C++ Object Persistency Using Object/Relational Databases

Petr Čermák

March 8, 2009

Contents

1 Introduction **1**

2 Persistence layer requirements **4**
2.1 The identity of persistent objects 4
2.2 Database mapping requirements 5
2.3 Object-relational databases 9
2.4 Database mapping requirements continued 14
2.5 Querying 15
2.6 Caching 15
2.7 Reflection 16
2.8 Library architecture 16
2.9 Conclusion 17

3 Evolution of the IOPC 2 library **19**
3.1 POLiTe 19
3.1.1 Architecture 19
3.1.2 The data access layer 20
3.1.3 Metamodel and object-relational mapping. 21
3.1.4 Persistent object manipulation 24
3.1.5 Querying 27
3.1.6 Conclusion 28
3.2 POLiTe 2 29
3.2.1 The cache layer 29
3.2.2 Using the persistent objects 30
3.2.3 Conclusion 31
3.3 The IOPC library 31
3.3.1 Features of the library 31
3.3.2 Architecture of the library 31
3.3.3 Conclusion 34

4 Literature used **37**

List of Figures

1.1 IOPC 2 evolution 3

2.1 Example class hierarchy 6
2.2 Vertical mapping tables 7
2.3 Horizontal mapping tables 8
2.4 Filtered mapping tables 8
2.5 Combined mapping tables 9
2.6 Proposed architecture of the O/R mapping library 17

3.1 Architecture of the POLiTe library 20
3.2 POLiTe persistent object states 25
3.3 Architecture of the IOPC library *TODO: predelat obr., pripsat OpenC++...* 32
3.4 Structure of the IOPC LIB 34

Anotace

Anotace *TODO: doplnit*

Chapter 1

Introduction

Today, object-oriented languages represent standard instruments for business application and information system development. These systems usually operate with large amounts of persistent data stored in relational database management systems (RDBMS). Data in RDBMSs are however represented differently from data in the application layer. Developers need to do a lot of programming overhead to deal with this so-called *impedance mismatch* everytime they want to move data between relational databases and application-level object models.

Object-oriented database management systems (OODBMS) try to mitigate the impedance mismatch for example by providing navigation using pointers instead of using joins as in the relational databases. Despite theirs advantages the object-oriented databases are not as widely used as the relational databases. Mostly because of the lack of various tools like reporting, or OLAP[1] and due to the industry standards pushed by the big players - Oracle, Microsoft and IBM. Moreover many of RDBMS creators already addressed the impedance mismatch issue by incorporating object-oriented features into their products. Doing it a new kind of database management system, object-relational database management systems (ORDBMS), were created.

Another approach to bypassing the impedance mismatch is to isolate the developer from direct data manipulation in the database at application level. This goal can be accomplished by using an object-relational (O/R) mapping layer. This layer transparently maps relational data into application object model and vice versa. The O/R tools usually offer additional services like object querying or caching.

Current information systems are often written in high-level languages like Java or C#. There exist established O/R mapping tools for these environments. Well-known is Hibernate[2] for Java and nHibernate[3] for C# or relatively new ADO.NET Entity Framework from Microsoft[4].

[1] see [2]
[2] http://www.hibernate.org/
[3] http://www.nhibernate.org/
[4] http://msdn.microsoft.com/en-us/library/bb399572.aspx

O/R mapping tools for such languages can use their feature called *reflection*. Reflection allows program to find out information about its own data model and to modify it at run-time. The O/R mapping layer then can easily inspect the structure of the classes being mapped, and based on this information, it can transparently load or save data from/to the underlying database. Langauges that support reflection are often referred to as *reflective languages*.

Another frequently used language in this area is C++. Unfortunately, C++ is not a reflective language, so the building of O/R mapping layer is a bit more difficult. The goal of this thesis is to develop such a mapping library, which should work as transparently as possible. *TODO: A good O/R mapping library can help C++ application developers to focus on other areas than on complicated database access and can move the C++ development towards the higher level reflective languages. To achieve this goal the library uses GCCXML, a XML output extension to GCC. GCCXML helps the O/R layer to get a description of the class model used in the program and to simulate the reflection. - do Abstraktu* Another goal of this thesis is to examine possibilities the ORDBMSs can provide to a O/R mapping library.

One of goals of this thesis was to use advantages of three previous projects.Their common predecessor is the POLiTe library[5] was developed as a part of doctoral thesis [4]. Two follow-ups came after this work as master theses focusing on different areas of the O/R mapping concept:

- Master thesis [1] (called POLiTe 2 in further text) addressed mainly the performance and notably enhanced functionality of the object cache. It also added multithreading support and made the interface of the library safer to use.

- Mater thesis [3] (IOPC[6]) designed a new persitence layer. The main advantage of this new layer is transparent application development without the need to additionally describe classes in it. It uses OpenC++ source-to-source translator to analyze and prepare the source code for the object-relational mapping. Even though brand-new interface was created, the library still supports classes written in the POLiTe-style.

[5] Persistent Object Library Test

[6] Implementation of Object Persistency in C++

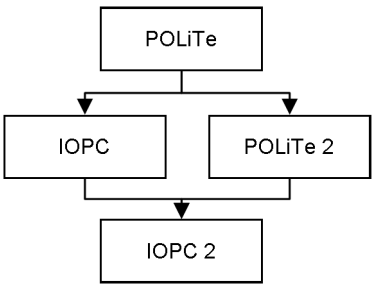

Figure 1.1: IOPC 2 evolution

The IOPC 2 library provided in this thesis not only merges the development back into one product offering most of previously implemented features without their drawbacks. Furthermore, the library implements new ideas like standalone reflection mechanism, additional mapping type using object-relational database abilities and many other. *TODO: It should represent a solid, flexible and extensible platform for use and for future development. - do zaveru*

In the next *TODO: two* chapters we will introduce basic concepts of the object-relational mapping, discuss new features of ORDBMSs and describe requirements ang goals of the IOPC 2 implementation. Then, in the *TODO: fourth* chapter, evolution of the IOPC/POLiTe libraries will be presented in the context of requirements placed and their features will be compared with the new IOPC 2 implementation. In conclusion we will evaluate the achievements of this thesis and will propose areas for further development. Appendices contain a user guide *TODO: and additional overview tables* .

Enclosed DVD contains library source code with examples, binary distribution for linux, documentation and in the first place a VMWare image with pre-installed environment. The image contains Ubuntu Linux, freely distributable Oracle XE database, latest GCCXML and all other IOPC 2 dependencies. The library source code and source code of the examples is stored in the image as an Eclipse CDT project. So all the examples can be modified, compiled and run right away.

TODO: Predelat citations - precislovat

Chapter 2

Persistence layer requirements

Sections in this chapter analyze the requirements that may be imposed on a O/R mapping library and were taken into account during design and implementation phase of the development.

2.1 The identity of persistent objects

In the world of object-oriented programming object identity represents an object property that helps to distinguish objects from each other. Even if two distinct objects have same values of all their attributes and so their inner state is identical, they are still different instances with different identity. A reference to an object is a closely related term to the identity as it uses this identity to describe the object it is referring to.

If we consider entries in a relational table as objects, the identity of these objects could be based on any key in the table - usually the primary key. Persistence layer would then use such a key as a description of object identity on the application level.

In object-oriented systems, where database is used only as a mere storage of the object model and/or the design focuses on the application-tier, an object identifier (*OID*) approach is used. OID is a name for a special table column and for corresponding class attribute which has no business meaning. On the application level it is usually hidden from users or application developers. OID contains an identifier, usually a number, UUID[1] or a list of numbers, which is unique for each persistent object within the database scope. Object's OID never changes during its lifetime. OID is mapped into database tables as a surrogate key. Instances of classes containing an OID attribute are called *OID objects* further in the text.

Another approach is needed for systems built upon an existing database, for systems where the use of natural, not surrogate, keys is required or for systems with read-only or no-schema-changes-allowed databases. The object-relational

[1] A Universally Unique Identifier

layer should be able to absorb identity of persistent objects from keys (even multi-column keys) in existing schema. We will call such objects as *database objects*.

