DAS
User Manual

Date:	18/02/2014	**Issue:**	0.5
Reference:	CIWS-OATS-TN-002		
Custodian	Stefano Sartor		

Prepared by:		**Date:**	**Signature:**
List of Core authors	Stefano Sartor Marco Frailis Andrea Zacchei		
Contributors:		**Date:**	**Signature:**
List of All contributors			
Approved by:		**Date:**	**Signature:**

Contents

1 Introduction

The Data Access System (DAS) is a software for storing, retrieving and querying data and meta-data acquired from instrument workstations or other elaboration steps. The DAS provides a powerful Data Definition Language (DDL) for describing a custom data model throught different ddl-types. Each ddl-type can contain a meta-data section, a binary data section, which can describe a binary table or an image, and a set of relation with other ddl-types. Through the DAS API the user can create new ddl-objects, populate its meta-data, data and then store the resulting object in the DAS system. Futhermore, the user can retrive a set of a ddl-objects, matching a filter specified as a query string, and then update the objects content.

1.1 Purpose of the document

The aim of this document is:

- provide a complete guide to installing the DAS system on the CIWSdev machine;
- provide a comprehensive guide to configuring the DAS system through the configuration files;
- describe the key concepts of how the DAS handles the ddl-objects persistency
- describe the foundametal API funtcions and methods.

1.2 Glossary

API	Application Programming Interface
CIWS	Customizable Instrument Workstation Software
CIWSdev	CIWS development environment
VM	Virtual Machine
XML	eXtensible Markup Language
json	JavaScript Object Notation
DAS	Data Access System
DDL	Data Definition Language
ddl-type	DDL data structure
ddl-object	instance of a particular ddl-type

2 Install

This chapter will help you to set-up the CIWSdev VM in order to install the DAS system.

2.1 Prerequisites

First of all, you need to install this packages from yum packet manager:

```
MySQL-python
python-ordereddict
```

as root, you can simply type

```
yum install MySQL-python python-ordereddict
```

2.2 odb compiler

The odb compiler is a program used internally by the DAS system during configuration steps. You can find the rpm package at

```
http://www.codesynthesis.com/download/odb/2.3/odb-2.3.0-1.i686.rpm
```

To install it type, as root

```
rpm --install odb-2.3.0-1.i686.rpm
```

2.3 Additional libraries

In order to work, the DAS system needs a few libraries, presented in the following sub-sections. For your convenience we provide the ciwsprod_install.sh script which will:

- add the needed environment variables to the user profile file
- download the source packages
- build the libraries
- install the libraries in the given path
- to run the script just type

```
ciwsprod_install.sh <lib-path>
```

where <lib-path> is the absolute path where the libs will be installed. If you choose to use this script you can skip the next sections, until section 2.4.

2.3.1 Boost

You need a version of boost libraries at least 1.47 which is newer then the one provided by the packet manager. Following this commands you will install the boost 1.54 libraries

```
wget
http://sourceforge.net/projects/boost/files/boost/1.54.0/boost_1_5
4_0.tar.bz2/download
tar jxvf boost_1_54_0.tar.bz2
cd boost_1_54_0
./bootstrap.sh --prefix=<lib-path>/boost_1_54
./b2
./b2 install
```

2.3.2 mySQL connector

The python wrapper needs mySQL library compiled with a specific MYSQL_UNIX_ADDR option.

```
wget http://dev.mysql.com/get/Downloads/Connector-C/mysql-
connector-c-6.1.2-src.tar.gz
tar zxvf mysql-connector-c-6.1.2-src.tar.gz
cd mysql-connector-c-6.1.2-src
cmake -G "Unix Makefiles" -DCMAKE_INSTALL_PREFIX="<lib-
path>/mysqlclient" -DMYSQL_UNIX_ADDR="/var/run/mysqld/mysqld.sock"
make
make install
cd ..
```

