C++ Object Persistency Using Object/Relational Databases

Petr Čermák

April 5, 2009

Contents

List of Figures

List of Tables

Anotace

Anotace CZ:

Práce se zabývá návrhem a konstrukcí knihovny poskytující služby objektově-relačního mapování programům psaných v jazyce C++. Důraz je kladen na snadnost a transparentnost použití. Pro dosažení tohoto cíle knihovna používá nástroj GCCXML, což je rozšíření kompilátoru GCC. GCCXML poskytuje knihovně popis modelu tříd aplikace ve formátu XML, čehož knihovna využívá pro napodobení reflexe.

V práci je provedena analýza požadavků týkajících se implementace této knihovny. Analýza se zabývá také možností využití nových vlastností objektově-relačních databázových systémů. Těch je využito pro zavedení nového typu mapování objektů do databází.

Implementace knihovny vychází ze tří podobných přechůdců - z knihoven POLiTe, POLiTe 2 a IOPC. Navržená a v práci implementovaná knihovna je sjednocuje v jedinou, flexibilní a rozšiřitelnou platformu. Díky její modulární architektuře je možno knihovnu používat v několika konfiguracích poskytujících různé podmnožiny implementovaných služeb - databázový přístup, reflexe a objektově-relační mapování.

Klíčová slova: Perzistence objektů, GCCXML, reflection, ORDBMS, POLiTe, IOPC

EN:

This thesis deals with design and implementation of a library providing object-relational mapping services for programs written in C++. Emphasis is put on its transparency and ease of its use. To achieve this goal the library uses GCCXML, a XML output extension to GCC. GCCXML helps the library to get description of the class model used in the user application and to simulate the reflection.

Requirements that may be imposed on such object-relational mapping library are analysed. For the mapping purposes, new object-relational database features are discussed and a new mapping type is proposed.

Implementation of the library is based on three related projects - the POLiTe, POLiTe 2 and IOPC libraries. The proposed and implemented library unifies them into one solid, flexible and extensible platform. Thanks to its modular architecture, the resulting library can be used in several configurations providing subsets of implemented services - database access, reflection and object-relational mapping.

Keywords: Object persistence, GCCXML, reflection, ORDBMS, POLiTe, IOPC

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* every time 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 their 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].

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. Languages 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. Second goal is to examine possibilities the ORDBMSs can provide to a O/R mapping library.

The result of this thesis was to use advantages of three previous projects. Their common pre-

[1] see [2]

[2] http://www.hibernate.org/

[3] http://www.nhibernate.org/

[4] http://msdn.microsoft.com/en-us/library/bb399572.aspx

decessor 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 persistence 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.

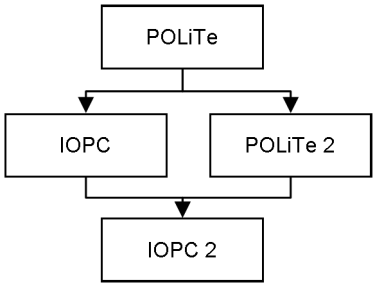

Figure 1.1: IOPC 2 evolution

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.

In the following section we will introduce basic concepts of the object-relational mapping, discuss new features of ORDBMSs and describe requirements and goals of the IOPC 2 implementation. Then, in the third chapter, evolution of the IOPC/POLiTe libraries will be presented in the context of requirements placed and their features will be compared with each other. Chapter 4 describes basic concepts of the IOPC 2 implementation whereas Chapter 5 contains detailed architectural information. 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

[5] Persistent Object Library Test
[6] Implementation of Object Persistency in C++

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 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 traceable 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.

[1] A Universally Unique Identifier

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 modelled 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.

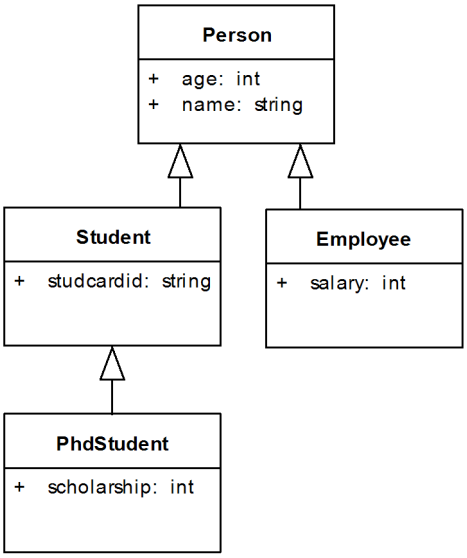

Figure 2.1: Example class hierarchy

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 polymorphism 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 polymorphic 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.

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 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 very well on polymorphic data and doesn't involve cascaded operations/joins. A disadvantage of this approach is excessive 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. 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 Section 4.3.4. 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 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 `PhonesTable` 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 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

[2]Binary large object

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 referred to as *ADT mapping* in this thesis[3]. 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 ADT mapping (see the Example 2.4.1), as all these tasks can be accomplished 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 the ADT 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 Section 4.3.2

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

[3]If not otherwise stated, we will consider the ADT mapping as one of the object-relational mapping types and the process of storing or loading ADT-mapped instances will also be called object-relational mapping.

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[4] looks 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

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

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.

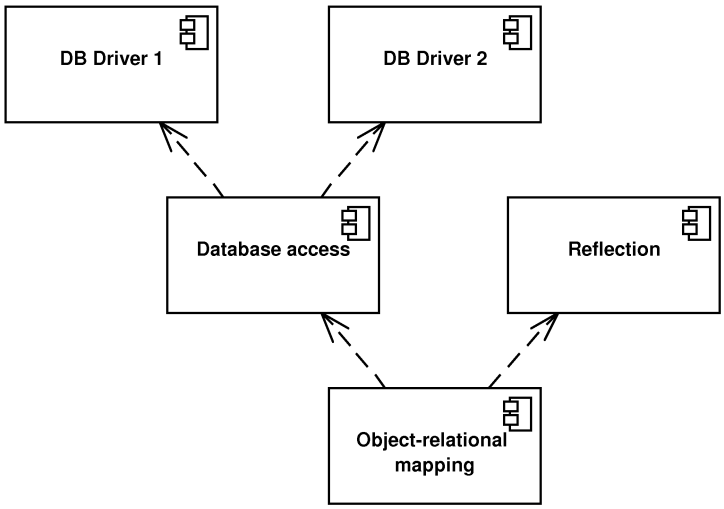

Figure 2.6: Proposed architecture of the O/R mapping library

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.

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.

[1] Oracle Call Interface

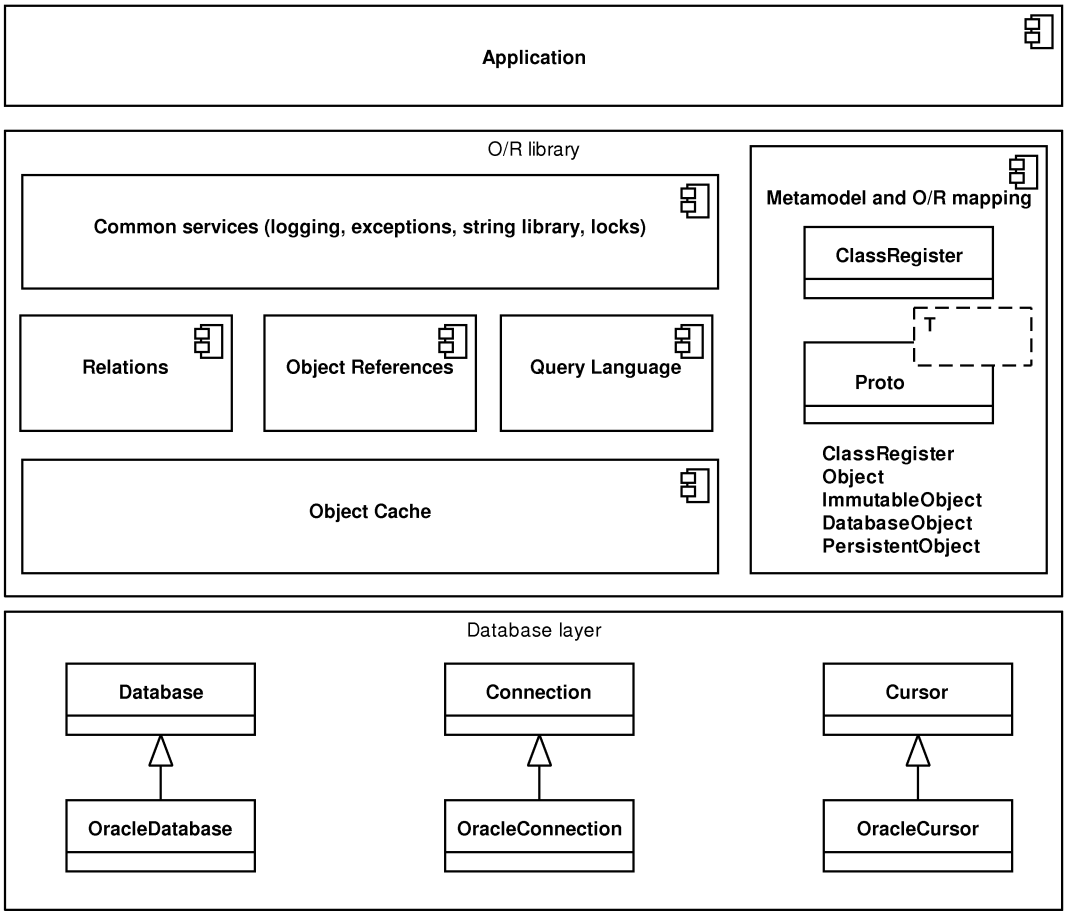

Figure 3.1: Architecture of the POLiTe library

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

```
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;
};
// Define method returning pointer to the prototype
CLASS_PROTOTYPE(Person);
// Define the solitaire prototype instance Person_class
PROTOTYPE(Person);

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

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, the programmer has to maintain this flag either by himself or better he should 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 the library creates an associated template class prototype `Prot-o<T>`. The solitaire instance of this prototype class holds information about the metamodel

described by the macros and provides the actual database mapping. Prototypes are registered within the `ClassRegister`. Using `ClassRegister`, the library and/or application can search for prototypes by their names, and access methods needed for CRUD[2] operations.

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 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 thus 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 descendants still don'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. 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 expected and supported.

As mentioned before, the library offers vertical mapping for descendants of the `PersistentObject`. Tables related to mapped inheritance hierarchies are joined using the surrogate OID key. If using `DatabaseObject` descendants, the object model can be created upon an existing (and in case of `ImmutableObject` descendants even upon the read-only) database tables with arbitrary keys. In this case, however, no inheritance between classes is allowed.

Associations in the POLiTe library are not modelled as references but as instances of the `Relation` class. There are five subclasses of this class - `OneToOneRelation`, `OneToManyRelation`, `ManyToOneRelation`, `ManyToManyRelation` and `ChainedRelation` according to cardinality of the association. 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.

[2]Create (insert) / Read (select) / Update / Delete

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 s 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):

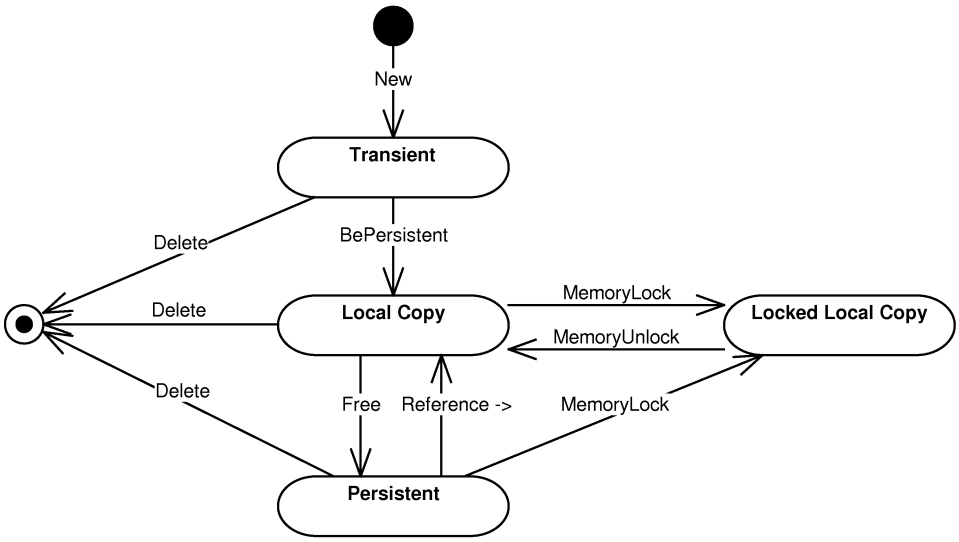

Figure 3.2: POLiTe persistent object states

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

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.

Because persistent object can exist in one of those states, library uses indirect references to access the object's attributes. Users don't have to know whether the object is loaded into the object cache or if it exists only in the database. Users can just access it via the `Ref<T>` reference type using the overloaded `->` operator. The library looks for the requested instance in the object cache and if not found, it loads it from the database. C++ chains the operator `->` calls until it gets to a type that doesn't overload the `->` operator and there it accesses the requested attribute or calls the requested function. So if the variable `e` is of the `Ref<Employee>` type, the following expression:

```
e->salary(65000);
```

doesn't invoke the setter method on the `Ref<Employee>` instance, but it looks for the object in the object cache, loads it eventually, and invokes the setter method on it. The process is illustrated by the Figure 3.3.

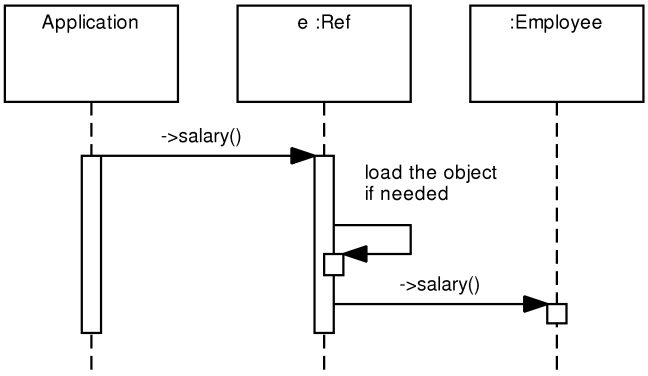

Figure 3.3: POLiTe references

POLiTe allows the users to specify several concurrent data access strategies the `ObjectBuffer` will use. These strategies are used to influence the safety or speed of concurrent access 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 clause of the final SELECT statement together with the ORDER BY clause specification, determines what object will be returned and how the result will be 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. Results of the query execution are accessed using instances of the `Result<T>` template class. The template is used similarly to the `Ref<T>` template. 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
Result<Employee>* result = Employee_class(q3, dbConnection);
while (++(*result)!=DBNULL) {
  // members of the current object are accessible
  // using (*result)->
};
result->Close();
delete result;
```

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 compile time 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 `Database`, `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

New version of the POLiTe library focuses on the library's performance and on the design of new rich-featured cache layer. The cache layer replaces the `ObjectBuffer` interface and

enhances the concept of indirect memory pointers by adding one more indirection level.

3.2.1 Architecture of the POLiTe 2 library

Architecture of the library remained almost unchanged. It is still compiled into one module and used the same way as the original POLiTe library was. Overview of the architecture can be seen in the Figure 3.4

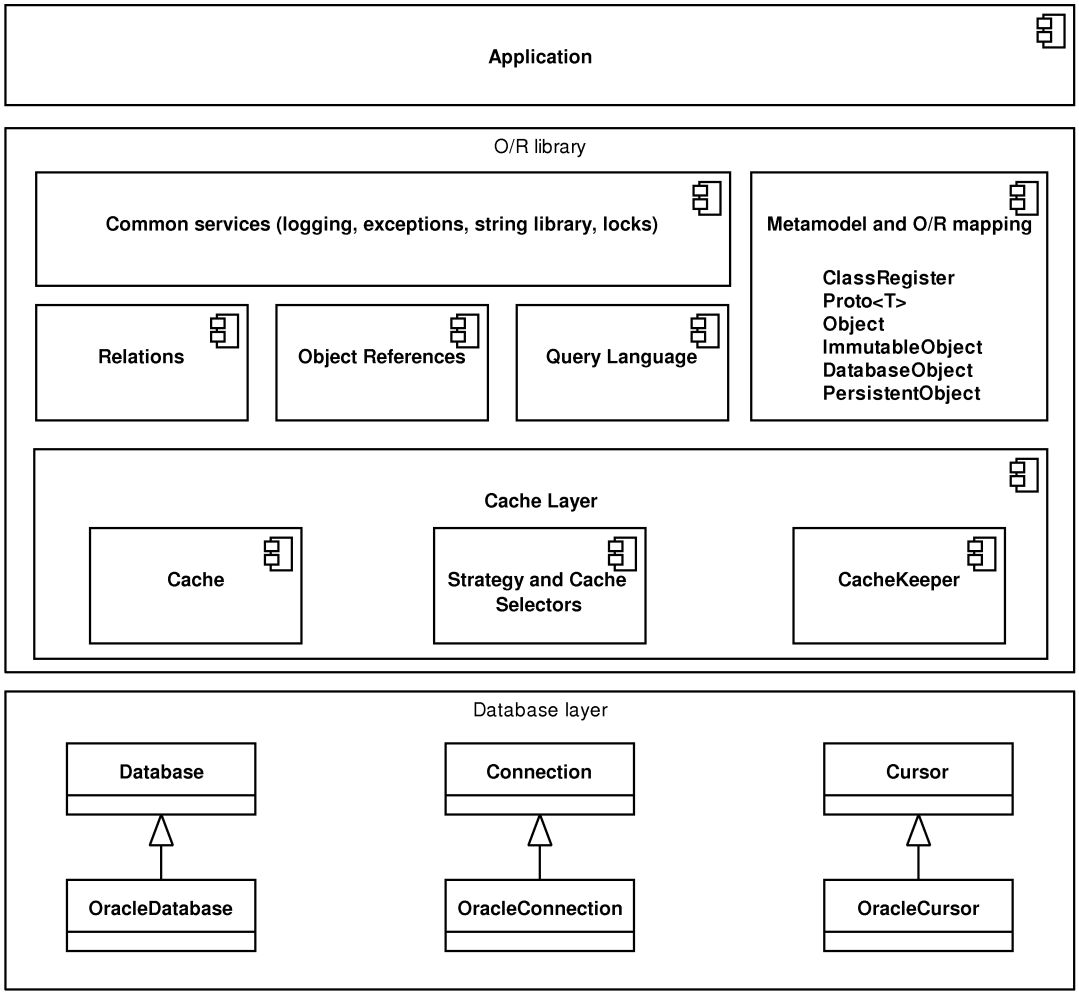

Figure 3.4: Architecture of the POLiTe 2 library

Database layer uses OCI 7 to connect to the Oracle 7 database. The layer has been slightly modified to be able to notify the cache layer of several events like Commit or Rollback.

On the contrary, both the object cache `ObjectBuffer` and the object references have been completely replaced by the new cache layer and the related infrastructure - database pointers. The differences are explained in the following sections.

Because the new cache layer may use additional maintenance threads, most parts of the library have been made thread-safe. Various synchronisation primitives were added to the common services. The main architectural change is that all basic database CRUD operations are routed through the cache layer which decides when or how to perform them.

3.2.2 The cache layer

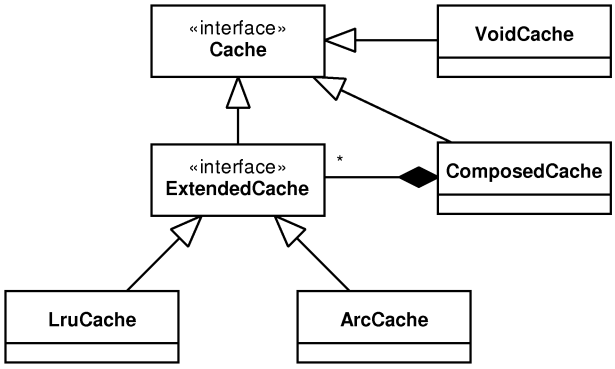

Figure 3.5: Caches in the POLiTe 2 library

The biggest part of the cache layer is certainly the cache implementations. Cache implementations may derive from one of two interfaces depending on what features they will provide (see the Figure 3.5):

- `Cache` - an interface providing basic synchronous caching functionality.
- `ExtendedCache` - extends the `Cache` interface with additional methods that are needed for asynchronous maintenance of the cache using the cache manager `CacheKeeper` (see later).

Concurrent data access strategies as introduced in the original POLiTe library can be used with caches that implement the `ExtendedCache` interface.

Caches can be grouped together using the `ComposedCache` class with the help of *selectors*. Selectors are classes whose instances determine which strategy or which cache should be used based on the given object type, the object itself or based on other criteria. The `ComposedCache` uses `CacheSelector` to decide which cache should be used for the object being processed and then it passes a `StrategySelector` to the selected cache as a parameter to each of its operations. Then the cache uses the `StrategySelector` to modify its behaviour with selected strategies.

Caches that implement the extended interface can be maintained by the cache manager called `CacheKeeper`[3]. `CacheKeeper` runs a 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.

Following cache implementations are provided with the POLiTe 2 library:

- `VoidCache` - implements the basic `Cache` interface. 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` (single threaded) implement the `ExtendedCache` interface. These classes use the LRU replacement strategy (see [6]). The multithreaded variant can even be used with the `CacheKeeper`'s asynchronous maintenance feature.

[3]Caches that implement the basic `Cache` interface can be maintained too, but the `CacheKeeper` doesn't provide the asynchronous maintenance for them.

- The `ARCCache` (multithreaded) and the `ARCCacheST` (single threaded) use the ARC replacement strategy (see [7]).

3.2.3 Persistent object manipulation

Original POLiTe transient objects are created using standard C++ dynamic allocation - using the `new` operator. After making these objects persistent, pointers to these objects are exchanged for indirect references represented by the `Ref<T>` template. As the ownership of these objects is transferred to the object cache, the pointers may be rendered invalid. In the POLiTe 2 library, the creation and destruction of the objects is managed by the library code and users can access its members by dereferencing indirect pointers.

The indirect pointer template `Ref<T>` was replaced by the template `DbPtr<T>`. Its instance may be referred to as a *database pointer* further in the text. `DbPtr<T>` is used similarly to the `Ref<T>` template. Example 3.2.1 demonstrates the creation of a transient instance and its manipulation analogously to the Example 3.1.3.

Example 3.2.1 Persistent object manipulation in the POLiTe 2 library

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

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

// Update the employee's salary
e->salary(65000);
// There is no Update method as the cache updates the persistent
// image automatically according to the Updating strategy in use.

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

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 or may not have a database representation. This represents in fact two states of the object).
- Persistent representation of the object. The pointer contains only the identity of the object.

The last object state, the locked local copy, is implemented as an additional level of indirection when dereferencing the database pointer. By dereferencing it using the $\star$ operator, user receives an instance of a *cache pointer* (`CachePtr<T>`). The existence of the cache pointer guarantees that the object is loaded into the memory and that the memory address of the object will not change until the instance of the cache pointer is destroyed (unlocked). Such process locks the objects in the cache so they can't be removed unless they are unlocked. The transitions between the object states are displayed in the Figure 3.6.

