Object Wrapper: an Object-Oriented Interface for Relational Databases

Sonia Bergamaschi
Dip. di Scienze dell'Ingegneria
Universitá di Modena
via Campi 213/B, 41100 Modena, Italy
sonia@dsi.unimo.it

Alessandra Garuti
Dip. di Scienze dell'Ingegneria
Universitá di Modena
via Campi 213/B, 41100 Modena, Italy
garuti@dsi.unimo.it

Claudio Sartori
Dip. di Elettronica, Informatica e Sistemistica
Universitá di Bologna
viale Risorgimento 2, 40136 Bologna, Italy
csartori@deis.unibo.it

Alberto Venuta
Dip. di Scienze dell'Ingegneria
Universitá di Modena
via Campi 213/B, 41100 Modena, Italy
venuta@dsi.unimo.it

Abstract

Most commercial applications have to cope with a large number of stored object instances and have data shared among many users and applications. For object-oriented as well as conventional application development RDBMS technology is currently being used in most case.

We describe a software module called Object Wrapper for storing and retrieving objects in a RDBMS. Having these capabilities in a separate component helps to isolate data management system dependencies and hence contributes to portable applications.

1 Introduction

Object-oriented technology is one of the major trends in software development. It provides a pratical, productive way to develop software applications.

Most object-oriented applications have to cope with a large number of object instances. A need that arises is the ability to store objects longer the execution that created it, that ability is called persistence. Persistence means that objects are copied from a fast and volatile primary memory to a slow and persistent secondary memory.

Currently the major persistent data management approaches to consider are relational database management systems (RDBMS), object-oriented database management systems (ODBMS) and file systems. While file systems are cheap, they do not offer the many infrastructure features provided by RDBMS, like transactions management, security, integrity, sharing between users and applications, and standard query language (SQL). On the other hand, ODBMS are just emerging and will become important in the future, but are considered less mature for supporting commercial applications today. For these reasons RDBMS technology is currently being used in most case.

In the object oriented applications information is organized as a collection of discrete objects that incorporate both data structure and behaviour. Each object has its own inherent identity. Objects with the same data structure (attributes) and behaviour (operations) are grouped into a class. Each class describes a possibly infinite set of individual objects. Each object is said to be an instance of its class. A class can be defined broadly and then refined into successively finer subclass. Each subclass inherits all of the properties of its superclass and adds its own unique properties.

In a relational database, information is stored in tables. This yields some problems for the ability of storing objects. Firstly, only data can be stored and not behaviour, secondly, only primitive data types can be stored and not complex structures of objects. A third problem is how to express inheritance in the database. Other problems that arise include locking of the database during longer transactions, integrity, and distribution.

Furthermore we deal with two states of a persistent object: one in volatile memory, and one stored in the database.This yields some problems for retrieving data from the database and creating objects: Which objects must be instantiated? How an object is istantiated? How data integrity is preserved?

A number of approaches have been undertaken by vendors, clients and researches to adapts relational database systems for objects [1]. These include:

- adding object extensions to the relational model;
- adding object layers on top of relational databases;
- adding object gateways to relational databases;
- building object middlewares for access to relational databases.

In this paper we describe a software module called Object Wrapper for storing and retrieving objects in a RDBMS. Having these capabilities in a separate component helps to isolate data management system dependencies and hence contributes to portable applications.

The outline of the paper is the following. In Section 2 a method for mapping classes into tables is presented. Section 3 introduces the Object Wrapper functionalities and a configuration database to implement the mapping of an object-oriented into a relational schema. Finally in Section 4 the Object Wrapper implementation is presented.

2 Mapping Object Classes into Tables

We suppose that an application have been designed by exploiting an object-oriented methodology such as OMT [3], thus producing an object data model. Furthermore, we assume that data must be supported persistently by a RDBMS. The first task is to design a suitable relational structure. This involves mapping object classes, attributes, relationship and inheritance hierarchies into relations (tables).

The problem we face in this task is referred to as the *impedance* problem: object oriented programming languages allow object attributes to be of complex types (e.g. class types) while most data management systems usually only allow primitive data types.

