# POLiTe User's Reference

Addendum to
Doctoral Thesis

**Object Persistency in C++**

by
Mgr. Michal Kopecký

# CONTENTS

# EXAMPLES

# FIGURES

# 1. INSTALLATION GUIDE

This section describes actions to be done to install the POLiTe library and to use it in the C++ applications.

## 1.1. Environment Configuration

### 1.1.1. Oracle Database Configuration

Before you can configure the library itself, you should install and configure Oracle client environment on the same machine. In order to be able to compile the applications, you should have at least ORACLE Pro*C pre-compiler components version 7.x or above installed. Also the SQL console named SQL*Plus is recommended. After the Oracle installation, the root directory for the ORACLE software package is referenced as Oracle home and set either in the variable $ORACLE_HOME (Unix) or in HOLM\Software\Oracle registry branch (Windows).

If the Oracle server is installed on the same machine, and you want to run applications against this local database, the database system identification should be set in the $ORACLE_SID variable or registry in the registry.

If you want running the applications across the network, the specification of the default database should be set in the $TWO_TASK variable (Unix) or in the registry key named Local. Having the name of the preferred database set, it is not necessary to specify the name of the database in the applications.

See Oracle installation manuals for more details.

### 1.1.2. C++ compiler configuration

Correctly installed C++ compiler is required in order to rebuild the POLiTe library and to compile the applications. Add following statements to your start-up script (Unix):

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib
export LD_LIBRARY_PATH
```

## 1.2. POLiTe installation

To install POLiTe, do following steps:

1. Extract archive/all_source.tgz to the desired directory. In the following text, this directory will be referred as polite_dir. After decompressing, you should obtain the following directory structure in the HOME directory:

| | |
|---|---|
| `include` | header files |

| | |
|---|---|
| `lib` | The compiled Linux library |
| `linux` | Linux library project |
| `msvc6dll` | Windows library project |
| `msvc6test` | Windows examples |
| `sql` | SQL scripts |

The `sql` subdirectory contains the following files

| | |
|---|---|
| `POLiTeDDL.sql` | The SQL script for creating all database tables needed by the library |
| `Optional.sql` | The SQL script for adding some optional indexes and constraints |

2. Adapt projects to to match your computer configuration: Mainly the path to the Oracle OCI include files and oci.lib file must be changed in MS VC++ 6 projects. The projects use path C:\Oracle\Ora81\oci\include and C:\Oracle\Ora81\oci\lib\. In Linux set variables in linux/include.mk file. Linux default settings are

- `COMPIL_C = gcc` C compiler
- `COMPIL_C++ = gcc` C++ compiler
- `LINKER = gcc` linker
- `AR = ar` ar
- `RANLIB = ranlib` ranlib
- `STRIP = strip` strip
- `OLHOME = ..` polite_dir directory of the package installation
- `ORAHOME = $(ORACLE_HOME)` set to `$ORACLE_HOME` directory

## 1.3. Database configuration

Before you can start the POLiTe applications, you need to prepare required database accounts in the database, and to create necessary tables and other database objects. Examples use user "polite" with the password "polite". When needed, change them in msvc6test/schema.h file and rebuild executables.

Test the database connectivity from your client computer. Try start SQL*PLUS using the command:

```
$ sqlplus your_database_user/your_database_password
```

POLiTe uses some database tables and sequences to store its internal data. These needed tables you can make from the library by calling *OracleDatabase::WriteDDL(ofstream &)* method. See test001 for details.

Created script would contain following text:

```
-- OID generator for Oracle database
CREATE SEQUENCE OID_GENERATOR
START WITH 1
MINVALUE 1
MAXVALUE 999999999
NOCYCLE
NOCACHE;

-- Version generator for Oracle database
CREATE SEQUENCE VERSION_GENERATOR
START WITH 1
MINVALUE 1
MAXVALUE 999999
CYCLE
NOCACHE;
```

1. Connect as the wanted database user

   ```
   $ sqlplus polite/polite
   ```

2. Run the script

   ```
   SQL> @name_of_the_script
   ```

3. Disconnect from the database.

   ```
   SQL> EXIT;
   ```

## 1.4. POLiTe built–in parameters

There are many parameters, which have an influence on the POLiTe behaviour and performance. Some of them you can change in the `lDefs.h` header file. After change, you must rebuild the library and applications to allow the changes to take effect. See section 1.5 for the information.

```
#define POLiTe_VERSION "1.0"          // Version info
```

- Defines version of the used library as the string.

```
#define POLiTe_TEST_FRIEND void POLiTe_Test_Procedure()
```

- If defined, all classes within the library declare this procedure as a friend procedure
- Due to its usefulness for the debugging is this function used in all example programs.

```
#define TL_INFO        0x01
#define TL_INFO_SQL    0x02
#define TL_WARNING     0x08                // 0x04 is not used
#define TL_ERROR       0x20                // 0x10 is not used
```

- Binary masks for different levels of the tracing messages.
    - Informative messages, noting that some method was invoked, finished, etc.
    - Information about SQL statement preparation, execution, etc.
    - Information about warnings generated during program execution
    - Information about errors generated during program execution
- Each POLiTe trace message is assigned to one of the above-defined groups.

```
    #define POLiTe_TRACE_MASK (TL_INFO | TL_INFO_SQL | TL_WARNING |
TL_ERROR)
```

- Defines filter for the logged messages
- If tracing is enabled (see `POLiTe_TRACE` label) messages belonging to one of the allowed groups are stored in the defined text file (see `POLiTe_TRACE_FILE` label)

```
    #define GENLIB_TRACE_MASK_TTY     TL_INFO_SQL
```

- Defines filter for the messages sent to the standard output
- If tracing is enabled (see `POLiTe_TRACE` label) messages belonging to one of the allowed groups are sent to a standard output.

```
#define POLiTe_TRACE                // Tracing enabled
#ifdef POLiTe_TRACE
  #define L_CMDS_TRACE
  #define L_EXCEPTIONS_TRACE
  #define L_STR_TRACE
  #define C_CURSOR_TRACE
  #define C_ORACLECURSOR_TRACE
  #define C_DATABASE_TRACE
  #define C_ORACLEDATABASE_TRACE
  #define C_CONNECTION_TRACE
  #define C_ORACLECONNECTION_TRACE
  #define C_QUEREFPROTO_TRACE
  #define C_QUERY_TRACE
  #define C_COMPLEXQUERY_TRACE
  #define C_PROTOBASE_TRACE
  #define C_QUEREFPROTO_TRACE
  #define C_QUERY_TRACE
  #define C_OBJECTBUFFER_TRACE
  #define C_OBJREF_TRACE
  #define C_OBJECT_TRACE
```

```
#define C_IMMUTABLEOBJECT_TRACE
#define C_DATABASEOBJECT_TRACE
#define C_PERSISTENTOBJECT_TRACE
#define C_REFBASE_TRACE
#define T_REF_TRACE
#define C_PROTOBASE_TRACE
#define T_PROTO_TRACE
#define C_QUERYRESULT_TRACE
#define C_RELATION_TRACE
#define C_ONETOONERELATION_TRACE
#define C_ONETOMANYRELATION_TRACE
#define C_MANYTOONERELATION_TRACE
#define C_MANYTOMANYRELATION_TRACE
#define GC_CHAINEDRELATION_TRACE
#endif //POLiTe_TRACE
```

- Collection of the defined labels for the POLiTe tracing control.
- Labels correspond to source files.
- If the corresponding label is defined, trace messages in the given part of the library are produced.
- If the corresponding label is not defined, trace information in the given part aren't produced.

```
#define POLiTe_TRACE_FILE "stdlog.txt"
```

- Name of the log file

```
#define DEFAULT_UPDATING_STRATEGY US_OnDemand
#define DEFAULT_LOCKING_STRATEGY LS_None
#define DEFAULT_WAITING_STRATEGY WS_Nowait
#define DEFAULT_READING_STRATEGY RS_Cache
```

- Default strategies used by the library
  - Objects changed in the application are propagated back to the database only when explicit request is sent either by the application or by the library itself.
  - Objects (database rows holding its data) are not locked when object is retrieved from the database
  - When object which should be changed are locked on the server from other connection, library will not wait and raise the exception ObjLibException_DatabaseLock

```
#define MAX_CONNECTION 32
```

- Maximal number of the existing connections in the POLiTe based application

```
#define MAX_CONNECTION_PER_DATABASE 16
```

- Maximal number of connections in the POLiTe–based application connected to one database object (note that each database server has its own limitation on total number of opened connections

```
#define AUTO_VIRTUAL_OBJECT_LOAD
```

- If this label is defined, *QueryResult* instances will automatically modify all retrieved instances of the *RefBase* class to refer to correct subclass of the class, which executes the query.
- If this label is not defined, *QueryResult* instances will produce instances of the *RefBase* class whose refer to the same class as the class, which executes the query.
- **This label should be defined to retrieve polymorph results automatically. Otherwise the virtualisation must be done explicitly by the application**

```
#define SQL_EXECUTION_VERSION_6 0x00
#define SQL_EXECUTION_VERSION_7 0x02
#define SQL_EXECUTION_NATIVE    0x01
#define SQL_EXECUTION_MODE      SQL_EXECUTION_NATIVE
```

- The `SQL_EXECUTION_VERSION_6` label defines, that all statements have to be processed using the semantics defined for Oracle server version 6.
- The `SQL_EXECUTION_VERSION_7` label defines, that all statements have to be processed using the semantics defined for Oracle server version 7.
- The `SQL_EXECUTION_NATIVE` label defines, that all statement have to be processed using the native semantics according to the version of the connected Oracle SQL server.
- The `SQL_EXECUTION_MODE` defines the wanted semantics of the executed SQL statements on the server. All SQL statement sent to the server will require this semantics.

```
#define NO_DATABASE_CASCADE_DELETE
```

- If this label is defined, POLiTe generates SQL DELETE statement for each table, containing piece of the deleted object.
- If this label is NOT defined, POLiTe generates SQL DELETE statement only for the root table and supposes, that all other pieces of the object will be deleted automatically on the server using cascade delete.
- NOTE: This label is not defined (it is commented out) in the lDefs.h file

```
#define HASH_TABLE_SIZE 64
```

- The size of the object buffer hash table belonging to one database connection.
- The size of the complete hash table is sizeof(void *) * MAX_CONNECTION * HASH_TABLE_SIZE bytes
- The minimal value is 1, maximal value is 256

```
#define MAX_OBJECTS_IN_BUFFER 2048
```

- Maximal number of objects allowed in the object buffer.
- If the number is exceeded, buffer frees all object not locked in the memory to free its space

```
#define MAX_CLASS_NAME_LEN 64
```

- Maximal length of the name of the table, associated with any of the persistent classes.

## 1.5. Rebuilding of the shared libraries and of the test applications

User can create two modifications of the library. The run-time version should not define `POLiTe_TRACE` and `POLiTe_TEST_FRIEND` labels.

### 1.5.1. Rebuilding of the run-time version of the POLiTe library

1. Comment out both above-mentioned labels in lDefs.h header.
2. Rebuild the library. On Linux, use linux/makefile
3. Copy the shared library to the place known by applications

### 1.5.2. Rebuilding of the testing version of the POLiTe library

4. Define both above-mentioned labels in lDefs.h header.
5. Rebuild the library. On Linux, use linux/makefile
6. Copy the shared library to the place known by applications (for example polite_dir/lib)

### 1.5.3. Rebuilding of the test applications

Set wanted project in the MS VC++ from msvc6test\test.dsw workspace and rebuild it.

On Linux use linux/make*nnn* makefile.

## 1.6. Using POLiTe library in application source files

In your source or header files, which use the library, include the `POLiTe.h` header file using the directive

```
#include <POLiTe.h>
```

In addition to compiling other ORACLE-based application:

On Linux:

1. Compile the object files with the `-I`polite_dir`/include` option.
2. Link your application with the `-L`polite_dir`/lib -lPOLiTe` options

# 2. POLITE USER MANUAL

POLiTe provides an interface between the object-oriented C++ application and the SQL database engine. POLiTe is a library which manages the persistency of application domain objects using relational database servers. Moreover, it presents rows of external (by other application maintained) tables as C++ objects.

## 2.1. POLiTe Interface

The basic interface for the communication between the application and the library is provided by the solitaire instances of *ObjectBuffer* and *ClassRegister* classes.

The *ObjectBuffer* class provides functions for setting the application environment.

The *ClassRegister* holds the information about the object model of the database.

Both instances are defined inside the library as.

