other hand, some of them are based on techniques as OMT [5] or, even, on the E/R model [24]. So, they have to be updated considering UML [7], SQL:1999 and ODMG 3.0 as their reference models.

Object databases are well suited for storing and retrieving complex data by allowing the user to navigate through data. However, object-relational technology, that is, relational technology extended with new capabilities, such as triggers, methods, user defined types, etc, presents two advantages compared with object databases: it is compatible with relational technology and provides a better support for complex applications. Therefore, object-relational databases are expected to have a bigger impact in the market than object databases [14]. For these reasons in this work we focus on object-relational databases design.

In this paper we propose a methodology for object-relational database design. As conceptual modelling technique we have chosen the UML class diagram. UML, as a Universal Modelling Language, is every day more accepted. It also presents the advantage of being able to model the full system, including the database model, in a uniform way. Besides, as UML is an extensible language, it is possible to define the required stereotypes for specific applications. The methodology provides some guidelines to translate a conceptual schema (in UML notation) into a logical schema. As logical model we use the SQL:1999 object-relational model so that the guidelines were not dependent of the different implementations of object-relational products. We use Oracle8*i* as an implementation example.

In this paper we focus on aggregation and composition design. In the framework of our methodology, we propose specific guidelines to design aggregations and compositions in an object-relational model.

Although the methodology, as we have explained above, is mainly based in SQL:1999, in this paper we focus on the aggregation and composition implementation in Oracle8*i*. The reason is that this product supports a collection data type, the nested table, which is specially appropriated to implement aggregations and compositions. This collection data type is not provided neither by SQL:1999 nor by other products, such as Informix Universal Server [10].

The rest of the paper is organised as follows: section 2 summarises the SQL:1999 and Oracle8*i* object-relational models; section 3 is an overview of aggregation and composition in UML; section 4 sums up the methodology focus, in section 5 the design differences between aggregation and composition are explained; finally, section 5 summarises the main conclusions and future work.

2 Object-Relational Model

In this section we summarise the object model of the current standard for object-relational databases, SQL:1999 [9,15], as well as the main characteristics of Oracle8*i* object-relational model, as an example of object-relational product [19,20]. SQL:1999 data model extends the relational data model with some new constructors to support objects. Most of last versions of relational products include some object extensions. However, and because in general these products have appeared in the market before the standard approval, current versions of object-relational products do not totally adjust to the SQL:1999 model.

2.1 Object model of the SQL:1999

SQL:1999 is the current standard for object-relational databases. Its data model tries 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. So, a column of an object table can contain a collection data type.

SQL:1999 allows 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 representing 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), being the table the extension of the type. In SQL:1999 these kinds of 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 system generated, it is unique for each object and the user cannot modify it. 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 a column of a table) whereas in (b) it is used as an object type (as the type of a table).

```
CREATE  TYPE employee AS (
        id          INTEGER,
        name        VARCHAR(20))
```

column1	column2	**employee**

(a)Structured type as column type

OID	**id**	**name**

(b)Structured type as object type

Fig. 1. Structured types used as value and object types [15]

A structured type can include associated methods representing its behaviour. A method is a SQL function, whose signature is defined next to the definition of the structured type. The body specification is defined separately of 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, restrictions, 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 reference type holds the OID of the referred object. So, the REF type permits implementing 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 a collection type: ARRAY. The ARRAY 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 representing multivalued attributes not forcing tables to be in 1NF.

2.2 Object model of Oracle8*i*

As well as SQL:1999, Oracle8*i* supports structured data types that can be defined by the user (although, with a different syntax). 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, being the table the extension of the type. Each row of this kind of tables is an object and, in the same way as in SQL:1999, they have a special column of reference type (REF) that allows identifying each object (OID). It is also possible to define an attribute as a reference to an object type. Oracle8*i* allows associating behaviour to object types, defining the signature of the methods as part of the type definition. The body of the method is defined separately.

Oracle8*i* 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. So, this column contains a table (called nested table) with a collection of values, objects or references. Figure 2 shows an example of nested table (**C_Table**).

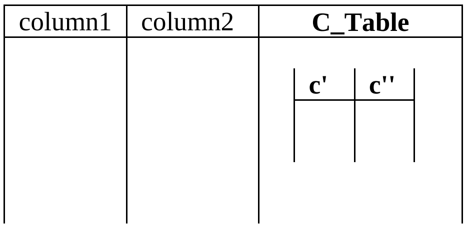


Fig.2. A nested table in Oracle8*i*

One of the main differences between Oracle8*i* and SQL:1999 object-relational model is that Oracle8*i* does not support inheritance, neither of types nor of tables. There exist, however, some relational products, as for example, Universal Server of Informix [10], those support the inheritance concept in a similar way as the standard.