A possible easy mapping should be the following, as proposed in [2] and [5]. Each class will be represented by a table (at least one) in the database. A class is mapped onto tables in the following manner:

1. Assign one table for the class.
2. Each primitive attribute become one column in the table. Any complex attribute must be translated into primitive RDBMS data types and, depending on the situation, either maps to a group of columns within the relation, or to a separate relation.
3. The primary key column will be the unique instance identifier, namely the identifier by which the instance is uniquely recognized.
4. An attribute which holds a pointer to an object of a related object class can be mapped by including a foreign key (the object ID of the linked instance) in the relation.
5. Each association with a cardinality greater than 1 will become a new table. This new table will connect the tables representing the objects that are to be associated.

The impedance problem yields yet another problem: it creates a strong coupling between the application and the RDBMS. To make the design minimally affected by the RDBMS, as few parts of our system as possible, have know about the RDBMS' interface.

Another problem is how to express inheritance in the database. There are mainly two alternatives for mapping inheritance hierarchies onto the relational model:

- the generalization class is in one table of its own, to which the tables of the descendant classes refers;
- the inherited attributes are copied to all the tables that represent the descendant classes. No table will represent the generalization class.

The first alternative yields the cleanest design which also contribute to extensibility. Nevertheless, the second way can be considered when a more efficient implementation is required.

3 The Object Wrapper

In the following we describe a software module called Object Wrapper which is a full object-oriented interface for storing and retrieving objects in a RDBMS. We assume as programming language an object-oriented language, such as C++, which provides RDBMS interfacing facilities (e.g. Microsoft Foundation Classes).

The purpose of the Object Wrapper is to make available to the object-oriented application programmers an object-oriented interface to the underlying relational database. The main achievements in using the Object Wrapper are:

1. The programmer does not need to know the underlying relational database structure.
2. It minimizes the coupling between the application and the RDBMS: only few parts of our system must know about the RDBMS' interface.

We establish the following premises:

- we deal with two states of a persistent object: one in volatile memory, the volatile state, and one stored in the database, the database state. The volatile state of a persistent object is defined in its class, while its database persistent state is described by the corresponding relational tables. The correspondence between both state is established through the primary key value;

- volatile state can survive database transaction limits. However, at the end of a transaction (commit or abort) we do not allow a persistent object to exist in volatile memory without having a stored state in the database. When a transaction commits, volatile and database states are consistent and a new transaction begins;

- we deal only with queries which return objects. Retrieved objects are instantiated. If the resulting set of retrieved objects is large, all objects are instantiated (database lists);

- database queries take only into account stored database states, not uncommitted changes to the volatile state;

- we do not assume any particular concurrency model (i.e. we rely on the underlying RDBMS concurrency control);

- when a persistent object which references other persistent objects is retrieved from the database all referenced objects need to be retrieved as well in order to avoid dangling pointers. This, of course, presents a problem when the number of objects to be retrieved becomes too large. Our solution is that the retrieved objects are only a first level view of the stored objects. They contain attribute values but not pointers to the associated objects in the aggregation hierarchy. A further operation must be used to retrieve the required objects, by traversing the aggregation hierachy.

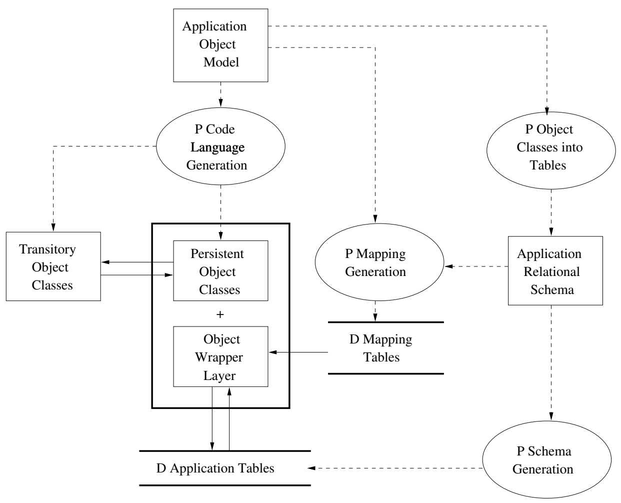

Figure 1. Functional architecture

Figure 1 describes the functional architecture of the Object Wrapper.