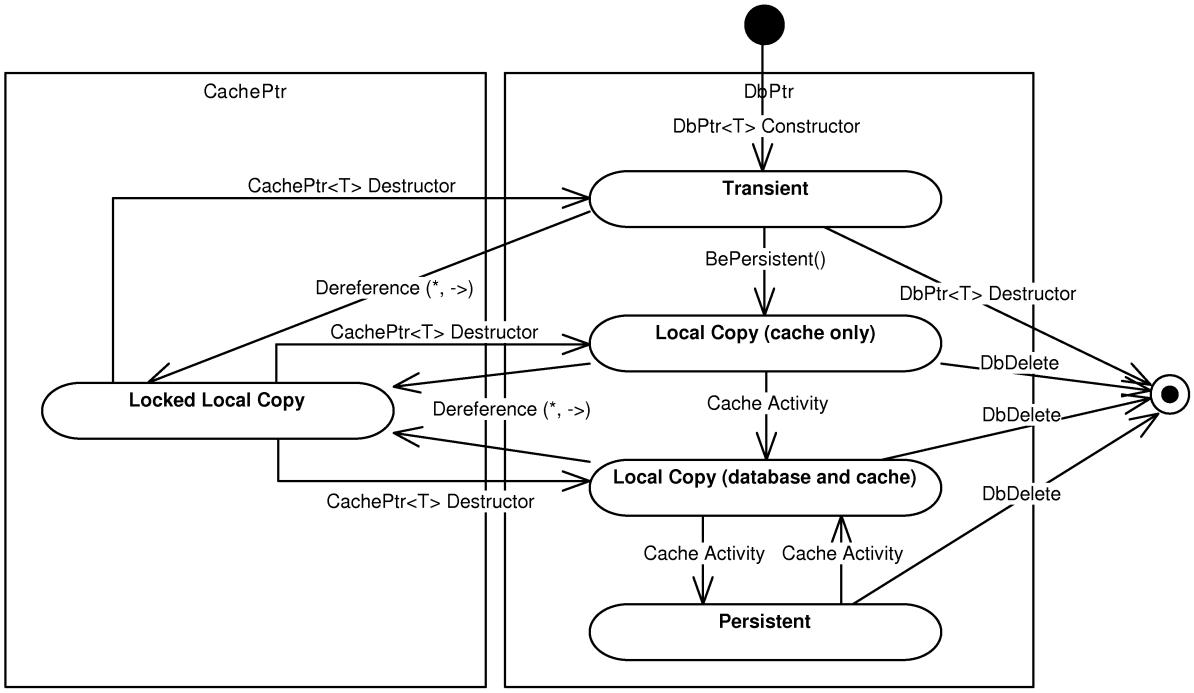

Figure 3.6: States of the POLiTe 2 objects

By using the $\star$ operator on a database pointer user receives a reference to a loaded and locked instance in a cache. Constructor of the created cache pointer asks cache to load relevant object from the database (if not already present in the cache) and locks the object in the cache. The cache pointer can be dereferenced again 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 invoked on it. Then the requested member of the instance is accessed. After that, the cache pointer instance 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). The process is almost the same as if dereferencing the `Ref<T>` references, the difference is the additional level - the cache pointer.

If the variable `e` is of the `DbPtr<Employee>` type, the following expression:

```
e->salary(65000);
```

may trigger a sequence of actions as displayed in the Figure 3.7

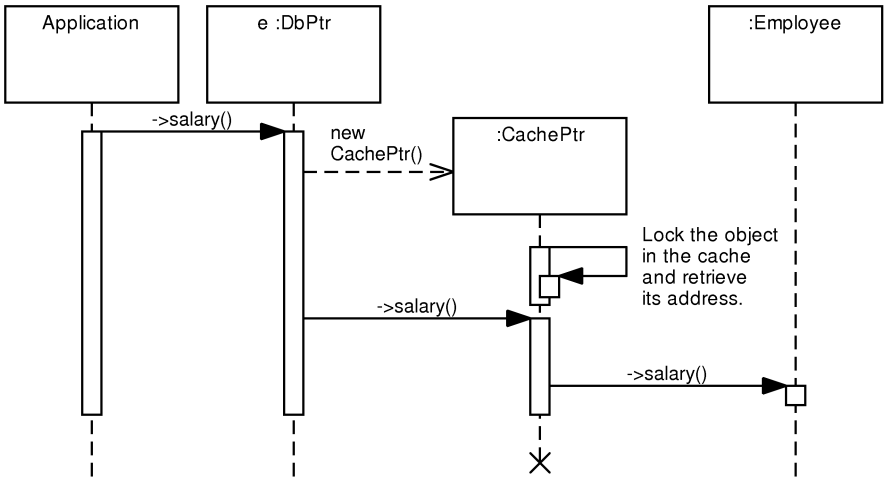

Figure 3.7: Dereferencing DbPtr in POLiTe 2

3.2.4 Querying

The query language and query classes as described in Section 3.1.5 have not been changed in the POLiTe 2 library. Only the query is now executed and its results fetched in a slightly different way:

Example 3.2.2 Executing queries in the POLiTe 2 library

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

// Execute q and iterate through the result
Result<Employee> result(dbConnection, q);
while (++result) {
  // members of the current object are accessible
  // using result->
};
result->Close();
```

3.2.5 Conclusion

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

The IOPC library [3] contains a new API for O/R mapping. This new interface coexists with the old POLiTe-style interface inside one library. 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.

3.3.1 Features of the library

The IOPC library offers all features of the original POLiTe library. We will look only at the new IOPC interface. 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).
- Database views that join data from all mapping tabes are generated. These views are used for object loading and they also represent a simple read-only interface for non-library users.
- Ability to define new persistent data types.
- 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[4].

3.3.2 Architecture of the library

The IOPC library consist of a number of modules, some of them are standalone applications. The hi-level architecture overview can be best explained on the library workflow displayed in Figure 3.8.

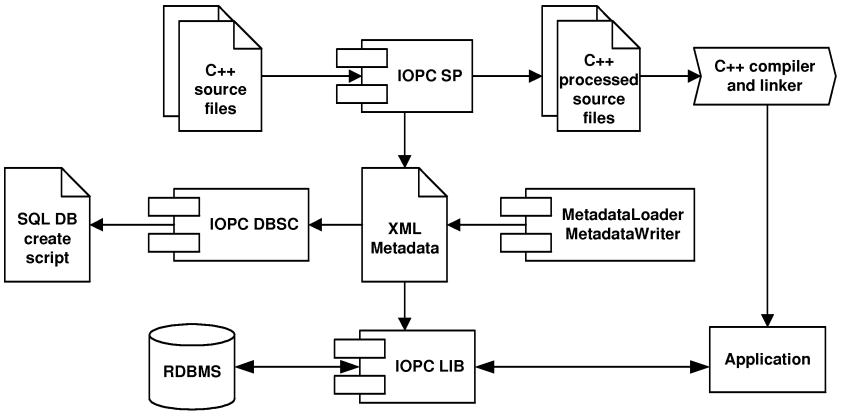

Figure 3.8: The IOPC library workflow

Three main modules can be observed from the figure:

- *IOPC SP* uses OpenC++[5] parser and source-to-source translator to modify the source code to support the object persistence and generates XML metamodel description of structure of persistent structure.

[4]However, recompilation of several IOPC modules is still needed
[5]http://opencxx.sourceforge.net/

- *IOPC DBSC* generates SQL scripts from the XML metamodel description. The SQL scripts create required database structured needed for the IOPC object-relational mapping.
- *IOPC LIB* is a library that provides the object persistence services to a program to which it is linked. It uses the metamodel generated by the IOPC SP module and database structures created using the IPOC DBSC scripts.
- XML metamodel description loading and storing is performed by two statically-linked libraries / classes - `XMLMetadataLoader` and `XMLMetadataWriter`. Both classes use the Xerces[6] parser to handle the XML files. Other type of metamodel storage can be implemented by subclassing the `MetadataLoader` and `MetadataWriter` interfaces.

3.3.3 IOPC SP

One of the most interesting aspects of the IOPC library is a new approach to metamodel retrieval. User-created specification of the class structure is not needed. The source classes are parsed and metamodel description is collected by the IOPC SP module.

IOPC SP is a standalone executable created from patched OpenC++ source code and a IOPC metaclass `IopcTranslator`. The metaclass affects the source-to-source translation of persistent classes done by the OpenC++ by performing the following operations:

- 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 their persistent attributes so that they use the generated get and set methods.
- Generates additional members to the processed class needed by the IOPC LIB.
- Inspects the processed class and writes information about its structure to a XML file by calling the `MetadataWriter` (its implementation `XMLMetadataWriter`).

In the end, IOPC SP runs compiler and linker on the translated source code.

As you can see in the Example 3.3.1, persistent classes in IOPC doesn't need any additional descriptive macros for the persistence layer to be able to understand their structure. The only rules are that they need to be descendants of the `IopcPersistentObject` and that the attribute types need to be supported by the IOPC. For example, IOPC doesn't understand the `std::string` STL type, only the basic C/C++ string representation `char*` (or its wide character variant) is supported. If the string is allocated dynamically, it needs to be deallocated in the desctructor.

[6] http://xml.apache.org/xerces-c/

Example 3.3.1 Definition of persistent classes in IOPC

```
class Person : public IopcPersistentObject {
public:
  char* name;
  short int age;
  Person() {
    name = NULL;
  }
  virtual ~Person() {
    if (name != NULL) free(name);
  }
};
class Employee : public Person {
public:
  int salary;
};
class Student : public Person {
public:
  char* studcardid;
  Ref<Employee> supervisor;
  Student() {
    studcardid = NULL;
  }
  virtual ~Student() {
    if (studcardid != NULL) free(studcardid);
  }
};
```

Processed source code generated by the IOPC SP is deleted immediately after compilation. It is not meant to be modified by developers. Following example displays the processed `Student` class. It was stripped by the age attribute because it was too long:

```
class Person : public IopcPersistentObject {
public:
  Person() {
    set_name ( __null ) ;
  }
  virtual ~Person() {
    Free();
    if ( m_name != __null ) free ( m_name ) ;
  }
protected:
   char * m_name;
   bool m_name_isValid;
public:
  virtual char * get_name()
  {
    if (!m_isPersistent || m_name_isValid || !m_classObject-> ↩
       isAttributePersistent(1)) return m_name;
    m_classObject->loadAttribute(1, this);
    if (!m_name_isValid)
      throw IopcExceptionUnexpected();
```

```
    return m_name;
  }
public:
  virtual char * set_name(char * _name) {
    m_name =_name;
    m_name_isValid = true;
    if (m_isPersistent) MarkAsDirty();
    return _name;
  }
protected:
  void iopcInitObject(bool loadingFromDB) {
    if (loadingFromDB) {
      m_name_isValid = false;
    }
    else {
      m_classObject = IopcClassObject::getClassObject(ClassName(), ↩
          true);
      m_name_isValid = true;
    }
  }
  virtual int iopcExportAttributes(IopcImportExportStruct * data,  ↩
     int dataLen) {
    if (dataLen != 1) return 1;
    data[0].valid = m_age_isValid;
    if (m_age_isValid) data[0].shortVal =  m_age;
    return 0;
  }

  virtual int iopcImportAttributes(IopcImportExportStruct * data,  ↩
     int dataLen) {
    if (dataLen != 1) return 1;
    if (data[0].valid) {
      if ((m_name_isValid) && (m_name)) free(m_name);
      m_name = strdup(data[1].stringVal);
      m_name_isValid = true;
    }
    return 0;
  }
public:
  static const char * ClassName() {return "Person";}
public:
  static IopcPersistentObject * iopcCreateInstance() {
    IopcPersistentObject * object = new Person;
    object->iopcInitObject(true);
    return object;
  }
  static RefBase * iopcCreateReference() {
    return new Ref<Person>;
  }
};
static IopcClassRegistrar<Person> Person_IopcClassRegistrar(" ↩
   Person");
```

Rather larger amount of code was inserted into the class definition. IOPC SP generated setter and getter methods for the attributes, serialisation and deserialisation routines (`iopcExportAttributes` and `iopcImportAttributes`) and other methods needed by the persistence layer.

Not only the class code was translated. Code that reads or sets attribute values was changed to use the getter and setter methods (like we would use in the POLiTe library):

```
// original:
Employee e;
e.age = 45;
// translated assignment expression:
e.set_age(45);
```

IOPC SP also generates a XML metamodel file describing structure of the classes, see Example 3.3.2.

Example 3.3.2 XML metamodel description file

```
<iopc_mapping project_name="example">
 <class name="Employee">
  <base_class>Person</base_class>
  <mapping db_table="EMPLOYEE" type="inherited">
   <group name="default_fetch_group" persistent="true">
    <attribute db_column="SALARY" db_type="NUMBER(10)" name=" ↩
       salary" type="int"/>
   </group>
  </mapping>
 </class>
 <class name="Person">
  <mapping db_table="PERSON" type="vertical">
   <group name="default_fetch_group" persistent="true">
    <attribute db_column="AGE" db_type="NUMBER(10)" name="age"  ↩
       type="short"/>
   </group>
   <group name="1st_persistent_group" persistent="true">
    <attribute db_column="NAME" db_type="VARCHAR2(4000)" name=" ↩
       name" type="char *"/>
   </group>
  </mapping>
 </class>
 <class name="Student">
  <base_class>Person</base_class>
  <mapping db_table="STUDENT" type="inherited">
   <group name="default_fetch_group" persistent="true">
    <attribute db_column="SUPERVISOR" db_type="NUMBER(10)" name=" ↩
       supervisor" type="Ref&lt;Employee>"/>
   </group>
   <group name="1st_persistent_group" persistent="true">
    <attribute db_column="STUDCARDID" db_type="VARCHAR2(4000)"  ↩
       name="studcardid" type="char *"/>
   </group>
  </mapping>
 </class>
</iopc_mapping>;
```

As mentioned, persistent attributes can be divided into several groups which are handled by IOPC separately. By default, two groups are generated - a `default_fetch_group` containing all numeric attributes and a `1st_persistent_group` containing attributes of all remaining data types (strings). The XML metamodel file can be customized by developers before running IOPC DBSC.

3.3.4 IOPC DBSC

IOPC DBSC is a standalone executable that generates SQL scripts for various purposes - in particular the scripts to create or delete database structures required by the object-relational mapping. It uses `MetadataLoader` to load the persistent class metamodel written by the IOPC SP. SQL script created from the previously generated XML metamodel file Example 3.3.2 follows.

First it creates database table for the class list and fills it:

```
CREATE TABLE EXAMPLE_CIDS
(
  CLASS_NAME VARCHAR2(64)
    CONSTRAINT EXAMPLE_CIDS_PK PRIMARY KEY,
  CID NUMBER(10) NOT NULL
    CONSTRAINT EXAMPLE_CIDS_UN UNIQUE
);
INSERT INTO EXAMPLE_CIDS VALUES ('Employee', 1);
INSERT INTO EXAMPLE_CIDS VALUES ('Person', 2);
INSERT INTO EXAMPLE_CIDS VALUES ('Student', 3);
```

Followed by a table (*main table*) containing OIDs of all persistent objects in the project:

```
CREATE TABLE EXAMPLE_MT
(
  OID NUMBER(10)
    CONSTRAINT EXAMPLE_MT_PK PRIMARY KEY,
  CID NUMBER(10)
    CONSTRAINT EXAMPLE_MT_FK REFERENCES EXAMPLE_CIDS(CID)
);
CREATE INDEX EXAMPLE_MT_CID_INDEX ON EXAMPLE_MT(CID);
```

Then the mapping tables associated with persistent classes are created (we used the default vertical mapping):

```
CREATE TABLE PERSON
(
  OID NUMBER(10)
    CONSTRAINT PERSON_PK PRIMARY KEY
    CONSTRAINT PERSON_CD REFERENCES EXAMPLE_MT(OID) ON DELETE ↩
       CASCADE,
  CID NUMBER(10)
    CONSTRAINT PERSON_FK2 REFERENCES EXAMPLE_CIDS(CID),
  AGE NUMBER(10),
  NAME VARCHAR2(4000)
);
CREATE INDEX PERSON_CID_INDEX ON PERSON(CID);
CREATE TABLE EMPLOYEE
(
  OID NUMBER(10)
    CONSTRAINT EMPLOYEE_PK PRIMARY KEY
    CONSTRAINT EMPLOYEE_CD REFERENCES EXAMPLE_MT(OID) ON DELETE ↩
       CASCADE,
  SALARY NUMBER(10)
);
CREATE TABLE STUDENT
(
  OID NUMBER(10)
    CONSTRAINT STUDENT_PK PRIMARY KEY
    CONSTRAINT STUDENT_CD REFERENCES EXAMPLE_MT(OID) ON DELETE ↩
       CASCADE,
  STUDCARDID VARCHAR2(4000),
  SUPERVISOR NUMBER(10)
```

```
);
```

Finally, two views are created for each persistent class. *Simple views* (`SV` suffix) join all tables needed for loading complete instances of particular persistent classes. Their columns correspond to attributes of the associated classes. *Polymorphic views* (`PV` suffix) have same structure as simple views: the difference is that they return not only instances of the associated classes, but also instances of their descendants.

```
CREATE VIEW Person_sv AS
SELECT OID, AGE, NAME
FROM PERSON
WHERE CID = 2;

CREATE VIEW Person_pv (CID, OID, AGE, NAME) AS
SELECT CID, OID, AGE, NAME
FROM PERSON
WHERE PERSON.CID IN (2, 1, 3);

CREATE VIEW Employee_sv (OID, AGE, NAME, SALARY) AS
SELECT EMPLOYEE.OID, AGE, NAME, SALARY
FROM PERSON, EMPLOYEE
WHERE EMPLOYEE.OID = PERSON.OID;

CREATE VIEW Employee_pv (CID, OID, AGE, NAME, SALARY) AS
SELECT 1, EMPLOYEE.OID, PERSON.AGE, PERSON.NAME, SALARY
FROM PERSON, EMPLOYEE
WHERE EMPLOYEE.OID = PERSON.OID;

CREATE VIEW Student_sv (OID, AGE, NAME, STUDCARDID, SUPERVISOR) AS
SELECT STUDENT.OID, AGE, NAME, STUDCARDID, SUPERVISOR
FROM PERSON, STUDENT
WHERE STUDENT.OID = PERSON.OID;

