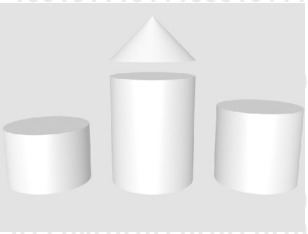


ODABA 9

User’s Guide

run Software-Werkstatt GmbH
Köpenicker Strasse 325
12555 Berlin

www.run-software.com

Tel: +49 (30) 65762791
Fax: +49 (30) 65762791
e-mail: run@run-software.com

Berlin, October 2003

Content

run-products comprises a range of independent but co-ordinated modules, which deal with databases and environments for software development. Within the scope of this range, **ODABA** is an object-oriented database including a software-developing environment for the MS-Windows-System.

run-products can be used for the creation of applications and for general software solutions without any additional authorisation. Nevertheless, software that is developed under the application of **run-products** should have a clear remark in a conspicuous place like “Made under application of **run-products**“.

Introduction

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

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.

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 XML definitions.

Accessing the database

This chapter explains the necessary steps for accessing the database.

ODABA Client Handle

Dictionary Handle

Database Handle

Database Object 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.

Transient Reference Properties

A reference can be defined as transient reference in an instance context. Transient references may contain a reference to another property, which can be set with SetTransientReference(). You may also create transient collections referenced by transient references.

Transient property handles are used to provide context specific or derived information.

Transient property handles should be initialised when opening or reading the instance containing the property handle. When, e.g. creating a new collection for each instance you can overload the DBRefresh() function in the context of the transient property to update the collection.

When creating copy handles from transient handle the refresh event is generated only ones, while open and close events are generated for each copy of the property handle.

SetTransientReference

As property handle in an instance transient reference properties are part of the instance and created and deleted with the instance. Within the application you may create temporary collections or associate other references with the transient reference using the SetTransientReference() function.

Metadata

All metadata-functions for transient references are returning information for the associated property handle and not for the transient reference node, i.e. when defining as type for the transient reference property “Person” and associating the property handle with a collection of Companies the GetType() function will return “Company”.

Opening the database

Opening the dictionary

Opening the database

Opening database objects

Opening property handle

Using data sources

Reading data from the database

Read by position

Read by key

Filter for Property Handle

Property handle provide two ways of filtering instances in a collection. One way is by setting a filter condition for the property handle (SetSelection()). The other possibility is to set an instance or key to hidden when reading an instance.

Expression selection

The SetSelection() function allows using an expression for selecting valid instances from a collection. An instance is accepted, when the expression passed to the property handle, returns true. Otherwise the instance is not displayed. This means, that the Get() function returns no instance. The Position() function or ++ and – operator skip non-selected instances. FirstKey() and NextKey() will skip non-selected instances an LocateKey() will return an error when the instance does not fulfil the selection criteria.

```
char            path = "Person{age > 18}.children";
PropertyHandle  ph(database,"Person",PI_Read);

Ph.SetSelection("age > 18");
```

Count

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

Context selection

Context selection is mainly based on the DBOBeforeRead() handler that handles the DBP_Read event. This event is generated 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 DBBeforeRead() handler can be overloaded to supress instances, that do not follow certain conditions. Since instance information is not available, the check can be made based on data of the key instance or the identity, only.

For checking instance data the overloaded context function may either use directly the key as returned by the GetKey() function.

```
Logical sPerson:: DBBeforeRead()
{
  char    *keyarea = GetKey();

  return( ( *keyarea <= 'P' ? YES : NO); // YES for Error
}
```

Another possibility is moving the key o the instance and using property handle functions to check the key component values.

```
Logical sPerson:: DBBeforeRead()
{
  PropertyHandle    ph(GetPropertyHandle());

  SetKey();
  return( ( ph <= 'P' ? YES : NO); // YES for Error
}
```

Since the event is generated when providing key values and before reading an instance, in some cases the event might be generated twice (when locating the key first and then reading the instance). This happens, e.g. when combining key access and expression selection (see Key check).

Hiding instances

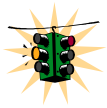
Instead of expression or context selection, the 'hidden' state for an instance can be set when reading the instance. The hidden instance is a context state that can be set using the HideInstance() context function in the structure context of an instance. This can be done by defining a structure specific context class derived from CTX_Structure and using the DBRead() function that handle the DBO_Read event. This event is generated after reading instances. At this time all the properties of the instance do have the values read from the database. Thus, conditions referring to other properties than key components can be checked in this case.

```
Logical sPerson::DBRead()
{
  if ( condition )
    HideInstance();
}
```

Hiding the instance will suppress the selection of the instance when trying to read it with Get(). When using Position() or the ++ or – operator, hidden instances are skipped.

The ‘hidden’ state is automatically reset when the selection in the property handle changes. It is, however, also possible to reset the ‘hidden’ state using the context function ShowInstance().

The ‘hidden’ state is inherited for derived structure instances.



For hiding instances while accessing keys (e.g. with NextKey() or LocateKey()), the ‘hidden’ state must be set in the DBBeforeRead() handler, since this is the only event that is generated when reading keys.

Key check

In many cases a selection condition is based on properties, which are key components. Unfortunately, key functions as NextKey() or FirstKey() will 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 to check the filter condition to check whether the key refers to an accepted instance or not, which reduces the access efficiency dramatically. This is true for context selection as well as for expression selection.

```
char            path = "Person{age > 18}.children";
PropertyHandle  ph(database,"Person",PI_Read);

ph.SetSelection("name > 'P'");
ph.SetOrder("sk_name");
ph.EnableKeyCheck();
```

You may use the EnableKeyCheck() property handle function to optimise expression based on key instance data. 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. When the expression refers to non-key properties, the initialised instance values are used for computation, i.e. the function does not check whether the expression refers to other properties that key components.

Special Identifier

Write to database

Create objects

Update objects

Rename and duplicate

Using transactions

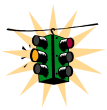
Access via Property Paths

Accessing data via property paths allows defining a simple type of view to the database. Via property paths you may access single instances as well as collections.

In contrast to views a property path provides a collection of instances according to the last property referenced in the path, while in a view each element of the path can be involved. This means accessing instances via a property path will navigate along the complete path but selecting instances in the last property handle, only.

In general a property path returns the union of object instances on the lowest level for all objects in the collection on the next higher level. When the referenced collections are not distinct the union set defined by the path may contain duplicates.

It is also possible defining selections via property paths or generic properties.



Property paths can be used for reading and updating instances in the result set of the path. It is also possible to delete selected instances in the path or creating new instances for the path. This, however, is not suggested, since in a path it is not obvious which hierarchy of instances is currently selected.

Accessing path

Mainly navigation functions as GetCount(), Position(), Get(), ToTop(), Cancel(), ExtractSortKey() and operators ++ and – work differently when being used for a property path. Other PropertyHandle functions apply to the property at the path end and work as usual.

Definition

Defining a path to a property is different from a property path. The property handle with the property path

```
PropertyHandle  ph(pers.ph,"children.cildren");
```

Returns the property handle for the given path only, i.e. a path property handle that will work only on the collection of the last property. Higher properties in the collection must be positioned by the application.

The following property path, however,

```
PropertyHandle  ph(pers.ph,"children().cildren");
```

will navigate through the complete hierarchy, i.e. is will provide the children for all children. Parenthesis () after a property indicates the start (top) of the navigation path.

The following path property

```
PropertyHandle  ph(dbhandle,"Person.children().cildren");
```

will provide the children for all children for the selected person. The person has to be selected by the application, i.e. the path starts with children(), which initiates the path.

Sub-handles

When defining a path property handle relatively to another property handle (sub-handle), the path property handle will not change the selected instance in the parent property handle.

```
char            path = "children().children";
PropertyHandle  ph(pers_ph,path,PI_Read);
```

This path will return only the grand children for the person currently selected in pers_ph.

Navigation

Navigating in a path property is possible by absolute position (Get(position)), by iteration (Position()) or by key (LocateKey()).

Position

Positioning an instance in a property path includes all collections that are referenced in the path. When the end of a collection is reached the path locates the next instance on the upper level and starts with the next collection. Therefore, Position() or ++ and – are optimal for processing a path completely.

Get

The Get(position) function locates the n-th instance in the virtual path collection. The function can be used to go through the function, but the position may change when one of the involved collections changes.

Hence, it is suggested to use rather Get(key) to locate a specific instance. The key passed to the path property must contain key components for each path component.