1. As a starting point we have the Application Object Model, that is an object-oriented description of the application domain including class structure, methods, relationships between classes. Classes must be differentiated into persistent and transitory. A class is classified persistent when it has a copy stored in a persistent memory, otherwise we classify it as transitory.

2. By the application of the mapping described in section 2, the subset of persistent classes is mapped into relational tables (P Object Classes into Tables).

3. Tables are implemented using a RDBMS into an internal schema (P Schema Generation). The internal schema consists of SQL statements that create the tables, attributes and performance tuning structures (e.g. Index).

4. The Application Object Model is also used to generate the code (P Code Language Generation) that implements the classes (both persistent and transitory). Each object class can be associated with each other. The processes described in 2., 3. and 4. are usually available in CASE tools, as System Architect [4].

5. The Object Wrapper Layer stores and retrieves data in the RDBMS (D Application Tables). In order to perform these operations the Object Wrapper Layer must know how the objects are mapped into tables (D Mapping Tables). The Configuration Database is the process (P Mapping Generation), described in the following section, performing the mapping of Object Classes into Tables.

3.1 Configuration database

Figure 2 shows the OMT Object Model of the metadatabase that maps an object-oriented schema into a relational schema. Each class corresponds to a table in the relational schema and each class attribute that maps on a simple domain corresponds to an attribute in the relational schema. The class attributes that maps on a complex domain do not have a relational correspondent; they are split up into three categories and they are retrieved and setted up through foreign keys values which are defined in the private definition of the class.

Description of the Configuration database classes

Attribute: represents an attribute of a persistent class. *class_attribute_name*: name of the attribute in the class definition

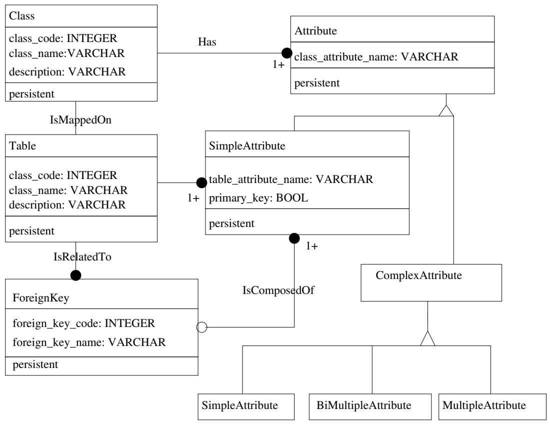

Figure 2. Configuration Database

BiMultipleAttribute: represents an attribute of a persistent class that maps a many-to-many association.

Class: represents a persistent class.
class_code : identification code of the class
class_name : name of the class
description : description of the class

Complex Attribute: representes an attribute of a persistent class that maps an association.

ForeignKey: represents a foreign key of the relational schema.
foreign_key_code : identification code of the foreign key
foreign_key_name : name of the foreign key

MultipleAttribute: represents an attribute of a persistent class that maps a one-to-many association.

SimpleAttribute: represents an attribute of a persistent class that maps on a simple domain. It has a correspondent attribute in the relational schema
table_attribute_name: name of the attribute in the relational schema
primary_key: it indicates if the attribute is part of a primary key in the relational schema

SingleAttribute: represents an attribute of a persistent class that maps a many-to-one association.

Table: represents a persistent class. *table_code*: identification code of the table
table_name: name of the table
description: description of the table

3.2 Object Wrapper Design

Figure 3 describes in a OMT diagram the basic design of the Object Wrapper.

The class **ApplicationData** represents the most general persistent class with its member functions which performs to update, delete or insert an object.

In order to include the select capability in the Object Wrapper, a new class (**Select**) is added. The Select class maps on a set of **ApplicationData** in order to design the multiplicity of the query result and on a set of **BooleanFactor** that defines each boolean factor of a search condition.

