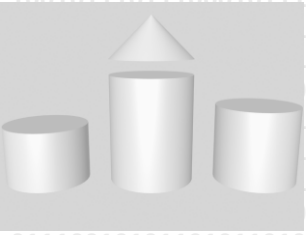

ODABA^{NG}

Documentation

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, October 2012

Content

Introduction **5**

User's Guides **6**

1.1 ODABA User's Guide 7
1.1.1 Using ODABA 8

1.2 Model definition 12
1.2.1 Schema definition 13

1.3 Database access in C++ 14
1.3.1 Creating Handle hierarchy 15
1.3.2 Access by data source 21
1.3.3 Accessing data 25
1.3.4 Advanced property handles 60
1.3.5 Buffered Access (buffer mode) 87
1.3.6 Property handle cache 89
1.3.7 Client/server configurations 90

1.4 Special Features 91
1.4.1 Object Identities 92
1.4.2 Versioning 94
1.4.3 Copy model 101
1.4.4 Check model 102
1.4.5 Recovery log-file 103
1.4.6 Workspace 104
1.4.7 License services 105

1.5 Locking and write protection 106
1.5.1 Locking Features 107
1.5.2 Write protection 108

1.6 Transactions 109
1.6.1 Starting and committing user transactions 110
1.6.2 Starting and committing workspace transactions 111

1.7 Database context programming 112
1.7.1 Associate context class with data model resource 113
1.7.2 Handling events 114
1.7.3 Providing actions 118

1.8 Data exchange 119
1.8.1 Data exchange definition 122
1.8.2 Data Exchange schema 127
1.8.3 External data formats 133

1.9 ODABA data storage formats 139
1.9.1 Storing ODABA data in relational databases 140

1.9.2 XML database 147
1.10 Internet Communication Engine 148

Introduction

ODABA2

ODABA2 is an object-oriented database system that allows storing objects and methods as well as causalities. As an object-oriented database, ODABA2 supports complex objects (user-defined data types), which are built on application relevant concepts.

ODABA2-applications are characterised by a high flexibility that is achieved by supporting in addition to object (concept) hierarchy, multifarious relations between objects (master and detail relations, relations between independent objects and others). This way conditions and behaviour of objects in the real world can be represented considerably better than in relational systems.

ODABA2-applications cannot only be drawn up as event-driven applications within the field of the graphical surface but also at the database level. This is one more way in which the application design is very close to the problem.

This makes ODABA2-applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

Platforms

ODABA2 supports windows platforms (Windows95/98/Me, Windows NT and Windows 2000) as well as UNIX platforms (Linux, Solaris).

You can build local applications or client server applications with a network of servers and clients.

Interfaces

ODABA2 supports several technical interfaces:

- C++, COM as application program interface (this allows e.g. using ODABA2 in VB scripts and applications)
- ODBC (for data exchange with relational databases)
- XML (as document interface as well as for data exchange)

User Interfaces

ODABA2 provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA2 provides a special ODABA2 GUI builder.

User's Guides

In contrast to reference manuals, the user's guide discuss typical programming patterns and specific programming or development situations.

1.1 ODABA User's Guide

This document describes different ways for modeling and accessing an ODABA database. Moreover, it describes the way of using special features as workspaces or transactions.

The document is not describing the details of functions, which are given in the appropriate reference manuals. It describes rather the way of using certain features by means of typical examples.

The document is rather new, but we found it important to add a document of this kind. Even though it is not ready yet, it might be a useful help in many situations.

1.1.1 Using ODABA

Running ODABA applications is relative simple. ODABA programms need not to register. They are just moved to a folder and started. Databases will be created on demand, when not yet existing.

Hence, the only activity left after installtion for starting a database application is preparing a configuration or ini-file containg information about database locations, which are usually not hard coded in the application database.

Application calls usually refer to a configutation or ini-file that defines necessary options as database locations,server connections etc.

Options in database applications

Most ODABA applications are referring to a configuration or ini-file in order to load actual option values. Moreover, ODABA provides an option dialog, which allows storing options in the application database.

Options are considered as thread global variables, i.e. each thread running has its own set of options. This allows changing option values in one thread without affecting other threads.

Options are supported in many places in th database system in order to increase flexibility of applications and programs. Thus, you may refer to options in

- file paths
- OSI expressions
- application programs

and many other places. You may create your own options in your application or refer to options defined in ODABA.

Option values are searched in the following hierarchical order.

- First the options set internally by the application are checked.
- When not being found, the option is searched in the configuration or ini-file associated with the application.
- When not being defined in the ini file, the options is requested from the application settings via the application context function BaseContext::option() (supposed, an application context has been set (setApplicationContext()).

When the option name is a simple name, it is searched in addition in the system environment as environment variable.

Option values not yet defined are created automatically and set to "". Thus, each option does exist by definition and returns a value, which might be empty.

File path options

Database paths but also most file path to external files opened in an ODBA application (e.g. locations for import or export) may contain option references (%sym_name%). Those references are resolved before opening the file. A file path may contain any number of option references.

When the file is opened on the server, optons must have been defined on the server side.

```
// the ini-file defines PROJECT_ROOT, e.g.as
// PROJECT_ROOT=C:/odaba/my_projects
// now, tha database path can be referred to as:
%PROJECT_ROOT%/database.dat
```

Referring to options in expressions

OSI expressions may refer to option variables. You may get but also set option values in an expression. Whithin an expression, you may refer to option variables as to normal expression variables, except, that they must be enclosed in %...%.

Referring to undefined option variables returns an empty string.

Options in expressions are typically used in order to define variable or generic expressions, which depend from one or more variables set at runtime.

```
void main () {
  %PROJECT_ROOT% = 'c:/odaba/my_projects'; // set option
  ::Message('Value for PROJECT_ROOT is: ' + %PROJECT_ROOT%);
}
```

Accessing options from whithin a program

In an application, one may retrieve option values by calling option(). In order to set or change an option value, setOption() can be called.

```
  setOption("PROJECT_ROOT","c:/odaba/my_project");
  printf( option("PROJECT_ROOT") );
```

Option hierarchies

Typically, options are arranged in a two level hierarchy. The first level corresponds to the section in an ini-file. The second level refers to variables defined for this section.

After options became widely used in applications, additional leves became necessary. When refering to an xml-configuration file, any level of nesting option variables becomes possible. Moreover, each option may have got a value (which is not possible for sections in an ini.file.

Options under the main section or option, which has got the name MAIN or the name of the program executed, are considered as level 0 options as well as section names in an ini-file. Options below other section in the configuration or ini-file are considered as level 1 and higher options.

Option path

I order to access options on higher levels in an program or expression, option paths can be defined, which describe a hirarchy path for the option. Option path can be referred to wherever an option value is required, i.e. instead of simple names, you may refer to an option path e.g. within an OSI option variable or file path.

There is no limit for the number of level, but one should not exceed 3 or 4.

In order to support option hierachies with a level greater than 1 in ini-files and environment settings, too, option names in ini-files and environment settings may contain option paths as well.

```
// ini-file
[DATA_SECTION]
DICTIONARY=c:/odaba/my_projects/sample.dev
DICTIONARY.TYPE=ODABA
// OSI expression
  ::Message("dictionary type is: "+%DATA_SECTION.DICTIONARY.TYPE%);
```

Database path

A database path refers to the location of the root base for the database. Paths can be defined as absolute or relative paths, or as reference path containing symbolic references enclosed in %% (%sym_name%).

Symbolic path references are resolved by replacing corresponding option values defined in the application or main option section in a configuration or ini-file. When running local or file server applications, symbolic names in the path must be defined on the local machine. When running a client/server application, the database file path is passed to the server and symbolic variables are resolved (and must be set properly) on the server side.

Option settings

Options can be set in different places. Primary options are read from the configuration or ini-file. Some applications (e.g. most ODABA system applications) add option values defined in the database (user specific option settings). Options, taht have neither been found in the configuration or ini-file nor in database user settings, are read from the system environment.

Finally, options can be set in the application (OSI expressions or implemented functions).

When an option has been det once, it remains until it si changes by the application, i.e. changes in the configuration or ini-file or in database setting will not apply automatically.

In order to refresh settings (e.g. applying updated user settings in the database) the application has to refresh options explicitly (e.g. by reading updated options and setting those).

1.2 Model definition

Model definitions can be provided for the data model, the dynamic model and the functional model. All models can be defined in terms of ODL definitions, XML schema or in the Class Editor.

The most comfortable way is the Class Editor, but sometimes, ODL definitions or XML schema might be a better choice.

In this chapter, we will discuss certain ways of defining model elements by using ODL, because it is a simple way of referring to examples.

1.2.1 Schema definition

A schema definition includes definitions for all three models the database model consists of. When defining a schema in ODL, the dictionary path must be passed as value for the DICTIONARY keyword or as parameter to the ODL utility when loading the schema.

A schema is stored in dictionary or resource database as project. Projects may form hierarchies, i.e. a schema may consist of any number of sub-schemata. On the other hand, each schema forms a module, which may have sub-modules again.

Since each schema is a module by definition, an explicit module definition is not required. Modules are namespaces and may consist of any number of subordinated namespaces. Again, explicit namespace definitions are not required for a module or schema (project), since each schema or module is a namespace.

```
// location for resource database
DICTIONARY = 'c:/ODABA2/Sample/Sample.dev'; // sample resources


UPDATE SCHEMA Sample {
// definition of schema resources ....
}
```

1.3 Database access in C++

This chapter explains the necessary steps for accessing a database in C++ programs. This is similar to accessing the database from other programming languages as C# (.NET interface) or PHP (ICE interface). Only the syntax differs slightly according to the specific program language requirements.

Accessing databases from within OSI is different, because OSI provides direct access to database variables. This is described in detail in "ODABA Script Interface".

Accessing a database is possible by creating a hierarchy of access handles on 6 levels or by referring to an externally defined data source. Here, we will consider the more particular way of creating handle hierarchies before discussing the data source concept, which provides a more generic way for accessing a database.

1.3.1 Creating Handle hierarchy

For getting access to database details you may open a hierarchical structure of database access objects:

Client

Dictionary

Database

[ObjectSpace]

Property

Value

Each of those access handles can manage any number of subordinated access handles, i.e. you may several databases for the same dictionary etc. The object space handle is required only, when you are working with multiple object space (or multiple universe) databases. Object space handles must be created also, when you are running transactions on the same database in different threads.

Handle hierarchies correspond to physical application resources, which are managed by the resource specific objects referenced by the handle:

CClient	- Server connection
DDictionary	- Resource database
DDatabase	- Database
DDBObject	- Universe/transaction
PProperty	- Instance/collection
VValue	- Value

The example below shows how to get access to persons in the sample database.

```
Dictionary sam_dict(mainClient(), "Sample.dev", PI_Read, true);
Database   sam_database(sam_dict, "Sample.dat", PI_Read, true);
Property   persons(sam_database, "Person", PI_Read);


while ( persons.next() )
  printf(persons.valueString("name"));
```

Database application

When running a database application, an application environment is created implicitly in order to make global resources available for database functions. The application manages several system resources, which can be defined in different sections of a configuration or ini-file:

SYSTEM - System resource database (contains e.g. error messages)

CACHE - initialize server or application cache settings (optional)

FILE-CATALOGUE - defines file locations for symbolic file names (optional)

DATA-CATALOGUE - defines data sources in a database (optional)

In order to initialize these resources, i.e. in order to make system errors and data catalogs available, odaba::initializeApplication() might be called with a configuration or ini-file. The function client will initialize the system resources from definitions passed in the configuration file. The application registers the process and activates the error log-file. It opens the system database for providing error messages and the data catalog. System resources are initialized once, only, and will be closed when leaving the application or when shutting down the application explicitely (shutDownApplication()).

Default system and catalog sections are defined in the ODABA.INI file that is stored in the ODABA installation path. The system section should always refer to the odaba.sys database. This might be located on a server, in which case the data source has to provide connection information (server name and port). In order to provide an application specific data or file catalog, application specific settings can be defined in the configuration file passed to the application.

The system database is opened automatically, when the configuration file passed to initializeApplication() contains a SYSTEM section or an appropriate top option element has been defined in the configuration file. When the system database is not opened, the application still runs fine, but system errors are written to the error log-file without explanatory test.

When running the SYSTEM database from a server, the main client automatically connects to the server with the system database (odaba.sys). When this connection refers to the same server as other data sources, the application should use the main client in order to connect to other databases.

Client Handle

The first you need to create an access handle hierarchy is a Client handle. The client handle guarantees the scalability for ODABA applications, since just by configuring the client in different ways, you may run your application as local, file server, object server or as replication server application.

The configuration of the client/server mode for the application is provided in a configuration file (traditional ini-file or xml option file) or in a data catalog. Details about defining data sources are described in the "Database Reference" manual.

Most applications work with one client, which is required only, for opening the dictionary handle(s). Complex client/server applications communication with different databases on different servers may, however, require several client objects. Several client objects might also become necessary, when running multiple thread applications in order to provide a separate server connection for each thread.

There is a big difference between client handle and a client object. The client handle is not much more than a managed pointer to a client object. Thus, many client handle may refer to the same client object. ODABA tries to minimize the number of client objects, i.e. the number of connections to the server. Simple applications use one client object only, which is called the system client. Multiple connection applications will implicitly create a number of client objects managed by ODABA.. But when the application becomes very complex concerning the connections to different servers, the application may implement their own client manager.

Empty client handle

When scalability is not requested and the application is able to obtain database location information from other sources, you may refer to the main client handle.

When referring to the main client, DataSource::open() will not work, unless you initialize the client explicitly, since no data source reference has been defined for the client handle.

Empty main clients need not explicitly connect to the database, when running in local or file server access mode. Running, however, in client/server mode, empty client handles must be connected explicitly to the server.

```
// empty client handle opens local dictionary

  ODABAClient       client();

  DictionaryHandle  dict(client, "C:/ODABA/Sample/Sample.dev",
PI_Read, true);


// empty client handle opens server dictionary

  ODABAClient       client();

  client.Connect("my_server", 6123);

  DictionaryHandle  dict(client, "%SAMPLE_DICT%", PI_Read, true);
```

Scalable client handle

When running an application in client/server mode, each client can manage maximum one connection to a server. Thus, when running an application communicating with several servers, several clients must be created. Each server connec-

tion allows accessing any number of dictionaries or databases provided on the server.

Scalable (initialized) client handles can be created also by constructing an empty client and initializing it.

Scalable clients work fine in local and client server mode as long as all data sources requested by the application are running locally or on the same server. A client may serve at the same time local or file server data sources, but not data sources residing in different servers.

For handling different servers or multiple server connections, you need a client handle for each server connection.

```
// scalable client handle

  ODABAClient   sam_client("c:/ODABA/Sample/Sample.ini", "Test
Application");
```

Initialize client explicitly

When a main client had been created, the client can be initialized later on by calling Client::reopen(). The client must be initialized before opening a data source or connecting to a database. Initializing a client should not be done while the client is connected, but initialization can be re-done, after the client has been disconnected.

Initialized clients need not explicitly connect to the database (server). The connection is established automatically, when opening the first data source for the client.

```
  ODABAClient        client();

  ...

  client.Initialize("c:/ODABA/Sample/Sample.ini", "Test
Application");
```

Multiple server connections

When referring to data source definitions defined in a data catalog or in an configuration file, the application need not care about server connections or local database access.

More complex applications requesting databases from different servers need an own connection management in the application or a well-organized set of configuration files. Since configuration files keep the application free from knowing anything about the client/server mode, this is the suggested way to handle multiple server connections.

One could create a client for each data source requested in the application, but when the application accesses several data sources residing on the same server, it is more efficient, using only one connection for all this data sources. The best

way is to put all data sources for one server in the same configuration file and creating a client for each configuration file. Still the application needs to know, which data source is defined in which configuration.

Instead trusting the automatic connection feature of the client handle, you may manage the server connections by your own in the application.

```
// connection manager (draft)
bool openConnection(DataSourceHandle &dsh, ClientPool &cp) {
  Client  *current_client = 0;
  if ( !(current_client = cp.Locate(dsh.server_name)) ) {
    current_client = new Client();
    if ( current_client->connect(dsh.server_name, dsh.server_port)
) {
      printf("server %s(%i) not accessable", dsh.server_name,
dsh.server_port);
      delete current_client;
      current_client = NULL;
    } else {
      cp.AddClient(dsh.server_name, current_client);
      dsh.Open(*currentClient);
    }
  }
}
```

Main client

The first client created in an ODABA application is considered to be the main client. The main client handle is used, whenever a client handle is required but not set or passed explicitly.

When initializing the application by calling odaba::initializeApplication(), teh main client is automatically created for connectin the SYSTEM data source. When the dara source refers to a database on a server, the main client automatically connects to this server.

Otherwise, the main client is created implicitly when creating the first Client object. The main client can be provided simply by calling odaba::mainClient(). Creating a default data source (without client reference), the data sourec uses the main client. When no main client has been created so far, calling mainClient() will automatically create one.

-

Using Dictionary Handle

Using Database Handle

Using database object space handle

Property handle

A property handle is used to access data for a defined property. A property could be a collection of data (instances or elementary values) but also a single value. A property handle provides methods to navigate within the property as well as methods for reading and updating the property content.

Detailed description of property features you can find in the Property Handle Class Reference.

Value handle

1.3.2 Access by data source

Usually, databases can be opened by [creating or opening a client handle], creating a dictionary handle and creating/opening a database and object space handle. Database locations have to be passed to those functions, which is simple but reduces flexibility of the program.

Accessing the database via predefined data source provides more flexibility to the application. There is still a client object required, but the location for database and dictionary can be provided externally in a data catalog or in an ini-file.