```
class ObjectBuffer ObjectCache;
class ClassRegister Class;
```

### 2.1.1. Environment

The following POLiTe environment properties can be set using the *ObjectBuffer* interface.

#### *2.1.1.1. Updating strategy*

The updating strategy determines the way, how the updated objects are written into the database. Changes can be either propagated immediately to the database or they can be deferred and written to the database explicitly by calling the object's *Update()* method.

There are defined following type and constants for the update strategy:

```
enum UpdatingStrategy {
  US_Default                = 0x0001,
  US_Current                = 0x0002,
  US_Inherited              = 0x0003,
  US_OnDemand               = 0x0004,
  US_Immediately            = 0x0005
  };
```

The demanded updating strategy can be set by methods.

***bool ObjectBuffer::SetUpdateStrategy(enum UpdatingStrategy anUpdateStrategy);***

> Sets the UpdateStrategy. Update strategy can be set to *US_OnDemand* or *US_Immediately*. Setting the update strategy to the US_Default sets the built–in default update strategy (see 1.4). Setting the update strategy to the US_Current does nothing. It can be used as the default value of parameters. Value US_Inherited changes the strategy to built-in default, because there is no more general settings to inherit form.

***enum UpdatingStrategy ObjectBuffer::CurrentUpdatingStrategy() const;***

Returns the current setting of the UpdateStrategy.

***Example 1 - Setting of the update strategy***

```
enum UpdateStrategy UStrategy;
UStrategy=ObjectCache.CurrentUpdatingStrategy();
// get the current strategy
ObjectCache.SetUpdatingStrategy(US_Immediately);
// set strategy to US_Immediately
```

#### 2.1.1.2.*Waiting strategy*

Waiting strategy identifies the strategy used when the corresponding rows or tables of a persistent object are locked in the database. Either an error can occur or the object, which tries to access the data from the database, will wait until the data is unlocked.

There are defined following type and constants for the waiting strategy settings in the library:

```
enum WaitingStrategy {
  WS_Default                = 0x0040,
  WS_Current                = 0x0080,
  WS_Inherited              = 0x00C0,
  WS_Wait                   = 0x0100,
  WS_Nowait                 = 0x0140
  };
```

The demanded waiting strategy can be set by methods.

***bool ObjectBuffer::SetWaitingStrategy(enum WaitingStrategy aWaitingStrategy);***

Sets the waiting strategy. Waiting strategy can be set to W*S_Wait* or W*S_Nowait.* Setting the waiting strategy to the WS_Default sets the built–in default waiting strategy.

***enum WaitingStrategy GenLibInterface::CurrentWaitingStrategy() const;***

Returns the current waiting strategy.

***Example 2 - Setting of the waiting strategy***

```
enum WaitingStrategy WStrategy;
WStrategy=ObjectCache.CurrentWaitingStrategy();
// get the current strategy
ObjectCache.SetWaitingStrategy(WS_Nowait);
// set strategy to WS_Nowait
```

#### *2.1.1.3.Locking strategy*

The object (when it loads data from the database) locks a table row either in none, shared or exclusive mode.

There are defined following type and constants for the locking strategy settings in the library:

```
enum LockingStrategy {
  LS_Default                = 0x0008,
  LS_Current                = 0x0010,
```

```
  LS_Inherited              = 0x0018,
  LS_None                   = 0x0020,
  LS_Shared                 = 0x0028,
  LS_Exclusive              = 0x0030
  };
```

The demanded locking strategy can be set by methods.

***bool ObjectBuffer::SetLockingStrategy(enum LockingStrategy aLockingStrategy);***

Sets the locking strategy that the object should use when it loads itself from the database. The locking strategy can be set to *LS_None*, *LS_Shared* or *LS_Exclusive*. Setting the locking strategy to the LS_Default sets the built–in default locking strategy.

The locking strategy is used for a row locking and for a table locking. The row locking takes place, whenever the object is retrieved from the database. Only LS_None and LS_Exclusive strategies work with the row locking. The LS_Shared strategy works the same way as the LS_None strategy for a row locking. The table locking is used only if the application calls the Object::LockTable() method explicitly. Table locking uses all three levels of locking. See section 2.2.5 for a LockTable() method explanation.

***enum LockingStrategy ObjectBuffer::CurrentLockingStrategy() const;***

Returns the current locking strategy.

#### *2.1.1.4.Reading strategy*

The object (when it is accessed via reference and is already loaded into the memory can either use the loaded copy, reload data from the database or check, if there are newer data in the database first..

There are defined following type and constants for the reading strategy settings in the library:

```
enum ReadingStrategy {
  RS_Default                = 0x0200,
  RS_Current                = 0x0400,
  RS_Inherited              = 0x0600,
  RS_Cache                  = 0x0800,
  RS_Database               = 0x0A00,
  RS_Timestamp              = 0x0C00
  };
```

The demanded reading strategy can be set by methods.

***bool ObjectBuffer::SetReadingStrategy(enum ReadingStrategy aReadingStrategy);***

Sets the reading strategy that the object should use when it it is already loaded into memory. The reading strategy can be set to *RS_Cache*, *RS_Database* or *RS_Timestamp*. Setting the reading strategy to the RS_Default sets the built–in default reading strategy.

The RS_Timestamp strategy works only with PersistentObject descendants. Otherwise the LS_Timestamp strategy works the same way as the RS_Cache strategy.

***enum ReadingStrategy ObjectBuffer::CurrentReadingStrategy() const;***

Returns the current reading strategy.

#### *2.1.1.5. More specific setting of strategies*

It is possible to change the individual strategies on more levels of the library. Additional levels are:

- Database class
- Connection class
- Object class
- Ref<T> class

All methods for settings and obtaining strategies are defined on all these levels of granularity.

Each new database inherits its settings from the ObjectBuffer. Each new Connection inherits its settings from the database and references provided by query results inherits this settings from the connection.

Each object remembers the setting of strategies from last reference that was used to access it.

### 2.1.2. POLiTe Tracing

#### *2.1.2.1.How the tracing works*

The POLiTe library provides the user many possibilities how to control trace information produced by the library. Depending on the settings different amount of tracing messages appear in a log file or on the standard output (or both).

The availability, resp. the unavailability of the POLiTe tracing features is determined by the *_TRACE labels. See section 1.4 for list of all possible labels. The main label is the label POLiTe_TRACE. If this label is defined, it is possible to use POLiTeLogWriter instance of LogWriter class, function logmsg(const char *) and its overloaded variants for the activity tracing.

In addition to the main label POLiTe_TRACE, other labels control amount of the generated information. If the label for the given source file is defined, library creates the tracing information for the given part of the library.

Each line of the tracing information belongs to one of predefined groups. The following group constants are defined in the library:

- `TL_INFO 0x01`

  Information about invoking and finishing POLiTe methods

- `TL_INFO_SQL 0x02`

  Information about executed SQL statements

- `TL_WARNING 0x08`

  Warnings

- `TL_ERROR 0x20`

Errors

POLiTe uses function logmsg(const int grp, const char *msg) and its variants for message generation. Two binary masks - one for the output to the text file and second for the output to the standard output - are then used to filter types of the information. Only messages belonging to the allowed groups are sent to the log file, or to the standard output. The control mask variables are:

- `int POLiTeLogWriter.TraceMask`

  Controls the log file output

- `int POLiTeLogWriter.TraceMaskTTY`

  Controls the standard output

Original values of those two variables are set to pre-defined built–in constants `POLiTe_TRACE_MASK` and `POLiTe_TRACE_MASK_TTY (see 1.4)`.

The name of the log file is set in the predefined built–in constant `POLiTe_TRACE_FILE (see 1.4)`.

### *2.1.2.2.How to use the tracing in the application*

First, the tracing must be allowed. User can define wanted set of the `*_TRACE` labels (at least the `GENLIB_TRACE` label must be defined). After changing the `lDefs.h` file the library must be recompiled.

After making the program, run it and see the log file for needed information. The default name of the log file is `stdlog.txt`.

***Example 3 - Tracing POLiTe statements***

The user wants to see the SQL statement executed by the program.

Define at least the `GENLIB_TRACE` (main trace label), C_CURSOR_TRACE, C_ORACLECURSOR_TRACE, C_DATABASE_TRACE, C_ORACLEDATABASE_TRACE, C_CONNECTION_TRACE and C_ORACLECONNECTION_TRACE labels.

Before and after tested section add statements, changing the values of the tracing masks. The code should look like:

```
#ifdef POLiTe_TRACE
POLiTeLogWriter.TraceMask = TL_INFO_SQL | TL_ERROR;
//output to the file
#endif
//here is(are) the statement(s) to be traced
#ifdef GENLIB_TRACE
POLiTeLogWriter.TraceMask = 0x00;
#endif
```

This approach produces only small log file, which contains needed information about important part of the program.

It is also possible to write own text to the logfile. Suppose that the build–in constant `POLiTe_TRACE_MASK` is set to value "`TL_INFO_SQL | TL_ERROR`". Then it is possible to use following code:

```
#ifdef GENLIB_TRACE
logmsg(TL_INFO_SQL, "HERE-IT-BEGINS");
#endif

//here is(are) the statement(s) to be traced

#ifdef GENLIB_TRACE
logmsg(TL_INFO_SQL, "HERE-IT-ENDS");
#endif
```

The second version produces large amount of the information about all of the executed SQL statements, binding variables, etc. The important part of the log file is enclosed between lines containing "HERE-IT-BEGINS" and "HERE-IT-ENDS" strings, which are easy to find.

## 2.2. Database

### 2.2.1. Definition of database dependent classes

The database server is accessed via three classes: *Database*, *Connection* and *Cursor*. These classes represent an abstraction of the database server, of the connection to the database server and of the database cursor. Each of these three classes should have one descendant, which implements the code for the communication with one family of database servers. In the POLiTe library the classes *OracleDatabase*, *OracleConnection* and *OracleCursor* for accessing the Oracle database servers version 7 and above are implemented.

### 2.2.2. Creating of a database

Each real database server that should be accessible from the application must be represented internally as the instance of the *OracleDatabase* class.

This instance must be created and then assigned to correspoding real server by calling the *Database::Assign(char *ConnectString)*. To obtain an assigned instance of the database class in one step, it is possible to pass the connect string directly to the constructor. The connect string must have a form required by the SQL*Net (a network communication protocol used between the Oracle servers and clients). The correct syntax of the connect strings for the SQL*Net version 2 is simply a string containing the name of the database.

If the connect string is empty, or the instance of the Database class was not assigned, the client tries to connect to the default database. See section 1.1.1 for information about the default database settings.

***Example 4 - Assigning a database***