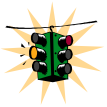
```
PropertyHandle  ph(database,"Person().children",PI_Read);
ph.Get(ph.StringToKey("{Miller|Paul}|{Miller|Anna}"));
```

Locate key

You can locate a key in the path by passing a composed key according to the keys for the path components. This way you can start processing the path at a certain position.

```
PropertyHandle  ph(database,"Person().children",PI_Read);
char key_area = "Miller     Paul     Miller    Anna     "
ph.Get(key_area);
```

To locate a certain instance in the path the path key can be extracted with ExtractSortKey(key area). You must pass a key area in this case. The key returned from a path consists of all keys along the path.



The key must be provided according to the key structure, since KeyToString() will not work on the path, but on the last property, only.

Union

By defining a property path that consists of a number of reference properties it is possible to get the union of all instances from the collection defined at end of the path.

```
char            path = "Person().children.children";
PropertyHandle  ph(database,path,PI_Read);
```

This property path will provide all grand children for all persons.

Filter

You may use filter for any component of the property path. A filter is either a single value (character key or position) or an expression. While character keys have to be enclosed in single quotes (') expressions must not be enclosed in any type of quotes. When the selection string starts with a numeric character it is interpreted as position.

Selection by path

The property path provides a simple way of selecting instances from a collection by means of filter expressions:

```
char            path = "Person{age > 18}.children";
PropertyHandle  ph(database,path,PI_Read);
```

This property handle refers to all children for persons older then 18 years. Filters are enclosed in {} and must be defined as valid expressions. As well as () in the path a filter expression will start a path.

Selection by key

A property path allows also selecting instances by key value or position:

```
char            path = "Person('Miller|Paul').children;
PropertyHandle  ph(database,path,PI_Read);
```

In this example the property handle provides all children for Paul Miller. This is the same as defining an appropriate selection expression {name='Miller' and first_name = 'Paul'}, but key search is much more efficient than expressions are.

Selection by position

A property path allows also selecting instances by key value or position:

```
char            path = "Person(0).children;
PropertyHandle  ph(database,path,PI_Read);
```

In this example the property handle provides all children for the first person in the person collection.

Generic properties

Property paths allow defining generic properties. Generic properties can be provided in a property handle by key or number, depending on the definition of the collection for the property. Typically, they are provided by key where the key value is the name of the generic property.

When the collection the path component is referring to is ordered by key, the key must be passed enclosed in single quotes. When the collection is unordered, the position must be passed as numerical value without quotes.

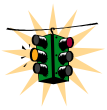
Example: Defining a property notes for person as a collection with any number of notes, the application can define different types of notes like

Experience
Praxis

and so on.

```
char            path = "Person.notes['Experience']";
PropertyHandle  ph(database,path,PI_Read);
```

In this example the property handle refers to the "Experience" property for the selected person. When the property does not exist it will be created. Usually, generic properties are defined at the end of the path.



When using auto-keys or auto-positions the property handle can be positioned only to instances with the defined key(s). When the instance should not automatically be provided, it is also possible to define the key in (). Then the system tries only to locate the key.

GetPropertyHandle

You may use path definitions also in the GetPropertyHandle() function. In this case you may pass instance selections by key or position enclosed in () or providing instances by key or position enclosed in [], but you cannot pass filters with the path definition.

```
char            path = "Person.notes['Experience']";
PropertyHandle *ph = GetPropertyHandle(path);
```

Using GetPropertyHandle will reuse the property handles in the hierarchy, i.e. no copy handles are created as for the path property. Thus property handles share the access nodes with other property handles provided with GetPropertyHandle(), which may influence the instance selection in any node in the path.

Passing a path to GetPropertyHandle() will not create a path property. It works only similar while providing the property handle, i.e. it will locate all defined handles on the way. Thus, you cannot pass filter expressions in {} but only position information in () or [].

Access via Views

Accessing data via views is a way to define a collection as well as a number of data for each instance in the collection. A view is similar to a structure in a sense that is consists of base structures, attributes and references. The view allows creating a flat structure from different instances in a database. Thus, views are an efficient way for displaying tables.

View Base

The view base defines the instances involved in the view. There are different ways to define the instances included in a view. If there are two extents as “Person” and “Company” the comma operator (,) creates the product set of persons and companies:

```
"Person,Company"
```

The other way to define joined collections is via relationships:

```
"Person.company"
```

would define a collection containing all persons that are associated with a company and the associated company. The dot-operator (.) can be used for defining a property path with any number of linked relationships. The property path

```
"Person.children.children"
```

would define a collection of all persons that have grandchildren including the instances for the person, the child and the child’s children.

You might combine the dot and the comma operator but the dot-operator has higher priority.

Each view instances consists of all instances involved in the view, i.e. data from all instances referenced in the view base is potentially available.

View Properties

While the view base defines the vertical dimension of the view the view properties define the horizontal extension of the view. View properties can be attributes as well as references or relationships. Within the view references and relationships are considered as collection references.

View properties are defined by a view property name and the source the property is taken from:

Property Path

You can refer to any property that is part of an instance included in the view base by means of a property path. Since instances of the view base are considered as base structures it is usually not necessary to prefix them. Thus you may refer to the companies address or persons first name just as:

```
"company_address", "address"
"person_fname", "first_name"
```

Only when names are ambiguous you have to add the instance that contains the property as prefix:

```
"person_name", "Person.name"
"company_name", "Company.name"
```

You may also refer to properties in related instances. When, e.g. a person is associated with a building that has not been included in the view base you may refer to the buildings identifier as

```
"building_id", "building.identifier"
```

When referring to properties in related instances the information is always taken from the first instance in the collection.

Expression

You may also refer to expressions that define a derived property. Expressions may refer internally to all properties that are part of the view base. Expressions cannot refer to view property names, i.e. instead of referring to "person_name" (view property name) you have to refer to "Person.name", the original property source.

Expressions may result in attributes or collection references.

Special properties

There are some properties, which are implicitly defined and available for each instance selected in a property handle.

__SORTKEY

Including __SORTKEY in the view will return the key value for the instance. The key value is returned according to the internal key structure as char* value and can be used to locate an instance in the view.

__SORTKEY_STRING

Including __SORTKEY_STRING in the view will return the key value in string format for the instance, i.e. key components are separated by '|' and non-character values are converted into character. The string key can be used for displaying but not for locating an instance in the collection. For locating an instance in the view the key has to be converted into the internal key format before (StringToKey() function).

__IDENTITY, __LOID

The property contains the local unique identifier for the instance, which is provided passing this attribute.

__GUID

The property contains the global unique identifier for the instance. When the instance is not equipped with global identifiers the local identifier within the database is returned.

__TYPE

The property contains the current type of the instance. This may differ from instance to instance in case the instance is element of a weak- or un-typed collection.

Filter

Filters can be defined for restricting the number of instances in the view. Filters are defined by means of expressions. You may define filter for each instance in the view base or for the whole view.

Instance filter

When defining a filter for an instance in the view base the instance must be addresses by a property path that is part of the view base definition:

```
"Person.cildren", "age > 10"
```

means the view will contain only instances where the children are older than 10 years. In this case the filter expression refers to exactly the instance (structure) that is referenced by the path, i.e. it may refer to properties in the referenced instance, only.

One filter expression can be defined for each instance included in the view. Instance based filter conditions are combined by logical "AND".

Pre-condition

Moreover, one filter condition (pre condition) for the whole view can be defined. The expression for the total filter may refer to any property of the involved instances by the property name in the instance (no view property names).

Pre-conditions are the only way to define a condition across all included base instances of the view.

When names are ambiguous they must be prefixed by the property path of the view base referring to the instance (as “Person.name”).

Post-condition

One filter can be defined as post condition, which is defined as an expression based on view properties. This expression is checked after all derived values in the view have been calculated. The post-condition may not refer to properties in view base instances.

Order

Usually the view instances are ordered according to the primary order of the involved collections. I.e., the view “Person.children” is ordered by the primary order of the “Person”-extent (e.g. “pers_id”) and by the primary order for children in Person (e.g. “first_name”).

You may change the order for each instance included in the view base by defining another valid order for the referenced collection.

Universe

The views universe is either the database or an instance selected in any property handle. When the view is defined relatively to an instance you may refer to any collection within the instance. In this case the view consists of the instance handled by the property handle and the instances included in the view base.

Defining a view base relatively to “Person” as:

```
"parents,cildren"
```

would create a collection for each person that contains a combination of each parent with each child. In this case the collection defined by the view base will change each time another instance is selected in the parent property handle.

Nested Views

Since views can be defined relatively to a collection handled by a property handle you might define nested views. A nested view can be defined by defining a view relatively to another view. Any nesting level is allowed.

Access mode

Views can be defined in read, update or in write mode. Opening the view in write mode will lock all instances involved in the view as long as they are selected for the current view instance.

You may update all properties referenced by means of property paths, but you cannot update view properties referred to as expressions.

Property in a view are updated as normal properties just by assigning the new value to the appropriate property of the view instance.

