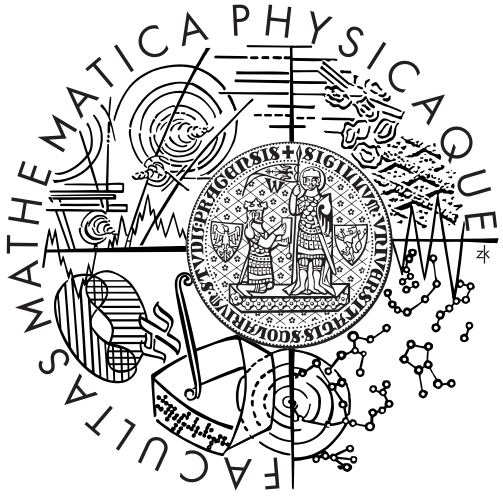
Charles University in Prague

Faculty of Mathematics and Physics



Doctoral Thesis

Mgr. Michal Kopecký

Object Persistency in C++

Department of Software Engineering
Supervisor: Prof. Jaroslav Pokorný, CSc.
Study program: Informatics

Prague, 2002

Acknowledgement

I deal a great deal to Prof. Jaroslav Pokorný for supervising my postgraduate studies at the Charles University, for his support and recommendations on this thesis. I thank also V. Bisová, M. Prokeš, R. Řípa, V. Tloušť and other members of ADOORE research team as well as to Filip Lázníček for their active participation on design, implementation and testing of GEN.LIB library. Finally, my thanks go to all other people who supported me during my writing this thesis.

Declaration

I declare that I wrote this thesis on my own and that the references include all the sources of information I have utilised. This thesis is freely available for study purposes

Prague, June 2002

Abstract

The methods of the persistent data storage and manipulation have rapidly changed during last years. Among older relational databases appeared implementations of object-oriented databases like O2. For accessing data in object database management systems, there exists well-defined ODMG interface, since January 2000 in version ODMG-3. Before ODMG, the lack of a standard for object databases was one of major limitations to their more widespread use. In spite of all theoretical advancements OO databases don't fulfil all expectations yet. Currently the evolution in data processing appears more likely and more acceptable by users than making revolution steps. Progressive features of object-oriented databases become available in recent versions of relational database management systems of major vendors as Oracle, Informix etc.

The wide acceptance of the SQL-92 Entry Level standard in the world of relational databases allows a high degree of portability and interoperability between relational systems. Successor of SQL-92, standard SQL-1999 accepted in December 1999 as part of SQL-3 standard proposal consists of ten parts define language elements for defining and using procedural objects and user defined types, methods for data binding and call level interfaces. However, ten years after, not all features defined even in the SQL-92 Full level are implemented in available databases. According to Jim Melton, member of technical stuff in Oracle Server Technologies, it will take two to three generations of database servers to conform SQL-1999. Considering delay in acceptance of SQL-92 it will take even more time.

This thesis is concerned on the aspects of persistent data storage of C++ objects in the relational databases. It introduces design of transparent interface and implementation of the library optimised for currently available SQL-92 Entry Level compliant databases, thus using classical database relations as the storage medium. Proposed query language – object algebra – combines operations of relational algebra with used object model. Although the main goal of the thesis storing of and manipulation with C++ objects with respect to the portability, designed object library allows uniform access also to legacy relational data through C++ objects.

Table of Contents

Figures

Definitions

Abbreviations used

ADT ..... Abstract Data Type
API ..... Application Program Interface
OCI ..... Oracle Call Interface
ODL ..... Object Definition Language
OO ..... Object-Oriented
OQL ..... Object Query Language
OODBMS ..... Object-Oriented Database Management System
ORDBMS ..... Object-Relational Database Management System
RDBMS ..... Relational Database Management System
SQL ..... Structured Query Language

1 Introduction

The world of databases went through the big development during last few years. Aside relational database management systems (RDBMS) appeared also object-oriented database management systems (OODBMS).

Object databases are usually referring to those data management products that are specifically designed for use with an object programming language and are very closely coupled with one or more object programming languages. Usually, the most of the recent OODBMSs use C++ and/or Smalltalk languages as their database programming language. The application programmer may access database objects directly using the database operations in the programming language, or may perform associative lookups of objects using the query language. Unfortunately, the latter includes many problems concerning both the designs of such languages and their implementations. Two significant development lines are recently at disposal: ODMG-93 standard with its query language OQL [Ca93] and the work proposed by the ANSI X3H2 group well-known as SQL3 [Me94]. Both ANSI X3H2 and ISO DBL committees accepted part of this proposal in 1999. Both ODMG and SQL-99 standards are briefly described in section 2.

A few years ago, the OODBMS seemed to win the competition with standard relational databases. There seemed to be two advantages of the OODBMS systems.

- OODBMSs could better handle complex data structures and storage binary content like images, video, and audio data.
- Object-oriented databases have the ability to model static properties such as objects, attributes and relationships, integrity rules over objects and operations, and dynamic properties of objects directly.

Application development community absorbed OO tools, both the object-oriented programming languages and specialised client-server tools, including OO CASE tools and associated OO methodologies for software design and specification.

In comparison with expectances, today's reality is much different. Relational technology is still dominant and thus mixing the worlds of relations and objects has appeared. The idea to implant the objects to relational database is not new. For example in [Pr91] authors describe a technique for constructing an OODBMS from existing relational technology. They denote their resulted architecture as the object-oriented relational database. During 90ties many different approaches to the problem and associated products have appeared. Paul Harmon in [Ha95] distinguishes among object/relational data managers, relational wrapper libraries and object/relational databases.

Object/relational data managers map objects directly to relational tables and manage objects. Products such as HP Odapter, Persistence, Ontos OIS, UniSQL belongs to them. For example, Odapter can scan SQL table definitions of existing databases and figure out the corresponding C++ to make object definitions. Today, it is available as an object management level on top of Oracle.

Relational wrapper libraries map objects to database objects, which are linked to relational database. The relational wrapper detects a change in the contents of an object and automatically generates the SQL to make the changes in the linked relational database. Similarly, it detects changes to relational database and moves that information back into the local objects. Obviously, the translations are transparent to the user. Good examples of this approach are Smalltalk tools, e.g. VisualWorks and VisualAge.

Object/relational databases store information using both objects and relations. They also let developers to access the data using either method. Today most important examples of this approach include products of traditional relational database vendors such as Oracle since version 8 [Or96],

Informix, and IBM [Ch96]. In spite of similarity between them the backward compatibility with older products and complexity of the SQL standard causes differences in approaches to their design.

As Dr. Mary Loomis, the architect of the Versant OODBMS, mentioned in her interview in DBMS Journal [Ka94], the typical C++ programmer is not familiar with the SQL language, and is not familiar with relational systems. He or she would like to abstract from using the database and consider the database as a very large extended virtual memory of objects. The manipulation with objects should be the same independently on the fact whether the objects are in main memory, on disk, or stored in the database remotely across the network. Programmers do not want to have to think whether an object is in the cache or it is necessary to get it from disk. Similarly, they do not want to learn new languages as OQL or SQL, for example.

This thesis concerns on solution of interoperability between the program written in OO programming language, especially in C++ and ordinary relational database. Second chapter of this thesis briefly describes relevant database standards together with examples of competitive approaches to program object persistence. Next two chapters form the main part of the document. First of them, chapter 3, describes the formal object model and the object algebra – proposed query language for retrieving object sets upon this object model. Fourth chapter thoroughly describes C++ interface of described of theoretical framework and practical aspects of its implementation. Included C++ library not only allows programmer to manipulate uniformly with both persistent objects and older relational tables, but also enhances C++ language by simple manipulation with directly or indirectly associated objects. First generation of the proposed interface was successfully used in the GEN.LIB[1], part of ADOORE[2] project. Its primary purpose was to implant the Rumbaugh's OMT methodology [Ru91] of OO analysis and design into the environment of building business applications. Aspects of the library described in [Ko96a] and [Ko96b] were published among others in [Ko97]. The experience from this project leads to significant enhancements of both the object model and query language.

[1] GENeral LIBrary

[2] Application specific Depository of Object ORiented Environment

2 Related Works

The idea of persistent storage of objects manipulated in the application is not new. Concerning the persistency achieved using database management system exist three main trends that differs in the complexity of data that can be maintained directly by the database server. Together with different approach to data representation those three kinds of databases use different standards for the data access and manipulation. Most advanced object databases are accessible through ODMG interface. In contrast the older relational databases use SQL dialects based on the SQL-92 standard. The newest class of databases we should take into account discussing the object persistency is so called object-relational databases. They stand in middle between relational databases from which they evolve and object databases. As in case of pure object and pure relational databases there exist SQL-1999 standard for communication with them. The existence of widely accepted standard for communication is important for portability. As we show below, not all standards on the paper are also standards currently accepted and supported by relevant database vendors on the market.

2.1 ODMG

The object database standard ODMG represents industry standard that for object databases provides or should provide the same work as the SQL does in world of relational databases. It promises cross-platform portability of applications written above object-relational database. ODMG draws from SQL, OMG and IDL standards. Upon them it builds object definition language ODL and object query language OQL.

2.1.1 ODMG Object Definition Language

The ODL proposed by ODMG standard is analogous to DDL – data definition language – for relational databases. It is extension of OMG language for interface definition. It abstracts from implementation details of OODBMS and so the source code can be ported to any compatible database.

New type in ODL is declared by definition of its interface. The simplified syntax of interface declaration is

```
<type_definition>       ::=  interface <type_name> [ : <parent_type_list> ]
                             {
                             [ <type_property_list> ]
                             [ <property_list> ]
                             [ <operation_list> ]
                             }
```

where

```
<type_name>           ::=  <string>

<parent_type_list>    ::=  <type_name>
                           | <type_name> , <parent_type_list>

<type_property_list>  ::=  <type_property>
                           | <type_property> , <type_property_list>

<type_property>       ::=  extent <string>
                           | key <property_name>
                           | keys <property_list>

<property_list>       ::=  <property_name>
                           | <property_name> , <property_list>

<property_name>       ::=  <attribute_name>
                           | <traversal_path>
```

```
<attribute_name>      ::=  [ attribute ] <string>
<traversal_path>      ::=  <string>
```

Complete grammar of ODL can be found in [Ca93] or in its last version in [Ca00].

The extent clause defines name of the extent – the set of all instances of given type. Each instance of class *Person* will be an element of the extent of class *Person*. If for example class *Employee* is subclass of class *Person*, then the extent of *Employee* is subset of extent of *Person*.

In case the instances of the class are uniquely identifiable by values they contain in some attribute or set of attributes, those identifying attributes can be declared as keys.

For example the class Person with its subclass Employee can be defined as follows.

```
interface Person
{
      extent People;
      keys FirstName, LastName;
      attribute string[30] FirstName,
      attribute string[30] LastName,
      unsigned short age()
};


interface Employee : Person
{
      extent Employees;
      key Id;
      attribute int Id,
      attribute float Salary
};
```

ODMG standard supports binary relationships of classes with 1:1, 1:N and M:N cardinality. Instead of self-standing relationship definitions the ODMG standard declares so called traversal paths inside related classes.

```
interface Employee
{
      ...
      relationship Employee Supervisor inverse Employee::Supervises;
      relationship Set<Employee> Supervises inverse Employee::Supervisor;
      ...
};
```

Return types of two inverse traversal paths express the cardinality of the relationship. This example shows definition of one supervisor-supervises relationship with 1:n cardinality.

2.1.2 ODMG Object Query Language

The OQL is a declarative language similar to SQL–92. Its extensions concern complex objects, polymorphism and traversal paths. Co-operation of ODMG and ANSI X3H2 groups ensures interoperability of OQL also with newer SQL-99 standard. The query statement

```
SELECT *
FROM Emploees AS e
WHERE e.Salary > 500
```

returns all employees with salary greater than $500.

2.2 SQL-99

The multi-part standard ISO/IEC 9075-n:1999, known as SQL-99, represents third generation of the SQL standard. This standard reflects join efforts between database vendors as IBM, Informix

(currently belonging to IBM), Microsoft, Oracle, Sybase, and others. It tries to unify the methods of defining, accessing and manipulating data in object/relational databases. Its goal is to enable portability of database applications across different servers. It represents significant enhancement over SQL-92, previous ANSI/ISO database language standard. SQL-99 extends SQL-92 in many ways. From our point of view the most interesting is the addition of an extensible, object-oriented type system defined in SQL/Foundation, part 2 of the standard. Apart from the ability to create flat tables, SQL-92 offers no facilities to define complex data types. In contrast, SQL-99 offers a fairly rich type system based on the notion of abstract data types (ADTs). ADT definitions correspond to a set of attribute and routine (procedure/function) definitions, as it is shown on following example.

```
CREATE TYPE Person_T
(
    first_name CHAR(30),
    last_name CHAR(30),
    birth_date DATE,
    FUNCTION age RETURNS INTEGER
    <the source code of the function age>
);

CREATE TYPE Employee_T UNDER Person_T
(
    id NUMBER(10),
    salary NUMBER(7,2)
);
```

An ADT is completely encapsulated. Only its behaviour is visible outside the type definition, but not the implementation of its attributes and routines. ADT routines can either be implemented using SQL-99 procedural extensions or using code written in external languages as Ada, C, COBOL, and many others including Java. ADTs can be related in subtype-supertype relationships, where one ADT in SQL-99 can be a subtype of at most one super-type, using syntax

```
CREATE TYPE TeamLeader_T UNDER Employee_T
```

Instances of sub-types inherit attributes as well as the behaviour of its super-type and can be substituted wherever instances of some its super-type are expected. Resolution of overloaded routines is based on types of all arguments in a routine invocation. Dynamic binding is also supported whenever compile-time binding is not possible. However, type checking is always performed at compile time.

ADTs can be used as data types of variables, parameters of routines, attributes of ADTs, or columns of tables. Persistent instances of ADTs can be stored in columns of tables and can be queried using the familiar SQL constructs. Queries can refer to attributes and functions of ADT instances.

Although the SQL-99 standard weights much over one thousand of pages, work on the standard is not fully finished and many features of SQL3 not contained in SQL-99 is still subject of changes. Due to complexity of the standard there exists no complete implementation of it yet. Moreover, there are currently no RDBMS supporting the highest (full) level of previous standard SQL-92. The current commercial DBMS are still usually compliant only with its lowest (entry) level. Of course, each of them provides many additional features, which are unfortunately not compatible with solutions from other vendors. The differences go from using non-standard data types to absolutely different languages for coding of stored procedures and triggers.

2.2.1 Object-oriented features in Oracle server

The Oracle Company becomes to the biggest database server vendor currently on the market. In reaction to appearance of OODBMSs it implants some object-oriented features to its relational database server. Since the version 8 the server supports storage of nested relations, definition of variable arrays and – from the scope of this thesis most interesting – support of user defined types and objects.

In spite of the fact, that the Oracle company is one of contributors of above discussed SQL-99 standard, its Oracle implementation lacks many important features. Firstly, it is not possible to derive types from another types. Inexistence of inheritance and polymorphism significantly decreases usability of ADT. Also syntax of object definition differs from the one proposed by the ANSI standard. In Oracle SQL the data type Person_T should be defined in two parts. First part declares all members of the type. The second defines bodies of all methods. This approach goes out of older definition of package syntax.

```
CREATE TYPE Person_T AS OBJECT
(
    first_name VARCHAR2(30),
    last_name VARCHAR2(30),
    birth_date DATE,
    MEMBER FUNCTION age RETURN NUMBER
);

CREATE TYPE BODY Person_T AS
    MEMBER FUNCTION age RETURN NUMBER
    IS
<the source code of the function age>
END;
```

The sub-type Employee_T can be defined either as independent type

```
CREATE TYPE Employee_T UNDER Person_T
(
    first_name VARCHAR2(30),
    last_name VARCHAR2(30),
    birth_date DATE,
    MEMBER FUNCTION age RETURN NUMBER
    id NUMBER(10),
    salary NUMBER(7,2)
);
```

or using reference

```
CREATE TYPE Employee_T AS OBJECT
(

    refPerson Ref Person_T,
    id NUMBER(10),
    salary NUMBER(7,2)
);
```

Together with object extensions of its original SQL language the Oracle Corporation brings newer generation of the OCI, the Oracle Call Interface, which provides the API between the applications and the database server. Specifically, the following capabilities have been added to the OCI:

- Support for execution of SQL statements that manipulate object data and object schema information.
- Support for declaring object references and instances as variables, fetching them from a database and passing them as input variables.
- Client side object cache.

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks object instances, which have been fetched by an OCI application from the server to the client side.

2.3 SQL-92

Though this ANSI/ISO standard is years older than its successor, its entry level is no doubt the most widespread database standard available. All current database vendors declare compatibility of its current products with it. It becomes perquisite of interoperability with many database interfaces as ODBC, JDBC and many others.

2.4 Object Databases and Object-Relational Wrappers

Advancements in object-oriented technologies have brought number of products that allow connecting of OO programs with object and relational data stores. Individual approaches differ in many aspects. We can mention number of supported languages, level of the transparency, database independence etc. One of important features is also quality of multi-user database access, which implies transactional database access, synchronisation between the database and memory.

Among the OO database management systems we can mention commercial products as ObjectStore, ONTOS DB or Versant.

ONTOS DB is an object-oriented database product developed by ONTOS, Inc., which provides C++ interface. ONTOS DB defines persistent storage capability for all and only objects derived from the class *Object*. It means that any class whose instances are to be stored persistently must be inherited from this class. On the other hand, all instances of all inherited classes are automatically persistent.

Persistent objects under ONTOS DB are accessed via pointers in an application. ONTOS DB retrieves objects from the database and places them in an application's memory space. There are three different concurrency control strategies available in the database, controlled by statements for beginning, aborting, committing transaction. Additionally the concurrency model supports savepoints. Savepoints allows storing checkpoint in the middle of running transaction and rolling the transaction partially back to this checkpoint. At the start of transaction the application can choose among pessimistic, optimistic, and time-based strategies. Associations between objects are modeled using uni-directional one-to-one references.

Versant, developed by Versant Object Technology Corp. binds not only to C++, but also to other languages as ANSI C and SmallTalk. Persistent capable classes allow both persistent and transient instances, depending on the way they are created. Similar to ONTOS DB the pointers in C++ are overloaded to ensure that referenced objects are located in memory when accessed. Association can be defined as both uni-directional and bi-directional. Depending on the declaration they return either one reference or collection of references that can be iterated. In case of bi-directional association the database automatically synchronizes both directions.

The ObjectStore database also access object instances via pointers. In difference with above discussed solutions it uses so called virtual memory mapping architecture. Missing object in the memory causes memory fault exception, handled by the system.

Apart of object-oriented databases, the closest to the goal of this thesis are object-relational wrappers. There are currently available wrappers, mostly ODMG compliant, like ObjectDriver developed by CERMICS Database Team. The ObjectDriver builds a virtual OODBMS over the relational database, accessible through an ODMG interface. It is available for Java and C++. Similar to it is also UFO-RDB wrapper developed at University of Frankfurt, but this one is fully oriented to Java language.

2.5 Résumé

The goal of this thesis is not implement yet another ODMG compliant object wrapper. It concerns on solution that supports object model with most of advanced object-oriented features with respect to relational structure of the database and its limitation.

The object model should support multiple inheritance. Unfortunately the multiple inheritance of ADT's is not standardised even in the latest SQL standard. Moreover, available object extensions regardless of the inheritance aspects implemented in current versions of database servers are not fully compliant with this standard. This fact substantially complicates the portability of solutions that require SQL-99 compliant relational features. Proposed solution should therefore require only SQL-92 entry-level compliant database. The mapping of object model onto database schema shouldn't, however, prevent utilisation of newer features in the future.

Portability of stored procedures is poorly supported. Databases use its own procedural SQL extensions with mutually different syntax and semantics. Thus the requirement to store object methods in SQL and invoke them from the application has low priority. This thesis doesn't consider persistency of object methods in the database at all. We consider methods as persistently defined in C++ code.

The unnecessary transformation between object and relational paradigms should be as most as possible transparent for the programmer to eliminate the impedance problem. This requirement includes an automatic SQL generation for object retrieval and manipulation. Object instances in the database should be automatically mapped into the memory during pointer dereference, as usual in object-oriented database interfaces. Changes of internal object states should be automatically propagated back to the database.

The approach allowing both persistent and transient instances has advantages over the system that allows only persistent instances of given class. The manipulation with both types of instances should be the same for the programmer.

Searching of objects according to their content is an essential extension of database application over the standard ones. Considering it, the solution should contain an object query language that is optimised for relational databases. Target OO language for us is the C++. Of course the query language should not be restricted by this assumption.

There should be supported uni-directional associations based on object pointers, as well as bi-directional associations between classes. The query language should incorporate manipulation with associated instances.

3 Object Persistency Concepts

This chapter formally defines the theoretical framework of the thesis. The relational nature of stored data is hidden behind the transparent object database interface. The object database interface is described formally using its database schema. The application accesses data using the query language similar to the relational algebra. The combination of features of relational algebra with object-oriented database model allows easy access to object instances and its sets with respect to the effectiveness of database operations. Operations of the object algebra, as we named the query language, are then translated to the SQL. This translation is described in two steps. Using of relational algebra allows more formalism and abstracts the translation from differences of particular SQL dialects.

3.1 Object Database Schema

The specific requirements concerning the representation of the object model in the relational database lead us to the proposal of the object database schema that handles differently class hierarchies and binary associations of classes.

Definition 3.1-1 – Object database schema

An *object-database schema* is a 7–tuple

$\mathbf{D_o} = (\mathbf{C}, \mathbf{M}, \mathbf{R}, \mathbf{isa}, \mathbf{mattr}, \mathbf{lclass}, \mathbf{rclass})$, where

1. $\mathbf{C}$ is a finite set of classes
2. $\mathbf{M}$ is a finite set of member attributes
3. $\mathbf{R}$ is a finite set of (binary) associations
4. $\mathbf{isa} \subseteq \mathbf{C} \times \mathbf{C}$ such that $(\mathbf{C}, \mathbf{isa})$ forms an acyclic *specialisation graph* .
5. $\mathbf{mattr} : \mathbf{C} \rightarrow \text{powerset}(\mathbf{M})$ is a total mapping such that

   $\mathbf{mattr}(X) \cap \mathbf{mattr}(Y) = \varnothing$ if $X \neq Y$
6. both $\mathbf{lclass} : \mathbf{R} \rightarrow \mathbf{C}$ and $\mathbf{rclass} : \mathbf{R} \rightarrow \mathbf{C}$ are total mappings

♦

According to this definition we will assume the object–database schema to consist of set of classes, forming acyclic subclass hierarchy. A set (possibly empty) of member attributes is assigned to each class. Moreover the scheme contains set of binary class associations. Associations are part of object-oriented paradigm. Through associations, objects can point one to another. This feature allows objects to create a network of interconnected nodes. The relationships between objects can be divided into categories. Depending on the point of view, we obtain different classification of relationships. As many other approaches, we concern now only on binary, not ternary or n-ary associations. Thus, each of associations interconnects two not necessarily different classes.

Definition 3.1-2 – Generalised specialisation graph

The *generalised specialisation graph* $(\mathbf{C}, \mathbf{isa}^*)$ is a reflexive and transitive closure of the specialisation graph $(\mathbf{C}, \mathbf{isa})$

1. $C_i \ \mathbf{isa}^* \ C_i$ for each $C_i \in \mathbf{C}$
2. Let $C_i \ \mathbf{isa} \ C_j$. Then $C_i \ \mathbf{isa}^* \ C_j$

3. Let C_i **isa*** $C_j \wedge C_j$ **isa*** C_k then C_i **isa*** C_k

♦

While specialisation graph determines direct predecessors and successors, its generalised form allows simpler formal manipulation with all predecessors and successors of particular class.

Definition 3.1-3 – Predecessors and successors

Specialisation graph ($\mathbf{C}$, **isa**), respectively its generalised form ($\mathbf{C}$, **isa***) defines four additional total mappings **pred**, **pred***, **succ** and **succ*** as follows:

1. **pred**: $\mathbf{C} \rightarrow$ powerset($\mathbf{C}$), such that **pred**(C) $=_{def}$ {X | C **isa** X} which returns all direct predecessors of given class C.
2. **pred***: $\mathbf{C} \rightarrow$ powerset($\mathbf{C}$), such that **pred***(C) $=_{def}$ {X | C **isa*** X} which returns all predecessors, direct or indirect of given class C.
3. **succ**: $\mathbf{C} \rightarrow$ powerset($\mathbf{C}$), such that **succ**(C) $=_{def}$ {X | X **isa** C} which returns all direct successors of class C.
4. **succ***: $\mathbf{C} \rightarrow$ powerset($\mathbf{C}$), such that **succ***(C) $=_{def}$ {X | X **isa*** C} which returns all successors, direct or indirect of given class C.

♦

Because we allow multiple inheritance (and thus the **isa** hierarchy doesn't form a tree but an acyclic graph), each of four above–defined mappings can return a set containing more than one element.

Example 3.1-1 – object database schema

Consider database schema with $\mathbf{C} = \{C_1,C_2,C_3,C_4,C_5,C_6,C_7\}$ and with member attributes $\mathbf{M} = \{M_1,M_2,M_3,M_4,M_5,M_6,M_7\}$ where each class contains one member attribute with the same index i.e. **memattr**(C_i) = {M_i}. Consider the **isa** hierarchy defined as shown on following picture.

