



# Thesis
# A generic, object oriented library for database transactions

*Ghent University*
*Faculty of Applied Science*
*Department INTEC (Information Technology), Software Engineering Group*
*Promoter : Prof.dr.ir. H. Tromp*
*Thesisconductors : ir. G. Premereur and lic. K. Carron*

## Contents



## Abstract

The library, which was written in ANSI-C++ using the latest additions to the standard (including Standard Template Library (STL) and exception handling), forms an object oriented layer above a relational database API (MS-ODBC).

## Problem situation

The relational datamodel, introduced in 1970 by E.F.Codd, still dominates the database world, while modern software development is ruled by the object oriented paradigm. Combining both worlds, by accessing relational data from an object oriented application, creates a few problems which are generally known by the term *impedance mismatch*. These problems arise because of the semantic gap due to the architectural differences between both data models.

**Architectural differences**

| | Relational model | Object oriented model |
|---|---|---|
| **Datastructure** | two-dimensional | complex grouping |
| **Application logic** | decoupled from the data | coupled with the data |
| **Transaction granularity** | set of rows | object |

*Unless you have the right tools, mixing objects and tables is like mixing oil and water. (DBMS, October 1996)*

## Solving the impedance mismatch

### Traditional solutions

A first method is the use of a traditional, procedural API for the database access. However, a lot of the benefits from using the object oriented model are getting lost in that way, f.i. data abstraction and encapsulation. This means that changes to the database schema implies changes to the application code and that's not wanted!

*Some distinct possibilities...*

Embedded SQL :

SQL statements are hard coded in the application program. Of course, this is not flexible and brings the typical disadvantages of SQL, f.i. :

- not user-friendly, f.i. transferring data to the application.
- preprocessor : not type safety
- database-platform dependent

C API, f.i. ODBC (Open DataBase Connectivity, Microsoft) :

Also this method suffers from an abstraction level that is too low and is not completely type safe because of the use of void pointers. However, ODBC is database independent.

In short: It's not wise to solve the impedance mismatch by putting a step back and to program procedural again.

### Existing OO libraries

Another possible solution is to try to approach the relational model as if it was an OO model. This can be achieved with a library that maps the relational concepts, such as tables, rows, columns,... into objects.
This has the disadvantage that the abstraction level still stays fairly low because the programmer still has to deal with tables, rows,...
Above all, such libraries use a lot of inheritance which affects efficiency in a negative way. It's even so, that objects have to be derived from some class before they can be stored in the database. This is certainly a wrong modeling because there's no conceptual difference between a persistent Person object and a Person object who isn't persistent. Being persistent is a kind of use of an object and not a property of the object itself.

### A new approach

The purpose of this thesis was to develop an OO library which raises the abstraction level high enough to make sure that the user doesn't have to work with tables and columns. The user should focus his attention on the business objects itself.
Also, these business objects shouldn't require any modification at all before they can be made persistent. This goal can easily be achieved by using template (generic) classes, which, in turn, brings the desirable property of static type checking (for reasons of efficiency and performance). Another advantage of using templates is their support for built-in datatypes. Wrapper-classes aren't needed as is often the case with the existing libraries.

## Library design aspects

- The library was perceived as a persistent STL-container, which could ultimately become an official part of STL. Special care was taken to allow the use of the STL-algorithms on the new container just as easy as on the other STL-containers.

- Database-platform independence was achieved by using the bridge design pattern. As an example of an engine, the ODBC library engine was developed. This means that the library is ready to be used by all databases for which an ODBC-driver exists. If someone persists at using a database in its native language, a specific engine can easily be constructed.
- The relational-to-object mapping occurs through two user-defined functions in a class derived from the Adapter-class : one for reading an object from the database and one for writing an object to the database.
- The mapping of the datatypes (C++ to database), which are defined as template arguments, happens with the use of RTTI (Run-Time Type Information). This limit's the use of C++ datatypes to the types that are supported internally in the library (extendible by overriding three engine-functions).

## Example

### SQL Database schema

```
create table admin {
  empid numeric(4) not null,
  name char(40),
  street char(25),
  city char(10),
  primary key (empid)
}

create table finan {
  empid numeric(4) not null,
  salary numeric(6),
  primary key (empid),
  foreign key (empid) referencing admin
}
```

### Setting up the library

Create an engine and specify the database schema:

```
Engine *gOdbc = new OdbcEngine();

// Database(database name, engine object, autocommit mode boolean)
Database gFinancial("financial", gOdbc, false);

// Table(database object, table name)
Table gAdmin(gFinancial, "admin");

// DataColumn<C++ datatype>(table object, column name,
//   primary key, foreign key object)
DataColumn<int> gAdminId(gAdmin, "EMPID", true);
DataColumn<string> gAdminName(gAdmin, "NAME");
DataColumn<string> gAdminStreet(gAdmin, "STREET");
DataColumn<string> gAdminCity(gAdmin, "CITY");
Table gFinan(gFinancial, "finan");
DataColumn<int> gFinanId(gFinan, "EMPID", true, &gAdminId);
DataColumn<long> gFinanSal(gFinan, "SALARY");
```

Derive a class from the Adapter-class with two mapping functions:

```
class FinanAdapter : public Adapter<Financial> {
  public:
    void readObject(const Financial& f) {
      gAdminId.setData(f.getId());
      gAdminName.setData(f.getName();
      gAdminStreet.setData(f.getStreet());
```

```
      gAdminCity.setData(f.getCity());
      gFinanSal.setData(f.getSalary());
    }
    void writeObject(Financial *f) {
      f -> Financial::~Financial();
      new (f) Financial(gAdminId.getData(), gAdminName.getData(),
        gAdminStreet.getData(), gAdminCity.getData(), gFinanSal.getData());
    }
}
```

Create a virtual persistent STL-container:

```
FinanAdapter fadap;

// Storage<C++ class>(database object, adapter object);
Storage<Financial> fstore(gFinancial, &fadap);
```

Use the container to perform database transactions:

```
fstore.insert(bob);
fstore.update(amy);
fstore.delete(bill);

// Retrieves the full object using the provided primary key
fstore.find(janie);
```

## Features

- Many-tables to many-objects relationships are possible.
- Automatic consistency update of the whole foreign key cycle.
- Different tables are sorted in an optimal way to eliminate problems with the referential integrity.
- Automatic join between two or more tables that forms one object.
- STL-compatible-and-alike persistent container.
- Insert, update, find and delete functions based on the primary key.
- Queries of arbitrary complexity are possible through an iterator class.