Defining a view

You can define a view as part of the data model. In many situations, however, it is more appropriate to define the view at run-time. Views defined at run-time are considered as transient and only available as long as the process defining the view is alive.

Create View

Transient views are defined in several steps. The first step is defining the view base and the universe:

```
DBViewDef  view_def(dictionary,"Person.children.cildren");
DBViewDef
rel_view(dictionary,"parents,children","Person");
DBViewDef  simple_view(prop_handle);
```

Structure based view

Structure based views are based on a structure that defines the universe for the view ("Person"). In this case the property paths defining the view are properties of the structure passed to the view definition. Structure based views are defined in the context of a selected instance of the given structure, only.

Simple view

A simple view is just based on a given collection. This collection defines the view base, which consists of instances of the passed collection.

Define properties

At least one view property should be defined for the view. View properties may refer to atomic or complex attributes as well as to references. View properties can also be defined by means of expressions:

```
view_def.AddProperty("person_name",
                     "Person.name",
                     ADT_PropertyPath);
view_def.AddProperty("parents",
                     "Person.mother+Person.father",
                     ADT_Expression);
```

When defining properties you must refer always to the property names in the original instances (view base).

Change size

In some situations (especially for large text fields) it might be useful to change the size of the property in the view. This can reduce the traffic extremely. Thus you may extent the view property definition by passing the size of the field in the view.

```
View_def.AddProperty("person_name",
                     "Person.name",
                     ADT_PropertyPath,
                     40);
```

When the referenced property is defining an array the size applies to each array element.

Set filter

Instance related filter can be defined for each instance defined in the view base:

```
view_def.AddFilter("Person",
                   "name = 'Miller'");
view_def.AddFilter("Person.children",
                   "age > 10");
```

Filter expressions are always defined in the scope of the referenced view base instance.

Set pre-condition

A pre-condition is defined based on all view base instances:

```
view_def.SetPreCondition("Person.name = 'Miller' and
                          Person.cildren.age > 10");
```

Pre-conditions are always defined in the scope of the view base instances. The view base instance path must precede ambiguous property names.

Set post-condition

A post-condition is defined based on the view, i.e. on the view properties. It will be checked after creating all view properties.

```
view_def.SetPostCondition("person_name = 'Miller');
```

Post-conditions are always defined in the scope of the view. When the view contains collection references the expression may refer to instances of those collections as well.

Set order

Setting the order for view base instances is possible as soon as there are several orders defined for a collection included in the view base.

```
view_def.AddOrder("Person.children","sk_age");
```

Referenced orders for the collection can be defined as persistent order as well as temporary ones.

You can achieve the same result after creating the view property handle by setting the order for the view collection directly:

```
DBViewDef       view_def(…)
PropertyHandle *ph;
                                                    ph      =
view_def.GetPropertyHandle(“Person.children’);
         ph->SetOrder(”sk_age”);
```

Accessing data via views

Views are considered the same way as other collections and handles by PropertyHandles. Property handles can be defined for views defined in the data model. In this case there is a view extend defined that is just opened as a normal database extent.

Create Property-Handle

Creating a property handle for an internal view is possible via an appropriate view property handle constructor. When opening the view all computations and filters are prepared for execution.

```
PropertyHandle          view_ph(database,view_def,PI_Write);
SDBCERR
```

When creating a property handle for an internal view definition the property path definitions are checked as well as the definitions of expressions for derived properties and filters. Invalid expressions or property paths will result in an empty view definition. Hence you should check the database error before using the constructed view property handle.

Structure based view

When creating a property handle for a structure-based view a property handle referring to a collection with instances of the given structure must be passed to the view definition.

```
PropertyHandle          view_ph(view_def,prop_hdl,PI_Read);
SDBCERR
```

The instance selected in the property handle passed to the view determines the view base, i.e. changing the selection in the property handle passed will change the view base and reset the view.

Simple view

Creating a simple view is based on a property handle that defines the collection representing the view base.

```
PropertyHandle                  view_ph(simple_view,accopt);
SDBCERR
```

For accessing the base instances in the view a copy handle is created from the property handle passed to the constructor so that the current selection in the original property handle will not influence the current selection in the view. The original handle, however, must not be closed or destructed before the view property handle has been destructed.

Accessing instances

A view property handle is considered in most aspects as normal handle, i.e. you may use all navigation function for selecting a view instances.

Get

Get by index will return a view instance only when the instance is

Buffered Access (block 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 read will reduce the number of packages sent between client and server and thus, reduce the communication time extremely.

Read what you see

In many cases only a subset of instances in a collection is displayed on the terminal. 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.

Update notifications

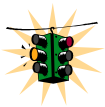
When reading instances in block mode all instances in a block are registered as being used by the property handle. Thus, the property handle will be notified when one of the instances in a buffer has been updated.

Enable block mode

You can activate the block mode simply by setting a buffer size:

```
PropertyHandle  ph(database,"Person",PI_Read);
Int buf_count = ph.ChangeBuffer(50);
```

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 ini-file variable "MAX_BUFFER_SIZE".



You can activate the block mode simply by setting a buffer size. Block mode can be set for instances opened in read mode, only. For updating instances selected from a list a copy handle can be used, which is opened in write mode.

```
PropertyHandle  write_ph(ph);
write_ph.ChangeMode(PI_Write);
```

Update instances

The block mode has been provided mainly for read mode. But it can be used in update or write mode as well. When using buffered access in update mode the application should ensure that the instances in the buffer are up-to-date.

Special events

When using block mode read events for the instances as well as for the property 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. To check whether the read event has already been executes the DateState can be checked in the read event handler (see DBO_Read event).

Activate Server Cache

The server cache can be activated on the server side to minimize read access to the database. When using the server cache instances are cached in memory for reuse.

The server cache can be activated for read and write access. The cache parameters are passed via an ini-file (usually Server.ini).

Activating cache

Activating the cache requires a [CACHE] section in the ini-file that is passed when starting the server.

In this version it is not possible to define different cache options for involved databases, i.e. the cache is active for all or for none database. This means you must multiply the cache size with the number of involved databases since each database will use its separate cache.

```
C:\ODABA\server.exe c:\ODABA\server.ini
```

Cache lock

When using database cache features the database can be used by one application, only (e.g. the server). Since not all operating systems guaranty exclusive access to the database an indication is written to the database prohibiting other applications to open the database, the **cache lock flag**. In case of system crash the cache lock flag remains and the database cannot be opened anymore. To open the database regardless of the state of the cache lock flag you can define

IGNORE_CACHE_LOCK=YES

in the [CACHE]-section of the ini-file.

Cache size

The read cache buffers instances until the maximum cache size is reached. The maximum cache size is passed in megabytes in the SIZE parameter.

SIZE=500 will reserve 500 MB for the cache.

Since the cache is build up dynamically caching dictionary information will usually not exceed half of the dictionary database size.

Free area

When the maximum cache size is reached the cache will be reorganizes removing the entries not used recently. The size of area to be released in the cache can be defined by the RELEASE parameter in the [CACHE] section:

RELEASE=30 will release 30% of the total cache size.

The default is RELEASE=25.

Write cache

When not using a write cache modifications are stored immediately to the database. The write cache will hold the instances in the cache writing modifications from the cache in defined time intervals to the database.

Check interval

The interval in seconds for checking the cache for updates is defined in the CHECK variable in the [CACHE] section:

CHECK=120 means checking the cache each two minutes.

Default is CHECK=0, i.e. no write cache.

High check values will improve the performance but reduce the security since modifications that are stored in the cache only will get lost when the system crashes for some reason. Suggested check values are between 60 and 300 seconds.

Save cache

When writing back the cache to the database there is a risk of serious damage when the system crashes in this phase. To avoid this risk the cache can be written to an intermediate file before being stored to the database. The path for the intermediate file is defined in the INTERMIDIATE variable in the [CACHE] section of the ini-file.

INTERMEDIATE=c:\temp\update.tmp will write updates to c:\temp\update.tmp first.

When the system crashes while creating the intermediate file the modifications in the cache are lost but the database is still consistent. When the system crashes while the data from the cache is stored to the database the database is not consistent but the modifications can be restored from the intermediate file. In this case you must start the server with the same INTERMIDIATE file path. The system automatically detects outstanding update requests and repairs the database before starting the server.

Statistics

You may request cache statistics. This will slightly decrease the system performance but it allows you checking the cache performance. For activating cache statistics

STATISTICS=YES

must be defined in the [CACHE]-section. You can use the SystemMonitor to display cache statistics.

Application programming

To activate the cache in an application programme you have to pass the ini-file to the local ODABA client. Moreover, you must activate the shared rootbase option that enables several threads using the same database:

```
ShareRootBase();
ODABAClient
local_client("E:/BridgeNA/Server.ini","Test");
```

Special features

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

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. WEB applications typically refer to local object identities.