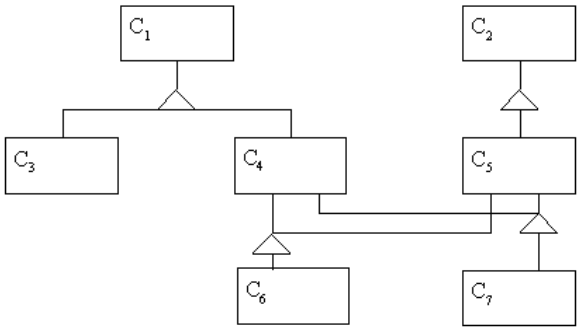


Figure 3.1-1 Class hierarchy

According to previous definitions **pred**(C_5) = {C_2}. Class can have none or more than one direct predecessor. In the graph above **pred**(C_2) = ∅ and **pred**(C_6) = {C_4, C_5}.

♦

From the definition of **isa** and **isa*** relations immediately follow below stated properties.

Lemma 3.1-1

1. $\forall X \in \mathbf{C}$ (X $\in$ **pred***(X)) – reflexivity

2. $\forall X \in \mathbf{C}\ (X \in \mathbf{succ}^*(X))$ – reflexivity
3. $\forall X \in \mathbf{C}\ (X \notin \mathbf{pred}(X))$
4. $\forall X \in \mathbf{C}\ (X \notin \mathbf{succ}(X))$
5. $\forall X,Y \in \mathbf{C}\ (((X \in \mathbf{pred}^*(Y)) \wedge (Y \in \mathbf{pred}^*(X))) \Rightarrow (X = Y))$ – antisymmetry
6. $\forall X,Y \in \mathbf{C}\ (((X \in \mathbf{succ}^*(Y)) \wedge (Y \in \mathbf{succ}^*(X))) \Rightarrow (X = Y))$ – antisymmetry
7. $\forall X \in \mathbf{C}\ (\mathbf{pred}(X) \subset \mathbf{pred}^*(X))$
8. $\forall X \in \mathbf{C}\ (\mathbf{succ}(X) \subset \mathbf{succ}^*(X))$
9. $\forall X,Y \in \mathbf{C}\ ((X\ \mathbf{isa}^*\ Y) \Rightarrow (\mathbf{pred}^*(Y) \subseteq \mathbf{pred}^*(X))$
10. $\forall X,Y \in \mathbf{C}\ ((X\ \mathbf{isa}\ Y) \Rightarrow (\mathbf{pred}^*(Y) \subset \mathbf{pred}^*(X))$
11. $\forall X,Y \in \mathbf{C}\ ((X\ \mathbf{isa}^*\ Y) \Rightarrow (\mathbf{succ}^*(X) \subseteq \mathbf{succ}^*(Y))$
12. $\forall X,Y \in \mathbf{C}\ ((X\ \mathbf{isa}\ Y) \Rightarrow (\mathbf{succ}^*(X) \subset \mathbf{succ}^*(Y))$

Proof:

1., 2.: From reflexivity of generalised specialisation graph (Definition 3.1-2) follows that

$\forall X \in \mathbf{C}\ (X\ \mathbf{isa}^*\ X)$).

According to Definition 3.1-3 hold

$\forall X \in \mathbf{C}\ (X\ \mathbf{isa}^*\ X)) \Rightarrow \forall X \in \mathbf{C}\ (X \in \mathbf{pred}^*(X))$

$\forall X \in \mathbf{C}\ (X\ \mathbf{isa}^*\ X)) \Rightarrow \forall X \in \mathbf{C}\ (X \in \mathbf{succ}^*(X))$

3., 4.: From $X \in \mathbf{succ}(X)$, respectively from $X \in \mathbf{pred}(X)$ follows that $X\ \mathbf{isa}\ X$, which is in contradiction with assumed acyclicity of generalisation graph.

5., 6.: Let $X \neq Y$, $X \in \mathbf{pred}^*(Y)$ and $Y \in \mathbf{pred}^*(X)$, respectively $X \in \mathbf{succ}^*(Y)$ and $Y \in \mathbf{succ}^*(X)$. In both cases exist classes $C_1, \ldots C_k, C_{k+1}, \ldots C_n$ such that $X\ \mathbf{isa}\ C_1\ \mathbf{isa} \ldots \mathbf{isa}\ C_k\ \mathbf{isa}\ Y\ \mathbf{isa}\ C_{k+1}\ \mathbf{isa} \ldots \mathbf{isa}\ C_n\ \mathbf{isa}\ X$, which is in contradiction with assumed acyclicity of the specialisation graph.

7.: $\forall X,Y \in \mathbf{C}((Y \in \mathbf{pred}(X)) \Leftrightarrow (X\ \mathbf{isa}\ Y) \Rightarrow (X\ \mathbf{isa}^*\ Y) \Leftrightarrow (Y \in \mathbf{pred}^*(X)))$.
Thus $\forall X \in \mathbf{C}\ (\mathbf{pred}(X) \subseteq \mathbf{pred}^*(X))$. Moreover, according to (1) and (3) hold $X \in \mathbf{pred}^*(X)$ and $X \notin \mathbf{pred}(X)$.

8.: $\forall X,Y \in \mathbf{C}((Y \in \mathbf{succ}(X)) \Leftrightarrow (Y\ \mathbf{isa}\ X) \Rightarrow (Y\ \mathbf{isa}^*\ X) \Leftrightarrow (Y \in \mathbf{succ}^*(X)))$.
Thus $\forall X \in \mathbf{C}\ (\mathbf{succ}(X) \subseteq \mathbf{succ}^*(X))$. The inequality of sets results from (2) and (4).

9.: Let $(X\ \mathbf{isa}^*\ Y)$ and $(Z \in \mathbf{pred}^*(Y))$. According to Definition 3.1-3 holds $(Z \in \mathbf{pred}^*(Y)) \Leftrightarrow (Y\ \mathbf{isa}^*\ Z)$. From the transitivity of $\mathbf{isa}^*$ we get $(X\ \mathbf{isa}^*\ Z)$, i.e. $(Z \in \mathbf{pred}^*(X))$.

10.: Let $(X\ \mathbf{isa}\ Y)$ and $(Z \in \mathbf{pred}^*(Y))$. From the definition of generalised specialisation graph $(\mathbf{C}, \mathbf{isa}^*)$ follows, that $(X\ \mathbf{isa}^*\ Y)$. From (9) we know that $\mathbf{pred}^*(Y) \subseteq \mathbf{pred}^*(X)$. To prove the rest of the affirmation we show that $X \notin \mathbf{pred}^*(Y)$. Let suppose the converse statement $X \in \mathbf{pred}^*(Y)$. Because $Y \in \mathbf{pred}^*(X)$, from (5) follows that X equals to Y. This equivalence produces $(X\ \mathbf{isa}\ X)$ which is in contradiction with acyclicity.

We know, that $X\ \mathbf{isa}^*\ Y$. Let suppose that $X \in \mathbf{pred}^*(Y)$, i.e. that $Y\ \mathbf{isa}^*\ X$. Then necessarily X is equal to Y from antisymmetry of $\mathbf{pred}^*$. But this equivalence is in contradiction with acyclicity of

11.: Similar to (9)

12.: Similar to (10)

♦

Definition 3.1-4 – Minimal an maximal elements

Let $\mathtt{C} \subseteq \mathbf{C}$. *Minimal element of set* $\mathtt{C}$ is any class, which has no successor within the set. Correspondingly, we can define *maximal element of set* $\mathtt{C}$. Total mappings

1. **min**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$)

where **min**(C) = {X∈C | **succ**(X)∩C = ∅} and

2. **max**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$)

where **max**(C) = {X∈C | **pred**(X)∩C = ∅} map sets of classes to sets of its minimal, respectively maximal elements.

♦

Classes in set **max**($\mathbf{C}$) have no predecessor at all and form roots of inheritance in the object model. Considering set $\mathbf{C}$ from an Example 3.1-1 we obtain results **max**($\mathbf{C}$) = {C_1, C_2} while **min**($\mathbf{C}$) = {C_3, C_6, C_7}.

Many of presented algorithms uses notion of *common predecessor*, *common successor*, or especially *least common predecessor* and *greatest common successor*. To define such concepts, we introduce four additional total mappings.

Definition 3.1-5 – (Least) common predecessors and (greatest) common successors

1. **cp**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$), such that **cp**(C) $=_{def} \cap_{X \in C}$**pred***(X) which returns all common predecessors of given set.
2. **cs**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$), such that **cs**(C) $=_{def} \cap_{X \in C}$**succ***(X) which returns all common successors of given set.
3. **lcp**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$), such that **lcp**(C) $=_{def}$ **min**(**cp**(C)) which returns all least common predecessors of given set.
4. **gcs**: powerset($\mathbf{C}$) $\rightarrow$ powerset($\mathbf{C}$), such that **cs**(C) $=_{def}$ **max**(**cs**(C)) which returns all greatest common successors of given set.

♦

For example **cp**(C_3, C_6) = {C_1, C_3} ∩ {C_1, C_2, C_4, C_5, C_6} = {C_1}. Featuring multiple inheritance all four functions can return multi-element sets. Here **cp**(C_6, C_7) = {C_1, C_2, C_4, C_5} and **lcp**(C_6, C_7) = {C_4, C_5}.

Following definition is essential for manipulation with associations.

Definition 3.1-6 – Connected classes

Let C_i, $C_j \in \mathbf{C}$. We say that classes C_i and C_j *are connected*, if and only if the set **cs**({C_i, C_j}) is not empty.

Pairs of classes C_1 and C_6, respectively C_4 and C_5 in Example 3.1-1 are connected. Classes C_3 and C_5 are not.

♦

Lemma 3.1-2

1. ∀X∈$\mathbf{C}$ (X is connected with X) – reflexivity

2. $\forall X,Y \in \mathbf{C}$ (X is connected with Y) $\Leftrightarrow$ (Y is connected with X) – symmetry

3. Let X **isa** Y. Then X is connected with Y.

Proof:

1.: Because $X \in$ **succ**$^*(X)$, the set **cs**$(\{X\})$ that is equal to **succ**$^*(X)$, is not empty.

2.: Symmetry follows from the fact, that the definition of common predecessor doesn't recognise order of classes in the set.

3.: $X \in$ **succ**$^*(X) \subset$ **succ**$^*(Y)$. Thus, $X \in$ **succ**$^*(X) \cap$ **succ**$^*(Y)$.

♦

Similar way, as the **isa** relation is extended, we can define a generalised form of **mematttr** mapping to obtain all member attributes including those indirectly inherited.

Definition 3.1-7 – Generalised membership

Generalised membership is determined by the total mapping **mematttr***: $\mathbf{C} \rightarrow$ powerset($\mathbf{M}$) such that **mematttr**$^*(X) = \cup_{Y \in \mathbf{pred}^*(\{X\})}$**mematttr**$(Y)$.

♦

In the Example 3.1-1 we among others obtain **mematttr**$^*(C_6) = \{M_1, M_2, M_4, M_5, M_6\}$.

The classes in the object database should be stored finally in the relational database. Table schemas usually describe their content. We can use similar approach here to unify both object and relational approaches.

Definition 3.1-8 – (Inherited) class schema

For each class $C \in \mathbf{C}$ we define its *class schema* $C(\mathsf{P}, \mathsf{M})$ and its *inherited class schema* $C^*(\mathsf{M}^*)$, where

1. $\mathsf{P} =_{def}$ **pred**(C) is a set of direct parent classes, and
2. $\mathsf{M} =_{def}$ **mematttr**(C) is a set of member attributes
3. $\mathsf{M}^* =_{def}$ **mematttr**$^*(C)$ is a set of member attributes including inherited members.

♦

Lemma 3.1-3

Let $Y \in \mathbf{C}$ is class with schema $Y(\mathsf{P}_y, \mathsf{M}_y)$. Let X **isa*** Y. Then $\mathsf{M}_x^* \supseteq \mathsf{M}_y^*$.

Proof:

From X **isa*** Y according to Lemma 3.1-1 (9) follows that **pred**$^*(Y) \subseteq$ **pred**$^*(X)$. Then also $\cup_{C \in \mathbf{pred}^*(\{Y\})}$**mematttr**$(C) =$ **mematttr**$^*(Y) = \mathsf{M}_y^* \subseteq \mathsf{M}_x^* =$ **mematttr**$^*(X) = \cup_{C \in \mathbf{pred}^*(\{X\})}$**mematttr**$(C)$.

♦

Relationships are binary and typed. They associate pairs of object instances, where each object must belong to the specified class. The required types can be described using schema of the association.

Definition 3.1-9 – Association schema

For each association $R \in \mathbf{R}$ we define its *association schema* $R(C_l, C_r)$

1. $C_l =_{def}$ **lclass**(R) is a *left-side class* of an association, and
2. $C_r =_{def}$ **rclass**(R) is a *right-side class* of an association

We will use terms *left* and *right side of the association* to refer to the classes on first and second place of association's schema

♦

Definition 3.1-10 – Domain of member attribute

Let $M \in \mathbf{M}$ is a member attribute. *Domain of member attribute* M, denoted as **dom**(M), is a set of all allowed values of attribute M.

♦

Definition 3.1-11 – (Inherited) domain of class

Let $C \in \mathbf{C}$. Then

1. **dom**(C) = $\times_{M \in \mathbf{mematttr}(C)}$**dom**(M) is *domain of class* C
2. **dom***(C) = $\times_{M \in \mathbf{mematttr}^*(C)}$**dom**(M) is *inherited domain of class* C

♦

Definition 3.1-12 – Object database instance

An *object database instance* of schema $\mathbf{D_o}$ is any quadruple $\mathbf{D} = (\mathbf{O}, \mathbf{inst}, \mathbf{val}, \mathbf{rel})$, where

1. $\mathbf{O}$ is a finite ordered set of object instances.
2. **inst** : $\mathbf{C} \to P(\mathbf{O})$ is a total mapping such that
   a. if C_j **isa*** C_i then **inst**(C_j) $\subseteq$ **inst**(C_i)
   b. if $o \in$ **inst**(C_i) $\cap$ **inst**(C_j) then exists its common successor C such $o \in$ **inst**(C)
3. **val** : $\mathbf{O} \times \mathbf{M} \to \cup_{M \in \mathbf{M}}$ **dom**(M) is defined as follows
   a. **val**(o, M) $\in$ **dom**(M) if $o \in$ **inst**(C) and $M \in$ **mematttr**(C)
   b. elsewhere **val** is not defined
4. **rel** $\subseteq \cup_{R \in \mathbf{R}}$(R $\times$ **inst**(**lclass**(R)) $\times$ **inst**(**rclass**(R)))

According to this definition the mapping **inst** assigns a set of instances to each class and mapping **val** assigns a value to each relevant attribute of that class. Relation **rel** defines couples of associated object instances.

To simplify further text, we will replace symbol "**inst**(C)" by shorter "**C**" in bold font. By the same reason, the expression "**val**(o, M)" is abbreviated as "o.M"

♦

Definition 3.1-13 – Instance class

Mapping $\mathbf{class_O}$: $\mathbf{O} \to \mathbf{C}$ is inverse mapping to **inst** that assigns class to any object instance such that **class**(o) is a minimal element of the set $\{C \mid o \in \mathbf{inst}(C)\}$.

♦

Lemma 3.1-4

Each object instance o has the set {C | o ∈ **inst**(C)}one and only one minimal element.

Proof:

This fact follows from points *2a* and *2b* of object database instance definition. If there were two different minimal elements C_1 and C_2, there should exist its common successor C such that **inst**(C) contains the object. This is in contradiction with our choice of minimal elements.

♦

Definition 3.1-14 –Memory attribute class mappings

We can define total mapping $\mathbf{class_M}$: $\mathsf{M} \rightarrow \mathsf{C}$ such that M ∈ **mattr**($\mathbf{class_M}$(M)).

♦

3.2 Object Algebra

This chapter describes the object algebra – a query language introduced in this thesis. Object algebra comes from the relational algebra, but its concept was modified to suite the object database model with its operations as well as the relational database concept, which is used to store persistent data.

Resulting set of language operators presented and implemented in this thesis is a compromise between the possible power of the query language, requirements of programmers and effectiveness of query evaluation.

Having the relational database engine to store persistently object data manipulated by the application, we can provide object instance retrieval based on internal values of member attributes. This feature in its general form is extension to usual possibilities, offered by non-persistent object-oriented languages. Its special form – retrieving of one complete particular instance having specified object identification is must-to-have feature. It corresponds to dereferencing of the object address (object pointer) in the object–oriented programming language. The general form we call *object selection* due to its similarity to the relational paradigm.

One of requirements to object selection is to provide results in form maintainable by the object-oriented program. To meet this requirement, we will allow only such queries, which returns sets of objects, derived from exactly one specified parent class, preserving their polymorphism. Then we can consider the result the polymorph set of successors of the given class.

3.2.1 Predicates

Before we can define the object selection, we need a notion of Boolean predicate.

Definition 3.2-1 – Boolean predicate

A *Boolean predicate* is any function

$\varphi : \mathsf{O} \rightarrow \{true, false\}$

We could consider all predicates as totally defined. If the predicate couldn't be evaluated on particular object, it can be easily extended to return *false* in all such cases. But for practical reasons, we will assume, that the given predicate is applicable (valid) only on instances of some classes.

For each predicate φ we define set

mattr(φ) of member attributes, that appear in the formula and also set

valid(φ) of classes the predicate can be evaluated on. The set **valid**(φ) can be determined from following equivalence.

$(C \in \mathbf{valid}(\varphi)) \Leftrightarrow (\mathbf{mematttr}(\varphi) \subseteq \mathbf{mematttr}^{*}(C))$

Because the set **mematttr***(C) grows going from predecessors to successors, any predicate that can be evaluated on particular class can be also evaluated on all its successors.

♦

Definition 3.2-2 – Atomic Boolean predicate

An *atomic Boolean predicate* is any expression in form

1. true, with **mematttr**(true) = ∅ and **valid**(true) = C
2. false, with **mematttr**(true) = ∅ and **valid**(true) = C
3. $expr_1$ Φ $expr_2$, with

mematttr($expr_1$ Φ $expr_2$) = **mematttr**($expr_2$) ∪ **mematttr**($expr_2$)

valid($expr_1$ Φ $expr_2$) = **valid**($expr_1$) ∩ **valid**($expr_2$)

where $expr_1$ and $expr_2$ are comparable well-bracket expressions consisting of names of member attributes, constants and applicable operators and symbol Φ stands for one of operators "=", "≠" , "≤ ", "≥ , "< , ">" and "like".

To those variants, we can add fourth form of the atomic predicate for each object instance o

4. o, with **mematttr**(o) = ∅ and **valid**(o) = C , where (o(x) = *true*) ⇔ (x = o)

♦

Definition 3.2-3 – General atomic predicate

Using logical operators, we can construct more general predicates from atomic ones.

Let φ_i, resp. φ_j are Boolean predicates valid on C_i, resp. C_j. Then

1. $\varphi_i \wedge \varphi_j$ is Boolean predicate valid on $C_i \cap C_j$ such that

   $(\varphi_i \wedge \varphi_j)(o) =_{def} (\varphi_i(o)) \wedge (\varphi_j(o))$
2. $\varphi_i \vee \varphi_j$ is Boolean predicate valid on $C_i \cap C_j$ such that

   $(\varphi_i \vee \varphi_j)(o) =_{def} (\varphi_i(o)) \vee (\varphi_j(o))$
3. $\neg\varphi_i$ is Boolean predicate valid on C_i such that

   $(\neg\varphi_i)(o) =_{def} \neg(\varphi_i(o))$
4. (φ_i) is Boolean predicate valid on C_i such that

   $(\varphi_i)(o) =_{def} \varphi_i(o)$

♦

3.2.2 Object selection

Definition 3.2-4 – Object selection

Let C is a class. Let φ is a Boolean predicate applicable on C. Then

the *object selection* $\mathbf{C}(\varphi) = \{x \in \mathbf{inst}(C) \mid \varphi(x) = true\} = \mathbf{inst}(C) \cap \varphi^{-1}(true)$

♦

Example 3.2-1 – Object database

Consider simple object database hierarchy as shown on following picture.

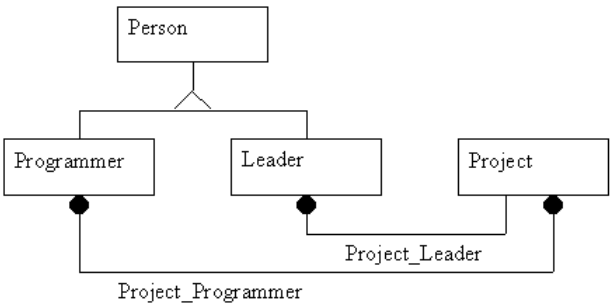


Figure 3.2-1 Object model

Member attributes are attached to classes as follows:

mattr(Person)={Name,Age},
mattr(Programer)={PreferredLanguage},
mattr(Leader)=∅,
mattr(Project)={Title,Language}

Object selection can retrieve all projects written in C++ using statement **Project**(Language='C++'). We can also obtain set of all persons younger than 40 years by object algebra query **Person**(Age<40). To find out all programmers in ADA older than 35 years we have to construct another query in form **Programmer**((Age>=35)∧(PreferredLanguage='ADA')).

♦

Relational algebra defines beside selection also projection as one of basic operators. As we want to retrieve a polymorph set of instances with all member attributes and properties, the projection has no use in object algebra. Here is first of differences with relational algebra, where all resulting tuples are supposed to have the same structure.

3.2.3 Set operators

Prospective adoption of set operators as set-union, set-intersection and set-difference known from relational algebra is all but straightforward. Our demand of uniquely defined resulting class of any expression restricts utilisation of set operators.

3.2.3.1 Set-difference

We start from the less complicated operator – the set-difference. Suppose that predicate φ_1 is valid on C_1 and φ_2 is valid on C_2. Then all resulting instances of the set-difference

$\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2)$

can be considered the instances of C_1. The expression **Person****Programmer**(Age>=35) returns all persons except programmers older than 35. There is no limitation on classes that can be subtracted. Classes need not be connected as in expression

Programmer(PreferredLanguage='C++')**Leader**(Age<50).

and the set-difference will work even if subtracted classes will not have common predecessor as in query

Programmer(PreferredLanguage='C++')**Project**(Language<'C++').

Lemma 3.2-1

Let predicate φ_1 is valid on C_1, let φ_2 is valid on C_2.

1. if classes C_1 and C_2 are not connected, then $\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1)$
2. if C_1 **isa*** C_2, then $\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_1(\varphi_2) = \mathbf{C}_1(\varphi_1 \wedge \neg\varphi_2)$

Proof:

1: Because classes are not connected, the intersection $\mathbf{inst}(C_1) \cap \mathbf{inst}(C_2)$ must be according the Definition 3.1-12 empty. Since $\mathbf{C}_1(\varphi_1) \subseteq \mathbf{inst}(C_1)$ and $\mathbf{C}_2(\varphi_2) \subseteq \mathbf{inst}(C_2)$, also $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \varnothing$. From emptiness of the intersection follows that $\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1)$

2: From $\mathbf{C}_1(\varphi_1) \subseteq \mathbf{inst}(C_1)$ follows that $\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1)\backslash(\mathbf{inst}(C_1) \cap \mathbf{C}_2(\varphi_2))$.

♦

3.2.3.2 Set-intersection

In case of set-difference we can positively determine the resulting class. On the other hand the set-intersect operator can be defined as

$\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1)\backslash(\mathbf{C}_2(\varphi_2)\backslash\mathbf{C}_1(\varphi_1)) = \mathbf{C}_2(\varphi_2)\backslash(\mathbf{C}_1(\varphi_1)\backslash\mathbf{C}_2(\varphi_2))$

Due to the symmetry of the operator we can consider resulting instances to belong to any of both classes.

Lemma 3.2-2

Let predicate φ_1 is valid on C_1, let φ_2 is valid on C_2. then

1. if classes C_1 and C_2 are not connected, then $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \varnothing$
2. if C_1 **isa*** C_2, then $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \mathbf{C}_1(\varphi_1) \cap \mathbf{C}_1(\varphi_2) = \mathbf{C}_1(\varphi_1 \wedge \varphi_2)$
3. if classes C_1 and C_2 are connected, and $\mathbf{gcs}(C_1,C_2)=\{C\}$, then

$\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \mathbf{C}(\varphi_1) \cap \mathbf{C}(\varphi_2) = \mathbf{C}(\varphi_1 \wedge \varphi_2)$

Proof:

1: Classes are not connected. Thus $\mathbf{inst}(C_1) \cap \mathbf{inst}(C_2) = \varnothing$. Since $\mathbf{C}_1(\varphi_1) \subseteq \mathbf{inst}(C_1)$ and $\mathbf{C}_2(\varphi_2) \subseteq \mathbf{inst}(C_2)$, also $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \varnothing$.

2: $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) = \mathbf{inst}(C_1) \cap \varphi_1^{-1}(\mathit{true}) \cap \mathbf{inst}(C_2) \cap \varphi_2^{-1}(\mathit{true})$. Because $\mathbf{inst}(C_1) \subseteq \mathbf{inst}(C_2)$, we can rewrite the formula in form $\mathbf{inst}(C_1) \cap \varphi_1^{-1}(\mathit{true}) \cap \mathbf{inst}(C_1) \cap \varphi_2^{-1}(\mathit{true})$ which correspond to the query $\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_1(\varphi_2)$

3: Because there is only one class derived from both classes C_1 and C_2, $\mathbf{inst}(C_1) \cap \mathbf{inst}(C_2) = \mathbf{inst}(C)$.