```
class OracleDatabase SampleDatabase("");
// Default database

class OracleDatabase SampleDatabase;
SampleDatabase.Assign("primary_db");
// Named database

class OracleDatabase SampleDatabase("primary_db");
// Named database in one step
```

The SQL*Net protocol uses the `tnsnames.ora` file to translate the form necessary for the successfull client-server communication.

The fragment of the `tnsnames.ora` file defining the primary_db database can look like:

```
primary_db =
  (DESCRIPTION =
    (ADDRESS_LIST =
        (ADDRESS =
          (COMMUNITY = TCP.world)
          (PROTOCOL = TCP)
          (Host = 192.168.1.1)
          (Port = 1526)
        )
    )
    (CONNECT_DATA =
       (SID = DB1)
       (GLOBAL_NAME = DB1.world)
    )
  )
```

This file as well as the other SQL*Net configuration files are stored in the /var/opt/oracle directory (Solaris), or in the $ORACLE_HOME/network/admin directory. See Oracle administration documentation for the detailed SQL*Net protocol settings.

### 2.2.3. Connection to and disconnection from the database

A connection to the particular database can be created by executing the *Database::Connect(const char *username, const char *password)* method. This method returns a pointer to a new *Connection* instance.

***Example 5 - Connecting to the database***

```
Connection *DbCon = NULL;
printf("Connecting to the database as user \"polite\" ...\n");
try {
  DbCon = SampleDatabase.Connect("polite","polite");
  }
catch (ObjLibException &X) {
  printf("... NOT connected\n");
  throw;
};
printf("... successfully connected\n");
```

Each *Connection* object represents one database connection. It provides functions for manipulating with the data in database. Each persistent object (i.e. object that was read from or written to the database) has it's associated database connection. Notice that there can be more connections to one physical database.

The user can close the database connection by calling the *Connection::Disconnect() method*.

```
DbCon -> Disconnect();
```

This method sends a COMMIT command to the database before closing the connection. To disconnect without committing the database the method *Connection::Abort()* can be used.

```
DbCon -> Abort();
```

### 2.2.4. Direct access to the database using SQL

The relational database can be accessed by POLiTe application via C++ objects. The SQL commands, which manipulate these objects in database are constructed and executed automatically by object methods hidden to the user.

POLiTe also provides a direct way of communication with the database using SQL commands written by the user.

These commands can be sent and executed by the Connection class using *the Connection::Sql(...)* method or the << *operator*. These methods are defined as follows.

***virtual bool Connection::Sql(const char *SqlCommand);***

The given SQL statement is executed using the given database connection.

***virtual Connection & Connection::operator << (const char *SqlCommand);***

This method is equivalent to the above-defined method.

***Connection &operator << (const class Cmd &Command);***

Sends a command specified by the object derived from the POLiTe class *Cmd* to the connection.

***Example 6 - Direct SQL statement execution***

```
DbCon->Sql(
"CREATE TABLE TEST ( "
"PK NUMBER(6) CONSTRAINT TEST_PK PRIMARY KEY, "
"DATA VARCHAR2(20) "
")"
);
```

The above statement is equivalent to :

```
*DbCon  <<
"CREATE TABLE TEST ( "
"PK NUMBER(6) CONSTRAINT TEST_PK PRIMARY KEY, "
"DATA VARCHAR2(20) "
")";
```

And also to:

```
*DbCon  <<  CmdSql(
"CREATE TABLE TEST ( "
"PK NUMBER(6) CONSTRAINT TEST_PK PRIMARY KEY, "
"DATA VARCHAR2(20) "
")"
);
```

Before the execution of this function all objects stored in memory from the same database are updated to assure a consistent state of database.

Some basic database commands are available as predefined sub-classes of the *Cmd* class. These classes are:

- *CmdSql* class

  use *CmdSql(const char *SqlCommand)* constructor for creating instances

- *CmdCommit* class

  use *CmdCommit()* constructor for creating instances,

- *CmdRollback* class

  use *CmdRollback()* constructor for creating instances,

- *CmdSavepoint* class

  use *CmdSavepoint(const char *Name)* constructor for creating instances,

- *CmdRollbackToSavepoint* class

  use *CmdRollbackToSavepoint(const char *Name)* constructor for creating instances

Instances of these classes can be sent directly to the database using the << operator and can be used together with ordinary SQL commands sent as strings. This mechanism provides an easy way of direct communication with the database.

***Example 7 - Direct SQL statement execution***

```
*DbCon
<<
"CREATE TABLE EMPLOYEES ("
"CREATE TABLE TEST ( "
"PK NUMBER(6) CONSTRAINT TEST_PK PRIMARY KEY, "
"DATA VARCHAR2(20) "
")"
<<
SqlCmd("CREATE UNIQUE INDEX TEST_DATA ON TEST(DATA)")
<<
CmdCommit();
```

### 2.2.5. Transaction control

The *Connection* class defines functions for managing database transactions.

***virtual bool Connection::Commit();***

Finishing and accepting the transaction. It means the saving the objects from the memory to the database and the request to the database to commit. This function can be issued also by using the command „<< CmdCommit()".

***virtual bool Connection::Rollback();***

Transaction rollback to the state at the beginning of the transaction, removing all the copies of the persistent objects from the memory and requesting the database to rollback. This function can be issued also by using the command „<< CmdRollback()".

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

Inserts a savepoint of the given name into the current transaction. This function can be issued also by using the command „<< CmdSavepoint(char *Name)".

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

Database rollback to savepoint of the given name. The changes made between Savepoint(Name) and the current state of the database are cancelled. This function can be issued also by using the command „<< CmdRollbackToSavepoint(char *)".

***Example 8 - Savepoints***

```
OracleDatabase SampleDatabase("");
Connection *DbCon = NULL;

DbCon = SampleDatabase.Connect(”polite”,”polite”);
// insert a new row
DbCon->Sql(
  "INSERT INTO TEST (PK,DATA) "
  "VALUES (1,’First row’)"
  );
// put savepoint
DbCon->Savepoint("sp1");
// change data
DbCon->Sql("UPDATE TEST SET DATA=’2nd row’ WHERE PK=1");
// rollback to savepoint sp1
DbCon->RollbackToSavepoint("sp1");
// commit database
DbCon->Commit();
```

the same piece of the program can be also written using the << operator

```
OracleDatabase SampleDatabase("");
Connection *DbCon = NULL;
DbCon = SampleDatabase.Connect(”polite”,”polite”);
// insert a new employee
DbCon << "INSERT INTO TEST (PK,DATA) "
         "VALUES (1,’First row’)"
// put savepoint
      << CmdSavepoint("sp1")
// change salary
      << "UPDATE TEST SET DATA=’2nd row’ WHERE PK=1"
// rollback to savepoint sp1
      << CmdRollbackToSavepoint("sp1")
// commit database
      << CmdCommit();
```

### 2.2.6. Object and table locking

Objects are locked in the database according to the settings of the LockingStrategy (see 2.1.1.3). If the LockingStrategy is set to `LS_None` or `LS_Shared`, the retrieved objects are not locked in the database. The rows containing object data are locked exclusively at the time the changes on the object instance are propagated back to the database. Until commiting of the connection, all other connections will read old data and attempt to change them from those connections (and therefore to lock rows exclusively) will detected by the server. If the LockingStrategy is set to LS_Exclusive all rows containing the retrieved instance will be exclusively locked in time of retrieving. Detected collisions with other connections will be solved according to setting of the WaitingStrategy parameter. If the WaitingStrategy is set to `WS_Wait`, the process communicating through the second connection will be suspended until the first connection release locks on the table

row. If the WaitingStrategy is set to `WS_Nowait`, the process communicating through the second connection raises the *ObjLibException_DatabaseLock* exception.

Moreover, the *ProtoBase* class defines method for table–level locking.

***virtual bool ProtoBase::LockTable(***
***class Connection *aDbCon,***
***enum LockingStrategy aLockingStrategy = LS_Default,***
***enum WaitingStrategy aWaitingStrategy = WS_Default***
***);***

This method will try to lock the whole table associated with the class in the database. If the table is already locked and the *WS_Nowait* strategy was chosen, the method raises the exception *ObjLibException_DatabaseLock* else the process will wait untill previous lock on the table is released. The following two exaples show using of the *LockTable* method for the co-ordination of processes inside critical region of the code.

***Example 9 - Database table locking, wait***

```
Semaphore_class.LockTable(DbCon,LS_Exclusive,LS_Wait);
// try to lock table exclusively and wait
/*
Critical section of the code.
No other connection can do the same think at the same time.
*/
DbCon->Commit();
// release the lock
```

***Example 10 - Database table locking, nowait***

```
try {
Semaphore_class.LockTable(DbCon,LS_Exclusive,LS_Nowait);
// try to lock table exclusively
/*

There is only one instance of this program

*/
DbCon->Commit();
// release the lock
}
catch (ObjLibException_DatabaseLock &X) {
    printf("Sorry, there can be only one ...\n"
}
```

## 2.3. Persistent classes and objects

POLiTe provides four base classes for maintaining object persistency:

- *Object* class
- *ImmutableObject* class
- *DatabaseObject* class

- *PersistentObject*

These classes are abstract and all user defined classes for persistent object should be derived from one of those classes.

### 2.3.1. Class Mapping

Class mapping describes the position of the class in the specialisation graph and associates it with the data source code.

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

**CLASS(NameOfClass);**
**ABSTRACT_CLASS (NameOfClass);**

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

**PARENTS("ParentClassList");**

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

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

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

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

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

### 2.3.2. Member attribute declaration

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

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

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

| | | | |
|---|---|---|---|
| | dbDouble(x); | | double x() const; |

### 2.3.3. Member attribute mapping

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

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

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

respectively

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

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

Available mappings are listed below

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

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

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

respectively

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

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

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

respectively

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

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

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

### 2.3.4. Defining classes for persistent objects

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

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

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

#### *2.3.4.1. Object descendants*

Descendants of this class can be used to access data resulted from any SQL select statement. They can represent any projection onto some columns of join of any number of database tables. They can be resulted from a complex SQL query. They allow using GROUP BY and HAVING clauses and aggregate function operators.

This type of objects serves for read-only access to the database. Instances retrieved from the database are in general not identified by any combination of attributes and are not really persistent. They can be changed after retrieving from the database, but changes will not be propagated back into database.

#### *2.3.4.2. ImmutableObject*

Descendants of this class provide access to standard database tables and views. The structure of the class (i.e. it's attributes) must correspond to the existing table. All persistent attributes of an object must correspond with some column of only one database table. Each class instance corresponds to one

row in a specified database table. Using of GROUP BY and HAVING clauses, aggregate function operators is not allowed.

Changes made in the memory are not propagated back to the database.

#### *2.3.4.3.DatabaseObject*

Descendants of this class work similar to ImmutableObject descendants. The only exception is that all changes of their instances are propagated back to the database.

#### *2.3.4.4.PersistentObject*

Descendants of this class act as real persistent C++ objects, which use relational database to achieve their own persistency. Each instance of PersistentObject class and it's descendants is identified by a unique OID. The structure of database tables must be defined to correspond to the class structure. PersistentObject class supports generalisation-specialisation hierarchy of classes. Multiple inheritance is allowed. Each derived class is responsible only for mapping, load and store of its own attributes to associated database table. For handling with attributes of its predecessors, parent methods are called.

The structure of relational database tables required for the classes derived from the PersistentObject class will be described later in this document.

When defining new class T, the library generates the reference type Ref<T> as well as query result type Result<T> automatically.

#### *2.3.4.5.Object prototypes*

Object prototypes were introduced to provide static-like functions and services of classes. Its presence decreases the number of functions, which should be redefined by the user of the library for each new persistent class. The prototype of class T has the type Proto<T> derived form ProtoBase class. It's suggested to name these prototypes *NameOfTheClass*_class, where *NameOfTheClass* is the class name. If you don't know the name of the prototype instance for particular class, its address can be found using the ClassRegister instance named Class.

The Expression Class["NameOfTheClass"] return address of the prototype according to the name of the class.

### 2.3.5. Deriving new persistent subclasses

The following sections describe the way, how to derive classes for maintaining persistent objects. All examples use the following class hierarchy.

*Figure 1 - Sample database structure*

#### 2.3.5.1.Deriving a new persistent subclass of Object class

It is supposed, that there are tables defined in the database, and the user wants to access the data, representing rows of the result of some complicated query, which can collect data from a number of tables. Each row of the result should appear as one object instance inside the application. This section describes how to define appropriate C++ class to maintain such type of data in the application.

For every class derived from the Object class the parts of the corresponding SQL query must be specified (i. e. names of database tables, attributes and functions which have to be selected, select conditions, grouping etc.).

***Example 11 - Deriving a new subclass of DatabaseObject class***

There exists views ALL_TABLES and ALL_USERS in the Oracle database