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

LOID

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

LOIDs will 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 GUID, 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:

```
PropertyHandle   pers_pi(dbhandle,”Person”,PI_Read)
                  pers_pi.Get(pers_pi.StringToKey(“Miller|
Paul”));
            pers_pi.GetLOID();
```

Another possibility is using the GetString() function that returns the LOID as string:

```
PropertyHandle   pers_pi(dbhandle,”Person”,PI_Read)
                  pers_pi.Get(pers_pi.StringToKey(“Miller|
Paul”));
            pers_pi.GetString(“__LOID”);
```

Access via LOID

For accessing instances via local object identities you need to create a so-called LOID-property handle that is able to access instances outside any context only by local object identity.

```
PropertyHandle   loid_pi(dbhandle,"__LOID",PI_Read)
            loid_pi.Get(loid);      //   passed   as   num.
parameter
```

You may access instances via a LOID-property handle in update mode as well. In this case, however, you cannot update any key field, i.e. any field that acts as key component in at least one key defined for the structure of the instance.

GUID

Global object identities can be used to access data instances within the ODABA database and to identify database instances globally. Thus, they are typically passed to other applications as instance references in the database. GUIDs are stable and will not change even when copying an instance or the whole database.

Getting LOID

The global object identity can be provided in different ways. The GUID is a string value that can be retrieved from the property handle for the currently selected instance:

```
PropertyHandle   pers_pi(dbhandle,"Person",PI_Read)
                  pers_pi.Get(pers_pi.StringToKey("Miller|
Paul"));
            char *string = pers_pi.GetGUID();
```

Another possibility is using the GetString() function that returns the GUID as string:

```
PropertyHandle   pers_pi(dbhandle,"Person",PI_Read)
                  pers_pi.Get(pers_pi.StringToKey("Miller|
Paul"));
            char *string = pers_pi.GetString("__GUID");
```

Access via GUID

For accessing instances via global object identities you need to create a so-called GUID-property handle that is able to access instances outside any context only by local object identity.

```
PropertyHandle   guid_pi(dbhandle,"__GUID",PI_Read)
             guid_pi.Get(guid);      //   e.g.   passed   as
parameter
```

You may access instances via a GUID-property handle in update mode as well. In this case, however, you cannot update any key field, i.e. any field that acts as key component in at least one key defined for the structure of the instance.

Create GUID

GUIDs are supported only when the structure is derived from __OBJECT, which is a system-defined structure that contains a member with the name __GUID (char 16). __OBJECT must be defined as base structure of the structure or of an imbedded (not shared) base structure. Shared base structure instances will create their own GUID, which is different from the GUIDs in derived instances.

GUIDs are created automatically when an instance is created. When, however, the GUID option is not set in the structure definition or in the collection definition where the instance is created the GUID must be created explicitly. This happens when the instance has been provided in update or write mode and the GetGUID() function is called.

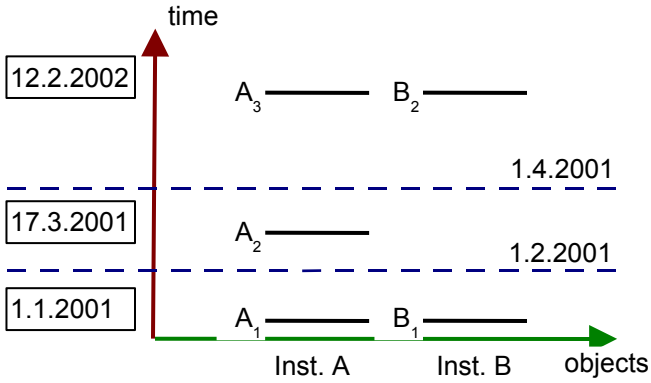
```
PropertyHandle   person(dbhandle,"Person",PI_Write)
char            *string;
            person.Get(FIRST_INSTANCE);
            string = person.GetGUID();
```

The result is passed as usual in the result area of the property handle and is valid only until the next property handle function call.

Versioning

Versioning

Database versioning is a feature that allows preserving old states for persistent instances. Versioning adds a time dimension to a database instance and creates new versions whenever required. Usually when reading an instance the current version will be provided. It is, however, possible to access data for older versions by passing the requested version as date or time stamp. The system then selects the instance version that was valid at this time.



Supposing that new instance versions for A have been created at 1.1.2001, 17.3.2001 and 12.2.2002. For B new instances have been created at 1.1.2001 a,d 12.2.2002. Reading instance A and B without version request will return the current version of A and B as A_3 and B_2. Requesting A and B as have been valid at 1st April 2001 A_2 and B_1 will be provided. Requesting A and B for 1st February 2001 A_1 and B_1 will be provided.

Versioning types

Modifications in an instance may affect indexes where key values may change. Moreover, deleted instances must be still visible for older versions. Hence, versioning must also include the states of indexes, collections and relationships.

ODABA supports consistent versioning, which includes versioning instances as well as indexes, collections and relationships. This can be done in two ways:

- automatic versioning (version slice)
- application controlled versioning (instance versioning)

These two concepts can be used alternatively, i.e. you cannot use sliced versioning and application controlled versioning at the in one database.

Version Slices

Introducing a version slice in the database will preserve the complete database at the defined time point. All modifications of instances that are stored before the date of the new version slice will lead to new versions for the instance and affected indexes or relationships. New versions will be created only for those database entries that are updated. Practically, you cannot change instance states “below” the version slice.

Create new version

Version slices can be defined for database objects or in case of single object database for the database (or the root object of the database):

```
DatabaseHandle   dbhandle(……)
            dbhandle.NewVersion(dttm(dbdt(1,1,2003)));
```

The version created gets an internal number that can be used to identify the version later. You can retrieve the number for the last version with:

```
            Version_nr = dbhandle.VersionCount();
```

Change starting date

You can change the starting date for the version created using the ChangeTimeStamp() function as long as the starting date for the version has not been reached.

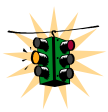
When the starting date for a new version slice is reached the database will automatically switch to the new version (in this case at 1st January 2003). Application that are running at this time will change the version when being restarted or when setting the new version value explicitly in the application:

```
            dbhandle.SetVersion(dbhandle.VersionCount());
```

Access older versions

You can use the SetVersion() function also for requesting an older database state in your application. In this case you only need to pass the date for which you want to see the database.

```
dbhandle.SetVersion(dbdt(1,4,2001));
```



Updating data in older versions is possible, but only when no newer instance exists for the data. Instances that have newer versions are provided as read-only. Sometimes, the instance can be updated (no newer instance exists) but finally the update fails, since a key attribute has been changed that requires changing an index, which has already a newer version. Hence, it is better not trying to change the past.

Instance versioning

Using instance versioning the application has to request a new version explicitly. Since versions must reflect a time relation each version is associated with a version number.

To avoid an inflation of version numbers a version granularity has to be defined for the database object or database. This is possible only as long as no version slice has been defined for the database. On the other hand you cannot define version slices after defining a version granularity for instance versioning.

Create instance version

Creating a new instance version has to be requested explicitly by the application before changing the instance:

```
PropertyHandle   pers_pi(dbhandle,"Person",PI_Read)
           pers_pi.CreateNewVersion();
           pers_pi.Save();
```

An instance has to be selected in the property handle. If there is no version defined according to the defined time granularity a new version is created for the database object.

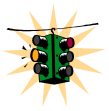
New versions are created only when there is a request for creating a new version. When, e.g. defining daily versions and no instance versions have been requested at the weekend no version numbers will be created for Saturday and Sunday.

Version inheritance

By default all affected indexes and relationships will create new versions, too. This guaranties consistency when browsing the database for older versions. You may, however, want subordinated instances to inherit the current version number as well. This is possible by defining version inheritance for the collection the instance is referenced in.

When defining version inheritance new instance versions are created automatically when updating a referenced instance with an older version. Usually the referenced instance will get the same version as defined for the referring instance (even though this might be already an old version).

Example: When defining version inheritance for employees in a company and creating a new version for the company at 10 January 2002 with a version granularity of one day, changing an employee on 15 January 2002 in the context of the company will create a new version for the employee for 10 January 2002.



When an instance is referenced in different contexts and inherits versions from different instances this may lead to conflicts since when it comes to upgrade the referenced instance, this may have already a higher version number than requested by the parent instance. In this case the modifications for the referenced instance are made at the current version since historical instance states cannot be updated.

To avoid multiple version inheritance one should define version inheritance only for references and primary relationships. Only one of the secondary relationships in a structure may have an inverse relationship that inherits version. In this case multiple version inheritance can be avoided.