In several cases, we may want the object-relational layer to manipulate objects without any identity. These objects may represent results of aggregation queries, rows from non-updateable views or rows from tables without any keys. *Transience* is an important feature of these objects: their modified state cannot be stored back to the underlying database - origin of the data they contain may not even be tracable back to particular row and/or particular table.

The all-purpose O/R mapping layer should support OID objects as well as database objects and even transient objects for query results.

2.2 Database mapping requirements

Persistence layer considered in the context of this thesis should be able to manipulate persistent instances of certain classes. These classes are called *persistent classes*, its instances *persistent objects*. Storing objects to database implies that the layer should store their attribute values to underlying database structures. Because all predecessors of this thesis used relational database systems as their persistent storage, let's focus first on this area first.

Persistent classes would be represented as tables and their attributes as their columns. Instances of persistent classes would be inserted into these tables as rows containing instance attribute values associated with corresponding table columns. Requirements on a basic persistence library could be:

- Ability to associate persistent classes with database tables
- Ability to map attributes of these classes on columns in associated tables. This means that the layer should be able to store attributes of certain C++ types (basic numeric types, strings) into the database as column values.

  Classes can also contain attributes of structured types or collections, which are often mapped into separate tables or split into more columns in the relational model.

  Last attribute type to be discussed is an association (C++ pointer/reference). Association can be modeled using foreign key relationship between matching tables. The persistence layer should be able to handle single associations as well as collections of associations.
- An optional requirement may be an ability to generate required database schema in form of a SQL create (or drop) script. The persistence layer may require its own structures in the underlying database or it may be able to operate upon existing database schema.
- Ability to query subsets of object model content. The layer should provide a query language that would abstract from the physical representation of the object model in the database.

Up to now we have considered only single classes without inheritance relations. However, in C++ classes can form complex inheritance hierarchies and it is a natural requirement to be able to store descendants of persistent classes too. There are several ways how to store these hierarchies into a relational database. To help to illustrate these *mapping types* see the Figure 2.1 for example class hierarchy. This hierarchy will also be used and modified further in the text.

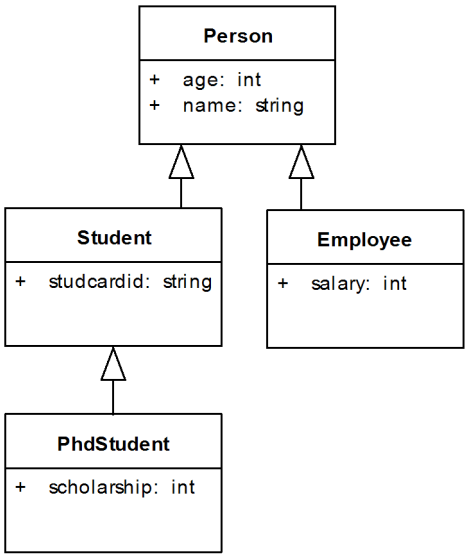

Figure 2.1: Example class hierarchy

Vertical mapping is a most common (and natural) way of mapping attributes of persistent classes in an inheritance hierarchy into tables in a relational database. Each class in this hierarchy has one associated table in the database. Only values from attributes declared in correspondent classes are stored into these tables. This means that attributes declared in current class are mapped into its associated table, attributes from parent class are mapped into a "parent" table etc. Storing one object invokes a cascade of database inserts. Similar rules apply for updates, deletes and selects. However, selects can be simplified using table joins and database views. This solution offers good performance for shallow hierarchies, which is getting worse with the inheritance graph getting deeper. It is a best choice for scenarios where polymophism does matter - by querying one table we easily get instances of associated class and its descendants.

Let's consider following instances of the classes from Figure 2.1:

```
Student(name: "Richard Doe", age: 22,
  studcardid: "WCD-3223")
PhdStudent(name: "Joe Bloggs", age: 27,
  studcardid: "PHD-1234", scholarship: 12000)
```

```
Employee(name: "Ola Nordmann", age: 45,
  salary: 60000)
Person(name: "Mary Major", age: 60)
```

Vertical mapping will spread data from these instances into four tables as illustrated by the Figure 2.2. The tables are arranged so that attribute values belonging to one particular class are displayed in the same row. To be able to join data from the tables we use surrogate OID as explained in the previous chapter. All rows belonging to one particular object are assigned the same OID.

Person

OID	**Name**	**Age**
1	Mary Major	60
2	Richard Doe	22
3	Joe Bloggs	27
4	Ola Nordmann	45

Student

OID	**StudcardId**
2	WCD-3223
3	PGS-1234

PgsStudent

OID	**Scholarship**
3	12 000

Employee

OID	**Salary**
4	60 000

Figure 2.2: Vertical mapping tables

Horizontal mapping offers better performance for scenarios where we don't need polymophic queries - accessing descendants of specific class. Again, each persistent class in a hierarchy has one associated table into which its instances store their attributes. The difference from vertical mapping is that these tables contain even attributes inherited from parent persistent classes. Rows in these tables contain enough information to load complete persistent class instances, thus no cascade operations are needed. Every instance of horizontally mapped class is mapped only into one table row in the database.

As you can see in Figure 2.3 - queries using polymorphism can be very hard to perform. Finding a specific object of Person type or its descendants involves looking into all the tables. However, opposite to the vertical mapping, if we work with objects of specific type (not including descendants), we don't need any joins in select statements or cascade inserts/updates.

Filtered mapping assigns only one database table to all persistent classes in one inheritance hierarchy - see Figure 2.4. This table contains columns that represent all attributes from all classes in that hierarchy. Filtered mapping doesn't suffer from disadvantages of two previous approaches - it performs wery well on polymorfic data and doesn't involve cascaded operations/joins. A disadvantage of this approach is excesive storage requirement. Most of the rows in the table will contain empty cells in columns that belong to attributes from classes not being in ancestor relationship of the matching class or from the matching class itself. Second thing is that it is necessary to add a column telling us which class the rows belong to.

Person

OID	Name	Age
1	Mary Major	60

Student

OID	Name	Age	StudcardId
2	Richard Doe	22	WCD-3223

PgsStudent

OID	Name	Age	StudcardId	Scholarship
3	Joe Bloggs	27	PGS-1234	12 000

Employee

OID	Name	Age	Salary
4	Ola Nordmann	45	60 000

Figure 2.3: Horizontal mapping tables

As we have all rows in one table there is no other easy way how to distinguish the instance types.

Person

OID	Name	Age	StudCardId	Scholardhip	Salary	Class
1	Mary Major	60	NULL	NULL	NULL	Person
2	Richard Doe	22	WCD-3223	NULL	NULL	Student
3	Joe Bloggs	27	PGS-1234	12 000	NULL	PgsStudent
4	Ola Nordmann	45	NULL	NULL	60 000	Employee

Figure 2.4: Filtered mapping tables

Combined mapping is a combination of mappings mentioned above. It allows users to use all kinds of mappings in one inheritance hierarchy. Combined mapping is the most sophisticated variation that allows users to specify these mappings according to their needs. It is also quite complex for implementation, it has some constraints how the mapping types can be used and it is not well maintainable on the database side. Database structures created for combined mapping would require nontrivial constraints if their content were modified other way than using

the persistence layer that created the structures. The persistence layer should hide this complexity beyond views to provide at least convenient read-only access. The mapping algorithm used in the IOPC 2 library will be described later in [?]. To see how the combined mapping can be used refer to Figure 2.5.

Person

OID	Name	Age
1	Mary Major	60
4	Ola Nordmann	45

Employee

OID	Salary
4	60 000

Student

OID	Name	Age	StudcardId	Scholarship	Class
2	Richard Doe	22	WCD-3223	NULL	Student
3	Joe Bloggs	27	PGS/1234	12 000	PgsStudent

Figure 2.5: Combined mapping tables

The `Employee` class uses vertical mapping, the `Student` class uses horizontal mapping and the `PhdStudent` class uses filtered mapping. Classes that use filtered mapping can choose into which of their ancestors they will be mapped to. Class `PhdStudent` is mapped to the table table belonging to the `Student` class.