The example below shows, how to open the Sample database data source defined in the "sample.ini" configuration file. The section in the configuration file (or xml element in an xml configuration file) must have got the name referred to as data source name in the openDataSource() function ("Sample"). Details for defining data sources are described in the "Database References" manual.

The example below illustrates two ways of accessing (opening) a data source. The first opens a database fith fixed parameters defined in the program. The second shows one way of opening the same database but controlled by options defined in an ini-file.

```
// open static resources
  Dictionary    dict("c:/odaba/sample/sample.dev");
  Database      base(dict,"c:/odaba/sample/sample.dat",Write);


// open data source parametrized in an configuration or ini-file:
// [Sample]
// DICTIONARY=c:/odaba/sample/sample.dev
// DATABASE=c:/odaba/sample/sample.dat
// ACCESS_MODE=Write
// NET=YES
  ObjectSpace base;
  base.OpenDataSource("Sample.ini", "Sample", Write);
```

Data source properties and option names

Data source properties can be initialized from options set in a section of a configuration or ini-file or from data source definitions in a data catalog. Within a configuration file, data source properties must be defined in a section with the data source

name ([datasourceName]). Data source properties are initialized from following options:

serverName: REPLICATION_SERVER or SERVER_NAME

serverPort: SERVER_PORT (6123)

connectionName: CONNECTION_ID

This variables define data source connection parameters, which are required for for object or replication server clients, only.

dictionaryPath: DICTIONARY

dictionaryType: DICTIONARY.TYPE (ODABA)

This variables define the dictionary database. This dictionary path is mandatory. The value may refer to a server variable that defines the path on the server. Server variables must be enclosed in % characters (e.g. %DICT_PATH%).

databasePath: DATABASE

databaseType: DATABASE.TYPE (ODABA)

This variables define the database to be opened. The path may refer to a server variable that defines the path on the server. Server variables must be enclosed in % characters (e.g. %DB_PATH%).

accessMode: ACCESS_MODE (Read)

This variable has to be defined in order to accessing a database in write mode (Write).

sharedDatabase: NET (NO)

When opening a data source in local or file server mode (no server defined) this option can be defined in order to share the database with other applications.

enableContext: ENABLE_CONTEXT (YES)

This option allows deactivating the database context defined for the project, i.e. disabling logical consistency or busines rules defined by the application. This is useful e.g. for maintenance or reorganisation processes.

onlineVersioning: ONLINE_VERSION (NO)

This option activates online-versioning in order to update instances to next higher schema version, when the database schema has been changed. When not using online version feature the database has to be reorganized before a new schema version can be used.

databaseVersion: VERSION

Version number for the object space or database, when the database or object space should be opened with an older (not the current) version. When no version

number is passed, the object space or database will be opened with the current version.

schemaVersion: SCHEMA_VERSION

Schema version has to be set in order to open the database for an older schema version (not the latest version) of the dictionary. When no schema version is passed, the database will be opened with the latest schema version.

objectSpaceName: OBJECT_SPACE

The name of an object space must be specified in order to access a sub object space in the database is to be opened.

accessPath: ACCESS_PATH

An access path to a collectin can be defined in order to refer to a collection instead to a database or object space.

resourceType: RESOURCES.TYPE (ODABA)

This variable defines access options for a resource database optional). When resources are stored in a database different from the dictionary, those can be made available using the resource database.

workspacePath: WORKSPACE

When the workspace feature is enabled for the database, a workspace can be defined as active workspace for the data source by passing a workspace name or a workspace path.

typeName: DATA_TYPE

The data type name is used in some cases for performing metadata operations (e.g. copying a data type definition to another dictionary). It is has no direct influence on the data source but can be retrieved by the application.

Alternatively to the ini-file definitions tha data source can be described in a data catalog. In this case you may refer to the data source name defined in the data catalog, instead defining the data source in a configuration file.

dataSourceName: DATA_SOURCE

The data source name refers to the data source to be opened. Usually this is the same name as the data source name passed to the function, but it is also possible to refer to another name in this place.

Additional options can be passed in the configutation file, which are not stored in the data source handle.

XS_NAMESPACE

Location for the xml database schema (application schema), when running ODABA as an xml database. The schema location can be local or a WEB URL.

OXS_NAMESPACE=odaba_schema_location

Location for the odaba shema definition (system schema), which has several extensions to the xml schema definition. The schema location can be local or a WEB URL.

```
[DataSource1]


; Data source


ODABA_SERVER=SRV008(6123)


DICTIONARY=%BridgeDict%
DATABASE=%BridgeProduction%
EXTENT=ClassificationVersion


ONLINE_VERSION=YES
ACCESS_MODE=Write
NET=YES
```

1.3.3 Accessing data

In program environments as C++, .NET, JAVA or PHP, access to data is managed by property handles. Property handles provide the lowest level in the access hierarchy.

Property handles can be used for accessing persistent data as well as transient. You may use property handle for iterating through a collection, but also for accessing an elementary value in an object instance.

For reading, creating, updating or deleting data, property handles are required as well. Besides simple data manipulation functions, property handles allow locking instances, support event handlers and transactions, provide data conversion, metadata and many others.

This chapter explains the basic functionality required for accessing data in a ODABA database. Advanced features are described in the next chapter and details about how to use property handles in details are described in the class reference.

How property handles work

Property handles form a hierarchical structure, where each property handle represents a property (attribute or reference) in an instance hierarchy. A property handle contains a pointer, which points to (property) node in a property handle hierarchy. Property nodes, which are referred from the property handle, provide instance, cursor and metadata functionality.

Property handles can be constructed or opened. It does not make any difference for the property handle, how it has been created. Since a property handle is nothing else than a pointer to a property cursor, which handles collections and selected instances, constructing or opening a property handle just provides a property node pointer in the property handle.

For most constructor functions there are similar open functions.

For accessing data, property handles provide instance and cursor functionality. This functionality follows some basic rules, which has to be taken into account, when navigating in property handle hierarchies.

Property handle hierarchies

Property handles can be opened in a hierarchical order as shown below. The top property handle must be opened with a database object handle as parent. Subordinated property handles are opened with a property handle as parent. Each property handle can be the parent of any number of subordinated property handles.

Considering a Person object in the sample database, you will find the children property. Since the children property refers to a collection of Persons, again, persons in the children collection may have children, too.

Property handles opened in a hierarchical order, as shown below, will always reflect exactly one path through the instance tree defined by the reference properties in the hierarchy. I.e. selecting a person in the top person property handle creates the set of children for the subordinated children property handle. Changing the selected instance in the top property handle person, automatically changes the collection represented in the subordinated children property handle to the set of children for the newly selected person on top.

Selecting a person in the children property handle will immediately provide a set of grand children in the lowest property handle, i.e. the children of the child selected from the set of children for the top most person.

```
void OpenHandleHierarchy ( DBObjectHandle &dbo ) {
  PropertyHandle     person(dbo,"Persons",PI_Read);
  PropertyHandle     children(person,"children");
  PropertyHandle     grand_children(children,"children");

  ...
}
```

Instance functionality

A property handle provides instance functionality for the instance currently selected in the property handle. Instance functionality allows reading or updating instance attributes or accessing instances in a subordinated property handle.

```
void AccessAttributes ( PropertyHandle &person ) {
  PropertyHandle    name(&person,"name");
  PropertyHandle    pid(&person,"pid");
  person.Get(0);    // select first instance
  pid = "00000";    // set person id value to 00000
  name = "Miller"   // set person name to Miller
  printf("Person ID: %s",pid.GetString());
}
```

Cursor functionality

The property handle provides cursor functionality besides instance functionality. This means, besides accessing properties in the selected instance, you may use the cursor functionality for selecting another instance or iterating through the collection represented by the property handle.

```
void Iterate ( PropertyHandle &person ) {
  person.ToTop();
  while ( person.Next() )
    printf("Person ID: %s", person.GetString("pid"));
}
```

Access requirements

Supposed all specifications you made for a property handle hierarchy are correct, there are some additional important rules to be taken into account when accessing properties in a hierarchy.

A property handle is accessible only, when it is the top property handle in a hierarchy or when the parent property handle is positioned or contains an initialized instance (instance selected).

When changing the selection in a property handle, all subordinated property handles become unselected.

When a property handle becomes unselected, all subordinated property handles become inaccessible.

When selecting an instance in a property handle opened in write mode (PI_Write), the instance is locked (pessimistic write lock). When this is not possible, because the instance is locked by another property handle or application, the instance can still be selected, but is accessible read-only.

When selecting an instance in a property handle opened in update mode (PI_Update), selected instances are locked (optimistic write lock). When saving changes from the instance, conflicts will be detected when existing. In case of a conflict the application can decide whether to discard the current changes or throwing away the changes made by the other application or property handle. We suggest, always to discard the changes made on the currently selected instance.

There are other rules described later on, but those are the most important ones.

Property handle macros

Since open property handle is an activity frequently required in application programming, the are some macros provided for making life easier:

PH - Property handle constructor have the same name as the property name

PHN - Property handle constructor, where the property handle name may differ from the property name

GPH - GetPropertyHandle

The rules for creating handle copies and cursor copies are the same, i.e. passing a property handle pointer to the macro will create a handle copy and passing a property handle will create a cursor copy.

```
void OpenHandleHierarchy ( DBObjectHandle &dbo ) {
  PropertyHandle     person(dbo,"Persons",PI_Read);
  PropertyHandle    *pchildren;

  // PropertyHandle  children(person,"children");
  PH(person,children);

  // PropertyHandle  grand_children(children,"children");
  PHN(person,children,grand_children);

  // pchildren = person.GetPropertyHandle("children");
  pchildren = person.GPH("children");

  ...
}
```

Opening access handles

Property handles form a hierarchical structure, where each property handle represents a property (attribute or reference) in an instance hierarchy. The top property handle must be opened with a database object handle as parent. Subordinated property handles are opened with a property handle as parent.

Considering a Person object in the sample database, you will find the children property. Since the children property refers to a collection of Persons, again, persons in the children collection may have children, too.

Accessing data is possible via generic or typed property handles. Generic property handles provide more control for database access and are save against database model changes. Typed property handles perform a little bit better than generic ones, but provide less control and require re-compilation when changing the database structure. Practically, we do not use typed property handles, since it has turned out, that generic property handles are more stable and provide additional support as update control and data conversion.

```
void OpenHandleHierarchy ( ObjectSpace &os ) {
  Property     person(os,"Persons",PI_Read);
  Property     children;
  children.open(person,"children");
  Property     grand_children(children,"children");

  ...
}
```

Constructing property handle

Constructing a top property handle creates a new property handle with a pointer to the property node in the property hierarchy tree. In most cases, it constructs an empty property handle, which is opened in the constructor. Thus, instead of calling a constructor, you may also call the Open() function with an appropriate parameter list.

A valid property handle can be constructed only, when the parent handle (DBObjectHandle or PropertyHandle) is valid. Otherwise, just an empty property handle without property node reference will be created.

```
 void ConstructHandleHierarchy ( rObjectSpace &os ) {
  Property     person(os,"Persons",PI_Read);
  Property     children(person,"children");
  Property     grand_children(children,"children");

  ...
}
```

Open property handle

The difference between opening and constructing a property handle is not relevant. Opening a property handle, however, can be performed any number of times.

You may open a property handle several times. Always when opening a property handle, the currently opened handle will be closed. Since open returns an error when failing, you may immediately check the success of the open function, in which case the function returns false (no error).

To open a property handle successfully requires a valid parent handle (DBObjectHandle or PropertyHandle).

```
void OpenHandleHierarchy ( Objectspace &os ) {
  Property     person;
  Property     children;
  Property     grand_children;

  if ( !person.Open(os,"Persons",PI_Read) )
    if ( !children.Open(person,"children") )
      if ( !grand_children.Open(children,"children");
  ...
}
```

Typed property handle

So far, generic database access has been discussed. In contrast to generic property handles, types property handles provide direct access to instance data in the context of a C++ class definition. Typed property handles are supported via template classes. Instead of referring to the selected instance in the property node, the application may directly refer to instance properties returned by the template class.

When using typed property handles, the system will not be informed about instance modifications and the application program has to signal updates explicitly. Moreover, no data conversion will be performed and has to be managed by the application program as well.

Another disadvantage is, that you must not define virtual functions for types used in typed property handles, since the ODABA instance factory does not create virtual function vectors.

Hence, we suggest always using generic property handles rather than typed property handles.

```
// using typed Property Handles
  PI<Person>        persons(obh, "Person", PI_Write");


  while ( persons.Next() )
    if ( persons.Get()->UpdateAge() )
      persons.Modify();
// Update function as implemented in the Person Class


bool Person::UpdateAge() {
  int        old_age;
  dbdt       current;


  current.SetDate();
  age = birth_date.Year() - current_year;
  return ( age != old_age);
}


// using generic property handles with update control
  Property    persons(os, "Person", PI_Write");
```

```
Property    birth_date(&persons, "birth_date");
Property    age(&persons, "age");
Date        current;
int         year = current.setDate().Year();

while ( persons.Next() )
  age = year - birth_date.GetDate().Year();
```

Copy property handles

When creating copies of a property handle, a copy can be created on the node level or on the handle level. A node copy creates a new and independent property node for the same property. A handle copy creates a new property handle, which shares the property node with its origin, i.e. original and copy handle always refer to the same property node and the state of the property handles is identical. Whatever is selected in the property handle p1 (see example HandleCopy() below) is selected in the property handle person, which refers to the same property node as p1.

Property nodes have got a reference count and the property node will be destroyed when destroying the last property handle referring to it.

In the CursorCopy() example, a new and independent property node had been created and was placed at the same position in the handle hierarchy as its origin.

```
 void HandleCopy ( PropertyHandle &person ) {
  PropertyHandle   p1(&person); // passing handle pointer to get a handle copy
  p1.Get(0);
  printf("Same instance s% = %s", p1.GetString("pid"), person.GetString("pid"));
  p1.Close();        // cursor in p1 is still allive
  person.Close();    // cursor is destroyed - last reference closed
}


void NodeCopy ( PropertyHandle &person ) {
  PropertyHandle   p1(person); // passing handle reference to get a cursor copy
  person.Get(0);
  p1.Get(1);
  printf("Different instance s% = %s", p1.GetString("pid"), person.GetString("pid"));
  person.Close();     // p1 cursor is destroyed - loosing its origin
}
```

Using keys

Keys are defined as data type projections from a complex data type, i.e. keys refer to a subset of properties of a complex data type. Ususally, keys consist of a number of key components, wich are related to attributes of the complex data type.

Key components can be marked as "not being case sensitive" and "descending". Both option will influence comparing operations and order in indexes. When descending is set for a key component comparing keys will invert the result, i.e. the lower key value becomes the greater one. Since key compare functions are called also in order to create and update indexes, key component setting allow determinig whether an index is ordered ascending or descending.

Keys can be assigned to indexes (by name) in order to sort instances in a collection according to the key definition. Instances are ordered in indexes in ascending order.

One key per complex data type can be defined as identifying key. This is a conceptual approach indicating, that this is a common key for identifying instances in any collection. There is no consistency check, i.e. the system does not look for instances with duplicate keys in a global scope. In order to avoid duplicate keys, instances can be stored in a global extent that is ordered by the identifying key and required unique key values.

In order to use keys in programming, keys are passed as key strings (Key) in object interchange format (OIF). The Key class provides some features in order to create proper OIF strings for the key.

Reading data from database

Reading data automatically happens, when selecting an instance in a property (handle). Thus, you may read instances by calling next() or prevoius() but also by reqesting a specific instance calling get().

Instances are read automatically, when opening a property handle for an access path referring to exact one instance.

After an instance has been selected, you may access instance data by extracting data from the property handle calling the instance() function. Instance returns serialized data for the selected instance in OIF format. This is good for data exchange or for passing data to other sysrtems, but it is not dificult to access in detail, since it requires complex syntax analysis.

More appropriate for accessing data from a instance selected in a property handle is property access by calling different property functions.

From within OSI expressions instance properties can be accessed directly by name. Nevertheless, also OSI expressins somtimes require generic property access (e.g. when combining data from different databases in an expression).

```
// C++
Property     persons(database,"Persons",PI_Read);
while ( persons.next() )
  printf("Persons ID: %s",persons.string("pid");
```

Read by Position

Reading data by position is a good mean in order to position the property on a certain position. For iterating through a collection, one should use rather next() and previous(), since get() will throw an exception, when the instance required is not available. Moreover, next() and previous() evaluate filter conditions and automatically skip instances that do not fulfil the filter condition.

The position of an instance in a collection depends on the sort order for the collection. When a sort order has been selected (setOrder()), instances are always provided according to this sort order.

Selecting instance by position is posible in different ways:

reading instance at absolute position

reading instance at relative position

scrolling forward or backward

Absolute position

The simplest one is calling get() with a set position (first instance always has position 0). The get() function selects the instance at requested position and returns the property with the selected instance. When the function does not succeed an exception is thrown.

There are two special positions in order to read the first and the last instance (FIRST_INSTANCE, LAST_INSTANCE), which allow accessing the first and the last instance in a collection. Reading the first or last instance can also be achieved by calling toTop() and previous().

Since the get() function returns the property with the selected instance, you may immediately access instance information.

Sometimes you may need to relocate the instance when terminating the function in order to restore the original state of the property handle. Therefore, you may store the position of the currently selected instance calling currentIndex() as in the example below. This is the only way restoring the selection by position, when a filter condition has been set.

Selecting an instance at absolute position becomes critical, when a filter condition has been set for the property (collection). In this case, it is a better way referring to relative position.

```
... :: function ( Property &persons ) {
  int        current = persons.currentIndex();
  // print PID of first person
  printf(persons.get(FIRST_INSTANCE).string("pid")); // same as
persons.get(0);
```

```
  // print PID of last person
  printf(persons.get(LAST_INSTANCE).string("pid"));

  // read last instance using previous()
  persons.toTop();
  persons.previous(); // last instance

  // relocate instance
  // should be the sam if nothing has changed in the collection
  persons.get(current);
}
```

Relative position

In order to read an instance by position from a collection with filter condition, getRelative() function is more appropriate. The getRelative() function returns the first instance fullfilling the filter condition.