Access older versions

The same way as for version slices you can the SetVersion() function for requesting an older database state in your application. In this case you only need to pass the date for which you want to see the database.

```
dbhandle.SetVersion(dbdt(10,1,2002));
```

Locking and write protection

ODABA provides several features for locking instances and protecting instances against updates. ODABA supports write and read locking for instances and collections. Write locking can be performed as pessimistic and optimistic locking.

The most common locking feature is automatic locking, which automatically locks an instance when being selected.

Moreover, ODABA provides write protection, which is similar to write locking in several situations.

Locking features

Locking features are provided for denying read or write access for an instance or collection. ODABA supports implicit (automatic) and explicit locking.

Implicit locking

Implicit locking is performed, when a property handle is opened in “write” mode. When selecting an instance in such a property handle, the instance sets a write lock for the instance. Instances are locked implicitly also, when being updated in a transaction.

Implicit locks are automatically removed (unlocked) when the process terminates.

Locking write-instances

Selecting an instance in a “write” property handle will automatically try to lock the instance for write access. When this is not successful, the instance is read-only (IsWrite() returns false).

```
PropertyHandle     students(dbhandle,”Student”,PI_Write);
PropertyHandle     name(&students,”name”);

students.Get(0);
if ( !students.IsWrite() )
  printf(“student is locked”);
else
  name = “Miller”;     // fails when being locked
```

Locking update-instances

Selecting an instance in an “update” property handle will not lock the instance. You may update instance properties also when the instance is updated by another user. When storing the changes made (Save()), ODABA checks whether the instance has been updated meanwhile by another user. If this is the case, you may cancel your changes or overwrite changes made by another user or application.

```
PropertyHandle     students(dbhandle,”Student”,PI_Update);
PropertyHandle     name(&students,”name”);

students.Get(0);
name = “Miller”;
if ( students.Save() && SDBError() == 67 )  // updated
  students.Save(YES);   // overwrite updates
```

Collection locking

When inserting or removing instances in/from a collection, collections are locked while updating the collection. This is, however, only a short period while executing the Add() or Delete() function. In contrast to instance locking collection locking waits until the collection (index) becomes available. When the collection is locked, however, in a transaction, the function fails with locking error (SDBError 6).

Implicit locking for collections makes the applications save, but may cause long time intervals for which collections are blocked for other applications, i.e. other applications may not insert or remove instances from those collections. To avoid long locking periods especially for extents, AddGlobal() can be used for creating new instances.

Explicit locking

Explicit locking can be performed for instances and collections. Explicit locking is always read locking, i.e as long as the instance is locked access to the instance is denied by other users or applications.

Explicit locking may cause wait states and should be used for short locking periods, only. For longer locking periods, we suggest persistent locking.

Explicit locks are automatically removed (unlocked) when the process terminates.

Locking instances

Locking an instance is possible, when an instance is selected in a property handle. The instance is locked until it is explicitly unlocked or another instance is selected in the property handle.

When trying to lock an instance that is locked by another application, the application waits until 10 seconds before terminating with an error. When locking an instance that has already been locked in the same application, the function terminates successfully. An instance locked any number of times can be unlocked only once.

```
PropertyHandle     students(dbhandle,"Student",PI_Write);

students.Get(0);
if ( students.Lock() )
  printf("could not lock instance");
else
{
  …                    // do something
  students.Unlock();   // release student
}
```

Locking collections

Locking a collection is possible for each property handle that refers to a instance collection or single reference. The collection is locked until it is explicitly unlocked.

When trying to lock a collection that is locked by another application, the application waits until 10 seconds for the collection before terminating with an error. When locking a collection that has already been locked in the same application, the function terminates successfully. A collection locked any number of times can be unlocked only once.

```
PropertyHandle     students(dbhandle,"Student",PI_Write);

if ( students.LockSet() )
  printf("could not lock collection");
else
{
  …                        // do something
  students.UnlockSet();  // release student
}
```

Persistent locking

Persistent locking provides a write lock, i.e. you may read instances locked persistently but not update it. Persistent locking is provided for instances, only, and allows locking an instance beyond process boundaries. Persistent locking should be used when instances are to be locked for a longer period.

In contrast to explicit and implicit locking, which automatically remove locks when the process terminates, persistent locks remain active also after process end. Thus, only the application can unlock instances locked persistently.

Persistent locking is possible only for property handles opened in write mode and having a selected instance. The instance is locked until it is explicitly unlocked (ResetWProtection()).

```
PropertyHandle     students(dbhandle,"Student",PI_Write);

students.Get(0);
if ( lock_persistent )  // lock instance
{
  if ( students.CheckWProtect() )
    printf("instance already locked");
  else if ( students.SetWProtect() )
    printf("could not lock instance");
}
else                    // unlock instance
{
  if ( !students.CheckWProtect() )
    printf("instance not locked");
  else if ( students.ResetWProtect() )
    printf("could not unlock instance");
}
```

Transaction locking

Instances or collections (indexes) updated in a transaction are also locked until the transaction it committed or cancelled (RollBack()). When accessing instances in update mode, instances are locked in the transaction, when being saved (Save()).

Instances locked in a transaction behave the same way as implicitly locked instances, i.e. other lock requests will not wait but terminate with an error immediately, when the instance is already locked by another application.

Instances in nested transactions are locked until the top-transaction is committed or until the transaction that has locked the instance is cancelled (RollBack()).

Write protection

Beside locking functions ODABA provides explicit and implicit write protection for different situations. Besides general write protection ODABA provides implicit write protection for concurrency control or locking. In some cases, write protection works hierarchical on subordinated instances.

Additionally, the application may provide write protection from within context functions. This is typically for providing access control by means of user rights.

Gereral write protection

General write protection is guaranteed, when the database or a property handle is opened in read-only mode. When opening a database handle or a DBObject handle in read-only mode, property handles can be opened also only in read-mode.

```
DatabseHandle      dbhandle(dicthdl,"ODMGSample.dat",
                            PI_Read);
PropertyHandle     students(dbhandle,"Student",PI_Read);
PropertyHandle     professors(dbhandle,"Professor",
                                 PI_Write);             //
fails
```

General write protection is suggested for all applications which support browsing functions, only, which is typically e.g. for many WEB applications.

Write lock

Different types of write-locking may prevent instances from being updated. ODABA provides instance and collection locking, but no locking on attribute level.

Write locks are checked only, when the property handle is opened in write mode (not for update mode). When a write lock is detected for an instance for any reason, it automatically applies on all instances referenced as non-updateable instances, which are usually references, or relationships without inverse relationship. For not applying write protection to linked instances from other extents, extent-based relationships should always be defined as updateable relationships in the data model.

Non-updateable instances, which cannot be updated because of its parents, are usually note locked, but just write protected.

Access control

Access control is a feature provided by means of context programming (“Database Context Programming”). Usually read-handlers (DBRead()) are used to enable or disable write access for an instance or collection.

In contrast to implicit write protection, context write protection is applied recursively to all subordinated property handles, which have not explicitly set a read-only state.

```
logical   sStudent::DBORead()
{
   SetReadOnly(!CanUpdate());
}
```

The example above sets the write permission for the student instance read (and all subordinated property ahandles) to no, when the student function CanUpdate() returns false and to yes otherwise.

This is a typical way for checking user rights in an application and setting write protectin, when the current user is not allowed updating the instance.

Context write protection is not removed when selecting a new instance and must be set explicitly by the application (context function).

Transactions

Transaction types

ODABA supports different types of transactions. In general, each database modification takes place within a transaction, Since each action, just updating a simple field in an object instance, may cause a series of consequences by reacting on events fired by the action. Therefore, each updating database action is enclosed in an **internal transaction**. Within an internal transaction, maintenance of indexes, inverse references and super sets is included, as well as all immediate reactions (usually handled in context classes). Not included in the internal transaction are delayed events handled by registered event handlers, and server events.

On the other side, there might be the requirement of having application or **user-defined transactions**. User-defined transactions are started by the application program. User-defined transactions can be nested. Within user-defined transactions, delayed events can be handles.

User-defined transactions can be defined as internal (in memory) and external (on disc) transactions.

The third type of transactions in ODABA are persistent transactions called **workspace transaction**. Workspace transactions are very long transactions, which may take days, weeks or even months. Workspace transactions can store updates from any number of processes and you may login into any number of workspaces, i.e. any number of workspaces may run in parallel.

Workspace transactions can be nested, i.e. beneath a workspace transaction any number of parallel workspace transactions can be created.

General behaviour

Regardless on the type of transaction, data stored in a transaction cannot be updated in parallel transactions, i.e. instances changed in a transaction are locked in the transaction. One may, however access data in subordinated transaction, which is locked in a higher transaction.

Collection lock

Beside instances that are locked in transactions, indexes, which have changed, are locked in the transaction as well. This means, that no parallel transaction may add or delete instances to a collection, which has been updated in a parallel transaction.