CREATE VIEW Student_pv (CID, OID, AGE, NAME, STUDCARDID,  ↩
   SUPERVISOR) AS
SELECT 3, STUDENT.OID, PERSON.AGE, PERSON.NAME, STUDCARDID,  ↩
   SUPERVISOR
FROM PERSON, STUDENT
WHERE STUDENT.OID = PERSON.OID;
```

3.3.5 IOPC LIB

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 provides the run-time functionality.

IOPC LIB is built on the POLiTe library, it uses some of its components and exposes its new interface side-by-side with the original POLiTe interface. The result is that the IOPC library can be used almost the same way as its predecessor. It supports the POLiTe-style persistent objects as well as new persistent objects inherited from the `IopcPersistentObject` class and processed with IOPC SP. Components from the POLiTe library that IOPC LIB uses are displayed in Figure 3.9.

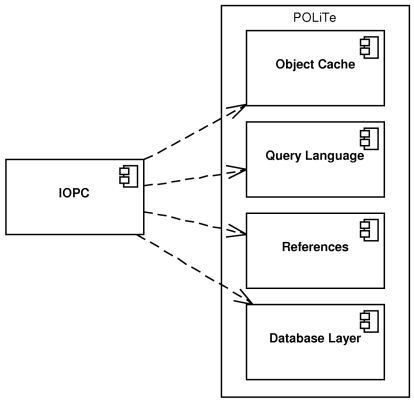

Figure 3.9: POLiTe library components used in IOPC LIB

Because the object cache and the object references are reused from the original, persistent object should enter same states using same state transitions as in Section 3.1.4. There is, however, one problem with the current implementation in that it doesn't implement the locking correctly and all persistent objects look like unlocked all the time. So the Locked Local Copy state can be entered only by instances of the POLiTe persistent classes.

Querying, the query language and all related classes or templates are also reused, so there is no change in this area either.

IOPC persistent objets can be associated exclusively using references as described at the end of the Section 3.1.3. Relations described earlier in that section can be used only for the POLiTe persistent objects. IOPC adds a new `RefList<T>` template to the standard POLiTe `Ref<T>` reference representing a persistable list of references. The list is stored into a separate table (one per project) which unfortunately doesn't have standard many-to-many join table schema. Each instance of the `RefList<T>` is stored as a linked list of OIDs of its members. The table completely unusable in SQL queries unless we are using its recursive features (if available) or procedural constructs like cursor iteration. Second issue is that these lists are always persistent. Every change is immediately propagated to the database regardless of the state of its owner, object cache is also bypassed. This renders the usage of `RefList<T>` in combination with transient objects quite dangerous as orphaned entries are created in the "join" table. Same problem occurs if persistent object with `RefList<T>` as a attribute is deleted - the associated linked list is not removed from the database.

Figure 3.10 displays the structure of the IOPC library and the relationship between new IOPC- and the original POLiTe interface.

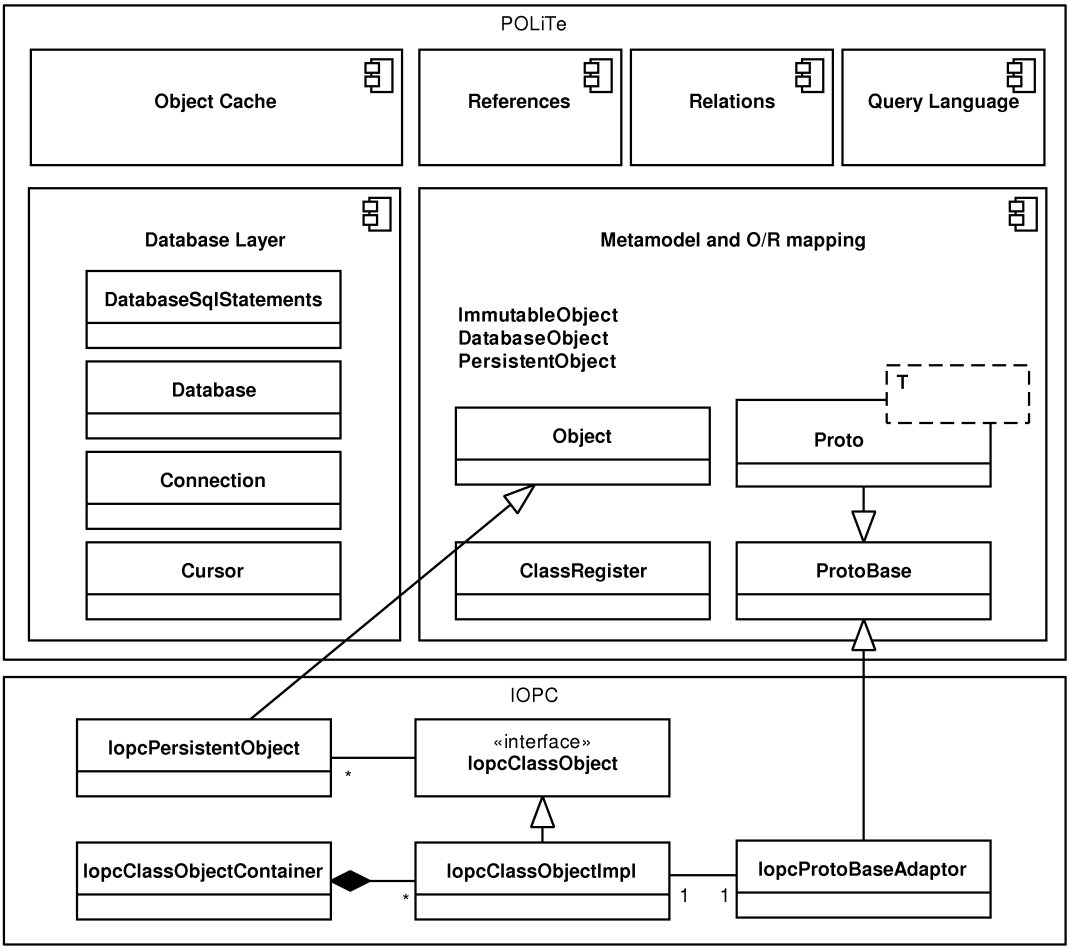

Figure 3.10: Structure of the IOPC LIB

Database layer uses slightly modified original interface. Along with the `Database`, `Connection` and `Cursor` classes there is a new interface class - `DatabaseSqlStatements`. It serves as an interface for database-dependent SQL statement generation. IOPC contains Oracle 8i implementation of these interface classes (using OCI 8).

IOPC persistent classes are created as descendants of the new base class - `IopcPersistentObject`. The original base class `Object` and its descendants (`ImmutableObject`, `DatabaseObject` and `PersistentObject`) were preserved and can be used by the POLiTe persistent objects. Note that `IopcPersistentObject` is derived from the original `Object` class allowing it to be used with POLiTe components like object cache or references.

For each descendant of the `IopcPersistentObject` class there is one `IopcClassObjectImpl` instance. The purpose of this class is very similar to the concept of prototypes described in the Section 3.1.3 - it contains all data needed for object-relational mapping of the associated class. `IopcClassObjectImpl` actually performs the mapping process. The class is linked to the POLiTe prototype system using the `IopcProtoBaseAdaptor` which maps the prototype interface on the `IopcClassObjectImpl` instance. Calls invoked on its `ProtoBase` interface are delegated back to `IopcClassObjectImpl`. Public interface to the `IopcClassObjectImpl` class is provided by the `IopcClassObject` class.

As the name suggests, `IopcClassObjectContainer` is a place where class object instances are stored. Its first responsibility is to load list of classes from the `MetadataLoader` and compare them with the class list stored in the database. Then it creates and initializes the `IopcClassObjectImpl` instances. Similarly to `ClassRegister`, it provides methods

allowing us to find the class objects by name or by class id.

3.3.6 Conclusion

The main goal of the IOPC library was to make the usage of the POLiTe library simpler and more transparent. The architecture of the library was redesigned for the sake of these requirements. 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.

IOPC uses the OpenC++ as a way to retrieve information about the structure of persistent classes. The OpenC++ project seems to be almost dead, last commit to the project's CVS occurred in 2005. OpenC++ can't handle most of template constructs, processing files that include GCC 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 translated with errors or warnings[7] will do what was intended. For this reason the IOPC allows only the 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 further maintenance of the OpenC++ code difficult. 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 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 sense 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". Second example is the mentioned local copy locking problem. Are we supposed to use the locking only on the POLiTe persistent objects or is it just a bug in the IOPC implementation? The presence of two distinct APIs makes usage of the library less clear and confusing.

The library itself remained as one monolithic block with only signs of library configurability. Many parameters are defined as pre-processor 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 at a time (the currently used implementation of the `DatabaseSqlStatements` is stored in a global variable).

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

The library cannot be used in multithreaded environment. There are shared state-aware data structures that 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.

[7] Although the IOPC SP states that those errors can be ignored

3.4 Library comparison

The following table provides side-by-side library comparison. The IOPC column displays features of the IOPC part of the library interface only (not POLiTe).

	POLiTe	**POLiTe 2**	**IOPC**
Transparent usage	- Macro descriptions	- Macro descriptions	+ Uses Open C++
Supported mapping types	- Vertical	- Vertical	+ Vertical, horizontal, filtered, combinations
Associations between objects	+ Simple reference and relations - one-to-many, many-to-one, many-to-many, chained	+ Simple reference and relations - one-to-many, many-to-one, many-to-many, chained	+/- Simple reference and reference list
Caching	- Simple object cache	+ Advanced caching features.	-- Simple object cache, no locking
Querying	C++-like syntax, combining queries	C++-like syntax, combining queries	C++-like syntax, combining queries
Read-only database / existing schema support	+	+	-
Library architecture	- Monolithic	- Monolithic	- Monolithic
Supported databases	Oracle 7 (OCI 7)	Oracle 7 (OCI 7)	Oracle 8i (OCI 8)
Multithreading support	-	+	-

Chapter 4

Basic concepts of the IOPC 2 library

The IOPC 2 library is based on the ideas behind the IOPC library. It takes over features of that library where it is useful while it adds the best from the POLiTe and POLiTe 2 libraries as well. Based on the discussion in the previous chapters IOPC 2 focuses on improvements in the following areas:

- Transparent usage - IOPC 2 retrieves description of persistent class metamodel using similar mechanism as IOPC, so no descriptive macros are needed. Though OpenC++ has been replaced by GCCXML due to reasons explained in Section 3.3.6 and Section 4.2.
- Reflection - IOPC 2 gives application developers the access to the metamodel through a simple reflection interface.
- O/R mapping - IOPC 2 offers all mapping types supported by the IOPC library plus the ADT mapping as presented in Section 2.4. Persistent objects in POLiTe libraries can be mapped on existing database schemas or read-only databases. IOPC 2 has inherited this feature.
- Caching - the POLiTe 2 cache layer has been reused in the IOPC 2 library. IOPC 2 supports all features added to the POLiTe library successor including multithreading support.
- Library architecture - IOPC 2 library can be used in several configurations depending on the developers' needs. Database drivers have been separated from the run-time library and now can be switched without the need of library source code modification.
- Removal of obsolete features – the implementation of the IOPC 2 library cuts off all the interfaces that were replaced by their newer and more complex versions. It simplifies the work with the library and its maintainability at the cost of lost of bacward compatibility.

In the following sections, we will discuss theoretical aspects of the IOPC 2 implementation - mostly the O/R mapping and transparency of the library usage. Following Chapter 5 provides more detailed insight into the library architecture and usage.

4.1 Library architecture

As stated earlier, run-time part of the POLiTe, POLiTe 2 or the IOPC libraries consist of one monolithic block. There is no way how to link the application against only a part of the library, it may even be impossible because of the internal library design. One of the goals of the IOPC 2 implementation was to make the run-time part configurable. For example database drivers have

become separate shared libraries. Thus the addition of another database driver does not result in the run-time library recompilation; drivers do not need to be part of the library source code - they can be developed separately in independent projects.

Configurability does not concern only the database drivers, but also other parts of the library. The library can be used in the following configurations:

- Common services - a configuration that provides only basic services like logging, tracing, thread synchronisation and several other utilities.
- Reflection library - a configuration that exposes an interface allowing application developers to use the IOPC 2 reflection capabilities.
- Database access library - a configuration that provides basic database access via loaded database drivers. It doesn't offer any advanced features as object-relational mapping or caching.
- Persistence library - a configuration that provides the full functionality of the IOPC 2 library. Its features include object persistence, caching, direct database access, reflection and all other features listed above.

The configurations are ordered by dependency, so the reflection library configuration includes features provided by common services and so on. The database access library configuration provides also features of the reflection library, and - as stated earlier - the persistence library configuration provides features of all other configurations.

4.2 Obtaining the metamodel description

As it was explained in Section 3.1.3, the POLiTe and POLiTe 2 libraries depend on preprocessor macro calls included in class definitions to obtain metamodel description.

The IOPC library tries to avoid disadvantages of this approach mentioned in Section 3.1.6. IOPC makes the process of metamodel descriptions retrieval more user-friendly and transparent by utilizing a modified version of the OpenC++ source-to-source translator. For more on this topic, see Section 3.3.3. The main point is that OpenC++ is incomplete and is not developed any more. This leaves users in a bad situation as they can't use STL data types (because the OpenC++ doesn't handle templates very well) or even they can't be sure if their code was processed correctly.

However, the idea to employ an external tool to analyze and process the source code is convenient, so IOPC 2 should use another tool to solve this situation. GCCXML[1] was chosen to replace OpenC++.

GCCXML is a XML output extension to GCC. GCCXML uses GCC to parse the input source code and then dumps all declarations into a XML file, which can be easily parsed by other programs. The downside is that it processes only declarations. Therefore it can't be used as a source-to-source translator. Yet GCCXML represents ideal out-of-the-box solution for discussed scenario. GCCXML can handle the C++ language in its entirety and its usage doesn't involve the risk of translated source code misinterpretation.

IOPC 2 uses the generated XML file to gain knowledge of the structure of classes in the processed source code. This information is then made accessible via reflection interface provided with the library. User applications can then query the metamodel and use it in a way

[1] http://www.gccxml.org

similar to applications written in reflective languages. The reflection mechanism used in IOPC 2 is outlined in Figure 4.1.

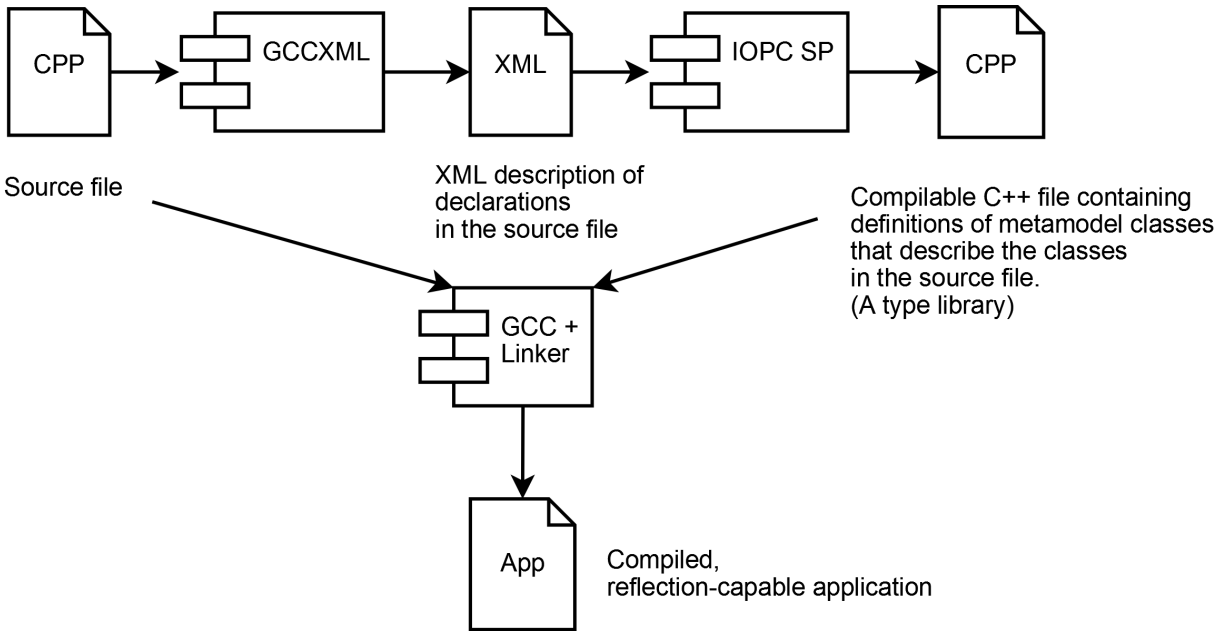

Figure 4.1: Reflection using the GCCXML

Example 4.2.1 illustrates persistent class definition in IOPC 2. No special macros are needed. However, it presents only a basic, least convenient variant as more features are provided by using enhanced data types (see Section 5.4.2) and by specifying additional class metadata (see Section 5.4.3).

Example 4.2.1 Basic persistent class definition in IOPC 2

```
class Person : public iopc::OidObject {
public:
  short int age;
  std::string name;
};
class Employee : public Person {
public:
  int salary;
};
class Student : public Person {
public:
  std::string studcardid;
  DbPtr<Employee> supervisor;
};
class PhdStudent : public Student {
public:
  short int scholarship;
};
```

The process performs the following steps:

1. First, the source file is processed by the GCCXML. GCCXML dumps all declarations from the source file into an easy to parse XML file. An excerpt from the XML file containing elements and attributes related to the processed `Person` class is displayed in Example 4.2.2

2. The XML file is then processed by the IOPC SP[2] utility (a part of the IOPC 2 project). IOPC SP produces compilable C++ file containing data structures that describe classes declared in the source file and their attributes (*type descriptions*). This file, respectively its compiled version, is called *a type catalogue*. For translated `Person` class see Example 4.2.3

3. In the last step, the original source files and created type catalogues are compiled and linked together into a final application.

Example 4.2.2 GCCXML output

```
<!-- Class Person
IOPC SP searches for members which are of Field type.
(Other members may be methods, constructors, operators and other  ↩
   -->
<Class id="_1319" name="Person" members="_2960 _2961 _2962 _2963  ↩
   _2964 _2965 " bases="_2649 ">
  <Base type="_2649" access="public" virtual="0"/>
</Class>

<!-- fields related to the Person class -->
<Field id="_2960" name="age" type="_195" access="public" />
<Field id="_2961" name="name" type="_1608" access="public"/>

<!-- field data types -->
<FundamentalType id="_195" name="short int"/>
<Typedef id="_1608" name="string" type="_1564"/>
```

Example 4.2.3 IOPC SP output

```
// Header file .ih
META_BLOCK1(::Person)
META_PARENT1(iopc::OidObject)
META_ATTR1(age, short int, Attribute::VISIBILITY_PUBLIC)
META_ATTR1(name, std::string, Attribute::VISIBILITY_PUBLIC)
META_BLOCK2(::Person)
META_ATTR2(::Person, age)
META_ATTR2(::Person, name)
META_BLOCK3(::Person, Person, )
META_BLOCK4

// Source file .ic
META_REGISTER_TYPE(::Person)
```

[2] Although its name matches the name of a IOPC module with similar functionality (see Section 3.3.3), this tool was written from scratch for the IOPC 2 library.

The process is flexible and customisable. Developers can integrate it with their favourite IDEs or build systems. IOPC 2 project contains an example of seamless IDE integration, see Section 5.4.4. This way the build process is almost transparent and users don't need to take any additional steps to make the reflection work.

Information stored in the type catalogues are accessible via reflection interface provided by IOPC 2. It allows developers to query the class metamodel and to execute reflective operations upon it. IOPC 2 library supports the following:

- Looking up a `Type` object by class name. The `Type` class (and its descendants) has similar purpose to prototypes from the POLiTe library or to `IopcClassObjectImpl` from the IOPC library. The difference is that it doesn't perform any database mapping, it just provides access to the metamodel.
- Obtaining a list of classes.
- Traversing a class hierarchy.
- Obtaining list of attributes, obtaining a concrete attribute by looking it up by its name, listing inherited attributes.
- Querying or setting attribute values at run-time.
- Creating object instances.
- Automatic dirty status tracking on an attribute or object level.

4.3 Object relational mapping in the IOPC 2 library

All mapping types offered by the IOPC library are also implemented in the IOPC 2 library - horizontal, vertical, filtered as well as combinations of them. IOPC 2 library provides as much freedom combining them within one inheritance hierarchy as the IOPC library does. Additionally, ADT mapping is available for use with object-relational databases. Developers can combine ADT mapping with the other mapping types under certain restrictions.

IOPC 2 offers most of its functionality if using a generated schema with surrogate keys (OID columns) and OID objects. Such schema contains additional views, tables or types which support the mapping process. IOPC 2 also handles existing schemas with no OID columns or additional database structures as well. It is able to map data from read-only databases or from aggregated query results into standard persistent objects.

Detailed aspects of the object-relational mapping implementation are discussed in the following sections.

4.3.1 Base classes

Figure 4.2 displays base class hierarchy for persistent classes defined in IOPC 2. Data classes in user applications should derive from one of these supertypes. Each of these base classes provides different level of functionality for its descendants.

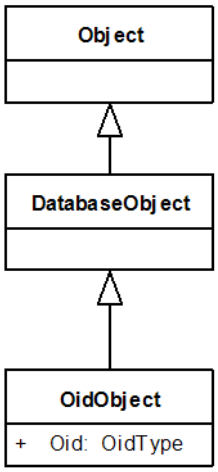

Figure 4.2: IOPC 2 base classes

The descendants of `Object` are capable of reflection. The reflection mechanism inspects descendants (direct or indirect) of this class only. Reflection is database independent and can be used separately as a standalone part of the IOPC 2 library. This also implies that direct descendants cannot be persisted. Persistence is provided for `DatabaseObject` and `OidObject` descendants.

Query results can be represented as instances of the `DatabaseObject` descendants. The objects (and thus the query results) may or may not have an identity. Their identity of `DatabaseObjects` is specified by supplying a list of key attributes. As described in Section 2.1, objects without identity can be used for example to hold results of aggregation queries. `DatabaseObjects` with the key attributes specified can be mapped into database tables. They even support vertical and horizontal database mapping. However, the subclasses have to share the same set of key attributes. This enables inheritance, but polymorphism is somewhat limited as IOPC 2 is not capable of recovering the effective type of an object from a set of key attribute values. IOPC 2 doesn't generate structures for classname resolution based on the key values, so when loading an object identified by its key values, the type of the loaded object must be specified. That is also the reason why filtered mapping is not supported for non-OID objects.

Example 4.3.1 DatabaseObject usage

```
class Person : public iopc::DatabaseObject {
public:
  EString name;
  EString idcard;
  EString country;
  static void iopcInit(iopc::Type& t) {
    t.getAttribute("idcard")["db.primaryKey"].setBoolValue(true);
    t.getAttribute("country")["db.primaryKey"].setBoolValue(true);
  }
};
class Student : public Person {
public:
  EString studcardid;
};
```

Class hierarchy in Example 4.3.1 shares two key attributes - `idcard` and `country`. The `IopcInit` static method is invoked by the reflection mechanism when the program starts. Usually it creates class metadata[3] modifying the database mapping behaviour of this class or of its attributes. In this case it tells the library which attributes are parts of the class composite key. In the generated SQL schema (using the default vertical mapping), all table definitions include the two corresponding key columns - see Example 4.3.2.

Example 4.3.2 SQL schema generated for DatabaseObject subclasses.

```
CREATE TABLE Person (
  country VARCHAR2(2000), idcard VARCHAR2(2000),
  name VARCHAR2(2000),
  CONSTRAINT Person_pk  PRIMARY KEY (idcard, country))
CREATE TABLE Student (
  studcardid VARCHAR2(2000), idcard VARCHAR2(2000),
  country VARCHAR2(2000),
  CONSTRAINT Student_pk  PRIMARY KEY (idcard, country))
```

`OidObject` guarantees an identity in form of an internally generated OID to all of its descendants. Unique OID is assigned to any instance of OidObject descendant at its creation time and remains unchanged during its lifetime. `OidObject` descendants are able to use all types of database mapping and persistent associations. Oid objects depend on database structures generated by the IOPC 2 library. These structures include simple or polymorphic views (as in the original IOPC library - see Section 3.3.4) or an Oid - classname catalogue. The catalogue is actually a regular mapping table associated with the `OidObject` class. Each OID object in the database is represented by a row in the catalogue regardless of the mapping type used. The catalogue table has three columns - `OID`, `CLASSNAME` (qualified name of the class this row - OID - belongs to) and `SERIALID` (timestamp used by the cache layer).

Example 4.3.3 contains schema generated for the classes from the earlier Example 4.2.1. Note that code of all SQL examples below is generated by the Oracle 10g driver. Source code of associated views can be found in appendix *TODO: priklad pohledu do appendixu* .

[3] Similar concept as java annotations. For more on this topic, see Section 5.4.3. Also note that this time we use the enhanced attribute datatypes, see Section 5.4.2.

Example 4.3.3 SQL schema generated for OidObject subclasses.

```
-- the oid - classname catalogue
CREATE TABLE OidObject (
  CLASSNAME VARCHAR2(1000), OID NUMBER(10), SERIALID NUMBER(10),
  CONSTRAINT OidObject_pk  PRIMARY KEY (OID))

-- mapping tables (using vertical mapping)
CREATE TABLE Person (
  age NUMBER(5), name VARCHAR2(2000), OID NUMBER(10),
  CONSTRAINT Person_pk  PRIMARY KEY (OID))
CREATE TABLE Student (
  studcardid VARCHAR2(2000), supervisor NUMBER(10), OID NUMBER(10) ↩
     ,
  CONSTRAINT Student_pk  PRIMARY KEY (OID))
CREATE TABLE PhdStudent (
  scholarship NUMBER(5), OID NUMBER(10),
  CONSTRAINT PhdStudent_pk  PRIMARY KEY (OID))
CREATE TABLE Employee (
  salary NUMBER(10), OID NUMBER(10),
  CONSTRAINT Employee_pk  PRIMARY KEY (OID))
```

4.3.2 ADT mapping

ADT mapping, as described in Section 2.3, is a new type of database mapping introduced in the IOPC 2 library. It requires the underlying DBMS to support definition of user defined abstract data types (ADT).

ADT mapping can be used with some restrictions. First, only OID objects can be marked for use with the object mapping type. Because structures required for this kind of mapping are generated by the library and will surely not be reused from existing databases, there is no reason why not to use OID objects. Working with OID objects is convenient and also faster than any other alternative. Second, ADT mapping is meant to be used for whole hierarchies of classes and thus no other mapping type can be used in any of their descendants. In contrast, ADT mapping hierarchy can be started at any level of inheritance tree of classes with other mapping type.

This leads to several ways of how to use ADT mapping which should be considered when designing classes and data model.

- One ADT hierarchy for all persistent classes in the application. It also means one table for all types and all classes. This can be achieved by telling the root of all OID objects (class `OidObject`) to use ADT mapping.

- One ADT hierarchy per top-level class. Top-level classes (direct descendants of `OidObject`) use the ADT mapping type. These classes and all of their descendants are stored in separate tables and are represented by separate ADT hierarchies. `OidObject` with all object OIDs is mapped into a separate relational table.

- ADT hierarchy started at a lower level of persistent class hierarchy. Ancestors of an ADT-mapped type can use any other type of database mapping. The base ADT definition consists of the OID attribute and attributes defined in the corresponding class. So the base ADT table has same structure as it was generated for a vertically-mapped class. The

horizontal approach (all inherited attributes in the ADT definition) can be considered in the future.

Example 4.3.4 shows how the ADT mapping can be set for the `Student` and `PhdStudent` classes and leaving its parent `Person` and the class `Employee` to use the default vertical mapping. Note, that the ADT mapping type is denoted by the code "object" in the IOPC 2 library.

Example 4.3.4 Using the ADT mapping.

```
class Person : public iopc::OidObject {
public:
  EShort age;
  EString name;
};
class Employee : public Person {
public:
  EInt salary;
};
class Student : public Person {
public:
  EString studcardid;
  DbPtr<Employee> supervisor;
  static void iopcInit(iopc::Type& t) {
    t["db.mapping.type"].setStringValue("object");
  }
};
class PhdStudent : public Student {
public:
  EShort scholarship;
  static void iopcInit(iopc::Type& t) {
    t["db.mapping.type"].setStringValue("object");
  }
};
```

This time we specify the class metadata `db.mapping.type` to change the mapping type of the two classes to ADT mapping. Example 4.3.5 displays an excerpt from the generated SQL schema.

The static methods `insert_object`, `update_object` and `delete_object` are used by IOPC 2 to insert, modify or delete the ADT instances. Constructors couldn't be used because Oracle 10g doesn't allow specifying its parameters in arbitrary order as required by the library.

Example 4.3.5 SQL schema from classes that use ADT mapping.

```
CREATE TABLE Person (
  age NUMBER(5), name VARCHAR2(2000), OID NUMBER(10),
  CONSTRAINT Person_pk  PRIMARY KEY (OID))

CREATE OR REPLACE TYPE tStudent AS OBJECT (
  OID NUMBER(10), supervisor NUMBER(10), studcardid VARCHAR2(2000) ↩
     ,
  STATIC PROCEDURE insert_object(
    p_OID NUMBER, p_studcardid VARCHAR2),
  STATIC PROCEDURE update_object(
    p_OID NUMBER, p_studcardid VARCHAR2),
  STATIC PROCEDURE delete_object(
    p_OID NUMBER)
  ) NOT FINAL

CREATE OR REPLACE TYPE tPhdStudent UNDER tStudent (
  scholarship NUMBER(5),
  STATIC PROCEDURE insert_object(
    p_OID NUMBER, p_studcardid VARCHAR2, p_scholarship NUMBER),
  STATIC PROCEDURE update_object(
    p_OID NUMBER, p_studcardid VARCHAR2, p_scholarship NUMBER),
  STATIC PROCEDURE delete_object(
    p_OID NUMBER)
  ) NOT FINAL

CREATE TABLE Student OF tStudent (
  CONSTRAINT Student_pk  PRIMARY KEY (OID) )
```

4.3.3 Mapping algorithm prerequisites

The mapping algorithm operates with several lists of classes and attributes which are pre-create at the start of the user application. These lists are generated for all persistent classes - descendants of the `DatabaseObject` class.

- `AllParents` - a list of ancestor classes that have a corresponding database table. This includes only classes that use horizontal or vertical mapping. Classes that use filtered mapping do not have a associated database table as their columns are inserted into one of their ancestors. Because ADT-mapped classes are handled in a different way, they are not inserted into this list either. The list represents tables that will be accessed when performing a CRUD operation on an instance of the current class. Only one database operation is needed to modify database data belonging to an instance of any class in an ADT-mapped class subgraph. See the examples at the end of this section and also refer to the mapping algorithm description.

  The list is created by level-order walking through the parent hierarchy starting with parents of the current class first. If the process encounters a horizontally-mapped class, its parents are not processed (because horizontally-mapped classes store all, even inherited

attributes into one table). If the current class derives from `OidObject`, and if `OidObject` doesn't use ADT mapping, `OidObject` is also appended to this list.

- `FilteredTypes` - list of classes that use table associated with the current class as a destination for their attributes when filtered mapping is used. Note that there must be at least one path between the source and destination classes in the inheritance hierarchy that does not contain a horizontally-mapped class.
- `PersistentAttributes` - list of persistent attributes declared in the current class. Developers may declare some of the class attributes as transient. Transient attributes are ignored by the persistence layer.
- `KeyAttributes` - list of key attributes that are inherited or declared in the current class. If the current class derives from `OidObject`, the OID attribute is included in the list. A class cannot re-define key attributes if they were already defined in one of its ancestors.
- `InheritedAttributes` - list of persistent attributes (retrieved from `PersistentAttributes`) inherited from all persistent ancestors.
- `AllPersistentAttributes` - union of the `PersistentAttributes` and `InheritedAttributes` lists.
- `MappedAttributes` - list of attributes that are physically mapped into table associated to the current class. In fact they represent columns of this table. The list may include attributes from other classes. Attributes are categorized by their origin:
  1. Persistent attributes of the current class. (`PersistentAttributes`)
  2. If the current class uses horizontal mapping, the list includes non-key inherited persistent attributes from the `InheritedAttributes` list. Attributes inherited from the `OidObject` are excluded as the Oid - classname catalogue (a table associated with `OidObject`) is accessed even when using horizontal mapping.
  3. Persistent attributes from classes that use filtered mapping - persistent attributes from classes in the `FilteredTypes` list.
  4. Inherited `KeyAttributes` - even if the class doesn't employ horizontal mapping.

  This list is not generated for classes that use ADT mapping.

To understand how the lists are generated, see the following examples. First, lists generated for the default scenario - all persistent classes use vertical mapping. Lists for the `Student` class:

- `AllParents`: `Person`, `OidObject`
- `FilteredTypes`: no members
- `PersistentAttributes`: `studcardid`, `supervisor`
- `KeyAttributes`: `oid`
- `InheritedAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`

- `AllPersistentAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`, `studcardid`, `supervisor`
- `MappedAttributes`: `oid` (key), `studcardid`, `supervisor` (regular columns)

Second, the class `Student` uses horizontal mapping and `PhdStudent` uses filtered mapping to map its attributes to table associated with the `Student` class. Only the contents of the `AllParents`, `MappedAttributes` and `FilteredTypes` lists would change:

- `AllParents`: `OidObject`
- `FilteredTypes`: `PhdStudent`
- `PersistentAttributes`: `studcardid`, `supervisor`
- `KeyAttributes`: `oid`
- `InheritedAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`
- `AllPersistentAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`, `studcardid`, `supervisor`
- `MappedAttributes`: `oid` (key), `studcardid`, `supervisor` (regular columns), `age`, `name` (inherited columns), `scholarship` (filtered column)

Schema generated for this scenario would look like this:

OidObject

OID	ClassName	SerialId
1	::Person	1
2	::Student	2
3	::PhdStudent	3
4	::Employee	4

Student

OID	Name	Age	StudcardId	Supervisor	Scholarship
2	Richard Doe	22	WCD-3223	4	NULL
3	Joe Bloggs	27	PGS-1234	4	12 000

Person

OID	Name	Age
1	Mary Major	60
4	Ola Nordmann	45

Employee

OID	Salary
4	60 000

Figure 4.3: SQL schema generated from classes using combined mapping

As you may see, there is no `PhdStudent` table as all its columns were appended to the `Student` table. As a result of the horizontal mapping, rows belonging to these two classes do not have entries in the `Person` table.

And the last example - classes from the previous section. ADT mapping is enabled for `Student` and its `PhdStudent` descendant, see Example 4.3.4 for the class model definition. This time, the lists for the `PhdStudent` class:

- `AllParents`: `Person`, `OidObject`
- `FilteredTypes`: no members
- `PersistentAttributes`: `scholarship`
- `KeyAttributes`: `oid`
- `InheritedAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`, `studcardid`, `supervisor`
- `AllPersistentAttributes`: `oid`, `serialid`, `classname` (all from `OidObject`), `name`, `age`, `studcardid`, `supervisor`, `scholarship`
- `MappedAttributes`: not generated.

4.3.4 The mapping algorithm

In this section, we will describe the algorithm for CRUD operations performed by the IOPC 2 persistence layer. First, the top-level part of the algorithm for inserting, updating and deleting persistent objects is described.

4.3.4.1 Inserting, updating and deleting persistent objects

Given an object O, the algorithm iterates through its parent class hierarchy (starting with the class of O itself) and calls one of the `Insert_Row()` or `Insert_Object()` methods on it (depending on which mapping type is used). The Insert_Row method is described in Figure 4.5. Update_Row or Delete_Row methods have similar structure.