Calling getRelative() function is sometimes critical, since you cannot trust the position returned fro the relativeIndex() function. The relative index is used for access optimization and usually differs when reading a collection backward or forward. Thus, getRelative() is good for locating a starting point, but not for relocating an instance. In order to reselect an instance, you should always use get().

```
... :: function ( Property &persons ) {
  int        current = persons.currentIndex();
  persons.setFilter("age >= 18");

  // print PID of first accepted person
  printf(persons.get(FIRST_INSTANCE).string("pid")); // same as
persons.get(0);
  // print PID of last accepted person
  printf(persons.get(LAST_INSTANCE).string("pid"));

  // read last instance using previous()
  persons.toTop();
  persons.previous(); // last instance

  // relocate instance
```

```
  // should be the sam if nothing has changed in the collection
  persons.get(current);
}
```

Scrolling

Scrolling throgh a collection is typical way for reading all or a number of subsequent instances from a collection. Usually, instances are selected one by one, in which case previous() and next() functions are the most appropriate. Since previous and next do skip instances not fulfilling a filter condition, thos functions work well for fitered and for unfiltered collections.

previous() and next() return false at end of collection. When calling previous() and next() for an unselected property handle, next() locates the first instance while previous() locates the last one. In order to unselect a collection properly, you may call the toTop() or cancel() function.

In order to go a number of instances forward or backward, you may call the position() function. position() scrolls forward in a collection the number of instances passed in count by skipping instances, which do not fulfill filter conditions.

In order to go backward in the collection, a negative value must be passed to the function. Thus, position(1) does the same as next() and position(-1) does the same as previous(). The difference is, that position() returns a property and throws an exception when not being succesful, while next() and previous() return false at end of collection.

```
... :: function ( Property &persons ) {
  int        current = persons.currentIndex();
  persons.toTop();
  while ( persons.next() )
    printf(persons.string("pid")); // same as persons.get(0);

  // relocate instance
  persons.get(current);
}
```

Accessing data by key

Reading data by key is a good mean in order to position the property on a certain instance, which can be identified by key. Keays are returned to the caller and passed to the functions in an internal Key format. A Key is a string, which contains key components separates by '|' or ';'.

The structure of a key depends on the sort order for the collection. When no sort order has been selected (unordered collection), instances are located by ident-key. When an unordered collections is not unique, it cannot be accessed by key.

Acessing data by key requires, that the application can provide a key for selecting an instance in a property handle. Therefore, several functions for accessing keys are provided. When a key became available, different functions can be used in order to select an instance for the key.

Since keys may consist of several key components, keyed access suppots key levels. A key level is the component number in the key, up to which key matching is requested.

Select order

When accessing a collection by position, instances are provided in order of the selected index (sort key). ODABA supports defining any number of persistent or transient indexes (orders) for extents and local collections in the data model. Typically, those index definitions are referred to as sort orders for a collection.

In order to select a sort order that differs from the default order for the collection (main order), the setOrder() function can be called with an appropriate key name.

One may also define an ad-hoc order, which implicitly creates a view with the defined order key. An ad-hoc order contains a list of attributes or attribute definitions rather than a sort key name.

```
Property      persons(obh,"Persons",PI_Read);
persons.setOrder("ik_Name");                    // schema order
persons.SetOrder("name; age=birthdate-Year()"); // ad-hoc order
```

Provide key value

Using instance filters

In order to reduce the instances displayed for a collection, property handles provide two ways of filtering instances in a collection. One way is by setting an OSI expression as filter condition for the property handle. The other possibility is to set an instance or key to hidden when reading an instance, i.e. using the doAfterRead() handler for filtering instances from the collection.

When a filter has been set, the property handle selects only those instances that return true (-> isTrue()) for the expression. Sequential retrievals as nextKey(), next(), previous() or position() automatically search for the next valid instance that fulfills the filter condition. locateKey() will return an error (not found) when the instance does not fulfil the selection criteria. firstKey() and nextKey() will skip non-selected instances as well.

The get() function, which is requesting a specific instance by index or key, throws an exception when the requested instance does not fulfill the filter condition.

When setting a filter for an update or write property handle, updating an instance may lead to an invalid instance (i.e. the instance is not fulfilling the defined condition anymore). In this case, the instance is unselected after storing the updated data.

Setting a filter condition slows down key operations as e.g. nextKey(), since the instance must be read in order to check the condition. When a condition is based on key component attributes, only, performance can be improved by calling setKeyFilter() instead. Since instance reading is much more expensive than key reading, filtering by key value is much more efficient than filtering by instance.

Filter by OSI expression

The filter conditions (OSI expressions) can be set in the filter() property. The set and get functions of the property allow changing and reading the filter condition. You may expand a filter condition by calling expandFilter(). hasFilter() can be called in order to check, whether a filter condition had been defined or not.

```
// OSI: access path with selection condition
set<Person>       adults = Person.Where{age > 18};
while ( adults.next )
  displayPerson(adults);

// C++. C# ...
Property  adults(database,"Person",PI_Read);
adults.SetFilter("age > 18");
while ( adults.next() )
```

```
  displayPerson(adults);
```

Get filtered count

The count() function does not reflect the selection condition, i.e. it returns always the total number of instances in the collection, independent on a filter condition set for the collection.The number of instances according to the filter set in the property handle can be retrieved using the relativeCount() function.

```
if ( adults.relativeCount() > 0 )
  Message("Person has grown up children");
```

Referring to filtered instances by position

Referring to an instance by position or key may fail, since the selected instance may not fulfil the selection condition. In this case, the function call fails. In order to access instances by bosition although a filter has been set, getRelative() might be called.

```
// C++. C# ...
Property  adults(database,"Person",PI_Read);
int       indx0 = 0;


adults.SetFilter("age > 18");
while ( adults.getRelative(indx0++) )
  displayPerson(adults);
```

Context selection

Even though, filter expressions can be as complex as possible, filtering may depend on other variables, e.g. run-time variables. In this case, filtering can be done esier by using context class functions. In order to support filtering based on key data, the doBeforeRead() handler can be used. Instance filtering should be done in the doAfterRead() handler.

Key filter

Context selection is based on the doBeforeRead() handler that handles the DBP_Read event. This event isgenerated always, when an instance is going to be read, i.e. after the instance has been located in the collection. The event is also generated when using key access methods as nextKey() or locateKey().

For ordered collections (when a sort order is selected for the property handle) the key is available as well as the identity of the instance. The doBeforeRead() handler can be overloaded to supress instances, that do not follow certain conditions. Since instance information is not guaranteed, the check can be made based on data of the key instance or the identity, only.

```
// accept persons with names lower than 'P'
```

```
bool   sPerson:: doBeforeRead() {

  Property       *ph = property();

  return( ( ph.getKey("name").string() <= 'P' ? true : false);
//true for accept

}
```

Instance filter

In order to check onstance filter conditions, the doAfterRead() handler must be used, which handles theDBO_Read event. This event is generated after reading or re-reading instances. When the handler has been called, instance data can be accessed. Since the doAfterRead() handler is an "after" event, which cannot influence the process logic, it must hide the instance explicitly.

Hiding the instance will suppress the selection of the instance when trying to read it with get(). When using position(), next() or previous(), hidden instances are skipped.

The ?hidden? state is automatically reset when the selection in the property handle changes the selection. It is, however, also possible to reset the ?hidden? state using the context function showInstance(). The ?hidden? state is inherited to derived structure instances.

```
// accept adult persons, only (age >= 18)

bool   sPerson:: doBeforeRead() {

  if ( property("age").integer() < 18 )

    HideInstance();}

  return ( true );

}
```

Key check

Sometimes, a filter condition is based on properties, which are all key components. Unfortunately, key functions as NextKey() or FirstKey() will normally read the instance before checking the condition, i.e. key access will not be as efficient as without selection.

When reading keys from the database and a filter is set, the instance is read in order to check whether the key refers to an accepted instance or not.

In order to provide fast selection for key filter, key check can be enabled, e.g. after setting the filter condition for the property handle by calling the enableKeyCheck() property handle function. When key check is enabled for the property handle,key functions will check the selection condition based on the key selected for the property handle, only. When the expression refers to non-key properties, the initialised instance values are used for evaluation.

```
// OSI: access path with selection condition
```

```
set<Person> lowCaseName = Person.OrderBy("sk_Name").Where{name >=
'a'};
lowCaseName.enableKeyCheck;
while ( lowCaseName.nextKey )  // filter by key access
  displayPerson(lowCaseName);

// C++. C# ...
Property  lowCaseName(database,"Person",PI_Read);
lowCaseName.enableKeyCheck();
lowCaseName.setOrder("sk_Name");
lowCaseName.SetFilter("name >= 'a'");
while ( lowCaseName.nextKey() ) // filter by key access
  displayPerson(lowCaseName);
```

Common properties on instance level

Beside properties defined in the data model, ODABA supports several instance properties, which are available vie instance descriptors maintained by ODABA for each instance. Many of those propertie are available by property functions, but sometimes it is more comfrtable accesing them via property names.

__IDENTITY, __LOID

Each object instance contains the local unique identifier for the instance, which is provided passing this attribute. The value can also be provided calling the loid() function.

```
  Property      persons(obh,"Persons",PI_Read);
  Property      p_loid(persons,"__LOID");
  int64         loid;

  persons.get("00001");
  loid = persons.loid();  // same as: p_loid.bigInteger()
                          // or:      persons.bigInteger("__LOID")
```

__GUID

The property contains the global unique identifier for an instance. In order to assign a global unique identity to an instance, the structure definition in the data

model must have switched the guid option on. Moreover, the guid option must be set for the collection owning the instance. When the instance is not equipped with global identifiers the local identifier within the database is returned.

Global identifiers differ from local once syntanctically, since they begin with an one diget version number and a dash, always (e.g. '1-...').

Global identifiers can also be read calling the guid() function.

```
Property      persons(obh,"Persons",PI_Read);
Property      p_guid(persons,"__GUID");
String        guid;


persons.get("00001");
guid = persons.guid();  // same as: p_guid.string()
                        // or:      persons.string("__GUID")
```

__SORTKEY, __SORTKEY_STRING

Referring to __SORTKEY_STRING will return the key value for the instance. The key can be used for displaying or locating an instance in the collection.

Used with the ODABA API, both properties return a key string. In the system API, however, __SORTKEY returns the internal key, while __SORTKEY_STRING returns the key value converted into string format.

Instead of referring to the __SORTKEY attribute, you may also call the sortKey() function.

```
Property      persons(obh,"Persons",PI_Read);
Property      p_sortkey(persons,"__SORTKEY");
Key           sortkey;


persons.get(0);
key = persons.sortKey();  // same as: p_sortkey.string()
                          // or:      persons.string("__SORTKEY")
```

__TYPE

The type property may change also for insttances in a collection (weak-typed or untyped collections). In order to retrieve the type of the selected instance, one may refer to the __TYPE property or call the currentType() function.

```
Property      persons(obh,"Persons",PI_Read);
```

```
  Property      p_type(persons,"__Type");
  String        typeName;

  persons.get(0);
  key = persons.currentType();  // same as: p_type.string()
                                // or:
persons.string("__Type")
```

Write to database

Create object instancess

Update object instances

Rename or duplicate object instances

Remove or delete object instances

Closing property handles

Property handles are closed automatically, when being destroyed. Property handles are also closed automatically, when opening another property with the same property handle. Sometimes, it makes also sense to close the property handle explicitly calling the Close() function.

Closing a property handle will reduce the reference count in the property node referred to by the property handle. When the reference count becomes 0, the property handle node will be deleted. When destroying a property node, which refers to modified instance data, the instance modifications are stored to the database before closing the property node.

Destroying a property node may have a number of unexpected side effects, since deleting a property node will destroy also all its copies and its children.

I.e. destroying a property node will delete all property nodes created implicitly for the node instance (instance property nodes). Instance property nodes are not deleted, as long as the node instance exists and the instance exists as long as the property node exists.

For maintaining property node copies properly in a property node hierarchy, copies for property nodes require a reference to their origin. Hence, closing a property node will also destroy all copies.

Destroying an property node will automatically close all property handles referring to it. Thus, Property handles may become inaccessible, because of being closed implicitly. Implicitly destroyed property nodes are destroyed bottom-up, always.

Usually property handles should be closed by the application (explicitly or by destroying the property handle object). When, for some reason, property nodes remain unclosed, the database will close at least all property handles referring to persistent properties before closing the database.

```
void CloseHandle ( PropertyHandle &ph ) {
  ph.Close();
}
```

Data conversion

ODABA provides builtin data conversion support. Most types of conversion between elementary data types are supported, but some conversions may result in empty values.

Converting values behave differently depending on the value type for source (right) and target (left) operand. Thus, elementary values might be vonverted into elementary values, but also into complex values or arrays. Reversely, arrays or complex values may be converted into elementary values.

Conversion is considered as a feture between attribute and complex values, but not for collection values. Builtin conversion functions will ignore collection properties, when being part of a value.

Most of elementary data type can be converted into each other. The following list shows the unsupported conversions. Unsupported conversions will signal an error or may throw an exception.

From

To

int

MEMO, DateTime

bool

MEMO, DateTime, Time, Date

Date

MEMO, Time, double

double

MEMO, DateTime, Time, Date

string

-

Time

MEMO, Date, double

DateTime

MEMO, int, double

Special handling is required for BLOB values, which can be converted into BLOB values, only.

Converting to elementary value

Practically, there exist two essential elementary data types: string and number. Considering all numeric types, bit, bool, Date, Time and enumerated values as numbers, remaining types (except BLOB) can be considered as string types. DateTime values are not considered as elementary value.

String to number conversion is supported as well as number to string conversion. Converting data may result in data truncation or data loss. In order to check compatibility, corresponding functions might be called explicitely from within te application.

When the target value has a restricted valu domain (enumerated type value, date, time), invalid values will cause an error or throw an exception. Conversion errors are always signaled, when trying to convert a BLOB value in anything else or reverse.

String conversion

Converting strings into strings or numbers depends on the data type for the target value. In most cases, string conversion will succeed, but the result might be truncated in the one or the other way.

To number

Converting strings into numbers accepts also strings not containing valid numbers. Strings will be converted into numbers as far as possible. Thus, a string value "abc" results in an empty value (0), when being converted, but does not cause an error or throws an exception. Similar, a string value "12abc45" would result in 12 ignoring follwoing "abc45" characters.

To string

Converting string values to string values will truncate the right operand (source), when the size of the target is smaller than the source size, e.g. assigning a string value "abcde" to a three character string will result in "abc".

When the target string zize is larger than the source, the target will be padded with spaces (' ') 0r 0 demending on the string type (0-terminated or buffer).

To Boolean

Converting a string to a bBoolean value checks, whether the string content corresponds to a true value or not. When the string has one of the following values, it is considered as true, otherwise as false:

ttrue, t, yes, y, 1

These values are not case sensitive. The string may contain trailing blanks, but no other trailing characters. Also, no spaces must preceed the value.

```
'TRUE'   --> true
```

```
'TRUE'   --> false
'TRUE  ' --> true
'  TRUE' --> false
```

To enumeration

Converting a string into an enumeration, the string value passed in the right operand (source) must match eactly (case sensitive comparision) the name of an enumerator defined in the enumeration assigned to the target value type. When no match could be found, conversion results in an error or throws an exception.

To date

Converting string to date requires a valid date string. Valid date strings comsist of year (2 0 4 characters), month (1 or 2 characters) and day (1 or 2 characters). Different orders of year (Y), month (M) and day (D) are supported:

dd.m.y (German)

mm/d/y (English)

yy-m-d (default)

The separator choosen determins the position of year, month and data in the string. When the values passed between separatord do not correspond to valid day, month or year numbers, the conversion results in an empty date value.

To time

Converting string to time requires a valid time string. Valid time strings comsist of hours, miniutes, seconds and hundreth seconds. One string format is supported for time values:

hhours [:minutes [:seconds [,hseconds]]]

Minutes, seconds and hundreth seconds are optional. When the values passed between separatord do not correspond to valid values, the conversion results in an empty time value.

Number conversions

There are no problems converting any type of numbers to string values. When the size of the string value is to small for storing the required value, it will be truncated. Also, most number to number conversions will perform well, but may result in truncation.

Tu numerical

Converting a numerical value into number returns the numerical value of the source (right) operand. date and time result in an integer value (number of days or hundredth seconds). Prceision will be adjusted to the format of the targer value. When the target value size is exceeded, the maximum or minimum value will be

returned. Thus, assigning a numerical value 1234.56 to an INT(3,1) value will result in 999.9.

Converting an enumerated type value into a number will return the assigned code value for the enumerator.

To string

Converting a numerical value to string returns the corresponding string representation including decimal points and thousands separator according to the target value definition. When the value exceeds the string size, the string is filled with *, but no error is signaled.

Boolean values are converted into Y and N. Enumerated values (code) are converted into enumerator names, when the numerical value passed is a valid cade in the enumeration.

Date and time values are returned as yy-mm-dd or hh:mm:ss,hs strings. In order to frovide other formats, special date/time formats might be set.

To Boolean

Converting a number to a boolean value results in false, when the value is 0 and in true otherwise. Date and time values result in false, when they are empty and in true otherwise.

To enumeration

Converting a number into an enumated type value will check the value before assigning it. When the value is not a valid value in the enumeration, conversion signals an error.

To date

Converting a numerical value into a date value assigns the number to the date value, wich is interpreted as number of days since January 1st 1870. When the value is less than 0, conversion signals an error.