Key lock

This may cause locking problems, especially for global collections (extents), since after adding a person to a person extent, no other user would be able to add or delete another person, until the first transaction terminates. To avoid such locking problems, you may declare a collection as “shared” collection. Shared collections are not changed within the transaction, but record differences to the original collection within the transaction, only. To avoid, that different transactions violate “unique key constraints”, keys for shared collections are locked separately.

While running a database in file server mode or using workspaces, key locks are stored in a database. This makes the key lock feature a little bit slowly, which is not the case, when running the database in client/server mode.

In contrast to instance and collection locks, key locks cannot be activated by the application but are an internal feature, only.

Consistency problems

While running a transaction, other applications will get the database state without the changes made in the application. This may cause several consistency problems

Validation rules

Using transaction features requires special actions for guaranteeing logical database consistence, e.g. when using validation rules checked for the modify event..

Example: Lets assume a logical consistency rule saying that the employee must not get more money than his boss. Now the following actions are performed:

Situation**:** Boss.income: 6000$; Employee.income: 3000$
TA1 Upd: Employee.income 5000$
TA2 Upd:. Boss.income 4000$ (still less than employee.income)

After storing both transactions, the Boss.income becomes less than the Employee.income, which violates the logical database consistency.

When the constraint had been checked within both transactions, it was valid. Also, the instances included in both transactions are different and do not conflict. The problem is, that the validation has been made on an instance, which had already been updated in another transaction.

To avoid such situations, all the instances included in the verification rules must be locked in the transaction, which can be done simply ba maring them as modified (e.g. Employee.Modify()).

User-defined transactions

Workspace

A workspace is a long-time transaction that may exist even after process termination. Workspaces are used for storing temporary updates. As long as working in a workspace changes are not written to the database but to a special workspace area.

A workspace is like a transparent slide on top of the database. After finishing a series of updates that may take several days or weeks you may consolidate the changes, i.e. storing all updates to the database or discard them.

Workspaces are identified by names and allocated by the database system. Usually workspaces reside in the same location as the root of the database.

In contrast to process transactions several users may work in the same workspace but only one can consolidate the changes. After consolidating or discarding changes the workspace is cleared up, i.e. it will be empty.

Workspace transactions

Workspace

A workspace is a long-time transaction that may exist even after process termination. Workspaces are used for storing temporary updates. As long as working in a workspace changes are not written to the database but to a special workspace area.

A workspace is like a transparent slide on top of the database. After finishing a series of updates that may take several days or weeks you may consolidate the changes, i.e. storing all updates to the database or discard them.

Workspaces are identified by names and allocated by the database system. Usually workspaces reside in the same location as the root of the database.

In contrast to process transactions several users may work in the same workspace but only one can consolidate the changes. After consolidating or discarding changes the workspace is cleared up, i.e. it will be empty.

Shadow database

One problem with long transactions is keeping the database consistent without blocking the whole system by locked instances and collections.

Example: One transaction adds an instance with a unique name “Paul” in a workspace. This is not visible from other workspaces but, nevertheless, no other user is allowed to add “Paul” again.

Hence the system must know in a way the state of the database, as it would look like when all workspaces have been consolidated. This is also important when checking update, insert or deletion rules, which may involve other instances.

For this purpose ODABA provides a shadow database when working with workspaces. The shadow database reflects the database state at any time as if all changes have been made directly in the database.

Thus, it becomes possible to refer to the states in the shadow database as well as to the states in the “real” database without updates saved in workspaces.

Nested workspaces

Within a workspace you may create another workspace below the current one. When consolidating the lower workspace the changes are stored in the upper workspace but not in the database. When consolidating the upper workspace while lower workspaces are opened only the consolidated changes from lower workspaces are stored into the database. In this case the upper workspace is not completely cleared up since all instances that are in use in lower workspaces are still locked in the database.

Ownership

A user can own workspaces. In this case only the user who has allocated the workspace is able to open it again. It is, however, also possible to create public workspaces that can be accessed by any user.

Usually private workspaces are created by a user from within an application. The application itself takes care about user control. Public workspaces are created by administrators by establishing a workspace for a certain user group (as system or developer).

Whether a workspace is private or public depends whether it is created with user identification or not.

Enabling workspace

Before using workspaces this feature must be explicitly enabled. Enabling the workspace feature will create the shadow database and installs a workspace 0 which is the base for all other workspaces. Workspace 0 is a system workspace and cannot be discarded or consolidated or deleted. It is allocates at the same location as the database with the extension .ws0.

```
DatabaseHandle dbhandle(dictptr,"C:/ODABA/test.db",NO");
                                                    //exclusive
database

dbhandle.EnableWorkspace("C:/ODABA/test.shadow");
```

Open workspace

You can create or open an existing workspace with a database handle.

```
dbhandle.OpenWorkspace("Wspace1");                  //publ.
workspace

dbhandle.OpenWorkspace("Wspace1","user27");//priv.
workspace
```

After opening the workspace all updates are stored in the opened workspace. When the workspace is used the first time it is created automatically. When it does already exist the existing workspace is opened.

Private workspaces can be opened only when passing the same user name that has been passed when opening the workspace the first time. Opening a public workspace any username or none can be passed.

You can check whether a workspace exists using the LocateWorkspace() function, which returns *true* when the workspace has already been created.

```
dbhandle.LocateWorkspace("Wspace1");
```

After opening you are in the context of the workspace. When opening another workspace now this is created on top of the current workspace. I.e. in the context of the current workspace:

```
dbhandle.OpenWorkspace("Wspace1");
dbhandle.OpenWorkspace("Subspace11");
```

This will create *Subspace11* in *Wspace1*. The same you achieve with calling

```
dbhandle.OpenWorkspace("Wspace1.Subspace11");
```

Finish workspace

You can finish a workspace by consolidating or discarding changes.

Consolidate

ConsolidateWorkspace() will consolidate all changes made in the workspace. You can consolidate the currently opened workspace, only, i.e. you must open the workspace before consolidating.

For consolidating a workspace is must be opened with exclusive use. Only when no other user has access to the workspace it is possible to consolidate it.

```
dbhandle.OpenWorkspace("Wspace1",NULL,YES);
dbhandle.ConsolidateWorkspace();
```

Discard

If you want to through away all changes made in the workspace you can use DiscardWorkspace() for the currently opened workspace.

```
dbhandle.DiscardWorkspace();
```

Finish nested workspace

There is a difference finishing a top workspace that has no other workspaces above or when finishing a workspace with higher workspaces.

You may consolidate or discard a top workspace but you cannot discard a workspace that has one or more lower workspaces on top. When consolidating a non-base workspace, i.e. a workspace that is on top of another one, changes are stored in the lower workspace. Thus, an upper workspace contains all consolidations from its lower workspaces or changes made directly in the workspace.

When consolidating a workspace that has lower workspaces Only changes made directly in the workspace or consolidated changes from lower workspaces are stored.

Close workspace

Workspaces are closed when closing the database. It is possible, however, to close the active workspace explicitly.

```
dbhandle.CloseWorkspace();
```

When working in a hierarchy of workspaces you can close all workspaces in the hierarchy. In this case further updates are stored directly in the database.

```
dbhandle.CloseWorkspace(YES);
```

Listing workspaces

You can list available workspaces for a special user, all workspaces or public workspaces. Workspaces are retrieved hierarchically based on a work space or the database itself.

Get Workspace

GetWorkspace() returns the name of a workspace by index relative to a given root.

```
// first top workspace
dbhandle.GetWorkspace("",0);

// first workspace below Wspace1 for user 'user27'
dbhandle.GetWorkspace("Wspace1",0,"user27");
```

When passing a user name to the function it returns only workspaces for the selected user. Usually the function returns the workspace name, which is the name of the workspace path without the root path, in the result buffer, i.e. the value is removed when another database handle function is called. You may, however, pass an area for storing the workspace name.

```
char         name[40];
dbhandle.GetWorkspace("Wspace1",0,"user27",name);
```

The workspace list is buffered internally. It is created when calling GetWorkspace() or LocateWorkspace() the first time. To ensure that the list is up-to-date you may pass the refresh-option (YES or true) for updating the list.

Delete workspace

After creating a workspace it will remain until it is explicitly deleted. Even Discard or Consolidate will not remove the workspace but leave an empty one. For removing a workspace you can use DeleteWorkspace().

A workspace can be deleted only when there are no lower workspaces defined and when it is completely empty.

```
dbhandle.DeleteWorkspace("Wspace1","user27");
```

When the workspace had been allocated with user name the user name has to be passed also for deleting the workspace.