2.3 Object-relational databases

Object-relational databases offer higher level of abstraction over the problem domain. They extend relational databases with object-oriented features to minimise the gap between relational and object representation of application data known as the impedance mismatch problem. One of the features allows developers to create new custom data types and extend them with custom functions. Main features of the object-relational databases are summarised below. For detailed information about user defined types and other features of object-relational databases refer to [8], [9]. This area is introduced by an 1999 revision of the ISO/IEC 9075 family of standards, often referred to as SQL3 or SQL: 1999. Because the level of implementation of the standard varies between available products, you may need to see their manuals too. For Oracle 10g refer to [10].

User-defined types are custom data types which can be created by users using the new features of object-relational database systems. These types are used in table definitions the same way as built-in types like NUMBER or VARCHAR. There are several kinds of UDTs - for example - distinct (derived) types, named

row types, and most importantly the *abstract data types* (ADT), which we will focus on in the following paragraphs.

ADT is a structured user defined type defined by specifying a set of attributes and operations much in a similar way to object-oriented languages like C++ or Java. Attributes define the value of the type and operations its behaviour. ADTs can be inherited from other abstract data types (in terms of object-oriented programming) and can create type hierarchies. These hierarchies can reflect the structure of data objects defined in application-tier modules. Instances of ADTs are called objects and can be persisted in database tables. See Example 2.3.1 for an illustration how these types are defined and used in Oracle ORDBMS.

Example 2.3.1 Using object types in Oracle

```
-- type definitions
CREATE TYPE TPerson AS OBJECT (
  name VARCHAR2(50),
  age NUMBER(3)
  ) NOT FINAL;
CREATE TYPE TStudent UNDER TPerson(
  studcardid VARCHAR2(20)
) NOT FINAL;
CREATE TYPE TPhdStudent UNDER TStudent(
  scholarship NUMBER(10)
) NOT FINAL;
CREATE TYPE TEmployee UNDER TPerson(
  salary NUMBER(10)
) NOT FINAL;

-- storage
CREATE TABLE Person OF TPerson;

-- fill it with data
INSERT INTO Person VALUES(
  TPerson('Mary Major', 60)
);
INSERT INTO Person VALUES(
  TStudent('Richard Doe', 22, 'WCD-3223')
);
INSERT INTO Person VALUES(
  TPhdStudent('Joe Bloggs', 27, 'PHD-1234', 12000)
);
INSERT INTO Person VALUES(
  TEmployee('Ola Nordmann', 45, 60000)
);
```

First, the supertype `TPerson` and its descendants `TStudent`, `TPhdStudent` and `TEmployee` are defined. The NOT FINAL keyword allows us to create

subtypes of given types. Then physical storage table `Person` is created. This table can hold not only instances of the `TPerson` type but also instances of its descendants. Accessing these instances is demonstrated in the following Example 2.3.2.

Example 2.3.2 Accessing objects in Oracle

```
SELECT VALUE(x) FROM Person x;
-- returns:
TPERSON('Mary Major', 60)
TSTUDENT('Richard Doe', 22, 'WCD-3223')
TPHDSTUDENT('Joe Bloggs', 27, 'PHD-1234', 12000)
TEMPLOYEE('Ola Nordmann', 45, 60000)

-- accessing descendant attributes:
SELECT
  TREAT(VALUE(x) AS TStudent).studcardid AS studcardid
FROM Person x
WHERE VALUE(x) IS OF (TStudent)
-- returns:
WCD-3223
PHD-1234
```

First, we list all objects stored in the `Person` table. Then we list all student card IDs of all student objects that are stored in the table.

Nested tables. Nested tables violate the first normal form in a way that they allow the standard relational tables to have non-atomic attributes. Attribute can be represented by an atomic value or by a relation. Example 2.3.3 illustrates how to create and use nested tables in Oracle database system. The example modifies the `Person` type by adding a list of phone numbers to it. Interesting is the last step in which we perform a SELECT on the nested table. To retrieve the content of the nested table in a relational form, the nested table has to be *unnested* using the TABLE expression. The unnested table is then joined with the row that contains the nested table.

Example 2.3.3 Nested tables in Oracle

```
-- a type representing one phone number
CREATE TYPE TPhone AS OBJECT (
  num VARCHAR2(20),
  type CHAR(1)
  );

-- a type representing a list of phone numbers
CREATE TYPE TPhones AS TABLE OF TPhone;

-- the modified TPerson type
CREATE TYPE TPerson AS OBJECT (
  name VARCHAR2(50),
  age NUMBER(3),
  phones TPhones
  ) NOT FINAL;

-- storage
CREATE TABLE Person OF TPerson
  NESTED TABLE phones STORE AS PhonesTable;

-- fill it with data
INSERT INTO PERSON VALUES(
  TPerson('Mary Major', 60, TPhones(
      TPhone('123-456-789', 'W'),
      TPhone('987-654-321', 'H')
) ) )

-- obtaining a list of phones of a particular person
SELECT y.num, y.type
FROM Person x, TABLE(x.phones) y
WHERE x.name = 'Mary Major';

-- returns rows:
123-456-789 W
987-654-321 H
```

Please note, that the nested table in Example 2.3.3 may need an index on an implicit hidden column *nested_table_id* to prevent full table scans on it.

Collection types. SQL3 defines also other collection types like sets, lists or multisets. In addition to nested tables, Oracle implements the VARRAY construct which represents an ordered set (list). The main difference is that VARRAY collection is stored as a raw value directly in the table or as a BLOB[2], whereas nested table values are stored in separate relational tables.

Reference types. We can think of the database references as of pointers in the

[2]Binary large object

C/C++ languages. References model the associations among objects. They reduce the need for foreign keys - users can navigate to associated objects through the reference. In the following Example 2.3.4 we will add a new subtype `TEmployee` and modify the `TStudent` type from previous examples by adding a reference to the student's supervisor, which is an employee, to it. Note that we need to cast the reference type REF(x) to REF `TEmployee` in the INSERT statement because REF(x) refers to the base type `TPerson`.

Example 2.3.4 References in Oracle

```
-- type definitions
CREATE TYPE TPerson AS OBJECT (
  name VARCHAR2(50),
  age NUMBER(3)
  ) NOT FINAL;
CREATE TYPE TEmployee UNDER TPerson(
  salary NUMBER(10)
);
CREATE TYPE TStudent UNDER TPerson(
  studcardid VARCHAR2(20),
  supervisor REF TEmployee
);

-- storage
CREATE TABLE Person OF TPerson;

-- insert an employee into the Person table
INSERT INTO Person VALUES(TEmployee('Ola Nordmann', 45,  ↩
   60000))
-- insert a student with a reference to his supervisor
INSERT INTO Person
  SELECT TStudent('Richard Doe', 22, 'WCD-3223',
    TREAT(REF(x) AS REF TEmployee)))
  FROM Person x
  WHERE x.name = 'Ola Nordmann';

-- select all students with their supervisors
-- dereferencing uses dot notation
SELECT x.name, TREAT(value(x) as TStudent).supervisor. ↩
   name
FROM Person x
WHERE VALUE(x) IS OF (TStudent)

-- returns a row:
Richard Doe, Ola Nordmann
```

2.4 Database mapping requirements continued

As we already know about the object-relational databases, These new features of object-relational databases can be used to enhance functionality of described mapping types. They can also be a basis for a new mapping type that will entirely depend on the use of ORDBMS. First, let's have a look how the attribute mapping can be improved:

- Collections (C++ containers) can be mapped into single columns as nested tables or instances of one of the SQL3 collection data types.
- Structured attributes (C++ struct or class) can be mapped into single columns as instances of SQL3 structured data types.
- Associations (C++ pointers or references) can be mapped as SQL3 references.

Second, let's solve mapping of classes and inheritance hierarchies. It is quite obvious that user-defined types can be used for this task. Abstract data types can be created for each class in the inheritance hierarchy by copying its inheritance graph. Instances of the types can be then inserted into one table. The earlier presented Example 2.3.1 displays structures that may be generated for classes from Figure 2.1 using such kind of database mapping.