2.3.3 odb libraries

We need 3 libraries from odb suite: odb common, odb-boost and odb-mysql. Please note that you may need to configure your environment variables, such as CPLUS_INCLUDE_PATH , LIBRARY_PATH and LD_LIBRARY_PATH in order to make boost and mySQL libs available to odb and DAS system.

```
wget http://www.codesynthesis.com/download/odb/2.3/libodb-
2.3.0.tar.bz2
tar jxvf libodb-2.3.0.tar.bz2
cd libodb-2.3.0
./configure --prefix="<lib-path>/odb"
make
make install
cd ..

wget http://www.codesynthesis.com/download/odb/2.3/libodb-mysql-
2.3.0.tar.bz2
tar jxvf libodb-mysql-2.3.0.tar.bz2
cd libodb-mysql-2.3.0
./configure --prefix="<lib-path>/odb"
make
make install
cd ..

wget http://www.codesynthesis.com/download/odb/2.3/libodb-boost-
2.3.0.tar.bz2
tar jxvf libodb-boost-2.3.0.tar.bz2
cd libodb-boost-2.3.0
./configure --prefix="<lib-path>/odb"
make
make install
cd ..
```

2.3.4 Blitz++

Blitz++ is a C++ class library for scientific computing which provides performance on par with Fortran 77/90. It uses template techniques to achieve high performance. We use Blitz++ as a back-end of the das::Array template class.

```
wget http://sourceforge.net/projects/blitz/files/blitz/Blitz%2B%2B
%200.10/blitz-0.10.tar.gz/download -O blitz-0.10.tar.gz
tar zxvf blitz-0.10.tar.gz
cd blitz-0.10
./configure --prefix="<lib-dir>/blitz"
make lib
make install
cd ..
```

2.3.5 SWIG

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. We use SWIG to provide the python wrapper.

```
wget http://prdownloads.sourceforge.net/swig/swig-2.0.11.tar.gz
tar zxvf swig-2.0.11.tar.gz
cd swig-2.0.11
./configure --prefix="<lib-path>/swig"
make
make install
```

2.4 Installing the DAS system

First of all, you need to set two cmake environment variables in order to allow cmake to find boost and odb. Assuming you've installed boost and odb in /opt and you also want to install das under /opt you can follow this example. If you used the ciwsprod_install.sh to install the libraries, you should already have these variables set.

```
export
CMAKE_INCLUDE_PATH=/opt/boost_1_54/include:/opt/odb/include:/opt/d
as/include

export
CMAKE_LIBRARY_PATH=/opt/boost_1_54/lib:/opt/odb/lib:/opt/das/lib
```

If you used the ciwsprod_install.sh and you still don't have those environment variables set, you can load them from the DAS profile:

```
. ~/.das/profile
```

2.4.1 Build das

The das system needs 3 configuration files in order to work:

- $DAS_ROOT/configure/config.json
- $DAS_ROOT/ddl/ddl.xml
- ~/.das/access.json

You can find the related documentation in the chapter 3.

When you've edit the configuration files, you need to run the configuration step, allowing cmake to

find the needed libraries and the das system to validate the files edited by the user (DDL files and config.json). Furthermore you need to specify the installation directory if differs from the system one (/usr/local). Again, assuming you want to install the das libraries under /opt directory, you can simply type

```
cmake -DCMAKE_INSTALL_PREFIX=/opt/das .
```

Note that the current building system allows only in-tree building. That is to say that you must run the cmake command inside the das building root directory.

If the configuration was successfully done, you can build the library executing the make command.

```
make
```

Now, you need to instantiate the schemas on the database. Before running the following target, you need to create in the back-end DBMS a database named "unit_test" and grant to the user referenced in the ~/.das/access.json config file the db-manager priviledges. If everything if properly set, the unit tests should build and run without any error.

```
make test
```

2.4.2 make targets

We provide different make targets in order to perform different operations on the system.