To time

Converting a numerical value into a time value assigns the number to the time value, wich is interpreted as number of hundredth seconds. When the value is less than -1 (empty time value), conversion signals an error.

Complex data type conversion

Assigning a complex value or array to an elementary value, usually assigns the first attribute of the array or complex value. When assigning an array of complex values, the first attribute value of the first array element will be assigned.

When, however, the target (left operand) is a string value, the source value (parameter or right operansd) is converted into an ESDF string, wich is able to carry compley values in string format. The ESDF format is a positioned value format that can store hierarchical data. It does, however, not carry schema information as at-

tribute names, which correspond to the complex data type of the source operand, in this case.

Convert data to complex value or array

When assigning a value to an array, as many values as defined in the right operand will be assigned to the array. Remaining array elements for the left operand will be initialized. Remaining elements for the right operand are ignored. When assigning an elementary value to a complex value, the first attribute of the complex data type is filled, while remaining attributes will be initialized.

Number conversion

When converting an elementary value (number) to an array or complex value, the value is assigned to the first array element or attribute in the complex value. In case of an array of complex values, the value is assigned to the first attribute of the first array element. All other elementary valus of the target value will be initialized.

String conversion

Strings may contain complex values in ESDF format. When assigning an elementary string value to a complex or array value, only the first elementary wvalue will be set to the value passed in the string. Remaining array values or values in a complex value will be initialized.

When passing a complex value in the string, the value is interpreted according to the type of the target value, i.e. value elements in the string are assigned to array elements, when the traget of conversion is an array or to attribute values, when the target is a complex value. Each sub-assignment follows the conversion rules described so far, i.e. the complexity of values that can be converted is not limited.

Converting complex values

Converting complex value to complex value means converting a complex attribute to another complex attribute. In this case. metadata is available and values are assigned by name. Values appering in the source operand (perameter or right hand operand) but not in the target, will be ignored (without warning). Values in defined in the target, but not in the source, will be initialized in the target. Values for attributes defined in both are converted according to the rules described so far.

1.3.4 Advanced property handles

The intension and extension managed by a property handle is defined in terms of access paths in many cases. An access path can be seen as an extension of an OQL query statement (like the SQL statement for relational databases). In fact, the OQL statement is only one posibility of defining an access path.

Simple property handles are based on property references, i.e. simple property or extent names, which are specific access paths as well. Besides, ODABA provides advanced property handles for different purposes, which can be constructed by means of more complex access paths. Similar to basic property handles, advanced property handles are used to manage collections, instances or elementary values, except, that those need not to refer to persistent data.

Other advanced property handles, which are not based on access paths, can be created in order to manage temporary or transient collections (transient properties) created by the application or defined in the data model.

In may cases, advanced properties just provide a sort of short cut for accessing data (property path and path property). Other access paths allow defining complex operations on data (views) or managing transient result instances or collections.

The functionality for advanced property handles is the same as for basic property handles. The essential difference is the data (collection) managed by the property handle.

Different types and functions of access paths

Access paths can be referenced in OSI expressions or as data source for property handles. Since access paths can be defined in many different ways, the following sections will explain a bit more detailled, how access paths can be used to achieve certain results.

Access paths can be devided in subsequent types:

property reference - directly refers to an extent or property in a structure instance (by name)

property path - refers to members wirhin a structure instance

path properties - define iteration paths by traversing relationships or references

operation paths - contain one or more operations in the path

view path - includes set operations as product or join and are, hence, an extension of OQL queries.

An access path performs mainly two essential functions:

execute - evaluate the result from the path definition

access - selecting and providing instances from the collection defined by the path

In an OSI expression the system decides, when to execute and when just to access the path. When using an access path as data source for a property handle, the application can decide, when to execute the path and when accessing it.

Simple access paths as property references or property paths can be called for execution but will not do anything. More complex path as operation or view path may call execution also while accessing.

An access path consists of various elements (Access Path Elements), which determin the behaviour of the path.

Local and static access paths

Staic access path are those, beginning with an extent name (or global variable). Since static access path do not have parents, they cannot change parent and need to be executed only once (e.g. in case of operation path). Nevertheless, the global collection may change and the result must be recalculated. This cannot be done by the system and must be triggered by the application.

Local access path are those, that have a parent property (collection or instance). In this case, the value of the access path depends on the instance selected in the parent property handle. When no instance is selected, the value of the access path is NULL, i.e. the data set is not only empty. You cannot even ask for the number of instances for the access path. Each time, when another instance in the parent property is selected, the path valu changes. In case of operation or view paths this requires recalculating the result, i.e. the path must be executed. This is done automatically, when trying to access data from the path data set. The application may, however, explicitly execute the path. When explicitly executing an operation path, the path is executed regardless wether a new instance has been selected in the parent or not. This enables the application to recalculate the path in order to reflect changes in the path for the currently selected parent instance.

Access via property reference

A property reference is a simple access path containing just an elementsry property name referring to a property in structure. Opening a property handle with property reference directly provides the property handle from the instance or a copy of it. The data set managed by the property handle is the data set stored for the property instance in the currently selected structure instance.

```
// access to city via basic property handles

  Property     persons(obh, "Persons", PI_Read);

  Property     address(&persons, "address");      // original
instance property handle

  Property     city(&address, "city");            // original
instance property handle

  Property     children(person, "children");      // copy of
instance property handle


// PH-Macro: the short way of definition

  PH(&persons,address)

  PH(        &address,city)

  PH(persons,children)
```

Access via property paths

A property path is a generic way of accessing properties, which are not directly defined as member in the type definition for the instances managed by the property handle.

A property path may refer to a reference or relationship property at the end (last path element), but also at the beginning or in the middle of the path definition. In contrast to a path property, however, the property path does not contain navigation or selection elements, i.e. it contains property names, only, separated by dots (.).

Usually, property paths do apply on properties, only, which are part of the data type managed by the property handle.

Typical example is the address defined as a member of person, which is a structured attribute. The two examples below illustrate the difference between using property references and property paths and make clear, that a property path ia just an abbreviation for a list of property references.

Property paths do not need to be executed. The result is represented by the last property referenced in the paroperty path.

```
// access to city via property references
  Property     persons(obh, "Persons", PI_Read);
  Property     address(&persons, "address");
  Property     city(&address, "city");


// property path to city
  Property     persons(obh, "Persons", PI_Read);
  Property     city(&persons, "address.city");


// osi examples
... Person::Test()
{
  ::message(address.city);            // property path
  address.{ ::message(city); };       // property references
}
```

Switchable property path

A property path containing reference or relationship properties in at least one path element, which is not the last one, the path can be switched by the axpression or application.

In the example below, the property path children.age returns a different value for each iteration depending on the person selected in the children collection.

```
bool Person::HasGrownupChildren()
{
  children.ToTop;
  while ( children.Next )
    if ( children.age >= 18 ) // age changes for each selected
child
      return(true);

  return(false);
}
```

Switching path

When a property path does not start with an extent node or global variable, the path value depends on the instance selected in the parent property. Thus, in the example below, the city name will change for each person selected in the person collection.

```
collection void Person::Streets()
{
  ToTop;
  while ( Next )
    ::Message(name + ": " + address.city);
}
```

Path properties

In contrast to property paths, path properties allow navigating through a defined property hierarchy. Syntactically, a property path differs from a path property by containing at least one navigation element ([]- or ()-operator).

The ()-operator is the selector operator, which indicates either an iteration element or an instance selection. The []-operator is the provide operator, which selects or creates an instance when not yet existing.

Depending on the selector and provide operators, a path property refers to a collection of instances of the type being defined in the last path element, i.e. a path property can be seen as the collection of all instances one receives when iterating through the property hierarchy, which is a sort of specific union operation.

In general, you can express each path property by a sequence of property references and appropriate function calls for instance selection and iteration. But the path property is a better way in many cases, since it prevents you from programming errors, is easier to write and more transparent for other readers. Path properties are typically used in OSI expressions. Using path properties in application programs is more a matter of taste.

```
// access to city via path proprty
void test(ObjectSpace &obh)
  Property     cities(obh, "Persons().address(0).city", PI_Read);
  while ( cities.next() )
    SystemClass::message(cities);
}
// osi example
void main()
{
VARIABLES
  set<STRING>    &cities = Persons().address(0).city;

  while ( cities.next );            // property path
    message(cities);
}
```

Access path elements

An access path may contain different kind of elements, which differ syntactically. In oder to iterate through a collection or selecting a specific collection instance, following element types are available:

reference

iterator

selector

instance provider

(operation)

Operation elements cannot be recognized syntactically, but usually, the user can recognize the difference between an operation and a property in the path by naming konventions.

1.3.4.1.1.1 Reference element

A reference element just consists of a property (or variable) name: Reference elements can be used at any position in a path.

<persons.children.children

Usually, reference elements will noct change selection and do refer to the currently selected instance in the path. Especially in OSI expressions, this is used to selects an instance in a collection and referring to it in the next statement(s).

The application is responsible to select propert instances in reference elements, because otherwise, the path returns nothing.

As long as the path uses reference elements, those will refer to the pre-selected instances. When the path, however, contains an iteration or selector element, subsequent reference elements do not have the same selection and are known in the context of the path, only. Thus,

<persons.children.children

and

<persons.children().children

refer to the same selection in person, but children() usually has a different selection from children and so do all subsequent reference elements.

```
  person.Get(0);        // selects the first person in the
collection

  Message(person.name); // prints the name of the selected person

  if ( person.children.Next ) // selects the next child for the
selected pwerson
```

```
    Message(person.children.name);     // prints the name of the
selected child
```

1.3.4.1.1.2 Iterator element

Iterator elements are property references followed by an empty parameter list: Iterator elements are used to indicate the iteration level:

<persons.children().children()

This path will iterate through all grand children of the person selected in the persons property. It will not iterate through the persons collection, since persons does not define the selector operator. When, however, defining the path as follows:

<persons().children().children()

The path provides all grand children for all persons. Iterator elements in a path are just an abbreviation. Instead using iteration path, one could always write a nested loop, e.g.

while (persons.Next)

while (persons.children.Next)

while (persons.children.children.Next)

In many cases, it is, however, much easier to refer to a path.

```
//prints name and first name for all grand children
//duplicates may appear, since a person may be grand
//child of maximum four grand parents.

// using path property
  PropertyHandle          grand_children(obh,
"Persons().children().children()", PI_Read);

  while ( grand_children.Next() )
    printf("Name: %s, first name: %s",
           grand_children.GetString("name"),
           grand_children.GetString("first_name"));

// using property references
// produces the same same result as the example above
  PropertyHandle          persons(obh, "Persons", PI_Read);
  PropertyHandle          children(&persons, "children");
```

```
  PropertyHandle          grand_children(&children, "children");


  while ( persons.Next() )
    while ( children.Next() )
      while ( grand_children.Next() )
        printf("Name: %s, first name: %s",
               grand_children.GetString("name"),
               grand_children.GetString("first_name"));


// OSI expression
void Person::Test()
{
VARIABLES
  set<Person>  &grand_children &= Persons().children().children();
PROCESS
  while ( grand_children.Next )
    ::Message("Name: " + name + ", first name: " + first_name);
}
```

1.3.4.1.1.3 Selector operator in a path property

Selector elements are property names followed by a selector parameter. The selector operator can be used for selecting a specific instance in a path element:

<persons("P0001").children().children()

In this example, the path returns all the grand children for the person with the person number (pid) P0001. With the selector operator, you may select a single instance in the path by posoition or constant or variable value, but you cannot define a filter condition. For defining filters, you may use an operation path.

The path will never iterate for selector elements, defining a selector element after an iteration element, iterating through the path automatically selects the instance in the selector element, when existing.

<persons().children(0).name

When not existing (currently selected person does not have a child), the next person will be selected, i.e. the path above returns the name for the first child of all persons havin children.

Instance selection is possible by key value or position or by an expression, in which case the value returned from the expression is interpreted as key value. The last case is an interesting feature, when using path properties in OSI expressions.

1.3.4.1.1.4 Selector operator in a path property

The provide operator guarantees, that the requested instance will be provided for the referenced property. I.e., an instance with the defined key will be created, when not yet existing.

<Persons['P00001"].children

will provide a person with the person number (pid) P00001, i.e. it will be read or created, when not yet existing.

Typically, the provide operator is used for single reference or relationship properties in order to create required instances on the fly. Considering the development resource instances, which are linked to a documentation topic (DSC_Topic) via a resource reference (SDB_ResourceRef), you may want to generate titles for functions. Resource references and documentation topics are created on demand, i.e. you can never be sure, that they already exist. The example below shows, how this can be handled simply by using the provide operator.

The example automatically creates resource reference and topic on the path, when titles do not yet exist or are empty (!title).

```
// using path property
void UpdateTitle(PropertyHandle &class_ph) {
  PropertyHandle     functions(&class_ph, "pfunctions");
  PropertyHandle     title(&functions,
"resource_ref[0].topic[0].definition.definition.name");


  functions.ToTop();
  while ( functions.Next() )
    // assign function name to title if title is empty
    if ( !title )
      title = functions.GetString("sys_ident");
}


// using base property hierarchy -
// produces the same same result as the example above
void UpdateTitle(PropertyHandle &class_ph) {
  PropertyHandle     functions(&class_ph, "pfunctions");
```

```
  PropertyHandle     resource_ref(&functions, "resource_ref");
  PropertyHandle     topic(&resource_ref, "topic");
  PropertyHandle     title(&topic, "definition.definition.name");

  functions.ToTop();
  while ( functions.Next() )
    if ( !resource_ref.Provide(0) )
      if ( !topic.Provide(0) )
        // assign function name to title if title is empty
        if ( !title.IsTrue() )
          title = functions.GetString("sys_ident");
}
```

1.3.4.1.1.5 Special behavior of path properties

In general, property handles for path properties behave the same way as basic property handles, i.e. they provide the same functionality. However, when using path properties, most of the functionality is delegated to the last element in the path property. Meta-information (type, dimension, size etc) also refers to the metadata for the last element.

In general, path properties should be used when sequential iteration is required or when information from related instances needs to be retrieved, which is connected to the current instance via a singular reference/relationship path.

As other property handles, the path property can be positioned or counted. Nevertheless, path property should be used mainly for sequential access, since positioning the collection to a certain position may take a while (even though, the property node always tries to find the shortest way from the current position). Counting the elements means adding the counts for the elements for the last path property element. In order to maintain the current position, a copy of the path is provided for counting.

You may also try to locate an instance by key (according to the sort key currently set for the last path property element by default or by ChangeOrder()). Locating an instance by key value, however, also tries to locate an instance with the requested key in all last element collections until it finds one.

Selections and instance event handler can be set for a path property, but those only receive events from the last path element.

Access via operation path

The operation path is any type of access path, which contains at least one operation or inline expression besides property references. In contrast to properties, operations are functions or expressions referred to in a path. Even though, transient properties are operational, the are considered as properties and behave as such. Thus, special behavior of an operation path is resulting from the fact, that the path contains at least one operation.

Operations included in a path definition can be of different type. Besides traditional query operations as SELECT, FROM, WHERE etc. additional query operations as INTERSECT or UNION and property handle functions as GetCount can be referenced in anoperation path. Also user defined expressions or context and interface class functions may appear in an operation path.

Usually, an operation path creates a virtual result set. The result set can be accessed by property handle functions (e.g. Get()). Since virtual collections can be processed sequentially, only

A view path is a path, which refers to a predefined view in the database schema or to traditional or extended query operations. ODABA supports view schemas as well as view collections. A view schema defines a transient structure and the sources for all its properties. Property sources can be defined in a structure context, i.e. they may refer to properties defined in the structure the view schema is based on. A view schema based on a structure is considered as method of the class formed by the structure. Thus, a view schema can apply to any structure instance of the given type. Applying a view schema to a structure instance is possible in an access path or by defining an appropriate property handle.

Persons().USE('PView')

In the path above, the view schema PView applies on all person instances of the person collection. In a way, USE replaces the SELECT clause in an OQL/SQL query, but predefined views provide more features for evaluating view properties.

View properties or transient properties can be defined as static or local properties. As local properties, they will be associated with a structure, but they are not considered as structure methods but as (transient) properties of the data type. Since evaluating transient properties is a performance critical issue, ODABA evaluates transient properties only, when trying to access transient property data. The application may, however, call the Execute() function in order to re-evaluate the transient property at any time.

Whe the parent changes for a local view property, it will be re-evaluated automatically after the next instance is selected in the parent property, when data is requested from the transient property.

In most cases, views are used for reading data. Sometimes, it might, however, be useful to update view properties and write them back to its origin. This is possible

only, as long as the view property receives data directly (1:1 relation) from a persistent source property.

1.3.4.1.1.6 Access via view definitions

A specific subset of operation paths are view paths. A view path is a path, which refers to a predefined view in the database schema. ODABA supports view schemas as well as view collections. A view schema defines a transient structure and the sources for all its properties. Property sources can be defined in a structure context, i.e. they may refer to properties defined in the structure the view schema is based on. A view schema based on a structure is considered as method of the class formed by the structure. Thus, a view schema can apply to any structure instance of the given type. Applying a view schema to a structure instance is possible in an access path or by defining an appropriate property handle.

Persons().USE('PView')

In the path above, the view schema PView applies on all person instances of the person collection. In a way, USE replaces the SELECT clause in an OQL/SQL query, but predefined views provide more features for evaluating view properties.

View properties or transient properties can be defined as static or local properties. As local properties, they will be associated with a structure, but they are not considered as structure methods but as (transient) properties of the data type. Since evaluating transient properties is a performance critical issue, ODABA evaluates transient properties only, when trying to access transient property data. The application may, however, call the Execute() function in order to re-evaluate the transient property at any time.