This type of mapping is called *object mapping* in the IOPC 2 library. Its benefit is that it moves most of the responsibilities of the persistence layer to the underlying database system. For example obtaining a list of fully-loaded instances of specific type and its descendants involves several joins in the vertical mapping (filtered mappings or other variations using the combined mapping). This must be "planned" by the persistence layer. All such polymorphic queries are best performed using the object mapping (see the Example 2.4.1), as all these tasks can be acomplished only using the user-defined types and SQL3 queries or statements.

Example 2.4.1 A polymorphic query using object-relational features of the Oracle database.

```
SELECT
  name, age, TREAT(VALUE(x) AS TPhdStudent).studcardid AS ↩
      studcardid,
  TREAT(VALUE(x) as TPhdStudent).scholarship as  ↩
     scholarship
FROM Person x
WHERE VALUE(x) IS OF (TPhdStudent)
```

A serious problem for a persistence layer using this object mapping is that there are major differences between database systems in the object-relational area.

For example DB2 doesn't offer any type similar to the Oracle VARRAY data type. Or another example - in DB2 you have to create whole table hierarchy for inherited object types - much like as you would when creating storage structures for a vertically-mapped data type. These issues imply that the persistence layer should be flexible and modular enough to be able to support different database systems.

Another problem is multiple inheritance of ADTs. Although SQL3 standard supports multiple inheritance, it is not implemented neither in the current version of Oracle nor in the current version of DB2. The problem is discussed later in [?]

2.5 Querying

Object-relational systems allow users usually to load data in two ways. Either by traversing the model of persistent objects through associations and letting the persistence layer to load missing data into referenced local copies or by exploiting the ability of the underlying DBMS to execute SQL queries against the data stored in it. The persistence layer can provide direct access to the database by allowing its users to run SQL queries on tables or views the layer generated. This approach is not very user friendly as it requires the users to know the internals of database mapping performed by the persistence layer. The layer should therefore offer its own query language which will hide the complexity of the database structures. Queries in such language can be passed as character strings or as objects which represent attributes, values, comparison criteria etc. Depending on the level of implementation, users may filter only objects of one inheritance hierarchy by their attribute values, perform polymorphic queries returning objects of specified type and its descendants or they may query associations between objects. This queries represented in natural language would be:

- Find all students older than 26. (Age >=26)
- Find all students including Ph.D. students. (Actually all queries may be modified to include Ph.D. students).
- Find all students which are supervised by Mrs. Ola Nordmann. (Using the modified model from Example 2.3.4)

2.6 Caching

The role of caching is to speed up applications that use persistent objects by delaying database mapping operations. This is generally achieved by taking ownership of these objects when they are not currently in use by the user application. If the user applications needs an already released object again, the caching facility[3] looks

[3] *Cache*. All related structures will be called as the *cache layer*.

it up in its catalogue and if found, returns it to the user application, saving the time-consuming database operations. The database operations for storing, updating or loading persistent objects are controlled by the cache layer, not by the user application. The layer is therefore responsible for creating and destroying persistent object instances.

2.7 Reflection

Users of the persistence layer may want to examine the structure of persistent classes at run-time. This is not of a big issue in reflective languages like Java or C#, but in the C++, which is not reflective language, this requirement may pose a problem. Yet not necessarily, because the persistence layer must know about the structure of classes it is mapping to database. So, it only depends on the particular implementation of a C++ O/R mapping library whether it provides access to this information and how.

If the library puts these introspection features behind unified interface and allows to inspect wider set of classes than only the persistent classes, it may provide at least simpler alternative to reflection features offered by reflective languages. This may be a big advantage, because developers tend to include O/R mapping features into their application frameworks and O/R mapping is often one of the pillars of infrastructural part of business applications. Therefore it reduces the need for other reflection library.

2.8 Library architecture

Based on the discussion in previous sections, we are able to specify three relatively autonomous areas a C++ persistence library should cover:

- *Database access*. The library should not be database dependent. To achieve this goal, the database access must be virtualised by providing an interface to other parts of the library which will hide the differences between databases the users may use. The library should contain modules called *database drivers* translating and dispatching requests from the interface to concrete database instances. Database drivers should be separate modules allowing users to select between them without the need to recompile the whole library. The architecture should be flexible enough to be able to handle relational as well as object-relational database systems. It would be also nice if the whole database access infrastructure was a stand-alone module as the discussed database interface could be used as a *database access library*.

- *Reflection*. If the reflection capabilities, as described in the previous section, were provided as a stand-alone module, the library could be used in a *reflection library* configuration.

- *Object-relational mapping*. The complete O/R mapping library would need both, the database access and reflection configurations: The reflection to inspect the structure of the persistent classes and the database access interface to load and store them from/to a database. The library should provide a module which will manage and perform the O/R mapping including related tasks as caching and querying. This O/R mapping module will depend on the previous two modules.

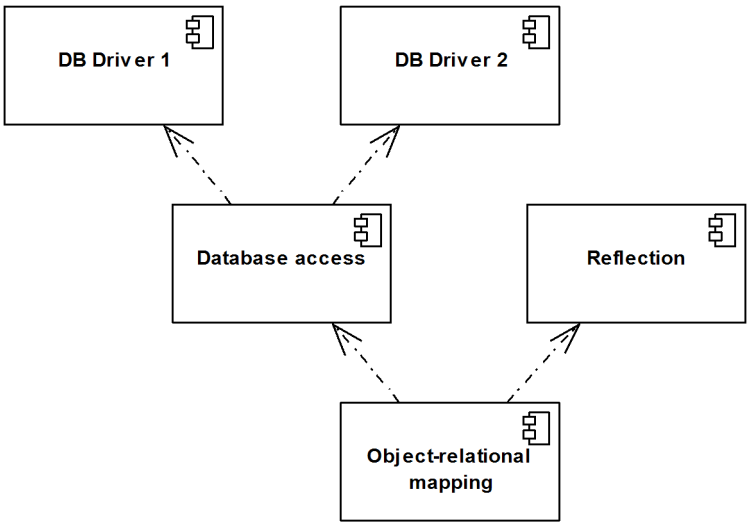

Figure 2.6: Proposed architecture of the O/R mapping library

2.9 Conclusion

In this chapter, we analyzed basic aspects of an O/R mapping library implementation. Based on this analysis, we can summarise the requirements on the library. Some of the requirements are required while some are optional. We list them for further reference.

- Common O/R mapping requirements
  - Ability to associate persistent classes with database tables
  - Ability to map attributes of persistent classes to database columns or attributes in instances of user-defined types
- Ability to use at least one type of O/R mapping (better all of them):
  - Horizontal
  - Vertical

  - Filtered
  - Object
  - Combination of the mapping types
- Ability to persist references between objects
- Ability to persist collections of objects
- Ability to generate required database schema or ability to work with existing non-mutable database schema
- Querying in the object model context.
- Object caching.
- Reflection
- Modular library architecture

Chapter 3

Evolution of the IOPC 2 library

The following paragraphs outline design and functionality of the IOPC 2 library predecessors.

3.1 POLiTe

The common predecessor of IOPC, IOPC 2 and POLiTe 2 libraries - POLiTe represents a persistence layer for C++ applications. The library itself is written in C++. Applications incorporate the library by including its header files and by linking its object code. The library offers following features:

- Persistence of C++ objects derived from specific built-in base classes. Class hierarchies are mapped vertically.
- Persistence of all simple numeric types and C strings (`char*`).
- Query language for querying persistent objects.
- Associations between persistent objects. Ability to combine more associations to manipulate indirectly associated instances.
- Simple database access.
- Common services like logging or locking.

3.1.1 Architecture

Even though the library can be divided to several functional units, it compiles as one shared library. The architecture of the library is outlined in the Figure 3.1. The main functional units are discussed in the following sections.