♦

In case when there are more independent common successors, i.e. if $\mathbf{gcs}(\{C_1,C_2\})$ contains more than one element we cannot re-formulate the intersection by corresponding object selection formula. Choosing one of common successors $C \in \mathbf{gcs}(\{C_1,C_2\})$ leads to more specific substitution $\mathbf{C}(\varphi_1 \wedge \varphi_2)$.

This query is of course not equivalent with the original one.

3.2.3.3 Set-union

Set-union is last of three set operators used in relational algebra. In its general form it doesn't correspond with our requirement of one resulting class. Supposing the query in form

$\mathbf{C}_1(\varphi_1) \cup \mathbf{C}_2(\varphi_2)$

the result should contain elements belonging to either class. To represent the result as polymorph set, we need to choose the result class from potentially more common successors. In some cases the common predecessor need not even exist. Transformation of the query to form

$\mathbf{C}(\varphi_1) \cup \mathbf{C}(\varphi_2) = \mathbf{C}(\varphi_1 \vee \varphi_2)$

where C is one of common predecessors of both classes may fail. Predicates φ_1 and φ_2 may not be valid on class C due to utilising some of not yet defined member attributes.

3.2.3.4 Object algebra without set operators

After consideration of advantages and disadvantages of set operators we choose not to implement them in its general form. As shown in previous sections most of cases are covered by object selection together with its polymorph behaviour. Some of others are feasible by utilisation of operators on associations as discussed in next chapters.

Sometimes, the inexistence of efficient object algebra query may be caused by improper object design of the application.

Example 3.2-2 – another variant of object database

This decision makes impossible to retrieve both programmers and project leaders younger than forty. Note that there can be persons, which don't write programs nor lead projects in the database. Query in form

Programmer(Age<40)∪**Leader**(Name<40)

is not supported. To allow such query, the database schema should be slightly redesigned as shown on next picture.

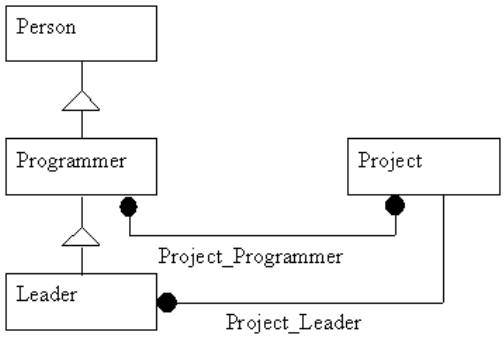


Figure 3.2-2 Modified object model

On this schema, the query in form **Programmer**(Age<40) match our needs. In this case the reasonable modification of object model also allows project leader to take part of programming job.

♦

3.2.4 Associations and object algebra

Above described selection operator is stronger than referencing objects from another objects by pointers. Pointers are the only way of accessing (dynamically allocated) objects in higher object-oriented languages as C++. Whenever object should be associated with another one, pointers or

sets of pointers must be defined inside instances. Associations based on pointers are unidirectional. It is possible to traverse them only in one direction. Whenever associations should be bi-directional, programmer must solve bi-directionality himself combining two independent references. In contrast with this approach relational databases provide mechanism for representing associations with arbitrary arity having schema $R(C_1, C_2,\ldots, C_n)$. Depending on the value of parameter n in the definition of the association, we can speak about binary associations (n=2), ternary associations (n=3), or generally about n-ary associations.

Associations represented in the relational database are always multi-directional. For each (n-1)-tupl $o_1, o_2, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n$ of object instances we can determine the set R_i of instances of class C_i associated with given (n-1)-tupl as follows

Definition 3.2-5 – Associated instances

Let R is n-ary association with schema $R(C_1, C_2,\ldots, C_n)$. For each $1<=i<=n$ we define mapping R_i : $\mathbf{inst}(C_1)\times\ldots\times\mathbf{inst}(C_{i-1})\times\mathbf{inst}(C_{i+1})\times\ldots\times\mathbf{inst}(C_n) \rightarrow \text{powerset}(\mathbf{inst}(C_i))$ such that

$R_i(o_1, o_2, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n) =_{def} \{ o \mid (o_1, o_2, \ldots, o_{i-1}, o, o_{i+1}, \ldots, o_n)\in R\}$

♦

We chose to support only binary associations in our object model, as well as most OODBMS standards including the standard ODMG-93 do. On the other hand, we want to preserve other advantages offered by relational model in the object algebra.

Each instance of binary association R with the schema $R(C_1, C_2)$ can be considered as a subset of Cartesian product $\mathbf{inst}(C_1) \times \mathbf{inst}(C_2)$. We can define two unary operators **left** and **right**, which return instances of left, respectively right class associated with at least one instance on the opposite site.

1. $\mathbf{left}(R) = \{o \mid \exists x\in\mathbf{inst}(C_2)(<o,x>\in R\}$ and
2. $\mathbf{right}(R) = \{o \mid \exists x\in\mathbf{inst}(C_1) (<x,o>\in R\}$

Possible cardinalities of associated instance sets $R_1(o)$ and $R_2(o)$ determine cardinality of the association R. If the cardinality of one of those sets cannot exceed one we speak about *one-to-many* association. *One-to-one* associations limit cardinality of both sets to one. The most general, *many-to-many*, associations don't limit number of associated objects on any side. Associations of different cardinalities can slightly differ in implementation details, but in scope of object algebra we can treat them uniformly.

3.2.5 Manipulating associations

Operators **left** and **right** together with object selection returns sets of object instances and can be used to query the object database. To implant possibilities offered by the database engine, we introduce set of operations for manipulation with associations. Beside association defined by the programmer we can use implicit associations defined by the class hierarchy.

Definition 3.2-6 – Natural associations

We introduce natural association $C_i\otimes C_j$ of any two classes C_i and C_j. in the class inheritance graph. It can be defined as

$C_i\otimes C_j = \{<o_i,o_j> \mid (o_i \in \mathbf{inst}(C_i)) \wedge (o_j \in \mathbf{inst}(C_j)) \wedge (o_i = o_j)\} = \{<o,o> \mid o \in (\mathbf{inst}(C_i) \cap \mathbf{inst}(C_j))\}$

♦

This association associates each instance belonging to both classes with self. If classes C_i and are not connected, the instance of the natural association is empty because there exist no instance belonging to both classes.

Example 3.2-3 – Natural association

Because class *Leader* is derived from class *Person*, the association Person⊗Leader contains one element for each project leader in the database. Both polymorph sets **left**(Person⊗Leader) and **right**(Person⊗Leader) are the same. The difference is in the base class of both sets – *Person* versus *Leader*.

♦

Object algebra provides powerful instruments for manipulation with associations. By the application of restriction, chaining, and reverse operators the programmer can construct complex associations from basic ones.

Definition 3.2-7 – Restricted association

Let R is a binary association between classes C_i and C_j. Let φ_i, resp. φ_j are Boolean predicates valid on classes C_i, resp. C_j.

1. *Left-side restricted association* φ_i*R is a binary association between classes C_i and C_j such that

   $(<o_i,o_j> \in (\varphi_i*R)) \Leftrightarrow (<o_i,o_j> \in R \wedge \varphi_i(o_i))$

2. *Right-side restricted association* R*φ_j is a binary association between classes C_i and C_j such that

   $(<o_i,o_j> \in (R*\varphi_j)) \Leftrightarrow (<o_i,o_j> \in R \wedge \varphi_j(o_j))$

♦

Example 3.2-4 – Restricted association

All programmers working on project 'Amoeba' is possible to obtain utilising query **right**((Title='Amoeba')***Project_Programmer**).

Correspondingly, the query **left**((Title<>'Amoeba')***Project_Programmer***(Name='Smith')) returns all projects except project Amoeba, on whose works programmer Smith.

♦

Definition 3.2-8 – Chained association

Let R_{ij} and R_{jk} are two associations between classes C_i and C_{j1} respectively C_{j2} and C_k. We define *chained association* R_{ij}*R_{jk} between classes C_i and C_k as

$(<o_i,o_k> \in (R_{ij}*R_{jk})) \Leftrightarrow (\exists C \in \mathbf{gcs}(C_{j1}, C_{j2}), \exists o \in C)(<o_i,o> \in R_{ij} \wedge <o,o_k> \in R_{jk})$

♦

Natural association together with restriction and chaining can be used instead of set-intersection operator. This fact results from equation

$\varphi_i*(C_i \otimes C_j)*\varphi_j = \{<o,o> \mid o \in (\mathbf{inst}(C_i) \cap \mathbf{inst}(C_j)) \wedge \varphi_i(o) \wedge \varphi_j(o)\} = \mathbf{C}_i(\varphi_i) \cap \mathbf{C}_j(\varphi_j)$

By analogy, the general set-intersection

$\mathbf{C}_1(\varphi_1) \cap \mathbf{C}_2(\varphi_2) \cap ... \cap \mathbf{C}_k(\varphi_k)$

can be expressed as

$\varphi_1*(C_1 \otimes C_2)*\varphi_2*(C_2 \otimes C_3)*...*(C_{k-1} \otimes C_k)*\varphi_k$

Note that if **gcs**(C_{j1}, C_{j2}) is empty i.e. the inner classes are not connected, chaining of associations produces an empty association. Due this reason the associations **Project_Programmer** and **Project_Leader** cannot be chained in any order. All four combinations (it is possible to chain an

association with itself) are empty. To make finding out all programmers working under supervision of leader 'Thompson' possible, we need to swap sides of one of associations.

Definition 3.2-9 – Reversed association

Let R is associations between classes C_i and C_j. *Reversed association* -R is such association between classes C_j and C_i that

$(<o_j,o_i> \in (-R)) \Leftrightarrow (<o_i,o_j> \in R)$

Now we can build the query to solve above-mentioned problem.

right((Name='Thompson')***-Project_Leader*Project_Programmer**)

We can even find out all collaborators of programmer Black using query

right((Name='Black')***-Project_Programmer*Project_Programmer**)

♦

3.2.6 Operator precedence and associativity

All binary operators of the same priority except left restriction evaluates from left to right. It means that

$R_i*R_j*R_k$ evaluates as $(R_i*R_j)*R_k$,

$R*\varphi*\psi$ evaluates as $(R*\varphi)*\psi$,

but

$\psi*\varphi*R$ evaluates as $\psi*(\varphi*R)$

$\psi*\varphi*R_i*R_j$ evaluates as $(\psi*(\varphi*R_i)*R_j)$

Evaluation from right to left in case of left restriction is enforced, because operation $\psi*\varphi$ on two predicates is not defined

Priorities of operators, ordered from higher to lower are

1. natural association
2. reverse
3. chaining, restriction
4. **left**(), **right**(), selection

Lemma 3.2-3

Above defined operators meet following rules

1. $-(-R) = R$
2. $-(R_i*R_j) = (-R_j)*(-R_i)$
3. $(R_i*R_j)*R_k = R_i*(R_j*R_k)$
4. $(\varphi*R)*\psi = \varphi*(R*\psi)$
5. $(R*\varphi)*\psi = R*(\varphi \wedge \psi)$
6. $(R*\varphi)*\psi = (R*\psi)*\varphi$
7. $\psi*(\varphi*R) = (\psi \wedge \varphi)*R$
8. $\varphi*(\psi*R) = (\psi \wedge \varphi)*R$

9. $R_i*\varphi*R_j = (R_i*\varphi 1)*R_j = R_i*(\varphi*R_j)$

♦

Although the chaining operator is associative, it is not commutative. Swapping operators in general produces an association with different schema.

3.3 Translation to Relational Algebra

This chapter describes translation of the object algebra into relational algebra. Translation is based on mapping of the object database onto relational database management system. This translation is not dependent on particular SQL dialect. Expressions in the relational algebra are then easily translatable to the final SQL language.

3.3.1 Relational Database Schema

In comparison to above introduced object database schema, relational database schema provides fewer capabilities. It consists of individual table schemas in at least first normal form.

Definition 3.3-1 – Relational database schema

Formally, *relational-database schema* is a couple $\mathbf{D_r} = (\mathbf{T}, \mathbf{A}, \mathbf{attr})$, where

1. $\mathbf{T}$ is a finite set of relational table schemas.
2. $\mathbf{A}$ is a finite set of attributes.
3. $\mathbf{attr} : \mathbf{T} \rightarrow \text{powerset}(\mathbf{A})$ is a total mapping such that

   $\mathbf{attr}(X) \cap \mathbf{attr}(Y) = \emptyset$ if $X \neq Y$

Relational database instance is then a set of table instances.

To represent data of introduced object model in relational database we need to define correct mapping, which transforms object database schema and its instance onto their relational equivalents.

3.3.2 Object mapping

There are four different approaches of representation object database schema in the relational database. In order of decreasing number of needed tables they can be shortly described as “table per meta-attribute”, “table per class”, “table per branch” and “table per inheritance tree” mappings. First – less used – approach stores each additional attribute of the newly created class in the separate table. This situation is demonstrated on following picture.

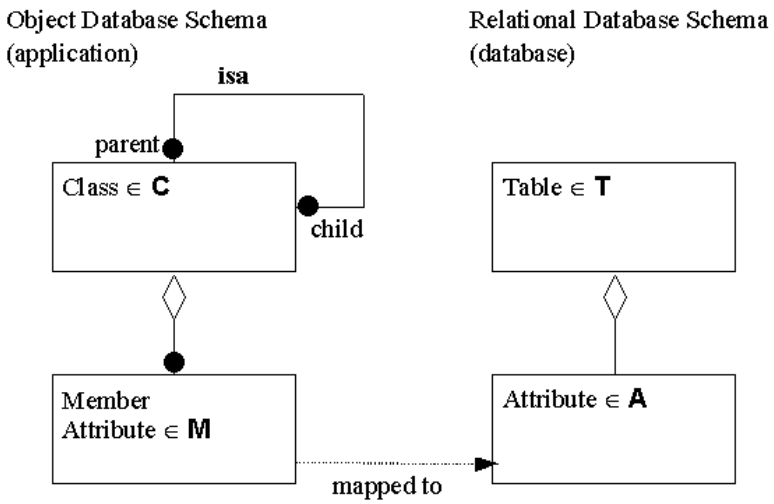


Figure 3.3-1 Table per meta-attribute mapping model

The second approach stores all member attributes of one class in exactly one relational table in the database.

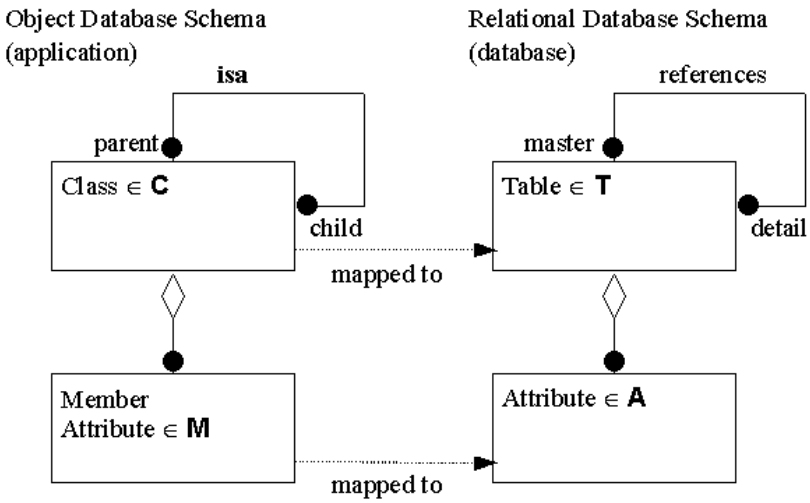


Figure 3.3-2 Table per class mapping model

In both cases, the values forming one single instance are stored in pieces spread among a set of tables. The number of tables depends on used mapping approach. In the first case the number of tables needed to store instances of class C is proportional to cardinality of **mematt**$\mathbf{r}^*$(C). The total number of needed tables equals to cardinality of set **M**. We have chosen the second possibility, where total number of needed tables decreases to size of the set **C**, storing instance of class C in number of tables equal to size of **pred***(C). The number of used tables is crucial for effectiveness of the application. The fewer tables to be joined, the quicker response we can expect from the database. Other mentioned models that map all classes laying on particular inheritance path or even all classes in one component of connectedness were not used in spite of even less number of tables. The reason of this decision is more complicated maintenance of relational model and its difficult synchronisation with the class diagram. Any local change inside particular class can have influence on more than one table.

To be able to join corresponding data fragments into complete instance, the instances must be uniquely identified. For this purpose the artificial number called „object identification“ and abbreviated as OID is usually used. Those identifications should be generated automatically by the system. Its value is then associated with each instance at the time of its creation and doesn't change

during its lifetime. To store the identification of the instance each table in the database has to contain one additional column called OID and declared as a primary key.

The mapping between arbitrary object database schema and its equivalent relational database schema can be formally defined in two steps. First step adds one additional class onto top of the inheritance graph with one attribute that holds information about the class of the instance. This additional class among others ensures that there is only one maximal element in the inheritance graph and simplifies the second step.

Definition 3.3-2 – Unified form of object database schema

Let $\mathbf{D} = (\mathbf{C}, \mathbf{M}, \mathbf{R}, \mathbf{isa}, \mathbf{mematttr}, \mathbf{left}, \mathbf{right})$ is an object–database schema. Then its *unified form* is a couple $\mathbf{D}_u = (\mathbf{C}_u, \mathbf{M}_u, \mathbf{R}, \mathbf{isa}_u, \mathbf{mematttr}_u, \mathbf{left}, \mathbf{right})$ where

1. $\mathbf{C}_u = \mathbf{C} \cup \{C_P\}$, $\mathbf{inst}(C_P) = \mathbf{O}$
2. $\mathbf{M}_u = \mathbf{M} \cup \{M_P\}$, $\mathbf{dom}(M_P) = \mathbf{C}$
3. $\mathbf{isa}_u$ extends **isa** by the rule $C_i\ \mathbf{isa}_u\ C_P \Leftrightarrow C_i \in \mathbf{max}(\mathbf{C})$
4. $\mathbf{mematttr}_u(C_P) = M_P$

♦

Translation of the unified form of object schema database with just one maximal element into its relational representation can be then done one table per class basis as defined below.

Definition 3.3-3 – Relational projection

Let $\mathbf{D} = (\mathbf{C}, \mathbf{M}, \mathbf{R}, \mathbf{isa}, \mathbf{mematttr}, \mathbf{left}, \mathbf{right})$ is an unified form of object database schema. Its *relational projection* is a relational database schema $\mathbf{D}' = (\mathbf{T}, \mathbf{A}, \mathbf{attr})$ where

1. $\mathbf{T} = \{C' \mid C \in \mathbf{C}\} \cup \{R' \mid R \in \mathbf{R}\}$
2. $\mathbf{A} = \mathbf{A_C} \cup \mathbf{A_M} \cup \mathbf{A_R}$ where
   a) $\mathbf{A_C} = \{C.OID \mid C \in \mathbf{C}\}$ with $\mathbf{dom}(C.OID) = \mathbf{N}$
   b) $\mathbf{A_M} = \{M' \mid M \in \mathbf{M}\}$ with $\mathbf{dom}(M') = \mathbf{dom}(M)$
   c) $\mathbf{A_R} = \{R.L\_OID \mid R \in \mathbf{R}\} \cup \{R.R\_OID \mid R \in \mathbf{R}\}$ with $\mathbf{dom}(C.L\_OID) = \mathbf{dom}(C.R\_OID) = \mathbf{N}$
3. $\mathbf{attr}(C') = \{C.OID\} \cup \{M' \mid M \in \mathbf{mematttr}(C)\}$
4. $\mathbf{attr}(R') = \{R.L\_OID, R.R\_OID\}$

♦

Example 3.3-1 – Equivalent relational database schema

According to the above formulas we can transform object database schema from the Example 3.2-1 to the relational database schema where

$\mathbf{T}$ = {C_P' , Person', Programmer', Leader', Project', Project_Leader', Project_Programmer'}

$\mathbf{A}$ = {
C_P.OID, C_P.M_P,
Person.OID, Person.Name, Person.Age,
Programmer.OID, Programmer.PreferredLanguage,
Leader.OID,
Project.OID, Project.Title, Project.Language,
Project_Leader.L_OID, Project_Leader.R_OID,

Project_Programmer.L_OID, Project_Programmer.R_OID
}

attr(Person')={Person.OID, Person.Name, Person.Age},
attr(Programer')={Programmer.OID, Programmer.PreferredLanguage},
attr(Leader')={Leader.OID},
attr(Project')={Project.OID, Project.Title, Project.Language}
attr(Project_Leader')=Project_Leader.L_OID, Project_Leader.R_OID,
attr(Project_Leader')=Project_Programmer.L_OID, Project_Programmer.R_OID

♦

Having the equivalent relational–database schema we can define total mappings that transform object–database instance onto its equivalent relational representation.

First, to each class C we uniquely attach corresponding class table

$\Pi_{\mathbf{C}} : \mathbf{C} \to \mathbf{T} : \Pi_{\mathbf{C}}(C) = C'$

Next, we map each of member attributes of class onto attribute in attached table.

$\Pi_{\mathbf{M}} : \mathbf{M} \to \mathbf{A_M} : \Pi_{\mathbf{M}}(M) = M'$, that $\Pi_{\mathbf{M}}(M) \in \mathbf{attr}(\Pi_{\mathbf{C}}(C)) \Leftrightarrow M \in \mathbf{mematttr}(C)$

Third, we map an object relational instance $\mathbf{I} = (\mathbf{O}, \mathbf{inst}, \mathbf{val}, \mathbf{rel})$ with the unified schema $(\mathbf{C}, \mathbf{M}, \mathbf{R}, \mathbf{isa}, \mathbf{mematttr}, \mathbf{left}, \mathbf{right})$ to its equivalent relational instance $\mathbf{I}'$ so that

$\mathbf{C}' = \{t \mid \exists o \in \mathbf{inst}(C)(t.OID = \mathbf{oid}(o) \wedge \forall M \in \mathbf{mematttr}(C)(t.M' = \mathbf{val}(o,M)))\}$

$\mathbf{R}' = \{t \mid \exists l,r(<R,l,r> \in \mathbf{rel} \wedge x.LOID = \mathbf{oid}(l) \wedge x.ROID = \mathbf{oid}(r))\}$

3.3.3 Retrieving Objects in Relational Algebra

Because object instances are stored in the relational database, we need to translate expressions of object algebra to equivalent expressions in relational algebra to retrieve them. We use usual notation for relational algebra operations

1. Operator $\sigma_p(E)$ means selection. Symbol p is a predicate
2. Operator $\pi_r(E)$ denotes projection. Symbol r represents a set of attributes
3. $\rho_X(E)$ is an operator that renames relation E to X.
4. $\rho_{A \to B}(E)$ renames attribute A of relation E to B
5. Cartesian product $E_1 \times E_2$.
6. Natural join or two relations $E_1 \otimes E_2$
7. Natural join of arbitrary count of relations $\otimes_{E \in \mathbf{E}}(E)$

Further, we use set operations

8. Union $E_1 \cup E_2$
9. Intersection $E_1 \cap E_2$
10. Set difference $E_1 \setminus E_2$

The translation of object algebra expressions on class C starts from evaluation of *extended attached table* instance $\mathbf{C}'^{*}$. Its rows contain all attributes of the class including those directly or indirectly inherited.

$\mathbf{C}'^{*} = \{t \mid \exists o \in \mathbf{inst}(C)(t.OID = \mathbf{oid}(o) \wedge \forall M \in \mathbf{mematttr}^{*}(C)(t.M^{\pi} = \mathbf{val}(o,M)))\}$

The resulting set of rows could be constructed as a natural join of all tables attached to all predecessors of the class. Shortly

$\mathbf{C}^{*} = \otimes_{X \in \mathbf{pred^*}(C)}(X')$

Extended attached table is used either for retrieving complete instance $o \in \mathbf{O}$ from the database or for evaluation of object selection.

In the first case we select one particular row of extended attached table with given OID by expression

$\sigma_{OID=o.OID}(\otimes_{X \in \mathbf{pred^*(class}(o))}(X'))$.

Object selection on class C translates to following selection on extended attached table.

$(\mathbf{C}(\varphi))' = \pi_{OID}(\sigma_{\varphi'}(\mathbf{C}'^{*})) = \pi_{OID}(\sigma_{\varphi'}(\otimes_{X \in \mathbf{pred^*}(C)}(X')))$

The final projection returns only object identifications of wanted instances. Remaining attributes of instances are not interesting at this time. Its amount differs for each instance and can contain additional attributes not included in the extended attached table. Rules for the translation of object algebra predicate to relational algebra are straightforward.

1. $(\varphi_1 \Theta \varphi_2)' = (\varphi_1' \Theta \varphi_2')$ for any predicates φ_1 and φ_2 and any operator Θ.
2. $(\neg\varphi)' = \neg(\varphi')$ for any predicate φ
3. $(M)' = \Pi_{\mathbf{M}}(M) = M'$ for any member attribute M
4. $(x)' = x$ for any constant x

Because we need only identifications of objects, it is often unnecessary to join all tables attached to all predecessors. Object selection $(\mathbf{C}(\text{true}))'$ need not to be translated to relational algebra expression $\pi_{OID}(\sigma_{true'}(\otimes_{X \in \mathbf{pred^*}(C)}(X'))) = \pi_{OID}(\otimes_{X \in \mathbf{pred^*}(C)}(X'))$. The same result we can obtain using much simpler expression $\pi_{OID}(\sigma_{true'}(C')) = \pi_{OID}(C')$.