Whe the parent changes for a local view property, it will be re-evaluated automatically after the next instance is selected in the parent property, when data is requested from the transient property.

In most cases, views are used for reading data. Sometimes, it might, however, be useful to update view properties and write them back to its origin. This is possible only, as long as the view property receives data directly (1:1 relation) from a persistent source property.

1.3.4.1.1.7 Query operations

A view path path is an operation path referring to several built-in query operations. Built-in query operations are all operations, which are allowed in an SQL/OQL SELECT statement including some ODABA specific extensions. SELECT is considered as query operation as well.

Syntactically, built-in query operations sometimes differ from other operation calls in more complex parameter lists (as e.g. SELECT or GROUP_BY). Therefore, you should check the reference manuals (OSI Language Reference) carefully.

ODABA supports view definition as view path but also as classical SELECT statement. Using access path provides, however, more flexibility. The general structure of a SELECT statement would be like:

SELECT (assignment rules)

FFROM (source)

WWHERE (pre-condition)

OORDER_BY (key definition)

GGROUP_BY (grouping rules)

HHAVING (post-condition);

HAVING and WHERE do both expressing a filter condition in the context of the preceding operand. Hence, in a view path you may use WHERE always instead of HAVING. Instead of SELECT, the view path may refer to a predefined view via USE. Finally, the FROM clause is typically replaced by an access path (e.g. property reference), which provides much more flexibility. For multiple set operations, the FROM operator acts like a set product operation. In a view path, the query specification looks differently, since the view path (operation path) implies a processing order from left to right:

FFROM(source).ORDER_BY(key definition).SELECT(assignment rules).

WWHERE(pre-condition).GROUP_BY(grouping-rules).WHERE(post-condition);

In addition, view paths definitions support several built-in query functions:

<JOIN - Join two collections

<UNION - Union two collections

<INTERSECT - Intersect two collections

<TO_FILE - Redirect output to file

TTO_DATABASE - Store output to database

<FROM_FILE - Redirect input to file

<USE - Predefined assignment rules (replaces SELECT)

Besides the view operations mentioned above, other operations or properties might be referenced within the path. Each path element represents an independent generic operation, which can be called with any collection. As long as the result of a path element is a collection, you may extent the view path by further elements.

Since view paths is a sequence of set or OQL/SQL operations, the view path is just a special operation path. In other words, each operation path may contain any number of view operations.

1.3.4.1.1.8 Set operations

Usually, the view source plays a role in traditional SELECT statements, only. Here an explicite source definition is refquired using the FROM clause.

FFROM(source)

In an operation path, usually the source collection is sufficient, i.e. the operation paths

FFROM(source).Where(..). ... and

source.Where(..). ...

are identical, as long as the FROM operation refers to only one operand. The source operand might by a simple property name, but also an access path or an inline expression, i.e. any type of syntactical operand, that returns a collection.

Compared with traditional OQL/SQL statements, the source definition in an OSI access path provides a number of additional features. In general, eac operation path can be referred to as source, but there are specific operations, that might be referred to as source in particular:

external files

union collections - result of Union operation

join collections - result of Join operations

intersect collections - result of Intersect operation

minus collections - result of Minus operation

product collection - result of From operation

External file source

The ExternExtent operation allows assigning different kinds of external files as source for an operation. External files usually do have an file location and a file definition (file and exchange scheme), which might be part of the file definition.

The extern Extent operation differs slightly from the FromFile operation, which im-mrdiatly imports the external file into the callection os the calling property (collection). The FromFile operation returns the calling property, i.e. the collection with the imported instances.

```
// access file
FileExtent( Path='c:/temp/my_persons.xml', FileType='xml',
            Definition='c:/temp/my_persons.def', Headline = false)
// import file
```

```
FromFile( Path='c:/temp/my_persons.xml', FileType='xml',
          Definition='c:/temp/my_persons.def', Headline = false)
```

Union operation

The union operation allows merging two collections. There are two ways of defining a union operation. Calling Union as

AA.Union(B)

Unites A with B and stores the result in A. Calling union, however, as

UUnion(A,B)

Creates a new collection containing the union of A and B.

Minus operatopn

The minus operation removes instances in the second operand collection from the first operand collection. There are two ways of defining a minus operation. Calling Minus like

AA.Minus(B)

Creates the difference between A and B and stores the result in A. Calling minus, however, as

MMinus(A,B)

Creates a new collection containing the difference of A and B.

Intersect operation

The intersect operation creates a collection containing all instances from the first and from the second collection. Calling Intersect like

AA.intersect(B)

intersects A with B and stores the result in A. Calling Intersect, however, as

IIntersect(A,B)

creates a new collection containing the intersection of A and B.

Join operation

The join operation flattens a path, i.e. it creates an instance containing all collection instances participating in the path. The join operation requires a properties, wich are elements of a property path as parameter, i.e. each operand must be a valid operation path for its predecessor.

In order to avoid naming conflicts, join operands can be assigned to target property names. Named parameters must also be used in order to assign expressions or access paths to a join operand. The result instance contains then one attribute instance for each parameter with the property name assigned in the parameter specification.

Passing other collection expressions as an access paths or an extent, the operation will return the collection passed to the operation without operating on it.

The result of a join operation contains all instances by iterating the property path. In the example below, the result contains all grand children, but each instance consists of a three person instances.

```
Join(Persons, childeren, grand_children=children)
```

Product operation

There is no specific product operation, but in order to create set products, the From operation can be used by passing any number of collections to the operation.

FFrom(Persons, Companies)

The product or FROM operation creates a collrction that contains any possible combination of instances from its operands. It creates an instance containing all collection instances participating in the operation. Each operand in a From operation must be a valid operand in the context of the calling operand or a static expression.

In order to avoid naming conflicts, From operands can be assigned to target property names. Named parameters must also be used in order to assign expressions or access paths to a From operand. The result instance inherits then from all instances involved in the operation. , In case of name ambiguity, you may refer to bas instances similar as to imbedded attributes since ODABA supports named base structures.

```
From( men = Person.Where(sex=='male'),
woman=Person.Where(sex=='female'))
```

1.3.4.1.1.9 View properties

From a view source, any number of view properties can be derived. View properties can be defined in a view schema, which can be referenced in the USE opration:

Persons().USE('PView')

where the view schema (PView) applies on all instances in the view source. Thus, USE replaces the SELECT clause in an OQL/SQL query. The SELECT clause is another way of defining view properties in an ad-hoc query:

Persons().DEFINE(name, address, full_name = getFullName)

View propertiey might be simple attributes, complex data type instances or collections. When not passing a calculation rule, the rule property is taken as being defined under tha same property name in the source. One mya, al-

so pass calculation expressions as operands, expressoions or expression calls defined in the context of the view source instances.

In case, the view source is untyped or weak-typed, view property definitions must be valid for all instance types, which may appear in the view source. One way of ensuring view property compatibility is calculating critical properties via virtual expressions.

```
// osi example with inline expression
  Persons.Select( pid, address,
                  STRING full_name = {
                    VARIABLES
                      bool                  first = true;
                      STRING                result = 'Mrs. ';
                      set<MessageEingang> &mi = message(0).an;
                    PROCESS
                      if ( sex == 'male' )
                        result = 'Mr. ';
                      result << first_name << ' ' << name;
                    return(result);
                  };
```

1.3.4.1.1.10 View conditions

An ODABA view usually may contain a pre- and a post-condition (filter). When defining ODABA views as operation path, however, conditions may be inserted at any position in the operation path. In an operation path, there is no distiction between WHERE and HAVING and you may use the one or the other. Only, in a traditional OQL query, WHERE and HAVING differ in meaning, since the WHERE condition is evaluated in the context of the result of the SELECT operation, while the HAVING condition is checked after the GROUP_BY operation.

The argument for a WHERE or HAVING condition is an operand that is valid in the context of the preceding operator (operation path) or of the result of the SELECT or GROUP_BY operation (traditional OQL statement). Thus, WHERE and HAVING simply work as setting a filter condition for a collection property.

1.3.4.1.1.11 Group by operation

1.3.4.1.1.12 Property handle operations

There are a number of generic property handle functions supported in operation paths and expressions. Since generic property operations are a subset of property

functions and differ in parameter lists, there is a separate topic 'Interface functions', which documents classes and functions that can be calles from OSI expressions and operation paths.

1.3.4.1.1.13 Expressions

Expressions can be referenced in an operation path by name or as inline expression. Referring to an expression by name requires, that an appropriate OSI expression has been implemented in the class the operation applies on (preceding path element or calling property handle for the first path element). OSI expressions can be defined in the resource database (dictionary) or must be loaded from an osi folder when opening the database (see OSI_PATH in the data source definition).

When re-calculation of person's age has been defined in an OSI expression CalculateAge in the person class, re-calculating the age for all persons could be achieved by the following expression:

```
// referring to pre-defined expression
  PropertyHandle         p_age(obh,"Persons()->CalculateAge");
  p_age.Execute();


// referring to inline expression
  PropertyHandle         p_age(obh,
                               "Persons()->{age = Date.Year -
birth_date.Year;}");
  p_age.Execute();
```

1.3.4.1.1.14 User interface functions

When an operation is critical concerning performance, it might be a good solution replacing expressions by C or C++ functions. ODABA provides a OSI function interface generator, which allows calling functions written in C or C++. When calling interface functions e.g. for Person, you must define the functions in the Person class as OSI interface functions and generating the class interface. Afterwards, you may call functions from the interface rather than calling expressions.

This usually does not affect the access path, since the function interface supports the same parameter types as an expression call.

1.3.4.1.1.15 Calling context class functions in an operation path

Since context class functions mainly support event handling, the parameters and return values passed to and returned from context functions are strongly limited. The only reason to allow calling context functions in an operation path is, that sometimes important (maintenance) functionality is called via context class actions.

This makes a lot of sense in an OSI expression. Defining access paths in a program environment, you usually have simpler ways of doing this.

Let us suppose, a function CalculateAge has been defined in the person context class sPerson. The example below shows, how you may calculate the age for all persons using an access path.

```
// using operation path for calculating the age
  PropertyHandle   p_age(obh,
                         "Persons()-
>ExecuteInstanceAction('CalculateAge')");
  p_age->Execute();


// using basic property handles for calculating the age
  PropertyHandle   persons(obh,"Persons",PI_Update);


  while ( persons.Next() )
    persons.ExecuteInstanceAction("CalculateAge",NULL);
```

1.3.4.1.1.16 Special behavior of operation path

In general, a property handle opened for an operation path behaves similar to any other property handle, i.e. it provides the same functionality. Nevertheless, the operation path defines a virtual collection of instances, which is calculated, when accessing the operation path. When accessing instances of a operation path frequently and not in sequence, it is suggested to run the operation path and create a transient or virtual collection for the result.

In contrast to property handles based on property, path properties or property path, the operation path is read only in most cases. Only, when the result returns persistent instances, updates might be possible in some cases. Simple operation paths are paths containing a filter condition (where operation) or an order statement (order operation).

<Persons.OrderBy("sk_name").Where(age > 40)

Even though this operation path reorders persons and selects persons over 40, the result set still refers to the original person collection and persistent person instances. Other operations as Intersect or From (Join) also allow updates, because tracing back to the origin is still possible. Partially, this is also true for the select statement, as long as the property assignments in the select statement refer to property paths or path properties, only.

This sounds rather difficult, but practically, you can update operation path properties, as long as you understand, what you are updating at the origin (this simple rule is not 100% true, but it helps). When a property in an operation path has lost its origin, the property handle will reject any modification attempt.

What usually not works in a virtual collection is locating instances by key. Other virtual collections can be access forward, only. Hence, one has to check in the

specific situation, what type of operation is supported by the operation path creating the virtual collection.

Operation paths are good for getting ad-hoc answers or processing query results sequentially (forward). When more flexible access is required for accessing the result, it is better to create transient or temporary collections from the access path storing the selection.

```
char            *query = "Persons.Select(                          \
                  pid, address,                                    \
                  STRING full_name = {                             \
                      VARIABLES                                    \
                        bool                 first = true;         \
                        STRING               result = 'Mrs. ';     \
                        set<MessageEingang> &mi = message(0).an;   \
                      PROCESS                                      \
                        if ( sex == 'male' )                       \
                          result = 'Mr. ';                         \
                        result << first_name << ' ' << name;       \
                       return(result);                             \
                 }";
Property  ph(dbo,query,PI_Read);
```

Transient Properties

Transient properties are properties not stored to the database. You may define transient properties (collections or attributes) known in the application, only, by defining property handles for properties without reference to the database. Those are typically attributes.

Defining transient collection requires more information which must be provided in the datamodel (see "Using non-persistent references"). The datamodel allows also defining transient attributes, references and relationships within persistent data types. Defining and using transient attributes is rather simple and described in the following section. Defining transient references or relationships is possible in various ways depending on purpose. Different variants of defining transient properties is described in structure topic "References" (data model).

```
// transient attribute 'number'
PropertyHandle       number(0);
// transient collection defined in the data model
```

```
  PropertyHandle        retired(obh, "RetiredPersons", PI_Read);
```

Using transient Attributes

Transient attributes are usually attributes properties in persistent instances, which are not stored. Of course, each property handle referring to a transient attribute (e.g. PropertyHandle(5)) is transient as well, but here it is clear, that the application program has the responsibility for the property state.

This is a little bit more difficult for transient attributes defined in persistent instances as the age attribute in the Person structure of the Sample database. The problem is, that the age value changes from person instance to person instance and must be re-evaluated when reading new instances.

Transient attributes can be evaluated on all three layers (application, business and database). It is, however suggested, to calculate transient attributes either on the business or on the database layer.

<html><head><meta name="qrichtext" content="1" /><style type="text/css">

p, li { white-space: pre-wrap; }

</style></head><body style=" font-family:'Arial'; font-size:10pt; font-weight:400; font-style:normal;">

** Initialize attribute on the application layer </body></html>**

You might evaluate transient attributes in the application, whenever required, i.e. the person's age could be evaluated after reading a person instance. In this case, the application developer must be aware, that each time before accessing the age it needs to be evaluated. This works fine, as long as you found one and only one place in the application, where this information is requested. In this case, it is probably the most efficient way to solve the problem.

When the application becomes more complex, you would probably write a function calculating and returning the value, in which case you really do not need a transient attribute. The advantage of transient attributes is, that the business or database layer cares about updating the transient attribute in an optimal way.

<html><head><meta name="qrichtext" content="1" /><style type="text/css">

p, li { white-space: pre-wrap; }

</style></head><body style=" font-family:'Arial'; font-size:10pt; font-weight:400; font-style:normal;">

** Initialize attribute on the business layer </body></html>**

Since the application layer usually does not care about initializing transient attributes, this job must be done on the business layer or on the database layer. When

initialization of transient attributes is implemented on the business layer, the typical way of handling initialization is in the read event handler (e.g. DBRead() handler in a structure context class for Person). In this case, the application considers the transient attribute as any other attribute in the structure instance.

In some cases, transient attributes need to be re-calculates in case of modify or store events (the age must be re-calculated, when the birth date changes). When the evaluation rule for the transient attribute becomes more complex, it might become necessary, to re-calculate the attribute, whenever an operand value referred to in the rule, changes.

The advantage implementing evaluation of transient attribute values on the business layer is also, that you may decide later on, turning the transient attribute into a persistent one.

Evaluating transient attributes from source

When the rule is simple as for the age attribute you may also provide an expression or function for initializing the value. The initialization code is called always, when accessing the attribute via PropertyHandle functions (e.g. GetString() or Get()). This does not work at all, when using typed property handles and accessing the age via C++ class member names, since this is an uncontrolled access and the initialization is, for performance reasons, not called when reading the instance, but when accessing the attribute.

The rule for calculating transient attributes can be defined in the source for the transient attributes (assigned value or sources). An assigned value in ODL is considered as source, when it is not a constant (but an expression) Sometimes, evaluating transient attributes from source definition might be time consuming (e.g. calculating the total income for a person and all its children and grand children etc.)

Thus, transient attributes should be evaluated just in time and only, when being requested.

Initializing transient attributes

When no source and no event handler has been defined, transient attributes are initialized, when selecting or reading a new instance (Get(), NextKey() etc). When selecting an instance, the transient attribute is set to the default value according to the type or to the value which has been assigned as constant initialization value in the data model (assigned value or initial value).

Transient vs. persistent attributes

The big deal with transient attributes is, that you avoid redundancy on your database. Since transient attributes are calculated, when being accessed, you can be sure, that the values are correct.

In some cases, evaluation of transient attributes may become a performance problem. Then you may turn transient attributes into a persistent ones, which are updated only, when one of the evaluation operands changes but not any more, when

reading the instance. This performs much better, when your application has many read and less write access. Another advantage is, that updates on a persistent field generate an instance updated event, which allows other applications to react on a modification in the derived field. This is not the case, when using transient attributes, since updating transient attributes generates application events, only, which are not sent to other applications. Moreover, modifying transient attributes does not alter the instance modification count. Thus, other applications are not able at all to recognize updates on the instance.

Using non-persistent references

Here, we consider transient references defined in persistent data types as non-persistent references. The next topic shortly describes how to use transient collections, which are not member of a complex data type.

Transient references defined in complex data types are used to create transient links to one or more object types for a certain object instance. Thus, the value or content of a transient reference may differ for each object instance.

1.3.4.1.1.17 Transient References

filter and selection settings

order settings

assigning value

-

1.3.4.1.1.18 Transient collections

1.3.5 Buffered Access (buffer mode)

Buffered access provides fast access especially in client server mode. While reading 10 000 instances per second on a local machine is no problem, sending 10 000 packages via the network may take several minutes. Using buffered access will reduce the number of packages sent between client and server and thus, reduce the communication time extremely.