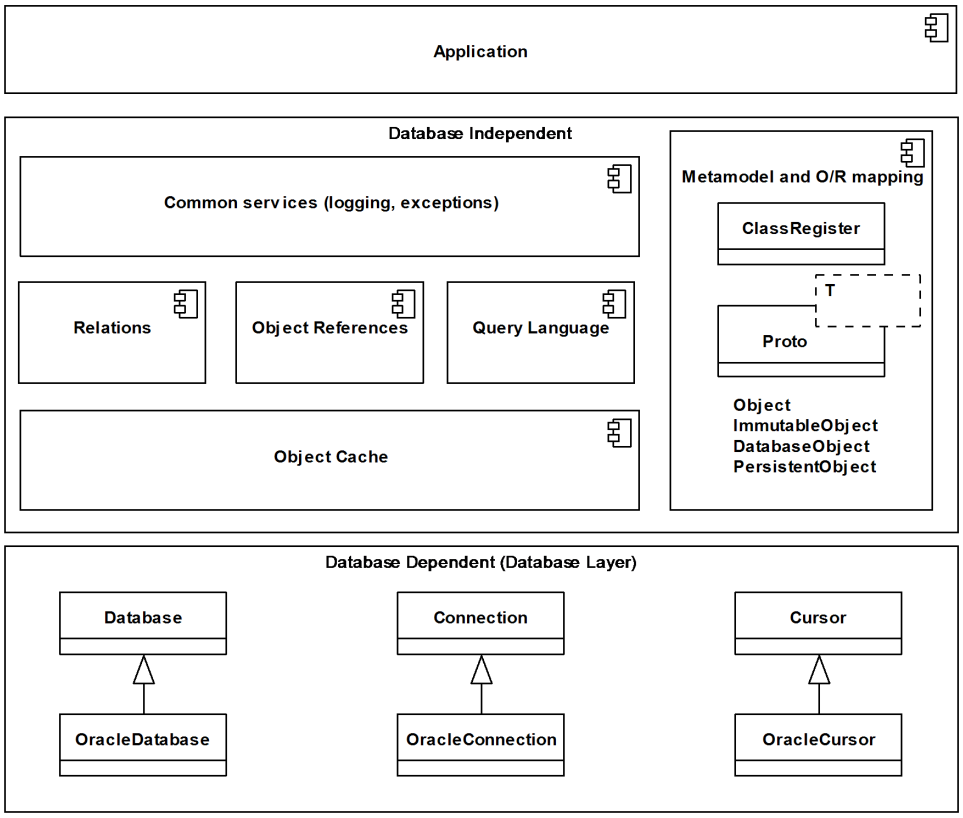

Figure 3.1: Architecture of the POLiTe library

3.1.2 The data access layer

The POLiTe library contains several classes that provide database access. At the time, the library supported the Oracle 7 database and the code used OCI[1] 7 interface to access it. The classes are accessed via common interface that can be used for implementing other RDBMs to the library.

The interface is a set of abstract classes - `Database`, `Connection` and `Cursor`. Communication with the database flows excusively through this interface and its implementation (`OracleDatabase`, `OracleConnection` and `OracleCursor`). The interface `Database` is a logical representation of a database (e.g. an Oracle instance), `Database` can create one or more connections (`Connection`) to the database. The `Connection` inteface is the main communication channel with the database. Using the implementations of the `Connection` interface it is posible to send SQL statements to database and receive responses in the form of cursors (`Cursor`). The response is a set of one or more rows that can be

[1] Oracle Call Interface

obtained from the `Cursor`.

3.1.3 Metamodel and object-relational mapping.

Every persistent class maintainable by the POLiTe library has to be described by a set of preprocessor macro calls. These calls are included directly into the class definitions or near them. Description of the class attributes and the necessary mapping information has to be provided together with declaration of every persistent class. Metainformation covers class name, associated database table, parents, every persistent attribute with its type, table columns and more. For complete list see [5]. Example 3.1.1 displays definition of our classes `Person` and `Student` in the POLiTe library.

Example 3.1.1 Definition of a class in the POLiTe library

```
// Person class
class Person : public PersistentObject
{
  // Declare the class, its direct predecessor(s) ...
  CLASS(Person);
  PARENTS("PersistentObject");
  // ... and its associated table
  FROM("PERSON");
  // Define member attributes
  dbString(name);
  dbShort(age);
  // Primary key OID is inherited from PersistentObject
  // Map other attributes
  MAP_BEGIN
    mapString(name,"#THIS.NAME",50);
    mapShort(age,"#THIS.AGE");
  MAP_END;
public:
};
// Define method returning pointer to the prototype
CLASS_PROTOTYPE(Person);
// Define the solitaire prototype instance Person_class
PROTOTYPE(Person);

// Student class
class Student : public Person
{
  CLASS(Student);
  PARENTS("Person");
  FROM("STUDENT");
  dbString(studcardid);
  MAP_BEGIN
    mapString(studcardid,"#THIS.STUDCARDID",20);
  MAP_END;
};
CLASS_PROTOTYPE(Student);
PROTOTYPE(Student); // Student_class

// Employee class
class Employee : public Person
{
  CLASS(Employee);
  PARENTS("Person");
  FROM("EMPLOYEE");
  dbInt(salary);
  MAP_BEGIN
    mapInt(salary,"#THIS.SALARY);
  MAP_END;
};
CLASS_PROTOTYPE(Employee);
PROTOTYPE(Employee); // Employee_class
```

Because the library needs to track the dirty status of persistent objects, you have to maintain this flag by own or better restrict manipulation with persistent attributes to the use of getter and setter methods defined by the macros. For every class `T` described by these macros there is an associated template class prototype `Proto<T>`. The solitaire instance of this prototype class holds infromation about the metamodel described by the macros and provides the actual database mapping. Prototypes are registered by the `ClassRegister`.

Persistent classes inherit their behaviour from one of four base classes defined in the library - the `Object`, `ImmutableObject`, `DatabaseObject` or `PersistentObject` class. Depending on what the library parent is, several features of the persistence are supported:

- `Object` - instances of descendants of this class can be obtained by database queries. These objects don't have any database identity and can represent results from complex queries containing aggregate functions. More obtained instances can be the same.

- `ImmutableObject` - instances of this class have a database identity mapped to one or more column(s) in the associated table (or view) and represent concrete rows in database tables or views. They can be loaded repetitively, but the ImmutableObject class doesn't propagate changes made to them back to the database. To use this class as a query result, the query has to return rows that match rows in corresponding database tables or views.

- The `DatabaseObject` class is much the same as the `ImmutableObject`, but changes are propagated back to the database.

- The `PersistentObject` class offers the most advanced persistence options from all the classes mentioned above. The `PersistentObject` defines and maintains a unique attribute OID that holds the identity of every `PersistentObject`'s instance within the database. Unlike previous classes, persistence of whole type hierarchies is allowed.

As mentioned, the library offers vertical mapping for descendants of the `PersistentObject`. Tables related to mapped inheritance hierarchies are joined using the surrogate OID key. Object model can be created upon an existing and even read-only database with arbitrary keys. In this case, descendants of the `DatabaseObject` or `ImmutableObject` respectively should be used and no inheritance between them is allowed.

Associations in the POLiTe library are not modeled as references but as instances of the `Relation` class. There are five subclasses of this class - `OneToOneRelation`, `OneToManyRelation`, `ManyToOneRelation`, `ManyToManyRelation` and `ChainedRelation`. Their names describe which kind of relation between the underlying tables they manage. `ChainedRelation` is

built from other relations and it can be used to define relation for indirectly associated objects. Example 3.1.2 demonstrates how a one-to-many relation between the `Employee` and `Student` classes can be created and used.

Example 3.1.2 Associations in the POLiTe library

```
OneToManyRelation<Employee, Student>  ↩
    Employee_Student_Supervisor(
  "EMPLOYEE_STUDENT_SUPERVISOR", dbConnection
);

// Let's suggest that the Employee variable represents
// a reference to a "Ola Nordmann" persistent object
// and Student represents a reference to a "Richard Doe"
// persistent object.
// Create a supervisor relation between "Ola Nordmann"
// and "Richard Doe".
Employee_Student_Supervisor.InsertCouple(
  *Supervisor, *Student
);
```

The relation can be queried for objects on both of its sides. So we may run queries like "Which students are supervised by Ola Nordmann?", "Who is the supervisor of Richard Doe?" or even more complex ones, but that would be out of the scope of this thesis.