```
if O uses ADT mapping then
❶  Insert_Object(O)  // Update_Object or Delete_Object

else if O uses horizontal or vertical mapping (O has a  ↩
   corresponding db table) then
❷  Insert_Row(O, class of O)   // Update_Row or Delete_Row
end if

foreach class C in AllParents do
❸  Insert_Row(O, C)   // Update_Row or Delete_Row
end
```

Figure 4.4: Top-level part of the object-relational mapping algorithm

❶ First, the algorithm handles either the whole subgraph of ADT mapping classes (as they are stored as one ADT instance at a time)

❷ or the class of the object being inserted (if it has its own associated database table).

❸ Then it iterates through ancestors of the objects' class and inserts values of attributes defined in them into a new database row.

```
Insert_Row(object O, class C):
  foreach attr A in C.MappedAttributes
    if A is not filtered attribute [category 1, 2, 4]
      insert value of O.A into the new db row
    else -- filtered attribute [category 3]
❶       if class of O is descendant of class containing the ↩
   attribute A
        insert value of O.A from O into the new db row
      else
        insert NULL as attribute A into the new db row
      end if
    end if
  end
```

Figure 4.5: Description of the `Insert_Row` method. Not used for ADT mapping.

The method iterates through attributes (or actually columns of the associated table to the class C) and inserts values to a new row. The process is quite self-explanatory except for filtered attributes (category 3). If a class has two descendants that use filtered mapping as shown in the Figure 4.6, a row representing an instance of class A (or its descendant) will contain NULLs in columns of attributes from class B and vice versa. Condition that solves this issue is marked (1) in the algorithm.

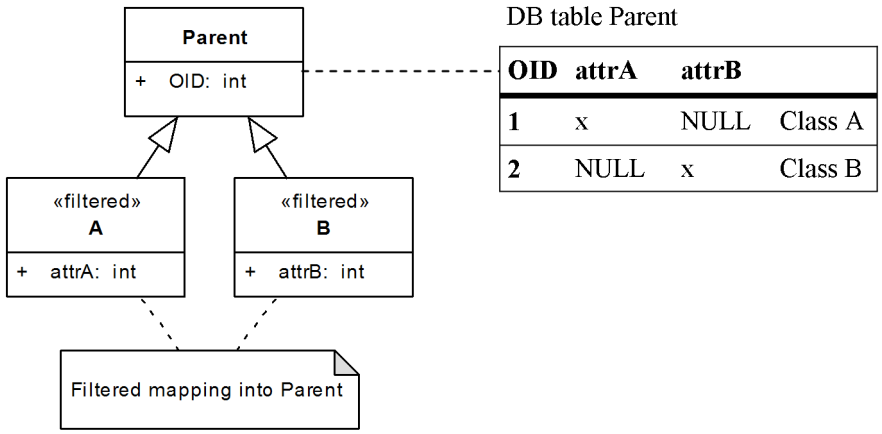

Figure 4.6: Inserting objects using filtered mapping

The table in Figure 4.6 further illustrates the need for classes that use filtered mapping to derive from `OidObject`. When loading an object with OID "1", the object-relational layer has to know which class to instantiate and only `OidObject` provides such database structures that store this information.

Same algorithm as in Figure 4.5 is used for storing changes from objects that already have a database identity (Update_Row method). Changes are propagated as SQL UPDATEs with the WHERE generated from list of `KeyAttributes` and their values. For OID objects this means that the WHERE filters only by the objects' OID value. Condition (1) from the algorithm is slightly modified in a way that no NULLs are inserted but simply the NULL-attributes are not updated. Deleting data is a trivial case - Delete_Row sends one SQL DELETE command to delete the requested row. The DELETE command uses same WHERE clause as the UPDATE does.

You can see that most of the work is done by correctly pre-generating the lists from the previous section, in particular the `MappedAttributes` and `AllParents` lists. They actually contain projection of the class model on the database structures according to the mapping types used.

4.3.4.2 Storing objects that use ADT mapping

As for the ADT mapping, the algorithm for inserting, updating or deleting is even simpler (see (1) in Figure 4.4). IOPC 2 relies on the object-relational features provided by the underlying database. Insert operation usually creates a new instance of relevant ADT and inserts it into a ADT hierarchy base table[4]. Moreover due to significant differences between DBMSs in the area of object-relational features, most of the work is delegated to the database driver side. IOPC 2 database drivers are responsible for generating the needed database structures including the ADTs. The library doesn't take any special steps to support multiple inheritance, so it is fully in charge of the database driver. If the ORDBMS doesn't support multiple inheritance of ADTs, it doesn't make much sense to bend the database engine to support it.

Example 4.3.5 in Section 4.3.2 discussing ADT mapping illustrates how the needed database structures may look like. If we assume that these database structures are already created, inserts, updates and deletes are reduced to simple calls to the database access layer. If using the Oracle 10g database driver, these calls are only delegated further to the `insert_object`, `update_object` or `delete_object` static methods of corresponding ADTs.

The only additional task the ADT mapping requires is to decide which persistent class attributes will be mapped to ADTs. The simplest scenario is if all of the persistent classes starting with `OidObject` use ADT mapping - all attributes will be mapped to corresponding ADTs and point (3) in the general algorithm (Figure 4.4) will be skipped. However, if the ADT mapping hierarchy is started at a lower level and derives for example from a class that uses vertical mapping, the attributes defined in classes "above" the root(s) of the ADT mapping hierarchy must be skipped and must not be propagated to the corresponding ADTs (as described in Section 4.3.2).

4.3.4.3 Loading objects from database

Figure 4.7 illustrates how object instances are loaded from database.

```
if O uses ADT mapping
  Load_Object(O)
else if O uses horizontal or vertical mapping (O has a  ↩
   corresponding db table) then
  Load_Row(O, class of O)
end if

foreach class C in AllParents do
  Load_Row(O, C)
end
```

Figure 4.7: Iterative loading algorithm

[4] In DB2 the instance would be inserted into a table associated with that ADT [**?**].

The algorithm is very similar to the previous one (Figure 4.4). Retrieving instances in such way - class by class - is quite inefficient. The algorithm can be implemented as pre-generated views executing queries that join the involved tables. The views (simple and polymorphic views) are generated for descendants of `OidObject`. Instead of sending multiple SELECTs to retrieve data directly form the mapping tables, only one SELECT executed on a simple view is sufficient. The algorithm just loads values of `AllPersistentAttributes` from corresponding simple view. Source code of these views can be found in [?].

Iterative algorithm (without views) is used for classes that derive from `DatabaseObject` (and not from `OidObject`) and for instances that are requested to be exclusively locked in database. Rows belonging to such instances must be locked one-by-one usually by sending SELECT FOR UPDATE statements.

Modifying these views to be updatable for inserts, updates and deletes was considered, but not included into this release. Current DBMSs place many restrictions on the way how updatable views should be constructed and used. Most of them etirely disallow using joined tables in updatable views; Oracle for example supports updatable join views, but allows only columns from one table to be updated at a time[5].

4.3.4.4 Loading classes that use ADT mapping

Again, loading instances of classes that use ADT mapping is a very simple task. In both Oracle and DB2, single SELECT statement can load all attributes of an ADT. If all classes of our example class model used ADT mapping, a SELECT statement to load a single `Student` instance would look like Example 4.3.6.

Example 4.3.6 Loading objects using ADTs from an Oracle database

```
SELECT
  TREAT(VALUE(X) AS tStudent).OID AS OID,
  TREAT(VALUE(X) AS tStudent).CLASSNAME AS CLASSNAME,
  TREAT(VALUE(X) AS tStudent).age AS Person_age,
  TREAT(VALUE(X) AS tStudent).name AS Person_name,
  TREAT(VALUE(X) AS tStudent).studcardid AS Student_studcardid,
  TREAT(VALUE(X) AS tStudent).supervisor AS Student_supervisor
FROM OidObject O
WHERE O.OID = 2
```

Because, as mentioned before, ADT mapping hierarchies can be started at lower level and its roots can derive from non-ADT-mapped classes, either iterative algorithm from Section 4.3.4.3 or pre-generated views are used to retrieve object instances from database.

4.4 Conclusion

If we look at the table from Section 3.4 we may see changes, mostly improvements in many of the listed areas:

- IOPC 2 added another mapping type - the ADT mapping and further enhanced the O/R mapping capabilities.

[5] see [10] - Database Administrator's Guide

- IOPC 2 supports read-only databases, it even supports inheritance between non-OID classes (tables) if the involved tables share the same set of keys.
- The library is based on a modular design allowing it to be used in several configurations and with various database drivers.
- IOPC 2 provides a reflection feature which allows users to inspect the structure of reflection capable classes at run-time. Reflection works almost transparently as it needs no descriptive macros from the application developers.
- Reflection provides metamodel description for the O/R mapping mechanism which uses them to provide effortless object persistence services to the application developers. However, there are some limitations when compared to the IOPC predecessor which result from the exchange of OpenC++ for GCCXML - for example the need to use enhanced data types in some scenarios. More on this can be found further in the text. *TODO: pridat odkazy*

Remaining topics are discussed in the following chapter. Thesis conclusion *TODO: odkaz* contains final library comparison and evaluation.

Chapter 5

Architecture of the IOPC 2 library

5.1 Architecture overview

Structure of the IOPC 2 project including library dependencies is displayed in Figure 5.1. IOPC 2 consists of a stand-alone application `iopcsp` and a set of shared libraries that are used at run-time.

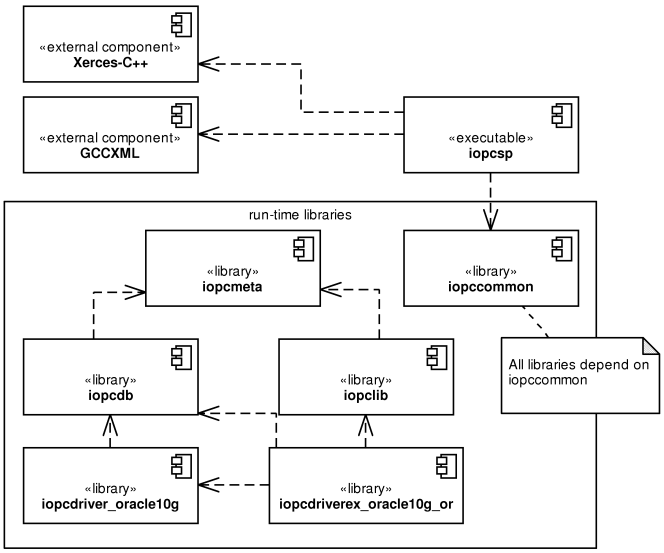

Figure 5.1: Overview of the IOPC 2 architecture

The following list provides a short description of each of the components.

- `iopcsp` is a stand-alone application that reads GCCXML output, extracts description of reflection-capable classes and generates source code of metamodel libraries. Metamodel libraries are later linked to the user application.
- `iopccommon` contains shared services for other parts of IOPC 2 including `iopcsp`. These services include tools for thread synchronization, tracing, logging or exception handling.
- `iopcmeta` is the reflection interface provided by the IOPC 2 library. It contains classes that whose instances describe metamodel of the source code processed by `iopcsp`.

- `iopcdb` supports the database layer modularity by providing a `DriverManager` facility by which loaded database driver register themselves. `iopcdb` provides access to these drivers via a unified interface and allows user applications to execute SQL statements.
- `iopcdriver_oracle10g` is Oracle 10g database driver for the IOPC 2 library.
- `iopcdriverex_oracle10g_or` is an extension to the Oracle 10g driver that adds object persistence capabilities to it.
- `iopclib` puts everything together into an O/R mapping layer. `iopclib` implements the O/R mapping algorithm as described in previous chapter. Further it provides persistent object caching features (originating from the POLiTe 2 library), script generation and querying functionalities and many more.

There are additional components that are not displayed in Figure 5.1. They are not directly related to the core functionality, but can be useful in the build process either of user applications or of the IOPC 2 library itself:

- `iopcsp_compile.sh` is a Bourne shell script that can be used with the Eclipse/CDT IDE[1] for managed build using the IOPC 2 library.
- `iopcsp.sh` is a Bourne shell script for use with standard Makefile to build application that uses the IOPC 2 library.
- `test` is a project that tests most of the library functionalities.

Section 4.1 stated that IOPC 2 can be used in various configurations. These configurations are realised by linking subsets of the presented libraries together with the user application. Available configurations are:

- Common services - `iopccommon`.
- Reflection library - `iopccommon`, `iopcmeta`. Source code must be processed using `iopcsp` (also applies to the following two configurations).
- Database access library - `iopccommon`, `iopcmeta`, `iopcdb` and a database driver - for example the `iopcdriver_oracle10g`.
- Object persistence library - `iopccommon`, `iopcmeta`, `iopcdb`, `iopclib`, a database driver supporting all required O/R mapping features - for example the `iopcdriver_oracle10g` along with its extension `iopcdriverex_oracle10g_or`.

5.2 Common services

The `iopccommon` library contains support classes and utilities that are used by other parts of the library. The classes can be divided into three groups:

[1] http://www.eclipse.org/, http://www.eclipse.org/cdt/

5.2.1 Thread synchronization classes

The thread synchronization classes were taken from the POLiTe 2 library. Synchronization primitives encapsulated by these classes are:

- Mutex (non-recursive) - the `Mutex` class.
- Mutex (recursive) - the `Lock` class.
- Conditional variable - the `CondVar` class.
- Read-write lock - the `RWLock` class.
- Read-write-exclusive lock - the `RWXLock` class.

5.2.2 Class metadata

Although the class metadata container is implemented as a generic structure in the `iopccommon` library, it is primarily designed to be used with the reflection features. Therefore we will provide its description later, in the Section 5.4.3.

5.2.3 Utilities

Utilities offered by the `iopccommon` library include reference counted pointer, various string manipulation methods, tracing and logging. Logging is encapsulated by the `LogWriter` singleton class.

Log messages have a severity level attached which helps to determine how critical the message is. Message filtering based on the level is also possible. Tracing macros defined in the same file as the `LogWriter` class are used through the IOPC 2 library allowing developers to dump detailed debugging information in case of application crash or malfunction. These macros can be enabled or disabled by setting appropriate filter in the `LogWriter` class or by (un)commenting macro options IOPC_DEBUG, IOPC_TRACE in the same file and by recompiling the whole IOPC 2 library. This way a diagnostic version of the library can be created.

5.3 Database access

One of the IOPC 2 library basic requirements is database independence. This can be achieved by separating the database access logic behind a shared interface.

The original POLiTe and the POLiTe 2 libraries access the database via such interface, the problem is that they contain several SQL statements outside the database layer code. These SQL statements are written for the Oracle 7 RDBMS and porting the library to another database system means to find and update them. In the end, recompilation of the whole library is needed.

The IOPC library tried to solve this issue by providing a new interface - the `DatabaseSqlStatements` class. Implementation of this interface encapsulates all SQL fragments needed by the library. Moving the library to another RDBMS became quite easy, but not as easy as it could be, because whole library needs to be recompiled every time a new driver is added or updated.

In the IOPC 2 library, database drivers are separated into shared libraries. The library provides a facility for managing and accessing these drivers. Adding or updating a database driver doesn't involve any modifications in the library code or its recompilation. Users can use more than one driver at a time. Even all SQL code is separated into the drivers.

If the user application needs to communicate with a database, it must be linked with the `iopcdb` library and with at least one driver. Currently, only Oracle 10g driver `iopcdriver_oracle10g` is implemented. Application can however use any number of linked drivers at a time. Drivers may be enhanced with several extensions. Extensions provide additional functionalities to them. Again, only one driver extension is implemented at the time - the `iopcdriverex_oracle10g_or` extension. It adds the `iopcdriver_oracle10g` driver support for O/R mapping.

5.3.1 Basic classes

Figure 5.2 shows an overview of basic classes and subsets of their methods.

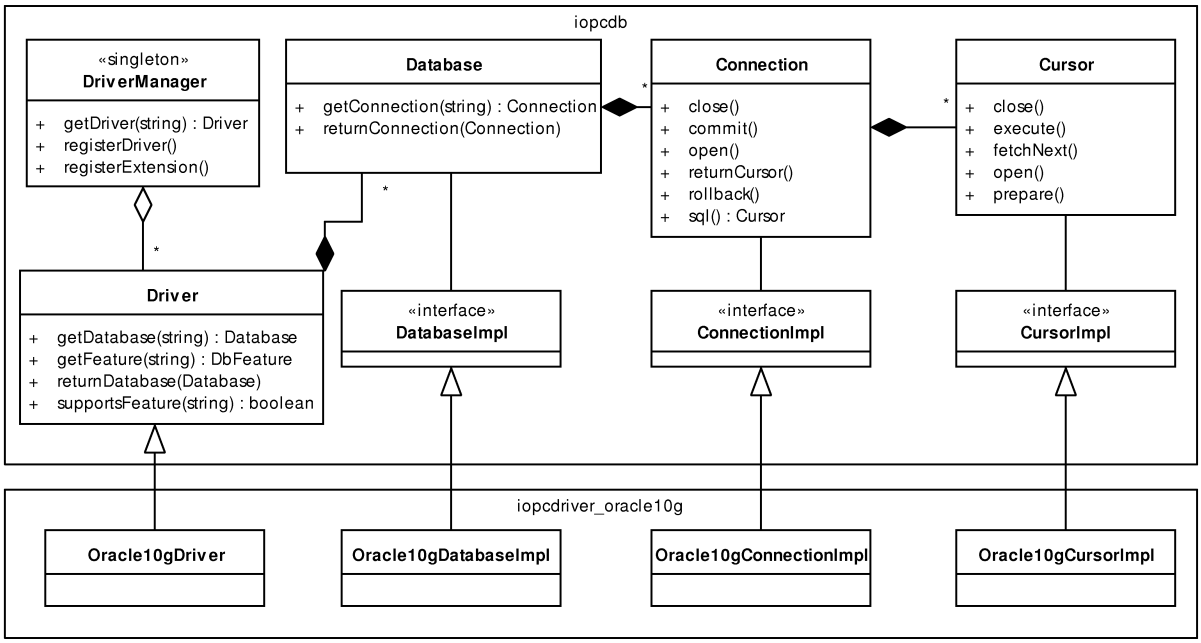

Figure 5.2: Basic classes of the database layer

`DriverManager` is a central part of the driver model. A single instance of this class holds references to all drivers loaded in the user application. Users can obtain reference to a database driver by calling its method `getDriver(name)`. Database driver is represented as an instance of the `Driver` class.

Next three classes - the `Database`, `Connection` and `Cursor` can be found in all previous versions of the IOPC and POLiTe libraries. Their interface was slightly modified in IOPC 2 and implementation was moved from their subclasses to subclasses of the abstract classes (interfaces) `DatabaseImpl`, `ConnectionImpl` and `CursorImpl`. Each database driver must implement these interfaces.

The reason for almost duplicating the `Database` and `Cursor` interfaces is that they are used in a decorator pattern to modify the effect of their interfaces at run-time. As you may see in Figure 5.3, the classes `CachedDatabase` and `CachedConnection` are designed to decorate the `BasicDatabase` and `BasicConnection` classes. `BasicConnection` and `BasicDatabase` implement the basic database independent functionality, mostly the connection and cursor management. Database dependent operations are delegated to database

drivers using the `-Impl` interface classes. The `CachedDatabase` and `CachedConnection` classes decorate behaviour of several fundamental methods like commit or rollback by performing a caching related operations - for example writing modified local copies to database before commit.

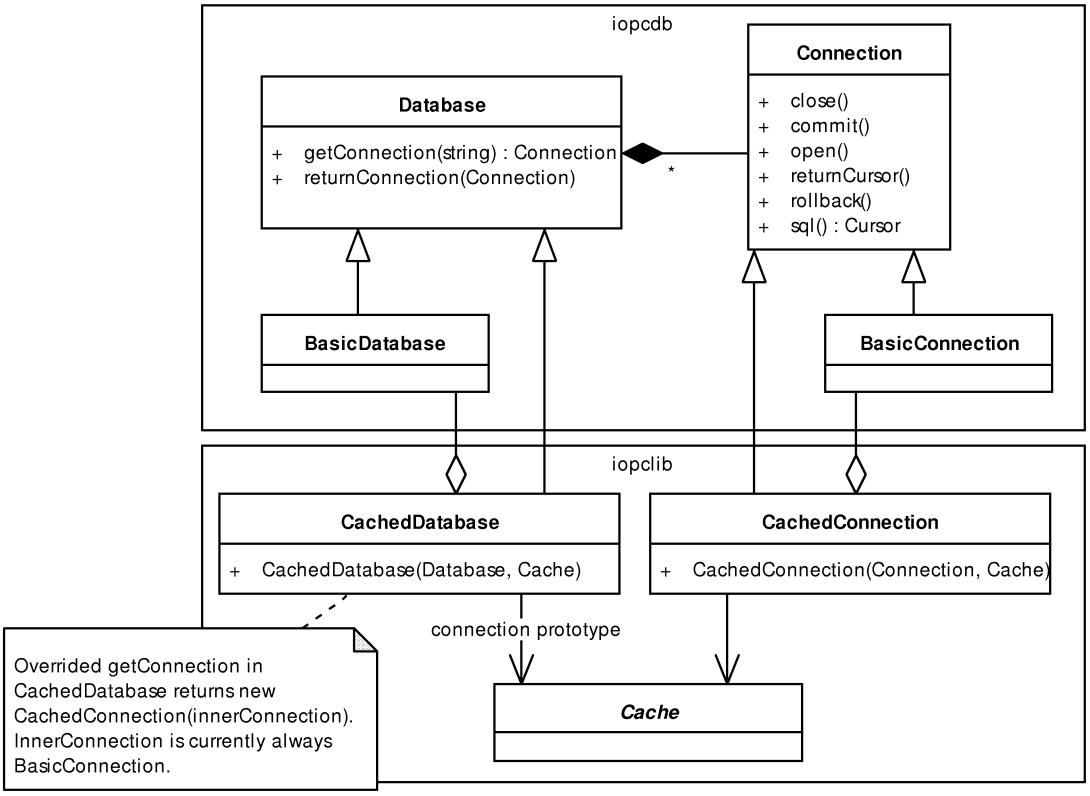

Figure 5.3: Using the decorator pattern

Example 5.3.1 shows how a connection to Oracle database instance "XE" can be made. First we create only a simple connection allowing only basic SQL statements to be executed, then we release this connection and a new one using the `CachedConnection` decorator class. This connection can be used with the O/R mapping features, see later.

Example 5.3.1 Connecting to an Oracle database in IOPC 2

