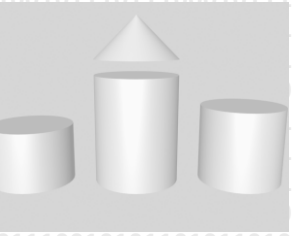

# ODABA$^{NG}$

# Data Exchange


**run Software-Werkstatt GmbH**
**Weigandufer 45**
**12059 Berlin**

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

Berlin, October 2012

# Content

# 1 Introduction

**ODABA$^{NG}$**

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

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

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

This makes ODABA$^{NG}$ applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

**Platforms**

ODABA$^{NG}$ supports windows platforms (Windows95/98/Me, Windows NT and Windows 2000) as well as UNIX platforms (Linux, Solaris).

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

**Interfaces**

ODABA$^{NG}$ supports several technical interfaces:

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

**User Interfaces**

ODABA$^{NG}$ provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA$^{NG}$ provides a special ODABA$^{NG}$ GUI builder.

# 2 Data Exchange

Data Exchange provides different ways of importing or exporting data from an ODABA$^{NG}$ database to extended comma separated files (ESDF, CSV), to xml files (OXML, XML) or to object interchange format files (OIF). OIF is the proposed standard format for exchanging data between object-oriented databases (ODMG).

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

Besides different file formats ODABA$^{NG}$ provides different data exchange technologies.

**Command line Tools**

ODABA$^{NG}$ provides two data exchange tools, one for import (**Import**) and another for exporting data (**Export**).

Import

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

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

Export

Import provides features for exporting selected data from an ODABA$^{NG}$ database to a file with one of the defined formats. Also, this is a preferred way for exporting data periodically, while ad-hoc data export becomes more comfortable from within OSI or by using the GUI Data Exchange tool.

**GUI Tool**

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

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

The GUI tool provides the most comfortable way for designing a data exchange schema. It allows also running

the defined data exchange (e.g. for testing purpose).

**OSI expressions**

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

ToFile

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

FromFile

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

**PropertyHandle**

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

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

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

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

Open

The file schema for external files can be defined in advance within the ODABA$^{NG}$ dictionary as structure and extent definition. In this case, the external file can simple be accessed via the extent name, similar to any other extent in the database.

OpenExtern

Often, it is not very comfortable defining structure and Property handles for external files in the dictionary. Es-

pecially CSV files carry metadata in the headline, which contains sufficient information for extracting a file schema. Thus, property handle support an additional function for opening external data sources, which are not defined in the dictionary. This allows accessing data ad-hoc and in much simpler in many cases.

# 3 Data exchange definition

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

**Access functions**

External files can be accessed in different ways.

- File – Read or write explicitly
- FromFile – Import from file
- ToFile – Export to file

File

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

FromFile

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

ToFile

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

**File access parameter**

All file functions refer to same set of parameters, which describe the location for data and file or exchange schema.

```
file             := 'File' foperand_list
from_file        := 'FromFile' foperand_list
to_file          := 'ToFile' foperand_list
foperand_list    := 'Path' '=' string [foptions(*)]
foptions         := ',' foption
foption          := file_type | file_schema | headline
file_type        := 'FileType' '=' type_name
type_name        := 'ODL' | 'OXML' | 'OIF' | 'CSV' | 'ESDF' |
                    'BINA'
file_schema      := 'Definition' '=' def_location,
def_location     := structure_name | string
headline         := 'Headline' '=' boolean
```

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

file_type

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

- BINA - binary flat file (.bina)
- CSV, ESDF - extended self delimiter file (.esdf, .csv)
- OXML - ODABA xml file (.oxml, .xml)
- OIF - object interchange format (.oif)

file_schema

The file or exchange schema can be provided together with the data. When the file or exchange schema is passed in a separate file, the Definition option refers to the location of the file definition. When no separate file schema is passed the file schema is supposed to be part of the external file (e.g. headline in an ESDF file).

When the file or exchange schema is passed with the data file (no file schema), the definition format has to correspond to the format of the data file.

- BINA – no file definition supported in the file
- CSV, ESDF – ESDF headline format
- OXML - ODABA xsd definition
- OIF – ODL definition

def_location

The file or exchange schema can be provided as definition in a dictionary, in which case the *structure_name* refers to a structure definition in the dictionary. The file or exchange schema might also be provided in a separate file as ODL (.odl), OXML (.oxsd) or ESDF (.esdf) definition files, in which case the location is passed as quoted string pointing to the file location.

The system determines the proper type from the file extension. When no valid extension could be found, the system tries to analyze the definition file type by file content:

first character '<' : OXML format

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

Beginning with **schema** keyword, ODL is assumed.

Headline

The headline option indicates, whether the external data

file contains an imbedded file schema (typically the headline in CSV or ESDF files). Either headline or schema location must be provided in order to obtain the file schema for input operation.