However, another difference that make Oracle8*i* more powerful than SQL:1999 is related with the collection types. The nested table, not supported by SQL:1999, allows to represented an object collection embedded in another object, that could be a natural way to implement the UML aggregation (in sections 3 and 5 we will see the differences between aggregation and composition).

3 Aggregation and Composition in UML

An aggregation is a special form of association between classes that represents the concept of “WHOLE -PART”. Each object of one of the classes that belong to the aggregation (the composed class) is composed of objects of the other class of the aggregation (the component class). The composed class is often called "whole" and the component classes are often called "parts". An intuitive example of aggregation is the relationship between a wood and its threes. The wood can be considered as the *whole* and the threes would be the *parts* that belong to the wood.

Aggregation has been briefly treated in the literature and different classifications of aggregations have been proposed [11,12,22,25]. However, UML distinguishes only between two kind of aggregation: simple aggregation and composed aggregation.

3.1 Aggregation

A simple aggregation is an aggregation where each part can be part of more than one *whole*. This kind of aggregation is very common in the literature and has been often refereed as *logical or catalogue aggregation*, even in the first drafts of UML [6]. As an example of simple aggregation we can think in the catalogue of dolls of the "ToysX" store that contains n Barbie models.

However, the same Barbie models can appear in the catalogue of dolls of different toy-stores. This is possible because there exists a logical aggregation, and the dolls do not compound physically the catalogues. Figure 3 shows a simple aggregation. It is represented placing a diamond next to the whole class.

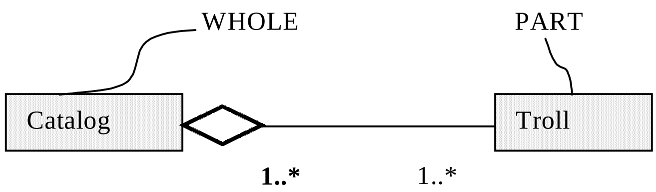


Fig.3. Simple Aggregation Example

Simple aggregation does not imply any kind of restriction over the life of the parts with regard to its whole.

3.2 Composition

A composition, also called composed aggregation, is a special kind of aggregation in which the *parts* are physically included in the *whole*. Once a part has been created it lives and dies with its whole. A *part* can be explicitly removed before removing its associated *whole*. As it is a physical aggregation, a *part* can only belong to a *whole*. Figure 4 shows an example of composition. University is the *whole* and departments are its *parts*. The live of a department depends on the live of the university to which it belongs. If the university disappears its departments disappears as well. Besides, a department can be joined only to a university. The representation of the composition is similar to the representation of the simple aggregation. The only difference is that the diamond is fulfilled.

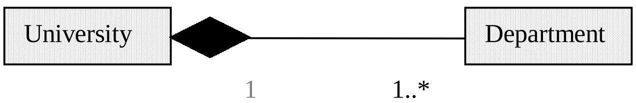


Fig.4. Composition Example

4 Object-Relational Database Design

The proposed methodology for object-relational database design is based on the proposal of Bertino and Marcos [3] for object-oriented database design and on the proposal of Marcos and Cáceres [16]. Figure 5 summarises the main steps of the methodology.