Performing consistency checks

Working with workspaces creates some problems for performing logical consistency checks (handling Store, Insert or Delete-Events). The application has to decide whether checks are performed against the original database (which does not reflect changes made in the workspace) or against the shadow database, which reflects all updates stored in workspaces. Assuming that most of the workspaces are consolidated later consistency checks against the shadow database are better than those against the original database. In any case there is a risk that discarding a workspace may create inconsistency.

For performing checks against the shadow database you can activate the shadow database.

Activate Shadow Base

Activating the shadow database will direct all read operations to the shadow database instead of the original database.

```
dbhandle.ActivateShadowBase();
```

Save consistency checks

To perform save consistency checks you must lock all instances involved in the checking. This can be done also when the shadow database is active as shown in the following example for the DBStored handler in a Person context class

```
Logical         sPerson::DBStore()
{
  logical          term = NO;
  PropertyHandle  *pers_pi = GetPropertyHandle();
  PropertyHandle  *boss_pi = pers_pi->GetPropertyHandle();
  Pers_pi->GetDataBaseHandle()->ActivateShadowBase();

  boss_pi = pers_pi->GetPropertyHandle("boss");
  if ( boss_pi->Get(FIRST_INSTANCE) )
     If ( boss_pi->GetPropertyHandle("income") <
                  boss_pi->GetPropertyHandle("income") }
       term = YES;             // update not possible
     else
       if ( boss_pi.Modify() ) // cannot lock instance
          term = YES; ;        // update not possible
       else
          poss_pi.Save();            //  Lock  instance  in
workspace

  Pers_pi->GetDataBaseHandle()->DeactivateShadowBase();
  Return(term);
}
```

Modifying and saving the instance will lock the instance for all other workspaces except subordinated ones. Thus no other user can modify the instance that has been involved in the consistence check until the workspace is consolidated. When Modify() fails the instance is locked by another user and the transaction is not save since the other user may discard the workspace or roll back a transaction.

Deactivate Shadow Base

At the end of the process you must deactivate the shadow database to ensure that processing continues with the original database.

```
dbhandle.DeactivateShadowBase();
```

Database Context Programming

Handling Data Events

Event types

Several types of events can be handled within a context class. Data events are defined as enumerated set of system events, which can be handled by overloaded functions.

Database events

Database events are generated when opening or closing the database.

DB-Object events

Database object events are generated when opening or closing the database object or when creating or deleting a database object..

Instance events

Instance events are generated when reading, storing or updating an instance. In case of imbedded structures events are generated ones for each imbedded structure or base class but not for referenced structures in reference property handles.

Property events

Property events are generated when opening or closing a property handle, when reading or updating data in base type properties or when inserting or removing instances in reference property handles.

Structure Events

Structure events are generated when an instance changes the system state, i.e. is loaded from the database into memory or stored to the database.

DBO_Read

The Read Event is generated when an instance is loaded into memory. The event is generated after reading the instance, i.e. the event cannot prevent reading the instance. The read event allows calculating transient properties, setting update state for the instance, selecting generic attributes etc.

Reading the instance is completed after running the read event. You may access properties within the instance but the date state is not yet set to *DAT_Filled*.

In some cases the read event is executed more than once. You can check whether the event has already been handled checking the DateState. When the read event has already been handled the date state is set to *DAT_Filled*.

Property events

DBO_Read

The Read Event is generated when an instance is selected in a property handle. The event is generated after selecting the instance, i.e. the event cannot prevent selecting the instance. The read event allows calculating transient properties, setting update state for the instance, selecting generic attributes etc.

After selecting the instance including handling the read event the date state is set to *DAT_Filled*.

In some cases the read event is executed more than once. When the read event has already been handled ones the date state is set to *DAT_Filled*.

DBO_Refresh

The refresh event indicates that the environment of a property handle has been changed. This usually happens for the sub property handles when another instance is selected in a parent property handle. In contrast to the read event the refresh event is generated only when the property handle is used.

The refresh event is used to update transient references or collections when an instance has changed. To avoid unnecessary updates the refresh is generated only when data is requested from the property handle the first time after the parent handle has changed and not when reading the parent instance.

It is also possible to generate the event from the application using the property handle function Refresh().

Handling System Events

Events

Events are state transitions in user instances or instances of system classes (as property handles), which are usually called system events. While user events (state transitions on database instances or collections) can be handled by causality definitions (event-reaction), special support is provided for handling system events.

There are two different groups of system events:

- Database events
- Server events

Database events

Database events are generated when instances or collections are created, changed or deleted. Database events are usually signalled to application event handlers or context handler functions. Database events are signalled within the application causing the event. Database events are not sent to other applications.

Database events are generated within a transaction, i.e. the handler functions are called and executed within the transaction as well. Database events are passed first to the context handler functions (see Database Context Programming).

Server events

Server events are generated when working in a client-server environment but also when running a local application. Server events signal a state transition on an object instance or collection.

Server events are generated at the end of a top transaction, i.e. in the moment when the updated instance or collection (index) is written to the database. All users (PropertyHandle) that are registered for the updated instance or index will be informed.

Multiple use of event handlers

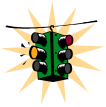
Some property handle functions create a property handle sharing the cursor with another property handle. Since event handlers are installed on cursor level, events are actually created for the property handle cursor.

Multiple Event Handler

To allow several property handles referring to the same cursor installing different event handlers, you may install several event handlers for a property handle cursor or for property handles referring to the same cursor. You may also install a number of event handlers for one property handle.

Handling Database Events

There are two ways of handling database events. One way is to handle database events in specific context classes that can be defined for a property or structure. The other way is to define an event handler class and setting event handlers for the property handle.



Database events are signalled within an application, only. When another user or even your application changes an instance or collection in one property handle, only resources associated with this property handle are informed about the modification. Other property handles might be notified by means of server events.

Another way to detect modifications when accessing a resource or an instance is comparing the modification count. When the modification count for an instance or collection has changes since the last access, the resource has been updated and should be refreshed. Collections are refreshed by the database automatically. Instances have to be refreshed explicitly by the application.

Context Class Event Handler

For handling events in property or structure specific context classes you must define a CTX_Structure context class (for handling instance events) or a CTX_Property context class for handling collection or attribute events. In this case the only thing to do is to overload the appropriate virtual context functions

```
DBRead ();
DBOpened ();
……
```

You will find a list of all database event handlers at http://www.run-software.com/ODABADocu below **ContextClasses/CTX_DBBase**. For each database event a virtual handler function is provided, which can be overloaded in the context class (Structure or Property context) and which is called, when the event is signalled.

Event Handler

For handling database events without implementing context classes you may use event handlers defined in an event handler class instead. This allows running type independent event handlers in ‘abstract’ applications. For implementing an event handler class you must do the following:

Defining an event handler class

Event handler classes must be derived from a dummy class **vcls**:

```
class MyEventHandler : vcls
{
 public:
  uchar   InstHandler (DB_Event event_id);
  uchar   PropHandler (DB_Event event_id);
};
```

The class should at least contain one handler function that can be called as notification event handler.

Setting event handler

Event handlers must be set for each property handle that should handle database events. You may set one or more handler for a property handle. In this case, all property handlers are processed in the sequence they have been added to the property handle.

The event handler is passed as an event link that consists of an event handler function and a class instance. The handler is called later with the instance of the event handler class set in the event link.

For setting an event handler you must define a link instance:

```
MyEventHandler  inst_hdl;
EventLink      *inst_link;

inst_link                          =                          new
ELINK(inst_hdl,MyEventHandler,InstHandler);
prop_hdl. SetInstanceProcessHandler(inst_link);

MyEventHandler  prop_hdl;
EventLink      *prop_link;

prop_link                          =                          new
ELINK(prop_hdl,MyEventHandler,PropHandler);
prop_hdl.SetPropertyProcessHandler(inst_link);
```

You can set an instance and a property handler. Instance handler are called for instance events, property handler are called for collection events. The handler class instance might contain additional information, which can be accessed by the handler function since the handler is called with the class instance passed to the ELINK macro.

Calling event handler

When setting an event handler in addition to a context class handler function the context class handler function is called after running the event handler set by the application. The context class handler function is not executed when the event handler returns false (NO).

Usually event handlers have to determine the event type, which is passed to the function:

```
uchar MyEventHandler:: InstHandler(DB_Event event_id)
{
  switch ( event_id )
  {
     case DBO_Insert   : // handle before insert event
                         break;
     case DBP_Inserted : // handle after insert event
                         break;
     ……                  // handle other events

     default;
  }
   return(false);
}
```

Returning an error (**true**) will deny the requested operation for all 'before' events (DBO_ events). It has no effect on calling other event handlers set for the property handle. If at least one of the registered handlers returns true, the requested operation is not performed.

Event Handler Class

Instead of manually creating and setting event handler you can use the EventHandler class. This is a base class for handling database and server event, which provides the basic functionality for activating handler functions.