```
ALL_USERS(
 USERNAME VARCHAR2{30),
 USER_ID NUMBER,
 ...
 );

ALL_TABLES(
 OWNER VARCHAR2{30),
 ...
 );
```

We want to obtain user names together with their identifications and numbers of their tables. The corresponding SQL query would be:

```
SELECT ALL_USERS.USERNAME, ALL_USERS.USER_ID, COUNT(*)
FROM ALL_USERS, ALL_TABLES
WHERE ALL_USERS.USERNAME=ALL_TABLES.OWNER
GROUP BY ALL_USERS.USERNAME, ALL_USERS.USER_ID
HAVING COUNT(*) > 1;
```

This query could be represented in the application by the following class:

```
class TableCount : public Object
{
 // Declare the class ...
 CLASS(TableCount);
 // ... and its associated SQL statement
 FROM("ALL_USERS, ALL_TABLES");
 WHERE("ALL_USERS.USERNAME=ALL_TABLES.OWNER");
 GROUP_BY("ALL_USERS.USER_ID,ALL_USERS.USERNAME");
 HAVING("COUNT(*)>1");
 // Declare member attributes and access methods
 dbString(Owner);
 dbLong(UserID);
 dbShort(NameLength);
 dbInt(NumOfTables);
 // Map attributes
 MAP_BEGIN
       mapString(Owner,"ALL_USERS.USERNAME",30);
       mapLong(UserID,"ALL_USERS.USER_ID");
       mapShort(NameLength,"LENGTH(ALL_USERS.USERNAME)");
       mapInt(NumOfTables,"COUNT(*)");
 MAP_END;
public:
 // Initialisation of the object.
 TableCount() {_Owner=NULL; _NumOfTables=0;};
 // Destruction of the object.
 // _Free() must be called at the beginning.
 ~TableCount() {_Free(); StrFree(_Owner);};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(TableCount);
// Define the prototype UserTable_class
PROTOTYPE(TableCount);
```

The POLiTe library can create SQL script that creates tables needed to store just defined objects. This script is made by method *ClassRegister::WriteDDL(ofstream &, Database &)* method. See exaple test001 for details.

Created script would contain following text:

```
-- Table OID_ROOT associated with class PersistentObject
CREATE TABLE OID_ROOT(
OID NUMBER(10),
VERSION NUMBER(10),
CLASS_NAME VARCHAR2(64),
CONSTRAINT OID_ROOT_PK PRIMARY KEY(OID)
);
```

```
    -- Table PROJECT associated with class Project
    CREATE TABLE PROJECT(
    OID NUMBER(10),
    TITLE VARCHAR2(15),
    LANGUAGE VARCHAR2(25),
    CONSTRAINT PROJECT_PK PRIMARY KEY(OID),
    CONSTRAINT PROJECT_FK_OID_ROOT FOREIGN KEY(OID) REFERENCES
OID_ROOT(OID) ON DELETE CASCADE
    );


    -- Table PERSON associated with class Person
    CREATE TABLE PERSON(
    OID NUMBER(10),
    FIRST_NAME VARCHAR2(25),
    LAST_NAME VARCHAR2(25),
    PHONE_NR VARCHAR2(15),
    CONSTRAINT PERSON_PK PRIMARY KEY(OID),
    CONSTRAINT PERSON_FK_OID_ROOT FOREIGN KEY(OID) REFERENCES
OID_ROOT(OID) ON DELETE CASCADE
    );


    -- Table PROGRAMMER associated with class Programmer
    CREATE TABLE PROGRAMMER(
    OID NUMBER(10),
    EMP_NR NUMBER(10),
    LANGUAGE VARCHAR2(25),
    CONSTRAINT PROGRAMMER_PK PRIMARY KEY(OID),
    CONSTRAINT PROGRAMMER_FK_PERSON FOREIGN KEY(OID) REFERENCES
PERSON(OID) ON DELETE CASCADE
    );


    -- Table LEADER associated with class Leader
    CREATE TABLE LEADER(
    OID NUMBER(10),
    CONSTRAINT LEADER_PK PRIMARY KEY(OID),
    CONSTRAINT LEADER_FK_PROGRAMMER FOREIGN KEY(OID) REFERENCES
PROGRAMMER(OID) ON DELETE CASCADE
    );
```

Table definitions are topologically ordered. Definition of table associated to class follows definitions of tables associated with all its predecessors.

It is also possible to generate definition for individual class using method *ProtoBase::WriteDDL(ofstream &, Database &).*

### 2.3.6. Deriving a new subclass of ImmutableObject class

It is supposed, that there is a table defined in the database, and the user wants to access the rows of this table. On the other hand it doesn't want or cannot change values in the database. Each row of the table appears as one object instance inside the application. Not all columns of this table or view need to be selected, but the subset of selected columns must contain the primary key. This section describes how to define appropriate C++ class to maintain such type of data in the application.

For every class derived from the *ImmutableObject* class the user must provide the name of a database table which stores the objects and the names of database columns corresponding to attributes of this class. ImmutableObjects are identified by values of their key attributes (not by OIDs). The names of the key attributes and corresponding database columns must be specified.

Destructor of each class must call a _Free() method at its beginning. This method unregisters object from the object buffer and stores its values to the database, if necessary, before the object is destroyed. After calling the _Free() method object can start other activities.

***Example 12 - Deriving a new subclass of ImmutableObject class***

We use similar view as in previous example, the view USER_TABLES

```
USER_TABLES(
 TABLE_NAME VARCHAR2{30),
 TABLESPACE_NAME VARCHAR2{30),
 PCT_FREE NUMBER{3),
 PCT_USED NUMBER(3),
 ...
 );
```

To access its data, the application can define class

```
class UserTable : public ImmutableObject
{
 // Declare the class ...
 CLASS(UserTable);
 // ... and its associated table
 FROM("USER_TABLES");
 // Define member attributes
 dbString(Name);
 dbString(Tablespace);
 dbShort(PctFree);
 dbShort(PctUsed);
 // Map primary key
 MAPKEY_BEGIN
      mapString(Name,"#THIS.TABLE_NAME",30);
 MAPKEY_END;
 // Map other attributes
 MAP_BEGIN
      mapString(Tablespace,"#THIS.TABLESPACE_NAME",30);
      mapShort(PctFree,"#THIS.PCT_FREE");
      mapShort(PctUsed,"#THIS.PCT_USED");
 MAP_END;
public:
 // Initialisation of the object.
 UserTable() {_Name=NULL; _Tablespace=NULL; _PctFree=0;
_PctUsed=0;};
 // Destruction of the object.
 // _Free() must be called at the beginning.
 ~UserTable() {_Free(); StrFree(_Name); StrFree(_Tablespace);};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(UserTable);
// Define the prototype UserTable_class
PROTOTYPE(UserTable);
```

### 2.3.7. Deriving a new subclass of DatabaseObject class

The definition of DatabaseObject descendants is the same as in case of ImmutableObject descendants. But this time the library assumes that it is possible to update data in the database according to changes made in the application.

***Example 13 - Deriving a new subclass of ImmutableObject class***

Let suppose the table OLD_PROJECTS

```
OLD_PROJECTS(
  YEAR NUMBER(4),
  NAME VARCHAR2 (25),
  STATUS CHAR2 (1), -- Open/Close
  SUPERVISOR VARCHAR2 (30),
  CONSTRAINT OLD_PROJECTS_PK PRIMARY KEY(YEAR,NAME)
  ) ;
```

in the database. An OldProject class which works with rows of this table as with objects could look as follows:

```
class OldProject : public DatabaseObject
{
 // Declare the class ...
 CLASS(OldProject);
 // ... and its associated table
 FROM("OLD_PROJECTS");
 // Define member attributes
 dbShort(Year);
 dbString(Name);
 dbChar(Status);
 dbString(Supervisor);
 // Map primary key
 MAPKEY_BEGIN
      mapShort(Year,"#THIS.YEAR");
      mapString(Name,"#THIS.NAME",25);
 MAPKEY_END;
 // Map other attributes
 MAP_BEGIN
      mapChar(Status,"#THIS.STATUS");
      mapString(Supervisor,"#THIS.SUPERVISOR",30);
 MAP_END;
public:
 // Initialisation of the object.
 OldProject() {_Year=0; _Name=NULL; _Supervisor=NULL;};
 // Destruction of the object.
 // _Free() must be called at the beginning.
 ~OldProject() {_Free(); StrFree(_Name); StrFree(_Supervisor);};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(OldProject);
// Define the prototype UserTable_class
PROTOTYPE(OldProject);
```

### 2.3.8. Deriving a new subclass of PersistentObject class

Descendants of this class represent real persistent objects. Instances of these classes are identified by an unique OID. In contrast to descendants of Object, ImmutableObject and DatabaseObject classes, classes derived from PersistentObject class can form a hierarchy. In case of Object, ImmutableObject and DatabaseObject descendants, the classes are created to fit to format of some existing table or to format of the result of concrete select statement. On the other hand, the descendants of the PersistentObjects class are defined first, and it is necessary to create database tables according to them.

Each Object subclass is associated to one table in the relational database. The PersistentObject class itself is associated with the table named OID_ROOT. Attribute values of each instance of the PersistentObject subclass are stored in the tables associated with the given subclass and its parents up to PersistentObject.

For every class derived from the *PersistentObject* class the user must provide the name of a database table which stores the objects and the names of database columns corresponding to attributes of this class. The key of the associated table nedd not to be declared, because it is inherited parents. PersistentObjects are identified by OIDs.

Let suppose the Person class derived from the PersistentObject class directly and associated with the PERSON table in the database. Instances of the Person class are stored in OID_ROOT and

PERSON tables. To obtain all attributes of the Employee class instance, both tables must be joined together. To allow correct join of all pieces of one object from the database, all associated tables must have defined additional column OID. This column should be defined as NUMBER(9) PRIMARY KEY column (the same type as the OID column in the OID_ROOT table).

Because an OID column of newly created table refers to the OID column of the table associated with all direct parent classes, it is useful to express this fact in the database explicitly by adding clause "REFERENCES name_of_parent_table(OID)" to the OID column definition for each direct parent of the class.

If the database cascade delete feature is wanted to be used, additional clause "ON DELETE CASCADE" must be added to the foreign key definition. See section 1.4 for information how to enable or disable cascade delete feature.

Table associated with the given class must have defined (except the OID column) one additional column for each attribute added by the given class. Column types must be compatible with the type of the attribute.

To keep data consistent, each attribute should be accessed via two access methods, one for reading and one for changing the value of the attribute. The access method for changing attribute must call the MarkAsDirty() method before its finish. This method marks the object in memory as a dirty object, and so it can be written back to the database before the occupied memory is released.

A prototype instance for the PersistentObject class is defined in the `cPersistentObject.cpp` file inside a persistent module:

```
class Proto<PersistentObject> PersistentObject_class;
```

***Example 14 - Deriving a new subclass of PersistentObject class***

Classes Person, Programmer, Leader and Project can be for example defined as follows:

```
    // Person class
    class Person : public PersistentObject
    {
     // Declare the class, its direct predecessor(s) ...
     CLASS(Person);
     PARENTS("PersistentObject");
     // ... and its associated table
     FROM("PERSON");
     // Define member attributes
     dbString(FirstName);
     dbString(LastName);
     dbString(PhoneNr);
     // Primary key is inherited
     // Map other attributes
     MAP_BEGIN
          mapString(FirstName,"#THIS.FIRST_NAME",25);
          mapString(LastName,"#THIS.LAST_NAME",25);
          mapString(PhoneNr,"#THIS.PHONE_NR",15);
     MAP_END;
    public:
     // Initialisation of the object.
     Person() {_FirstName=NULL; _LastName=NULL; _PhoneNr=NULL;};
     // Destruction of the object. _Free() must be called at the
beginning.
     ~Person() {_Free(); StrFree(_FirstName); StrFree(_LastName);
StrFree(_PhoneNr);};
    };
    // Define method returning address of the prototype
    CLASS_PROTOTYPE(Person);
    // Define the prototype UserTable_class
    PROTOTYPE(Person);
```

```
// Programmer class
class Programmer : public Person
{
 // Declare the class, its direct predecessor(s) ...
 CLASS(Programmer);
 PARENTS("Person");
 // ... and its associated table
 FROM("PROGRAMMER");
 // Define member attributes
 dbInt(EmployeeNr);
 dbString(PreferredLanguage);
 // Primary key is inherited
 // Map other attributes
 MAP_BEGIN
      mapInt(EmployeeNr,"#THIS.EMP_NR");
      mapString(PreferredLanguage,"#THIS.LANGUAGE",25);
 MAP_END;
public:
 // Initialisation of the object.
 Programmer() {_EmployeeNr=0; _PreferredLanguage = NULL;};
 // Destruction of the object. _Free() must be called at the
beginning.
 ~Programmer() {_Free(); StrFree(_PreferredLanguage);};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(Programmer);
// Define the prototype UserTable_class
PROTOTYPE(Programmer);


// Leader class
class Leader : public Programmer
{
 // Declare the class, its direct predecessor(s) ...
 CLASS(Leader);
 PARENTS("Programmer");
 // ... and its associated table
 FROM("LEADER");
 // No additional member attributes
 // Primary key is inherited
 // Map other attributes
 MAP_BEGIN
 MAP_END;
public:
 // Initialisation of the object.
 Leader() {};
 // Destruction of the object. _Free() must be called at the
beginning.
 ~Leader() {_Free();};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(Leader);
// Define the prototype UserTable_class
PROTOTYPE(Leader);
```

```
// Project class
class Project : public PersistentObject
{
 // Declare the class, its direct predecessor(s) ...
 CLASS(Project);
 PARENTS("PersistentObject");
 // ... and its associated table
 FROM("PROJECT");
 // Define member attributes
 dbString(Title);
 dbString(Language);
 // Primary key is inherited
 // Map other attributes
 MAP_BEGIN
      mapString(Title,"#THIS.TITLE",15);
      mapString(Language,"#THIS.LANGUAGE",25);
 MAP_END;
public:
 // Initialisation of the object.
 Project() {};
 // Destruction of the object. _Free() must be called at the
beginning.
 ~Project() {_Free();StrFree(_Title); StrFree(_Language);};
};
// Define method returning address of the prototype
CLASS_PROTOTYPE(Project);
// Define the prototype UserTable_class
PROTOTYPE(Project);
```

## 2.4. Retrieving persistent objects from database

The POLiTe library provides two ways of creating persistent objects. The first way is to read an object from the database.

The second way is to first create an object in memory (as a transient instance) and then make it persistent by writing it to the database.

### 2.4.1. Retrieving objects using Query and QueryResult classes

While starting the application no pointer to stored data is known. The only way how to obtain first data from the database is then reading them by using an SQL query. Having at least one object (or its pointer) known, the application can start usual process of spreading activity by traversing from object to object using associations between the instances of the object classes.

While reading first objects from the database, the following steps must be executed. ( Suppose that corresponding object classes are already defined. )

- Create a new Query
- Execute the Query using a class prototype
- Browse the QueryResult to get object values

### 2.4.2. Creating a Query object

Objects of class Query represent queries to the database that search for objects of the specified class. In a query the user can specify a condition which must be satisfied by searched objects. This condition is added (via the AND operator) to the default WHERE clause of the corresponding class. Sorting conditions can be also specified in a query.

A query object is independent on the persistent object classes and can be potentially executed on any of these classes (i.e. descendants of Object, ImmutableObject, DatabaseObject and PersistentObject classes). The database columns used in a query condition must, of course, match the table columns used for the specified class.

The query translates C++ notation used in queries automatically to the SQL notation. In more detail, following fragments are translated:

- Each occurrence of "==" is replaced by "="
- Each occurrence of "!=" is replaced by "<>"
- Each occurrence of "!" is replaced by " NOT "
- Each occurrence of "&&" is replaced by " AND "
- Each occurrence of "||" is replaced by " OR "
- Each occurrence of "ClassName::AttributeName" is replaced by "(EXPR)"

Where EXPR is the SQL expression defined in the attribute mapping. The EXPR has usually form TAB.COL where TAB is the table associated to the class ClassName and EXPR is the expression (usually column name).

A Query class defines four constructors and copy operator.

***Query();***

Constructs a Query without condition.

***Query (const char *const a_where);***

Constructs an unsorted query. The *a_where* parameter sets the query condition.

***Query (const char *const a_where, const char *const a_order_by);***

Constructor for sorted query. The *a_order_by* parameter sets the ORDER BY clause.

***Query (const Query &const X);***

Copy constructor for Query instance. It sets both the WHERE and the ORDER_BY clause.

***Query &Query::Operator =(const Query & X);***

Copy operator on Query class. Both the WHERE and the ORDER_BY clauses are copied.

Existing Queries can be combined into Queries with more complex conditions by using the following methods and operators.

***Query &Not();***

***Query operator !() const;***

Change the query condition to its negation.

***Query &Or(const Query & const Q);***

***Query operator ||(const Query & const Q) const;***

Provide disjunction of two queries. The resulting condition consists of conditions of original queries concatenated with OR.

***Query &And(const Query & const Q);***

***Query operator &&(const Query & const Q) const;***

Provide conjunction of two queries. The resulting condition consists of conditions of original queries concatenated with AND.

WHERE and ORDER BY conditions are stored in the Query instance and can be accessed using following methods (for details see SDR sec. 3.9.5).

***char *Where() const;***

Returns WHERE clause hold by the query.

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

Sets WHERE clause of the query. Where clause of the query is concatenated (through AND operator) with the where clause of the object to which the query is sent.

***char *OrderBy() const;***

Returns the ORDER BY clause hold by the Query.

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

Sets the ORDER BY clause hold by the Query.

Three constant instances of the Query class with the names `ALL, NONE and EQUERY` are defined in POLiTe. These Query objects retrieve contain conditions "0=0". "0<>0" and "" (empty condition). ORDER BY fragments are defined as empty stings.

```
const class Query ALL("0=0");
const class Query NONE("0<>0");
const class Query EQUERY("");
```

***Example 15 - Queries***

```
// create new Query from class Person which selects workers
// whose names start with "P"
Query Q1("Person::LastName like 'P%'");
```

```
// create new Query which selects programmers
// that prefer C++
Query Q2("PROGRAMMER.LANGUAGE = 'C++'");


// change the first Query to find programmers
// that satisfy both conditions
Q1.And(Q2); /* Equivalent to Q1 = Q1 && Q2; */


// the result should be sorted by EmployeeNr
Q1.OrderBy("Programmer::EmployeeNr ASC");
// create a Query that retrieves programmers
// that prefer C++ or Pascal
Query Q3("Programmer::PreferredLanguage == 'Pascal' ");
Q3.Or(Q2); /* Equivalent to Q3 = Q3 || Q2; */
```

### 2.4.3. Executing a query

After constructing a Query instance, the objects can be obtained from the database by executing this query. The *Result<T> *Proto<T>::operator() (const Query &Q, class Connection *DbCon)* method must be called for the corresponding persistent object prototype.

An existing *Query* and *Connection* instances are passed to this operator as arguments. It returns address of new instance of the *Result<T>* class. QueryResult keeps inside all necessary information to access all retrieved objects.

The Query Result is a collection of objects, respective a collection of references to class T.

***Example 16 - Query results***

```
QueryResult *QR = Programmer_class(Q1, DbCon);
```

### 2.4.4. Browsing a QueryResult

A query result can be searched as a collection of items. Each item identify unique resulting object (i.e. an instance of the Object directly, or an object identification in case of descendants of other classes).

Each Result object holds its database connection, the SQL command with which it was created and a pointer to the prototype of the selected class.

To start searching the Result, the *ResultBase::Open()* method must be called first. This causes the stored SQL command to be executed. When the query is executed by above-mentioned operator, the result is opened automatically so the user needs to call it only to re-execute the SQL command ( for example to refresh the Result) or after closing it.

The following methods can be use to browse a QueryResult.

***long Count();***

Returns the number of the items in the collection.

***bool Prev();***

Sets the index to the previous object in the collection. It returns true if the setting was successful, otherwise it returns false.

***bool Next();***

Sets the index to the next object in the collection. It returns true, if the setting was successful, otherwise it returns false.

***bool First();***

Sets the index to the first object in the collection. It returns true, if the setting was successful, otherwise it returns false.

***bool Last();***

Sets the index to the last object in the collection. It returns true, if the setting was successful, otherwise it returns false.

***bool IsOnFirst();***

Returns true, if the index points to the first item of the collection, otherwise it returns false.

***bool IsOnLast();***

Returns true, if the index points to the last item of the collection, otherwise it returns false.

***bool GoToPosition (long Pos);***

Sets the index to the given position.

***long Position();***

Returns the current position in the result.

The *QueryResult* must be closed (by calling *QueryResult::Close()*) after finishing the work with it. When QueryResult is closed, the database cursor inside it is also closed. No operations except *Open()* can be invoked on closed Result.

Use prefix ++ operator for standard iteration through the result. Instances of Result<T> can be directly use as references to objects. The simplest way for manipulating with retrieved objects is:

***Example 17 - Browsing of query results***