In order to activate buffered access, changeBuffer() needs to be called. In order to deactivate buffer mode, you may call releaseBuffer() or changeBuffer() with buffersize 0 or 1.

Buffered access can be used for reading collections with fixed data type, only. Trying to enable buffered access for untyped or weak-typed collections will fail without writing an error to the error log file.

When a filter has been defined for the property handle only those instances are read into buffer that fulfill the filter condition.

Usually the buffer is filled automatically when attempting to reading an instance that is not yet in buffer. The system tries to read as many instances as defined in buffer size from the current position. Subsequent get() request will read from the buffer as long as possible. In order to empty the current buffer cancelBuffer() can be called. cancel() will cancel the current selection in the property handle but not the instances in the buffer.

To position the buffer on a certain instance you can use the readBuffer() function. For resetting the buffer you can use the cancelBuffer() function.

You cannot use block mode for views containing references. In this case the function will ignore the request and buffer size remains 1. There are also problems in client/server mode when referring to sub property handles for references in instances that have been reading in block mode. Moreover, updating instances read in block mode is not allowed.

Read what is necessary

In many cases, only a subset of instances in a collection is displayed in the application. In such cases it is more efficient to read only the instances displayed. Reading them in block mode will not only reduce the access time but also reserve the instances for the client.

Update notifications

When reading instances in block mode all instances in a block are registered as being used by the property handle. Thus, the property handle will be notified when one of the instances in a buffer has been updated. In an ADL application, the application can react on this notification and, e.g. redisplay the line or list that has been updated

Enable block access mode

You can activate the block mode simply by setting a buffer size. The number of buffers allocated might be smaller than requested. For optimisation reasons the maximum block size can be restricted in client/server mode on the server side by the server option "MAX_BUFFER_SIZE".

When terminating a function or process, which has enabled the buffer mode, it should reset the buffer mode by calling changeBuffer(1).

```
... :: function ( Property &persons ) {
  person.changeBuffer(50);

  // processing ...

  // reset buffer in order to re-read instances
  person.cancelBuffer(1);

  // processing ...

  // reset buffered read
  person.changeBuffer(1);
}
```

Special events

When using block mode, read events for instances as well as for properties are generated when filling the buffer. When selecting an instance from the buffer into the handle, a read event for the property handle is generated once more. In order to check, whether the read event has already been executes the DateState can be checked in the read event handler (see DBO_Read event).

1.3.6 Property handle cache

In order to optimise access by key the property handle cache can be activated. Usually the property handle cache is deactivated. The property handle cache can be activated by calling changeCache() and passing a buffer count greater than 0. In order to deactivate the cache, changeCache() can be called again passin 0 as buffer count.

When the cache is active instances are saved in the cache and are read from the cache when accessing instances by key value. Reading by index will not access the cache. Hence the cache should be activated for random key access, only.

Using the cache in Write mode will lock all instances buffered in the cache. Updating an instance that is stored in the cache can be stored immediately using the save function. Changing the selection in the cache will store the instance to the cache, only.

In order to store all instances in the cache to database, flushCache() can be called.

1.3.7 Client/server configurations

1.4 Special Features

This chapter summarizes some special features provided with ODABA data access.

1.4.1 Object Identities

ODABA differs between global and local object identity. While global identities are global unique identifiers (GUID), local object identities (LOID) are unique within the context of an ODABA database. Local identifiers are created by default for each object instance. Global identifiers must be created explicitly or semi-automatically. Local object identities can be used to access data instances within a process. WEBapplications typically refer to local object identities.

LOIDs will not change as long as working in the same database. Copying the database or reorganizing it may, however, change the LOIDs, since LOIDs are sort of internal object addresses in a database.

LOID

Local object identities can be used for accessing data instances within an ODABA database. Thus, they are typically used in WEB applications as parameter to WEB pages that shall display information about a specific object.

LOIDs may change when copying an instance or the whole database. Since LOIDs are stable only within a given database they should not be used persistently, i.e. outside the "process" that has determined the LOID, except they are used just for identification but not for data access in the database (e.g. as id in an XML-file).

Getting LOID

The local object identity can be provided in different ways. The LOID is a numerical value that can be retrieved from the property handle for the currently selected instance. Another way is requesting the LOID from the index, which might perform bettwe in many cases, since no instance need to be read.

The LOID can also be accessed as an instance property, because it is considered a property of any persistent instance.

```
Property  persons(dbhandle,"Person",PI_Read)


// get identity from selected instance
persons.get(Key("Miller|Paul"));
persons.loid();
// get identity string from selected instance
persons.GetString("__LOID");


// get identity by key
```

```
persons.loid(Key("Miller|Paul"));

// get identity at position
persons.loid(27);
```

1.4.2 Versioning

The version concept describes the idea to maintain several instances for an object in order to reflect several states for the object. Usually (but not necessarily), a version is associated with a time point or interval, i.e. it reflects the state of an object at a certain time (interval).

Maintaining versions enables a system to represent previous (historical) states of an object.

A (data base) dimension defines the components necessary for identifying a data item within the database. Many database systems are two dimensional in the sense, that a data item is identified by an object instance identification (e.g. identity) and a property name. However, databases supporting different versions for object instances need one more component in order to identify the version of an object instance.

Version control in a most general way means that any object reflected in the data base is reflected including its history. There is no need (and no database) to have a continuous version update but successive time intervals might be defined in order to create version slices. Within a time interval (version slice) all modifications of the objects are treated as "corrections".

ODABA supports different types of versioning:

instance versioning

consistent versining

schema versioning

Instance versioning and consistent (object space) versioning cannot be mixed, i.e. when consistent versioning has been activated, instance versioning will be denied.

Instance versioning

The simplest way of creating versions is creating instance versions. Instance Versions can be created to keep the previous instance state. This means, that attributes and relationships are saved and can be accessed later on in order to browse the history.

Since "froozen" instance versions are not maintained, deleting instances may cause access errors later on in old collections (references or relationships). Therefore, this type of versioning is called inconsistent versioning. That means also, that keys referenced in indexes may become inconsistent, since older instance version attributes will not updated, when updating key attributes.

Instance versioning allows creating new versions for an instance whenever required and for each instance separately.

Consistent versioning

In order to provide consistent versioning, version slices need to be defined fo the complete object space. When creating a new version slice for an object space, modifications on instances will create new instance versions automatically. Moreover, indexes and relationships of the older versions are duplicated before being modified in the newer version. A roll back to an earlier version can recreate an earlier state of the database.

Schema versioning

When considering the complex relationships in an ODABA dictionary, it becomes obvious that it does not make sense to create versions of a single complex data type. By creating schema versions, which are object space versions for the dictionary, the complete state of the dictionary including type definitions, actions, method definitions, forms and many other project resources, are preserved.

Using object space versions for creating chema versions enables ODABA to support online schema evolution, which means, that databases need not to be reorganized after schema modifications. Instead, old instances are vonverted from older schema versions to the current type definition. Schema evolution is supported for any number of schema versions.

Versioning modes

In an ODABA database, only one of the version modes described above can be supported at the time. In order to create a new version slice, the DBVersion utility might be called or versions could be created by calling ObjectSpace::createVersion(). When creating a new version, the current version will get a termination time point, wich is the starting time point for the new version slice. By default, the starting point is the current time. In order to start the version slice later, one may, however, pass a timestamp referring to a timepoint in future.

Instance versioning

In order to create a backup for an instance at a certain time point, an instance version can be created by calling Property::createInstanceVersion(). Creating an instance version implies that all related ownined instances will create a version as well.

Instance versioning and consistent (object space) versioning cannot be mixed, i.e. when consistent versioning has been activated, instance versioning will be denied.

Version number are syncronized between an instance and its owning instances. Thus, creating a version for an instance owned by another instance will automatically crate a version for the owner instance.

Since ownership is unique in ODABA, i.e. each instance is owned by exactly one collection, each instance is either owned by an extent collection or by another in-

stance (which is the owner of the collection owning the instance). Thus, instance versioning always results in versioning an instance owned by an extent.

Instance version numbers are set consecutively. The oldest version is version 0. Instance versioning runs in two modes.

individual instance versioning

synchronized instance versioning

Both types of versioning are disabled, when version slice has been set as database versioning mode.

Individual versioning

Individual versioning creates individual versions for each instance tree, i.e. each instance tree gets its own version numbers. In this case, version numbers do not reflect a sequence of changes between different objects.

In order to activate individual versioning, the DBVersion utilityor ObjectSpace::versioningMode() might be called.

No access version can be selected when running s database in individual instance versioning mode, since version numbers do not have any meaning except the version sequence for an individual instance.

```
// DBVersion Utility: set version mode
  DBVersion.exe c:\Sample\sample.dat -M:I[ndividual]
// set version mode from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.versioningMode(individual);
```

Synchronized instance versioning

When running a database with synchronized versioning mode, each version request will create a new consecutive version number. Thus, version number represent a history, since lower versions are older than higher versions.

In order to activate synchronized versioning, the DBVersion utilityor ObjectSpace::versioningMode() might be called.

An access version can be selected when running s database in synchronized instance versioning mode, which causes the database reading instance data for versions older or equal to the requires access version.

```
// DBVersion Utility: set version mode
  DBVersion.exe c:\Sample\sample.dat -M:S[ynchronized]
// set version mode from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.versioningMode(synchronized);
```

Consistent versioning

There is no need (and no database) to have a continuous version update but successive time intervals might be defined in order to create version slices. Within a time interval (version slice) all modifications of the objects are treated as "corrections". Starting a new object space version, new versions of object instances are created whenever the object instance is modified. The version interval can be a fixed interval (e.g. each hour, weekly or monthly, yearly) or a variable interval explicitly defined by the database administrator.

In fact the version creates a new dimension for the database, which should not affect the data model at all.

ODABA supports a differential version control which allows to create up to 65535 versions for a database (beginning a new version each day versions for 180 years can be created in the database). In a database supporting full version control each version reflects a consistent state of the database at a certain time, not only for the instances but also for the relationships between them, as well as for indexes.

Defining versions slices

In order to define new version slice, the DBVersion utility might be called or versions could be maintained calling appropriate ObjectSpace functions. New version slices can be defined, removed or timestamp of version slice might be changed.

When no versioning mode has been set so far, versioning mode is set to consistent when terminating the first version slice. The version mode might, however, als be set in advanve.

By default, the version 0 is defined for each object space as open version slice, which will never terminate, until a new version slice will be created. The first version slice has got an open starting point. The last version slice always has got an open termination point. Time intervals for time slices cannot overlap and there are never gaps between time slices.

One may reset a complete time slice, which will remove all changes made in this time slice.

```
// DBVersion Utility: set version mode
  DBVersion.exe c:\Sample\sample.dat -M:C[onsistent]
// set version mode from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.versioningMode(consistent);
```

Creatre new version slice

In order to create a new version slice, the DBVersion utility might be called or versions could be created by calling ObjectSpace::createVersion(). When creating a new version, the current version will get a termination time point, wich is the starting time point for the new version slice. By default, the starting point is the current time. In order to start the version slice later, one may, however, pass a timestamp referring to a timepoint in future.

```
// DBVersion Utility: Create new version slice
  DBVersion.exe c:\Sample\sample.dat -C -T(2010-01-01 00:00:00,00)
// create new version from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.createVersion(DateTime("2010-01-01 00:00:00,00"));
```

Update version slice

In order to change version slice borders, version slice intervals might be updated. Updating version slice termination date is possible via the DBVersion utility or by calling ObjectSpace::changeVersion().

Changing version slice borders is possible for active and future version slices, but not for version slices, which had already been closed, i.e. where the termination date has already expired. When updating the termination date for a version slice, the timestamp value has to be between the termination data of the previous and the next version slice. The termination date for the last version slice is always open and cannot be updated.

```
// DBVersion Utility: Update version slice
  DBVersion.exe c:\Sample\sample.dat -U -V:5 -T(2010-06-01)
// Update version slice from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.changeVersion(5,DateTime("2010-01-01"));
```

Reser changes made in a version slice

Resetting the current version for an object space to a previous version can be done by calling the DBVersion utility or by calling ObjectSpace::resetVersion(). Resetting a version will remove all database version entries from the database with a version number equal or above the version number to be reset. Finally, the object space version number is reset to the predecessor of the version slice to be reset.

```
// DBVersion Utility: Update version slice
  DBVersion.exe c:\Sample\sample.dat -R -V:5
// Update version slice from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  osh.resetVersion(5);
```

List version slices

In order to retrieve nformation about start and stop time for version slices, one might call ile list operation of the DBVersion urility or call ObjectSpace::versionInterval().

```
// DBVersion Utility: List version slices
  DBVersion.exe c:\Sample\sample.dat -L
// Update version slice from within a program: ObjectSpace osh;
  osh.open(mainClient(),"Sample",write,local);
  int       i= 0, count = osh.versionCount();
  DateTime  start, stop;
  while ( i < count ) {
    osh.versionIntervall(i,start,stop);
    // print i, start, stop
```

```
    }
```

1.4.3 Copy model

Copying data is often rather dificult. Deep copy usually leads to infinitive recursion or tends to copy large parts of the database. Copying instances only is often to weak. ODABA provides some enhanced copy methods in addition, e.g. copying primary relationships, only.

Thus, there are several ways of controlling copy operations from a application. Nevertheless, this is usually not sufficiant, since copying linked objects is usually a conceptual decision. Many copy problems can be solved by defining a copy model. A copy model can be defined by setting copy dependencies for each property.

Calling copy functions with the COPY_dependent option will copy related instances according to the defined copy model.

Since copy dependencies are part of property defintion in the data model, ODABA supports only one copy model. Praxis, however, has shown, that the one copy model wil propably solve 99% of your copy problems.

1.4.4 Check model

ODABA supports a simple check model by defining check levels for properties in data type definitions. This allows running logical consistency checks against data to be stored in the database. A more sofisticated check model can be defined by setting up constaints for properties or complex data types or by implementing check handler (DBCheck), which are called when check events are fired.

ODABA does not automatically generate cecks for logical database consistency. In order to perfom locgical consistence checks, the CheckData() function must be called.

1.4.5 Recovery log-file

In order to track changes, the recovery log file feature might be enabled.

1.4.6 Workspace

Workspaces are a special ODABA feature, which provide persistent transactions. Workspace transactions will survive the process and are available for any number of processes. Results of a workspace transaction are stored in a separate database and are copied to the original database when being consolidated by the user.

The workspace features can be enabled running the Workspace database tool.

1.4.7 License services

In order to check customer licences for comercial database applications, the function might be called. Licence agreements are defined in the database definition in the resource database. The licence key generated from the database definition for the project and its applications can be used by the customer in order to initialize its license.

1.5 Locking and write protection

1.5.1 Locking Features

1.5.2 Write protection

1.6 Transactions

1.6.1 Starting and committing user transactions

1.6.2 Starting and committing workspace transactions

1.7 Database context programming

Context programming allows defining business rules for instances. ODABA distinguishes between context classes for complex instances (type context) and context classes decribing the behaviour of property instances (property context).

Whithin a context class, system events generated by the system can be handled. System events are generated, when an instance changes its state.

Context programming is provided for two reasons:

Property context classes

Property context classes can be implemented in order to provide specific event handlers or business rules for elementary or collection instances. Property events are generated, when the property instance changes its state or when a system event is signaled for the property instance.

Type context classes

Type context classes can be implemented in order to handle events generated for complex instances, i.e. when a complex instance changes its state (instance or system state). The type context class is associated with the complex data type of the instance.

Object space context

Object space context classes can be implemented in order to handle events for an object space. Usually, object space context classes should be implemented only, when the database is working with different object spaces. In order to link an object space with a context class, a named object (object model resource) has to be defined and a resource reference has to be assigned and associated with the corresponding context class.

The root object space is handled, usually, by the database context.

Database context classes

Database context classes can be implemented in order to handle events for a database. In order to link a database with a context class, a database object (object model resource) has to be defined and a resource reference has to be assigned and associated with the corresponding context class.

1.7.1 Associate context class with data model resource

In order to link a complex data type or property within a complex data type (object model resources) to a context class, a resource reference has to be created for the object model resource, which provides a unique resource identifier (number).

Context classes can be assigned in the ClassEditor tool, which also generates link tables between context class neme and resource identifier. Those are compiled in a C function (CreateContext()), which creates context class instances for complex and property instances, wich are linked to a context class.

Any number of resources can be associated with one context class.

1.7.2 Handling events

There are different ways of handling system and instance events in ODABA. Within an application, events are typically handles by appropriate context classes. In order to handle system and instance events by context class event handlers, corresponding event handlers have to be overloaded in the context class implementation.

Context class event handlers can be enabled and disabled calling appropriate context class or property handle functions (BaseContext::enabled()).

Besides, generic event handler can be provided. Generic event handlers have to be implemented on system interface level (PropertyHandle, EventHandler). Generic event handlers can be enabled or dsabled by calling appropriate system interface functions (PropertyHandle::BlockEvents(), DBObjectHandle::EnableEventHandling()).

Disable and enable context handlers

In order to disable context functions completely, you may set the contextEnabled property in a DataSource definition to false. before opening the data source or passing false in bContextEnabled parameter when opening a database. Disabling context when opening a database or data source disables context behavior completely. i.e. context rules cannot be enabled from within the application while the database is opened.

When context behaviour is enabled for the opened database (default), it might be disabled partially. In order to disable a certain context (e.g. the type or property context for a property handle), the appropriate context can be disabled(enabled by setting the BaseContext::enable property.

Since the type context for a property depends on the type of the currently selected property, disabling the type context becomes more difficult, when reading e.g. instances for a weak-typed collection. Since reading an instance in a weak-typed collection may change the current type of the selected instance, disabling the context before reading the instance will not have any effect, when the new instance has got another type. Hence, Property::setType() has to be called before reading the instance in order to disable the proper context. This becomes difficult, when the type of the instance to be read is not known. A possible solution is disabling the property context and checking the property context state in critical context functions.