Install

Install the header files and the library in CMAKE_INSTALL_PREFIX directory, if provided on configuration time, in /usr/local otherwise. Note that you may need root privileges.

```
make install
```

Schema

Generate .sql sources for the database referred by the alias property in the config.json file in build/db/<alias> directory.

```
make schema-<alias>
```

DB-install

Instantiate the schemas in the database referred by the alias property in the config.json file.

```
make db-<alias>
```

Shared lib

Compile the library as shared object.

```
make DAS_SO
```

Static lib

Compile the library as static object.

```
make DAS_A
```

cron job

The Raw data engine described in section 3.1.1 needs a garbage collector in order to delete out-dated data files (section 3.1.2). To set, or reset the crontab job schedule, run this target

```
make cron_job-<alias>
```

Unit tests

Compile unit tests, install the unit_test schema in the DBMS, compile and run th e unit tests.

```
make test
```

Documentation

If doxygen is installed in the system, an additional target for make is created in order to generate the code documentation

```
make doc
```

3 Configuring DAS system

In order to configure and build the DAS system you need to edit some configuration files:

- config.json : is located in the configure directory under the DAS source directory and contains the basic information about the database, the way the data is persisted, and the reference of the ddl files.
- access.json : is located in the .das directory in the $HOME of each user who runs the das library. This file contains the credentials for the databases which the user has access.
- ddl.xml : the files located in the ddl directory under the DAS source directory, in XML format which define the data model.

3.1 config.json

The configuration file contains an array of objects. Each one contains a configuration for a different database. The file is parsed and validated during the configuration step (cmake command) through the json-schema file resources/config_schema.json. Note that each change to this file must be followed by a reconfiguration and consequet build of the DAS system in order to make them active.

```
[
  {
    "host"          : "localhost",
    "port"          : 3306,
    "db_type"       : "mysql",
    "mysql_socket"  : "/var/lib/mysql/mysql.sock",
```

```
    "alias"          : "test_level1",
    "db_name"        : "test_level1",
    "ddl"            : "ddl_level1_types.xml",
    "storage_engine": {
      "name"           : "Raw",
      "root_dir"       : "/mnt/data/test_level1/",
      "default_path"   : "%F/%t/%n_%v/",
      "custom_path"    : "%s/%B/%n/%v",
      "temp_path"      : "/mnt/data/temp/%$LOGNAME$",
      "unref_data_expiration_time" : 864000
    }
  },
  {
    "host"           : "localhost",
    "port"           : 3306,
    "db_type"        : "mysql",
    "mysql_socket"   : "/var/lib/mysql/mysql.sock",
    "alias"          : "test_level2",
    "db_name"        : "test_level2",
    "ddl"            : "ddl_test_types.xml",
    "storage_engine": {
      "name"           : "Raw",
      "root_dir"       : "/mnt/DBs/data/test_level2/",
      "default_path"   : "%F/%t/%n/%v/",
      "custom_path"    : "%s/%B/%n/%v",
      "temp_path"       : "/mnt/DBs/data/temp/%$LOGNAME$",
      "unref_data_expiration_time" : 60
    }
  }
]
```

Each of the properties shown in this example are mandatory:

- host : the hostname or ip address where the database is located
- port : the listening port of the database
- db_type : the database vendor, currently only mysql is supported
- mysql_socket : the location of the mysql pipe file to use when connecting to local MySQL server
- alias : human readable name to use in the code to refer to this database
- db_name : the name of the database to access for storing meta-data
- ddl : the relative path of the file that contains the DDL for this database
- storage_engine : this object contains the configuration of the data storage engine. Except for name, all other properties change for each engine type. Please refer to the storage engine specific documentation in the following sections. Note that currently we provide only the Raw storage engine.

3.1.1 Raw Storage engine

This storage engine creates one file per column if the data type is binary table, and one file per image otherwise. when the user updates the data, for example a column, a new file is created containing the updated data while the old one is kept allowing other sessions (users) which refer to out-date

data to keep reading that data. Once in awhile, depending on the crontab configuration, the garbage collector runs and remove the obsolete files.

this properties must be provided in the config.json file:

- name : name of the storage engine, the value "Raw" identifies this one.
- root_dir : absolute path to the data. each data file will be stored in a sub-directory of this path
- default_path : expression for the default relative path for the data. The resolved expression appended to the root_path forms the complete path for the data file. The possible token will be analysed in the next session. Note that the tokens '%n' and '%v' are mandatory, as say they have to appear somewhere in the expression.
- custom_path : expression for the custom relative path for the data. Like the default_path expression, the resolved expression appended to the root_path forms the complete path for the data file. The token '%s' is resolved run-time with the string provided by the user as argument to the persist methods. Please note that the tokens '%n' and '%v' are still mandatory. If the token '%s' doesn't appear in the expression the possible string argument provided in the persist methods will be ignored.
- temp_path : this expression represent the absolute path for the temporary data, i.e. the data not yet persisted in a transaction. This data file will be eventually moved in the final path during the transaction commit of the owning object. Note that if the temp_path and root_dir are mounted in the same file-system volume the move operation does not involve any copy.
- unref_data_expiration_time : this parameter refers to the garbage collector. The out-dated data older then unref_data_expiration_time seconds will be deleted next time the garbage collector runs.

path tokens

the default_path, custom_path and temp_path expressions are regular unix paths plus some special tokens escaped by the character '%'.

Time related tokens

token	replaced by	example
%a	Abbreviated weekday name*	Thu
%A	Full weekday name*	Thursday
%b	Abbreviated month name*	Nov
%B	Full month name*	November
%c	Date and time representation*	Thu_Aug_23_14-55-02_2001
%C	Year divided by 100 and truncated to integer (00-99)	20
%d	Day of the month, zero-padded (01-31)	23
%F	Short YYYY-MM-DD date, equivalent to Y-m-d	2001-08-23
%g	Week-based year, last two digits (00-99)	01
%G	Week-based year	2001
%H	Hour in 24h format (00-23)	14
%I	Hour in 12h format (01-12)	02
%j	Day of the year (001-366)	235
%m	Month as a decimal number (01-12)	08
%M	Minute (00-59)	55
%p	AM or PM designation	PM
%r	12-hour clock time *	02-55-02_pm

%R	24-hour HH:MM time, equivalent to H-M	14-55
%S	Second (00-61)	02
%T	ISO 8601 time format (HH:MM:SS), equivalent to H-M-S	14-55-02
%u	ISO 8601 weekday as number with Monday as 1 (1-7)	4
%U	Week number with the first Sunday as the first day of week one (00-53)	33
%V	ISO 8601 week number (00-53)	34
%w	Weekday as a decimal number with Sunday as 0 (0-6)	4
%W	Week number with the first Monday as the first day of week one (00-53)	34
%X	Time representation *	14-55-02
%y	Year, last two digits (00-99)	01
%Y	Year	2001
%z	ISO 8601 offset from UTC in timezone (1 minute=1, 1 hour=100). If timezone cannot be determined, no characters	+100
%Z	Timezone name or abbreviation. If timezone cannot be determined, no characters *	CDT

The specifiers marked with an asterisk (*) are locale-dependent.

Object related tokens

token	replaced by	example
%t	type name	measure
%n	object name	meas_001
%v	version of the object	1

Other tokens

The '%s' token can be used in the temp_path expression and represents the custom string provided by the user as the persist methods argument.

Environment variables may by used as tokens. You just have to surround the variable name with '$'.

```
some/path/%$ENV_VARIABLE$some/other/path
```

3.1.2 Garbage Collector

In order to allow concurrent data access, the Raw Storage Engine does not clear immediately out-date data files. They are cleared when the garbage collector runs and the modification time plus the unref_data_expiration_time parameter is less then the current time. Note that the current time is retrieved from the machine which runs the garbage collector, hence this machine must be kept synchronised (e.g. using a common ntp server) with the machine serving the data file-system. You can schedule the garbage collector modifing the cmake variable CRON_SCHED_<alias> stored in che cmake-cache and then update the crontab table running the cron_job-<alias> make target.

3.2 access.json

This file must exists in the $HOME/.das/ directory of the user and contains the credentials to access the databases. This file is read run-time therefore changes on them may be performed without any further configuration.

```
[
  {
    "alias"     : "test_level1",
    "user"      : "foo",
    "password"  : "bar"
  },
  {
    "alias"     : "test_level2",
    "user"      : "foo",
    "password"  : "secret"
  }
]
```

- alias : the database alias that match the one specified in the config.json file.
- user : the username of the database.
- password : the password for that database.