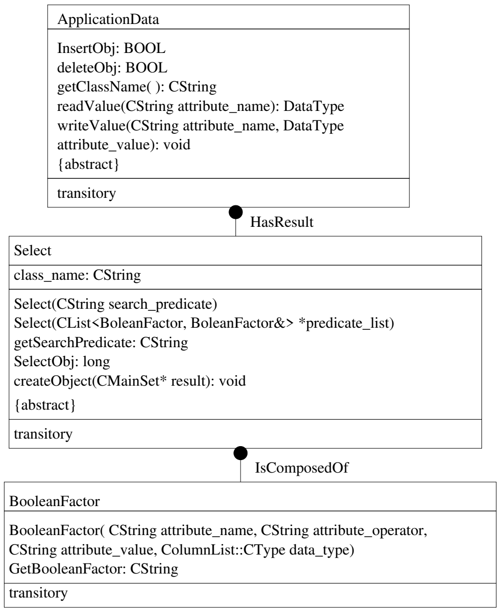

Figure 3. Object Wrapper design

Description of the Object Wrapper classes

BooleanFactor: Boolean factor which is part of a search predicate (where-clause).

BooleanFactor(CString attribute_name, CString attribute_operator, CString attribute_value, ColumnList::CType data_type): creates a BooleanFactor
getBooleanFactor():CString: returns the Boolean factor

ApplicationData: represents all the persistent data that are contained in the application.

insertObj():BOOL: performs an insert transaction of an object and returns TRUE if the transaction succeed, FALSE

otherwise
deleteObj():BOOL: performs a delete transaction of an object and returns TRUE if the transaction succeed, FALSE otherwise
updateObj():BOOL: performs an update transaction of an object and returns TRUE if the transaction succeed, FALSE otherwise
getClassName():CString: pure virtual member function that must be overridden. It returns the class name
readValue(CString attribute_name):DataType: pure virtual member function that must be overridden. It returns the value of the attribute 'attribute_name'
WriteValue(CString attribute_name, DataType attribute_value):void: pure virtual member function that must be overridden. It returns the class name of the object

Select: select transaction which requires a parameter specification (where-clause).

class_name:CString: name of the class on which the query is executed
Select(CString search_predicate) : creates a Select by specifying the where-clause of the query. The user must know the relational structure of the database
*Select(CList <BooleanFactor, BooleanFactor&> *predicate_list* : creates a Select by specifying a list of Boolean factors. The user doesn't know the relational structure but only the object-oriented interface
getSearchPredicate():CString: returns the search_predicate
selectObj():long: performs a parameter select transaction and returns the number of objects selected
createObject(CMainSet result):void*: pure virtual function that must be overridden. It creates an object for each record selected

Associations
Has Result: specifies the result objects for select transactions.
IsComposedOf: specifies the Boolean predicates which the search predicate is composed of.

4 Implementation of the Object Wrapper

Insert, update and delete capabilities

In order to obtain these capabilities, a persistent class must have as base class the abstract class *ApplicationData* and it must contain all the attributes of the relational table (foreign keys and attribute which maps on a complex domain are always private). The programmer must redefine three pure virtual functions (words in **bold** style are variables and they must be replaced by correct names):

A. getClassName(): CString - this member function returns the class name.

Template code
```
CString class_name::getClassName()
{
    return "class_name";
}
```

B. readValue(CString attribute_name): DataType - this member function reads the value of a class attribute. At present the object wrapper can manage ten **data_type** (class *DataType*): d_BOOL, d_BYTE, d_int, d_LONG, d_float, d_double, d_CTime, d_ByteArray, d_CString, d_CLongBinary. We assume that the class **class_name** has N attributes which map on a simple domain.

Template code
```
DataType class_name::readValue(CString att_name)
{
    DataType data = DataType();
    if(!att_name.Compare("attribute_name1"))
            *data.data_type1 = attribute_name1;
    else
            if(!att_name.Compare("attribute_name2"))
               *data.data_type2 = attribute_name2;
            else
               if ...
               else
                    if(!att_name.Compare("attribute_nameN"))
                         *data.data_typeN = attribute_nameN;
    return data;
}
```

C. writeValue(CString attribute_name, DataType attribute_value): DataType - this member function sets the value of a class attribute.