The one-to-many relation can be replaced by a reference to the supervisor in the `Student` class definition:

```
...
dbPtr(supervisor);
dbString(studcardid);
MAP_BEGIN
mapPtr(supervisor,"SUPERVISOR");
...
```

Usage of the references is closer to the object-oriented approach in which we navigate using such pointers or references to gain access to the related objects. The drawback is, that the navigation is usually one-way and in this case, the retrieval of all supervised students of an employee is not trivial.

3.1.4 Persistent object manipulation

Persistent objects can enter one of the following states (see the state diagram in Figure 3.2):

- *Transient* - Each new instance of persistent class enters this state. The instance data are stored only in the application memory and are not persisted.

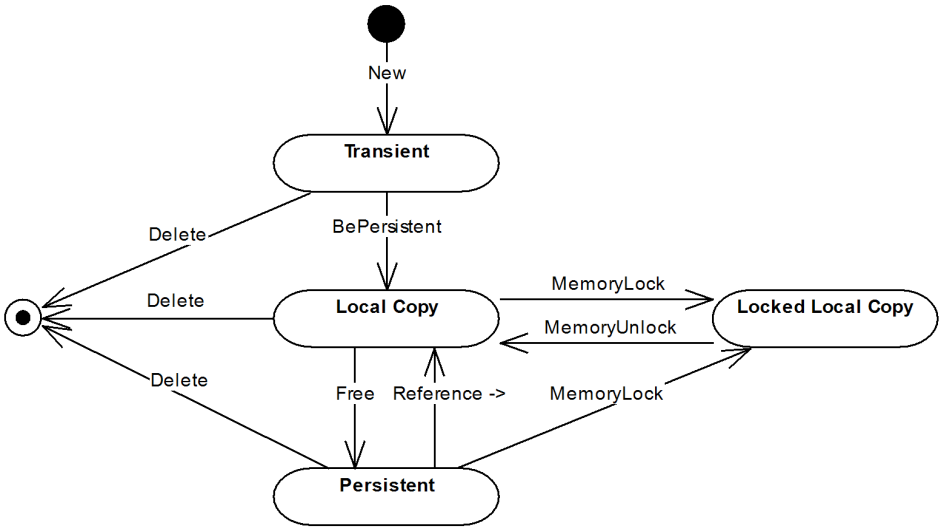

Figure 3.2: POLiTe persistent object states

- *Local copy* - A persistent image of the transient instance can be created by calling the `BePersistent` method. The method inserts attribute values of the instance to the database. The memory instance can be deallocated at any time as it is considered as a cached copy of the inserted database data. This state can be entered also at a later time when loading a persistent instance which has no local copy in the application memory.

- *Locked local copy* - To prevent the local copy deallocation, the local copy can be locked in the application memory. Local copy is not deallocated until its lock is released. After unlocking, the locked local copy enter the local copy state.

- *Persistent instance* - A persistent object can enter this state if its local copy is removed from the application memory. During the state transition, the changes in the local copy are usually propagated to the database. The object exists now only in the database; it can be loaded later and enter one of the local copy states.

Because persistent object can exist in one of those states, library uses indirect references to acces the object's attributes. Users don't have to look after the object's current state and can just access it via the `Ref<T>` reference type. The reference handles required state transitions by itself.

All local copies and local locked copies are managed by the `ObjectBuffer` which acts as a trivial object cache. The buffer is implemented as an associative

container between object identities and local object copies. If the buffer is full, all non-locked local copies are freed and dirty instances updated in the database.

POLiTe allows you to specify several concurrent data access strategies the `ObjectBuffer` will use. These strategies are used to influence the safety or speed of concurrent acces and cached data coherence.

- *Updating strategy* - determines whether changes done to local copies are propagated to the database immediately or they can be deferred.
- *Locking strategy* - determines how the rows in the database are locked when they are loaded into local copies. Shared, exclusive or no locking can be requested.
- *Waiting strategy* - if the application tries to access a locked database resource (by another session), this strategy specifies whether the application waits until the resource gets unlocked or an exception is thrown.
- *Reading strategy* - determines behaviour of the persistence layer if a local copy is accessed using the indirect reference. The local copy can be either used right away or it can be refreshed with the data stored in the database. The refresh option can be speeded up by comparing timestamps of the local copy and of the stored image.

Object manipulation is illustrated by the Example 3.1.3. Two objects - an employee and a student, which is supervised by that employee, are created as transient instances and inserted into the database. The `BePersistent()` call returns a reference to the unlocked local copies of the created objects. Then the salary of the employee is modified and the change propagated to the database. In the end, the student object is deleted from the database and also from the memory.

Example 3.1.3 Persistent object manipulation

```
// Create a new employee
Employee* e = new Employee();
e->name("Ola Nordmann");
e->age(45);
e->salary(60000);
Ref<Employee> employee = e->BePersistent(dbConnection);

// Create a new student supervised by the employee ↩
   created
Student* s = new Student();
s->name("Richard Doe");
s->age(22);
s->studcardid("WCD-3223");
s->supervisor(empl);
Ref<Student> student = s->BePersistent(dbConnection);

// Update the employee's salary
e->salary(65000);
e->Update(); // Propagates the change to the database
// The change could be propagated immediately if the
// updating strategy was set to the "immediate" setting.

// Delete the new student
s->Delete();
);
```

3.1.5 Querying

Queries in the POLiTe library search for objects of a specified class. Search criteria restricting the result set can be specified. Queries are represented as instances of the `Query` class which contains only two data fields: The search criteria, in fact the WHERE part of the final SELECT statement, and a ORDER BY specification which determines how the result will ordered. The search criteria can be written using SQL (referencing physical table and column names) or using a C++-like syntax. The C++-like syntax hides the O/R mapping complexity and allows the users to use more convenient class and attribute names. The query objects can be then combined using the C++ !, && and || logical operators. Example 3.1.4 illustrates how the queries are created, combined and executed.

Example 3.1.4 Queries in the POLiTe library

```
// All employees with salary > 40000
Query q1("Employee::salary > 40000")

// All employees with the first name Ola
Query q2("Person::name LIKE 'Ola %');

// All employees with salary > 40000 having the first  ↩
   name Ola
Query q3 = q1 && q2;

// Order the result by the salary descending.
q3.OrderBy("Employee::salary DESC");

// Execute q3 and iterate through the result
QueryResult* result = Employee_class(q3, dbConnection);
while (++(*qr)!=DBNULL) {
  // process object accessible using (*QR)->
};
qr->Close();
```

3.1.6 Conclusion

The library provides solid and rich-featured ORM solution. However, there are several areas in which the library can be improved:

- *Transparency* - persistent classes have to be precisely described by macros. Typos in this description may lead to unclear compiletime or runtime errors. Attributes must be accessed via the getter and setter methods.
- *Library design* - library is one monolithic block and compiles into one shared library. There is no other way to add additional database drivers or features than changing the makefile and recompiling the library. Same applies to the library configuration - many parameters are configured as preprocessor macros. Changing them implies library recompilation.
- *Database dependency* - without modifications, the library supports only the Oracle platform. Adding new database support supposes to derive new descendants of `Databas`, `Connection` and `Cursor` classes, implement their code and recompile the library. The library also contains several SQL fragments that aren't separated into the database driver layer.

These disadvantages are addressed mostly by the IOPC library [3] and its descendant described further in this thesis. But first, we will look at the performance enhancement provided by the succeeding library POLiTe 2.

3.2 POLiTe 2

The new version of the POLiTe library focuses on the library's performance and on the design of new rich-featured cache layer. The new layer replaces the `ObjectBuffer` interface and enhances the concept of indirect memory pointers by adding one more indirection level.

3.2.1 The cache layer

The cache layer consists mainly of implementaitions of the `Cache` and the `ExtendedCache` interfaces.

- The `Cache` interface defines methods for manipulation with persistent objects and methods to synchronize the cache state with the database.
- The `ExtendedCache` interface extends the `Cache` interface by additional methods that allow asynchronous maintenance of the cache.