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

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

# 4 Data Exchange schema

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

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

- Dictionary – Structure definition in an ODABA dictionary
- CSV/ESDF – Headline definition format
- OSI ODL – Schema definition language
- OXML – extended XML schema definition

The data exchange schema is an extended file schema with additional mapping rules for assigning external data fields to database properties. The database property correspondence is always defined in the source attribute, which is an extension for all file schemata.

## File Schema

**File schema**

The file schema contains the structure or data type definition(s) required for describing the data in the external file. Most of the rules for defining schemata of types mentioned above are described in other documents. Thus, only specific rules to be taken into consideration when providing a file schema will be described here.

Since all supported file formats are hierarchical formats, i.e. properties or fields may contain sub-properties or collection of related instances. There are limitations in a

few cases (e.g. for binary files), but this is no contradiction for providing common principles when defining a file schema.

Common file schema

In contrast to database schema (object model), the file schema does not support orders and relationships. The following BNF definition provides an idea of the common definition elements provided in all definition formats.

```
record          := field_list
field_list      := field(*)
field           := [name] [data_type] [size]
                   [sub_fields] [dimension] [db_source]
sub_fields      := field_list
```

record

Even though it be confusing to speak about a record in a hierarchical data structure, we will the term record as entry for the definition, since in many cases, one knows exactly what a record is. Sending Person data might be as complex as possible, but we will probably consider data for each person as a record in this data set.

name

A record (or structure) consists of a number of fields (properties, attributes). Each field may have got a name (if not, artificial names are created as *field0001*, *field0002* etc.).

data_type

The default data type is STRING (except for binary files, which may contain binary data as well).

sub_fields

When a field (or field Instances) are structured, a list of sub-fields (describing a structure again) can be defined. Each field in the sub-field list may have sub-fields etc.

dimension

The default dimension is 1. Any other positive number for fixed arrays or 0 for collections with undefined number of elements may replace the default dimension.

db_source

The database source defines the corresponding data source in a database. This might be a property name or path, but not an expression. A data source is required for importing or exporting data from/to external files but not for reading external files by property handle, only.

Each schema supporting these requirements is able to describe a file schema. It becomes obvious, that OXML schema and ODL provide these requirements, as well as the ODABA dictionary does. For CSV or ESDF a specific file definition format has been defined, which, in the simple case, corresponds to the CSV head line.

**Dictionary**

Describing an external file structure in the dictionary

might be the most comfortable way for complex data structures. Dictionary structures for external files may consist of attributes, references and exclusive base structures. External file definitions must not contain relationships.

For assigning a data source to a field in an external file is possible by means of the property *source* reference. In case of defining more than one source references for a property the assignment is done by field name, which must be assigned as source definition name in this case.

**ODL**

The ODL schema definition is a script equivalent to the dictionary definition. It follows the same rules as defining a structure in the dictionary. The example below shows a complete definition for a Person data exchange file.

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

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

Depending on import or export functions the database source acts as target or source.

**OXML schema**

An OXML schema is another equivalent for a dictionary structure definition and can be used instead of an ODL or dictionary definition.

**CSV/ESDF**

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

In order to support more complex data structures in a comfortable and CSV compatible format, we introduced ESDF, which is a CSV extension, since it supports complex attributes as well as references.

The rules for defining a CSV or ESDF file are described in the subsequent BNF definition.

```
Headline       := fields
fields         := field [ field_ext(*) ]
field_ext      := sep field
field          := [name] [size] [sub_fields] [dimension] [source]
source         := '=' path
size           := '(' number ')'
dimension      := '[' number ']'
sub_fields     := '{' fields '}'
path           := path_element [ path_extension(*) ]
path_extension:= '.' path_element
path_element   := name [ parameter ]
parameter      := get_parm | provide_parm
get_parm       := '(' value ')'
provide_parm   := '[' value ']'
value          := path | constant

Data           := items
items          := [ item ] [ item_ext(*) ]
item_ext       := sep [ item ]
item           := dvalue | item_set | item_block
item_set       := '[' items ']'
item_block     := '{' items '}'

sep            := ';' | '|' | '\t'
```

The BNF describes the ESDF header and the data lines. In contrast to CSV, ESDF limits field delimiter to ';', tab and '|'. Undefined symbols *name*, *string, dvalue* and *constant* are standard symbols and do have the following meaning.

*name* Is a field name which usually starts with an alphabetic character or underscore.

*number* Is an integer value.

*dvalue* Any sequence of characters not containing field, string, instance, collection or line separators (see "Delimiters" below)..

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

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

When names in the headline are identical with database

source names, source assignments can be omitted:

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

When defining the file definition separately instead of providing a headline, the definition may contain line breaks:

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

**Delimiters**

ESDF defines a reserved set of delimiter characters. Delimiter characters must not appear in values without being quoted.

Field delimiter

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

String delimiters

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

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

Instance delimiters

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

Collection delimiter

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

# 5 External file formats

ODABA supports different external file formats, which can be accessed directly via PropertyHandle access functions or via file functions File(), ToFile() and FromFile.