4 Managed Object Model

The way which DAS manages persistent objects is inspired by some concepts in the JPA (Java Persistent API) standard. Objects in a managed context are automatically synchronized with their persisted counterpart.

4.1 Definitions

- Ddl-object: object, that is, instance of a c++ class, that maps a specified DDL data type.
- pu-instance: (Persistence Unit instance) couple of database and and shared directory fully managed by the system. Each pu-instance is uniquely identified by a string set in the "db-alias" option of the das configuration file.
- pm-object: (Persistence Manager object) instance of the class Database that handles connections, transactions, I/O and other operations on a specific pu-instance. Multiple pm-objects referring the same pu-instance can be active (in the same scope) at the same time.
- session: block of code where ddl-objects can be synchronized with a pm-object. A session, by default, starts with the creation of a transaction and ends on the commit of the latter.
- persistent counterpart: data stored in a database and shared file system representing a ddl-object data structure(s).

4.2 Dll-object states

- new: a transient ddl-object without persistent counterpart. A fresh created object is in the new state and it can pass to attached state trough the Database::persist() method.
- attached: a ddl-object with a persistent counterpart managed by a database instance (pm-

object) in a session.

- detached: an object with a persistent counterpart stored in a pu-instance but not attached to any session. Attached objects automatically become detached when the session ends; as say the transaction is committed/rolled-back for default session, the method Database::end_session() is called for extended sessions. A detached ddl-object can be later attached to the owner pu-instance session through the Database::attach() method. This means that the new and the old database instance must refer the same pu-instance.

4.3 DDL objects

A ddl-object newly created is in the new state. You can freely modify and delete it without involve any database operations.

```
shared_ptr<measure> m = measure::create("measure_name",
"db_alias");
m->run_id(12345);
m.reset();
```

Once you want to make a ddl-object persistent, you first have to create a pm-object, then create a transaction and finally you can call the method persist passing the ddl-object as argument. When you have finished persisting the objects, just call the commit method on the transaction object.

```
shared_ptr<measure> m1 = measure::create("measure1",
"test_level1");
shared_ptr<measure> m2 = measure::create("measure2",
"test_level1");

shared_ptr<Database> db = Database::create("test_level1");

Transaction t(db->begin());
db->persist(m1);
db->persist(m2);
t.commit();
```

If your application spend most of the time elaborating data with no interaction with pm-objects until the the very end of the program, you may find convenient load from a pm-object the needed ddl-objects, then delete the pm-object allowing the system to release resources, do your computation, and finally update the persistent objects attaching the modified ddl-objects to a new pm-object which refers the same pu-instance as the first one. You must be aware, however, that the persistent counterpart of the objects you want to attach may not be the same ones you loaded before, because of the concurrent access to the system. If this is the case, attaching those objects will end up losing the updates made by the other(s) user(s) on them.

```
shared_ptr<Database> db = Database::create("test_level1");

Transaction t(db->begin());
shared_ptr<measure> m1 = db->load<measure>(34);
shared_ptr<measure> m2 = db->load<measure>(35);
t.commit();
db.reset();
```

```
/*
 * long computation goes here
 */
db =  Database::create("test_level1");

Transaction t2(db->begin());
db->attach(m1);
db->attach(m2);
t2.commit();
```

4.4 Sessions

The contract of the session is to guarantee the uniqueness of a ddl-object; as say that for each persistent counterpart, at most one ddl-object is managed by the session. Furthermore, each modified ddl-object held by the session will trigger an update in the database during the next transaction commit. This contract allow us to associate ddl-objects and work with them without warring about dealing with multiple copies of the same persistent counterpart. A database transaction is always paired with exactly one session. If there is an extended session, then this one is used; Otherwise, the system creates a new session which ends the transaction is either committed or rolled-back.