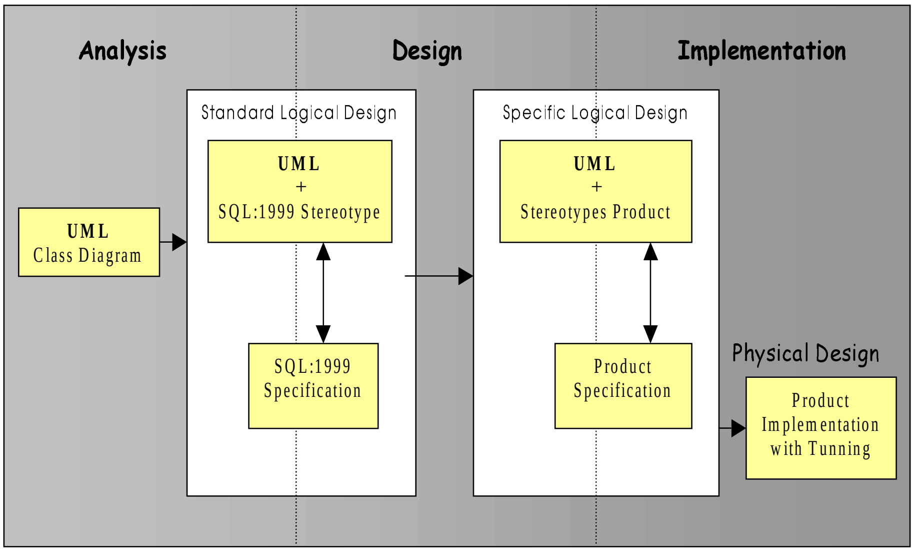


Fig. 5. Object-Relational Database Design Methodology

The methodology proposes three phases: analysis, design and implementation. Nonetheless, as it is shown in figure 5, differences between analysis, design and implementation phases are not as strong as in structured design.

At the **analysis** phase, we propose to use the UML class diagram 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 it allows to design the entire system making easier the 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 (Oracle8*i*, Informix, etc.) without considering tuning or optimisation 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 easier migration between products. As it is shown in figure 5, 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 relational graph that represents the logical design of a relational database [2]. As graphical notation, and due to UML can be extended, we propose to use UML extended with the required stereotypes for the SQL:1999 object-relational model. In this paper, we leave out the design graphical representation.

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. Besides, we also propose to use optionally a graphical technique to improve the documentation and the understandability of the generated SQL code. The graphical notation is also UML substituting the SQL:1999 stereotypes with the specific stereotypes for the selected product.

Finally, the **implementation** phase includes the physical design tasks. In this phase the schema obtained in the previous phase should be refined, making a tuning to improve the response time and storage space according to the specific needs of the application.

Relational database methodologies propose some rules to transform a conceptual schema into a standard logical schema. In the same way, we also propose a technique that allows transforming a schema from one phase to the next. This technique suggests some rules that have to be considered only as guidelines.

These rules are summarised in table 1.

Table 1. Guidelines for object-relational database design.

UML	SQL:1999	Oracle8*i*
Class	Structured Type	Object Type
Class Extension	Typed Table	Table of Object Type
Attribute	Attribute	Attribute
Multivalued	ARRAY	VARRAY
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
Many-To-Many	ARRAY/ARRAY	Nested Table/Nested Table
Aggregation	ARRAY	Nested Table
Generalisation	Types/Typed Tables	Oracle cannot represent directly the generalisation concept

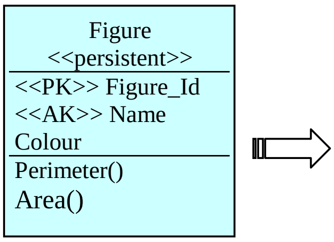
Class Transformation. Only persistent classes have to be transformed into a class of the database schema. A persistent class in UML is marked with the stereotype <<persistent>> (or, in Rational Rose notation with <<schema>>). To transform a

UML persistent class into a SQL:1999 or Oracle8*i* 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 table of the aforementioned object type. A UML persistent class is translated into Oracle8*i* in the same way as into SQL:1999. They only differ in the syntax of the structured type (in Oracle8*i* the structured type specifies ”AS OBJECT"). Figure 6 shows an example of a UML persistent class and its corresponding specification in SQL:1999 and Oracle8*i*.

SQL:1999

```
CREATE TYPE Figure AS
( Figure_Id INTEGER,
  Name VARCHAR(30),
  Colour VARCHAR(15));

CREATE TABLE Table_Figure OF Figure
(PRIMARY KEY (Figure_Id),
 UNIQUE (Name));
```



Oracle&*i*

```
CREATE TYPE Figure AS OBJECT
( Figure_Id INTEGER,
  Name VARCHAR(30),
  Colour VARCHAR(15));

CREATE TABLE Table_Figure OF Figure
(PRIMARY KEY (Figure_Id),
 UNIQUE (Name));
```

Fig.6. Transformation of classes

Attribute and Method Transformation. Each attribute of a UML class is transformed into an attribute of the type. Nor SQL:1999 neither Oracle8*i* support 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 Oracle8*i* with a collection type. In SQL:1999 the collection type is the ARRAY type, because it is the only collection type supported by the standard, whereas in Oracle8*i* it is possible to choose between the VARRAY and the nested table types. Using the VARRAY is recommended if the maximum number of elements is known; if the number of values is unknown, or very uncertain, it is recommended to use a nested table. We can notice that the possibility of defining multivalued attributes without additional tables, eliminates one the first rules in a relational database design: the first normal form is mandatory in every table.