Lemma 3.3-1

Let $C \in \mathbf{C}$ is a class, let φ is a Boolean predicate valid on class C. Then

$(\mathbf{C}(\varphi))' = \pi_{OID}(\sigma_{\varphi'}(\otimes_{X \in C \cup \{Y \in (\mathbf{pred^*}(C) \setminus C) \mid \mathbf{mematr}(Y) \cap \mathbf{mematr}(\varphi) \neq \emptyset\}}(X')))$

Proof:

According to definition of object database instance holds the implication

C_1 **isa** $C_2 \Rightarrow \mathbf{inst}(C_1) \subseteq \mathbf{inst}(C_2) \Rightarrow \pi_{OID}(C_1') \subseteq \pi_{OID}(C_2')$

From it follows that

$\pi_{OID}(C_1' \otimes C_2') = \pi_{OID}(C_1') \otimes \pi_{OID}(C_2') = \pi_{OID}(C_1')$

♦

Thus, joining a table attached to class predecessor without additional conditions on its attributes doesn't change number of rows in the relation. Only number of columns is enlarged by attributes of the attached table. In case the added columns are not necessary for the evaluation of the object selection, we can omit the table at all. On the other hand the table attached to class C itself cannot be omitted. Its omission may cause the growth of row count in the result.

3.3.4 Retrieving Associated Objects in Relational Algebra

The object database schema mapping represents each association as one relation containing a pair of associated keys named L_OID and R_OID. With respect to this notation, a natural association $C_1 \otimes C_2$ is translated to relational algebra as

1. $(C_1 \otimes C_2)' = \sigma_{L\_OID=R\_OID}(\rho_{OID \to L\_OID}(\pi_{OID}(C_1')) \times \rho_{OID \to R\_OID}(\pi_{OID}(C_2')))$

Left-sided restriction $\varphi * R$, respectively right-sided restriction $R * \varphi$ of association R between classes C_1 and C_2 can be translated into relational algebra as

2. $(\varphi * R)' = \pi_{L\_OID,R\_OID}(\sigma_{OID=L\_OID}(\sigma_{\varphi'}(\mathbf{C}_1'^*)) \times R')$,

respectively

3. $(R * \varphi)' = \pi_{L\_OID,R\_OID}(\sigma_{R\_OID=OID}(R' \times \sigma_{\varphi'}(\mathbf{C}_2'^*)))$

Operators **left** and **right** translates to

4. $(\mathbf{left}(R))' = \pi_{L\_OID}(\rho_{L\_OID \to OID}(R'))$

and

5. $(\mathbf{right}(R))' = \pi_{R\_OID}(\rho_{R\_OID \to OID}(R'))$

Most complicated is translation of chained association. It can be translated either according to bracketing or – due to the associativity of used operators – from left to right using above introduced formulas. We introduce equivalent translation that corresponds better with the further SQL evaluation.

$$
\begin{aligned}
&(\varphi_1 * R_1 * \varphi_2 * \ldots * R_{n-1} * \varphi_n)' = \\
&\pi_{A1\_2.L\_OID,An-1\_n.R\_OID}(\\
&\sigma_{\varphi'1 \wedge T1.OID=A1\_2.L\_OID \wedge A1\_2.R\_OID=T2.OID \wedge \sigma_{\varphi'2 \wedge \ldots \wedge An-1\_n.R\_OID=Tn.OID \wedge \sigma_{\varphi'n}(} \\
&\rho_{T1}(\mathbf{C}_1'^*) \times \rho_{A1\_2}(R'_1) \times \rho_{T2}(\mathbf{C}_2'^*) \times \rho_{A2\_3}(R'_2) \times \ldots \times \rho_{An-1\_n}(R'_{n-1}) \times \rho_{Tn}(\mathbf{C}_n'^*) \\
&) \\
&)
\end{aligned}
$$

This translation produces a Cartesian product of all needed relations and then joins corresponding rows of relations together. At the end it filters out all unnecessary attributes. Temporary renaming of relations in expression is necessary, because one class or one association can appear more than once in the object algebra expression.

3.4 SQL Statement Generation

Achieving of object persistency with high level of transparency suppose automatic generation of the SQL code by the system with minimal needed adjusting made by the programmer during application design and coding. Relational algebra expressions give us good fundament for SELECT statement generation. In addition to it, object instances have to be able to INSERT themselves to the database, modify its data and DELETE themselves from the database at the end of its lifetime. Algorithms used for the SQL statement generation are described in general in this chapter.

Algorithms are described for objects identified by arbitrary keys. In case that all persistent classes use simple numeric primary key OID, the SQL statement generation would be slightly simpler.

We suppose, that each class C provides following basic information about itself:

1. *ClassName*(C) ..... Name of the class, e.g. ‘Programmer’
2. *ParentClassNames*(C)..... Coma-separated list of direct predecessor names, for example ‘Person’
3. *Select*(C)..... Coma-separated list of column names, used in SELECT clause, in case of *Programmer* class it should return ‘OID, PREFERRED_LANGUAGE’
4. *From*(C) ..... Name of the attached table, used in FROM clause, e.g. ‘PROGRAMMER’
5. *Where*(C)
6. *GroupBy*(C)

7. *Having*(C) ..... Functions *Where*(), *GroupBy*() and *Having*() are defined for completeness and more flexibility of SQL generation. In case of above-described object database model they should be empty, i.e. return an empty string.

8. *PrimaryKey*(C)..... Coma-separated list of primary key columns. It must be a sub-set of *Select*(C). It typically returns value 'OID'. Its generality allows uniform maintaining of wider set of class types, not only those containing OID.

Furthermore we require from every object instance $o \in \mathbf{O}$ to offer information

9. *PrimaryKeyValue*(o) ..... Coma-separated list of values of primary key columns. Again, it allows more generality of design than simpler function *o.Oid*()

There are three main types of queries:

- Queries returning set of instances. They correspond to object selection and are used to obtain keys (object identifications) of all instances of particular class that fulfil some condition.
- Queries returning one particular instance. It is used to retrieve object instance according to its already known primary key (identification) during pointer dereference.
- Queries returning set of instances associated via stored, restricted or chained associations.

Among queries there are three operations: INSERT, UPDATE and DELETE one particular instance.

3.4.1 Transformation of recursive algorithms to iteration

Almost all algorithms working on the inheritance hierarchy graph becomes recursive once the hierarchy allows multiple inheritance and the hierarchy tree adapts to hierarchy graph. The graph must be traversed to process all predecessors of particular class every time the object should be updated in the database, refreshed in inserted into database, deleted from it and many others.

To simplify the complexity algorithms, we need to flatten the graph of predecessors of class to the list of nodes. Each class in any of those lists must follow all its predecessors. To achieve that feature, following algorithm should be executed for each class.

```
procedure build_parent_list(C)
  procedure process_class(X)
  begin
    if (not_already_processed(X))
    then
      for each Y in ParentClassNames(X) loop
        process_class(Y);
      end loop;
      C.parent_list := C.parent_list + X;
      mark_as_processed(X);
    end if;
  end;
begin
  C.parent_list := ∅;
  process_class(C);
end;
```

This algorithm that traverses the graph should be run for each class only once. Without it, the graph should be traversed in similar manner during every operation with the object instance. Advantages of the SQL generation speed-up exceed disadvantages of memory consumption overhead. Really, we need $o(|\mathbf{pred}^*(C)|)$ extra space to maintain the predecessors list. Because $|\mathbf{pred}^*(C)| <= |\mathbf{C}|$.

Wee need o($|\mathbf{C}|^2$) of space and time to build and hold this information in worse case, when inheritance hierarchy degenerates to linear list. In the average, when the inheritance graph is close to the balanced tree, we can estimate time and space overhead as proportional to o($|\mathbf{C}|\log(|\mathbf{C}|)$).

3.4.2 Queries returning set of instances

The simplest case belonging to this category is the request to retrieve all identifications of the given class C. Such query can be expressed in SQL as follows.

SELECT *PrimaryKey*(C)
FROM *From*(C)
WHERE *Where*(C)

Generally, only specific sub-set of all instances fulfilling some condition is needed. To express object selection in SQL, query must be constructed in form

SELECT *PrimaryKey*(C)
FROM *FullFrom*(C)
WHERE *FullWhere*(C) **AND** *Translate*(C,φ)

where:

i. Function *FullFrom*(C) returns comma-separated list of all tables attached to direct and indirect predecessors. Algorithm takes advantage of the linear list of predecessors to eliminate the recursion.

```
function FullFrom(C) return string
begin
  result := ‘’; separator := ‘’;
  for each X in C.parent_list loop
    result := result + separator + From(X);
    separator := ‘,’;
  end loop;
  return result;
end;
```

ii. Function *FullWhere*(C) returns conjunction of elementary joins. Each elementary join binds primary key of class C with primary key of one predecessor as shows following code.

```
function FullWhere(C) return string
begin
  result := ‘’; separator := ‘’;
  for each X in C.parent_list loop
    if X<>C then
      result :=
        result + separator
        + From(X) + ‘.’ + PrimaryKey(X) + ‘=’ + From(C) + ‘.’ + PrimaryKey(C);
      separator := ‘ and ‘
    end if;
  end loop;
  return result;
end;
```

Joining all needed tables together in this manner, i.e. every table directly with the one attached to the class C, can produce more optimizable SQL statements than its equivalent, which joins table attached to given class with the one attached to its direct predecessors. In proposed variant the structure of inheritance graph need not be taken into account. Moreover, some tables can be easily cut out from the statement, if queries are optimised according to chapter 3.3.3.

iii. Function *Translate*(C,φ) translates predicates from object algebra syntax to the SQL syntax.

There exist two specific groups of predicates in the object algebra. All classes X ∈ **C** belong to first of them. The SQL predicate *Translate*(C,X) in form

(*PrimaryKey*(C)) **IN** (**SELECT** *PrimaryKey*(X) **FROM** *From*(X))

retrieves only objects belonging to the given class (and all its descendants) and filters out all other instances.

The set **O** of all instances form then the second specific group of predicates. The expression *Translate*(C,o) where o ∈ **O** produces an SQL predicate that returns single-element set containing the object itself. Its form is

Merge('(',*PrimaryKey*(C),'=',*PrimaryKeyValue*(o) ,')',')AND('.

iv. The multi-purpose function *Merge*(Prefix,List1,SepItems,List2,Posfix,SepGroups) merges two equally long lists together. It arranges corresponding items of both lists into pairs separated by item separators. Group separators then interleave individual groups. The result is then put between given prefix and postfix.

```
function Merge(Prefix,List1,SepItems,List2,Posfix,SepGroups) return string
begin
  result := Prefix; separator := '';
  while (lists are not empty) loop
    item1 := first item of List1;
    List1 := rest of List1;
    item2 := first item of List2;
    List2 := rest of List2;
    result := result + separator + item1 + SepItems + item2;
    separator := SepGroups;
  end loop;
  result := result + PostFix;
  return result;
end;
```

According to this implementation of *Merge* function, *Translate*(C,o) generates an SQL fragment

$(pk_1=value_1)AND(pk_2=value_2)AND...\ AND(pk_k=value_k)$

3.4.3 Queries returning complete instance

Most statements in the application retrieve only object identifications. But at the latest in the moment the application wants to access data in the instance, the instance must be retrieved completely from the database including all attributes. SQL statement is similar to statement for object selection. Instead of columns, forming primary key, all columns are selected. Modifications lead to statement in form

```
SELECT   FullSelect(C)
FROM     FullFrom(C)
WHERE    FullWhere(C)
   AND   Merge('(',PrimaryKey(C),'=',PrimaryKeyValue(o) ,')',') AND ('
```

where:

i. Function *FullSelect*(C) returns comma-separated list of all columns from all tables attached to direct and indirect predecessors. Each column qualified by table name. It can be pre-computed by following pseudo-code.

```
function FullSelect(C) return string
begin
  result := ‘’; separator := ‘’;
  for each X in C.parent_list loop
    for each Y in Select(X) loop
      result := result + separator + From(X) + ’.’ + Y;
      separator := ‘,’;
    end loop
  end loop;
  return result;
end;
```

3.4.4 Queries returning associated objects

Finding of associated objects implements **left**(R) and **right**(R) operators from object algebra in SQL. Generation of SQL statements depends on fact, that associations are strictly typed. Each association “knows” what exactly classes it associates, for example workers and projects. Information about associated objects is usually stored in some relational table in form of pairs of foreign keys. Among this type of associations, some associations can be computed from more than one table, as in case of chained associations. Pairs of associated object keys are obtained using SELECT statement involving more tables in this case. Assume that each association $R \in \mathbf{R}$ can describe itself by following set of functions.

1. *lClassName*(R)
   rClassName(R) ..... Names of associated classes
2. *lPrimaryKey*(R)
   rPrimaryKey(R) ..... Names of columns forming foreign keys of associated classes
3. *From*(R) ..... Name(s) of table(s) to be used to retrieve associated data. Value R’ for $R \in \mathbf{R}$, comma-separated list of tables for chained association etc.
4. *Where*(R)..... Join condition joining tables. Empty for regular associations.

Because tables for associations are constructed, we can derive names of foreign key columns from names of primary key columns of corresponding class. To differentiate left-side foreign key from right-side one, we can add prefix “L_” to the first and “R_” to the second. Typically, the association holding table would have two columns L_OID and R_OID.

Based on the assumption, we can consider following SQL statements producing set of associated object pairs:

```
SELECT DISTINCT lPrimaryKey(R), rPrimaryKey(R)
FROM    From(R)
WHERE   Where(R)
```

From its format follow translation of both **left**(R), respectively **right**(R) operators in form

```
SELECT DISTINCT lPrimaryKey(R)
FROM    From(R)
WHERE   Where(R)
```

respectively

```
SELECT DISTINCT rPrimaryKey(R)
FROM    From(R)
WHERE   Where(R)
```

We can derive more complex associations from simple ones by modifying information that return above-introduced functions that describe it. Different modifications lead to different results. Each association operation defined in 3.2.5 can be achieved by appropriate modification.

3.4.4.1 Reversed Associations in SQL

Deriving reversed association from original, non-reversed one is simple. This operation only needs to swap left and right side of the association. Thus

lClassName(-R) := *rClassName*(R)

rClassName(-R) := *lClassName*(R)

lPrimaryKey(-R) := r*PrimaryKey*(R)

rPrimaryKey(-R) := *lPrimaryKey*(R)

Other properties of association remain intact.

3.4.4.2 Natural Associations in SQL

Associated pairs are not explicitly stored in the database, but computed from the information provided by associated classes.

lClassName($C_l \otimes C_r$) := *ClassName*(C_l)

rClassName($C_l \otimes C_r$) := *ClassName*(C_r)

lPrimaryKey($C_l \otimes C_r$) := *PrimaryKey*(C_l)

rPrimaryKey($C_l \otimes C_r$) := *PrimaryKey*(C_r)

From($C_l \otimes C_r$) := *From*(C_l),*From*(C_r)

Where($C_l \otimes C_r$) := *Merge*('(',*PrimaryKey*(C_l),'=',*PrimaryKeyValue*(C_r) ,')',') AND ('

3.4.4.3 Restricted Associations in SQL

Left restriction φ*R, respectively right restriction R*φ of the association produces sub-association of R as described in 3.2.5. This restriction keeps only such pairs of objects that fulfil the given condition. Needed modifications for the left restriction (right one is symmetrical) are

From(φ*R) := *FromAll*(*lClassName*(R)), *From*(R)

Where(φ*R) := *WhereAll*(*lClassName*(R))
AND *Merge*(
'(',PrimaryKey(*lClassName*(R)),'=',*lPrimaryKey*(R),')',')AND('
)
AND *Where*(R)

Left restriction o*R can be constructed as a special case of restricted association in more efficient manner. Only *Where*(o*R) needs to be modified according to formula

Where(o*R) := *Merge*('(',*lPrimaryKey*(R),'=',*PrimaryKeyValue*(o),')',')AND(')
AND *Where*(R)

Left restriction C*R, in addition to modified value of function *Where*(R), changes also type of left side of the association. *From*(R) need not to be modified.

Where(C*R) := (*lPrimaryKey*(R)) **IN** (**SELECT** *PrimaryKey*(C) **FROM** *From*(C))

lClassName(C*R):= *ClassName*(C)

3.4.4.4 Association Chaining in SQL

Last supported object algebra operation chains two associations R_1 and R_2 together. Resulting chained association is in general described by following characteristics.

lClassName(R_1*R_2) := *lClassName*(R_1)

rClassName(R_1*R_2) := *rClassName*(R_2)

lPrimaryKey(R_1*R_2) := *lPrimaryKey*(R_1)

rPrimaryKey(R_1*R_2) := *rPrimaryKey*(R_2)

From(R_1*R_2) := *From*(R_1), *From*(R_2)

Where(R_1*R_2) := *Where*(R_1)
AND *Merge*('(',*rPrimaryKey*(R_1),'=',*lPrimaryKey*(R_2),')',')AND(')
AND *Where*(R_2)

3.4.4.5 Aliases in Generated SQL code

Translation of object algebra queries into the SQL language is slightly more difficult than above described statements. This fact was omitted in presented statements to simplify its notation. The difficulties are caused by the possibility of multiple independent occurrences of some tables in the translated query. Chaining of two identical associations, or associations based on the same table will, without caution, produce wrong statement. Let suppose chained association

-Project_Programmer * Project_Programmer

that returns couples of co-programmers on some common project (see Figure 3.2-2). As described above, the association would be translated to SQL statement

SELECT DISTINCT *rPrimaryKey*(Project_Programmer), *rPrimaryKey*(Project_Programmer)
FROM *From*(Project_Programmer), From(Project_Programmer)
WHERE *rPrimaryKey*(Project_Programmer)=*rPrimaryKey*(Project_Programmer)

In pure SQL, the statement would have form

SELECT DISTINCT Project_Programmer.R_OID, Project_Programmer.R_OID
FROM Project_Programmer, Project_Programmer
WHERE Project_Programmer.L_OID=Project_Programmer.L_OID

which is apparently wrong. The correct statement needs to distinguish both occurrences of the table Project_Programmer. To achieve this goal, we need to produce statement in form similar to

SELECT DISTINCT R$_1.R_OID, R$_2.R_OID
FROM Project_Programmer **AS** R$_1, Project_Programmer **AS** R$_2
WHERE R$_1.L_OID= R$_2.L_OID

To adapt the SQL statement, tables should have defined temporary aliases. These aliases then should be used in translated queries instead of original table names. One of possibilities of excluding duplicities in aliases is to base the alias of the table on the order of appearance of the table name in the association as shown above. Aliases of tables that store associations $R \notin \mathbf{R}$ are generated in format "R$_n". Table attached to class $C \notin \mathbf{C}$ should have alias in format "T$_n_m" with two indexes for better maintenance. First index is common to all tables from *AllFrom*(C) list, the second one express order in that list.

Thanks to that assigning of aliases,

From($\varphi_{Programmer}$ * -Project_Programmer * Project_Leader * φ_{Leader})

would look like

Person **AS** T$_1_1,
Programmer **AS** T$_1_2,
Project_Programmer **AS** T$_2,
Project_Leader **AS** T$_3,
Person **AS** T$_4_1,
Programmer **AS** T$_4_2,
Leader **AS** T$_4_3

instead of

Person,
Programmer,
Project_Programmer,
Project_Leader,
Person,
Programmer,
Leader

3.4.5 Actualisation Of The Database

Relational algebra describes only data retrieval from the database and doesn't cover statements that manipulate data in the database. Generation of those SQL statements is described here. We need generate five different types of such statements – object insertion, update and deletion for individual classes and also insertion into and deletion from associations.

3.4.5.1 Object Insertion

With respect to database schema mapping is the insert split to multiple steps. Each step manipulates with member attributes of one class and modifies only its attached table. Class by class data insertion procedure processes classes from pred*(C) set one by one in order from predecessors to successors. The order of class processing is important due to foreign key references between attached tables.

```
procedure InsertInstance(C,o)
begin
  for each X in C.parent_list loop
    InsertFragment(X,o);
  end loop;
end;
```

Each partial insert builds and executes SQL statement in form

INSERT INTO *From*(X)
VALUES (*AddPrefix*(*Select*(X),':'))

Function *AddPrefix* simply adds given prefix, at this time a colon, in front of each element of the list. Adding colon to the name of the column makes a placeholder – an SQL parameter that is later bound to the correct value of the member attribute. For more detailed description of this process see chapter 4.14.

3.4.5.2 Object Actualisation

The actualisation of data works similar. Tables attached to individual classes are updated one by one. The order of actualisation is not critical, but we keep it the same as in previous case.

```
procedure UpdateInstance(C,o)
begin
  for each X in C.parent_list loop
    UpdateFragment(X,o);
  end loop;
end;
```

Each update is done by SQL statement

UPDATE *From*(X) **SET**
Merge('',*Select*(X),'=',*AddPrefix*(*Select*(X),':') ,'','','')
WHERE *Merge*('(',*PrimaryKey*(X),'=', *PrimaryKeyValue*(o),')','')AND(')

3.4.5.3 Object Deletion

The deletion of particular instance can be accomplished by removing corresponding row from the table attached to the root of the inheritance hierarchy. The rest of work completes the relational database, assuming that there are defined foreign keys on all descendant tables. Needed SQL statement then has a form

DELETE FROM *From*(*RootClassName*(C))
WHERE *Merge*('(',*PrimaryKey*(*RootClassName*(C)),'=', *PrimaryKeyValue*(o),')','')AND('')

The root class name is the first one from the list of all parent classes.

3.4.5.4 Associated Pair Manipulation

If it is possible to modify the content of the association, i.e. it is one of associations defined in the object database schema or its reversed variants, the SQL statement that inserts new item into, respectively deletes one item from the table can be composed in form

INSERT INTO *From*(R)
VALUES (*PrimaryKeyValue*(o_1) **,** *PrimaryKeyValue*(o_2) **)**

respecively

DELETE FROM *From*(R)
WHERE *Merge*('(',*lPrimaryKey*(R),'=',*PrimaryKeyValue*(o_1),')','')AND('')
AND *Merge*('(',*rPrimaryKey*(R),'=',*PrimaryKeyValue*(o_2) ,')','')AND('')

This implementation can differ for associations with other than many-to-many cardinality. Associations with less cardinality can be imbedded directly into table attached to one of associated classes instead of forming additional table. Associations with one-to-many cardinality add needed columns forming foreign key to the table attached to the right class. In this case we need UPDATE statements instead of INSERT and DELETE. The statement

UPDATE *From*(R) **SET**
Merge('',*lPrimaryKey*(R),'=',*PrimaryKeyValue*(o_1),':') ,'','','')
WHERE *Merge*('(',*rPrimaryKey*(X),'=', *PrimaryKeyValue*(o_1),')','')AND('')

associates the object instance on the left side with the one on the right side. Deletion of the association is similar, only values of left primary key are replaced by NULL values.

4 Library Design

The goal of this chapter is to describe the **POLiTe**[3] library that adopts the object algebra. This library provides transparent object persistency built on relational database written in C++ language. The structure of the library, represented on following figure is based on requirements described in 2.4 and on features provided by object algebra introduced in 3.2.

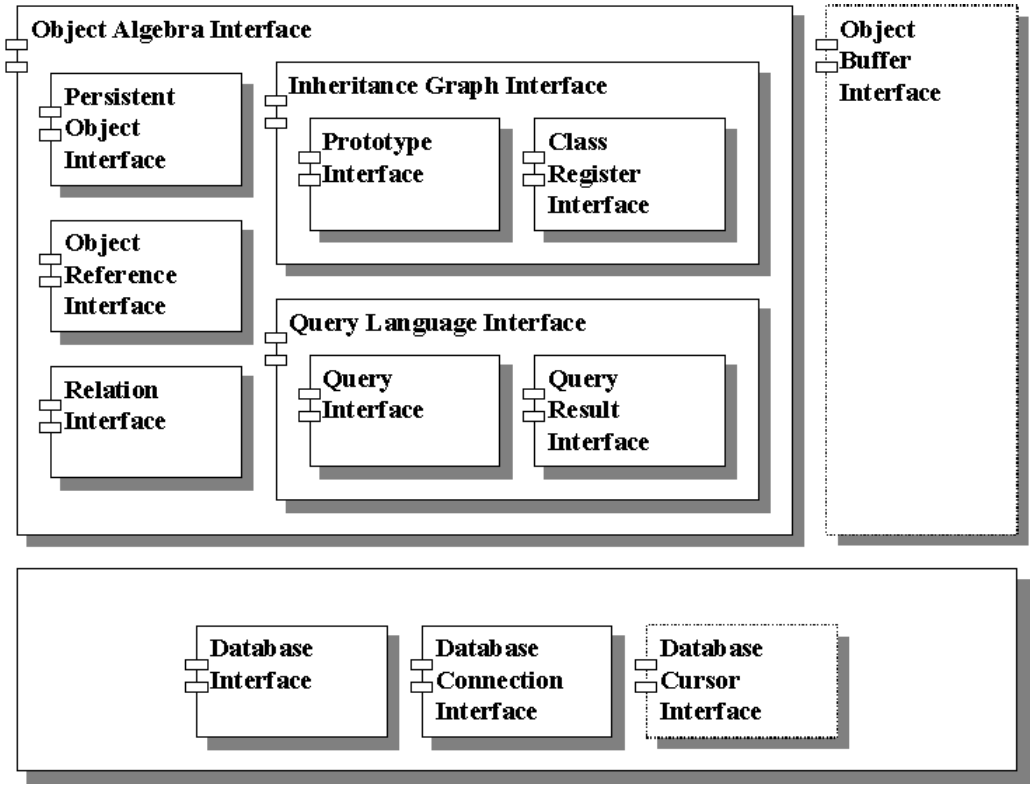


Figure 3.4-1 The internal structure of the library

This chapter defines the object model of the library. Described object model specifies the set of supported operations and its semantics.