4.4.1 Extended Sessions

As a best practice, you want the database transaction last as little as possible. But you also want to take advantage of the guarantees (uniqueness and synchronization) offered by the session. You can fulfill this two goals using the extended sessions. An extended session life is managed by the methods Database::begin_session() and Database::end_session() and can span multiple transactions.

```
shared_ptr<Database> db = Database::create("test_level1");

db->begin_session();

Transaction t(db->begin());
shared_ptr<measure> m1 = db->load<measure>("measure_456");
t.commit();


Transaction t2(db->begin());
shared_ptr<measure> m2 = db->load<measure>("measure_789");
t2.commit();

db->end_session();
```

Because of the unicity guarantee, if we try to load a ddl-object more than once, even in different transactions spanned by the same extended session, the system will return us the same object pointer without even touching the database. Note that when you use extended sessions you can occur in the same lost update problem described before: in a subsequent transaction (spanned by the same extended transaction) the system may override a persistent counterpart updated concurrently by another user.

In order to provide you a valid result, before any query execution the system automatically flushes any modification made on the ddl-objects attached on the current session. As you might expect, this can also cause lost updates.

```
shared_ptr<Database> db = Database::create("test_level1");

db->begin_session();

Transaction t(db->begin());
shared_ptr<measure> m = db->load<measure>("measure_456");
t.commit();

m->run_id(555);

Transaction t2(db->begin());
// m related persistent counterpart is flushed in the database
Result r = db->query<measure>("run_id > 500","name asc");
t2.commit();

db->end_session();
```

4.5 Query Language

The Das system provides a powerful query language using the simple and familiar object-oriented point notation. A Das query is composed by two clauses: a query expression and an optional ordering clause. The query expression in turn is a composition of one or more boolean expressions.

4.5.1 Keyword reference

We can refer to the keywords of the ddl-type query target by simply typing its name in an expression.

```
<keyword_name_1> + <keyword_name_2> == constant-value
```

To refer a keyword of an associated ddl-type we prepend the name of the association to the keyword.

```
<association_name>.<associated_keyword_name> == constant-value
```

Given this ddl configuration:

```
+---------+      +---------+      +---------+
| type A  |  +-->| type B  |  +-->| type C  |
+---------+  |   +---------+  |   +---------+
| assoc_b |--+   | assoc_c |--+   | key_1   |
+---------+      +---------+      +---------+
```

We can navigate through the association chain with the same point notation.

```
assoc_b.assoc_c.key_1 == const-value
```

4.5.2 Boolean expressions

Boolean expressions may be composed with the C boolean operators: && , || , ! . We can also use round brackets in order to explicit the operator associativity and priority.

```
exp1 && exp2 || !(exp3 || exp4)
```

The Das query system provides two string operators: startsWith() and endsWith() as valid boolean expressions. We can use this operators like a method invocation on a keyword of type string with a const string argument (a string enclosed by single quotes).

```
keyword_name.startsWith('cmp') && keyword_name.endsWith('_test')
```

Other valid boolean expressions are the comparison expressions.

4.5.3 Comparison expressions

Comparison expression allow us to compare arithmetic expressions, keyword references and constant values. Again, we use the C syntax with the well known semantic: < , > , <= , >= , == , != . Round brackets may be used here as well for more readability.

```
keyword_name == 'test_1'
keyword_1 >= keyword_2 + keyword_3
keyword_1 < 5.6749
```

4.5.4 Arithmetic expressions

We can compose an arithmetic expression using keyword references, integer and floating point constants and C arithmetic operators: + , - , * , / .

```
keyword_1 * (keyword_2 + keyword_3)
```

4.5.5 Ordering Clause

The Ordering clause allow us to order the result set according to the values of the specified keywords. We can specify a list of comma separate couples keyword/order where keyword is a name of the target ddl-type and order may be the word ascending, descending, asc or desc.

```
keyword_1 asc
keyword_1 asc, keyword_2 desc
```

5 API documentation

5.1 Data API

This section of the API shows how you can manage the data of the ddl-objects.