Composed attributes can be represented in the object-relational model without creating an associated table, transforming it into a SQL:1999 ROW type and into an

Oracle8*i* object type without extension (that is, defining the object type and not specifying the associated table).

Derived attributes can be implemented by means of a trigger or by means of a method in both models, SQL:1999 and Oracle8*i*.

Each UML class method is transformed into SQL:1999 and Oracle8*i* 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.

Association Transformation. UML associations can be represented in an object-relational schema either as unidirectional relationships or as bi-directional relationships. A unidirectional association means that the association can be cross only in one direction whereas a bi-directional association can be crossed in the two directions. If we know that queries require data in both directions of the association then it could be recommended to implement them as bi-directional relationships improving in this way the response times. However, we have to take into account that bi-directional relationships are not maintained by the system, so the consistence has to be guaranteed by means of triggers or methods. Therefore two-way relationships, despite of improving in some cases the 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, we propose the following transformation rules (considering bi-directional associations):

- **One-to-One.** It would be implemented through a REF type attribute in each object type involved in the association. If the minimum multiplicity were 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).
- **One-to-Many.** It would be transformed including a REF type attribute in the object type that participates in the association with multiplicity N and including an attribute of collection type in the object type that participates with multiplicity one. The collection types contain references (REF type) to the other object type involved in the relationship. In SQL:1999 the collection type is an ARRAY (because it is the only collection type supported by the standard). However, in Oracle8*i* it is possible to use nested tables instead of VARRAYS because this constructor allows maintaining collections of elements without a predefined dimension. If the maximum cardinality were known (for example, suppose that a plain could not contain more than 10 figures) then it would be more advisable to use a VARRAY.
- **Many-to-Many.** Following the same reasoning that in the previous case, a many-to-many association would be transformed into SQL:1999 defining an ARRAY attribute in each object type involved in the relationship. In Oracle8*i* VARRAYs should be replaced by Nested Tables.

If the association represents the navigability then it would be implemented as we have explained above, but just in one direction. Therefore the attribute of REF type (or the collection of REF type) would be defined only in a class.

Generalisation Transformation. SQL:1999 supports generalisation of types and generalisation of typed tables. The first one allows implementing the inheritance concept associated to a generalisation; the second one allows implementing the subtype concept (every object that belongs to the subclass also belongs to the superclass) that is also associated to a generalisation. 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.

Oracle8*i* does not support inheritance. Therefore, generalisations are implemented using foreign keys, as in the relational model, or using REF types. Besides, it is necessary to specify restrictions (CHECK, assertions and triggers) or operations that permit to simulate their semantics.

There exist, however, some commercial products that implement inheritance such as Informix Universal Server. Although Informix does not support the entire semantics of the inheritance concept (for example, tables inherit only attributes), it allows to define it and supports some of its characteristics. It is expected that future versions of object-relational products will include inheritance. Meanwhile, when an object -relational database is being designed it is important to specify the SQL:1999 schema in order to maintain the semantics that is lost in the implementation phase due to lacks of the product models.

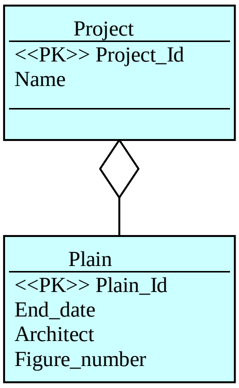
5 Aggregation and Composition Design

In this section we present the rules to transform UML simple aggregations and compositions into SQL:1999 and Oracle8*i*, discussing the main differences between both of them.

5.1 Simple Aggregation Design

To represent this kind of aggregation in an object-relational model we propose to include in the definition of the *whole* type an attribute of collection type. This collection contains references to its *parts*, that is, references to the objects that compound the whole. For example, in figure 7, we can see an aggregation between a project and its plains. As we can see, it has been defined in SQL:1999 and Oracle8*i* including an attribute *Has_plain* in the definition of the class *project*. This attribute is a collection of references to class *plain*. In SQL:1999 the collection is defined by means of an ARRAY of references. In Oracle8*i* the collection should be a nested table. If the maximum number of components (maximum cardinality) is known (for example, suppose that in a plain there could not be more than 10 figures) then it would be more advisable to use a VARRAY.

We propose to define the collection as a set of references, because, as we have explained in section 3, a simple aggregation is an aggregation where each *part* can be part of more than one *whole*. It does not imply any kind of restriction over the life of the parts with regard to its whole.