```
Database* db = DriverManager::getDriver(
  "IopcOracle10g").getDatabase("XE");
Connection* conn = db->getConnection(
  "username=iopc;password=iopc");
conn->sqlNonQuery("INSERT INTO test VALUES('Test')");

// Connections need to be released when no longer needed
db->returnConnection(conn);
DriverManager::returnDatabase(db);

// CachedDatabase returns CachedConnection instances
// (no need to decorate them)
CachedDatabase* db2 =
  new CachedDatabase(DriverManager::getDriver(
  "IopcOracle10g").getDatabase("XE"), new VoidCache());
CachedConnection* conn2 =
  (CachedConnection*)db2->getConnection(
  "username=iopc;password=iopc");
conn2->open();
...
```

5.3.2 Driver features

Driver features reflect differences between database systems. Different database systems provide different features or services. In IOPC 2, interfaces to such services(features) can be grouped into units called driver extensions. Modules of the IOPC 2 library define interfaces (representing the features) and database driver creators optionally implement them to their drivers or grouped in driver extensions. The library then asks the drivers what features they offer. Based on the answers it enables or disables some of its functionalities provided.

Driver extensions are shared libraries that are linked together with the drivers and user applications. One such extension implemented for the Oracle 10g driver is `iopcdriverex-_oracle10g_or`. It enhances the `iopcdriver_oracle10g` Oracle 10g driver with the support for object-relational mapping.

The features are descendants of the `DbFeature` abstract class. Features declared in the `iopcdb` library are:

- `SqlStatementsFeature` - declares methods that are used to generate basic SQL statements like the INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE and others.
- `TypeMappingFeature` - declares methods that are used to translate C++ data types to the SQL.

As part of the `iopclib` library, there are additional features declared that are needed for the object-relational mapping proccess:

- `MappingStatementsFeature` - declares methods that are used to generate shared SQL statements needed for the object-relational mapping.

- `ORStatementsFeature` - declares methods that are used to generate SQL statements supporting object-relational mapping to purely relational tables. (Relational databases)
- `ObjectStatementsFeature` - declares methods that are used to generate SQL statements needed by the ADT mapping type. (Object-relational databases)

Figure 5.4 illustrates how the driver features and extensions are used by the Oracle 10g database driver.

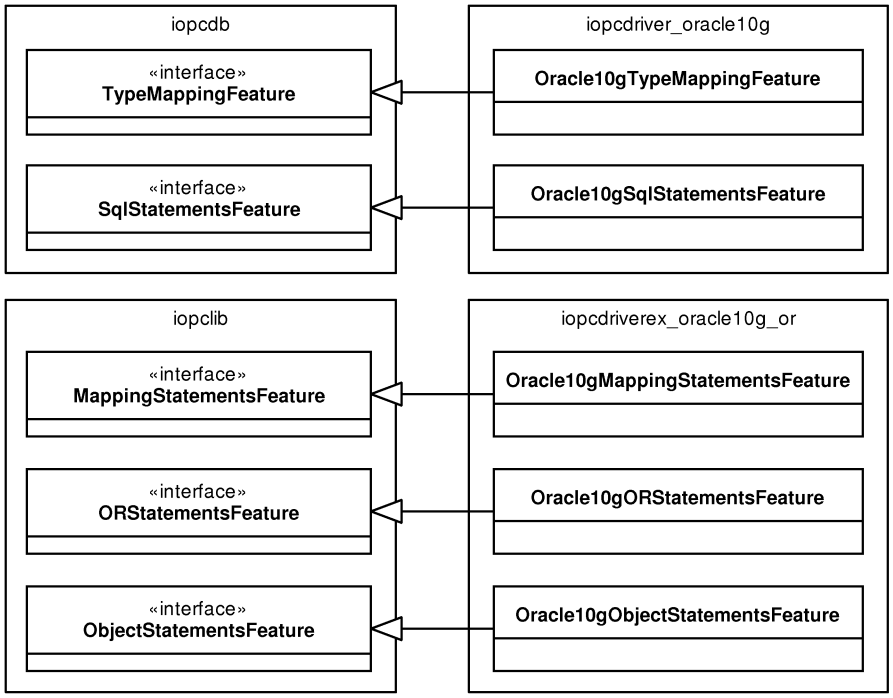

Figure 5.4: Oracle 10g database driver extensions and driver features

5.4 The metamodel

Metamodel description is created for all descendants of the `Object` class. Source files must be processed using `iopcsp` and the result must be compiled and linked together with the `iopcmeta` library and the user application. The metamodel can be then inspected using classes and structures defined in `iopcmeta`.

5.4.1 The metamodel classes

Figure 5.5 provides an overview of these classes.

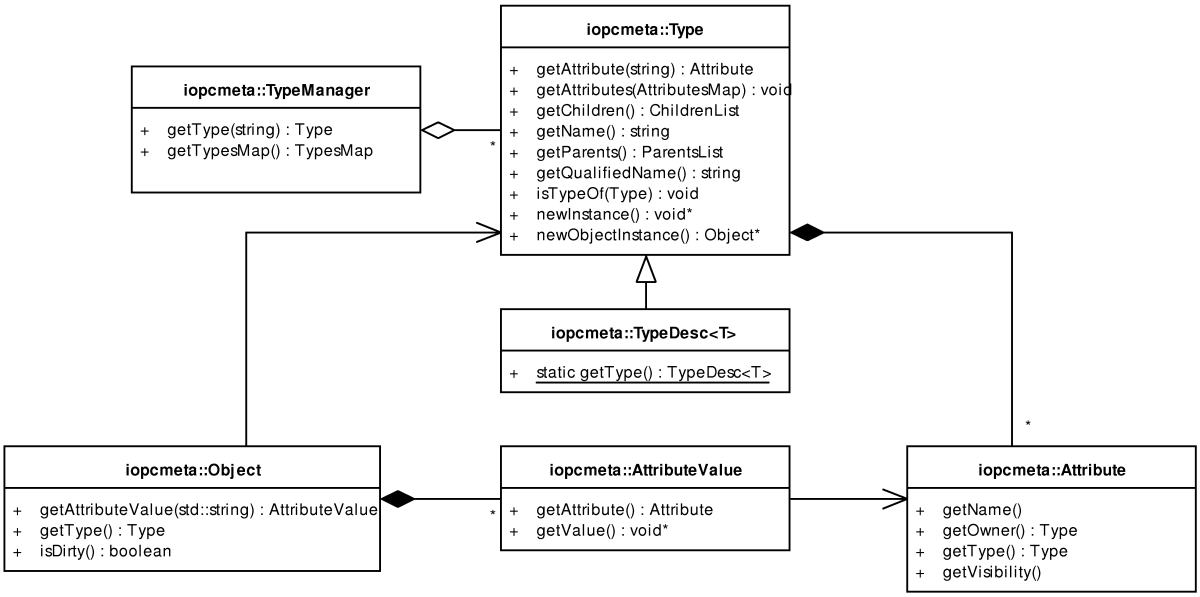

Figure 5.5: The iopcmeta classes

The `Type` class and its descendant, the `TypeDesc<T>` template class, are the central point of the reflection mechanism. Instances of the `TypeDesc<T>`specializations represent reflection-capable classes and all basic C++ types. There is only one instance of the template specialization for a reflection-capable class or basic type. As mentioned before, the concept is similar to prototypes in the POLiTe libraries and to `IopcClassObjectImpl` from the IOPC library. However, classes defined in `iopcmeta` do not provide any object-relational mapping capabilities.

The `TypeDesc<T>` template class uses technique called *type traits* to encapsulate various type-dependent data and to provide access to the metamodel description at compile-time in a static way. For example, given the class `User`, the correspondent type trait is `TypeDesc<User>` and the solitaire type instance is accessible via the `TypeDesc<User>::getType()` call. Type traits are defined in a manner illustrated in Example 5.4.1. As you can see, this is a good way how to unify the metamodel interface for C++ basic types along with the object types declared in the IOPC 2 library.

Example 5.4.1 Type traits

```
// Generic template
template<typename T>
class TypeDesc : public Type {
public:
  virtual const std::string getName() const;
  ...
}

// Template specializations for particular types
template<>
class TypeDesc<int> : public Type {
  virtual const std::string getName() const { return "::int"; }
}

template<>
class TypeDesc<Object> : public Type {
  virtual const std::string getName() const { return "iopc::Object ↩
     "; }
}
```

`TypeManager` is a singleton class which maintains a list of the instances of the `Type` subclasses. As the call to `TypeDesc<T>::getType()` is a method how to gain access to the metamodel in a static way, the call to `TypeManager::getType(std::string typeName)` can be used to obtain the type information from a run-time constructed type-name string.

The `Object` class as a predecessor of all reflection-capable classes doesn't offer much functionality. It provides infrastructure for the dirty status tracking and list of `AttributeValue` objects that allow developers to modify object attribute values using the reflection. This leads us to the `Attribute` class which instances represent attributes of a particular class. `Object` also contains a `getType()` method which is the last way leading us to the corresponding `Type` instance.

Example usage of reflection can be found in Example 5.4.2. The example prints a brief description of the `Person` class we defined earlier. For more examples, see Section A.2.

Example 5.4.2 Using the reflection interface

```
const Type& t = TypeDesc<Person>::getType();
cout << "Type name: " << t.getQualifiedName() << endl;
cout << "Attributes: " << endl;
const Type::AttributesMap& attributes = t.getAttributes();
for(Type::AttributesIterator it = attributes.begin(); it !=  ↩
   attributes.end(); it++) {
  cout << it->first << ": " <<
    it->second.getType().getQualifiedName() << endl;
}
// For simplicity we assume that multiple inheritance
// is not used
cout << "Parent: " << t.getFirstParent().getQualifiedName() <<  ↩
   endl;
cout << "Children: " << endl;
const Type::ChildrenList& children = t.getChildren();
for(Type::ChildrenIterator it = children.begin(); it != children. ↩
   end(); it++) {
  cout << (*it)->getQualifiedName() << endl;
}

// Output is:
Type name: ::Person
Attributes:
age: iopc::EShort
name: iopc::EString
Parent: iopc::OidObject
Children:
::Employee
::Student
```

5.4.2 Enhanced data types

Mostly because of the need of dirty status tracking for the object-relational layer a set of classes encapsulating C++ built-in types was created. The basic requirement is that starting from a specific time, an object needs to be aware if value of any of its attributes has changed. There are several transparent ways how to fulfil this requirement (like preserving a copy of the object and comparing it with the original when needed) and one of the simplest is to have the attributes to be aware of their own changes. This is achieved in the library by creating new data types that mimic behaviour of built-in types. These types are called *enhanced data types*. Developers are, however, not forced to use these types, they can use built-in types and manually notify the object that its state has changed.

POLiTe libraries solved this problem by using setter methods generated from descriptive macros. The additional task of the setter methods is simply to modify the dirty status flag of the containing object. IOPC library took advantage of the `OpenC++` parser's ability to process entire source code and changed every relevant assignment operation to use generated setter methods similar to the POLiTe ones.

Second goal was to create a common interface for data types of class attributes that will unify the way how they are mapped to the database. Thanks to enhanced data types, string values, numeric values or references to other objects (using the `DbPtr<T>` class as attribute)

can be treated in a similar way in the database drivers and in the database part of the library (`i-opcdb`). Enhanced data types are also given the ability to represent NULL values. The NULL status becomes part of an attribute itself as opposed to the preceding versions of the library. Such design helps to further minimize impedance mismatch by unifying data domains of C++ data types and column types of database tables. Structure of the enhanced data type classes is displayed in Figure 5.6.

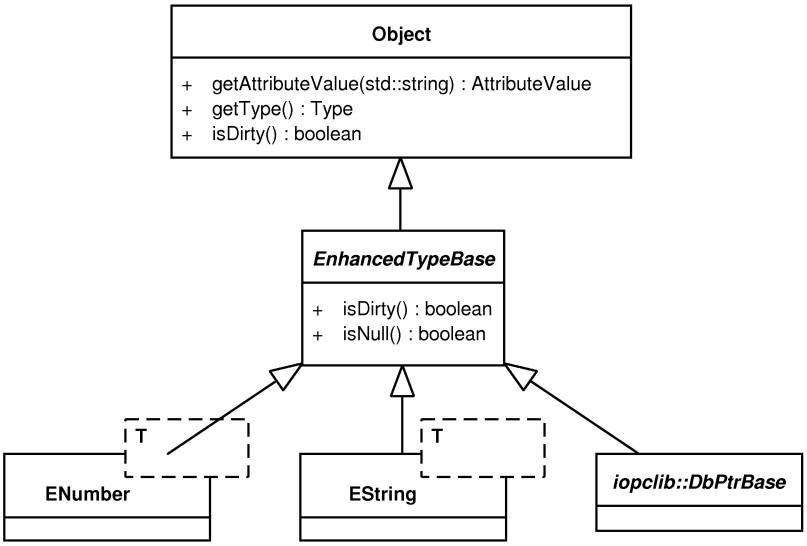

Figure 5.6: Structure of the enhanced data type classes

Enhanced data types are designed to be used like built-in C++ types. Example 5.4.3 illustrates this feature along with the NULL values and dirty status tracking.

Example 5.4.3 Java annotations and C# attributes

```
EInt a = 8;
a = a + 2;
int b = 2 + a;
EFloat d = 2.5;
d = (d + 2.5 + a) / 2;
EDouble c = (a + 2)*(a + 3);
cout << "a: " << a << " b: " << b << " c: " << c << " d: " << d << ↩
    endl;
d.setNull();
cout << "a.isDirty(): " << a.isDirty() << endl;
cout << "c.isDirty(): " << c.isDirty() << endl;
cout << "d: " << d.toString() << endl;

Output is:
a: 10 b: 12 c: 156 d: 7.5
a.isDirty(): 1
c.isDirty(): 0
d: null
```

5.4.3 Class metadata

TODO: Metadata loader - interface v iopcMeta, bude to volano na spravnem miste v inicializaci Modern languages like Java or languages based on the .NET platform provide language constructs that allow programmers to add custom information to class metadata. These constructs - attributes in C# or annotations in Java - have declarative meaning and do not directly impact semantics of a program. Information stored in attributes or annotations can be accessed at run-time using reflection. Class metadata concept goes well with object-relational mapping layer as the information about into which database tables to map, which mapping type to use or how the table columns should be named have also declarative character. Such information don't change at run-time.

Example 5.4.4 Java annotations and C# attributes

```
// Demonstration on the Hibernate library
// http://www.hibernate.org
// Java
@Entity
@Table(name="tab_person")
public class Person implements Serializable {
  @Column(name = "name", nullable = false)
  public String getName() { ... }
}
// C#
[Class(Table="tab_person")]
class Person
{
  [Property(Name = "name", NotNull=true)]
  public string Name { ... }
}
```

IOPC 2 library also implements this concept, but in a very simple way. In `iopccommon` there is a `MetadataHolder` class which, when instantiated, behaves like a dictionary. `MetadataHolder` can contain data of several types - integer or boolean values, void pointers or strings. Data stored in the dictionary are identified by a string key.

Classes that describe the metamodel like `Type` or `Attribute` derive from `MetadataHolder`. Their instances inherit the dictionary features which simulates the behaviour of the Java or .NET language constructs. However, a significant difference is that the metadata is appended at run time. It can be hardcoded into a *class initialization method* (see the Example 5.4.5 and Section A.3 for further explanation) or loaded from a configuration file during the library initialization using the supplied `TextFileMetadataLoader` (also described in Section A.3.

Example 5.4.5 Example usage of metadata in the IOPC 2 library

```
class Person : public iopc::OidObject {
public:
  iopc::EString name;
  // the class initialization method
  static void iopcInit(iopc::Type& t) {
    t["db.table"].setStringValue("person");
    t.getAttribute("name")["db.column"].setStringValue("name");
    t.getAttribute("name")["db.type.notNull"].setBoolValue(true);
  }
  ...
};
```

5.4.4 iopcsp

`iopcsp` is a standalone application, already introduced in the Section 4.2. The application processes XML output of GCCXML and generates C++ files containing structures that describe classes from files originally processed by the GCCXML. To get better understanding of what the application does, we will repeat the description of the metamodel generation process in more detail:

- The process is invoked on every CPP file of a user's project.[2]
- GCCXML reads the input CPP file and generates a XML file describing all declarations from the input file and from all files included (it runs the C preprocessor). In the provided implementation this output file is given a filename "input-filename.cxml".
- `iopcsp` reads the CXML file and transforms it into a couple of C++ compilable source files named "input-filename.ih" and "input-filename.ic". `iopcsp` searches for all descendants of the `Object` class and for each of them it writes several lines of macro code into both IH and IC files. The macro code in the IH file expands to a full definition of the `TypeDesc<T>` template specialisation whereas the IC file contains the template instantiation and a static instance of a `TypeRegistrar` class which registers the type by the `TypeManager` singleton. The registration is initiated when the user application starts.
- Last step may vary between scenarios. All IC and IH files compiled into object files can be archived using the `ar` program and the created archive, as a "*type catalogue*", can be linked to the rest of the user application. Another approach is to create archives containing the IC/IH object files together with the object files of the original input CPP sources.

The process is customisable and currently is driven by a shell script. The script can be modified by developer needs and even integrated into their favourite IDEs. An example is provided with this thesis illustrating how to use it with simple makefile or integrated into Eclipse's CDT[3] managed build system.

Managed build system generates project build files automatically based on the project resources and user settings. Build file generation can be customized by users. Example project provided with this thesis changes the default compile command executed on each compilation

[2] This is not necessarily true because the build process can be modified to accept only a subset of the project files.

[3] http://www.eclipse.org/cdt/

unit in the project (`g++`) to `iopc_compile.sh` shell script call. `iopc_compile.sh` runs the steps described above these paragraphs and compiles the compilation unit using `g++`.

The example project provided on the enclosed DVD is named `example_managed`. During its build, a type catalogue archive is created and in the end of the build process the catalogue is lined to the resulting binary. Second example provided - `example` - illustrates a project which is built using standard Makefiles. This time, a type archive for every compilation unit is created.

5.4.5 Usage transparency differences from IOPC

IOPC uses OpenC++ source-to-source translator which gives it full control over the source code structure. IOPC processes not only class declarations, but also method bodies and everything else. On the other hand, IOPC 2 uses GCCXML which processes only declarations. GCCXML output doesn't contain function bodies, therefore it is not possible to modify the original source code and dump the modified version into a new file that will be compiled. Such approach could work if the class declarations were in separate files and completely without any function code. This solution was however rejected as it led to other source code modifications and to build process complications.

Solution used in IOPC 2 is less intrusive as it generates only new structures (the type descriptions - `TypeDesc<T>`) into additional compilation units. It doesn't modify the original source code in any way and to read attribute values from reflection-capable instances it uses only "safe" C++ techniques [4]. There are two restrictions that somehow violate the usage transparency and that the developers should keep in mind when creating the reflection-capable (or persistent) classes.

The first problem is, how to access private or protected members in reflection-capable classes from the type descriptions. There are several ways how to accomplish this task, but all aside from direct memory access to these members are either not considered "safe" by the author or involve modification of the class declaration. For IOPC 2 the latter option was chosen. Developers need to include a friend class declaration as shown in Example 5.4.6 if they want to access such members via reflection or to make them persistent using the O/R mapping provided. The friend declaration should be understandable for developers as it clearly says what it does (provides access to the private members to some other entity - the reflection). It is not needed if all such members are declared as public.

[4] No direct memory access to these members, etc.

Example 5.4.6 Java annotations and C# attributes

```
class Person : public iopc::OidObject {
private:
  EShort age;
  EString name;
public:
  void setData(std::string name, short int age) {
    this->name = name;
    this->age = age;
  }
  std::string getName() const { return name; }
  short int getAge() const { return age; }

  // Needed
  friend class TypeDesc<Person>;
};
```

Second point concerns the dirty status tracking of persistent objects. All IOPC 2 library predecessors used getter and setter methods for this task. However, these methods couldn't be used in IOPC 2 because it would involve source code modification. For this reason, the enhanced data types were created - see Section 5.4.2. They provide automatic dirty status tracking and they can also represent the SQL NULL values. Developers do not need to use the enhanced data types if they manually call the `Object::setDirty()` method to tell the library that value of the object has changed. In contrast, if developers need to set or read NULL values using the O/R mapping, they must use the enhanced data types as there is no alternative for such task in the library.

5.5 The persistence layer - iopclib

`iopclib` puts together all other components into an object-relational library. It uses `iopcdb` with appropriate database drivers (`iopcdriver_oracle10g`) for database access. `iopclib` requires the drivers to have the object-relational features implemented and linked (the `iopcdriverex_oracle10g_or` extension). Metamodel description is generated using the GCCXML and `iopcsp` and is accessed using the `iopcmeta` library. The `iopccommon` library is implicitly used by all of these components and also by `iopclib` itself.

5.5.1 Database mapping

`iopclib` defines the base classes `DatabaseObject` and `OidObject` as described in the Section 4.3.1. The OID data type is defined by a typedef, so it can be changed in the future. Now it defaults to `EULong`. If we want the non-oid objects to be cached, we need a way how to describe their identity. In `iopclib` the identity of persistent objects is expressed as an instance of the `KeyValues` class. `KeyValues` contain a list of key-value pairs which represents any single-attribute or composite (multi-attribute) key.

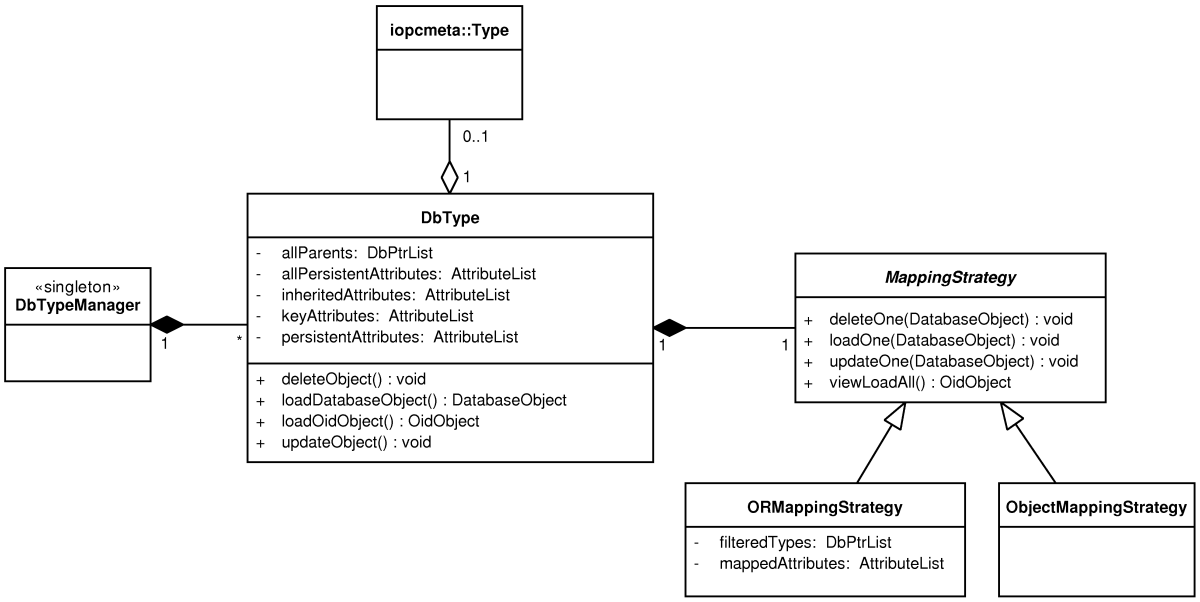

Figure 5.7: Classes involved in the database mapping process

The idea behind the O/R mapping part is to wrap each type representing a persistent class[5] into a class that will provide persistence related tasks. Such class has been named `DbType` as displayed in the Figure 5.7. For each persistent class type there is exactly one instance of the `DbType` class. These instances are managed by the `DbTypeManager` singleton.

`DbType` class implements the mapping algorithm described in the Section 4.3.4. The algorithm has been split between the `DbType`, `ObjectMappingStrategy` and `ORMappingStrategy` classes. `DbType` performs the table-level part of the algorithm while the strategy classes implement the attribute-level operations - this means the Load_Object or Insert_Object[6] methods for the ADT mapping type (`ObjectMappingStrategy`) and Load_Row or Insert_Row[6] methods for the other mapping types (`ORMappingStrategy`). The lists described in Section 4.3.3 needed by the algorithm can be found in these classes.

When one of the main CRUD methods of `DbType` is invoked, the program iterates through the inheritance hierarchy of the related class type according to the algorithm and calls the `ObjectMappingStrategy` or `ORMappingStrategy` to perform the attribute-level operations. The strategies in turn call the database driver's `ObjectStatementsFeature` or `ORStatementsFeature` to generate SQL statements that will be executed. If possible, these statements are cached and not re-created each time. After that, the SQL statements are sent to the underlying DBMS via the database driver being used.

Another important class defined in `iopclib` is the `ScriptsGenerator`. `ScriptsGenerator` has only two methods - `getDbCreateScript` and `getDbDrobScript`. As the method names suggest, the class generates SQL CREATE and DROP scripts which create or delete database schema required for the O/R mapping to work. The generator first creates a list of topologically ordered persistent classes according to their inheritance dependencies and then calls the `MappingStatementsFeature` for each of them in order to generate the scripts. Example 5.5.1 how to prepare required database schema.

[5] Class that derives from `DatabaseObject`

[6] And its update and delete variants.

Example 5.5.1 Using the ScriptsGenerator to create required database schema.