Disabling context for weak-typed collections

Since the type context for a property depends on the type of the currently selected property, disabling the type context becomes more difficult, when reading e.g. instances for a weak-typed collection. Since reading an instance in a weak-typed collection may change the current type of the selected instance, disabling the context before reading the instance will not have any effect, when the new instance has got another type. Hence, Property::setType() has to be called before reading the instance in order to disable the proper context. This becomes difficult, when the type of the instance to be read is not known. A possible solution is disabling the property context and checking the property context state in critical context functions.

Simular situation happens when inserting instances to weak-typed or untyped collections. In order to disable the proper context, i.e. the context of the instance to be inserted, the type has to be set before disabling the context. In case of calling Property::insertReference(), the type to be set should be extracted from the property handle passed to the function.

```
// read handler for Empoyee (fragment)
int tEmployee::doAfterRead () {
  BaseContext &prop_ctx = highContext();
  if ( prop_ctx.enabled() { // execute, when property context
enabled
    // do something
  }
}

// fragment:
// members is a weak.typed Person collection
  members.propertyContext().enabled(false);
  members.next();
  members.propertyContext().enabled(true);
```

Diabling context for transient properties and operation paths

Enabling or disabling property context for operation nodes is an important feature, since the property context often is responsible for evaluating the result for a transient property or operation. After evaluating an operation result, the result will be returned internally in a result property, which can be retrieved by calling Property::resultProperty(). It is possible to disable property context for the result property by providing the context for the result property and disabling it. Disabling context for the result property, however, should be avoided, since it may produce unexpected side effects.

Usually, an operation or transient property does not ahve got an own instance, and thus, no own type context. When a property handle refers to an operation path or has been evaluated by the application, instances are provided in a result property. When selecting an instance in a transient property handle, the instance is passed from the result to the transient property handle. When disabling the type context after selecting an istance, this works fine, but the instance owner is the result property, i.e. practically, the type context for the result property has been disabled. Similar to the property context for the result property, this may lead to unexpected side effects and should be avoided.

```
// read handler for Empoyee (fragment)
int tEmployee::doAfterRead () {
  Property    &person = property();
  // local clients will execute both blocks
  if ( !isClient() ) { // run on server side, only
    // do something
  }
  if ( !isServer() ) { // run on client side, only
    // do something
  }
}
```

Disable context for property

The property handle provides a property (Property::contextEnabled), which allows disabling or enabling the context for a property handle. Since disabling context sometimes becomes difficult (weak-typed collections, operations, transient properties), this property allows controling all context instances associated with the property.

Disabling the context for a property handle includes disabling follwowing context class instances associated with the property:

property context for the property

type context for all instance types (weak-typed or untyped)

property context for result property (operation and transient properties)

type contexts for instance types for the result property

When a result property of a transient or operation property changes, the contexts for the result node are automatically enabled and contexts of the new result property are disabled.

Scalability for context rules

When running an object server application, context functionality may be executed on the server, on the client or on both. The object server is the only server type, which is able to run context functions.

In order to avoid running context function on client or server, the context function have to check, whether they are running on a server or client. In order to run context functions on client, only, !isServer() should be checked. !isClient() can be checked in order to execute a context function on server side, only.

Since all clients, which are not object server clients, are considered as client, but also as server, this works properly in each client server configuration.

1.7.3 Providing actions

Besides event handlers, actions can be implemented in a context class, which are considered as business rules. Actions provide application independent behaviour, which might be activated from whithin any application.

In contrast to event handlers, actions are not called implicitely by the system but have to be called explicitly by the application (e.g. Property::executeInstanceAction())

Passing data between context class and application

When calling an action,

1.8 Data exchange

Data Exchange provides different ways of importing orexporting data from an ODABANG database to extended comma separatedfiles (ESDF, CSV), to xml files (OXML, XML) or to object interchange formatfiles (OIF). OIF is the proposed standard format for exchanging data betweenobject-oriented databases (ODMG).

While the capabilities of ESDF (CSV) are limited, OXMLand OIF allow transferring the complete content of a database. Even though XMLis the more common format, OIF has the advantage that it should be supported byall object oriented databases and it consumes less space.

Besides different file formats ODABANGprovides different data exchange technologies.

Command line tools

ODABANG provides two data exchange tools,one for import (Import) and another-for exporting data (Export).

Import

Import provides features for importing a file with a valid import format into an ODABA database. This is a preferred way for importing data periodically, in which case a batch job cab be prepared and called whenever required.

It is a possible but not the most comfortable way for ad-hoc import processes,which can be solved better with the GUI Data Exchange or from within OSI programs.

Export

Export provides features for exporting selected data from an ODABANG database to a file with one of the supported external data formats. This is also a preferred way for exporting data periodically, while ad-hocdata export becomes more comfortable from within OSI or by using the GUI DataExchange tool.

GUI tool

The Data Exchange GUI tool provides features for designing the content of a data exchange and running the data exchange directly or creating a data exchange definition file (data exchange schema).

The data exchange schema can be referred to later when calling the command line tool or from within an OSI script.

The GUI tool provides the most comfortable way for designinga data exchange schema. It allows also running the defined data exchange (e.g.for testing purpose).

OSI Expression

Often, data exchange is simple and can be directly called from within an OSI expression. OSI OQL provides two built-in functions in order to import and export data. Those functions are the same functions which are called from the command line and the GUI tool.

ToFile

ToFile writes data from a defined collection to an external file with one of the defined formats. The OSI query may create a view for the data to be exported. Since the data exchange schema supports property selections, this is, however, not necessary in most cases.

FromFile

FromFile imports data from an external file with one of the supported formats into a collection. In general, one cannot import data into a view, but there are views that are partially updateable, which would allow importing data as well.

Property

You may access an external file by property handle.This allows reading or writing data from a program or from within an OSI expression. Property handles for external files will not, however, import or export data automatically.

Opening a file via property handle activates the rich property handle functionality for the external file. Although there are many features, which cannot be supported for an external file, many helpful functions of property handle are still working for this data source type.

Accessing external data via property handle does not require an exchange schema. A file schema, which does not define data mapping, would be sufficient. Since file schemata for CSV files can be derived very simple in many cases, the external file does not require additional information for being accessed.

The property handle access functionality is the base for the OSI functions FromFile and ToFile.

The file schema for external files can be defined in advance within the ODABANG dictionary in terms of structure and extent definition. In this case, the external file can simply be accessed via the extent name, similar to any other extents in the database.

Open extern

Often, it is not very comfortable defining structure and property handles for external files in the dictionary. Especially, CSV files often carry metadata in the headline, which contains sufficient information for extracting a file schema. Thus, property handle supports an additional function for opening external data sources, which are not defined in the dictionary. This allows accessing data ad-hoc and in much simpler in many cases.

In order to access external files that do not have a file schema definition at all, ad-hoc schemata can be created for semi-structured files as XML, OIF and OEL. In this case, the file is analysed and a file schema is derived from property names passed with the data.

The copy model

Part of the object model is the copy model, which allows defining the instances and references to be copied when importing or exporting data.

The copy model is defined by setting the copy level for references and relationships. This allows defining copy rules for copying related instances for an instance, which is going to be copied.

1.8.1 Data exchange definition

Data exchange definitions describe the file data source, the schema location and format types for exchange file and schema. Usually, the exchange definition is specified in a ToFile, File or FromFile operation, but this might be hidden behind a more comfortable user interface.

File definitions are provided as predefined data exchanges in the dictionary or in external file descriptions. Besides different external file formats, ODABA also supports different ways of external file descriptions.

Access functions

External files can be accessed in different ways.

File - Read or write explicitly

FromFile - Import from file

ToFile - Export to file

Depending on the required functionality, a file schema and a data exchange schema should be provided.

File

The File() operation allows accessing an external file structure, which is defined by an explicit or implicit file schema. External files can be read or written, but depending on the file structure, there are several restrictions. Most external file formats do support appending data to the file, only.

External data sources can be accessed in OSI script files or access path via the File() operation, but also via property handle.

```
// OSI or access path
VARIABLES
  set<VOID>  &extFile = File( Path='c:/temp/my_persons.xml',
FileType='xml',
                              Definition='c:/temp/my_persons.def',
Headline = false);
PROCESS
  while ( extFile.next )
//    ... processing;
```

```
// C++ property handle
  Property     extFile;

extFile.openExtern(os,"c:/temp/my_persons.xml","c:/temp/my_persons.
def",
                     'xml',false);
  while ( extFile.next() )
//    ... processing;
```

FromFile

The FromFile() operation supports importing data from external files into a database. Importing files requires a (usually explicit) data exchange schema (extended file schema), which provides a mapping to database locations in addition to the structure definition of the import file.

Data can be imported in an OSI script, but also directly in an application program via ODABA API functions. Calling import() with the property handle requires a self-contained import file, i.e. the data exchange schema must be part of the import file. In order to import data for files with a separate exchange schema, an operation path can be used, which is more flexible, since it allows defining different locations for the exchange schema.

```
// Import external data to extent Persons
// OSI version
  Persons.FromFile( Path='c:/temp/my_persons.xml', FileType='xml',
                    Definition='c:/temp/my_persons.def', Headline =
false);

// C++ version
  Property     persons(os,"Persons",Update);
  persons.import("c:/temp/my_persons.oif");

// C++ operation path version
  Property     persons(os,"Persons",Update);
  Property     impPh(persons,
                      "Path='c:/temp/my_persons.xml',
FileType='xml',   \
                       Definition='c:/temp/my_persons.def', Headline
= false");
```

```
  impPh.execute();
```

ToFile

The ToFile() operation supports exporting data from a database to an external file format. Es well as the FromFile() operation, ToFile() requires a data exchange schema.

Data can be exported in an OSI script, but also directly in an application program via ODABA API functions. Calling export() with the property handle requires a self-contained export file, i.e. the data exchange schema must be part of the export file. In order to export data for files with a separate exchange schema, an operation path can be used, which is more flexible, since it allows defining different locations for the exchange schema.

```
// Import external data to extent Persons
// OSI version
  Persons.ToFile( Path='c:/temp/my_persons.xml', FileType='xml',
                  Definition='c:/temp/my_persons.def', Headline =
false);

// C++ version: export Persons to an OIF file
  Property     persons(os,"Persons",Update);
  persons.export("c:/temp/my_persons.oif");

// C++ operation path version
  Property     persons(os,"Persons",Update);
  Property     expPh(persons,
                     "Path='c:/temp/my_persons.xml',
FileType='xml',   \
                      Definition='c:/temp/my_persons.def', Headline
= false");
  expPh.execute();
```

File access parameter

Referring to external files requires a specific parameter list, which defines file location and structure. The parameter list may contain the following named parameters:

Path - file location

FileType - Type of file to be read or created

Definition - file and exchange schema

Headline - headline option

```
FileExtent( Path='c:/temp/my_persons.xml', FileType='xml',
            Definition='c:/temp/my_persons.def', Headline = false)
```

File path

At least the file path must be passed as operand to the file access functions. Additional file options can be passed for providing file and exchange schema and file type.

The path name points to the location for storing the file. The complete path name should be enclosed in quotes (single or double) to avoid misinterpretations. The default for path is 'Console', which will direct the input or output to the screen.

File type

ODABA supports different external file types. The file type need not to be defined,when the file name passed in Path has one of the following extensions:

CSV, ESDF - extended self delimiter file (.esdf, .csv)

OXML - ODABA xml file (.oxml, .xml)

OIF, OEL - object interchangeformat (.oif, .oel)

BINA - binary flat file (.bina)

File type definitions are not case sensitive.

File schema

The file or exchange schema can be provided as definition in a dictionary, in which case the structure_name refers to a structure definition in the dictionary. Beside property definitions (file schema), data sources can be assigned, that are referred to in case of import or export operations.

The file or exchange schema might provided in a separate file as ODL (.odl), OXML (.oxsd) or ESDF (.esdf) definition files, in which case the location is passed as quoted string pointing to the file location. The format of the file schema need not to correspond to the format of the data file, i.e. one may pass an ODL exchange schema for importing an XML file.

When the file schema is not passed explicitly, the file schema is supposed to be part of the external file (e.g. headline in an ESDF file). The file or exchange schema can also be provided together with the data (and in the format according to the file type). In this case, the definition format has to correspond to the format of the data file.

BINA - no file definition supported in the file

CSV, ESDF - ESDF headline format

OXML - OXSD schema definition

OIF - ODL definition

The system determines the proper type for the definition file from the file extension. When no valid extension has been detected, the system tries to analyze the definition file type by file content:

first character '<' : OXML format

first character '{' or beginning with a word followed by a separator : ESDF format

Beginning with schema keyword: ODL format.

Beginning with META: OEL format

Headline option

The headline option indicates, whether the external data file includes the file schema (typically the headline in CSV or ESDF files). Either headline or schema location must be provided in order to obtain the file schema for input operation.

In case of output operation, the file schema is written to the export file prior to the data. Existing schema definition in the output file header will be ignored, i.e.the exchange schema must be defined in a separate definition (database schema or schema file). When no exchange schema has been provided, the input structure is used as an implicit exchange schema, i.e. all attributes and depending object instances are exported to the output.

When defining both, the schema location is used. In some cases, schema location is verified against the headline definition.

1.8.2 Data Exchange schema

A data exchange schema is required for any type of data exchange in order to provide the mapping rules between internal and external data. The data exchange schema is an intensional schema, i.e. it refers to structure definitions, only. Thus, a data exchange schema can apply on any collection (database) or file (external data source), which fits into the rules defined in the data exchange schema.

Data exchange schemata can be provided in different formats. The format of the dataexchange schema does not depend on the file format for the external data-source. Thus, you may still use the same data exchange schema definition, eventhough you have changed the format of the external file. Data exchange schematacan be provided in one of the following formats:

Dictionary - Structure definition in an ODABA dictionary

CSV/ESDF - Headline definition format

OSI ODL - Schema definition language

OXML - extended XML schema definition

The data exchange schema is an extension of a file schema, wich defines additional mapping rules for assigning external data fields to database properties. Database property correspondences are always defined in the source attribute for the file property.

ODL and XML schema definitions follow the common rules for structure definitions in an ODABA schema definition. ESDF exchange or file schema definitions are an extension of the CSV headline, i.e. the csv headline is a special case of an external file schema.

```
// ESDF exchange schema for DSC_Keyword

name     = definition.name;

number   = __AUTOIDENT;

lex_term { name = definition.name; number = __AUTOIDENT }=
lex_base[0]
```

Dictionary exchange schema

Describing an external file structure in the dictionary might be the most comfortable way for complex external data structures. External files can be defined as structure definitions in the dictionary. Structure definitions for external files may consist of attributes, references and exclusive base structures. External file definitions must not contain relationships.

Assigning a data source to a field in an external file is possible by means of the property source definition for properties. The first (and only) source definition for a property is considered as database location for the external field.

ESDF exchange schema definition

The definition for CSV or ESDF (Extended Self Delimiter Files) is an extension of a CSV file headline. In the minimal case itonly consists of variable names.

In order to support more complex data structures in a comfortable and CSV compatible format, we introduced ESDF, which is a CSV extension, since it supports complex attributes as well as references. In addition, the definition of mapping rules has been added to the file schema in order to provide exchange schema definitions.

Specification

The rules for defining a CSV or ESDF file are described in the subsequent BNF definition. All fields are assumed to be presented as ASCII text (string data type).

The BNF describes the ESDF header. In contrast to CSV, ESDF limits field delimiter to ';', tab and '|', which can be used simultaneously. Undefined BNF symbols name, number and constant are standard symbols.

The file or exchange schema for an ESDF file is usually passed in the first line of the file (headline). I might be passed, however, also separately from the data file or dictionary.

name

A name or identifier is a field name which usually starts with an alphabetic character or underscore and contains ASCII characters and numbers, only.

Number

In this context a number is an integer value.

constant

A constant is either a number, a string constant or a Boolean value.

Usage

The file or exchange schema for an ESDF file is usually passed in the first line(s) of the file (headlines). It might be passed, however, also separately from the data file or dictionary.

```
// ESDF headline definition

f_pid = pid; fname = name; f_first_name = first_name; f_birth_date=
birth_date; f_sex = sex; f_married = married; f_income = income;
f_location {f_zip = zip; f_city = city; f_street= street; f_number
= number} [3] = location
```

Defaults

When names in the headline are identical with databasesource names, source assignments can be omitted:

```
pid; name; first_name; birth_date; sex; married; income; location
{zip; city ;street; number } [3]
```

Schema file

When providing the exchange or file schema separately instead of providing it in the headline, the definition may contain line breaks:

```
// Separate exchange schema
f_pid         = pid;
f_name        = name;
f_first_name  = first_name;
f_birth_date  = birth_date;
f_sex         = sex;
f_married     = married;
f_income      = income;
f_location {
  f_zip = zip;
  f_city = city;
  f_street= street;
  f_number = number
} [3]          = location
```

ODL exchange schema definition

The ODL schema definition is a script equivalent to the dictionary definition. It follows the same rules as defining a structure in the dictionary.

In order to provide self-contained OIF files, the ODL schema can be passed on top of an OIF file.

Specification

The BNF describes the common structure of an OIF file. The OIF file might be preceded by an exchange schema definition, which must start with the SCHEMA keyword. The details for schema definitions are described in OSI language reference.