5.1.1 Array

Array data type is the base data type managed by the DAS system. It is a thin wrapper of the Blitz++ Array object. Yuo can use all the methods provided by blitz::Array, with the exception of storage order which must be row-major, and the non-contiguous sub arrays. An exaustive guide about blitz can be found here: http://blitz.sourceforge.net/resources/blitz-0.9.pdf.

5.1.2 DasObject

DasObject is the parent of each ddl type. It provides the interface for data access and the polimomorphic interface to access metadata and associated objects.

There are also a few object introspection methods which provide information about keywords, columns and images stored in the object:

```
const KeywordInfo&
get_keyword_info(std::string keyword_name);

const ColumnInfo&
get_column_info (std::string column_name);
```

These methods return the essential metadata stored in each ddl object:

```
long long das_id () const;
const std::string & dbUserId () const;
const boost::posix_time::ptime& creationDate() const;
const short& version () const;
const std::string& name() const;
```

Given a ddl-object, for each keyword defined in the ddl-type, a get and a set method named as the keyword is provided. For your convenience the DasObject present generic set and get methods. Again, if the name of the keyword, passed as the first argument, isn't known, an exception will be thrown.

```
template< typename T >
void set_key (const std::string &keyword_name, const T &value);

template<typename T >
das::optional< T > get_key (const std::string &keyword_name);
```

As for the keywords, even for manipulating the object associations, a get and set method named as the association, is provided.

The DasObject, present set and get methods. Be carefoul to use the correct method according to the multiplycity of the association: if you have a “many” association, you must use the method set_associated_objects() method even if you want to associate only one object. Note also that there are no such append_associated_object/s method, which means that any previously associated object not present in the new association vector you provide as argument, will not be associated to the

object anymore.

The DasObject provide the generic get and set associated methods:

```
shared_ptr< DasObject >
get_associated_object (const std::string &assoc_name)

std::vector< shared_ptr < DasObject > >
get_associated_objects (const std::string &assoc_name);

void set_associated_object (
   const std::string &assoc_name, shared_ptr< DasObject > &obj);

void set_associated_objects (
   const std::string &assoc_name,
   std::vector< shared_ptr< DasObject > > &vec);
```

You can append or retrive a column table data from a ddl-objects which includes a binary table through these methods. You have to provide the name of the column you want to access as the first argument. If the ddl-object does not contain the specified column or it contains image data, an exception will be thrown.

```
template<typename T>
das::Array<T>
get_column (const std::string &col_name,
          size_t start = 0,
          ssize_t length = -1);

template< typename T >
void
append_column (const std::string &col_name,das::Array< T > &a);

long long get_column_size (const std::string &col_name);
```

Similar methods are provided for columns containing arrays.

```
template<typename T , int Rank>
das::ColumnArray< T, Rank >
get_column_array (const std::string &col_name,
                size_t start = 0,
                ssize_t length = -1);

long long get_column_array_size (const std::string &col_name);

template< typename T , int Rank>
void append_column_array (const std::string &col_name,
das::ColumnArray< T, Rank > &a);
```

For what concern the image data access, you can retrive a patch of the image, providing a range object for each extent the image has. If you are interested in all the image, you can provide no range or Range::all() as the method argument.

```
template<typename T , int Rank>
das::Array< T, Rank >
get_image (das::Range r0, das::Range r1, das::Range r2,
das::Range r3, das::Range r4, das::Range r5, das::Range r6,
das::Range r7, das::Range r8, das::Range r9, das::Range r10);
```

You can use the set_image method to clear the currrent content of the image and set the provided one. If you want to append one or more tiles, intstead, use the append_tiles method.

```
template< typename T , int Rank>
void set_image (das::Array< T, Rank > &i);

template< typename T , int Rank>
  void append_tiles (das::Array< T, Rank > &i);
```

5.2 Persistence api

5.2.1 Transaction

The transaction object manages and represent a database transaction. Both of its methods finalize the transaction. If neither of them is called before the Transaction object goes out of scope, the transaction will be automatically rolled back.

```
void commit();
```

This method cause all the objects involved in the transaction to flush their data into the store, and their metadata into the database. Following the atomicity property of the database transaction, if any problem arise while saving data files, the transaction will be rolled back.

```
void rollback();
```

This method cause the transaction to abort and to roll back all the changes made since the beginning of the transaction. The new files will be moved back to the temporary location and the database transaction will be rolled back by the DBMS.