```
while (++(*QR)!=DBNULL)
{
...
// process object accessible using (*QR)->
...
};
QR->Close();                        // close result set
```

```
QR->Open();                          // open the result once again
If (QR->GoToPosition(5))             // go to the 5th position
 // process element
...
QR->Close();                         // close result set
```

### 2.4.5. Using database pointers

The purpose of database pointers (also called references) i.e. instances of *Ref<T>* class or their descendants Result<T> is to provide an access to persistent object attributes and methods independently on the fact whether the object is currently in the memory or not.

A database pointer points to one object loaded through a specific database connection and thus it includes database connection identification and object identification too. The same object can be loaded into memory via two different connections, and can be changed independently in the context of each of the connection.

A database pointer can point to objects derived from any of four basic classes. When a specified object is accessed through its database pointer, the pointer is automatically dereferenced, i.e. the object is loaded (if necessary) from the database and placed into the memory.

The Ref<T> class defines operators * and -> that allow the usage of Ref<T> in a similar way as standard pointers in the C++ programming.

### 2.4.6. Accessing object attributes and methods

Having reference to the object. The programmer can easily access its members using operators of dereference.

## 2.5. Manipulating polymorph results

The queries are executed on some particular class. But in fact, result of such query may contain instances of the queried class as well as instances of any derived subclass. The database pointers obtained from the result of the query are immediately correctly retyped. The internal pointer to the prototype is set to the prototype of the appropriate class. This approach ensures, that when the pointer is dereferenced, correct object including all attributes is retrieved.

To achieve this behaviour, the POLiTe library stores the name of the class in the OID_ROOT.CLASS_NAME column at the time the object is created using *BePersistent()* method. If the correct prototype should be set during the query result retrieval, the ClassRegister finds correct prototype according to the stored name.

The `AUTO_VIRTUAL_OBJECT_LOAD` label (see 1.4) must be defined to allow the POLiTe to provide this feature. This label is defined by default.

## 2.6. Creating, updating and deleting persistent objects

### 2.6.1. Creating new objects in database

To create a new persistent object the user must first create its transient variant in memory. Then this object can be made persistent by inserting into the database. Suppose the corresponding subclasses of ImmutableObject, DatabaseObject or PersistentObject classes are already defined and the necessary database tables exist.

- Create a new instances of the desired class in memory in a standard way, by calling the C++ *new* statement. The object attributes are initialised by one of the defined constructors or using access methods. Change the values of object attributes as desired.

```
// create objects ...
Programmer *P = new Programmer();
P->FirstName(...);
P->LastName(...);
...
P->PreferredLanguage(...);
```

- Connect to the database using *Database::Connect(...)* method.

```
Connection *DbCon = NULL;
DbCon = SampleDatabase.Connect("polite","polite");
```

- Make the new objects persistent by calling the virtual method *BePersistent()*.

```
Ref<Programer> PPtr = P->BePersistent(DbCon);
```

This method causes the insertion of the corresponding rows into the database (an INSERT command is automatically constructed and sent to the database).

From this point the object is treated as persistent and all its changes are propagated to the database according to the UpdatingStrategy settings.

The Insert command is not committed and the user must commit it explicitly, for example by calling the *Connection::Commit()* method, that causes to commit all changes made on objects manipulated through this connection.

The *Connection::Commit()* method is called automatically by the library while disconnecting the connection.

### 2.6.2. Modifying objects

After retrieving or creating a persistent object the user can change values of the object attributes. All primitives, which change the content of the instance should call the *MarkAsDirty()* method of that instance.

Persistent objects are updated in two ways depending on the update strategy. If the update strategy is set to *US_Immediately*, an UPDATE command is constructed and sent to the database immediately after any change to this object, respectively each time the *MarkAsDirty()* method is invoked.

If the update strategy is set to *US_OnDemand* all changes must be written to the database explicitly by calling *PersistentObject::Update()* method of the selected instance.

***Example 18 - Modifying persitent objects***

```
PPtr->PreferredLanguage("Java");
// changes preferred language only in memory
// MarkAsDirty() method is called inside
PPtr->Update();
// sends an UPDATE command to the database
```

Each object has its own update strategy. When the object is accessed, the UpdateStrategy is initialised according the reference that was used to retrieve it.

All methods defined by the user that perform changes on the object attributes should mark this object as dirty (by calling *Object::MarkAsDirty()* method). If the object is not marked as dirty, no action is performed by the *Update()* method.

The changes made to the database are not committed and the user must commit it explicitly.

While the object is loaded into memory, other users can possibly change the object values in the database. To force the object to be read from the database again, the virtual method *Object::Refresh()* can be used.

***Example 19 - Refreshing persitent objects***

```
PPtr->Refresh();
```

### 2.6.3. Deleting objects

A persistent object instance can be deleted only from the memory or both from the memory and the database.

The virtual method *Object::Free()* destroys the object in memory only (and removes it from the object buffer). If the object is marked as dirty, it is updated before the copy in the memory is destroyed.

This method succeeds if all tests (for example, if the object is not locked) are passed successfully. If the object is locked in memory it can not be removed and a ObjLibException_*MemoryLock()* exception is thrown.

The virtual method *Object ::Delete()* removes the object from the memory as well as from the database. This method succeeds if all tests (for example, if the object is not locked) are passed successfully.

The object is first deleted from the database. One or more (if necessary) DELETE commands are constructed and sent to the database. Then it is marked as clean and removed from the memory by calling *DatabaseObject ::Free()*.

If the NO_DATABASE_CASCADE_DELETE label is defined, the POLiTe library supposes, that it must delete instances of descendants of the *PersistentObject* piece by piece from all tables going from the bottom to the top of the hierarchy tree. If the label is not defined (it is the default), the POLiTe library supposes, that it is possible to delete the corresponding row only in the table OID_ROOT. This table is associated with the *OidBasedPersistentObject* class. The SQL server will delete the rest of the instance using cascade delete constraints defined on the descendant tables.

***Example 20 - Deleting persitent objects***

```
    PPtr->Free();     // delete object from memory
    PPtr->Delete();   // delete object from memory and from the
database
```

Groups of objects can be updated, or removed from the memory using methods provided by the ObjectBuffer.

***bool UpdateAll();***

Update all changed objects.

***bool UpdateAll(const class Connection *const DbC);***

Update all changed objects retrieved from the given database.

***bool UpdateAll(const class Database *const DB);***

Update all changed objects retrieved from the given connection.

***bool RemoveAll();***

Free all changed objects.

***bool RemoveAll(const class Connection *const DbC);***

Free all changed objects retrieved from the given database.

***bool RemoveAll(const class Database *const DB);***

Free all changed objects retrieved from the given connection.

### 2.6.4. Locking objects in the memory

As mentioned above a mechanism for locking objects in memory is implemented in the POLiTe library. This prevents undesired removing of objects from memory (and from the object buffer). It also speeds-up the access to the object, because standard pointers can be used to access them. Multiple locks can be placed on the object. The object can be removed from memory only when all the locks are released. Following methods are used to manage memory locking.

***virtual class Object *ObjRef::MemoryLock();***

Locks the object in memory.

***virtual bool ObjRef::MemoryUnlock();***

Unlocks the object in memory.

***virtual unsigned int ObjRef::MemoryLocked() const ;***

Returns the number of locks.

***virtual bool ObjRef::RemoveAllMemoryLocks();***

Releases all remaining memory locks on the object.

Some of the actions performed on multiple objects can be called also through the ObjectBuffer class using methods.

***bool RemoveAllMemoryLocks();***

Removes all memory locks on all registered instances.

***bool RemoveAllMemoryLocks(class Database &DB);***

Removes all memory locks on all registered instances loaded from given database.

***bool RemoveAllMemoryLocks(class Connection &DbConn);***

Removes all memory locks on all registered instances loaded from giben conection.

## 2.7. Relations

Associations between objects are represented as instances of the *Relation* class. Relation links together instances of two classes.

The *Relation* class has five subclasses representing one-to-one, one-to-many, many-to-one, many-to-many and chained relations (template classes *OneToOneRelation*, *OneToManyRelation*, *ManyToOneRelation, ManyToManyRelation* and *ChainedRelation*).

The member of the relation is a couple of objects. We call the first member of this couple as the left member and the second one as the right member. Accordingly terms „left side" and „right side" of the relation are used to identify the first or the second class associated together.

The items of the relation, can be inserted, deleted and accessed step-by-step. The items of relations are couples of objects or their references. The changes to the relations are immediately written to the database, they are not buffered.

The relations remember the connection that should be used to execute SQL statements. It can be be changed anytime by method.

***Relation &Relation::Connection(Connection *);***

This method allows to have only one connection instance for all opened connections and change it accordingly before each statement that issues SQL statement to the database. Other way is to have one instance of particular relation for each connection. It is even possible to have more instances of the same relation simultaneously.

### 2.7.1. Creating relations

Object of any new classes derived from ImmutableObject can be linked via relations. The Relation class uses the prototype mechanism to create the appropriate SQL statements.

Each instance of Relation class remembers it's left-side and right-side class templates and corresponding relation name. Optionally, the relation can be created with alternative names for the column names, which represents relation in the database.

The way of representing relations depends on their cardinality. But accessing and manipulating them is independent on it.

#### *2.7.1.1.One to one relation*

In one to one relation each object can have only one object linked with it via this relation. To create an instance of the *OneToOneRelation*, one of the corresponding constructors must be called.

The *OneToOneRelation* object must be properly initialised. Because there exists only one associated left-side object for each right-side object (and vice versa), this link is represented as an additional column(s) in both database tables associated with the participating persistent object classes.

OneToOneRelation<L,R> class provides the following constructor.

```
OneToOneRelation<L,R>::OneToOneRelation(
       const char *a_table_name,
       class Connection *a_database_connection,
       const char *a_left_column_name = NULL,
       const char *a_right_column_name = NULL
       );
```

The first parameter should contain the name of the relation. This name is used for different purposes depending on the cardinality of the relation.

Next parameter represents a connection through which the relation is manipulated.

The last two parameters are optional and specify the names of the corresponding database columns, which refers to the primary key of the left class in the right table and to the primary key of the right class in the left table in the database. If these parameters are omitted they are created automatically as the names of the left table primary keys with the prefix '*ATableName*_L_' and as the names of the right table primary keys with the prefix '*ATableName*_R_'. The database column with those names (given or constructed) must, of course exists in the database - left column names in the right table and vice versa.

***Example 21 - Creating one to one relation***

```
OneToOneRelation<Programmer,Project> Prog_Proj(
    "PROGRAMMER_PROJECT",
    DbCon,
    "PROGRAMMER_OID",
    "PROJECT_OID"
  );
```

This definition says, that each programmer can work on one project and on each project can work only one programmer. Because it defines names of column names that form the association in the database, it supposes that there is defined column PROJECT_OID in the PROGRAMMER table and PROGRAMMER_OID in the PROJECT table.

Method WriteDDL(ofstream &, Database&) can generate script for appropriate changes in the database. In this case, the script looks as follows.