SQL:1999

```
CREATE TYPE Project AS
( Project_Id INTEGER,
  Name VARCHAR(30),
  Has_Plain REF(Plain) ARRAY[1000] );

CREATE TABLE T_Project OF Project
(PRIMARY KEY (Project_Id));
```

Oracle8*i*

```
CREATE TYPE Plain AS OBJECT
Plain_ID NUMBER,
End_Date DATE,
Figure_Numbers NUMBER,
Architect VARCHAR (20);

CREATE OR REPLACE TYPE List_Plain
AS TABLE OF REF Plain;

CREATE TYPE Project AS OBJECT
( Project_Id NUMBER,
  Name VARCHAR(30),
  Has_Plain List_Plain);

CREATE TABLE T_Project OF Project
(PRIMARY KEY (Project_ID))
NESTED TABLE Has_Plain STORE AS
Table_Plain;

CREATE TABLE T_Plain OF Plain
(PRIMARY KEY (Plain_Id));
```

Fig. 7. Simple Aggregation Transformation

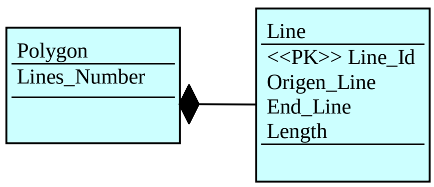
5.2 Composition

The composition, as we have explained in section 3.2, is a special kind of aggregation in which the *parts* are physically linked to the *whole*. So, a composition defines three restrictions with regard the aggregation concept:

Restriction 1: A *part* can not simultaneously belong to more than one *whole*.
Restriction 2: Once a *part* has been created it lives and dies with its whole.
Restriction 3: A *part* can be explicitly removed before to remove its associated *whole*.

Translating the UML concept of composition into an object-relational schema depends on the target model. Considering the translation to the SQL:1999, as the standard object-relational model, there is not any difference with the aggregation implementation. To represent an aggregation or a composition in SQL:1999 we have to introduce an attribute in the specification of the *whole*. As SQL:1999 provides only the ARRAY collection type, this attribute has to be an ARRAY in both cases. The restrictions mentioned above have to be implemented by means of checks, assertions and/or triggers. This is just like some object-relational products, such as Informix that provides the set, list and multiset collection types.

However, in Oracle8*i* it is possible to implement directly the concept of composition maintaining the differences with regard to the aggregation concept. This is because Oracle8*i* besides supporting the VARRAY collection type, it also provides the nested table. A nested table is a collection type but it is also a table. Being a table, it 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*). Figure 8 shows the specification of a composition in Oracle8*i*.



Oracle8*i*

```
CREATE TYPE Line AS OBJECT
(Line_Id NUMBER)

CREATE OR REPLACE TYPE Lista_Lines
AS TABLE OF Line

CREATE TYPE Polygon AS OBJECT
(Polygon_Id NUMBER,
Lines_Number NUMBER,
Tiene_Lineas Lista_Lineas)

CREATE TABLE T_Polygon OF Polygon
(PRIMARY KEY (Polygon_Id))
NESTED TABLE Has_Lines STORE AS
Table_Lines;
```

Fig.8. Composition Transformation

When the composition is represented in the way defined above, the three composition restrictions defined previously are fulfilled. Therefore any check, assertion or trigger has to be defined to implement the composition semantics. So, the nested table allows implementing the composition and the simple aggregation maintaining their semantics differences, in the same way as in UML.

6 Conclusions and Future Work

In this paper we have summarised a methodology for object-relational database design focused on the aggregation and composition implementation. The methodology proposes three phases: analysis, design and implementation. As conceptual modelling technique we have chosen the UML class diagram. As logical model we have used the SQL:1999 object-relational model, so that the guidelines are not dependent of the different implementations of each object-relational product. As a product example we have used Oracle8*i*.

We have briefly explained the rules to transform a UML aggregation into an object-relational model considering the differences between aggregation and composition. We have focused on the implementation in Oracle8*i* because this product supports a collection data type, the nested table that is specially appropriated to implement aggregations and compositions. This collection data type is not provided neither by SQL:1999 nor by other products.

In the methodology we have proposed two alternative techniques for the standard design phase: defining the schema in SQL:1999, because it does not depend on a specific product; and/or using a graphical notation describing a standard object-relational model (the SQL:1999 model). This graphical notation corresponds with the relational "graph" that represents the logical design of a relational database. As graphical notation, and due to UML can be extended, we propose to use UML extended with the required stereotypes for the SQL:1999 object-relational model. Although there are some proposals of UML stereotypes for database design, [1,18], they are focus on the relational model. So, we are working now extending UML with stereotypes for object-relational design.