```
vector<string> script;
script = ScriptsGenerator::getDbCreateScript(conn->getDriver());

for(vector<string>::const_iterator it = script.begin();
  it != script.end(); it++) {
  conn->sqlNonQuery(*it);
}
```

5.5.2 Persistent object manipulation

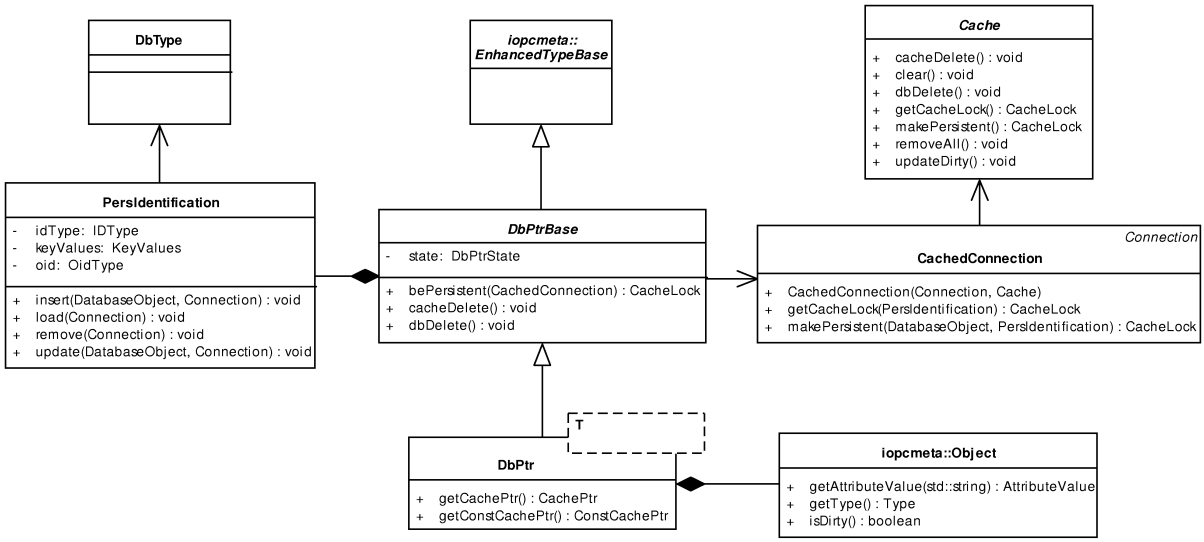

Figure 5.8: Classes manipulating with persistent objects

Most classes presented in this and in the next section are adapted from the POLiTe 2 library, so we will provide only a brief overview of their structure and of the way how they interact. For detailed description, see [1].

Instances of the `PersIdentification` class represent identity of persistent objects. The class encapsulates both the OID and the `KeyValues`. It also provides access to CRUD methods defined in `DbType`. `PersIdentification` instances are used as keys in the cache layer containers and in this context they represent the only communication channel between the cache layer and the O/R mapping services - see the sequence diagrams below.

As in the POLiTe 2 library, developers must use indirect references to manipulate database object in their transient or persistent state. IOPC 2 uses exactly the same concept as described in Section 3.2.3.

The indirect references - database pointers - are represented by instances of the `DbPtr<T>` template class. `DbPtr<T>` can point to database object instances in one of the following states:

- Transient instances which are not yet managed by a cache. The owner of these instances is the database pointer. If the pointer is destroyed before delegating the object ownership to a cache, the object is also destroyed.

- Instances managed by a cache. They may or may not already have a database representation.
- Persistent instances that are not present in a cache. The pointer contains only their identity description (a `PersIdentification` instance).

`bePersistent` is the method that transfers the object ownership to a cache. It has only one argument - a database connection (`CachedConnection`). The cache that is associated with this connection becomes the new owner of the object.

Example 5.5.2 Using the bePersistent method

```
// Creates a new transient Person instance
// owned by the database pointer p
DbPtr<Person> p;
p->name = "Mary Major";
p->age = 60;
// Transfers the ownership to the cache associated
// with the connection conn.
p.bePersistent(conn);
```

Example 5.5.2 illustrates how to use the `bePersistent` method. First, as outlined in the Figure 5.9, the transient instance is passed to a cache associated with the used connection. Then the cache takes ownership of the object and stores a reference to it into its internal structures. A `CacheLock` instance is returned. `CacheLock` class is used to lock local object copies in a cache so that they can be safely used by the user application. The local copy cannot be manipulated by the cache while it is locked.

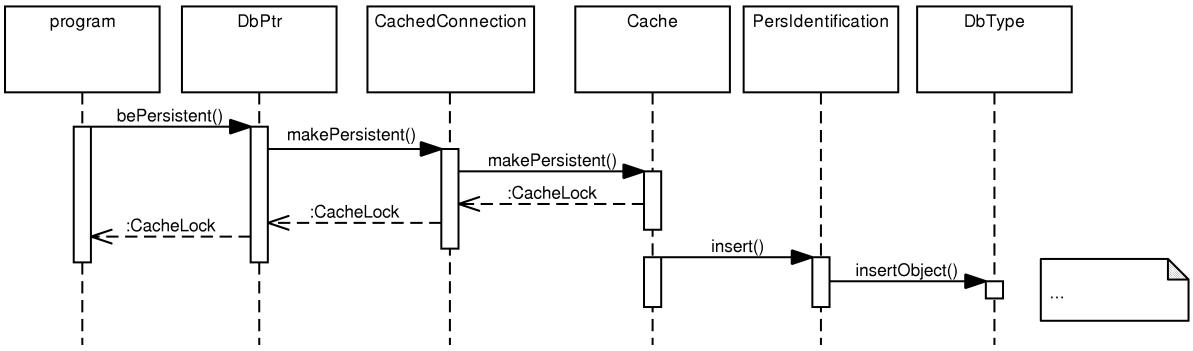

Figure 5.9: The `bePersistent` operation

When the lock is released, the cache is free to create a persistent copy of the object in the underlying database and eventually to remove the local copy from memory. Depending on the cache implementation and updating strategy used, the actual flow of events may differ slightly.

By dereferencing a database pointer instance, or by calling one of the `getCachePtr` or `getConstCachePtr` methods, an instance of a cache pointer can be retrieved. Cache pointer can be constructed for transient as well as for cache-managed objects. If a cache pointer for a cache-managed object is created, the cache is instructed to lock (or to load from a database and lock) the object it points to (the `CacheLock` class is used). The lock is released as soon as

the cache pointer instance is destroyed. Cache locks can also be dereferenced and, this time, a pointer to the actual database object instance is returned. The cache pointer is implemented as the `CachePtr` and `ConstCachePtr` classes. The only difference between them is that the latter one returns only a constant pointer to the object and locks it as read-only in the cache. Objects that have a read-only lock can be read by multiple threads at a time.

Both pointer classes (`DbPtr<T>` and `(Const)CachePtr<T>`) overload the `->` operator. As described in Section 3.2.3, if a call to the `operator->()` function is made and if the object returned also overloads the `->` operator, C++ automatically calls the `operator->()` on it again. Figure 5.10 illustrates what happens if the following source code excerpt is executed.

```
p->age = 59;
```

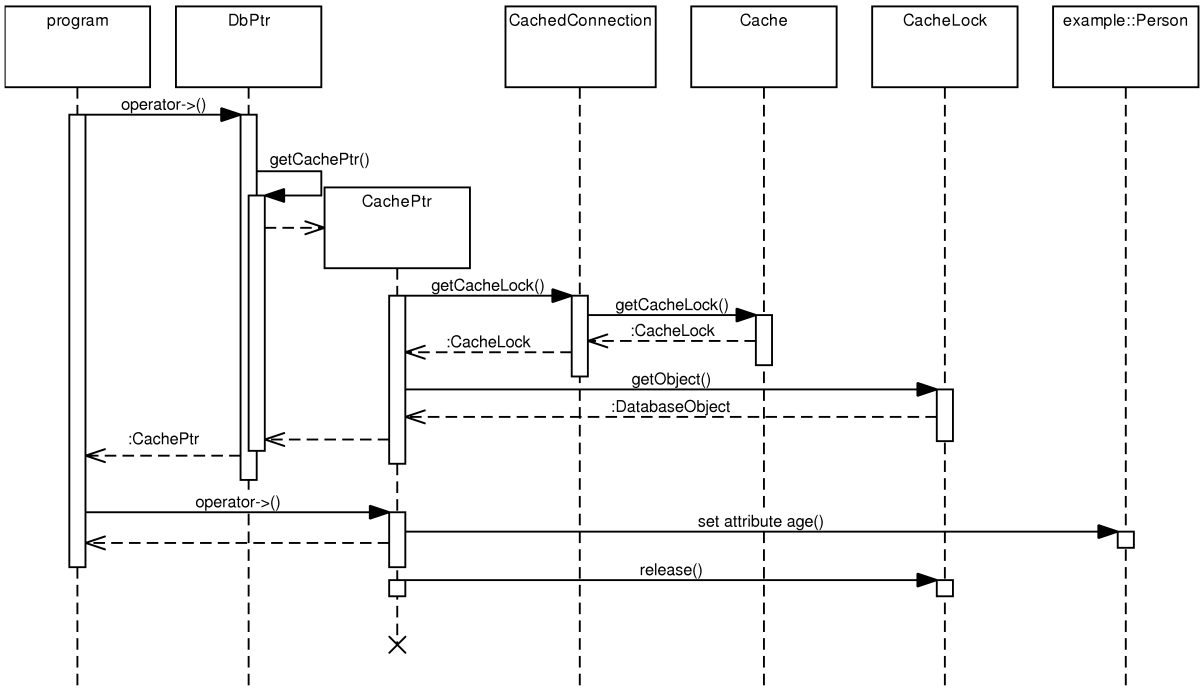

Figure 5.10: Database pointer and cache pointer interaction

The first `->` operator call results in the `CachePtr<T>` instance creation. Its constructor retrieves a `CacheLock` instance containing a reference to the locked local copy. When obtaining a `CacheLock`, the cache looks for the requested object in its containers and if not found, the database mapping part of the library is asked to load it from a database. Then, the object instance is locked as described above and `CachePtr<T>` retrieves a direct reference to the locked local copy from the `CacheLock`.

After that, C++ chains the `operator->` call and invokes it on the newly created `CachePtr<T>` instance. `CachePtr<T>` in turn returns reference to the database object itself and the `age` attribute is accessed. Immediately after the assignment operation, the life cycle of the `CachePtr<T>` instance ends and its destructor is called. Inside the destructor, the cache lock is released.

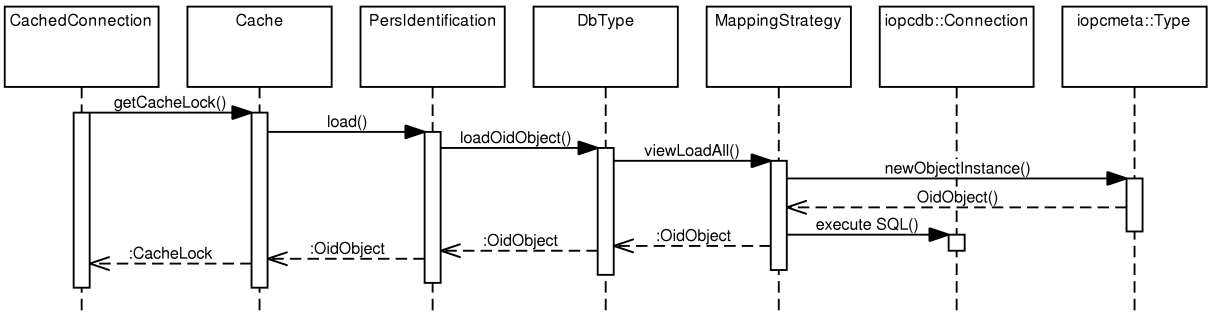

Figure 5.11: Interaction with the O/R mapping services

Figure 5.11 indicates how the flow of the previous scenario would continue from the `getCacheLock` call, if the requested object was not present in the cache. As mentioned earlier, the cache layer uses the `PersIdentification` class to call the database mapping routines.

5.5.3 The cache layer

The cache layer was ported from the POLiTe 2 library. Its interfaces and overall architecture remained almost unchanged. Cache layer depends on infrastructure described in the previous section. Because these infrastructure classes were preserved or enhanced, there was no reason to change the architecture of the cache layer. Again, the structure and behaviour of the cache layer were described before, so for detailed information about it please refer to [1]. Following paragraphs describe its hi-level architecture and design of its interfaces.

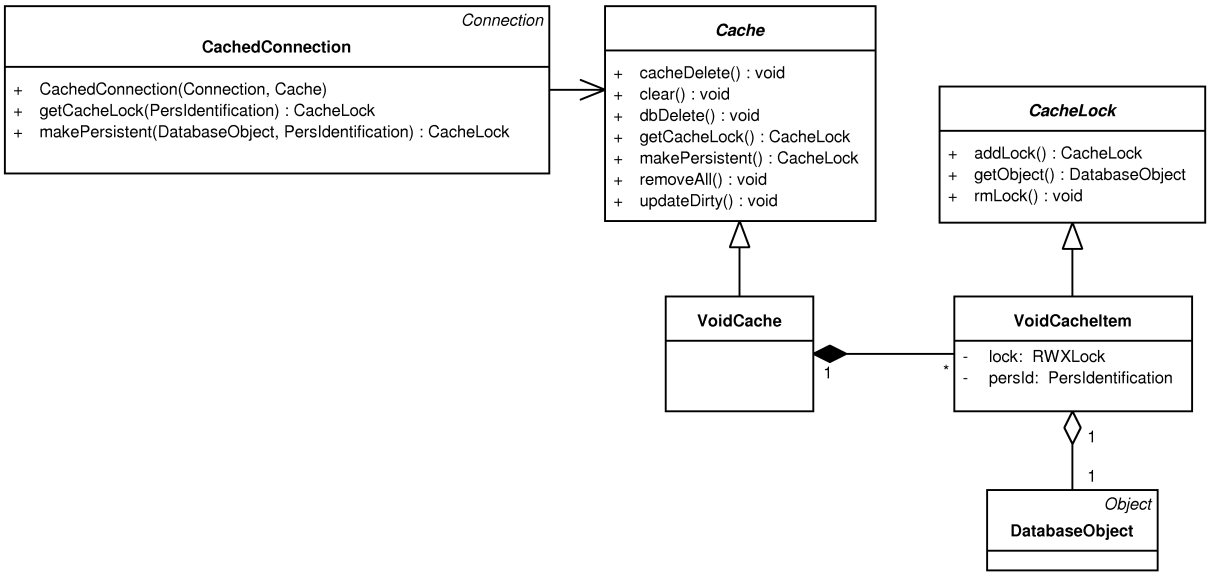

Figure 5.12: Basic classes of the cache layer. `VoidCache` architecture.

Each cache must implement the `Cache` interface. It defines following operations:

- `makePersistent` (insert) - cache obtains ownership of a new database object. The object is made persistent according to the updating strategy and caching algorithm used. Persistent copy creation can be enforced by invoking the `updateDirty` method.

- `getCacheLock` (load) - if needed, cache loads requested object from the database, locks it and returns a reference to it.
- `updateDirty` (update) - writes all changes including new objects to the database. `updateDirty` is called from `CachedConnection.commit()`.
- `dbDelete` (delete) - deletes the specified object from the cache as well as from the database.
- `clear` - writes all changes to the database and removes all objects from the cache.
- `removeAll` - clears the cache, and discards all changes. Nothing is written back to the database. Called from `CachedConnection.rollback()`.

Classes in Figure 5.12 represent the basic framework for building simple synchronous cache implementations. One such implementation is the `VoidCache`. The `VoidCache` actually doesn't cache anything - it merely loads and locks objects from the database as user requests. After the lock is released, changes made to the object are immediately propagated back to the database and the object removed from the cache. `VoidCache` stores the locked items in its containers as instances of the `VoidCacheItem` class.

IOPC 2 (and POLiTe 2) contains also an advanced interface designed for more complex cache implementations - the `ExtendedCache` interface. It adds methods that are needed by the cache manager `CacheKeeper` to run an asynchronous maintenance. Additionally, it allows to specify various strategies that modify behaviour of extended caches. Developers can define rules that specify which extended caches and which strategies should be used for particular objects. Architecture of the extended interface is displayed in Figure 5.13.

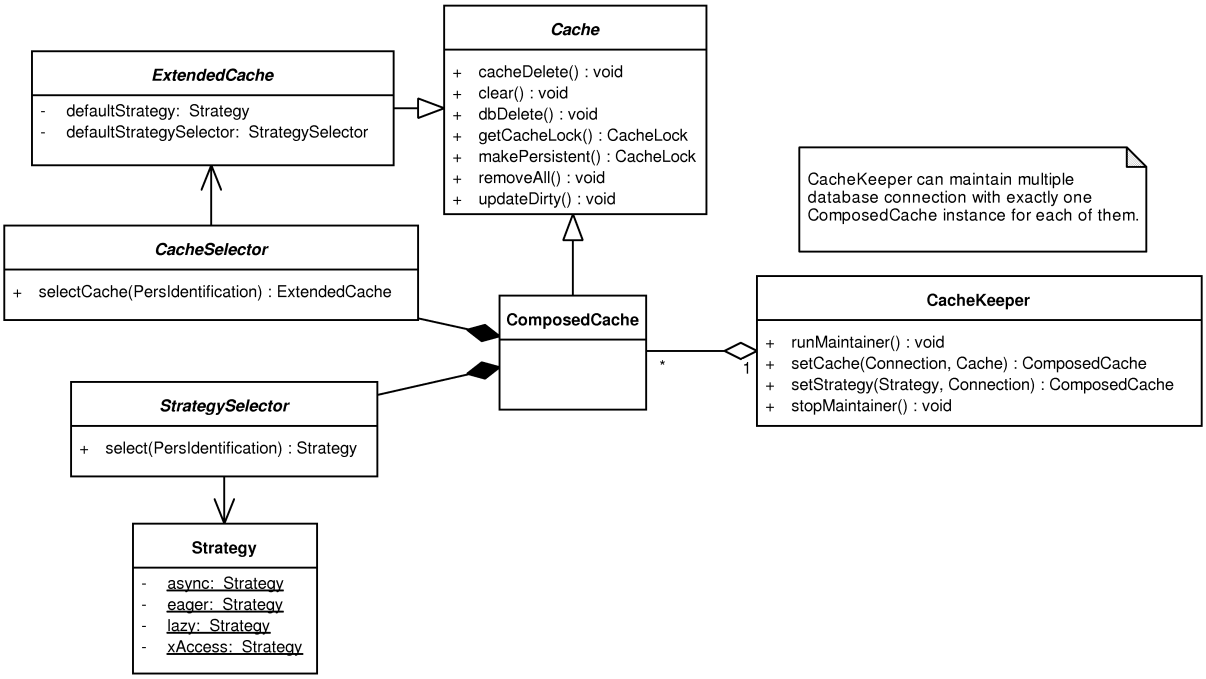

Figure 5.13: Extended interface of the cache layer.

- `Strategy` - instances of this class can be used to modify behaviour of an extended cache. The instances represent combinations of the updating, reading, locking and waiting strategies introduced in the original POLiTe library - see Section 3.1.4. The class contains four of such predefined strategy combinations that can be used right away (as static instances), see [1].

- `CacheSelector` and `StrategySelector` - selector interfaces are used to specify rules that decide which strategy or which cache will be used for specific objects. `SimpleStratSelector` and `SimpleCacheSelector` implement these interfaces and provide basic rules which associate specific strategies or caches respectively with persistent object types. This means that the decisions are based on the processed object types.

- `ComposedCache` derives from the basic cache interface `Cache` and acts as a proxy that combines different extended caches into one "basic" cache. It uses `CacheSelector` to determine which cache should be used for the object being processed. `StrategySelector` is then passed to the cache where it is used to customize its behaviour.

- `CacheKeeper` - a single `CacheKeeper` instance is assigned one or more database connections. For each of these connections an instance of `ComposedCache` is created. `CacheKeeper` provides a facade for these caches by exposing an interface that allows developers to specify caches and strategies for particular connection and database object type combinations. However, the main benefit of using `CacheKeeper` is that it runs a second thread that scans managed caches and removes instances considered as 'worst'. If dirty, these instances are written back to the database.

The library offers two caches that implement the extended interface - `MapLruCache` and `MapArcCache`. As their names suggest, they use LRU and ARC replacement policies, respectively. For detailed description please refer to [1]. There are also singlethreaded variants of these two caches (suffixed with "ST") which cannot be used in multithreaded environment and thus cannot use the asynchronous maintenance that the `CacheKeeper` provides.

5.5.4 Querying

Queries in IOPC 2 are realised as instances of the `SimpleQuery` and `FreeQuery` classes. Both classes derive from the same predecessor - the `Query` class as displayed in Figure 5.14. The base class declares the `getSql` method, whose implementation translates the query represented by the particular `Query` descendant instance to SQL. The `Driver` passed as a parameter is used in the translation process to accommodate the result to the destination SQL dialect.

The `SimpleQuery` class is used to create basic queries which select data only from a single view or table specified by the result type of the query result (see below). The result can be ordered and filtered by restricting attribute values of the type. `SimpleQuery` is used to retrieve instances of one particular type. `SimpleQuery` represents only the WHERE and ORDER BY parts of the resulting SQL SELECT statement.

For more complex queries there is the `FreeQuery` class. This class allows developers to specify everything after the FROM keyword. There are absolutely no restrictions on joins, subselects or other SQL language constructs.

To make the queries independent from the table, view and column naming, users can insert application-level class names, attribute names and class metadata (see Section 5.4.3) references into any part of the query. This is probably best explained by an example:

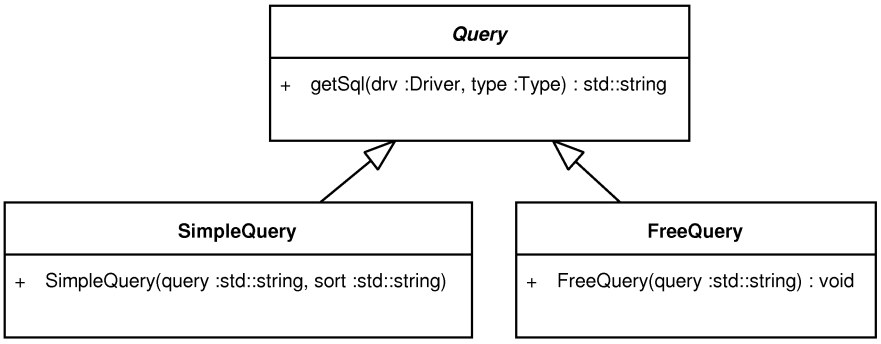

Figure 5.14: The Query classes