```
    -- One to one relation PROGRAMMER_PROJECT between classes
Programmer and Project
    ALTER TABLE PROGRAMMER ADD(
    PROJECT_OID NUMBER(10),
    CONSTRAINT PROGRAMMER_PROJECT_FK_R FOREIGN KEY(PROJECT_OID)
REFERENCES PROJECT(OID)
    );

    CREATE UNIQUE INDEX PROGRAMMER_PROJECT_RK_R ON
PROGRAMMER(PROJECT_OID);

    ALTER TABLE PROJECT ADD(,
    PROGRAMMER_OID NUMBER(10),
    CONSTRAINT PROGRAMMER_PROJECT_FK_L FOREIGN KEY(PROGRAMMER_OID)
REFERENCES PROGRAMMER(OID)
    );

    CREATE UNIQUE INDEX PROGRAMMER_PROJECT_RK_L ON
PROJECT(PROGRAMMER_OID);
```

#### *2.7.1.2.One to many relation*

In one to many relation each object on the right side can be linked with only one left-side object and each object on the left side can be linked with many right-side objects. To create an instance of the *OneToManyRelation* template, the corresponding constructor must be called. Because there exists only one associated left-side object for each right-side object , this link is represented as an additional column in the database table corresponding to the right-side class.

OneToManyRelation class provides the following constructor.

```
OneToManyRelation<L,R>::OneToManyRelation(
      const char *a_table_name,
      class Connection *a_database_connection,
      const char *a_left_column_name = NULL,
      const char *a_right_column_name = NULL
      );
```

The parameters have similar meaning as parameters of *OneToOneRelation()* constructor. If no column names are specified they are created automatically. The left column names are costructed as names of the corresponding primary keys with the prefix '*ATableName_*'. The right column names are not used anyway, and the parameter is present only for compatibility with other types of relations. The database columns with these names (given or constructed) must, of course exists in the right table in the database.

***Example 22 - Creating one to many relation***

Consider the same association as before, but with different cardinality.

```
OneToManyRelation<Programmer,Project> Prog_Proj(
    "PROGRAMMER_PROJECT",
    DbCon,
    "PROGRAMMER_OID",
    "PROJECT_OID"
  );
```

This time, the needed changes in the database (generated by the WriteDDL method) are

```
    -- One to many relation PROGRAMMER_PROJECT between classes
Programmer and Project
    ALTER TABLE PROJECT ADD(
    PROGRAMMER_OID NUMBER(10),
    CONSTRAINT PROGRAMMER_PROJECT_FK_L FOREIGN KEY(PROGRAMMER_OID)
REFERENCES PROGRAMMER(OID)
    );

    CREATE INDEX PROGRAMMER_PROJECT_RK_L ON PROJECT(PROGRAMMER_OID);
```

#### *2.7.1.3.Many to one relation*

This type of the association is the same as OneToManyRelation, only the sides are swapped. We need one additional column to the left table instead. The script generated by the WriteDDL method will be similar to previous one.

```
    -- Many to one relation PROGRAMMER_PROJECT between classes
Programmer and Project
    ALTER TABLE PROGRAMMER ADD(
    PROJECT_OID NUMBER(10),
    CONSTRAINT PROGRAMMER_PROJECT_FK_R FOREIGN KEY(PROJECT_OID)
REFERENCES PROJECT(OID)
    );

    CREATE INDEX PROGRAMMER_PROJECT_RK_R ON PROGRAMMER(PROJECT_OID);
```

#### *2.7.1.4.Many to many relation*

In many to many relation the number of object linked to each other is not limited. To create an instance of the *ManyToManyRelation*, the corresponding constructor must be called. The *ManyToManyRelation* object must be initialised with the appropriate database tables and column names. Because there can exist many object associated to object on each side, the relation must be implemented as a new database table containing key attributes of both participating classes.

ManyToManyRelation class provides following constructor.

```
ManyToManyRelation<L,R>::ManyToManyRelation(
       const char *a_table_name,
       class Connection *a_database_connection,
       const char *a_left_column_name = NULL,
       const char *a_right_column_name = NULL
       );
```

The parameters have similar meaning as parameters of OneToOneRelation() constructor. The a_table_name parameter must contain the real name of the database table containing the relation.

If no column names are specified, they are constructed automatically as names of corresponding primary keys with the "L_" prefix for the left side and with the "R_" prefix for the right side.

*Example 23 - Creating many to many relation*

Consider the earlier defined classes Lets define a many to many relation *Prog_Proj* where each programmer can work on many projects and each project can be solved by many programmers.

The WriteDDL method provides following script for representing the relation:

```
    -- Many to many relation PROGRAMMER_PROJECT between classes
Programmer and Project
    CREATE TABLE PROGRAMMER_PROJECT(
    PROGRAMMER_OID NUMBER(10) NOT NULL,
    PROJECT_OID NUMBER(10) NOT NULL,
    CONSTRAINT PROGRAMMER_PROJECT_PK PRIMARY
KEY(PROGRAMMER_OID,PROJECT_OID),
    CONSTRAINT PROGRAMMER_PROJECT_FK_L FOREIGN KEY(PROGRAMMER_OID)
REFERENCES PROGRAMMER(OID) ON DELETE CASCADE,
    CONSTRAINT PROGRAMMER_PROJECT_FK_R FOREIGN KEY(PROJECT_OID)
REFERENCES PROJECT(OID) ON DELETE CASCADE
    );

    CREATE UNIQUE INDEX PROGRAMMER_PROJECT_RK ON
PROGRAMMER_PROJECT(PROJECT_OID,PROGRAMMER_OID);
```

#### 2.7.1.5. Chained relation

Chained relations are build from other relations. All chained relations are considered as computed and read-only. It is not possible to associate couples of objects through this type of relations.

### 2.7.2. Relation Indexes and Integrity Constraints

Methods WriteDLL defined for different types of relations generate not only needed tables and columns, but also indexes. The following indexes should be created in the database to speed–up the access to the related objects and the other relation manipulations:

- an UNIQUE INDEX on foreign keys for one to one and one to many relations
- an UNIQUE INDEX on set of all columns in the table representing many to many relations
- INDEX (not UNIQUE) on the set of left columns and on the set of right columns in the table representing many to many relations

The following integrity constraints should be defined in the database to enhance the relation integrity checking:

- a NOT NULL constraint for each column in the many to many relation table.
- a PRIMARY KEY constraint on set of all column in the many to many relation table.

  This constraint creates the UNIQUE INDEX on this set of columns.
- a FOREIGN KEY with the ON DELETE CASCADE constraint associating the many to many relation table with the primary keys of the tables associated with the left as well as with the right classes.

### 2.7.3. Browsing relations

Obtaining of all instances associated with some concrete instance is similar to obtaining of all instances of some class, which satisfies some condition about its contents.

The objects in the relation can be searched using a *Result<L>* and *Result<R>* classes. The *Relation* class defines several functions that return a list of object linked to the selected instance as a Result. Some of these methods allow adding an additional condition, which the selected objects must satisfy. Two main methods are

***virtual Result<L> Left(Relation& x, const QueRefProto &right)***

***virtual Result<R> Right(Relation& x, const QueRefProto &left)***

These methods return all left-side objects that are associated with any right-side object that satisfies the "right" condition, respectively all right-side objects associated with any instance that satisfies the "left" condition. As a condition can be used a Query instance, an Object instance or a Proto instance. Each object instance represents the condition, which is satisfied by the instance only. Each prototype defines condition, which is satisfied by any instance of given class and its descendants.

***virtual Result<L> Relation::LGetAll(const QueRefProto &qL, const QueRefProto &qR)***

***virtual Result<L> Relation::RGetAll(const QueRefProto &qL, const QueRefProto &qR)***

These methods are more general. Return all left-side objects satisfying "left" condition that are associated with any right-side object that satisfies the "right" condition and vice versa.

***virtual bool *Relation::ExistsCouple(const ObjRef &left, const ObjRef &right);***

It returns true, iff the couple of objects is related via this relation.

Every new class can all its relations provide access methods returning the query result containing all associated instances. Another way is to return only the relation. The user then can use other methods to obtain associated instances from the database.

### 2.7.4. Inserting and deleting relations between objects

Couples of objects (represented by DatabasePointers) can be inserted to and deleted from a relation.

The *Relation* class defines method that allows creating relations (links) between objects. A couple of objects can be inserted into relation using method

***virtual bool *Relation::InsertCouple(const ObjRef &left, const ObjRef &right);***

The *Relation* class defines also method that allows deleting couples of object from a relation. A single couple of objects can be deleted from a relation using

***virtual bool DeleteCouple(DatabasePointer &left, DatabasePointer &right);***

Following two methods delete all associations between given right-side object and any associated left-side object

***virtual bool Relation::LDeleteAll(class RefBase &right);***

***virtual bool Relation::LDeleteAll(class Object &right);***

Following two methods have similar functionality on opposing sides of relation.

***virtual bool Relation::RDeleteAll(class RefBase &left);***

***virtual bool Relation::RDeleteAll(class Object &left);***

## 2.8. Using Database Pointers for Referencing Objects

The relations allow the user to traverse from one instance of some class to all related instances. It is necessary to use this approach in case of many to many relations, but sometimes the object is associated only with one other object of the specified class.

Let suppose the situation, where each project has only one leder.

In the standard C++ approach the pointer to Leader class is added to the definition of the Project class.

It is possible to define an one-to-many relation between these two classes in the POLiTe application, but yet another possibility, similar to standard C++ approach, is applicable. In this approach the usual pointer to Leader class is replaced by the Ref<Leader> reference. Instead of the memory location of the boss of the given worker it is necessary to remember the OID of the boss. In addition to this attribute two access methods manipulating this remembered boss OID within the Workers class instances are required. Those two methods can be written to manipulate with the DatabasePointers or even with the regular pointers as it is shown in the following example.

***Example 24 - Using Database Pointers for Direct Traversing***