5.2.2 Template API

The template API allows you to use narrower ddl-object types in order to use the specific methods each type provides. For example the load() method returns a ddl-object typed as the template argument provided.

5.2.3 Database

As discussed in chapter 4, the Database class implements the the persistence manager interface. This object handles DBMS connections, sessions and managed objects.

```
shared_ptr<Database>
create(const std::string& alias);
```

This factory method returns a Database object which accesses the DBMS configured with the alias provided as argument. You can have multiple Database objects active referencing the same database schema. Each one manages its own connection with the database. Note that you don't have to provide any DBMS credentials, because those are automatically fetched from the configuration files.

```
 Transaction
 begin(isolation_level isolation = databaseDefault);
```

This method returns a Transaction object which manages a new database transaction. Yuo can provide the transaction isolation level for this transaction as argument; if you don't provide any argument, the default isolation level configured for the DBMS will be used (usually repeatable read).

The das::already_in_transaction exception will be thrown if another transaction is active.

```
template<typename T>
shared_ptr<T>
load(const long long &id);
```

This method return the ddl-object correspoding to the key provided as argument.

```
template<typename T>
shared_ptr<T>
load(const std::string& name, int version = -1);
```

This methods return the ddl-object correspoding to the name, version provided as argument. If no version number is provided, the last version available is returned

For both the previous and thi methods, If no such object exsists, an exception is thrown.

Note that if the object is already cached in the session, no load query against the database occour.

```
template<typename T>
long long
persist(const shared_ptr<T> &obj, std::string path = "");
```

This method persist the object in the database, you can provide an optional path argument which will be used as a sub-path to where the data will be stored.

```
template<typename T>
Result<T>
query(const std::string& expression,
    const std::string& ordering = "",
    bool last_version_only = true);
```

This method returns an object representing the result of the query provided as expression. Result can be used to iterate through the result set. Note that the Result object is no longer valid when the

relative transaction is finalized. See section 5.2.4 for further about the Result object.

If you plan to perform heavy work while iterating through the result, consider using the following methods which return a list of object ids and a list of name/version pairs. You can then iterate through the list and load one object at time.

```
template<typename T>
std::vector<long long>
query_id(const std::string& expression,
               const std::string& ordering = "",
               bool last_version_only = true);


template<typename T>
std::vector< std::pair<std::string, short> >
query_name(const std::string& expression,
                    const std::string& ordering,
                    bool last_version_only = true);

template<typename T>
void
attach(const shared_ptr<T> &obj);
```

This method allows you to attach an object to the current session, either an extended one or a transaction bound.

```
void
flush();
```

With this method you can force the update of the data and metadata of the objects managed by the current session, this method is automatically called in the transaction commit phase.

As described in section 4.4, you can take advantage of the sessions spanning multiple transactions using those methods in order to begin and and an extended session.

```
void
begin_session();

void
end_session();
```

5.2.4 Result

The Result object encapsulates the result of a database query. It provides iterators and constant iterators in order to access to the result ddl-objects. Moreover, you can use the methods empty() and size() to know the amount of the objects returned by the query. Note that once the the transaction

used by Result to retrive the ddl-objects is finalized, any attempt to access or dereference the the result_iterator will end in an exception. The result ddl-objects aren't loaded until you dereference the iterator. Furthermore you can increase the iterator or even get the current object id without loading the entire object.

```
bool empty();
size_t size();
Iterator begin();
iterator end();
const_iterator cbegin();
const_iterator cend();
```

5.2.5 Polimomorphic API

The polimomorphic API is more suitable all those scenarios where the exact data type is known only at run time. This API provide very similar methods to the template one, exept the type returned, which is the polimomorphic DasObject, and the way you specify the data type when required: an additional string parameter instead of a template argument.