Caches can be grouped together using the `ComposedCache` class (which also implements the `Cache` interface). Strategies can be specified for selecting which cache for given object should be used. The updating, locking, waiting and reading strategies can also be specified.

Caches that implement the extended interface can be maintained by the cache manager called `CacheKeeper`. `CacheKeeper` runs an second thread that scans managed caches and removes instances considered as 'worst'. If dirty, these instances are written back to the database. `CacheKeeper` acts also as a facade for composing caches and specifying strategies.

The library contains implementations of several caches which can be used in user applications:

- The `VoidCache` - implements the basic interface `Cache`. This implementation actually doesn't cache anything, but just holds locked local copies. As the copies are unlocked, changes are propagated immediately to the database and objects are removed from the cache.
- The `LRUCache` (multithreaded) and the `LRUCacheST` (singlethreaded) implement the `ExtendedCache` interface. These classes use the LRU replacement strategy ([6]). The multithreaded variant can even be used with the `CacheKeeper`'s asynchronous maintenance feature.
- For the `ARCCache` (multithreaded) and the `ARCCacheST` (singlethreaded) the things are the same with one exception, that they use the ARC replacement strategy ([7]).

3.2.2 Using the persistent objects

In the previous version of the library, transient objects that did not have database representation could be accessed directly via pointers. Indirect references were used after making these objects persistent (the `Ref<T>` template). In the POLiTe 2 library, the creation, descruction of all the objects is managed by the library code and users can acces its members by dereferencing indirect pointers.

The indirect pointer template `Ref<T>` was replaced by the template `DbPtr<T>` (database pointer). The `DbPtr<T>` can point to instances in several states:

- Transient instances which are present only in memory (and do not have a persistent representation yet), the owner of these instances is the pointer.
- In-memory instances owned by a cache (which may already have a database representation, but this is not neccessary).
- Persistent representation of the object. The pointer holds only the indentity of the object.

The `DbPtr<T>` type overloades `->` and $\star$ operators (as the `Ref<T>` did) and thus can be used similarly as standard pointer to the object.

By dereferencing the database pointer using the $\star$ operator user receives an instance of a cache pointer (`CachePtr<T>`). The cache pointer can point to either transient or persistent instances. However the existence of the cache pointer guarantees that the object is loaded into the computer's memory and the pointer contains direct reference to the instance itself. Constructor of the cache pointer asks cache to load relevant object from the database (if not already present in the cache) and obtains a lock of the object from the cache. Locked objects cannot be removed from the cache unless they are unlocked. This task is done by the cache pointer's destructor.

Using the $\star$ operator on the database pointer representing a persistent object user receives a reference to loaded and locked instance in a cache. This reference takes a form of a cache pointer. The cache pointer can be again dereferenced resulting in retrieval of direct pointer to the in-memory instance. This process can be simplified just by using the `->` operator on the database pointer. An implicit instance of the cache pointer is created and the `->` operator repeatedly called on it (this is a feature of the C++ language). Then the desired member of the instance is accessed. After that, the cache pointer is destroyed and cache lock released. (This may result in significant performance degradation if using the VoidCache, because only locked copies are contained by the cache. Calling the `->` operator causes a persistent object to be loaded from database into the memory, then desired member is accessed and object - if modified - is written back and removed from cache).

3.2.3 Conclusion

New version of the POLiTe library allows users to utilize advanced persistent object caching features. Unmentioned remained the support for multithreaded environment which includes encapsulation of several synchronization primitives.

Disadvantages listed in the Section 3.1.6 also apply to this version of the library as they were not the point addressed by the thesis [1].

3.3 The IOPC library

The IOPC library [3] contains a new api for object-relational mapping of persistent objects. This new iterface coexists with the old POLiTe-style interface inside one library and doesn't interfere much with it. The interface is (with few exceptions) clearly divided into two parts - the IOPC and the POLiTe part. Classes written for the first version of the POLiTe library should be usable with this library with minor modifications.

3.3.1 Features of the library

The IOPC library offers all features of the original POLiTe library. New features and main differences are listed below.

- No need to describe structure of persistent classes. Persistence works almost transparently.
- All three basic types of class hierarchy mapping are supported - horizontal, vertical and filtered. Combinations of these types in one class hierarchy are also allowed (with a few exceptions).
- IOPC supports persistence of all simple numeric types and C strings (`char*`, associations and collections of associations. The library allows even to create a new persistent data type.
- Loading persistent object attributes by groups. Persistent attributes can be divided into several groups which can be loaded separately.
- Easy implementation of new RDBMS into the library[2].

3.3.2 Architecture of the library

One of the most interesting aspects of the IOPC library is a new approach how to retreive the persistent class metamodel. User specification of the class structure is not needed (if not still using the POLiTe classes). This task is performed by

[2]However, recompilation of several IOPC modules is still needed

a source-to-source translator OpenC++[3] and its IOPC wrapper IOPC SP (see the library architecture displayed in Figure 3.3).

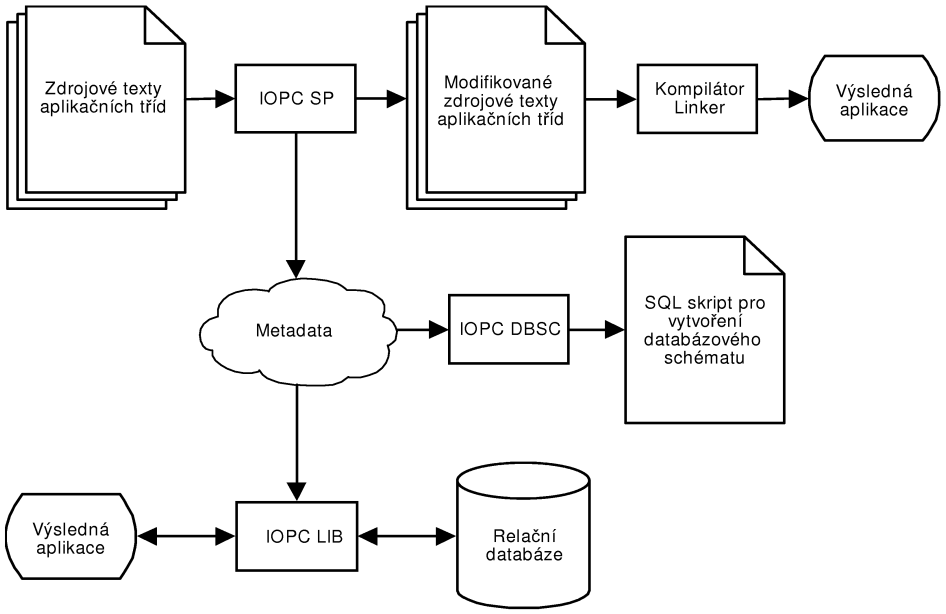

Figure 3.3: Architecture of the IOPC library *TODO: predelat obr., pripsat OpenC++...*

The IOPC library consists of three main parts:

- *IOPC SP* uses the OpenC++ parser to modify the source code to support the object persistence and generates XML metadata describing structure of persistent classes.
- *IOPC DBSC* generates SQL scripts to prepare database for storing persistent objects.
- *IOPC LIB* is library that does the main task - object persistence. The library is linked to the final program, uses metamodel generated by the IOPC SP and database structures created by the scripts from IOPC DBSC.

Loading and storing XML metadata is done by two statically-linked libraries - `XMLMetadataLoader` and `XMLMetadataWriter`. They implement interfaces `MetadataLoader` and `MetadataWriter` respectively. `XMLMetadataLoader` and `XMLMetadataWriter` use the Xerces[4] parser. In order to use

[3] http://opencxx.sourceforge.net/
[4] http://xml.apache.org/xerces-c/

other type of metamodel storage, new implementation of the interfaces has to be created and the MetadataLoader/MetadataWriter libraries recompiled.