The strict interpretation of object transparency should mean that the programmer doesn't recognise the fact that its legacy application stops to run as usual C++ application and start to use object persistency. Achievement of this goal in all its aspects is not possible due to many differences in the programming techniques used in database applications on one side and in object–oriented programming. Among advantages we can mention the ability of querying objects according its inner values as well as simply manipulate huge amount of data. In contrary, the application is expected to restrict number of allowed constructs and also to run more slowly with limited set of allowed data structures. This chapter describes such differences, and suggests solutions, which minimise impacts of using the object persistency on one side and allow utilising of advanced features of the relational database servers.

[3] **P**ersistent **O**bject **Li**brary **Te**st

4.1 Supported types

4.1.1 Persistent capable classes

The syntax and semantics of persistent types is taken from the C++ language. The library manages C++ classes that implement the *Persistent Object Interface*. As described in 4.2, there is a family of classes derived directly or indirectly from the base class named *Object* in the proposed library. There are, however, some restrictions on the complexity of objects. Each of persistent classes can have arbitrary number of direct predecessor, but no more than one of those predefined. The count of direct successors of class is also not limited. Though there are no explicit limits on the maximum depth of the hierarchy tree, the technical limitations of used relational database can restrict it[4]. Object management algorithms used in the library (see chapter 4.9) limits its usage only for dynamically allocated instances.

4.1.2 Atomic attributes

Allowed atomic member attributes of persistent classes are all attributes declared as non-structured C++ type. The library should handle attributes of those atomic types, which can be stored in the column of the database table.

- All variants of type *int* and their sub-types (*short int*, *int*, *unsigned int*, *long int*, *bool* etc.).
- Floating-point numeric types (*float* and *double*).
- All variants of type *char* (char and unsigned char).
- String types *char[n]* and *char** (zero terminated string). Unfortunately, the maximum length of CHAR and VARIANT CHAR data types in the databases are limited and the maximum allowed length depends on the used database. This restriction must be followed also by the application.

4.1.3 Structured attributes

It is possible to create persistent classes containing structured attributes, considering the following rules of type constructing are followed:

- Structures containing members of supported types are supported.
- Arrays of supported types are theoretically supported, but not recommended[5].

The persistent object that directly or indirectly contains another persistent object is not supported. This situation must be solved either by an association or by a reference.

4.1.4 Pointers

The library cannot handle object attributes declared as pointer to other variables. This restriction is forced by the fact, that pointers becomes invalid once the object is stored to the database and retrieved later by different process. Instead of usual pointers, the library introduces references and associations that can be used instead. Detailed description of those concepts can be found below in chapters 4.3 and 4.13.

---

[4] For example count of tables allowed in one SQL SELECT statement can be limited.

[5] Due to limitation of relational (SQL-92 Entry level compliant) databases, where all tables must be in 1-st normal form, all structured types must be mapped to the flat database table. Each element of the array must be then mapped to one column. Thus, creating of two associated classes is preferred over the class, which contains an inner array.

4.1.5 Methods

The library doesn't handle the code of object methods. The methods survive the periods of time between runs in the code of the C++ program. The persistency of objects means here the persistency of data, not persistency of method's code. We don't want to store the code in the database. Reasons for that choice are:

- The storage of methods on the server side is too much dependent on the used RDBMS. Changing the database would mean to completely rewrite the code of methods in the whole application.
- Though the latest versions of many relational databases allow access from the database engine to the code written as shared library, the code is executed on the server side. The load of applications cannot be balanced between tiers of the application.
- Client application C++ code cannot be stored in the database, then loaded into client process memory and executed.

4.2 Persistent classes

The hierarchy of persistent classes comes from needs of the object algebra and its translation described in chapter 3.3. Object algebra works with object database model and maps it to relational database. The library takes into account that real applications would need to manipulate also with legacy relational tables and even with data aggregated from the database using general SELECT statement.

Those categories of data manipulation differ in many details. Therefore, persistent classes can be in the application derived from one of four generic parents that implement needed *Persistent Object Interface*

```
abstract class Object;
abstract class ImmutableObject : public Object;
abstract class DatabaseObject : public ImmutableObject;
abstract class PersistentObject : public DatabaseObject;
```

defined in the library.

On OID based objects derived from class *PersistentObject* support multiple inheritance and can be fully synchronised with the database.

Synchronising must be possible also for descendants of *DatabaseObject* class. This *PersistentObject* predecessor implements manipulation with rows of relational tables. It takes into account that tables are not derived one from another, but can have arbitrary primary keys with more than one column.

Data from the database can be available also in form of non-updateable view or trough arbitrary SELECT statement. As an example can serve the SQL SELECT statement

```
SELECT DISTINCT P.Title, COUNT(PP.R_OID) AS NrOfSolvers
FROM    Project P, Project_Programmer PP
WHERE   Project_Programmer.L_OID=Project.OID
GROUP BY P.Title
```

that provides number of programmers involved in particular project. In this case produced data can be read from the database, but cannot be changed. More precisely the changed data cannot be stored back in the database. Class *ImmutableObject* from which the *PersistentObject* class is derived suppose that the view or SELECT statement produces unique rows. Owing to that assumption instances of *ImmutableObject* descendants can be repeatedly read from the view or SELECT statement using correct set of columns as primary key. The fact that those instances are identifiable allows them to be associated with other instances or referenced from them. Upon the SELECT statement above we can

build new class *Statistics* with two member-attributes *Title* and *NrOfSolvers*. The altered database schema including built-in abstract classes will then look as follows.

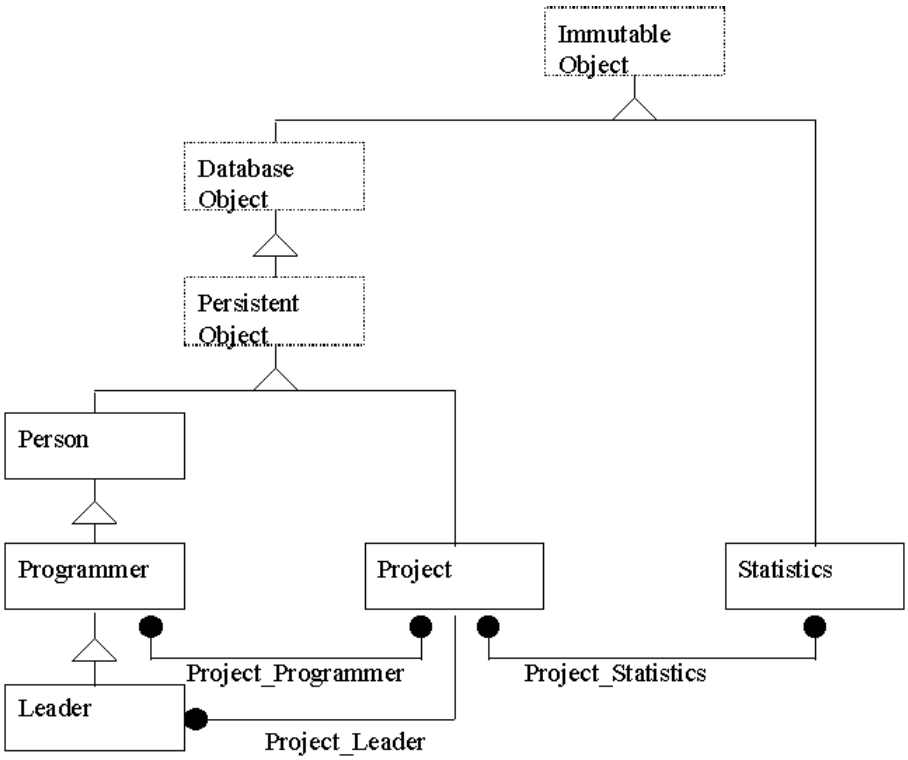


Figure 4.2-1 Object model with *ImmutableObject* descendant

In the most general case the view or SELECT statement can return non-unique set of rows. Particular row then cannot be identified and re-loaded from the database at all. Manipulation with such type of data provides the *Object* class, predecessor of all three other classes.

Differences are shown in following table.

Feature	Object	Immutable Object	Database Object	Persistent Object
Data source	Any SELECT statement.	Identifiable rows that can be repeatedly fetched.	Identifiable rows from table or updateable view.	Specially constructed tables with artificial keys
Query supported	✓	✓	✓	✓
Identifiable instances	×	✓	✓	✓
Update supported	×	×	✓	✓
Inheritance supported	×	×	×	✓

4.3 Object references

The operational memory is in most cases much smaller than the capacity of the database. On the other hand the data stored in the database need not to be in the memory during the whole application run-time. Because the application shouldn't be worry about location of data, we need mechanism allowing access to the data regardless the location. We have chosen solution based on indirect access

to dynamically allocated persistent objects via specially designed object references. Object reference is the abstraction of the object address. It identifies the persistent instance not by address in the memory, but by values of their primary key columns and thus is independent on the fact, if the transient copy of the object is stored in the local client memory or not.

Corresponding reference type is automatically derived from common predecessor *RefBase* for each class C starting with the *Object* class using template

template <class C> class Ref : public RefBase

Those references play the role very similar to role of regular C++ pointers. With respect to this similarity object references overloaded both operators of dereference

C &Ref<C>::operator*()

and

C *Ref<C>::operator->()

Both operators, together with overloaded operator of address acquisition

class RefBase Object::operator&() const

on *Object* class hide object persistency and possible absence of object instance in the memory. As in the case of standard pointers, the database pointers are comparable. Due to not supported arrays of persistent objects, only both basic operators of equality and inequality

bool operator RefBase::==(const RefBase &) const
bool operator RefBase::!=(const RefBase &) const

have to be defined while other comparisons as "less or equal to", "greater than" etc. don't. Assignments are allowed though copy constructor and copy operator

Ref<C>::Ref(const RefBase & const)

and assignment operator

Ref<C> &Ref<C>::operator=(const RefBase & const)

The empty constant reference DBNULL of the Ref<Object> class that points to none object must be defined inside the library. Its role is the same as the role of NULL pointer in the C++ language. Hierarchy of references corresponding to Figure 4.2-1 is shown on following figure.

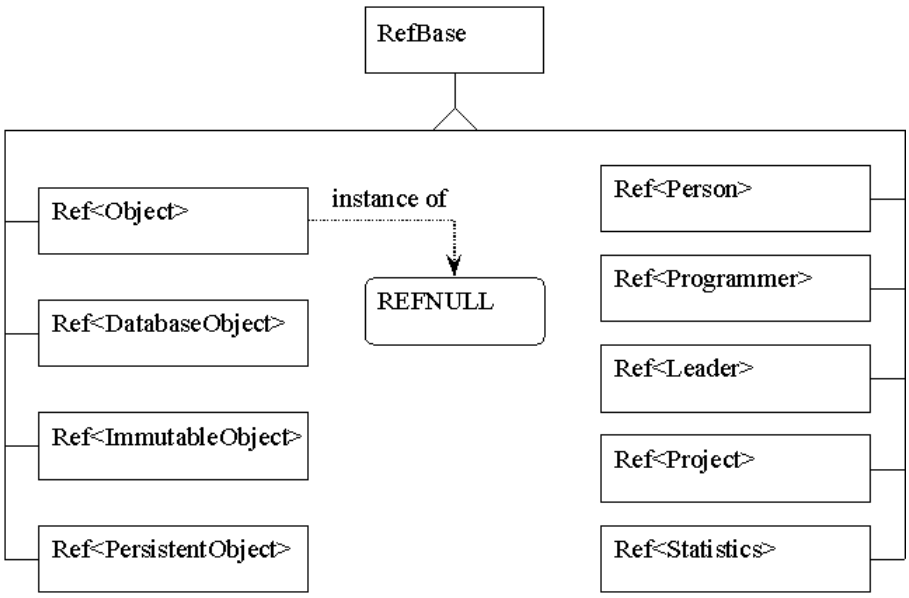


Figure 4.3-1 Object reference Class Diagram

4.4 Persistent class prototypes

The translation of object algebra to SQL, as described in chapter 3.4, is based on the knowledge of internal structure of classes and their mappings. To successfully generate SQL statements we need to know name of attached table, names of columns in it and so on. Apart from other things each class must also "know" all its direct predecessors.

The application must have this information at disposal at any time, even if no instance of particular class exists. To make needed information available, the library introduces concept of prototype classes and prototype instances.

For each class C defined in the application, starting with the class *Object*, the library creates prototype template class

template <class C> class Proto : public ProtoBase;

that implements the *Prototype Interface*. The application must declare exactly one global static instance of that prototype class. Prototype instances are defined even for abstract classes. They are named by name of the class itself with the suffix "*_class*". The adherence of this naming convention simplifies work with them and makes program code more readable. There are four prototypes defined inside the library:

class Proto<Object> Object_class
class Proto<ImmutableObject> ImmutableObject_class
class Proto<PersistentObject> PersistentObject_class

and

class Proto<DatabaseObject> DatabaseObject_class

Prototype instances reproduce the object model, respectively the (**C**,**isa**) inheritance graph in the application memory. Besides this graph, they form an associative array of prototypes accessible through the class register instance *Class*, defined as

class ClassRegister Class

Thanks to operator

class ProtoBase *ClassRegister::operator[](const char*) const

the associative array can be indexed by the class name. Expression `Class["Leader"]` then looks through the object model and returns address of *Leader_class* prototype.

As (**C**,**isa**) specialisation graph forms (not total) ordering on set **C**, the prototypes can be ordered using complete set of comparison operators.

bool *ProtoBase::operator==(const class ProtoBase &) const
bool *ProtoBase::operator!=(const class ProtoBase &) const
bool *ProtoBase::operator<(const class ProtoBase &) const
bool *ProtoBase::operator<=(const class ProtoBase &) const
bool *ProtoBase::operator>=(const class ProtoBase &) const
bool *ProtoBase::operator>(const class ProtoBase &) const

It holds that Class["X"]<=Class["Y"] $\Leftrightarrow$ X **isa*** Y. Other operators are self-explanatory. One part of Prototype Interface provides also methods corresponding with those described in 3.4. Methods

virtual const char *ProtoBase::RootClassName()
virtual const char *ProtoBase::ClassName()
virtual const char *ProtoBase::ParentClassNames()
virtual const char *ProtoBase::Select()
virtual const char *ProtoBase::Into()
virtual const char *ProtoBase::From()

virtual const char *ProtoBase::Where()
virtual const char *ProtoBase::GroupBy()
virtual const char *ProtoBase::Having()
virtual const char *ProtoBase::PrimaryKey()

describe the class itself, meanwhile methods

virtual class ProtoBase *ProtoBase::RootPrototype()
virtual int ProtoBase::ParentPrototypeCount()
virtual class ProtoBase *ProtoBase::ParentPrototype(const int)

allow traversing the inheritance graph.

Next part of *Prototype Interface* combines information about own class with information about all predecessors and produce full information about particular class through set of additional methods

virtual const char *ProtoBase::FullParentClassNames()
virtual const char *ProtoBase::FullSelect()
virtual const char *ProtoBase::FullFrom()
virtual const char *ProtoBase::FullWhere()
virtual int ProtoBase::FullParentPrototypeCount()

and

virtual class ProtoBase *ProtoBase::FullParentPrototype(const int)

Every instance of persistent object and every reference to persistent object implement the method

virtual class ProtoBase *RefObj::Prototype() const

that provides the address of instance prototype, respectively referenced object prototype's address and must be correctly redefined on every class derived directly or indirectly on from class *Object*. Section of prototype hierarchy corresponding to the class definition on the Figure 4.2-1 is shown on Figure 4.4-1. Note that the prototype *Statistics_class* has no parent. The inheritance graph starts with the *PersistentObject* class. All other classes are self-standing nodes.

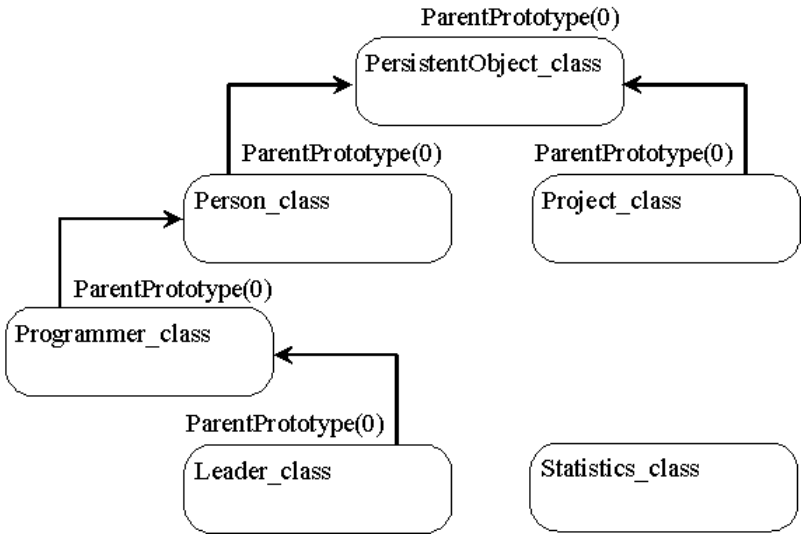


Figure 4.4-1 Object prototype hierarchy

4.5 Object Lifetimes

The lifetime of an object determines the way of memory allocation for the object instances. Library utilises both transient and persistent instances of objects.

Transient instance of the object is that one, of which lifetime doesn't exceed the lifetime of the running process that created them. To this category belongs all classes defined in the library and application except those derived from *Object* class. The transient object instances can be used as usual without limitation. In contrary, persistent instances can be allocated only dynamically on the heap. Statically allocated persistent objects are not supported. Dynamic allocation, together with database pointers allows the library to take over the memory management of all persistent objects.

Descendants of *DatabaseObject* class can be really persistent. Their instances can be stored in the database and updated synchronously with changes in memory. We distinguish four basic states of persistent instances, as shown on Figure 4.5-1.

State A: Each instance of persistent class is firstly created by the application as a standard transient instance. The data inside the instance are not stored in the database at this time. Instances in this phase of the lifecycle are irrecoverably destroyed at the end of a program execution. This state allows the application to fill-in all attributes of the instance in the memory before the instance becomes persistent and thus without any delay. It is also possible to intentionally create instances, which never become persistent in the application.

State B: At this moment the data are stored primarily in the database and the existing object instance in the operating memory is considered a cashed copy of the instance. This copy can be removed from the memory without destroying the original stored in the database.

State C: This state represents the persistent object stored in the database without a copy of its data in the application memory. In this state the instance survives the periods of the application inactivity.

State D: In the last described state can be found instances having local copy located and locked in the memory. Until unlocking, the instance is located at the fixed memory address and the application can access the object using standard pointers bypassing object reference mechanism. The object can be locked in memory more times concurrently. All locks must be unlocked before instance switches to state **B**.

Transitions between states implement a group of methods defined on *Object*, *ProtoBase* and *RefBase* classes. Besides above-mentioned operators of deference defined on *Ref*<*C*> class to this group belongs also method

virtual class Object *ProtoBase::New() const

that creates appropriate instance according the prototype on which is invoked. It is also possible to use standard *new* operator with the same result. The rest of transitions and manipulation with object instances manage interface methods defined on class *RefObj*, the common predecessor of both *Object* and *RefBase* classes.

class RefBase RefObj::BePersistent()
bool RefObj::Refresh()
bool RefObj::Free()
bool RefObj::Delete()
bool RefObj::Update()
class Object *RefObj::MemoryLock()
bool RefObj::MemoryUnlock()
unsigned int RefObj::MemoryLocked()

and

bool RefObj::RemoveAllMemoryLocks()

Method *BePersistent* inserts object into the database and switch it to the state B. The method *Refresh* simply re-reads data from the database and refreshes the state of the copy located in the application memory. Method *Free* de-allocates the memory copy of the object preserving the object in the database. To remove object completely, the method *Delete* should be used. Whenever the copy of an object instance is to be freed, the system checks the dirty flag to decide whether to update database before it, or not. The dirty flag is set by method

bool RefObj::MarkAsDirty()

and checked by method

bool RefObj::IsDirty()

The former of them should be used in every access method of object that changes the instance. The system reset the flag automatically after synchronising object with the database either using *Update* or *Refresh* method. The object can be locked at the current position in the memory using the *MemoryLock* method. This method provides memory pointer to the object that can be used until the object is unlocked by the *MemoryUnlock* method. It is possible to lock object more times simultaneously. The object is not released from the memory until all locks are unlocked. The number of locks can be checked at any moment by the method *MemoryLocked*. Locks can be removed from the object instance one by one, or they can be removed at once by the method *RemoveAllMemoryLocks*.

The current state of the object can be checked by set of methods, that besides already mentioned method

unsigned int RefObj::MemoryLocked()

contains methods

bool RefObj::IsTransient()
bool RefObj::IsPersistent()
class Object *RefObj::IsInMemory()

States of instances can be determined according to result of those methods as shown in following table.

State	IsTransient()	IsPersistent()	IsInMemory()	MemoryLocked()
A	true	false		
B	false	true	<> NULL	0
C	false	true	NULL	
D	false	true	<> NULL	> 0

The complete state transition diagram for *DatabaseObject* and *PersistentObject* descendants is shown below.

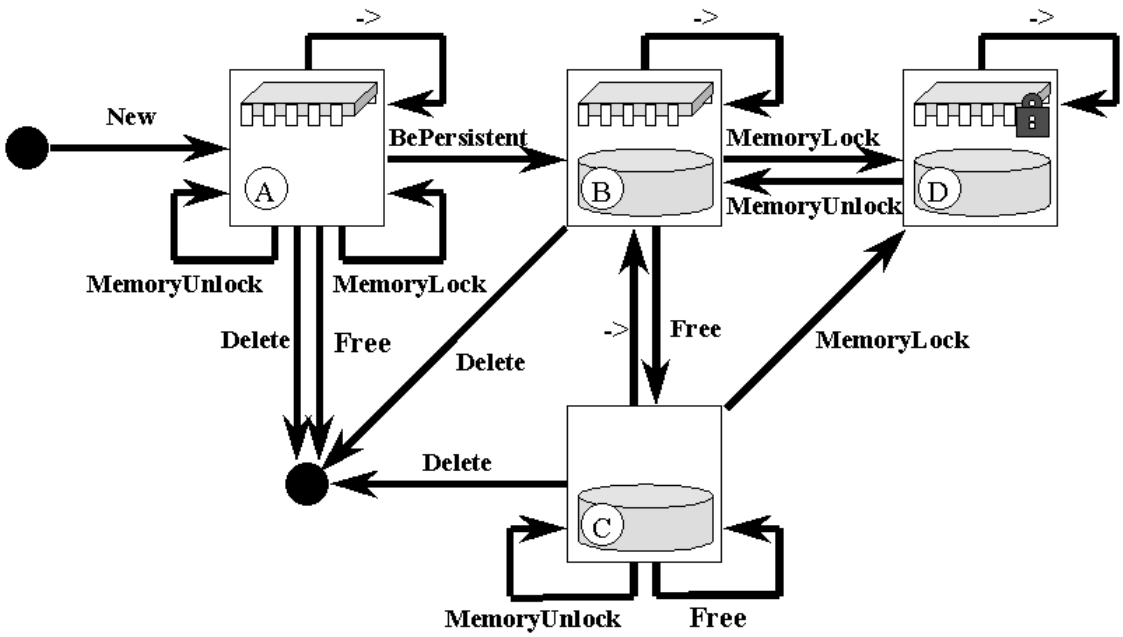


Figure 4.5-1 State diagram of *DatabaseObject* and *PersistentObject* descendants

Common interface of object instances and their references brings many advantages. To accomplish some operations, the instance need not even be loaded in the memory. For example the reference declared as

```
Ref<Programmer> refProgrammer;
```

allows deleting of the object by invocation of the statement

```
refProgrammer.Delete();
```

with the same effect as statement

```
refProgrammer->Delete();
```

In the application defined direct descendants of Object class cannot make persistent instances. They are designed to represent resulting rows of arbitrary SELECT statement. Data inside their instances cannot be synchronised with the database. Data inside them can be computed or aggregated form arbitrary number of tables. Though *Object* successors share the same interface for state transition, their state diagram is much simplified. The state diagram (see Figure 4.5-2) corresponds to diagram of usual transient object and has only one state. Differences flow from change in the *BePersistent* method.

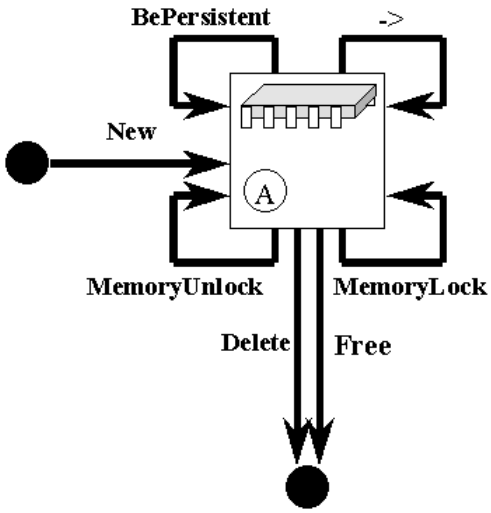


Figure 4.5-2 State diagram of *Object* descendants

Instances of objects derived from *ImmutableObject* class strictly distinguish between instances created by the application and those loaded from the database. Instances cannot be stored nor deleted from the database.