Define derived handler class

You have to define a derived handler class that contains your handler functions for handling instance and/or property (collection) events.

The EventHandler class provides standard functions for registering database events and the only thing that you need to do is overloading the two event handler functions for instance and collection events:

```
class MyEventHandler : EventHandler
{
  long             inst_evt_count;
  long             prop_evt_count;
public:
      virtual   uchar   ProcessInstanceHandler   (DB_Event
event_id);
                { inst_evt_count++;
                  return(true); };
      virtual   uchar   ProcessPropertyHandler   (DB_Event
event_id);
                { prop_evt_count++;
                  return(true); };
  MyEventHandler(PropertyHandle &ph)
            :     EventHandler(ph),     inst_evt_count(0),
prop_evt_count(0)
                { ActivateProcessEventHandler(); };
  ~MyEventHandler(){};
};
```

Constructor

The property handle is passed as parameter to the constructor. The base class constructor will create the event links and set the links in the property handle.

Handler functions

The base class has implemented two virtual handler functions that must be overloaded in your handler class. You need not to call the base class handler functions in your handler function.

When not overloading the instance or property handler no handler function is executed and the context class notification handler is called.

Signal Server Events

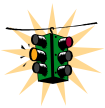
Register indexes

Indexes for collections are registered as soon as they are used for locating instances. When updating instances usually all indexes defined for a collection are registered. Since index is a more general term in OM it includes also singular references.

For receiving database events you need to register the property handle. Than, updates on resources activated by your application will be notified. The application will receive notifications on instance and collection updates.

Register Property-Handles

To avoid unnecessary communication you need to register property handles that shall receive event notifications from the server. Property handles that are not explicitly enabled for receiving event notifications will not be informed about server events.



The application will receive notifications only for those indexes and instances that are registered. These are usually the instances selected in a property handle, and not the instances that are, e.g. displayed in a list. Thus, your application will not receive notifications for instances displayed in the list, which have been changed meanwhile, but only for the instance that is selected in the property handle.

Events are not sent to the property handle, that has caused the event, i.e. the property handle storing an updated instance will not receive an event from the server. The application will, however, receive also events that are caused by other property handles in the same application.

Instance events

A property handle is registered as user for an instance when the instance is selected in the property handle or when the instance is member of a block that has been read by a property handle. Events are sent only for instances that selected in a property handle or cashed in a buffer.

Collection events

Collection events are sent when an index for a collection changes. When working in read only mode, you will receive notifications for the selected index, only (SetOrder). When working in write or update mode, you will receive notifications for all indexes for the collection.

Signaled events

State transitions that are signalled as events or notifications are:

- updated (SE_Updated)
- modified (SE_Modified)
- created (SE_Created)

SE_Updated

This event indicates a modification on an instance or index. A modification on an index includes insert, remove or rename operations. Since several actions might be taken before an updated index is written to the database there is no information about the type of modification.

SE_Deleted

This event indicates that an instance or index has been deleted, i.e. completely removed from the database. The event is not generated when removing an instance from a collection without deleting it.

When?

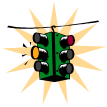
Server events are generated, when an instance or collection is stored to the database (or server cache). This happens usually when a top transaction is terminated.

When not working with transactions the modification is signalled when the PropertyHandle function returns control (closing the internal transaction).

In some cased events can are generated outside any transaction (e.g. when using AddGlobal).

Who?

Server events will be notified to all PropertyHandles that are registered for the updated or deleted instance.



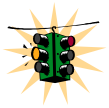
When no transaction has started the event will be generated at the end of the function causing the event. In this case the PropertyHandle that has caused the modification will not be notified. When running updates within a transaction the event will be generated at the end of the transaction. In this case it is not possible anymore to determine, which property handles did cause the event and all property handles registered for the resource will be informed.

Handling Server Events

There are two ways of handling server events. One way is to handle server events in specific context classes that can be defined for a property or type. The other way is to define an event handler class and setting event handlers for the property handle.

Local and Server applications

In local applications only “server events” that are generated by the application itself can be handled. In this case the local application is considered as client and server at the same time. In this case event handler are working synchronously. The transaction is closed after processing all notifications.



When running the database in file server mode (Windows only), that allows several applications running at the same time on the database, notifications are **not** sent between applications. For sending notifications across a number of applications you must run the server version of OM.

In a server environment the server sends notifications to all active clients that have PropertyHandles registered for the event. In contrast to local applications the event handler are working asynchronously. Event handlers are called at the end of a transaction commit. It is, however, possible that the commit terminates before the last notification is processed.

Context Class Event Handler

For handling events in property or type specific context classed you must define a CTX_Structure context class (for handling instance events) or a CTX_Property context class for handling collection (index) events. In this case the only thing to do is to overload the virtual context functions

```
NotifyDeleted (long objid);
NotifyUpdated (long objid);
```

Objid is the database instance or index identity that has been updated.

The instance handler is called before closing the transaction. Hence the instance, the notification event refers to, might not be selected anymore in the PropertyHandle.

Event Handler

You can use event handler defined in an event handler Class instead of context classes. This allows running type independent event handlers in 'abstract' applications. For using an event handler you must do the following:

Defining an event handler class

Event handler classes must be derived from a dummy class **vcls**:

```
class EvtHandler : vcls
{
 public:
  uchar   InstHandler (CSA_Events event_id,
                       long objid,
                       PropeertyHandle &ph);
  uchar   PropHandler (CSA_Events event_id,
                       long objid,
                       PropeertyHandle &ph);
};
```

The class should at least contain one handler function that can be called as notification event handler.

Setting event handler

Event handlers must be set for each property handle that should handle notification events.

The event handler is passed as an event link that consists of an event handler function and a class instance. The handler is called later with the instance of the event handler class set in the event link.

For setting an event handler you must define a link instance:

```
EvtHandler   inst_hdl;
EventLink  * inst_link;
inst_link = new ELINK(inst_hdl,EvtHandler,InstHandler);
prop_hdl.SetInstanceEventHandler(inst_link);

EvtHandler   prop_hdl;
EventLink  * prop_link;
prop_link=new ELINK(prop_hdl,EvtHandler,InstHandler);
prop_hdl.SetPropertyEventHandler(inst_link);
```

You can set an instance and a property handler. Instance handler are called for instance events, property handler are called for index (collection) events. The handler class instance might contain additional information, which can be accessed by the handler function since the handler is called with the class instance passed to the ELINK macro.

Calling event handler

When setting an event handler in addition to a context class handler function the context class handler function is called after running the event handler set by the application. The context class handler function is not executed when the event handler returns false (NO).

Usually event handlers have to determine the event type, which is passed to the function:

```
uchar EvtHandler:: InstHandler(CSA_Events event_id,
                          long objid,
                          PropeertyHandle &ph)
{
  switch ( event_id )
  {
     case SE_Updated : // handle update event
                       break;
     case SE_deleted : // handle delete event
                       break;
     default;
  }
   return(false);
}
```

The *objid* passed to the handler is the instance identity of the instance or collection (index) that has been changed. The PropertyHandle is the PropertyHandle that received the notification.

When several copies exist for a PropertyHandle referring to the same collection cursor the handler is called only once. When, however, having several property handles (which might be copies of each other) referring to different cursors the handler is called for each property handle that is registered and positioned to the instance or collection, which has generated the event.

Returning **false** will avoid calling the context class handler function, which might be implemented for a corresponding context class.

Event Handler Class

Instead of manually creating and setting event handler you can use the EventHandler class. This is a base class for server event handlers that provides the basic functionality for activating handler functions.

Define derived handler class

You have to define a derived handler class that contains your handler functions for handling instance and/or property (collection) events.

```
class MyEventHandler : EventHandler
{
  long             inst_mod_count;
  long             prop_mod_count;
public:
  virtual uchar InstanceEventHandler (CSA_Events event_id,
                                      long objid );
                { inst_mod_count++;
                  return(true); };
  virtual uchar PropertyEventHandler (CSA_Events event_id,
                                      long objid )
                { prop_mod_count++;
                  return(true); };
  MyEventHandler(PropertyHandle &ph)
             :     EventHandler(ph),     inst_mod_count(0),
prop_mod_count(0)
                { ActivateServerEventHandler(); };
  ~MyEventHandler(){};
};
```

Constructor

The property handle is passed as parameter to the constructor. The base class constructor will create the event links and set the links in the property handle.

Handler functions

The base class implemented two virtual handler functions that must be overloaded in your handler class. You need not to call the base class handler functions in your handler function.

When not overloading the instance or property handler no handler function is executed and the context class notification handler is called.

To avoid calling the context class notification handler you should return false (NO) from the handler function. Returning true (YES) the context class notification function is called as well.