IOPC SP is a standalone executable created by linking together modified (patched) OpenC++ source and a new metaclass. The metaclass modifies the source-to-source translation of persistent classes and its references. Tasks performed by this tool are:

- Generates the set and get methods for all persistent attributes. Setters modify the dirty status of the object and getters ensure that corresponding attribute groups are loaded.
- Modifies every reference to persistent attribute to use generated get and set methods.
- Generates remaining members needed for the IOPC LIB to be able to persist the processed class.
- Inspects the processed persistent class and writes information about its structure to XML by calling `MetadataWriter` (its implementation `XMLMetadataWriter`)
- Runs compiler and linker on the translated source code.

IOPC DBSC is a standalone executable that generates SQL scripts for various purposes - script to setup the database, script to drop the created database structures or explain plan script. It uses the `MetadataLoader` to load the persistent class metamodel written by the IOPC SP.

IOPC LIB is a shared library that represents the core of the IOPC project. It is linked to the outputs (object files) of the IOPC SP and other modules and provides the run-time functionality.

The Figure 3.4 shows how the new functionality is integrated to the original POLiTe library.

Database layer was prepared using OCI 8 for the Oracle 8i database, slightly modified original interface was used (the `Database`, `Connection` and `Cursor` classes). The `DatabaseSqlStatements` interface is a new element added to the database layer. Its implementation should generate all SQL statements needed by the library. Again, the Oracle 8i functionality is already present in the library. As we see, the interface and functionality of the database layer remained similar to the POLiTe original.

IOPC persistent classes are now derived from a new base class - the `IopcPersistentObject`. The original base class `Object` and its descendants (`ImmutableObject`, `DatabaseObject` and `PersistentObject`) were preserved. Notice, that the `IopcPersistentObject`'s base class is the original `Object` class.

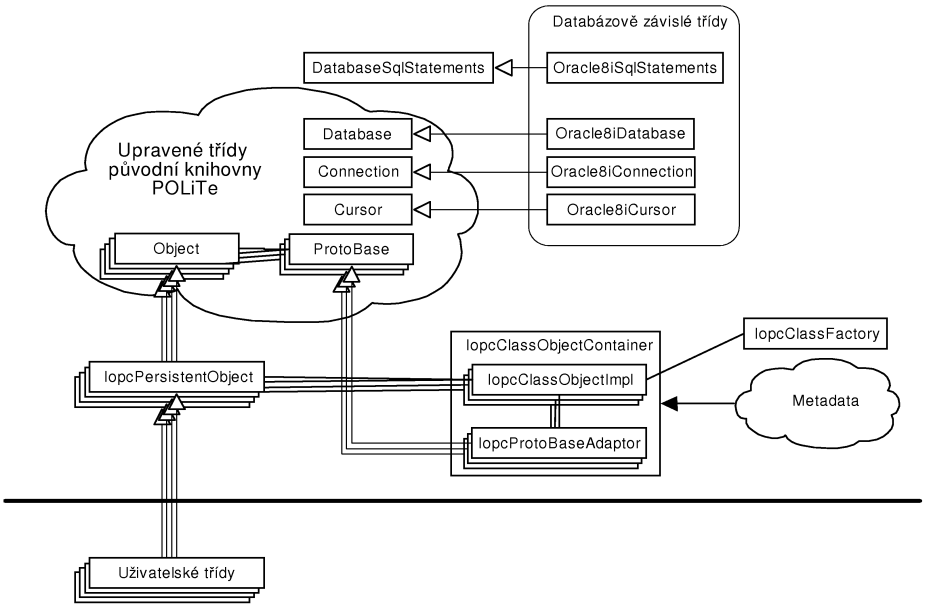

Figure 3.4: Structure of the IOPC LIB

Functionality of prototypes[5] for the IOPC persistent classes is provided by the `IopcClassObjectImpl` class. One instance of this class is created for every persistent type declared in the user code, while the instances of `Proto<T>` prototypes are created for the POLiTe persistent classes. For implementational resons, the IOPC library generates POLiTe-style prototypes for all `IopcClassObjectImpl` instances. Their type is `IopcProtoBaseAdaptor`. `IopcProtoBaseAdaptor` overrides most of the `ProtoBase`'s methods and delegates its work to the `IopcClassObjectImpl` class.

3.3.3 Conclusion

The main goal of the IOPC library was to make the usage of the POLiTe library simpler, more transparent, and to add new features. The architecture of the library was completely redesigned for the sake of these changes. At first glance the IOPC library looks like a big improvement over the original library. But if we look further into the source code and used technologies, we come upon several critical issues that may cause the usage and deployment of this library almost impossible. *TODO: uvest nasl. odstavec*

IOPC uses the OpenC++ as a way to retreive information about the structure of persistent classes. OpenC++ as a project seems to be almost dead, last commit

[5] Prototypes were explained in [?]

to the project's CVS occured in 2005. OpenC++ can't handle most of template constructs, processing files that include GCC's STL headers (tested on versions 3.4 and newer) produces a lot of errors. Because the source-to-source translation is used, users can't be sure if any of their code that was translated with errors (although the IOPC SP states that the errors can be ignored) will do what was intended. *TODO: Zjistit jak to je s STL - jde to vubec zkompilovat?* Probably for this reason the IOPC allows only C strings (`char*` and `w_char*`), not the C++ STL strings (`std::string` and `std::wstring`).

Next, IOPC uses its own modified version of OpenC++ (called 2.6.t.0) and integrates it into the IOPC SP utility. This approach renders difficult further maintenance of the OpenC++ code used in the utility. New changes from the OpenC++ CVS have to be merged manually into the IOPC SP source.

Second point is a question, why there are two parallel interfaces in the IOPC library - one for the new persistent objects and one for the POLiTe-styled objects. If there is no known implementation that uses the POLiTe library, there is no need to be backward-compatible. The POLiTe part of the source code will just remain unmaintained (as no one is supposed to use the POLiTe objects in new applications). Because several IOPC objects inherit from the POLiTe classes, many inherited methods doesn't make sence any more. This makes the API less readable and can lead to user's confusion. An example of this situation is the `IopcProtoBaseAdaptor` which contains a number of methods commented as "Fake function". The presence of two APIs makes usage of the library less clear and less maintainable.

The library itself remained as one monolithic block with no signs of library configurability. Many parameters are defined as preprocessor macros, enabling other options (like adding a new database driver) leads to changes in the source code of the library and recompilation. The design of the library even makes impossible to use more than one database driver.

Bad design of the program interface. Much useful information is hidden in internal structures of the library, these structures are not visible via the library's api. This concerns mostly the data retreived from the `MetadataLoader` - the library could implement some kind of reflection api to be able to query the metamodel.

Wrong and dangerous constructions in the code may cause crashes in the runtime. Two examples:

- Dependencies between statically initialized objects - static instances of the `IopcClassRegistrar` use the statically initialized instance of class `IopcClassFactory`. This may work and may not, it depends on the initialization order of compilation units.
- Inability to use the library in multithreaded environment. Some data structures are reused between persistence layer calls, this prevents to make the library multithreading-friendly without major modifications.

Despite good idea behind the IOPC library, the implementation is deeply flawed, unusable and unmaintainable. For these reasons, the author of this thesis decided

not to continue development upon the source code of this library.

Chapter 4

Literature used

[1] Jan Hadrava, *Správa persistentních objektů*, Master Thesis, 2004.

[2] Wikipedia, *Object database*, URL: http://en.wikipedia.org/wiki/-ODBMS.

[3] Josef Troch, *Persistence objektů v C++*, Master Thesis, 2004.

[4] Michal Kopecký, *Object persistency in C++*, Doctoral Thesis, 2004.

[5] Michal Kopecký, *POLiTe User's Reference*, 2002.

[6] A. Silberschatz et al, *Applied Operating System Concepts*, 2000.

[7] N. Megiddo and S. M. Dharmendra, *ARC: A Self-Tunning, Low Overhead Replacement Cache*, March 31, 2003.

[8] Frank Manola and Jeff Sutherland, *SQL3 Object Model*, URL: http://www.objs.com/x3h7/sql3.htm.

[9] Michael Stonebraker and Dorothy Moore, *Object-Relational Dbmss: the Next Great Wave*, 1996.

[10] Oracle, *Oracle Database Online Documentation 10g Release 2*, URL: http://www.oracle.com/pls/db102/homepage.