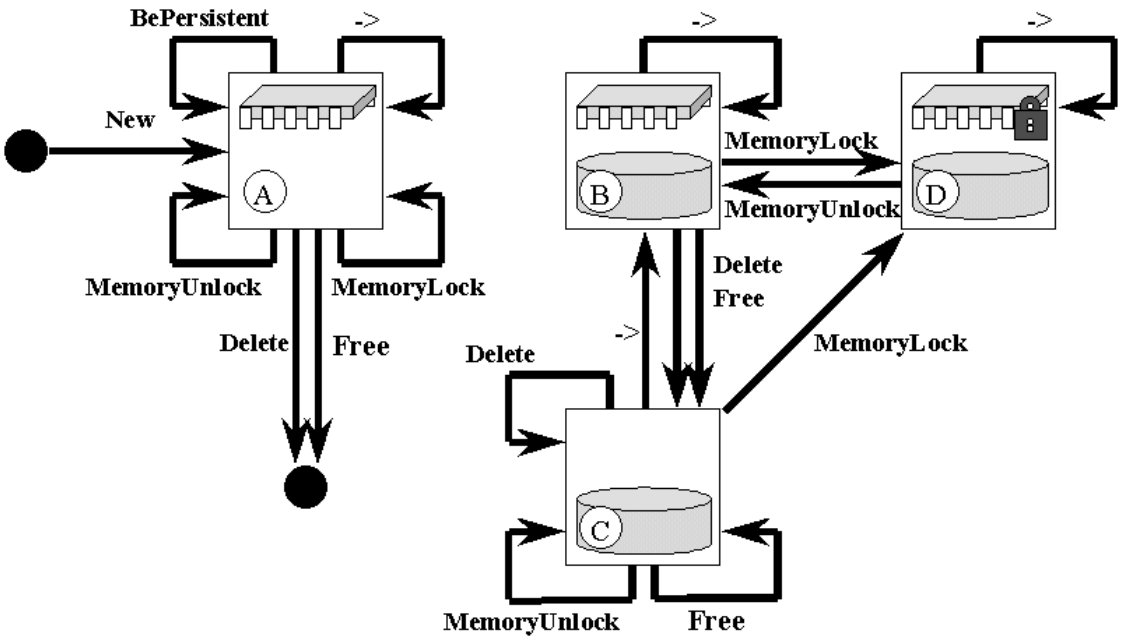


Figure 4.5-3 State diagram of *ImmutableObject* descendants

4.6 Database

The application in C++, running upon **POLiTe** library may communicate with one or more logical databases. Each of them represents one physical database. One of the basic features is the independence of the application on the RDBMS provider. The application written for one particular RDBMS family should be easily portable to another one. Thus, the library must hide the differences in the low-level communication protocols used for the client-server communication as well as differences in SQL dialects recognised at the server side. To overcome differences between database systems from different providers, the library contains two publicly visible abstract classes

class Database

class Connection

and additional one – from the outside invisible – class

class Cursor

These classes represent the abstraction of the used database. Each logical database is an instance of the concrete *Database* subclass. Existing concrete subclass

class OracleDatabase : public Database

implements the *Database Interface* as needed for RDBMS Oracle. New instances of type OracleDatabase are created using constructor

OracleDatabase::OracleDatabase(const char * = NULL)

Optional parameter denotes the name of the physical Oracle database according to Oracle naming conventions. The name of the database can be supplied later using method

virtual bool Database::Assign(const char * const = NULL)

Application can create arbitrary number of concurrent connections to particular database. Communication with the database ever goes through one of those connections. As in case of databases, individual connections are represented by instances. Common predecessor named Connection defines the *Database Connection Interface*, implemented by subclasses for different databases.

Application connects to the database using method

virtual class Connection *Database::Connect(
const char * const username,
const char * const password
)

and disconnects using

virtual bool *Connection::Disconnect()

Connections allow also limited dynamic SQL statement execution. The application can send and directly execute any DML statement, which needs not any parameter, in any connection. For this purpose is defined method

virtual bool Connection::Sql(const char *const Statement)

and equivalent operator

virtual Connection &operator << (const char * const Statement)

Example 4.6-1 – Databases, connections and dynamic SQL execution

```
class OracleDatabase SampleDatabase("");
// default oracle database at this machine
Connection *DbCon;
DbCon = SampleDatabase.Connect("user_name","password");
// connected to the database
(*DbCon)<<"UPDATE PERSON SET AGE=AGE+1";
// all ages are raised by 1
DbCon->Disconnect();
// disconnected from the database
```

♦

4.7 Multi-user data sharing

As the objects can temporarily exist in both persistent store (RDBMS) and the operational memory (see diagrams in chapter 4.5), problem with keeping all copies of objects consistent arises.

Maintaining perfect consistency is especially painful, when programs runs more times in parallel on different computers and can communicate only using slow (in comparison to the speed of the operational memory) communication medium, as the RDBMS is. The solution is to accept less than perfect consistency as the price for better performance. We introduce flexible solution allowing trade off overall application performance for better consistency and vice versa. Each consistency model specifies set of rules, which are needed to obey by the programs to ensure correct behaviour of the shared data storage. If the application violates the given rules, the correctness of the results is not guarantied.

4.7.1 Consistency models

4.7.1.1 Strict consistency

Strict consistency is the strictest possible consistency model. This model is based on the consistency achieved in single-processor operating system, where all instructions of all processes can be easily ordered in time. The strict consistency model can be expressed by the following rule.

Each operation READ on the shared object returns the value, stored to the object by the most recent WRITE operation.

The definition is natural, but assumes the existence of the absolute global time to define the term "most recent *WRITE operation*". Uni–processors achieve strict consistency and so many programmers automatically assume it. When the object storage is strictly consistent, all writes are immediately visible to all processes.

4.7.1.2 Sequential consistency

While strict consistency is the ideal model, it is mostly impossible to guarantee it in distributed computational environment. Sequential consistency is a slightly weaker model, where the storage space satisfies following condition:

The result of any execution is the same, as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This definition says, that all processes running in parallel must see the same sequence of memory accesses, Moreover, the sequence of instructions from one particular processor must be ordered according the order in this process. Fulfilling of those conditions is more easily achievable in the client server environment, as the central server can order all incoming requests.

4.7.1.3 Causal consistency

The causal consistency model represents further weakening of the sequential consistency model. Meanwhile the previous model required the same order of all operations on all computer nodes, the causal consistency model requires the same order only for potentially *causally related* couples of operations. If operation B is caused or influenced by the earlier event A, causality requires, that all processes see A first, then B. Operations which are not potentially causally related are concurrent and can appear on different nodes in different order.

4.7.1.4 Transactional consistency

The transactional consistency is naturally achieved by the RDBMS, and thus most important for us. This model of consistency is quite different than previously discussed models.

It is based on the mechanism of transactions. Transaction is the sequence of operations executed by one process (by one session in the database terminology), which satisfies four basic conditions abbreviated as ACID: *atomicity*, *consistency*, *isolation* or *independence* and *durability*.

- *Atomicity* requires, that all operations inside the transaction are executed either all, or none. If some of operations fail, the transaction is rolled back and all effects caused by the transaction are removed.
- *Consistency* assures consistency of data at the end of the transaction, even the consistency of data can be violated during the transaction.
- *Independence* guaranties that the results of operations of one running transaction are independent on other concurrently running transactions.
- *Durability* of transactions means that after the committing of the transaction all changes made by it becomes permanent and visible to other transactions. The changes should preserve even in case of software or hardware failure.

4.7.2 Transaction model

The transactional mechanism used by RDBMS cannot and shouldn't be completely hidden to the programmer of the application. The only possibility of its hiding is to permanently use the database auto-commit feature, which significantly decrease the application performance.

Transactional model allows application or thread within the application open more simultaneous connections to the same logical or physical database. In case of multiple parallel connections to one physical database, the data maintenance should be indistinguishable from the situation when there are more independent processes running concurrently on the same database. All three equivalent configurations of two connections working concurrently on the same physical database are shown on following figure.

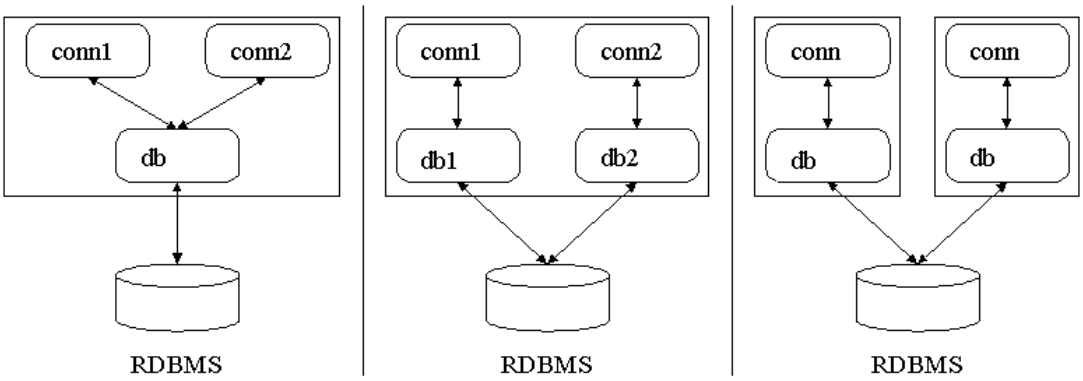


Figure 4.7-1 Concurrent connections on one database

Figure on the left shows two concurrent connections within one application those share the same logical database. Middle section shows the same connections with its own logical databases assigned to common physical database. Third figure shows two concurrent applications. Proposed transactional model assumes that transactions start automatically. At the end of the transaction, application can issue one of methods

```
virtual bool Connection::Commit();
virtual bool Connection::Rollback();
virtual bool Connection::Disconnect();
virtual bool Connection::Abort();
```

that either commit the transaction or roll it back. The *Abort* method rolls back running transaction and then disconnects the connection from the database. Method *Disconnect* described in 4.6 commits last transaction automatically.

Most relational databases allow working in so-called auto-commit mode. When this mode is active, transactions are inactive and each statement that change the state of the database is immediately committed. This mode can be set for given session through method ###

virtual bool Connection::Autocommit(bool);

Example 4.7-1 – Transaction control

The simple application can connect at the start of process. At the end it can either commit, or roll back the transaction according to possible error in the code execution.

```
int main()
{
      int error = 0;
      OracleDatabase SampleDatabase("personal");
      DatabaseConnection * DbCon;
      DbCon = SampleDatabase.Connect("scott","tiger");
      /* the code of program that sets error when needed */
      if (error)
            DbCon->Rollback();
            DbCon->Disconnect();
            // the same as DbCon->Abort()
      else
            DbCon->Commit();
            // not needed,
            // Disconnect() commits last running transaction
            DbCon->Disconnect();
      return error;
}
```

♦

To commit or to roll back all opened connections on the same logical database, application can use methods

virtual bool Database::Commit();

or

virtual bool Database::Rollback();

defined on the *Database* class. The library model doesn't support nested transactions, but supports savepoints. Savepoints are named marks defined inside the transaction. When necessary, application can return the state of data back to one of previously defined savepoints and partially roll back the transaction. Two additional methods

virtual bool Connection::Savepoint(const char * const);

and

virtual bool Connection::RollbackToSavepoint(const char * const);

serve for this purpose. Advantages of savepoints appear in case that some operation that must be done fails. It is possible to repeat the operation more times, as in following example.

Example 4.7-2 – Savepoints

```
// try it five times
for(int i = 0; i < 5; i++) {
      DbCon->Savepoint(”sp1”);
      if (ProcessData()==0)
            break;
      // not successfull, return and try it again
      DbCon->RollbackToSavepoint(”sp1”);
};
```

♦

In spite of standardisation different implementations of RDBMS can differ in details of transactional processing. Oracle RDBMS, which was used for pilot implementation, fulfils following conditions.

- The database server uses by default row-level locking rather than table level locking.
- Server exclusively locks row of data automatically when the session updates it.
- The lock on the row doesn't deny other sessions to read the original data stored in the row.
- Until transaction is committed, the updated data are visible only from the session, which changed them. After the commit operation, all locks made by the transaction are released, changes become permanent and visible to other sessions.

4.8 Object Manipulation and Consistency Control

The object references allow indirect access to persistent objects regardless of its location in the memory. In time of their dereferencing, it must be possible to find out if the wanted instance of the object is already placed in the memory or not. If the corresponding persistent instance is already in the memory, its address can be used. If not, the instance must be loaded from the database. Re-using of previously loaded object copies significantly speeds up the application. On the other hand, keeping of multiple copies of the same object on more places across the network requires additional overhead to preserve data consistency.

The object instance management inside the library works in the similar way as on-demand paging module inside the virtual memory management in the operating system. Different settings of object instance management allow trade off the speed of the application for more strictly data consistency and vice versa. The more strictly data locking, the more secure multi-user access and the slower application. Thus, the programmer should appropriately balance the security needs with the program speed accordingly the particular application. Below described data access strategies can be defined on three levels. Most general settings on database level can be further redefined on session or even on the object level. This approach allows the application to distinguish objects importance case by case.

The concurrent access to the shared data can be set: using combinations of four characteristics – updating, waiting, locking and reading strategy.

4.8.1 Updating strategy

The update strategy determines the way in which the updated objects are handled. Changes can be either propagated immediately to the database or they can be deferred and written to the database either explicitly by the programmer using the object's *Update()* method or by the object instance management when necessary.

The update strategy can be set to one of enumerated values given by type

```
enum UpdatingStrategy {
        US_Default,
        US_Current,
        US_Inherited,
        US_OnDemand,   //default setting
        US_Immediately
        };
```

The required update strategy can be set on different levels by methods

```
bool Database::SetUpdatingStrategy(enum UpdatingStrategy)
bool Connection::SetUpdatingStrategy(enum UpdatingStrategy)
bool ObjRef::SetUpdatingStrategy(enum UpdatingStrategy)
```

and obtained by methods

```
enum UpdatingStrategy Database::CurrentUpdatingStrategy() const
enum UpdatingStrategy Connection::CurrentUpdatingStrategy() const
enum UpdatingStrategy ObjRef::CurrentUpdatingStrategy() const
```

Setting the strategy to *US_Default* uses default setting (*US_OnDemand*), hard-coded to the object management. Value named *US_Current* doesn't change the strategy and so it can serve as default value of parameters inside methods. Using of *US_Inherited* setting has no effect on the database level. On other levels it sets the strategy accordingly to the setting of the higher level.

4.8.2 Locking strategy

When the required object is loaded from the database into program memory, rows containing the object data in the database can be locked either in none, shared or exclusive mode. According to those possibilities, allowed values are enumerated by type

```
enum LockingStrategy {
        LS_Default,
        LS_Current,
        LS_Inherited,
        LS_None, //default setting
        LS_Shared,
        LS_Exclusive
        };
```

Current settings of locking strategy can be found out and set through following set of methods

```
bool Database::SetLockingStrategy(enum LockingStrategy)
bool Connection::SetLockingStrategy(enum LockingStrategy)
bool ObjRef::SetLockingStrategy(enum LockingStrategy)
enum LockingStrategy Database::CurrentLockingStrategy()
enum LockingStrategy Connection::CurrentLockingStrategy()
enum LockingStrategy ObjRef::CurrentLockingStrategy()
```

4.8.3 Waiting strategy

Setting of the waiting strategy determines the behaviour of the object management in case the object locked in another session should be changed or locked. It is either possible to wait until the object becomes unlocked, or throw an exception. The exception can be detected by the application. In case the exception is detected, the application can do another work before it re-tries the update.

By analogy to updating and locking strategies, possible waiting strategy settings are enumerated by type

```
enum WaitingStrategy {
        WS_Default,
        WS_Current,
        WS_Inherited,
        WS_Wait,//default setting
        WS_Nowait
        };
```

The requested waiting strategy can be set and obtained using access methods

```
bool Database::SetWaitingStrategy(enum WaitingStrategy)
bool Connection::SetWaitingStrategy(enum WaitingStrategy)
bool ObjRef::SetWaitingStrategy(enum WaitingStrategy)
enum WaitingStrategy Database::CurrentWaitingStrategy()
enum WaitingStrategy Connection::CurrentWaitingStrategy()
enum WaitingStrategy ObjRef::GetWaitingStrategy()
```

4.8.4 Reading strategy

This setting influences behaviour of the object management at the moment the object should be accessed by its reference and its copy already exists in the memory. It can be used either cached copy or new data can be loaded from the database first. Repeated reading of object from the database can noticeably slow down the application. As a compromise, the object management can first compare the timestamp of the instance in the database with those remembered during last fetch. Possible reading strategy settings are enumerated by type

```
enum ReadingStrategy {
        RS_Default,
        RS_Current,
        RS_Inherited,
        RS_Cache,          //default setting
        RS_Database,
        RS_Timestamp
        };
```

As usual, changes are handled by access methods

```
bool Database::SetReadingStrategy(enum ReadingStrategy)
bool Connection::SetReadingStrategy(enum ReadingStrategy)
bool ObjRef::SetReadingStrategy(enum ReadingStrategy)
enum ReadingStrategy Database::CurrentReadingStrategy()
enum ReadingStrategy Connection::CurrentReadingStrategy()
enum ReadingStrategy ObjRef::GetReadingStrategy()
```

The optimal choice between *RS_Database* and *RS_Timestamp* depends on the frequency of object changes. The greater probability of object change, the larger overhead of *RS_Timestamp* choice is. Really, if the object was not changed, only the object's timestamp must be read from one table in the database. If the timestamp differs from the last fetched one, the fetch of complete object must follow.

Some important combinations of setting of above-listed strategies are shown in following table.

Autocommit	Updating Strategy	Locking Strategy	Waiting Strategy	Reading Strategy	Consistency
Yes	*US_Immediate*	*LS_None*	-	*RS_Database*	*Causal*
No	*US_Immediate*	*LS_None*	-	*RS_Database*	*Transactional*
No	*US_Immediate*	*LS_None*	-	*RS_Timestamp*	*Transactional*

No	*US_OnDemand*[6]	*LS_None*	-	*RS_Database*	*Transactional*
No	*US_OnDemand*	*LS_None*	-	*RS_Timestamp*	*Transactional*
No	*US_OnDemand*	*LS_None*	-	*RS_Cache*	*Default*

Last row of the table corresponds to the default settings. It doesn't prevent reading out-dated data from the object cache, even when there are newer committed version in the database. But in many cases it is sufficient and most efficient in terms of speed of the application.

Flexibility of settings allows finding suitable compromise between the consistency and performance. Some object may be volatile, and its changes must be immediately propagated to the database and available to other sessions. Another objects may be constant, infrequently changed or exclusively maintained by only one process. In this case the overhead caused by consistency checking is unnecessary.

4.9 Object Cache

The object cache is implemented by class *ObjectBuffer* with its solitaire instance named *ObjectCache*. The main functionality of this class is to keep track of all object instances loaded currently from the database into memory and translate database pointers to memory pointers as fast as possible. From the logical point of view the object buffer works as an associative array of memory pointers indexed by database pointers.

Each database pointer contains three components, a definition of strategies for object manipulation, a pointer to the database connection and an identification of the object in the database. The identification of the object is stored in attribute of class *ObjectIdentification* type. Because two different database connections can see different states of data in the database (changes made by one connection can be invisible to other connections until commit), the object buffer works as two-dimensional array. One dimension is indexed by the database connection, the second one by the object identification itself.

The object identification consists of the pointer to correct prototype of the persistent capable class and the list of values of the primary key.

Whenever the application dereferences a database pointer, it invokes the method

Object *ObjectBuffer::GetReferencedObject(const class RefBase &DbPtr);

that searches for corresponding object. If it finds the registration record, the registered address of the object is returned. If not, the new instance of the class represented by the prototype is created using method

C *Proto<C>::New();

and then the just created object instance is filled by needed data from the database and registered within the buffer. The locking strategy stored in the database pointer is taken into account to decide if the loaded object should be locked in the database or not.

The buffer stores registration records separately according to hash value of primary key values. This significantly reduces the amount of information it has to be read during address translation. The object buffer represents one of central point in the library implementation. The object buffer provides methods that impact default behaviour of the library and manipulates with more objects at the same time.

[6]Supposed that the database runs not in dirty read mode.

It is the object buffer from witch all database instances derive its default strategies (Updating, Reading, etc.) at time of their creation. Settings can be changed using the same interface as in case of databases, connections and pointers.

bool ObjectBuffer::SetReadingStrategy(enum ReadingStrategy)
enum ReadingStrategy ObjectBuffer::CurrentReadingStrategy()
bool ObjectBuffer::SetUpdatingStrategy(enum UpdatingStrategy)
enum UpdatingStrategy ObjectBuffer::CurrentUpdatingStrategy()
bool ObjectBuffer::SetWaitingStrategy(enum WaitingStrategy)
enum WaitingStrategy ObjectBuffer::CurrentWaitingStrategy()
bool ObjectBuffer::SetLockingStrategy(enum LockingStrategy)
enum LockingStrategy ObjectBuffer::CurrentLockingStrategy()

All changed objects currently registered in the memory can be synchronised with the database using one of methods

bool ObjectBuffer::UpdateAll();
bool ObjectBuffer::UpdateAll(const class Connection *const DbC);
bool ObjectBuffer::UpdateAll(const class Database *const DB);

that differs in amount of processed objects. First of mentioned methods updates all changed objects from all databases, the second one processes only objects retrieved from one database and the last one updates only objects read through given connection.

It is also possible to free all objects registered in the buffer. For this purpose are defined methods

bool ObjectBuffer::RemoveAll();
bool ObjectBuffer::RemoveAll(const class Connection *const DbC);
bool ObjectBuffer::RemoveAll(const class Database *const DB);

Before they are released from the memory, it is possible to unlock them all using methods

bool ObjectBuffer::RemoveAllMemoryLocks ();
bool ObjectBuffer::RemoveAllMemoryLocks (const class Connection *const DbC);
bool ObjectBuffer::RemoveAllMemoryLocks (const class Database *const DB);

Note that each instance must be unregistered from the memory before its memory image is destroyed. To do it the destructor of each persistent capable class must call its inherited method

bool Object::_Free();

It must be called before destructors start to destroy the instance. The method *_Free()* causes the object to be updated to the database if it is necessary and this action cannot be deferred.

4.10 Exceptions and Exception Handling

The object library use capabilities of C++ language to handle exceptional states during data manipulation. During the execution of the application, some exceptions may appear due to incorrect statement executed on the database server, the error in the network communication, incorrect use of library services, insufficient amount of resources (memory), etc. All exceptions are derived from the common predecessor class named *ObjLibException*, as it is shown on picture below.

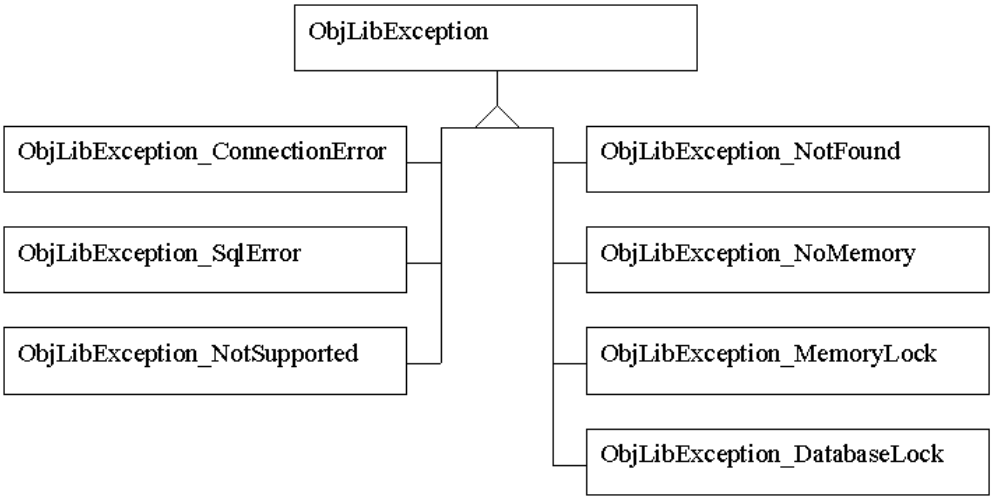


Figure 4.10-1 Class diagram of exception hierarchy

The *ObjLibException* class includes method

virtual const char *ObjLibException::Name()

that returns the name of the caught exception. Programmer can declare try block with exception handler capable of handling exception of type *ObjLibException* or any of its descendants. If any operation within the try block raises the exception, the handler is activated.

Usage of exception handling during connection to the database shows following example.

Example 4.10-1 – Exception handling

```
class OracleDatabase SampleDatabase("");
//default oracle database
DatabaseConnection *DbCon = NULL;
try {
      printf("Trying to connect to the database ...\n");
      DbCon = SampleDatabase.Connect("user_name","password");
}
catch (ObjLibException &X) {
      printf("... NOT connected\n");
      throw;
};
printf("... connected\n");
```

♦

There are seven exception subclasses defined by the library.

class ObjLibException_NotSupported : public ObjLibException

This exception can appear whenever the library tries to use any feature, which is not supported by the used SQL engine. In case of the prototype implementation based on Oracle service, the exception is thrown, whenever the application tries to go backward in the query result.

class ObjLibException_ConnectionError : public ObjLibException

The *ObjLibException_ConnectionError* is thrown, if the library can't communicate with the database server through the given connection. The connection can’t be established, was already closed by the application or by the server.

class ObjLibException_SqlError : public ObjLibException

This exception is thrown, if the database server does not recognise the issued SQL command due to either its wrong syntax or semantic error.

class ObjLibException_DatabaseLock : public ObjLibException

This exception is thrown, if the specified object can't be changed in the database because the table or the row, representing the object is locked by another connection and the library should not wait for its release.

class ObjLibException_MemoryLock : public ObjLibException

The library throws memory lock exception, whenever the application demands de-allocate the object previously locked in the memory. The object must be unlocked first.

class ObjLibException_NoMemory : public ObjLibException

If there is not enough memory to complete requested operation, library generates this exception.

class ObjLibException_NotFound : public ObjLibException

Last sub-type of exception appears whenever the application tries to load and access object that does not exists in the database. The most probably it was already permanently deleted.