```
SimpleQuery query("$::Person::name$ = 'Ian'")
```

The translation process looks for the dollar signs and substitutes data type and attribute names that it finds between them. The `::Person::name` string will be substituted for the name of the corresponding database table/view and column. By default, the Oid object names reference the associated polymorphic views, while the non-Oid object names are replaced by the names of physical tables.

The data type and attribute names can be further expanded with class metadata codes. This way, we can force the data type name to be substituted with corresponding table name instead of its pregenerated view name:

```
FreeQuery query("$::Person[db.table]$ WHERE $::Person[db.table]:: ↩
   name[db.column]$ = 'Ian'")
```

The `db.table` and `db.column` class metadata are created by the `iopclib` library by default; however, they can be customised. For full list, see [?].

Results of the queries are encapsulated by the `Result` template class. Its only template parameter determines the type of the objects in the result set the query returns. Queries are executed by invoking the `open` method. After that, an iterator (`ResultIterator`) can be obtained and used in a standard C++ way. Example 5.5.3 shows how the example query from Section 2.5 ("Find all students older than 26.") can be implemented in IOPC 2.

Example 5.5.3 Example query in IOPC 2

```
SimpleQuery q("$::Student::age$ >= 26");
Result<Student> r(conn, q);
r.open();
for (Result<Student>::iterator it = r.begin(); it != r.end(); it ↩
   ++) {
  cout << "name: " << it->name << endl;
}
```

Chapter 6

Conclusion

IOPC 2 managed to unify the POLiTe 2 and IOPC libraries into one solid, flexible and extensible platform. Thanks to its well-designed interface and modular architecture, the library can be easily customized, enhanced and used. Its ability to integrate with IDEs and its reflective features provide the developers with almost transparent object-relational mapping functionality.

If we compare the table from Section 3.4 with what has been written about IOPC 2 in the last two chapters, we may see changes, mostly improvements in many of the listed areas:

	POLiTe	**POLiTe 2**	**IOPC**	**IOPC 2**
Transparent usage	- Macro descriptions	- Macro descriptions	+ Uses Open C++	+ Uses GCCXML
Supported mapping types	- Vertical	- Vertical	+ Vertical, horizontal, filtered, combinations	+ Vertical, horizontal, filtered, ADT, combinations
Associations between objects	+ Simple reference and relations - one-to-many, many-to-one, many-to-many, chained	+ Simple reference and relations - one-to-many, many-to-one, many-to-many, chained	+/- Simple reference and reference list	- Only simple reference
Caching	- Simple object cache	+ Advanced caching features.	-- Simple object cache, no locking	+ Advanced caching features (same as POLiTe 2).
Querying	C++-like syntax, combining queries	C++-like syntax, combining queries	C++-like syntax, combining queries	SQL syntax, class metadata embedding
Read-only database / existing schema support	+	+	-	+
Library architecture	- Monolithic	- Monolithic	- Monolithic	+ Modular, configurable
Supported databases	Oracle 7 (OCI 7)	Oracle 7 (OCI 7)	Oracle 8i (OCI 8)	Oracle 10g (OCI 10)

	POLiTe	POLiTe 2	IOPC	IOPC 2
Multithreading support	-	+	-	+

IOPC 2 was designed with emphasis on modularity. IOPC 2 doesn't provide only object-relational mapping services. It is a set of libraries which can be used in one of four configurations - an object persistence library, a database access library, a reflection library and an all-purpose library providing some useful tools. The database access library can be further customised with separate database drivers.

The reflection library represents a unified interface which allows developers to inspect the structure of the reflection-capable classes at run-time. It offers some of the features which are incorporated into reflective languages like Java or C#. To get the metamodel description IOPC 2 uses GCCXML, which represents a more reliable solution then OpenC++ used by IOPC.

Object-relational mapping capabilities of the original IOPC library has been enhanced by one additional mapping type - the ADT mapping. ADT mapping is a result of discussion on possibilities of object-relational databases presented in this thesis. It demonstrates how their features can simplify the development of an object-relational layer. IOPC 2 also retained the ability to work with existing or read-only database schemas.

Although the IOPC 2 library covers many of the needs developers may pose on a object persistence library, there is a lot of room for improvement and future enhancements. A major issue of this library is the lack of a persistable data type representing a list of references. To express various kinds of associations between persistent objects, POLiTe libraries used relation objects as described earlier in Section 3.1.3. POLiTe libraries also allowed to use a simple reference to model the one-to-many relation. IOPC replaced the concept of relation objects with the persistent references and reference lists. The reference lists were however almost unusable, so they were removed from the IOPC 2 implementation and remained to be implemented in the future. The future implementation should also use some of the ORDBMS features like references or arrays to store these lists in the database. Reintroduction of the relation objects may also be possible.

This leads us to the second, smaller, issue - IOPC 2 aside from ADTs doesn't use other ORDBMS features. There are many other ways how to enhance functionality of the library in this area. Other areas may also improved - the reflection can be given the ability to describe and invoke methods.

IOPC 2 also lacks *proof of concept* - a real-life or real-life-like application that would demonstrate that the library is usable for the purpose for which it is designed. The application could be web-based as well as a thick client. Such test would probably reveal other issues or misconceptions which may result in additional requirements on the library. For example the web application requirements would surely include a connection pool facility to speed up the connection creation process. Similarly, another non-Oracle database driver could be implemented - support for a lightweight in-process database like SQLITE[1] would be interesting.

[1] http://www.sqlite.org

Appendix A

User's guide

This chapter will guide you through the steps needed to create a simple IOPC 2-based application. We will start with library initialization and termination and then we will continue with metamodel definition and metamodel description browsing. Further we will describe how the database drivers and their extensions are used and finally, we will discuss usage of the persistence and caching layers to create a code that can persists or load database objects from the database.

A.1 Library initialization and termination

The IOPC 2 library can be used in several configurations (Section 4.1). Depending on the selected configuration, corresponding initialization and termination routines must be called. These routines are listed in the Table A.1. Configurations are ordered by complexity and by amount of features they provide. First row of the table contains the most basic configuration - the `iopccommon` library. Libraries/configurations in following rows depend on libraries from previous rows and also implicitly call their initialization/termination routines. So calling the initialization/termination routine only for the selected configuration is sufficient. There exist also header files with the "Headers" suffix in their names (like `iopcMetaHeaders.h` or `iopcDbHeaders.h`), which include most commonly used header files for the selected configuration.

library / configuration	**header file**	**initialization / termination routine**
`iopccommon` Library support services	`iopccommon/` `iopcCommon.h`	`IopcCommon::Init()` `IopcCommon::Shutdown()`
`iopcmeta` A reflection library	`iopcmeta/` `iopcMeta.h`	`IopcMeta::Init()` `IopcMeta::Shutdown()`
`iopcdb` A database access library	`iopcdb/iopcDb.h`	`IopcDb::Init()` `IopcDb::Shutdown()`
`iopclib` An object persistence library	`iopclib/iopc.h`	`Iopc::Init()` `Iopc::Shutdown()`

Table A.1: Library configurations and their initialization/termination routines

Basic library setup for a configuration that uses IOPC 2 as a database access library can then look like this:

```
// Include all commonly used headers for the
// iopcdb configuration
#include "iopcdb/iopcDbHeaders.h"

// IOPC 2 structures are defined within the iopc napespace
using namespace iopc;

int main(int argc, char *argv[])
{
  try {
    // Initializes the library and also sets the LogWriter masks
    // Throws an exception if not successful
    IopcDb::init(LogWriter::ERROR, LogWriter::ERROR |
      LogWriter::WARNING);

    // ...

    // Library cleanup. Releases allocated resources,
    // closes open connections.
    IopcDb::shutdown();
  } catch (IopcException& e) {
    // Error handling
  }
}
```

First parameter of the `IopcDb::init` specifies mask for messages written to standard output, second parameter specifies mask for the log file which is created by the library at its initialisation. Its name is `iopc_log.txt` by default, but it can be overriden in the third (not shown) parameter of the `IopcDb::init` method.

The logger masks filter the log messages by their category. Supported categories are:

- ERROR - messages generated usually as a result of an exception that prevented required operation to be finished. Can be generated by the user application.
- WARNING, INFO - warning or informational messages. Can be generated by the user application.
- INFO_SQL - SQL trace messages representing SQL statements that has been sent to the underlying database. Messages in this category are exclusively generated by the `iopcdb` library.
- DEBUG - messages useful for debugging purposes. May be also disabled by commenting out the IOPC_DEBUG macro in the `iopccommon/trace.h` file. Release versions of IOPC 2 should have this macro undefined.
- TRACE - more detailed messages which trace the library execution. Turning this filter on may result in large amount of logged messages. May be also disabled by commenting out the IOPC_TRACE macro in the `iopccommon/trace.h` file. Release versions of IOPC 2 should have this macro undefined.

If an error occurs during any of library activities an exception is thrown. All exceptions thrown by the IOPC 2 library are descendants of the `IopcException` class. For a detailed list of all excpetion types please refer to the library documentation.

A.2 Inspecting objects with reflection

Reflection and object persistence features will be presented on the following classes. We will start with a simple class hierarchy with which we are already familiar from the Section 2.2. To present the provided reflection mechanism and later the database mapping, we need to define some test classes. Additional features will be added later.

```
// We use the reflection library configuration
#include "iopcmeta/iopcMetaHeaders.h"
#include <string>

using namespace std;

class Person : public iopc::Object {
public:
  EString name;
  EShort age;
};
class Employee : public Person {
private:
  EInt salary
public:
  Employee() {};
  Employee(string name, short int age, int salary) :
    name(name), age(age), salary(salary) {};
  void setSalary(int salary) { this->salary = salary; };
  int getSalary() { return salary; };

  friend clas iopc::TypeDesc<Employee>;
};
class Student : public Person {
public:
  EString studcardid;
};
class PhdStudent : public Student {
public:
  EShort scholarship;
};
```

Classes discoverable by the IOPC 2 reflection must conform to the following rules:

- The classes must be defined in header files. These headers will be included by the generated IH files containing type descriptions (see Section 5.4.4).
- The classes must derive drectly or indirectly from the `Object` defined in `iopcmeta/object.h`.
- The classes must contain a non-parameterised constructor (implicit or explicit). A constructor with all default parameters is also acceptable.

- If the classes contain any non-public members, they must declare the associated type description as friend - see the `Employee` class definition. For more information, refer to Section 5.4.5.

Attribute definitions use enhanced data types (see Section 5.4.2). This will come in handy later in sections discussing object persistence and querying.

To see how to use the reflection interface, we will provide a set of examples which demonstrate its features. The code goes between the `init` and `shutdown` which we presented in the first example.

Following example lists all data types that are recognised by the reflection mechanism.

```
const map<string, const Type*>& types =
  TypeManager::getTypesMap();
cout << "Registered types:" << endl;
for(
  map<string, const Type*>::const_iterator it = types.begin();
  it != types.end(); it++) {
  cout << (*it).first << endl;
}
```

When compiled using the supplied makefile/compile script or using the managed-build Eclipse project and when run, the program will print a list of types registered by the `TypeManager` singleton (see Section 5.4).

```
::Employee              iopc::EBool
::Person                iopc::EChar
::PhdStudent            iopc::EDouble
::Student               iopc::EFloat
::bool                  iopc::EInt
::char                  iopc::ELDouble
::double                iopc::ELong
::float                 iopc::ESChar
::int                   iopc::EShort
::long                  iopc::EString
::long double           iopc::EUChar
::short                 iopc::EUInt
::signed char           iopc::EULong
::unsigned char         iopc::EUShort
::unsigned int          iopc::EWCharT
::unsigned long         iopc::EWString
::unsigned short        iopc::Object
::wchar_t
std::string
std::wstring
```

The types can be divided into three categories:

- Built-in C++ numeric types, STL string types.
- Enhanced data types which correspond to the basic types and string types.
- All descendants of the `Object` class which is also included and can be inspected using the reflection. Such types are classified as *complex* datatypes in the IOPC 2 library.

All the listed types can be used as attribute types. Attributes of other types are ignored[1]. Complex type attributes can't be however persisted by the object persistence layer (`iopcli-b`) at the moment. In this case, complex type attributes must be marked as transient using the attribute metadata `db.transient`. See [?].

The programmer can use the reflection for more complex tasks, as it is shown in following examples.

First, we obtain an object which represents a type description. See Section 5.4.1.

```
const Type& t = TypeManager::getType("::Person");
```

The `TypeManager::getType()` call searches in the catalogue of registered types for the type of given name and returns its description. The type description is obtained as a reference to a `Type` object representing the requested type. This class provides many methods that can be used to introspect the structure of the associated type. The type description can be also obtained in a static way:

```
const Type& t = TypeDesc<Person>::getType()
```

Having the `Type` reference we can start examinig the `Person` class by obtaining a list of its parents:

```
const Type::ParentsList& parents = t.getParents();
for(Type::ParentsIterator it = parents.begin();
  it != parents.end(); it++) {
  cout << (*it)->getQualifiedName() << endl;
}
```

If we are not using multiple inheritance, the following method can be used:

```
cout << t.getFirstParent().getQualifiedName() << endl;
```

The example prints:

```
iopc::Object
```

Similarly, we list the child types of the `Person` class.

```
const Type::ChildrenList& children = t.getChildren();
for(Type::ChildrenIterator it = children.begin();
  it != children.end(); it++) {
  cout << (*it)->getQualifiedName() << endl;
}
```

We get the following output:

```
::Employee
::Student
```

If we want to obtain even indirect parents or children, we have to use the `Type::getAllParents()` or `Type::getAllChildren()` methods instead.

Attributes can be listed using the following code:

```
const Type::AttributesMap& attributes = t.getAttributes();
for(Type::AttributesIterator it = attributes.begin();
  it != attributes.end(); it++) {
```

[1] This should change in the future

```
  const Attribute& attr = it->second;
  switch (attr.getVisibility()) {
    case Attribute::VISIBILITY_PUBLIC: cout << "public "; break;
    case Attribute::VISIBILITY_PRIVATE: cout << "private "; break;
    case Attribute::VISIBILITY_PROTECTED: cout << "protected ";  ↩
       break;
  }
  cout << it->second.getType().getQualifiedName()
    << " " << it->first << endl;
}
```

The example iterates over a map whose keys are attribute names and values are `Attribute` class instances. Using `Attribute` methods we can obtain attribute names, their data types (which are again described by the familiar `Type` interface) and determine if the attributes were declared as private, protected or public. In a similar way, inherited attributes can be listed using the `Type::getInheritedAttributes()` method. The previious code yields the following output:

```
public iopc::EString name
public iopc::EShort age
```

Using reflection we can also create objects or manipulate with values of their attributes (regardless of their visibility). In the following example we create a new instance of the `Employee` class and initialise values of all of its attributes.

```
Employee* e =
  (Employee*) TypeManager::getType("::Employee").newInstance();
*((EString*)e->getAttributeValue("name").getValue()) =
  "Ola Nordmann";
*((EShort*)e->getAttributeValue("age").getValue()) = 45;
*((EInt*)e->getAttributeValue("salary").getValue()) = 60000;
cout << e->name << " " << e->age << " " << e->getSalary() << endl;
```

The `Object::getAttributeValue()` method returns an instance of the `AttributeValue` which allows us to get or set values of the attribute it represents. Using its `getValue()` method we obtain a void pointer to the requested field in the `Employee` instance. By casting it to the actual attribute data type we can manipulate with its value. Output of the example is:

```
Ola Nordmann 45 60000
```

Attribute values can be iterated similarly as attributes. The list of attribute values can be obtained using the `Object::getAttributeValues()` method:

```
const Object::AttributeValuesMap& attrVals =
  e->getAttributeValues();
for(Object::AttributeValuesIterator it = attrVals.begin();
  it != attrVals.end(); it++) {

  const AttributeValue& val = it->second;
  cout << it->first << ": ";
  if (val.getAttribute().getType().getDataTypeClass() == Type:: ↩
     IOPC_TYPECLASS_ENHANCED) {
   cout << ((EnhancedTypeBase*)val.getValue())->toString()
    << endl;
```

```
  }
}
```

In the example we take advantage of the enhanced datatypes and of the `Object::toString()` method they implement. For regular objects it prints just an address on which they reside in the memory. It is however overridden by the enhanced datatype implementations so that the returned string contains a value they represent. Output of the example is:

```
age: 45
name: Ola Nordmann
salary: 60000
```

Types recognised by the reflection are categorised into several groups called *type classes*. They help to easily determine their "nature":

- IOPC_TYPECLASS_SIMPLE - built-in C++ numeric types
- IOPC_TYPECLASS_STRING - STL strings.
- IOPC_TYPECLASS_COMPLEX - class types, descendants of `Object`. Excluding the enhanced datatypes.
- IOPC_TYPECLASS_ENHANCED - enhanced datatypes, descendants of `EnhancedTypeBase`.

Last thing we may need from the reflection interface is the ability to compare the types. For example, we may want to ask if some type is descendant of another type or if it represents the same type. The following example illustrates how such comparisons can be done in IOPC 2.

```
const Type& t1 = TypeDesc<Person>::getType();
const Type& t2 = TypeDesc<Student>::getType();
if (t1 == t2) { cout << "t1 is t2" << endl; }
if (t1 != t2) { cout << "t1 is not t2" << endl; }
if (t1.isTypeOf(t2)) { cout << "t1 is t2 or t1 is descendant of t2 ↩
   " << endl; }
if (t2.isTypeOf(t1)) { cout << "t1 is t2 or t2 is descendant of t1 ↩
   " << endl; }
if (t1.is<Student>()) { cout << "t1 is of type Student" << endl; }
if (t2.is<Student>()) { cout << "t2 is of type Student" << endl; }
if (t1.isTypeOf<Person>()) { cout << "t1 is of type Person or its  ↩
   descendant" << endl; }
if (t2.isTypeOf<Person>()) { cout << "t2 is of type Person or its  ↩
   descendant" << endl; }
```

Output of the example is:

```
t1 is not t2
t1 is t2 or t2 is descendant of t1
t2 is of type Student
t1 is of type Person or its descendant
t2 is of type Person or its descendant
```

A.3 Class metadata

The concept of class metadata was explained in Section 5.4.3. Basically, class metadata can be specified on three levels:

- At a global level using `MetadataHolder::getDefaultsMeta()` in `iopccommon`.
- At a type level.
- At an attribute level.

Metadata specified at higher level are inherited by objects at lower levels. So if we specify metadata for a type, all attributes of that type inherit such information. If we specify the same metadata on an attribute, the type level metadata are overridden by the attribute level metadata. Global level metadata are inherited by both, the type and attribute objects.

The `Type` and `Attribute` classes inherit from the `MetadataHolder`, `MetadataHolder::getDefaultsMeta()` returns a `MetadataHolder` instance directly. `MetadataHolder` provides its descendants with a dictionary-like interface as illustrated by the following example:

```
// Global level metadata (all objects will use ADT mapping)
MetadataHolder& globalMeta = MetadataHolder::getDefaultsMeta();
globalMeta["db.mapping.type"].setStringValue("object");

// Type level metadata
// The ADT type associated with the Person class
// will be named Person_type. (The default is tPerson).
const Type& t = TypeDesc<Person>::getType();
t["db.type"].setStringValue("Person_type");

// Attribute level metadata
// The Person::name attribute will be a 20 characters long string
t.getAttribute("name")["db.type.length"].setIntValue(20);
```

Class metadata can be set in three different ways at the program startup. First way is to set metadata in the static method `iopcInit()` which can be defined for each reflection-capable type. It is used to set type level and attribute level metadata for the class in which it is defined. `iopcInit()` methods are invoked during the `iopcMeta` library initialisation (`IopcMeta::Init()`). To demonstrate how `iopcInit` is used, we modify the `Person` class definition:

```
class Person : public iopc::OidObject {
public:
  EString name;
  EShort age;

  static void iopcInit(iopc::Type& t) {
    t["db.type"].setStringValue("Person_type");
    t.getAttribute("name")["db.type.length"].setIntValue(20);
  }
};
```

Second way is to implement the `MetadataInitializer` interface declared in `iopcmeta`. It defines only one method - `initializeMetadata()` - into which we can insert the

metadata initialization code. In this case, the global level values can also be set. The instance of the `MetadataInitializer` implementation is then passed as first paremeter to any of the `iopcmeta`, `iopcdb` or `iopclib` initialisation methods:

```
class OurMetadataInitializer : public MetadataInitializer {
public:
  virtual void initializeMetadata();
};
...
void OurMetadataInitializer::initializeMetadata()
{
  // metadata initialisation code
  // same as in the first example of this section
}
...
int main(int argc, char *argv[])
{
  Iopc::init(new OurMetadataInitializer(), LogWriter::ERROR ... );
  ...
}
```

The `OurMetadataLoader` instance is deallocated during the library initialization after its `initializeMetadata()` method is called. During the initialization phase, `iopcInit()` methods are invoked before `MetadataInitializer::initializeMetadata()`, so the metadata changes done in `MetadataInitializer::initializeMetadata()` may overwrite metadata created in the `iopcInit()` methods.

`iopcmeta` contains a `MetadataInitializer` implementation named `TextFileMetadataLoader` which represents the last way how we can set the class metadata. Its constructor takes a single parameter, a filename (may include a path as well) of a configuration file, which contains a description of metadata assignments. The class is used in same way as the `OurMetadataInitializer` in previous example. When its `initializeMetadata()` method is invoked, it loads the configuration file and sets all global level, type level and attribute level metadata according to it.

The format of the configuration file is described by the following example:

```
# global level
defaults[db.mapping.type];string;object
# type level
::Person[db.type];string;Person_type
# attribute level
::Person::name[db.type.length];int;20
```

`TextFileMetadataLoader` skips all empty lines or lines starting with the '#' character. The file is case-sensitive. Lines describing the metadata contain three semicolon delimited values. First value specifies the location and code of the metadata to be set. It uses the same format as metadata in queries (see Section 5.5.4). Second columns denotes the data type of the metadata value. Only `string`, `int` and `bool` values are allowed. Last column contains the metadata value. Boolean values can be either `true` or `false`.

A.4 Basic database access

IOPC 2 can be used as a database access library. It is able to send SQL commands to supported database engines and to execute SQL queries and iterate through their results. Transactions are also supported - see further. To use the provided database features we need to include and link either the `iopcdb` or the `iopclib` library / configuration.

To send any SQL commands to the database, we need to obtain and open a connection:

```
Database* db =
  DriverManager::getDriver("IopcOracle10g").getDatabase("XE");
Connection* conn =
  db->getConnection("username=iopc;password=iopc");
conn->open();
```

First the `Driver` instance is obtained using the `DriverManager::getDriver()` method. The `Driver` doesn't provide much functionality. Usually it is used to obtain the `Database` object using its `Driver::getDatabase()` method. Its other responsibilites will be discussed later in this section.

`Driver::getDatbase()` takes one parameter denoting name of the database instance the returned `Database` object will represent. For other database systems, the parameter can refer for example to a filename of a database that will be loaded. The `Database`'s only interresting method at the moment is `Database::getConnection`.

By calling the `Database::getConnection()` method a physical link to the DBMS is established[2] and a `Connection` object representing the link is returned. The string passed to the `Database::getConnection()` method is called *connection string*. The the connection string syntax may vary between different drivers. For the `Oracle10g` driver the string is a semicolon delimited list of key-value pairs as shown in the example. Recognised keys are:

- username - database account username
- password - its password
- autocommit - determines whether the current transaction is commited after every statement executed. Optional, the default value is "false".

When the connection is no longer needed, it should be closed and returned back to the database object by calling `db->returnConnection(conn)`. The same goes for the database and driver:

```
...
conn->close();
db->returnConnection(con);
DriverManager::getDriver("IopcOracle10g").returnDatabase(db);
// alternatively:
DriverManager::returnDatabase(db);
```

`Oracle10g` driver supports *transactions*[3]. First SQL statement executed after the connection is created begins first transaction. Transactions can be ended by calling `Connection::commit()` or `Connection::rollback()`. Next transaction starts again when next SQL statement is executed. A Transactions get commited implicitly when a DDL statement is issued.

[2] Other drivers may behave differently.

[3] For more on what database transactions are refer to http://en.wikipedia.org/wiki/Database_transaction

Closing an open connection also commits any running transaction. If autocommit is enabled, transactions are committed immediately after SQL statements are executed.

`Connection::savePoint()` and `Connection::rollbackToSavePoint()` operations are also provided by the `Connection` interface. They interact with the cache analogously.

Simple commands that do not return any value and that are not parametrised can be executed using the `sqlNonQuery` method. Let's create a new table and insert some testing values into it.