Template code
```
class_name:: writeValue( CString att_name, DataType att_value)
{
if(!att_name.Compare("attribute_name1"))
    attribute_name1 = *att_value.attribute_value1;
else
    if(!att_name.Compare("attribute_name2"))
       attribute_name2 = *att_valueattribute_value2;
    else
       if ...
       else
            if(!att_name.Compare("attribute_nameN"))
               attribute_nameN = *att_valueattribute_valueN;
}
```

Select capability

In order to obtain this capability for a predefined persistent class, the programmer builds a new class which must have as base class the abstract class **Select**. The name of the new class is the concatenation of the string "Select" and the name of the class that is managed for the retrieve operations (e.g. for the class Installation, the class that manages select operations has name "Select" + "Installation" = "SelectInstallation"). The programmer adds to the new Select**class_name** class the attribute query_result. This attribute represents a list of **class_name** objects (Clist<**class_name, class_name&**>) and it contains the result of the query. The programmer redefines one pure virtual function (words in **bold** style are variable and they must be replaced by correct names):

A. createObject(CMainSet* result): void - this function creates an object of the specified class and adds it to the query result list.

Template code
*void Select**class_name**::createObject(CMainSet* result)*
{
  class_name** one = **class_name();
  //default costruction of the object "one"
  one.fillInObject(result);
  query_result.AddTail(one);
}

The Object Wrapper requires some programming efforts when an object is to be retrieved together with its components. For example, the operation "retrieve all the geographic areas inside a given rectangle and their associated maps" (see figure 4) requires fist the retrieval of the geographic areas and then, for each area, *getMap()*. Furthermore, when a selection predicate refers to the characteristics of an associated object, it is necessary to program the equivalent of a nested loop algorithm. For example, the operation "retrieve all the geographic areas containing installations of building year xxx" requires for each geographic area the invocation of the method *getInstallation()*, then, if any of the retrieved installations has the appropriate building area, the geographic area is included in the result.

4.1 Conclusions and future work

We presented a software module called Object Wrapper which is a full object-oriented interface for storing and retrieving objects in a RDBMS.

The implementation of the current version is quite rudimentary, requiring some programming efforts in order to introduce in the classes description both the redefined versions of the pure virtual functions and the standard class Select**class_name**.

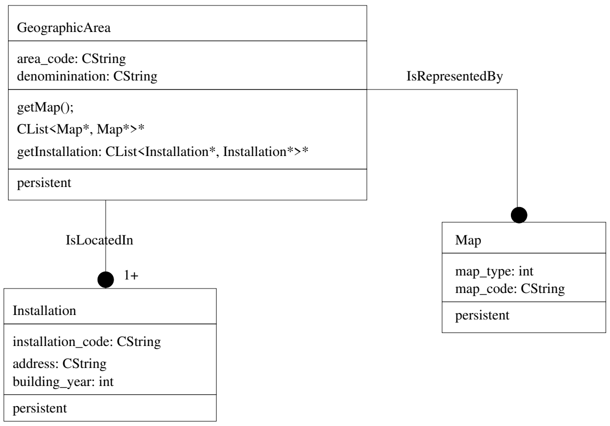

Figure 4. Example

An automatic generation of the above code modifications will be developed as it constitutes a relevant improvement of the usability of the Object Wrapper.

Furthermore, a tool to support definition of the mapping between the object classes and the underlying relational tables will be implemented.

References

[1] Chung et al. Object and relational databases. *OOPSLA '95 Addendum to the Proceedings*, pages 164–169, October 1995.
[2] Jacobson et al. *Object-Oriented Software Enginering: A Use Case Driven Approach*. Addison-Wesley, ACM Press, 1992.
[3] Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall International Editions, 1991.
[4] P. Software and S. incorporated. System architect, 1995.
[5] Yourdon et al. *Mainstream Objects: An Analysis and Design Approach for Business*. Yourdon Press, Prentice Hall Building, 1995.