4.11 Table-level locking and mutual exclusion

By default, objects are locked in the database as necessary. The object locking protocols are driven by the *LockingStrategy* and *WaitingStrategy* settings. The rows containing data are locked exclusively at latest at the time the session changes them in the database. Until committing of the connection, all other sessions will read old data. Any attempt to change the same object from another connection will be detected by the server.

In addition to the object-level locking, the application can try to lock whole table associated with given class. For this purpose the application should call the method

bool Proto<C>::LockTable(
Connection *,
enum LockingStrategy = LS_Current,
enum WaitingStrategy = WS_Current
)

defined on the *ProtoBase* class. It tries to lock the whole table associated with the class *C* in the database. If the method fails and the *WS_NoWait* strategy was chosen, the method raises an exception *GenLibException_DatabaseLock*. Otherwise the process will wait until all previous locks on the table are released. One of applications of table-level locking is implementation of semaphores. Semaphores can be then used for mutual exclusion of processes.

Each semaphore is represented by one of program subclasses. Because locked objects in the database can be unlocked only at the end of transaction, each semaphore needs its own database connection that can be committed independently on the others. The application can define class *Semaphore* with needed successors. Having such classes and dedicated connection DbConn, we can simulate the semaphore in following way:

Example 4.11-1 – Critical section based on table-level locking

```
// ...
//Try to lock table exclusively. Wait until succeeded.
Class[”Semaphore”]->LockTable(DbCon,LS_Exclusive,WS_Wait);
//Begin of critical section
// ...
DbCon->Commit();
//End of critical section
```

♦

Drawback of this technique is the necessity having one class for each planned semaphore. To avoid such requirement, the proposed persistent object library allows another solution of mutual exclusion using row-level approach.

In difference to previous solution each semaphore will need only one instance of persistent class Semaphore and one database connection. At the entry of the critical section process simply loads the instance from the database through given database connection. When the process leaves the critical section, it rolls back the connection and releases the acquired row-level lock. Rolling the transaction back also discards the *Semaphore* instance from the object cache. Semaphores can be identified by its unique name in the system.

When the total count of needed semaphores together with their names are not known in advance, the program can create new and destroy obsolete semaphores at run-time. During the time when one process creates new semaphore instance with the given name it must be guarantied that no other process will try to create the same one. Let imagine two identical programs running in parallel on different computers.

```
Process No. 1                          Process No. 2

try {                                  try {
  fetch and lock semaphore "X"           fetch and lock semaphore "X"
} catch("X" not exists) {              } catch("X" not exists) {
  create semaphore "X" /**!**/           create semaphore "X" /**!**/
  fetch and lock semaphore "X"           fetch and lock semaphore "X"
};                                     };
...                                    ...
unlock semaphore X                     unlock semaphore X
```

In this case both processes fail lock the semaphore X in the database due its inexistence. Consecutively both processes start create the semaphore and one of them fails again due to required uniqueness of semaphore names. Solution is to use one predefined semaphore and close above shown piece of code to critical section.

```
Process No. 1                          Process No. 2

try {                                  try {
  fetch and lock semaphore "X"           fetch and lock semaphore "X"
} catch("X" not exists) {              } catch("X" not exists) {
  fetch and lock semaphore "NewSem"      fetch and lock semaphore "NewSem"
  create semaphore "X" /**!**/           create semaphore "X" /**!**/
  fetch and lock semaphore "X"           fetch and lock semaphore "X"
  unlock semaphore "NewSem"              unlock semaphore "NewSem"
};                                     };
...                                    ...
unlock semaphore "X"                   unlock semaphore "X"
```

The third implementation of semaphores is fully based on processing of unique values in the used relational database. Whenever session S_2 tries to create row with already existing value in unique column but not committed by session S_1, the database defer the insertion until already existing value is committed or rolled back by first session. If S_1 commits the value, S_2 fails. Else S_2 succeeds.

```
Process No. 1                          Process No. 2

create semaphore "X"                   create semaphore "X"
// ... critical section                // ... critical section
rollback the session                   rollback the session
```

One of processes creates the object and enters critical section while second one will wait until the section will be empty.

4.12 Querying objects

The library implements object algebra described in section 3.2. Each predicate of the object algebra is represented by one instance of *Query* class. New instance of the query can be declared and initialised using constructor

Query::Query(
const char *const Where = NULL,
const char *const OrderBy = NULL
);

When used without parameters, an empty query is created. The empty query doesn't restrict the WHERE condition at all. Its execution returns all instances of the target class. It also doesn't define ordering clause of SELECT statement. There is one empty constant query instance named EQUERY predefined in the library. This instance, defined as

static const Query EQUERY();

can be used whenever condition is required and no restriction should be used. The same constructor used with one parameter only builds a query containing no ordering definition. The last allowed format with two parameters defines also ordering of the resulted set of objects.

Except *EQUERY*, there are three additional predefined queries *ALL*, *NONE* and *EQUERY* declared as

static const Query ALL("1=1");
static const Query NONE("1<>1");
static const Query EQUERY("");

Both condition and ordering clause of queries can be further modified and read using defined access methods

const char *Where() const;
bool Where(const char *const Where);
const char *OrderBy() const;
bool OrderBy(const char *const OrderBy);

The language that can be used to express the condition and ordering combines C++ expressions together with SQL ones. Restriction to only SQL language causes unnecessary and unwanted impedance problem where because the programmer must take into account how the attributes are mapped onto table columns. On the other hand, allowing only the C++ syntax reduces significantly possibilities of the language. The conditions couldn't use any of columns or tables that are not mapped by the library. The library automatically translates following C++ expressions and operators to SQL:

1. C++ equivalence operator "==" is translated to SQL operator "=".
2. C++ operator "!=" (not equal to) is translated to operator "<>"
3. C++ operator "!" (logical not) is translated as " NOT "
4. C++ operator "&&" (logical and) is translated as " AND "
5. C++ operator "||" (logical or) is translated as " OR "
6. Expression Class_name::Attribute_name is translated to SQL expression in form TABLE_NAME.COLUMN_NAME.

This translation allows writing C++-like query

```
Query Q("Person::Surname=='Smith' && !(Driver::LicenseNr==12345")
```

instead of its SQL transcription

```
Query Q("TPERSON.SURNAME='Smith' AND NOT(TDRIVER::LICENCE_NR=12345")
```

that depends on the mapping. But if there is, for example a table LOST_LICENCE that contains a list of lost driving licences, the query can use it without limitation.

```
Query Q("Driver::LicenseNr NOT IN (SELECT NR FROM LOST_LICENCE)")
```

A template class *Result<C>* derived from common predecessor *ResultBase* is defined for each persistent object class C starting with *Object* class as follows.

template <class C> class Result : public ResultBase

Its instances represent query results obtained by query execution of selection operator *round brackets*

```
Result<T> * Proto<T>::operator() (
        const class QueRefProto & const,
        const Connection *const
        )
```

defined on appropriate object prototype. The second parameter is needed to define the connection, on which the query should be executed. Execution on different connections can result in different results due to uncommitted changes in their transactions.

Abstract class *QueRefProto* used as first formal parameter is a common predecessor of classes *Query*, *ProtoBase*, *Object* and. *RefBase*. The hierarchy structure is in detail shown on Figure 4.12-1.

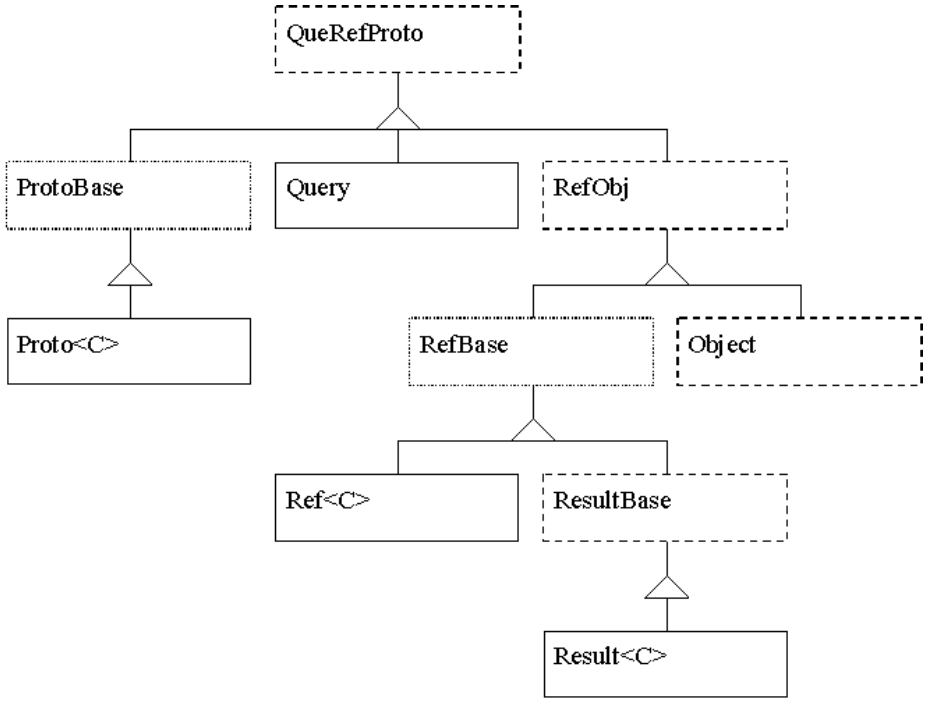


Figure 4.12-1 Class diagram of query and result hierarchy

All *QueRefProto* successors can be used as condition.

ProtoBase class represents individual persistent classes. When used as condition, produces WHERE fragment

(RootPrototype()->PrimaryKey()**) IN (SELECT** PrimaryKey() **FROM** From())

The *Object*, respectively *RefBase* classes represent object instance, respectively a reference to object instance. Both represent condition

Prototype()->PrimaryKey() = PrimaryKeyValues()

inherited from *RefObj* class.

The selection operator returns a pointer to the dynamically created instance of the *Result<C>* class. Query results are based on database cursors. The same way as database cursors, query results must be opened before accessing the data, closed at the end of work with them. They can be opened, reopened and closed using methods

bool Result<C>::Open();

and

bool Result<C>::Close();

Query execution returns query result already opened for more convenient work. In other aspects query results behave similar to arrays of object references. The original idea was to implement a query result to look like an ordinal C++ array of objects using method

class C &Result<C>::operator [](long int i)

Unfortunately not all databases support random access or two-way fetching from the database cursor. It could be still possible to use this approach on such a database, supposing that the index in the array will not decrease. Because there is not much difference between arrays of objects and pointers in C++, the final approach considers the query result to be an object reference. This approach keeps the C++-like style of data manipulation with query results. Most of needed methods are inherited by deriving *ResultBase* class from the *RefBase* class. To increase the similarity between the query results and the C++ style of manipulating arrays, the increment operators

Result<C> &Result<C>::operator ++();

and

Result<C> &Result<C>::operator +=(const int i);

are defined. Both operators fetch one, respectively given number of rows from the cursor. If the fetch succeeds, it points to next retrieved object instance. If there were not enough rows in the result, the query result becomes equal to DBNULL[7].

Example 4.12-1 – Query execution and result processing

To retrieve all persons older than forty ordered by the name and process all retrieved objects, the programmer should write:

```
// define the query
Query q("Age>40", "Name");
Result<Person> *r;
// execute the query
r = Person_class(q,DbCon); // resp. r = (*Class[”Person”])(q,DbCon);
// fetch next row from result until all workers are processed
while (++(*r) != DBNULL)
{
      // process the current element of the query result
};
// close the query result
r->Close();
// destroy the result instance
delete(r);
```

♦

As usual in the C++ language, queries can be combined using Boolean operators

class Query Query::operator &&(const QueRefProto &const Q) const;

---

7 Object references can be compared to equality and non-equality

class Query Query::operator ||(const QueRefProto &const Q) const;
class Query Query::operator !() const;

for corresponding operations on them. Programmer can copy queries using copy constructor

Query::Query(const QueRefProto &const Q);

and also by assignment operator

Query &Query::operator =(const QueRefProto &const Q);

Example 4.12-2 – Query manipulation

Using those operators and copy constructors, the programmer can write

```
Query q1("Name LIKE 'Smith %'");
Query q2("Age>25");
Query q3("Age<=40");
Query q4 = !q1 || (q2 && q2);
```

The query q4 will be the same as constructed using statement

```
Query q4(
      "NOT(Name LIKE 'Smith %')OR((Age>25)AND(Age<=40))"
      );
```

♦

4.13 Associations

The manipulation with associated objects within object algebra was defined in chapter 3.2.5. We considered one generic binary association, but in fact, we need to distinguish between more different sub-types. Individual sub-classes of generic class *RelationBase* differs in cardinality of the association, which results in different implementation of database operations. As it was represented, the application can define instances of five concrete classes

template <class L, class R>
ChainedRelation<L,R> : public ChainedRelationBase
template <class L, class R>
OneToOneRelation<L,R> : public OneToOneRelationBase
template <class L, class R>
OneToManyRelation<L,R> : public OneToManyRelationBase
template <class L, class R>
ManyToOneRelation<L,R> : public ManyToOneRelationBase
template <class L, class R>
ManyToManyRelation<L,R> : public ManyToManyRelationBase

The complete tree of all classes beginning with abstract class *Relation* is shown in figure below.

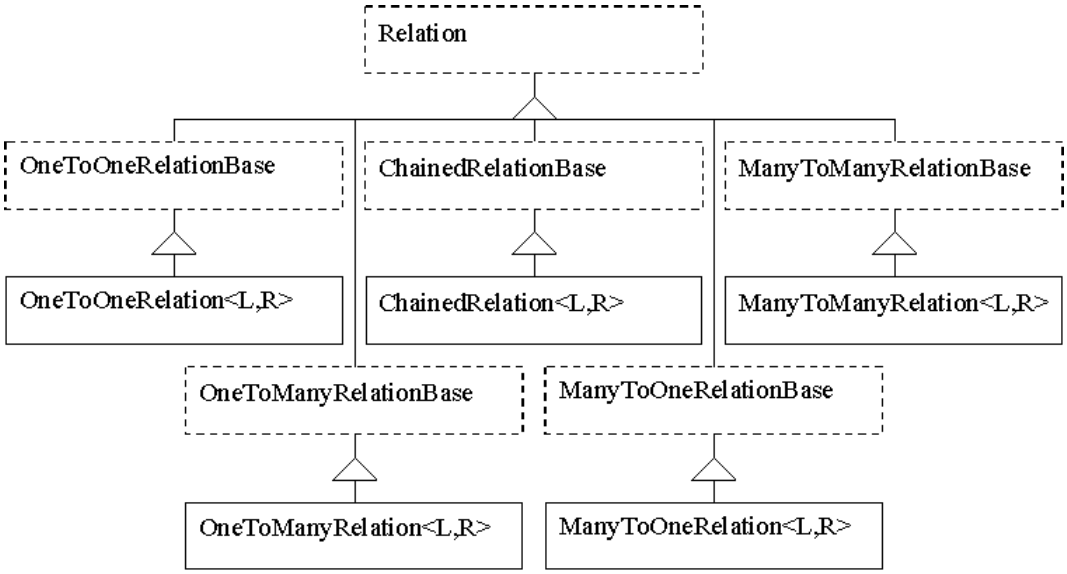


Figure 4.13-1 Class diagram – Relation hierarchy

Leaf classes in the hierarchy implement associations physically stored in the database. The *ChainedRelation<L,R>* class represents computed associations. Individual couples can be inserted to and deleted from the physical associations. Computed ones cannot be modified.

If modifications are possible, they can be accomplished by four methods

```
virtual bool Relation::InsertCouple(
        const RefObj &left, const RefObj &right
        )
virtual bool Relation::DeleteCouple(
        const RefObj &left, const RefObj &right
        )
virtual bool Relation::DeleteLeft(
        const RefObj &right
        )
virtual bool Relation::DeleteRight(
        const RefObj &left
        )
```

that allow to insert new couples and delete them. Last two methods delete all associations between given object instance *right*, respectively *left* and all object instances on the opposite side of the association. It is possible also test the existence of association between two object instances by method

```
virtual bool Relation::ExistsCouple(
        const RefObj &left, const RefObj &right
        )
```

Note that using abstract *RefObj* class as formal parameters of methods allows use references to objects instead of object instances themselves.

We have defined left-side restriction φ*R, right-side restriction R*φ_j, chained association R_1*R_2 and reversed association -R in the object algebra. In addition, we need both unary operators **left**(R) and **right**(R).

All above-mentioned operators are implemented also in the C++ library. Operator for reversed association must be defined five times, because result type of reversed association depends on cardinality of initial association. So we get the operator

```
template <class  L, class R> ChainedRelation<R,L> ChainedRelation<L,R>::operator –()
```

together with four additional definitions for other four sibling classes.

```
template <class  L, class R>
    OneToOneRelation<R,L> OneToOneRelation<L,R>::operator –()
template <class  L, class R>
    ManyToOneRelation<R,L> OneToManyRelation<L,R>::operator –()
template <class  L, class R>
    OneToManyRelation<R,L> ManyToOneRelation<L,R>::operator –()
template <class  L, class R>
    ManyToManyRelation<R,L> ManyToManyRelation<L,R>::operator –()
```

A bit more complicated thing is to define correctly all chaining operators. Operator must be defined for every combination of template types. One of them is the method

```
template <class  L, class X, class Y, class R> ChainedRelation<L,R> operator*(
    const ChainedRelation<L,X> &r1, const ChainedRelation<Y,R> &r2
    );
```

all other chaining methods differ only in types of parameters. The same situation is with operators

```
template <class  L, class R> ChainedRelation<L,R> Relation<L,R>::operator*(
    const QueRefProto &q, const ChainedRelation<L,R> &r
    )
```

for the left-side restriction, and

```
template <class  L, class R> ChainedRelation<L,R> Relation<L,R>::operator*(
    const ChainedRelation<L,R> &r, const QueRefProto&q
    )
```

for the right-side restriction. Again, the result type is the same for all combinations of parameters.

Operators **left**(R) and **right**(R) are implemented using two methods returning new Result<C> instance as.

```
template <class  L, class R> Result<L> *ChainedRelation<L,R>::Left(
    const class QueRefProto &r = EQUERY
    )
```

and

```
template <class  L, class R> Result<R> *ChainedRelation <L,R>::Right(
    const class QueRefProto &l = EQUERY
    )
```

In comparison with originally introduced object algebra we allow here its extension. Both **left** and **right** operators accept additional condition at the time of their execution. However, this extension doesn't extend the power of query language, because the expression *Left*(*R, q)* is equivalent to *Left(R*q)*.

To create instance of relation it is necessary to provide a name of the table that holds couples of associated primary keys (in case of *ManyToManyRelation*), a pointer to database connection that executes all SQL statements and optionally names of foreign key columns. If names of foreign key columns are not defined, they are derived from names of primary key columns in associated classes according to following rules:

- primary key columns belonging to the left associated class are prefixed with prefix "L_"
- primary key columns belonging to the right associated class are prefixed with prefix "R_"

Example 4.13-1 – Associations in C++

Thanks to copy constructors and assign operators on all relation classes the programmer can write expression

```
OracleDatabase MyDatabase();
Connection *DbCon = MyDatabase.Connect("user","password");
Ref<Leader> Lref;
Result<Programmer> *Wresult;
...
OneToOneRelation<Project,Leader> PL("PROJECT_LEADER",DbCon);
OneToManyRelation<Project,Programmer> PP("PROJECT_PROGRAMMER",DbCon);
WResult = Right(Lref*(-PL)*PP);
//the same as: WResult = Right(-PL*PP,Lref);
```

and obtain this way all workers taking part of any project supervised by given leader, whose database reference is stored in *Lref* variable.

♦

4.14 Binding

At run-time the object model described in 3.1 must be transparently translated to relational model used by RDBMS and vice versa.

Usual solution of the mapping problem is based on the mandatory pre-processing of the source code. Such solution presumes two phases of compilation. The programmer writes the wanted source code containing constructs in C++, respectively other host programming language together with constructs in specific ODL. As it is shown on following figure, the source code is first pre-processed by specific ODL pre-processor. ODL pre-processor generates both pure C++ source code for the application and SQL script needed for that generates needed relational schema. Generated source code is then compiled the standard way and linked with RDBMS run-time that provides communication with the database.

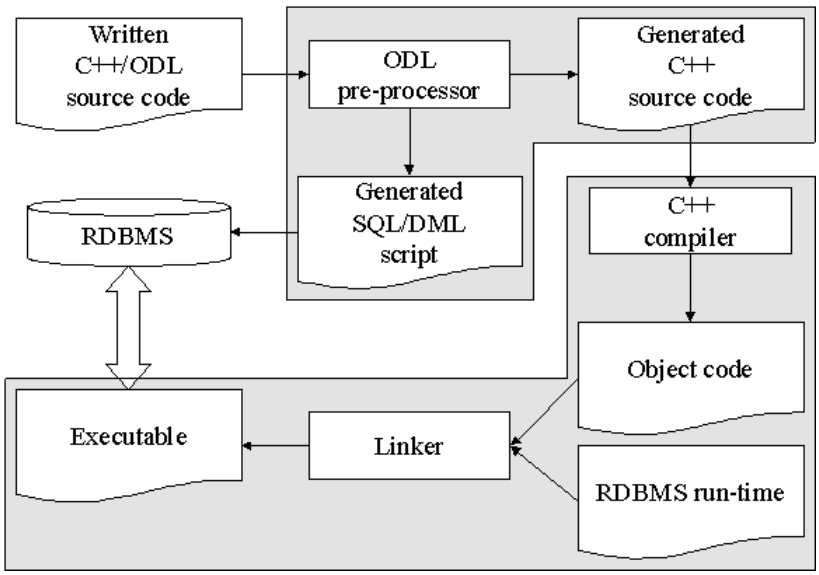


Figure 4.14-1 Compilation of the application

The automatic pre-processing of the source code allows two different approaches to C++ code generation. First of mentioned methods leaves original definitions of persistent C++ classes intact and generates code of auxiliary classes that manipulate with data. The schema of auxiliary code generation reflects following figure.

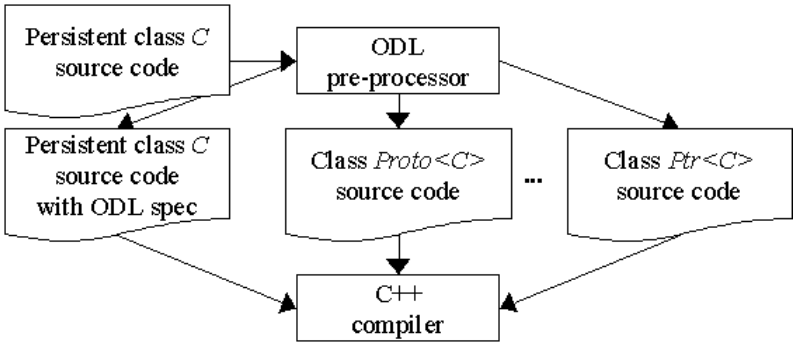


Figure 4.14-2 Class pre-processing – method I

Our goal was to allow also hand-made mapping definition. To achieve this goal the mapping should be as most simple as possible. The removal of ODL pre-processor from the compilation causes the necessity of synchronous changes of persistent class definition and all its auxiliary classes. Such a situation complicates the thing. This reason leads to the idea of keeping all relevant information directly on one place – in persistent class definition itself. Application of this requirement leads to the pre-processing that adds needed piece of code to the persistent class definition itself. The C++ compiler then derives all auxiliary classes automatically from its templates.

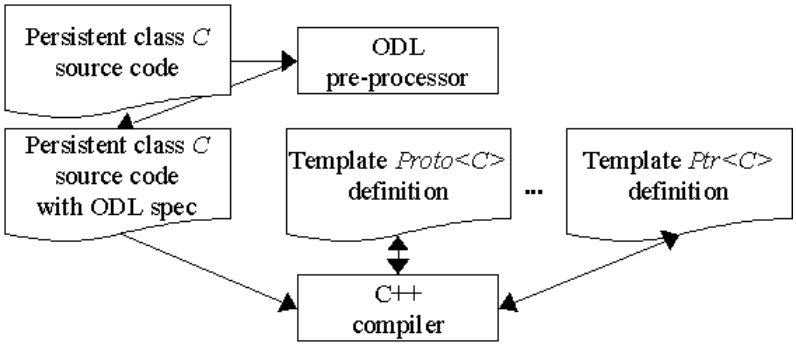


Figure 4.14-3 Class pre-processing– method II

By the removal of ODL-pre-processing we obtain much simpler diagram

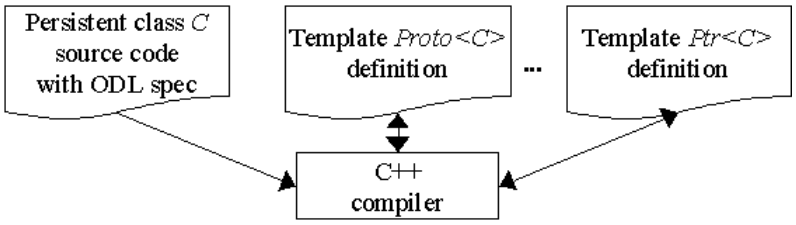


Figure 4.14-4 Method II without pre-processing

The proposed syntax of the ODL enhancements is compliant to internal C++ pre-processor. ODL constructs map every persistent class onto the data source in the relational model. Member attributes of such classes must be declared, mapped onto corresponding columns of the associated data source (table, view, SELECT statement etc.) and bound with generated SQL statements.