```
conn->sqlNonQuery("CREATE TABLE blacklist(id NUMBER PRIMARY KEY,  ↩
   name VARCHAR2(20))");
conn->sqlNonQuery(
  "INSERT INTO blacklist VALUES(1, 'Richard Doe')");
conn->sqlNonQuery(
  "INSERT INTO blacklist VALUES(2, 'Ola Nordmann')");
// we have to call commit() because autocommit is disabled
// for conn
conn->commit();
```

Now we will load and print the inserted data:

```
Cursor* cur = conn->sql("select id, name from blacklist");
std::string name;
int id;
cur->addOutParam(1, &id, TypeDesc<int>::getType());
cur->addOutParam(2, &name, TypeDesc<std::string>::getType());
cur->execute();
while (cur->fetchNext()) {
  cout << id << " " << name << endl;
}
cur->close();
conn->returnCursor(cur);
```

By calling the `Connection::sql()` method we obtain a `Cursor` which represents results of the query passed to the method. The query is not yet executed; nothing is sent to the database system until the `Cursor::execute()` is invoked. Before we execute the query, we may specify input and output parameters (if any). This is done by calling one of the `Cursor::addXXParam()`, `Cursor::addEnhancedXXParam()` where XX is "In" or "Out", or `Cursor::addNullInParam()`. Whether to call the first or the second (enhanced) method depends on if we are going to use enhanced data type or basic C++ data type (plus the STL strings) parameters. The example above uses the basic variant.

First parameter specifies the parameter position in the query[4] and second parameter the variable to be stored/loaded. Last parameter of the `Cursor::addXXParam` method tells the database layer of what type the value passed as pointer in the second parameter is. In some situations, the database layer and drivers must know the memory size (sizeof) occupied by the input or output parameters. This information is also obtained from the type descriptions. However, because memory occupied by string parameters (STL or enhanced) is of variable size, a number describing memory size to be allocated for buffers needed by the database layer must be specified. This number is defined as maximum string length and defaults to 2000 characters. It can be overridden by supplying the last argument of the `Cursor::addXXParam` methods in this way:

[4]Parameters specified by name are not currently supported.

```
MetadataHolder meta;
meta["db.type.length"].setIntValue(100);
cur->addOutParam(1, &id, TypeDesc<int>::getType());
cur->addOutParam(2, &name,
  TypeDesc<std::string>::getType(), meta);
```

The `db.type.length` metadata can be also specified on the type or attribute level as described in Section A.3. Type level represents the default value for all parameters of such type (by default 2000 as mentioned above), attribute level is used by the object persistence layer - see later in this guide.

Same example as above, this time using the enhanced data types:

```
Cursor* cur = conn->sql("select id, name from blacklist");
EString name;
EInt id;
cur->addEnhancedOutParam(1, id);
cur->addEnhancedOutParam(2, name);
cur->execute();
while (cur->fetchNext()) {
  cout << id << " " << name << endl;
}
cur->close();
conn->returnCursor(cur);
```

As you see, specifying enhanced datatype parameters is even simpler, the additional type description is not needed.

If the cursor is not needed, it should be closed to free blocked database resources. If it is not needed any longer, it should be released by returing it back to its connection which also releases allocated application resources. After the cursor is closed, it can be re-opened and executed again (it even be re-executed without closing and opening). The parameters are not discarded. If a cursor is returned back to its connection in an open state, it is automatically closed.

Input pramaters are used in a similar way:

```
Cursor* cur = conn->sql(
  "SELECT name from blacklist where id = :id");
EString name;
EInt id = 2;
cur->addEnhancedOutParam(1, name); // first input parameter
cur->addEnhancedInParam(1, id); // first output parameter
cur->execute();
while (cur->fetchNext()) {
  cout << id << " " << name << endl;
}
cur->close();
conn->returnCursor(cur);
```

This example loads and print only the second row inserted ("Ola Nordmann").

All objects from the `iopcdb` library are thread safe except for the `Cursor` class, which should be used always only from one thread.

A.5 Driver features

Driver features were introduced in Section 5.3.2. A driver feature is actually an interface (abstract class) which each driver may or may not implement. Two basic features are defined by `iopcdb` - the `TypeMappingFeature` and `SqlStatementsFeature`. See their documentation for detailed description.

The database driver feature interface provides basically only two things - it allows us to ask whether the driver supports the feature specified and allows us to obtain a reference to its implementation. This is best illustrated by an example:

```
Driver& d = conn->getDriver();
if (d.supportsFeature<SqlStatementsFeature>()) {
  TypeMappingFeature& f = d.getFeature<TypeMappingFeature>();
  string sqlType = f.getTypeSql(TypeDesc<string>::getType());
  cout << sqlType << endl;
}
```

First, we obtain a driver reference from an existing connection. We want to use the `TypeMappingFeature` to obtain a SQL type definition for the `std::string` type. We ask, if the driver even supports such feature and then we obtain a reference to its implementation. Then we may call methods of the feature according to its interface and documentation. This example returns a `VARCHAR2(2000)` string representing a SQL type which corresponds to std::string. The number 2000 is taken from the `db.type.length` default metadata specified on the std::string type.

A.6 Driver implementation

Database driver is an instance of a `Driver` subclass. Each driver is identified by a unique name. Implementation of a new database driver begins by creating a subclass of the `Driver` class. The subclass requires the following to be implemented:

- Driver initialisation and termination code if needed (`Driver::doInit()` and `Driver::doShutdown()` methods). Usually the initialisation routine registers features the driver supports:

  ```
  void Oracle10gDriver::doInit() {
    // typeMappingFeature is defined as an attribute in the  ↩
       Oracle10gDriver class:
    // Oracle10gTypeMappingFeature typeMappingFeature;

    registerFeautre<TypeMappingFeature>(typeMappingFeature);
  }
  ```

- `getDatabaseImpl()` and `returnDatabaseImpl()` methods which should create, repspectively free the driver-specific instance of the `DatabaseImpl` subclass.

- A constructor implemented in the following way:

  ```
  Oracle10gDriver::Oracle10gDriver(const std::string&  ↩
     driverName)
    : Driver(driverName) {}
  ```

- Driver registration code. A single static driver instance which is initialized as soon as the driver is loaded. Run-time calls the driver constructor that in turn registers the driver at the `DriverManager`.

```
static Oracle10gDriver oracle10gDriverInstance("IopcOracle10g ↩
   ");
```

As you can see, the `Driver` class is only an entry point to the driver. Most of its code can be found in its `DatabaseImpl`, `ConnectionImpl` and `CursorImpl` implementations.

Driver extensions are usually sets of additional driver features which are appended to the features the drivers provide. Driver extension is a `DriverExtension` subclass instance which is created and registered in a similar way to driver:

- The `DriverExtension::doInit()` initialisation method takes one parameter which is a reference to the associated driver. In the `IopcOracle10gOORExtension`, additional driver features are registered:

```
void Oracle10gExtension::doInit(Driver& driver) {
  driver.registerFeature<ORStatementsFeature>( ↩
     orStatementsFeature);
  ...
}
```

- Driver extension constructor has two parameters. One parameter is a name of a driver the extension extends, second is a name of the extension. Again, its implementation should call the parent constructor as in the following example:

```
Oracle10gORExtension::Oracle10gORExtension(const std::string& ↩
    driverName, const std::string& name) :
  DriverExtension(driverName, name) {}
```

- Driver extension registration code:

```
static Oracle10gORExtension Oracle10gORExtensionInstance(" ↩
   IopcOracle10g", "IopcOracle10gORExtension");
```

A.7 Persistent object manipulation

The most complex configuration of the IOPC 2 library we can use is `iopclib`. It contains all features described to this point, plus it offers object persistence using four kinds of database mapping - vertical, horizontal, filtered and ADT. Objects that can be persisted must derive from the `DatabaseObject` or `OidObject` classes. We will start with the OID objects as they are easier to handle and offer more. Let's take the classes defined in Section A.2 and prepare them for the persistence layer.

```
// Use the iopclib headers
#include "iopclib/iopcHeaders.h"

using namespace iopc;
class Person : public iopc::OidObject {
public:
```

```
  EShort age;
  EString name;
};
class Employee : public Person {
public:
  EInt salary;
};
class Student : public Person {
public:
  EString studcardid;
  DbPtr<Employee> supervisor;

  static void iopcInit(iopc::Type& t) {
    t["db.mapping.type"].setStringValue("horizontal");
  }
};
class PhdStudent : public Student {
public:
  EShort scholarship;

  static void iopcInit(iopc::Type& t) {
    t["db.mapping.type"].setStringValue("filtered");
    t["db.mapping.insertto"].setStringValue("::Student");
  }
};
```

We changed the predecessor of the `Person` class to `OidObject` as we intend to use OID objects. We also specified some metadata that modify the default mapping type from vertical to horizontal for the `Student` class and filtered for the `PhdStudent` class. If filtered mapping is used, the `db.mapping.insertto` metadata must also be specified. The `::Student` value tells the library that attributes of the `PhdStudent` class will be stored into the table associated with the `Student` class.

Before we can start using these classes, we must prepare required database schema. `ScriptsGenerator` can be used for this task as described in Section 5.5.1. Once more, the code snippet that creates the database schema for persistent classes in the application:

```
vector<string> script;
script = ScriptsGenerator::getDbCreateScript(conn->getDriver());
for(vector<string>::const_iterator it = script.begin();
  it != script.end(); it++) {
  conn->sqlNonQuery(*it);
}
```

When manipulating peristent objects, we need to use `CachedConnection` decorator for the standard connection usually obtained. There is also the `CachedDatabase` class which decorates the "basic" database and creates already decorated `CachedConnections`. The reason why we are using these decorators is that the caching layer is tightly integrated with the object persistence layer and object persistence can't be used without it. The `CachedConnection` provides us with additional methods for cache manipulation. It can be created in the following way:

```
CachedDatabase* db = new CachedDatabase(
  DriverManager::getDriver("IopcOracle10g").getDatabase("XE"),
```

```
  new VoidCache());
CachedConnection* conn =
  (CachedConnection*)db->getConnection("username=iopc;password= ↩
     iopc");
conn->open();
```

`CachedConnection` needs to have an associated cache. In this scenario we use the basic cache implementation - the `VoidCache` - see Section 5.5.3.

Now, we have all the prerequisites we need to be able to manipulate persistent objects. First, we create some persistent objects and store them in the database.

```
DbPtr<Person> person;
person->name = "Mary Major";
person->age = 60;
person.bePersistent(conn);

DbPtr<Employee> employee;
employee->name = "Ola Nordmann";
employee->age = 45;
employee->salary = 60000;
employee.bePersistent(conn);

DbPtr<Student> student;
student->name = "Richard Doe";
student->age = 22;
student->studcardid = "WCD-3223";
student->supervisor = employee;
student.bePersistent(conn);;

DbPtr<PhdStudent> phd;
phd->name = "Joe Bloggs";
phd->age = 27;
phd->studcardid = "PHD-1234";
phd->scholarship = 12000;
phd->supervisor = employee;
phd.bePersistent(conn);
conn->commit();
```

Transient instance of a persistent class is automatically created by declaring a variable of a database pointer type as shown in the example. Its attributes are accessible using the `->`. The `DbPtr::bePersistent()` method then transfers the ownership of the transient instance to the cache associated with the connection `conn`. In this case, the `VoidCache` immediately inserts the object into correspoinding database table.

If we used another cache, the objects might be inserted into the database later. However, the `CachedConnection::commit()` method call forces the associated cache to store all changes (including new objects) into the database before the actual commit is issued to the database. This update can be invoked manually by calling `CachedConnection::updateDirty()`. Actually, commit calls `CachedConnection::updateDirty()` to perform the update. Analogously, `CachedConnection::removeAll()` is called before `CachedConnection::rollback()` to discard all changes or new objects by emptying the associated cache. (The objects will be loaded from database into the cache again as soon as they are accessed).

Note that when we assign a database pointer to the `supervisor` reference attribute, the database pointer must already point to an instance which is managed by a cache. That means that `DbPtr::bePersistent()` must have already been called on the database pointer before.

Every time we access attributes of a persistent object managed by a cache using the `->` operator, the object is looked up in the cache and returned as a cache pointer - `CachePtr`. Then the attribute is accessed and the cache pointer instance is destroyed. If we use the `VoidCache`, the object is even loaded from the database every time when the `CachePtr` is created and stored back when destroyed. If the object is transient, just the `CachePtr` is created and destroyed repeatedly, cache is not used.

If we want to prevent such behaviour to speed up repeated access to the persistent object attributes, we need to store the `CachePtr` instance into a variable. This can be done in two ways:

```
// Using the getCachePtr() method
CachePtr<PhdStudent> phdPtr = phd.getCachePtr();

// or by dereferencing the DbPtr using the * operator:
CachePtr<PhdStudent> phdPtr = *phd;
```

We may now modify the attributes, the persistent object is locked in the cache. If we want the cache to take back the control of the persistent object, we must release the cache pointer. Cache pointer can be released either explicitly by calling the `CachePtr::release()` or implicitly by destroying its instance:

```
 // explicitly
CachePtr<PhdStudent> phdPtr = *phd;
phdPtr->scholarship = 13000;
phdPtr->supervisor = employee;
phdPtr.release();

// implicitly:
{
  CachePtr<PhdStudent> phdPtr = *phd;
  phdPtr->scholarship = 13000;
  phdPtr->supervisor = employee;
}
```

Persistent objects can be locked for read-only access using the `DbPtr::getConstCachePtr()` call. We can't modify the object attributes, but the advantage is that the cache lock used in this case is not exclusive and the object can be locked in this way by more threads at a time.

Cache pointers may be copied, the lock is released as soon as last instance of the cache pointers is released or destroyed. Database pointers may also be copied, the behaviour of the copies depends on whether we are copying a pointer to a transient object or to an object managed by a cache. Copies of the transient objects are reference counted. We are not allowed to call the `DbPtr::bePersistent()` method if more than one reference pointing exists, an exception would be thrown.

By creating a copy of a database pointer pointing to a cache managed object, only its identity (`PersistentIdentification`) and connection infromation are copied. If we delete the object using one of the pointers from cache and from the database and then access its attributes using different pointer, we get an exception saying that such object couldn't be found in the database.

```
// Copying cache pointers
CachePtr<PhdStudent> phdPtr2 = phdPtr;

// Copying database pointers
DbPtr<PhdStudent> phd2 = phd;
```

`DbPtr` provides two additional noteworthy methods - `DbPtr::dbDelete()` and `DbPtr::cacheDelete()`. The first method deletes the persistent object from the database as well as from the cache, the second one hints the cache to drop its cached local copy (the hint may be ignored).

A.8 Querying persistent objects

Before we start with this topic, we need to get familiar with database objects - persistent objects without an OID, descendants of `DatabaseObject`. Database objects usually represent query results from read-only tables or from tables not generated by the IOPC 2 library. They can be also mapped to database tables which are not part of an IOPC 2 generated schema. As query results they don't need to have a database identity, but if they are mapped into some table, they do. This identity is an instance of the `KeyValues` class. They contain a list of key-value pairs which represents any single-attribute or composite (multi-attribute) key.

To illustrate the concept of database objects, we will re-use the previously created table `blacklist`:

```
CREATE TABLE blacklist(id NUMBER PRIMARY KEY, name VARCHAR2(20))
INSERT INTO blacklist VALUES(1, 'Richard Doe')
INSERT INTO blacklist VALUES(2, 'Ola Nordmann')
```

Now we create a database object class `BlackListEntry` whose instances will represent rows from that table:

```
class BlacklistEntry : public iopc::DatabaseObject {
public:
  EInt externalId;
  EString name;

  static void iopcInit(iopc::Type& t) {
    t["db.table"].setStringValue("blacklist");
    t.getAttribute("externalId")["db.column"].setStringValue("id") ↩
       ;
    t.getAttribute("externalId")["db.primaryKey"].setBoolValue( ↩
       true);
    t.getAttribute("name")["db.type.length"].setIntValue(20);
    // not necessary, attribute name and column name are the same:
    t.getAttribute("name")["db.column"].setStringValue("name");
  }
};
```

Because the name of the class and the names of its attributes differ from the schema, we must specify class metadata which describe mapping between this class and the `blacklist` table. The metadata can be specified in a configuration file as well (see Section A.3). Metadata used in this example are listed in the following table:

Metadata	Level	Description
db.table	type	Database table associated with the class
db.column	attribute	Table column associated with the attribute
db.type.length	type / attribute	Database column length, maximum number of characters that can be fetched or stored from/to database for the associated type/attribute.
db.primarykey	attribute	Attribute is part of the class/table primary key.

The simplest way to load an object is to pass its OID or key values to the `DbPtr` constructor. In the follwing example, we will load a blacklist entry with id 2:

```
KeyValues key;
key.add("externalId", 2);
DbPtr<BlacklistEntry> entry(conn, key);
cout << entry->externalId << ": " << entry->name << endl;
```

Output of the example is:

```
2: Ola Nordmann
```

Because the class has identity defined, we can modify the loaded object or even create a new one:

```
DbPtr<BlacklistEntry> newEntry;
newEntry->externalId = 3;
newEntry->name = "Mary Major";
newEntry.bePersistent(conn);
```

Now, to the objects wihtout identity. We create a class that will represent `USER_TABLES` view from the Oracle Data Dictionary, and we will use it to list all tables in the current schema.

```
class UserTable : public iopc::DatabaseObject {
public:
  EString tableName;
  EString tablespaceName;

  static void iopcInit(iopc::Type& t) {
    t["db.transient"].setBoolValue(true);
    t["db.table"].setStringValue("USER_TABLES");
    t.getAttribute("tableName")["db.column"].setStringValue(" ↩
       TABLE_NAME");
    t.getAttribute("tableName")["db.type.length"].setIntValue(30);
    t.getAttribute("tablespaceName")["db.column"].setStringValue(" ↩
       TABLESPACE_NAME");
    t.getAttribute("tablespaceName")["db.type.length"].setIntValue ↩
       (30);
  }
};
```

The `db.transient` metadata tells the library that the class is transient and that it does not have an identity. If we didn't specify this metadata, an exception would be thrown during the library initialisation routine saying that the class needs at least one primary key to be defined.

Basic query that just returns all rows from a table associated with the specified class is very simple. We don't need to construct any `Query` objects:

```
Result<UserTable> userTables(conn);
userTables.open();
for (Result<UserTable>::iterator it = userTables.begin();
  it != userTables.end(); it++) {
  cout << it->tableName << "   " << it->tablespaceName << endl;
}
userTables.close();
```

The example lists all tables from the current schema along with tablespaces containing the tables. Query is executed by calling the `Result<T>::open()` method, a database cursor is created. The results can be then iterated as shown in the example. To free the allocated database cursor, `Result<T>::close()` must be called when the result is not needed any more.

If we want to filter or to sort the query results, we must create a `SimpleQuery` instance and specify our needs. The following example lists the table names in ascending order and restricts the table results to tables from the USERS tablespace.

```
SimpleQuery q(
  "$::UserTable::tablespaceName$ = 'USERS'", "$::UserTable:: ↩
     tableName$ ASC");
Result<UserTable> userTables(conn, q);
userTables.open();
for (Result<UserTable>::iterator it = userTables.begin();
  it != userTables.end(); it++) {
  cout << it->tableName << " " << it->tablespaceName << endl;
}
userTables.close();
```

First parameter of the `SimpleQuery` constructor filters the query result, it is inserted into the WHERE part of the resulting SQL query. Second parameter represents the ORDER BY part. For the query format description please refer to Section 5.5.4.

Finally, database object classes even need not to be bound to one table, they can represent results of any query. In the following example, we add a `columnCount` attribute to the `UserTable` class to represent number of columns of each table. If we join the USER_TABLES and USER_TAB_COLS tables, we may calculate the column count by grouping the results by TABLE_NAME (and TABLESPACE_NAME) and by adding a COUNT(*) column. The modified `UserTable` class would look like this:

```
class UserTable : public iopc::DatabaseObject {
public:
  EString tableName;
  EString tablespaceName;
  EInt columnCount;

  static void iopcInit(iopc::Type& t) {
    t["db.transient"].setBoolValue(true);
    t.getAttribute("tableName")["db.column"].setStringValue(" ↩
       TABLE_NAME");
    t.getAttribute("tableName")["db.type.length"].setIntValue(30);
    t.getAttribute("tablespaceName")["db.column"].setStringValue(" ↩
       TABLESPACE_NAME");
```

```
    t.getAttribute("tablespaceName")["db.type.length"].setIntValue ↩
       (30);
    t.getAttribute("columnCount")["db.column.sql"].setStringValue ↩
       ("COUNT(*)");
    // Not necessary:
    t.getAttribute("columnCount")["db.column"].setStringValue(" ↩
       COLUMN_COUNT");
  }
};
```

The `db.table` metadata was removed. The `db.column.sql` metadata specify the column expression used to select value for this attribute. It is rendered as COUNT(*) AS column_name.

Instead of `SimpleQuery` we now use a `FreeQuery` instance because it allows us to specify everything after the FROM keyword in the SQL SELECT statement:

```
FreeQuery q("USER_TABLES JOIN USER_TAB_COLS USING (TABLE_NAME)  ↩
   GROUP BY TABLESPACE_NAME, TABLE_NAME");
Result<UserTable> userTables(conn, q);
userTables.open();
for (Result<UserTable>::iterator it = userTables.begin();
  it != userTables.end(); it++) {
  cout << it->tableName << " " << it->tablespaceName << " "
    << it->columnCount << endl;
}
userTables.close();
```

The resulting SELECT statement will be:

```
SELECT
  COUNT(*) AS COLUMN_COUNT,
  TABLE_NAME,
  TABLESPACE_NAME
FROM
  USER_TABLES JOIN USER_TAB_COLS USING (TABLE_NAME)
GROUP BY
  TABLESPACE_NAME,
  TABLE_NAME
```

OID objects are queried in a similar way. To illustrate this, we list all persons in our database who are older than 30:

```
SimpleQuery q("$::Person::age$ >= 30");
Result<Person> r(conn, q);
r.open();
for (Result<Person>::iterator it = r.begin(); it != r.end(); it++) ↩
    {
  cout << "name: " << it->name << " age: " << it->age << endl;
}
r.close();
```

Output of the example is:

```
name: Mary Major age: 60
name: Ola Nordmann age: 45
```

As you see, the query selects from the polymorphic views, so even the `Person` descendants are returned. To select only the `Person` instances we must change the query to `FreeQuery` and specify the corresponding simple view name from which to select:

```
FreeQuery q2("$::Person[db.view.sv]$ WHERE $::Person[db.view.sv]:: ↩
   age$ >= 30");

// SQL executed:
SELECT OID FROM Person_SV WHERE Person_SV.Person_age >= 30
```

Only OID column is selected for OID objects because selected objects may be present in the cache. Full object is then either obtained from the cache or loaded from database when accessed[5].

A.9 Caching

Setting up the cache layer is quite complicated topic. Because it was already described in user guide in [1] we will provide only a few basic examples.

In the previous sections we learned how to use the `VoidCache`. Similarly we could use the single-threaded variants of the other two cache implementations provided - the `ArcCacheST` and `LruCacheST` caches:

```
db = new CachedDatabase(
  DriverManager::getDriver("IopcOracle10g").getDatabase("XE"),
  new HashArcCacheST());
db = new CachedDatabase(
  DriverManager::getDriver("IopcOracle10g").getDatabase("XE"),
  new HashLruCacheST());
```

To take advantage of the cache/strategy selector facade and asynchronous cache maintenance provided by `CacheKeeper` we must first associate the `ComposedCache` (no other cache) with the connections. Then we will define cache and strategy selection rules (see Section 5.5.3) on each connection and start the maintainer thread:

```
db = new CachedDatabase(
  DriverManager::getDriver("IopcOracle10g").getDatabase("XE"),
  new ComposedCache());
conn = (CachedConnection*)db->getConnection(
  "username=iopc;password=iopc;autocommit=false");
CacheKeeper cacheKeeper(-15000, 100, 200);
cacheKeeper.setCache(conn, TypeDesc<Employee>::getType(), new  ↩
   HashArcCache());
cacheKeeper.setCache(conn, new HashLruCache());
cacheKeeper.setStrategy(conn, Strategy::lazy);
cacheKeeper.runMaintainer();
```

In the example, we associated the `HashArcCache` (multithreaded variant) with the `Employee` class. Instances of this class manipulated on the `conn` connection will use this cache. All other persistent objects will use the default `HashLruCache`. The lazy strategy will be

[5] In the future, users should be able to choose whether to load full copies from the query results or use the mechanism currently implemented.

used for objects of all types. For other strategy types and their descriptions see [1]. `CacheKeeper` constructor allows to specify maximum cache capacities and other parameters that influence the cache maintenance.

Appendix B

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.