**BINA**

Binary files are files with a fixed data structure and can be considered as the most compressed format for data exchange.

There are, however, several limitations in using binary files.

- Binary files always require an external file definition (no headline definition supported).
- Binary files do support arrays or references with fixed number of elements, only.

In contrast to other files formats, binary files may contain integer and float values or other binary data types.

**ESDF, CSV**

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

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

Specification

ESDF has a simple BNF specification as described below:

```
ESDFFile        := [ header ] esdf_record(*)
Header          := Headline nl
esdf_record     := Data nl
```

As line break, new line (NL), carriage return (CR) or both are accepted after headline and between data lines. Headlines are optional. File definitions might be also passed separately.

Headline

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

contains a headline.

**OIF**

The object interchange format is a standard suggested by ODMG. The ODABA OIF has some extensions and some limitations compared with the ODMF OIF. Nevertheless, there is a big common denominator between both, which makes it possible exchanging data in OIF format with any other object-oriented database supporting OIF.

In contrast to other external file formats, OIF supports additional features as the distinction between creating and overwriting data in the database during import.

Specification

An OIF file is an alternate recursion between property and instance values. Each property value may consist of a number of instance values and each instance value consists of one or more property values. Thus, an OIF file may contain a number of instances, but also a collection (property), which contains a number of instances.

```
OIF                 := OIFData | OIFInit
OIFData             := prop_init(*)
OIFInit             := prop_list | inst_list

prop_init           := identifier ['='] prop_value [',']
prop_value          := inst_init | inst_list
inst_list           := '{' inst_init(*) '}'

inst_init             := [ inst_intro ] [ locator ] inst_value
[',']
inst_intro          := [ identifier ] scoped_name
locator             := update_locator | create_locator
update_locator      := '(' loc_init ')'
create_locator      := '[' loc_init ']'
loc_init            := constant | prop_init(*)
inst_value          := constant | prop_list
prop_list           := '{' prop_init(*) '}'
```

Delimiters

Value delimiter ‘,’ and assignment operator are optional and should not be used when compatibility is required.

Locators

In order to distinguish between replacing or creating, ODABA OIF supports create and update locators. Create locators follow the standard and will create new instances when not yet existing.

In contrast to ODMG OIF, which supports numerical locators, only, ODABA OIF supports key locators, as well. When passing a number in *loc_init*, this is interpreted as position in a collection or in an array. When passing a string, it is interpreted as key value. Component

key values can be defined either by passing a string value with component values separated by '|' ( [ 'Miller|Paul' ] ) or by passing the values by property names ( [ first_name 'Paul', name 'Miller ] ).

Property lists — ODMG OIF supports property values by position, i.e. without preceding property name. Since this implies a high risk for value mismatch, ODABA OIF does not support this feature and requires a property name in front of each value assignment.

Head line — When defining a "headline" at the beginning of an OIF file, this must be defined as ODL schema definition beginning with the **schema** keyword.

```
SCHEMA {
          …. // file schema definitions
}
```

**OXML** — OXML is an xml format with several ODABA specific schema extensions. Thus, xml is able to reflect the complexity of ODABA database object model definition completely.

Providing an OXML schema separately, xml files can be accessed via an OXML dictionary as OXML database.

Using file access for accessing xml data, however, allows importing or exporting xml data directly from/to an external file according to the database source definitions. Moreover, an xml file can be describes using an ODL definition, which might be more comfortable.

## Accessing external files

There are different ways for accessing external files. External files can be accessed from within a program using the PropertyHandle function OpenExtern(). Another way is accessing external files via OSI scripts using File(), FromFile() or ToFile() functions.

It is also possible defining extents in the database referring to external files. In this case, the external file can be accessed simply by opening the extent with appropriate PropertyHandle functions.

**PropertyHandle** — PropertyHandle for external files can be created two

ways. One is defining an external extent in the dictionary in advance. The other is to open a property handle calling OpenExtern().

External extent

When defining an extent for an external file, this can be accessed by property handle functions after creating an appropriate property handle.

Defining an extent for accessing allows defining one or more sort orders (indexes) for the extent, which are created when opening the extent.

```
PropertyHandle   ph(obhandle,"ExtFile",PI_Read);
```

After opening the property handle simple property handle access functions (Get, Position, NextKey etc.) can be used for selecting instances in the external extent.

OpenExtern

OpenExtern() allows opening a property handle for external file access without defining it in the dictionary in advance.

```
PropertyHandle   ph;
char            *path = "externalFile.esdf";
char            *filetype = NULL;   // ESDF from extension
char            *definition = NULL; // definition in headline

ph.OpenExtern(obhandle,path,definition,filetype,PI_Read);
```

After opening the property handle simple property handle access functions (Get, Position) can be used for selecting instances in the external extent.

OpenExtern() provides access to the external file but does not automatically import or export the file.