4.14.1 Class mapping

ODL clauses of C++ class definition describe the position of the class in the specialisation graph and associate it with the data source code.

The name of the class is declared by one of two clauses

CLASS(NameOfClass);
ABSTRACT_CLASS (NameOfClass);

located within class declaration. Using of ABSTRACT_CLASS clause forbids creation of instances of the given class. Descendants of *PersistentClass* class must specify comma-separated list of all direct parent classes. For this purpose the ODL provides clause

PARENTS("ParentClassList");

The association of declared class with the corresponding source of data provides set of clauses with names derived form the SQL syntax:

FROM("content-of-from-clause");
WHERE("content-of-where-clause ");
GROUP_BY("content-of-group-by-clause ");
HAVING("content-of-having-clause ");
ORDER_BY("content-of-order-by-clause ");

The information declared by ODL during the class design is available also at the run-time. Functions corresponding to individual ODL specifications are listed in following table.

ODL clause	**Run-time access methods**
CLASS(NameOfClass)	static const char * ClassName() static Object *New()
ABSTRACT_CLASS(NameOfClass)	static const char * ClassName() static Object *New()
PARENTS("ParentClassList")	static const char * ParentClassNames()
FROM("content-of-from-clause")	static const char * From()
WHERE("content-of-where-clause ")	static const char * Where()
GROUP_BY("content-of-group-by-clause ")	static const char * GroupBy()
HAVING("content-of-having-clause ")	static const char * Having()
ORDER_BY("content-of-order-by-clause ")	static const char * OrderBy()

The ODL specifications CLASS and ABSTRACT_CLASS also define method that creates one empty transient instance of the class. Of course, in case of abstract class, none instance is created and NULL is returned due to inability of abstract classes to be instantiated.

4.14.2 Member attribute declaration

The object model allows atomic member attributes capable of storing in the relational tables. Static members of classes are also allowed. Their declaration is not subject of ODL, as they are not stored to the database.

Instead of standard C++ declaration of the member attribute the programmer should use the ODL constructs listed in following table. The ODL member attribute declaration declares automatically access methods for its reading and setting. Member attribute itself is declared as protected and its name is prefixed by underscore character. Besides the standard (read-write) declarator we define also read-only declarator that suppresses creation of write access method. Read-only decelerators have "RO" suffix in its name.

C++ declaration	ODL declaration	Run-time declaration	Run-time access methods
class C *x;	dbPtr(C,x);	long int _x;	Ref<C> x() const; void x(const Ref<T>&);
	dbPtrRO(C,x);		Ref<C> x() const;
char *x;	dbString(x);	char *_x;	const char *x() const; void x(const char *);
	dbStringRO(x);		const char *x() const;
char x;	dbChar(x);	char _x;	char x() const; void x(char);
	dbCharRO(x);		char x() const;
short int x;	dbShort(x);	short int _x;	short int x() const; void x(short int);
	dbShortRO(x);		short int x() const;
unsigned short x;	dbUShort(x);	unsigned short _x;	unsigned short x() const; void x(unsigned short);
	dbUShortRO(x);		unsigned short x() const;
int x;	dbInt(x);	int _x;	int x() const; void x(int);
	dbIntRO(x);		int x() const;
unsigned int x;	dbUInt(x);	unsigned int _x;	unsigned int x() const; void x(unsigned int);
	dbUIntRO(x);		unsigned int x() const;
long int x;	dbLong(x);	long int _x;	long int x() const; void x(long int);
	dbLong(x);		long int x() const;
unsigned long x;	dbULong(x);	unsigned long _x;	unsigned long x() const; void x(unsigned long);
	dbULong(x);		unsigned long x() const;
bool x;	dbBool(x);	bool _x;	bool x() const; void x(bool);
	dbBool(x);		bool x() const;
float x;	dbFloat(x);	float _x;	float x() const; void x(float);
	dbFloat(x);		float x() const;
double x;	dbDouble(x);	double _x;	double x() const; void x(double);
	dbDouble(x);		double x() const;

4.14.3 Member attribute mapping

After the data source for the C++ class is known, persistent attributes of the class must be associated with the corresponding columns of the SQL SELECT statement defined above.

To map member attributes of defined class provided ODL use two constructs. First of them maps member attributes associated with primary key of the data source, the second one is used for all other member attributes. Their syntax described by regular expression is

```
MAPKEY_BEGIN ( map_specificator )* MAPKEY_END;
```

respectively

```
MAP_BEGIN ( map_specificator )* MAP_END;
```

where the specificator can be repeated any times. One map specificator corresponds to one member attribute of the class. The format of the map specificator depends on the ODL data type of the member attribute. It associates SQL expression with the name of the member attribute. In case of string (ODL data type dbString) it defines also its maximal allowed length. SQL expressions should be fully qualified by table name to avoid confusion when more tables are joined together. Independently on the fact, if the member attribute can be changed from outside of the instance the programmer can map the attribute as read-only to the database. Attributes that are read-only mapped are neither inserted into, nor updated in the database. Note that primary keys are treated as read-write regardless on used specification. Descendants of *Object* and *ImmutableObject* classes are not stored and updated at all.

Available mappings are listed below

ODL member attribute declaration	Corresponding map specificator
dbPtr(C,x);	mapPtr(x,expr); mapPtrRO(x,expr);
dbString(x);	mapString(x,expr,length); mapStringRO(x,expr,length);
dbChar(x);	mapChar(x,expr); mapCharRO(x,expr);
dbShort(x);	mapShort(x,expr); mapShortRO(x,expr);
dbUShort(x);	mapUShort(x,expr); mapUShortRO(x,expr);
dbInt(x);	mapInt(x,expr); mapIntRO(x,expr);
dbUInt(x);	mapUInt(x,expr); mapUInt(x,expr);
dbLong(x);	mapLong(x,expr); mapLongRO(x,expr);
dbULong(x);	mapULong(x,expr); mapULongRO(x,expr);
dbBool(x);	mapBool(x,expr); mapBoolRO(x,expr);
dbFloat(x);	mapFloat(x,expr); mapBoolRO(x,expr);
dbDouble(x);	mapDouble(x,expr); mapDoubleRO(x,expr);

At the run-time the application can obtain all necessary information by calling protected static methods

```
protected: static void C::_MapKey(
        int i,
        const char *&attr,
        unsigned char Object::*mem
        const char *&expr,
        char &type,
        unsigned int &len,
        bool &rw
        );
```

respectively

```
protected: static void C::_Map(
        int i,
        const char *&attr,
        unsigned char Object::*mem
        const char *&expr,
        char &type,
        unsigned int &len,
        bool &rw
        );
```

of the corresponding class C. They are accessed virtually through virtual protected methods

```
protected: virtual void Proto<C>::_MapKey(
        int i,
        const char *&attr,
        unsigned char Object::*mem
        const char *&expr,
        char &type,
        unsigned int &len,
        bool &rw
        );
```

respectively

```
protected: virtual void Proto<C>::Map(
        int i,
        const char *&attr,
        unsigned char Object::*mem
        const char *&expr,
        char &type,
        unsigned int &len,
        bool &rw
        );
```

defined on the corresponding prototype. First parameter is the order of the member attribute counted from zero. The method returns the name of the member attribute, the address of the member attribute inside the instance, the SQL expression that must be used to retrieve data from data source, the type of the attribute coded by internal type code, the attribute's maximal length and read-write characteristics. In case the attribute on the i-th position is not defined, methods return empty attribute name, empty expression and "unknown" data type with zero length. Addresses of attributes are used to transfer their values between object instances and the database. Internal type codes are

Internal code	**Value**	**Type of member attribute**
TYPE_INT	'i'	Signed integer (of any length)
TYPE_UNSIGNED	'u'	Unsigned integer (of any length)
TYPE_FLOAT	'f'	Real number (float or double)
TYPE_CHAR	'c'	Char
TYPE_STRING	's'	Zero-terminated string (char *)
TYPE_UNKNOWN	'?'	Non-existing member attribute

4.14.4 Deriving object classes

Amount of information that can or must be provided for particular class depends on the basic type of persistent class. Following table summarises all differences.

	Object	Immutable Object	Database Object	Persistent Object
Class hierarchy definitions				
CLASS ABSTRACT_CLASS	✓REQ ×	✓REQ ×	✓REQ ×	✓REQ
PARENTS	×	×	×	✓REQ
Associated table/select definitions				
FROM	✓REQ	✓REQ	✓REQ	✓REQ
WHERE	✓	✓	✓	×
GROUP_BY	✓	✓	×	×
HAVING	✓	✓	×	×
Prototype definitions				
CLASS_PROTOTYPE ABSTRACT_CLASS_PROTOTYPE	✓REQ ×	✓REQ ×	✓REQ ×	✓REQ
PROTOTYPE	✓REQ	✓REQ	✓REQ	✓REQ
Attribute mapping				
MAPKEY_BEGIN ... MAPKEY_END	×	✓	✓	×
MAP_BEGIN ... MAP_END	✓	✓	✓	✓

ODL specification from the first column of the table must be used, if the class is derived, directly or indirectly, from classes that are checked with the ✓REQ sign. The sign ✓ allows redefinition of the class property meanwhile value × forbids it. Abstract descendants of the *PersistentObject* class should use definitions prefixed by ABSTRACT.

Example 4.14-1 – deriving *Object* descendant

Suppose the existence of the table

```
BANK_ACCOUNT(
      ACCOUNT_NUMBER STRING(20) PRIMARY KEY,
      ACCOUNT_OWNER INTEGER REFERENCES CUSTOMER.CUST_NR,
      ACCOUNT_CURRENCY STRING(3),
      ACCOUNT_BALANCE FLOAT
      );
```

in the database. We can obtain statistical data from this table using select statement

```
SELECT
      BANK_ACCOUNT.ACCOUNT_CURRENCY,
      COUNT(*),
      AVG(BANK_ACCOUNT.ACCOUNT_BALANCE)
FROM BANK_ACCOUNT
GROUP BY BANK_ACCOUNT.ACCOUNT_CURRENCY
HAVING COUNT(*)>100
ORDER BY COUNT(*) DESC;
```

If we want to process them in C++ program, we should define new descendant of *Object* class capable to store rows from this select

```
class AccountStat : public Object {
      dbString(Currency,3);
      dbInt(Count);
      dbFloat(Avg);
};
```

and map it onto the select statement. The mapping of class will add information to the class definition.

```
class AccountStat : public Object {
      dbString(Currency);
      dbInt(Count);
      dbFloat(Avg);
      CLASS(AccountStat);
      FROM("BANK_ACCOUNT");
      GROUP_BY("BANK_ACCOUNT.ACCOUNT_CURRENCY");
      HAVING("COUNT(*)>100");
      ORDER_BY("COUNT(*) DESC");
      MAP_BEGIN
            mapString(Currency,"BANK_ACCOUNT.ACCOUNT_CURRENCY",3)
            mapInt(Count,"COUNT(*)")
            mapFloat(Avg,"AVG(BANK_ACCOUNT.ACCOUNT_BALANCE)")
      MAP_END;
};
CLASS_PROTOTYPE(AccountStat);
```

Requirement of fully qualified expressions brings overhead to the class design. To avoid problems when the name of the table changes we allow using meta-name #THIS for reference to the associated table name. Using this syntax extension we can define the *AccountStat* class more easilly.

```
class AccountStat : public Object {
      dbString(Currency);
      dbInt(Count);
      dbFloat(Avg);
      CLASS(AccountStat);
      FROM("BANK_ACCOUNT");
      GROUP_BY("#THIS.ACCOUNT_CURRENCY");
      HAVING("COUNT(*)>100");
      ORDER_BY("COUNT(*) DESC");
      MAP_BEGIN
            mapString(Currency,"#THIS.ACCOUNT_CURRENCY",3)
            mapInt(Count,"COUNT(*)")
            mapFloat(Avg,"AVG(#THIS.ACCOUNT_BALANCE)")
      MAP_END;
};
```

♦

Example 4.14-2 – deriving *DatabaseObject* descendant

Suppose that we do not want read statistics about bank accounts, but also process rows of the table directly, we can define another persistent class, this time derived from the *PersistentObject* class and map it onto table BANK_ACCOUNT.

```
class Account : public DatabaseObject {
      dbString(AccountNr);
      dbInt(Owner);
      dbString(Currency);
      dbFloat(Balance);
      CLASS(Account);
      FROM(”BANK_ACCOUNT”);
      MAPKEY_BEGIN
            mapString(AccountNr,”#THIS.ACCOUNT_NUMBER”,20)
      MAPKEY_END;
      MAP_BEGIN
            mapInt(Owner,”#THIS.ACCOUNT_OWNER”)
            mapString(Currency,”#THIS.ACCOUNT_CURRENCY”,3)
            mapFloatRO(Balance,”#THIS.ACCOUNT_BALANCE”)
      MAP_END;
};
CLASS_PROTOTYPE(Account);
```

Due to read-only mapping of the account balance the program will not be able to change the amount of money on individual accounts albeit it is possible to change it in the memory.

♦

Example 4.14-3 – deriving *ImmutableObject* descendant

The identical definition of the class *Account*, this time derived from the *ImmutableObject* class, allows reading accounts to the memory but not changing them in the database at all.

```
class Account : public ImmutableObject {
      dbString(AccountNr);
      dbInt(Owner);
      dbString(Currency);
      dbFloat(Balance);
      CLASS(Account);
      FROM(”BANK_ACCOUNT”);
      MAPKEY_BEGIN
            mapString(AccountNr,”#THIS.ACCOUNT_NUMBER”,20)
      MAPKEY_END;
      MAP_BEGIN
            mapInt(Owner,”#THIS.ACCOUNT_OWNER”)
            mapString(Currency,”#THIS.ACCOUNT_CURRENCY”,3)
            mapFloatRO(Balance,”#THIS.ACCOUNT_BALANCE”)
      MAP_END;
};
CLASS_PROTOTYPE(Account);
```

This time would be better to declare all four member attributes as read-only to deny changes of instance in memory.

♦

Example 4.14-4 – deriving *PersistentObject* descendant

Let suppose now that we want to define regular C++ class Account capable to be further specialised in its sub-classes. In this case the class should be derived from *PersistentClass* class or any of its descendants. To the class

```
class Account : public PersistentObject {
      dbString(AccountNr);
      dbPtr(Person,Owner);
      dbString(Currency);
      dbFloat(Balance);
};
```

should be added needed ODL specifications as follows

```
class CppAccount : public PersistentObject {
      dbString(AccountNr);
      dbPtr(Person,Owner);
      dbString(Currency);
      dbFloat(Balance);
      CLASS(CppAccount);
      PARENTS(”PersistentObject”);
      FROM(”CPPACCOUNT”);
      MAP_BEGIN
            mapString(AccountNr,”ACCOUNTNR”,20)
            mapPtr(Owner,”OWNER”)
            mapString(Currency,”CURRENCY”,3)
            mapFloat(Balance,”BALANCE”)
      MAP_END;
};
CLASS_PROTOTYPE(Account);
```

In this case, of course, the table in the database must be declared according to the class definition.

```
TABLE CPPACCOUNT (
      OID NUMBER
            CONSTRAINT CPPACCOUNT_PK PRIMARY KEY
            CONSTRAINT CPPACCOUNT_FK_PERSISTENTOBJECT
                  REFERENCES PERSISTENTOBJECT(OID) ON DELETE CASCADE,
      ACCOUNTNR VARCHAR2[8](20),
      OWNER NUMBER
            CONSTRAINT CPPACCOUNT_OWNER_FK
                  REFERENCES PERSON(OID),
      CURRENCY VARCHAR2(3),
      BALANCE FLOAT
};
```

The OID column should be declared as primary key column also in the database. The reference to the parent class table together with ON DELETE CASCADE clause simplifies deleting of object. The table definition can be generated algorithmically.

♦

4.14.5 DDL statement generation

The mapping between the object library and the relational database assumes, respectively defines the structure of the database. The class hierarchy derived from the *PersistentObject* class is stored in many tables. To make the mapping easy, the library is able to generate the SQL script that creates all needed database objects in the needed format.

There are four methods for DDL script generation inside the library.

public: virtual bool Database::WriteDDL(ofstream &S);
public: virtual bool ProtoBase::WriteDDL(ofstream &S, class Database &Db);

[8] Oracle RDBMS type name for VARIABLE CHARACTER

public: virtual bool Relation::WriteDDL(ofstream &S, class Database &Db);
public: virtual bool ClassRegister::WriteDDL(ofstream &S, class Database &Db);

All of them take open output stream and write needed SQL statement to it. Database generates database objects needed for generation of object identifiers and serial numbers. As this depends on particular database provider, each new database support should redefine this method as needed. The Oracle database uses two sequences that are created by the script. Sequences are used in two protected methods

virtual bool Connection::_NextOID(long int &nxtoid, long int &nxtsn);

and

virtual bool Connection::_NextSN(long int curoid, long int &nxtsn);

that returns new couple of OID and serial number for new persistent instance, respectively new serial number for changed instance.

Note: There is a naming conflict between the C++ fstream implementation and the Oracle OCI implementation. Both of them define identifier "text" in different way. To solve this conflict, definition of OCI type text in oratypes.h must be commented out. If necessary, all other usage of text type must be replaced by the oratext type, which is equivalent to original one.

Each prototype of class derived from the *PersistentObject* class generates CREATE TABLE statement that creates needed table to store instances of the class. To use corresponding database types for columns, the *Database* class defines method

public: virtual char *Database::ColumnTypeDDL(char coltype, int colsize);

that provides correct columns types for given attribute type and length. The *OracleDatabase* class returns "VARCHAR2(20)" for TYPE_STRING of length 20, "CHAR(1)" for TYPE_CHAR etc. Other databases can redefine this method as necessary.

Each instance of Relation descendant generates statements that create or alter some table or tables to be able to store pairs of related instances. Chained relations are not stored directly in the database. Thus, they generate no statement. To use corresponding database types for columns, they use method

public: virtual char *ProtoBase::ColumnTypeDDL(int i, class Database &Db);

that provides correct columns types for i-th attribute inside the class.

To make SQL code generation easier, the *ClassRegister* class generates SQL CREATE TABLE statements for all registered persistent classes in correct order. Table associated with particular class is created after all tables associated with all its predecessors.

5 Conclusion

Thesis provides complete solution of interoperability between object-oriented language and relational databases, which are, and in near future probably stay, the main storage system. It consists of proposal and pilot implementation of the persistent object library. The solution offers comparable functionality on relational databases as the ODMG standard. It provides easy-to-use application interface, which significantly decreases the impedance problem.

The library design comes out of own, in the thesis thoroughly described, object model. Proposed object model supports most of advanced features as multiple inheritance and abstract classes. Only member attributes are limited mostly on atomic values that can be stored in the database. With some extra effort on side of the developer the application can define also structures and arrays inside persistent classes. In this case the mapping of structured class instance onto flat relational table is less straightforward, but not impossible. Persistent classes defined inside other persistent classes were not considered in the object model and are not supported. In case the application should define persistent classes as members of another classes, the application design must be modified to use either database pointers or associations instead.

The mapping between C++ objects and database tables is implemented through the object definition language that combines capabilities of standard C pre-processor with the syntax of SQL SELECT statement. Description of mappings is included directly into class declaration part of the source code and so both data and mapping definition are visible at the same place. The simplicity of the definition makes possible to define the mapping by hand. Another possibilities are to write program that will parse the C++ headers and adds necessary code automatically.

The programmer gets simple but powerful query language derived from relational algebra. In case of C++ the query language uses its overloaded operators and so its expressions can be used in a natural way known to programmers. However, query language, based on relational algebra is not so strong as OQL based on SQL. Common queries that allow searching instances according their inner values, obtaining directly or even indirectly associated instances or retrieving particular instance are directly supported. Suggested and implemented operators defined on classes and association allow formulating of broad range of most frequent queries. Very complex queries that cannot be directly expressed in relational algebra are uncommon. Nevertheless such queries can be still formulated in SQL WHERE clause fragment, passed to the *Query* instance. Owing to possibility to access legacy tables through provided unified interface, the complexity of the query can be easily hidden in the database in form of view.

Although the query language was designed mainly for manipulation on object hierarchies, it is uniformly extended to work also on regular database tables with arbitrary primary keys. This feature allows applications to create own persistent classes and simultaneously access and manipulate older relational data produced by another database applications. It is allowed to make associations between both types of classes without limitation. The proposed query language has strong instrument at disposal concerning associations. It allows easy retrieving indirectly associated instances. For this purpose the programmer can use chaining operators together with both-sided restriction.

Both the object model and the query language are fully independent on concrete database server provider. The library implementation itself separates database dependent code and allows easy portability on different databases.

The object buffer implemented inside the object library the database instance appears similar to virtual memory to the application. Whenever the application accesses the object, it is transparently retrieved from the database. The application need not care about the SQL code generation.

Transactional processing allows keeping data consistent in multi-user environment. User has possibility to widely change the data sharing strategies and to gain either high level of consistency and

security, or higher speed of application. Balancing of performance can be done on more levels starting with the database, continuing through connection level and ending by setting of individual rules for each particular object instance.

Proposed class hierarchy in the library unifies behaviour of persistent classes together with database pointers, queries and its results. This approach simplifies the application interface and allows better optimisation of the application. Programmer can do almost all database operations directly on instances, as well as on its pointers. The object instances need not often be in the memory at all. Objects and its database pointers can stand in place of query, which is helpful mainly in work with associated objects. Another advantage of this solution is representation of results in form of pointer to current retrieved instance in the result set. This approach combines database cursors with behaviour of C++ arrays that correspond to pointers.

The openness of the object model proposal, query language and the implementation make future extensions and modifications easy. It is supposed that this thesis becomes the good base of further research.

One of main topics that the future development should provide is the extended object model incorporating ternary associations. This feature was not in the centre of interest, because the standard object model doesn't work with concept of ternary association. Object-oriented languages work with uni-directional pointers and even bi-directional binary associations must be modelled using pointers or collections of pointers. The programmer of object-oriented application is responsible for keeping binary associations consistent, although object oriented databases can take over this responsibility. Ternary and more-ary associations are not directly supported at all. Both entity-relationships models and relational databases on the other hand use and handle binary, ternary or more-ary associations with ease. Implementing of ternary associations in object algebra will allow direct modelling of associations with parameter as well as associations similar to those between producers, consumers and traded commodities.

The automatic SQL code generation supposes the table per class mapping of classes onto tables. Type of mapping was intentionally projected separately. Another of evolving steps would be the extension that allows different mapping schemas for different sub-graphs or branches in the class hierarchy graph.

References

[Ca93] *Cattel, R.G.G. (Ed.): The Object Database Standard: ODMG-93. Morgan Kaufman Publishers, 1993.*

[Ca94] *Cattel, R.G.G.: Object Data Management. Revised Edition, Addison-Wesley Publishing Company, 1994.*

[Ca00] *Cattell, R.G.G.: The Object Data Standard: ODMG 3.0. Morgan Kaufman Publishers, 2000.*

[Ch96] *Chamberlin, D.: Using the New DB2: IBM's Object-Relational Database System. Morgan Kaufman Publishers, 1996.*

[De97] *Dedecek, D., Dusek, L., Fedacko, V., Gatnar, P., Kutmon, T., Pavelka, J., Stourac, D., Wolny, J.: WISE Software Specification Report, ADOORE Consortium, DCIT, Prague, 1997.*

[Ha95] *Harmon, P.: Tutorial 7. ObjectWorld Conference, Frankfurt, Germany, October 9, 1995*

[Ka94] *Kalman, D.M.: DBMS Interview - December 1994. DBMS, December, 1994.*

[Ko96a] *Kopecky, M., Pokorny, J.: GEN.LIB Specification Report, ADOORE Consortium, Charles University, Prague, 1995*

[Ko96b] *Kopecky, M., Pokorny, J.: GEN.LIB Design Report, ADOORE Consortium, Charles University, Prague, 1996*

[Ko97] *Kopecky, M., Pokorny, J.: Objects Through Relations: the ADOORE Approach September 1997. Plenum Publishing Corporation, New York, 1997.*

[Me94] *Melton, J.: The „What's What" of SQL3. Database Programming and Design, Vol. 9, No. 12, 1996, pp. 66-69.*

[Or96] *ORACLE: Introduction to Oracle 8. Server Technologies WW Marketing Summit, September, 1996.*

[Pr91] *Premeriani, W.J. et al: An Object-oriented Relational Database. CACM, Vol. 33, No. 11, 1990, pp. 99-109*

[Ru91] *Rumbaugh, M., et al: Object-Oriented Modeling and Design. Prentice Hall, 1991.*

[St96] *Stonebraker, M., Brown, P.: Object-Relational DBMSs: The Next Great Wave. Morgan Kaufman Publishers, 1996.*