The next step will be completing the methodology taking into account the UML use cases diagrams to design the behaviour of the classes.

Acknowledges

This work is being carried out as part of the MIDAS project. MIDAS is partially financed by the Spanish Government and the European Community (reference number: 2FD97-2163). We want also thank the anonymous reviewers for their useful comments.

References

1. Ambler (1999), *Persistence Modelling in the UML*. In: http://www.sdmagazine.com/articles/1999/0008/0008q/0008q.htm.
2. Atzeni, Ceri, Paraboschi and Torlone (1999). *Database Systems. Concepts, Languages and Architectures*. McGraw-Hill.
3. Bertino and Marcos (2000), "Object Oriented Database Systems". In *Advanced Databases: Technology and Design,* O. Díaz and M. Piattini (Eds.). Artech House.
4. Bertino and Martino (1993), *Object-Oriented Database Systems. Concepts and Architectures*. Addison-Wesley.
5. Blaha and Premerlani (1998), *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall.
6. Booch and Rumbaugh (1995), "Unified Method for Object-Oriented Development". Grady Booch y James Rumbaugh. Documentation Set V0.8. Rational Software Corporation, 1995.
7. Booch, Rumbaugh and Jacobson(1999), *The Unified Modelling Language*. Addison Wesley.
8. Cattell and Barry (2000), *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann.
9. Eisenberg and Melton (1999), "SQL:1999, formerly known as SQL3". *ACM SIGMOD Record*, Vol. 28, No. 1, pp. 131-138, March 1999.
10. Informix Corporation (1999), *Informix Guide to SQL: Reference*. Electronic Documentation, Informix Press.

11. Henserson - Sellers (1997), "Open Relationships-Compositions and Containments". Journal of Object - Oriented Programing. Sección ROAD (Report on Object Analysis & Design). SIGS Publications, Nov.-Dec. 1997, Vol 10, No. 7, pp. 51-55, 72.

12. Kim et al. (1989), "Composite Object Revisted". Won Kim. Proc. of the SIGMOD Int. Conf. on the Management of Data, 1989, pp. 337-347.

13. C. Kovács and P. Van Bommel (1998), "Conceptual modelling-based design of object - oriented databases". *Information and Software Technology*, Vol. 40, No. 1, pp. 1-14.

14. Leavit, N. (2000), "Whatever Happened to Object-Oriented Databases?". *Computer*, pp. 16-19, August 2000.

15. Mattos, N. M (1999), *SQL:1999, SQL/MM and SQLJ: An Overview of the SQL Standards*. Tutorial, IBM Database Common Technology.

16. Marcos, E. and Cáceres, P. (2001), "Object Oriented Database Design". In: *Developing Quality Complex Database Systems: Practices, Techniques, and Technologies*. Ed. Shirley Becker. Idea Group (accepted to publish in 2001).

17. Muller (1999), *Database Design for Smarties*. Morgan Kaufmann.

18. Naiburg, E. (2000), "Database Modeling and Design Using Rational Rose 2000e". *Rose Architect,* Vol. 2, Issue 3, pp. 48-51.

19. Oracle Corporation (1998), "Objects and SQL in Oracle8". Oracle Technical White paper. In: *Extended DataBase Technology conference (EDBT'98)*. Valencia (Madrid).

20. Oracle Corporation (2000), Oracle8*i*. *SQL Reference. Release 3 (8.1.7)*. In: www.oracle.com.

21. Silva and Carlson (1995), "MOODD, a method for object-oriented database design." *Data & Knowledge Engineering*, Vol. 17, pp.159-181.

22. Pastor y Ramos (1995), "OASIS 2.1.1: A Class - Definition Language to Model Information Systems Using an Object-Oriented Aproach". *Dpto. de Sistemas Informáticos y Computación, SPUPV- 95.788. Universidad Politécnica de Valencia*, March 1995.

23. Stonebraker and Brown (1999*). Object-Relational DBMSs. Traking the Next Great Wave*. Morgan Kauffman.

24. Ullman and Widom (1997), *A First Course in Database Systems*. Prentice-Hall.

25. Winston et. al (1987), "A taxonomy of part-whole relations". *Cognitive Science*, Vol. 11, 1987, pp. 417-444.