The example below shows a complete definition for a Person data exchange schema.

```
SCHEMA {
  STRUCT XAddress {
         STRING   f_zip         SOURCE(zip);
         STRING   f_city        SOURCE(city);
         STRING   f_street      SOURCE(street);
         STRING   f_number      SOURCE(number);
  };

  STRUCT XPerson {
    ATTRIBUTE {
         STRING   f_pid         SOURCE(pid);
         STRING   f_name        SOURCE(name);
         STRING   f_first_name  SOURCE(first_name);
         STRING   f_birth_data  SOURCE(birth_date);
         STRING   f_sex         SOURCE(sex);
         STRING   f_married     SOURCE(married);
         STRING   f_income      SOURCE(income);
    };
    REFERENCE XAddress  f_location[3] SOURCE(location);
  };
};
```

Usage

Since OIF is able to transfer complex data, the file or exchange schema is usually defined in the dictionary. In order to transfer self-contained files, it is, however, sugested to create OIF files containing data and schema.

OEL exchange schema definition

OEL (object exchange language) is a predecessor of OIF and rather similar to it. OEL is supported for compatibility reasons, only. Normally, it is sugested using OIF rather than OEL.

In contrast to OIF, OEL does not support locators. Thus, links to other object instances can be provided by key attributes, only, which is supported by exporting data. Thus, OEL is sufficient for importing data but will not work properly for exporting data in many cases.

Similar to OIF, OEL files may contain an exchange schema definition in order to provide self-contained OIF files.

Specification

The BNF describes the common structure of an OEL file. The OEL file might be preceded by an exchange schema definition, which must start with the SCHEMA keyword. The details for schema definitions are described in OSI language reference.

The example below shows a complete definition for a Person data exchange schema.

```
SCHEMA {
  STRUCT XAddress {
         STRING   f_zip         SOURCE(zip);
         STRING   f_city        SOURCE(city);
         STRING   f_street      SOURCE(street);
         STRING   f_number      SOURCE(number);
  };

  STRUCT XPerson {
    ATTRIBUTE {
         STRING   f_pid         SOURCE(pid);
         STRING   f_name        SOURCE(name);
         STRING   f_first_name  SOURCE(first_name);
         STRING   f_birth_data  SOURCE(birth_date);
         STRING   f_sex         SOURCE(sex);
         STRING   f_married     SOURCE(married);
         STRING   f_income      SOURCE(income);
    };
    REFERENCE XAddress  f_location[3] SOURCE(location);
  };
};
```

Usage

Traditionally, OEL has been using for importing data from MS Word documents. Hence, OEL is still suported in order to keep old import functions running. In any case, OIF is the better choice and should be used instead.

XML exchange schema definition

An OXML schema is another equivalent for a dictionary structure definition and can be used instead of an ODL or dictionary definition. Usually, XML schema definitions are more dificult to provide than ODL, but In order to provide self-contained XML files, the OXML schema can be passed on top of an XML file.

A better way, however, is to provide a separate schema file and make this available in the network. Than, OXML files may refer to the exchange schema defined in this XML schema definition.

1.8.3 External data formats

ODABA supports different external file formats, which can be accessed directly via property handle access functions or via file functions File(), ToFile() and From-File().

Beseides external file formats, several external file schema formats are supported. External file format and file schema format need not to correspond to each other, i.e. you may describ a flat file, which does not have a file schema format, in terms of an OIF schema or an OIF file in terms of an ESDF schema.

External file formats

ODABA supports the following external file types:

CSV, ESDF - extended self delimiter file (.esdf, .csv)

OXML - ODABA xml file (.oxml, .xml)

OIF - object interchangeformat (.oif)

OEL - object interchangeformat (.oel)

BINA - binary flat file (.bina)

File type definitions are not necessary but may held identifying the storage type for an external file by default.

Flat or binary files

Binary files are files with a fixed data structure. Binary files can be considered as the most compressed format for data exchange. In contrast to other file formats, binary files do not support subordinated collections.

There are several limitations in using binary files.

- Binary files always require a separate file definition (no headline definition supported).
- Binaryfiles do support arrays with fixed number of elements, only.

In contrast to all other external data formats, which are limited to ASCII data, binary files may contain any type of data.

ESDF or CSV format

The Extended Self Delimiter File format is an extension of the CSV format. ESDF files contain one record per line, i.e. line break indicated the end of a record. In contrast to CSV, ESDF supports complex attributes and reference collections with variable number of instances.

Since ESDF does not require any tags, it is an efficient way of exchanging large data files. On the other hand, it requires fields being defined in a correct sequence.

ESDF files may carry the file or data exchange schema directly in the data file (headline). The file or data exchange schema can also be defined in the dictionaryor passed separately in any file schema definition format.

Specification

ESDF has a simple BNF specification as described below. As line break, new line (NL), carriage return (CR) or both are accepted after headline and between data lines. Headlines are optional. File definitions might be also passed separately and in any other format.

New lines are not considered as instance separator when being defined whithin a locator,anitem set or an item block.

Data exchange schema

ESDF files may contain a headline defining the file or exchange schema. Since headlines need not differ syntactically from data lines, the file definition must pass the headline option in order to indicate, that an ESDF file contains a headline at the beginning.

When passing the exchange schema in the data file headline, the schema must be defined completely in the first line.

Delimiters

ESDF defines a reserved set of delimiter characters. Delimiter characters must not appear in values without being quoted. In contrast to CSV, ESDF requires additional delimiters for instances and collections.

Field delimiter

Characters ';', '|' and '\t' (tab) are considered as field delimiters. Field delimiters may appear also mixed, i.e. also when creating an ESDF file using '\t' as field separator, values containing a ';' must be enclosed in string delimiters.

String delimiter

" and ' are considered as string delimiters. The starting string delimiter must be the terminating delimiter, too. Starting a string value with ", the value may contain ' and reverse. When starting string delimiters need to be coded within the string, those must be preceded by an '\'.

```
'my name is"Paul"'   // valid
'my name is\"Paul\"' // valid, same as above
'my name is\'Paul\'' // valid
"myname is 'Paul'"   // valid, same as above
```

Instance delimiter

Instance delimiters '{' and '}' are used to define begin and end of complex (structured) data values. Instance delimiter may appear within value collections but also outside collections. Instance delimiters are not required for base structure members.

Collection delimiter

Collection delimiters '[' and ']' are used to define value or instance collections.

Object Interchange Format (OIF)

ODABA XML format

1.9 ODABA data storage formats

ODABA provides the feature of storing data in different database storage formats. Following data storage types are supported:

Relational databases (ORACLE, MySQL, MS SQL Server)

OXML (XML based on ODABA schema extensions)

This does not mean, that ODABA is able to access any relational database or xml file. When running ODABA on external data formats, those are managed by ODABA in order to keep all extensional features provided by the system. Thus, external formats must follow some basic rules defined for the different database formats.

1.9.1 Storing ODABA data in relational databases

ODABA supports storing data in several relational databases. This is not the most efficient way of accessing data stored in ODABA, but it provides additional data access by well known SQL tools. Thus, running ODABA based on an SQL database might increase acceptance by customers.

The following SQL databases have been chosen for ODABA support:

ORACLE

Microsoft SQL Server

My SQL

This list might be expanded when ever required.

RDB access architecture

Whenrunning ODABA with a relational database, instances data is stored in relational tables. Optional, the administrator may decide whether to maintain m:n relationships in the RDBM or not. Thus, one may store data tables, only or data tables plus relationship tables.

In order to obtain extended ODABA features as collection events, extended instance and collection information etc. an additional database (Object Manager) is required.

Extended information as update counts for instances or collections, weak-typed or untyped collections or __IDENTITY/type mapping could hardly be handled in an relational database. Thus, an Object Manager maintains collections (relationships and references), but also update counts, locking and persistent write protection.

All services as transaction management, locking or workspace features are managed by ODABA, since SQL databases do not provide sufficient support e.g. for locking the children collection of a person. Moreover, ODABA cares about extended deletion features, maintaining inverse references and other specific object-oriented database features.

OR mapping rules

Since the information content of a relational database is a subset of the information, that can be stored in an object oriented database, mapping rules can be defined for the "relational data" in the object-oriented database.

Instance data is stored in tables having the same name as the complex data type defined in the ODABA object model. All tables get an additional property SYS__LOID, which held the unique object identity for each instance. All relational tables are indexed by SYS__LOID.

Complex attributes are provided as resolved attribute names including dots, which are part of the attribute path (address.city). This usually requires apostrophes when referring to attribute names. Attributes in exclusive base types are not prefixed.

When a data type inherits shared from its base structure, attributes are stored in separate table for the base type using the same names as in the data model definition. An attribute with the name of the base type member is added to the table, which refers to the base type table entry LOID (SYS__LOID) value for the base instance in the referenced table. When the data type inherits exclusive from its base type, attributes of the base type become attributes of the inheriting data type.

Enumeration values are stored as numerical data. Lookup tables are generated and filled with the enumeration name. Enumaration tables contain two attributes: code, name.

Enumerator attributes get a reference to the enumerator table.

References without collection identity (single references, single not updatable relationships)

typed collections create a link attribute with the reference/relationship name. The link attribute creates a reference to the target type table.

ALTER TABLE "GeoNameLand" ADD ("kontinent" NUMERIC(20,0) REFERENCES "GeoNameKontinent");.

Weak- or untyped collections create a link with the reference/relationship name without reference.

Generic attributes are considered and handled as references.

All references and relationships, which have got a collection identity (multiple and updateable references)

are stored in the table as attributes with its property name proceeded by double underscore (children --> __children), which contains the identity referring to the referenced collection.

ALTER TABLE "GeoName" ADD ("__timezone" NUMERIC(20,0));

primary relationships will create a m:n relationship table constructed from the current type name and the relationship name (cars --> Person__cars)

CREATE TABLE "Person__cars"

(

"SYS__LOID" NUMERIC(20,0) NOT NULL REFERENCES "Person",

"SYS__REF" NUMERIC(20,0) NOT NULL REFERENCES "Car",

PRIMARY KEY ("SYS__LOID","SYS__REF") USING INDEX TABLESPACE "ODABA_INDEX"

);

Types referred to in references and owning relationships will get an additional owner attribute

ALTER TABLE "Car" ADD ("cars__Person" NUMERIC(20,0) REFERENCES "Person");

MEMO (CLOBs - large character objects) fields and BLOBs (large binary objects) are stored in two separate tables - (SYS__BLOB and SYS__MEMO) - which consist of the SYS__LOID attribute and a CLOB or BLOB field with name SYS__ENTRY.

One to many relationships are stored as attributes with the property name of the reference or relationship. The (link) attribute contains the SYS__LOID value for the referenced instance.

References, which always have one owner but no link field to it, will get an additional owner attribute with the name SYS_OWNER that contains the SYS__LOID value of the owning instance.

Many to many relationships and singular updateable relationships can be stored optionally in additional tables, which get the name according to the involved primary relationship name and the data type defining the primary relationship separated by '__' (e.g. Person__children).

The last rule is optional, since in an object modelthere are often more than 10 many to many relationships defined for an objecttype, i.e. the generated relational database might contain e.g. 100 data tables, but 1000 relationship tables in addition. Maintaining all those tables might become a performance problem. Thus, the administrator may choose when generating the relational model (table statements), whether to support many to many relationships in the relational data storage or not.

Limitations

Running ODABA with a relational database underneath includes some restrictions. The first and most important one is, that the relational data storage might be accessed by SQL tools in order to perform queries, but not in order to update the database. All update operations must pass through the object manager. Otherwise, the Object Manager database might become inconsistent.

Since relational databases usually do not support namespaces for tables, data model definitions running with relational data storage must not define persistent namespaces. Instead, type names should be prefixed or marked in any other way. In the model definition, you may define object types in modules or namespaces, but those must not be marked as active namespaces, i.e. type names must be unique with the dictionary.

In order to guarantee proper maintenance of inverse relationships, ODABA supports updateable relationships. In a relational database, updateable relationships

behave similar as many to many relations. This means that queries against the relational database must include an additional join operation when referring to singular links.

Property names in exclusive base types must be unique in order to avoid naming conflicts.

Names of complex attributes are resolved. In case of deep nesting, this might exceed name length limits in the target system. Hence, attribute nesting and name length should be selected in a way that meets the target system requirements.

Instance versioning is not yet supported for relational storage.

Other limitations are of minor importance. There are several features that require specific ODABA storage. Thus, when using workspaces, all workspace data is stored in ODABA databases and is accessible via SQL only, when the workspace data has been consolidated to the root base.

Similar, long external transactions require an external ODABA transaction database and data becomes available only after committing the external transaction.

__IDENTITY values are the base for all links and instance identification and must not be changed after being created.

Implementing an access package

Implementing an access package for supporting another not yet supported type of relational database means implementing an RDB access package, which inherits from RootBase_RDB. The SQL_RootBase package provides some basic functionality that is helpful for most RDB access packages (conversion tables, link cache etc).

The typical implementation of an access package is documented in XSQL_RootBase class, which provides a list of functions to be implemented in order to support an SQL access package.

The access logic is mainly managed by the SQL_RootBase base class. In order to provide an enhanced access management, the following functions have to be overloaded in the access package:

DeleteInstance

LinkInstanceIntern

LocateInstance

StopCommit

UnlinkInstanceIntern

UpdateInstance

In this case, ODABA functionality implemented in this functions have to be handled with care.

Implementing an access package

Implementing an RDB access package requires overloading the following functions in RootBase_RDB:

Close

DeleteRow

EndRow

GetColumn

GetRootBase

GetRow

InsertRow

LinkInstance (referencec and relationships)

Open

RBType

UnlinkInstance (relationships, only)

UpdateColumn

UpdateRow

destructor

Instance operations are introduced by a row functionn (GetRow(), InsertRow(), UpdateRow(), DeleteRow()), which usually locate the requested row for the operation. After locating a row in a table, several column function calls are made (GetColumn(), UpdateColumn()) in order to read or update column values. Column values are provided as character data. Finally, the EndRow() function is called in order to indicate the end of row processing. The function might be overloaded in order to perform final row processing.

Appart from updating object attribute values, link information will be updated after updating attribute values in instances. In order to update link information, LinkInstance() and UnlinkInstance() have to be implemented. Both functions are called only ones for a table row in order to create or delete a parent (reference) or m:n (relationship) link.

In case of parent links, the link value has to be updated in the attribute passed to the function. In case of a relationship link, a mapping row has to be inserted into the m:n relationship table.

Optional, the following functions can be re-implemented:

LinkInstanceIntern

StartCommit

StopCommit

TACancel

TAStart

TAStop

UnlinkInstanceIntern

Data conversion

Data but also table column names require conversion. In order to convert column and table names properly, the base class SQL_RootBase provides a name conversion function Name(). From the name and the database specific maximum name length, the function constructs an appropriate database specific table or attribute name, which correspond to the name generated as table or attribute name when generating the table definition. All functions receiving table or attribute names, receive the original ODABA type or property names, which have to be vonverted to table or attribute names.

Attribute values are always passed in string formats (ASCII or Latin1) with a terminating 0. Following data formats are passed:

string - ASCII string (latin1)

integer - "[-]n*[.n*]" (decimal point according prcision definition)

float - "[-]n*[.n*][E[-]n*]"

time - "hh:mm:ss,hs"

date - "yy-mm-dd"

datetime - "yy-mm-dd hh:mm:ss,hs"

guid - "A-xxxxxxxx-xxxxxxxx-xxxx-xxxx-xxxxxxxx"

Values have to be passed in both directions referring to the same format, i.e. the access package will obtain values in the format above when updating columns and has to return values in an approriate format when reading values.

Transaction management

Transaction management is mainly organized on ODABA level, i.e. a request of storing instances to the database is submitted by ODABA only, when committing a transaction. Thus, all update requests are send to the root base in the commit phase.

There are, however, RDB specific requirements passed to RootBaseRDB while a transaction is running. Thus, LinkInstance() and UnlinkInstance() requests are sent while running a transaction and will be cached by the RootBase_RDB.

As long as StartCommit() has not been called, the access package can assume, that access is read-only. This is true also after EndCommit(). TACancel() will empty the link cache in case of a cancelled transaction.

When committing a transaction, StartCommit() is called in order to indicate the beginning of the commit request. EndCommit() indicates, that committing data has been finished. Between StartCommit() and EndCommit() all Updated() requests are submitted by ODABA. Update() requests before StartCommit() and after EndCommit() are illegal and must not happen.

By default, link requests are submitted in the EndCommit() function. The function reads all link and unlink requests from the cache (link_cache) and call LinkInstance() or UnlinkInstance() in order to handle the request. Those functions must be overloaded in the appropriate access package.

For write optimization, it might, however, be more efficient processing the link cache in the access package. in this case, outstanding link requests must be written to database before terminating the commit phase. Link requests can be obtained from the link cache (link_cache.RemoveHead()).

Create, delete and update instance

New entries are usually created via an update request. In order to distinguish new instances from old instances, the data position (acb::GetPosition()) can be checked. In case the position is 0, the instance is considered as new instance. In order to mark the instance as existing after creating is, the position should be set to a positive value (loid is suggested).

In order to maintain update counts, SQL_RootBase::Update() should be called after each update operation, as well as SQL_RootBase::Delete() should be called at the end from the overloaded Delete() functions.

1.9.2 XML database

ODABA provides features for accessing XML files like an ordinary ODABA database. The idea is not maintaining persistent data in an XML file, but opening the posibility accessing XML data by the same means as accessing an ODABA database.

XML schema extensions

ODABA schema definitions require some ODABA specific schema extensions. Schema extensions are available at www.odaba.com/OXMLExtensions.xsd. Using this schema extensions allow providing complete schema definitions via an XML schema.

A summary of ODABA XML schema extensions is given in the definition below.

1.10 Internet Communication Engine

Ice is our decision to bind several external programming languages to our cpp library. These languages are cpp, java, .net, visual basic, python, php and ruby. The Ice development is ongoing and it is likely that a new succeeding language will be implemented.

The decision is in favour of the still existing COM interface as our unix support evolved and recent webapplications are requested as linux installations.