```
// standard C++ approach          // POLiTe approach
class Project{                    class Project{
...                               ...
class Leader *ProjLeader;         dbPtr(Leader,ProjLeader);
                                  // OID of the boss
...                               ...
};                                );
```

The Project class has then two additional access methods

***Ref<Leader> Project::ProjLeader()***

***void Project::ProjLeader(Ref<Leader> &)***

## 2.9. Exceptions, types and recovery

During execution of POLiTe based application, some exceptions may appear due to incorrect work of database server, network layers of software, incorrect use of POLiTe services, insufficient amount of resources (memory), etc. These exceptions are represented by classes, derived from common predecessor ObjLibException.

These exceptions are described below. They are the successors of the ObjLibException class in the class hierarchy (see the figure).

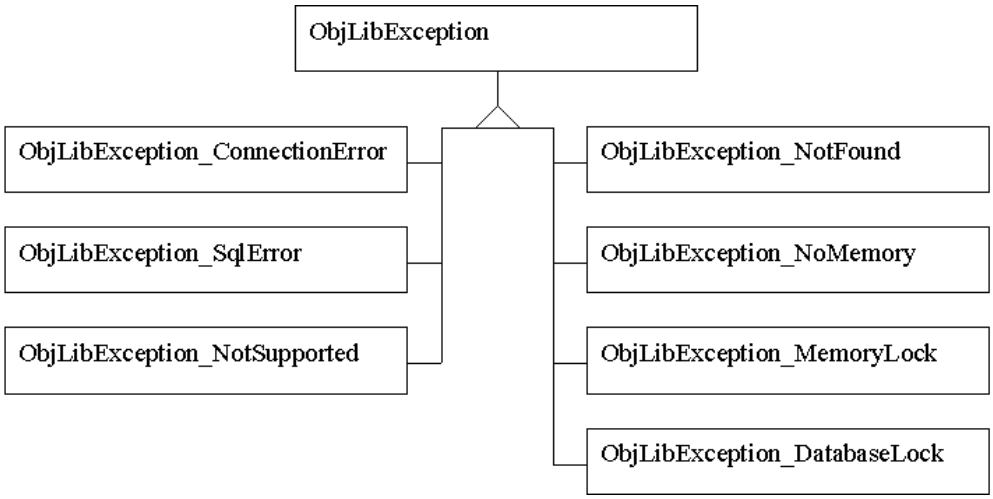


*Figure 2 - Class hierarchy for exceptions*

- ObjLibException

  This is an exception, which represents all exceptions thrown by the POLiTe. The user can catch this exception if he wants to handle all exceptions in the same way.

***Example 25 - Catching of ObjLibException exception***

```
class OracleDatabase SampleDatabase("");
  //default oracle database

  Connection *DbCon = NULL;
  printf("Connecting to the database ...\n");
  try {
  DbCon = SampleDatabase.Connect(”polite”,”polite”);
  printf("... connected\n");
  }
  catch (ObjLibException &X) {
  printf("... NOT connected\n");
  throw;
  };
```

- ObjLibException_NotSupported

  This exception is thrown, whenever POLiTe tries to use any feature, which is not supported by the used SQL engine.

- ObjLibException_ConnectionError

  This exception is thrown, if POLiTe can't communicate with the database server.

- ObjLibException_SqlError

  This exception is thrown, if the database server does not recognise SQL command due to either syntax error, or semantic error (table not exists, etc.).

***Example 26 - Catching of ObjLibException_SqlError exception***

```
class OracleDatabase SampleDatabase("");
  //default oracle database

  Connection *DbCon = SampleDatabase.Connect(”polite”,”polite”);
  char *Dynamically_Created_SQL_Statement;
  ...
try {
  (*DbCon) << Dynamically_Created_SQL_Statement;
  }
  catch (ObjLibException_SqlError &X) {
  printf("... wrong SQL statement executed\n");
  throw
  };
```

- ObjLibException_DatabaseLock

  This exception is thrown, if the specified object can't be changed in the database because the table row, representing object is locked by another connection.

- ObjLibException_MemoryLock

  This exception is thrown, if the specified object can't be freed, because it is currently locked on a fixed address of memory.

- ObjLibException_NoMemory

  This exception is thrown, if there is not enough memory to complete operation.

- ObjLibException_NotFound

  This exception is thrown, if the specified data were not found in the database. The library maintains it itself in the Result<T>::Next() method.

## 2.10. Internal libraries

Internal libraries provide some useful services used internally by the POLiTe itself. Some of those services can be used by POLiTe users.

POLiTe provides set of ordinal functions (not a methods of any class), which manipulates with the char* variables.

***int StrLen(const char * const src)***

Counts number of chars in src, returns 0 if src == NULL

***char *StrCpy(char * (&dst), const char * const src)***

Copies the source *src* into destination *dst*. Frees previously allocated memory for the *dst.* Allocates the memory for new *dst*

***char *StrExp(char * dst, const char * src, int len)***

Copies the source *src* into the destination *dst.* The buffer for the result must already exists, because it is not allocated inside this function. The maximal number of the copied character is *len*. The copy of the source is terminated by \0 character, so the buffer must be at least *len+1* characters long.

***int StrCmp(const char * const dst, const char * const src)***

Compares the source *src* with the destination *dst.* Returns zero if the strings are equal and non–zero if the strings differ each from the other. Handles NULL values of parameters correctly.

***int StrCmpUp(const char * const dst, const char * const src)***

Compares the source *src* with the destination *dst* in upper-case letters.

***char *StrFree(char * (&dst))***

Frees a dynamically allocated string. Sets *dst* to NULL. If *dst* is NULL does nothing.

***char *StrCat(char * (&dst), const char * const src)***

Concatenates *dst* and *src* in the given order. Method frees previously allocated memory for *dst* and allocates memory for the result.

***char *StrCat(char * (&dst), const int n, const char * const src, ...)***

Concatenates the strings *src_1*, *src_2*, ... , *src_n* to the *dst* in the given order. Method frees a previously allocated memory for *dst* and allocates a memory for the result.

***char *StrAnd(char * (&dst), const char * const src)***

Returns a string in form "(*dst*) AND (*src*)". Method frees the previously allocated memory for *dst* and allocates a memory for the result. If one of the parameters is NULL, the other one is returned.

***char *StrOr(char * (&dst), const char * const src)***

Returns a string in form "(*dst*) OR (*src*)". Method frees the previously allocated memory for *dst* and allocates a memory for the result. If one of the parameters is NULL, the other one is returned.

***char *StrNot(char * (&dst))***

Returns a string in form "NOT(dst)". Method frees the previously allocated memory for *dst* and allocates a memory for the result.

***char *StrClause(char * (&dst), const char * const prf);***

Returns a string in form "*prf dst* " if *dst* is not empty. If *dst* is the empty string, the function returns NULL. Method frees the previously allocated memory for *dst* and allocates a memory for the result.

***void StrSwap(const char * (&s1), const char * (&s2));***

Swaps two strings *s1* and *s2*.

***char *StrSplit(char * (&src), const char delim, char * (&dst));***

Splits the string *src* in form "*dst delim rest_of_string*". This method returns *dst* in the separate string and the *rest_of_string* in original variable *src*. Method frees the previously allocated memory for its parameters and allocates a memory for the results. If no delimiter is present in the original string, *dst* returns the whole string and *src* is set to NULL. The leading and trailing white characters are removed from the resulting *dst* string. Delimiters inside quotas or double–quotas are ignored.

***char *StrPrefix(char * (&src), const char delim, char * (&dst));***

This function is similar to the StrSplit() function. But if no delimiter is found in the string *src*, it returns *dst* == NULL, meanwhile StrSplit() returns *src* == NULL

***char *StrPrefixCut(char * (&src), const char delim);***

This function is similar to the StrPrefix() function, but doesn't return the prefix. The prefix is cut off and forgotten.

***char *LongToStr(const long n);***

Returns long number *n* as a char* and allocates space for the result.

```
char *StrMergeLists (
char * (&dst),
char * (&list1), char * (&list2),
const char lists_delimiter,
const char sep_begin,
const char sep_values
const char sep_middle,
const char sep_end
);
```

```
char *StrMergeLists (
char * (&dst),
char * (&list1), char * (&list2),
const char lists_delimiter, const char prefix_delimiter,
const char sep_begin,
const char sep_values
const char sep_couples,
const char sep_end
);
```

These two functions provide most complicated string manipulation in the library. They are used to merge two lists of values, separated by the *lists_delimiter* character and return result in the *dst* parameter and also as return value. It is supposed that both lists contain the same number of elements. If the number of values is different, only the less number of values is processed. Processing is done in the following way.

1. Values are taken one by one from both lists.

2. If the *prefix_delimiter* is specified (second variant) values are searched for prefix and existing prefixes are removed.
3. Couple of values from both lists are put together and separated by the *sep_values* string.
4. Couples are separated using the *sep_couples* string.
5. Before the first couple is put the *sep_begin* string.
6. After the last couple is put the *sep_end* string.
7. The rests of lists are emptied.

***Example 27 - Internal libraries - StrMergeLists() function***

```
char *list1 = StrCpy(list1 = NULL, "T.A, T.B, T.C, T.D");
//dynamically allocated list of columns in the table
char *list2 = StrCpy(list1 = NULL, "1, 2, 3, 4");
//dynamically allocated list of values
char *result = NULL;
StrMergeLists(
  result,list1,list2,  // destination string and source lists
  ',',           // values separated by commas
  '.',           // prefixes separated by dots
  "SET ",
  " = ",
  ", ",
  NULL
  );
//result == "SET A = 1, B = 2, C = 3, D = 4"
//           ^^^^ ^^^ ^^

char *list1 = Strcpy(list1 = NULL, "T.A, T.B, T.C, T.D");
//dynamically allocated list of columns in the table
char *list2 = Strcpy(list1 = NULL, "1, 2, 3, 4");
//dynamically allocated list of values
char *result = NULL;
StrMergeLists(
  result,list1,list2,  // destination string and source lists
  ',',           // values separated by commas, no prefixes
  "WHERE (",
  " = ",
  ") AND (",
  ")"
  );
//result == "WHERE (T.A = 1) AND (T.B = 2) AND (T.C = 3) AND
(T.D = 4)"
//           ^^^^^^^   ^^^ ^^^^^^^
```

***char *AliasR(int i);***

Returns alias of i-th relation in form "$R_i" and allocates space for the result.

***DLL_External char *AliasT(int i, int j);***

Returns alias of *j*-th table of *i*-th PersistentObject in form "$T_i_j", allocates space for the result.

***char *AliasQ(int i);***

Returns alias prefix of table aliases of tables for *i*-th PersistentObject in form "$T_i_". Allocates space for the result.

***char *StrAddPrefix(char * &dst, const char * const src, const char * const delim);***

Adds prefix *src* to string *dst*. Returns result in form *src delim dst*

***char * RealiasR(char * &fragment, int begin, int end, int incr);***

Increases the numbers of relation aliases beginning with *i*-th and ending by *j*-th by *incr*

***char * RealiasQ(char * &fragment, int begin, int end, int incr);***

Increases the numbers of query aliases beginning with *i*-th and ending with *j*-th by *incr*

```
bool StrReplace(
  char * &dst, const char * const oldstr, const char * const newstr,
   bool lwb = false, bool rwb = false
  );
```

Replaces firts occurrence of *oldstr* in *dst* with *newstr*. Returns true if succeeds. Last two parameters control, if the pattern should begin (end) at the word boundary. If they are set to true, the word in pattern must not continue over boundaries of its occurrence.

```
char * StrReplaceAll(
  char * &dst, const char * const oldstr, const char * const newstr,
   bool lwb = false, bool rwb = false
  );
```

Replaces all occurrences of *oldstr* in *dst* with *newstr*. Returns *dst*. Last two parameters control, if the pattern should begin (end) at the word boundary. If they are set to true, the word in pattern must not continue over boundaries of its occurrence.

***char *StrAndOnly(char * (&dst), const char * const src);***

Returns "dst AND src" without parameters enclosed in round brackets. Frees previously allocated memory for *dst*. Allocates memory for the result.

***char *StrEncode(char * (&dst), const char * const  src);***

***char *StrEncode(char * (&src));***

Encodes string by doubling all backslashes and apostrophes inside it. Encloses result in apostrophes. Allocates new space for the result. It is used to encode some comma-separated lists to be able to find delimiters.

***char *StrDecode(char * (&dst), const char * const src);***

***char *StrDecode(char * (&src));***

Takes encoded string and return original one