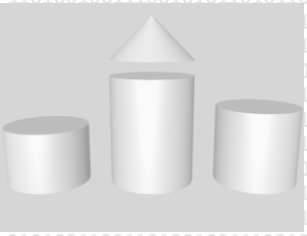


ODABA NG

Data Storage Formats

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

Table of Contents

1Introduction 4

2ODABA data storage formats 6

2.1Storing ODABA data in relational databases 7
2.1.1Implementing an access package 22

2.2XML database 26
2.2.1XML schema attribute extensions 27

2.3File access via property handle 29
2.3.1Flat or binary files 31
2.3.2Comma separated format 32
2.3.3ESDF format 33
2.3.4Object Interchange Format (OIF) 36
2.3.5ODABA XML format 37

1 Introduction

ODABA

ODABA is an terminology-oriented database system that allows storing objects and methods as well as causalities . As terminology-oriented database, ODABA supports complex object types (user-defined data types) defined in a terminology model, which reflect application relevant concepts.

ODABA applications are characterized by high flexibility. In addition to object type or context hierarchies, ODABA supports multifarious relations between object instances (master and detail relations, relations between independent object instances and others). This way, behavior of objects in the real world can be represented considerably better than in relational database systems.

ODABA supports event-driven applications concerning the graphical user interface as well as the database level. Thus, application design is tightly related to the experts or customers problem, since it refers to the same names and concepts as being defined by subject matter experts. This enables ODABA to solve highly complex jobs in administrative and knowledge areas.

Platforms

ODABA supports windows platforms (from Windows 95 up to Windows 7) as well as UNIX platforms (Linux, SUN Solaris). ODABA supports 64 and 32 bit technologies.

ODABA also runs well in heterogeneous client/server environments or with Internet servers.

Interfaces

ODABA supports several technical interfaces:

- C++, .Net as application program interface (this allows e.g. using ODABA in C# or VB scripts and applications)
- ODABA Script Interface (OSI) for accessing data via a script language, which is similar to C# or JAVA.
- Multiple storage support for using relational databases for storing ODABA data
- XML for supporting data exchange with complex data structures
- OIF (object interchange format), flat files and ESDF (extended self delimiter fields) for accessing data provided in external file formats
- Document exchange support for importing or exporting data from/to open office or Microsoft office documents.

Tools

ODABA provides a number of database maintenance tools, but also development tools in order to provide terminology model definitions, data model specifications, application design and others.

To support just-in-time documentation, all ODABA tools provide extended documentation facilities, which are the base for generating system and WEB documentation, but also online help systems.

2 ODABA data storage formats

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

- Relational databases (ORACLE, MySQL, MS SQL Server)
- OXML (XML based on ODABA schema extensions)

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

2.1 Storing ODABA data in relational databases

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

The following SQL databases have been chosen for ODABA support:

- ORACLE
- Microsoft SQL Server
- My SQL

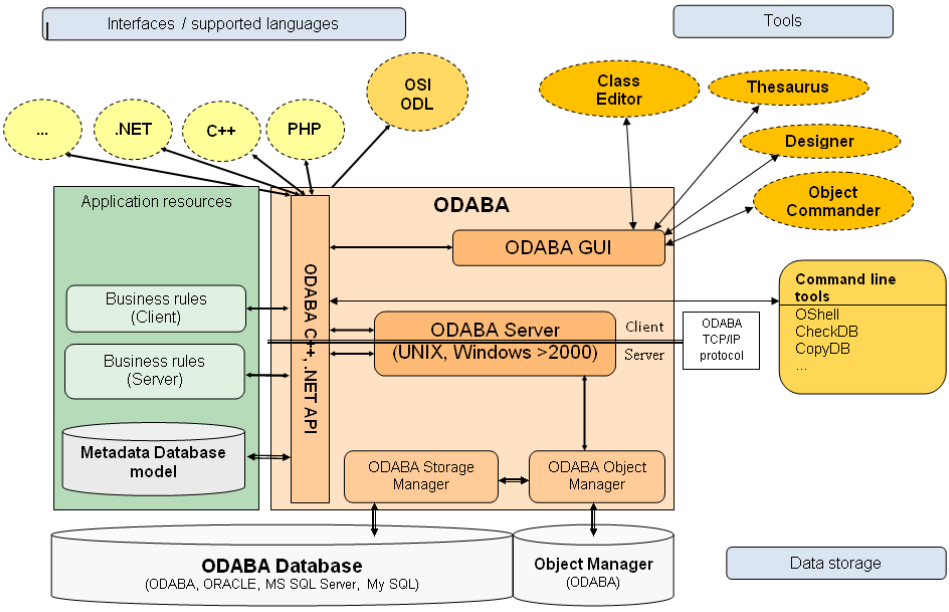
This list might be expanded when ever required.

In order to create SQL table definitions for a project or module, one may call an OSI expression as described below or use the ODE ClassEditor (see Class Editor/Generate external resources/SQL Definitions).

RDB access architecture

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

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



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

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

OR mapping rules

Since the information content of a relational database is a subset of the information, that can be stored in an object oriented database, mapping rules can be defined for the "relational data" in the object-oriented database. The requirement for mapping rules results from the fact, that relational databases do not support complex attributes, which will be resolved to property path (`address.city`). Moreover, relationships are transferred to mapping tables (m:n relationships).

Specific attributes result from the fact, that collections in ODABA (e.g. the children of a person) are considered as objects. In order to benefit from this feature also

when running the ODABA application in a relational database, collection attributes are created, which refer to the local unique identifier for the collections (LOID).

In order to define relational tables, ODABA creates tables and attributes according to the rules described below. First, ODABA type and property names are converted into ODABA table and attribute names. The ODABA table or attribute name is always a name constructed from ODABA type and property names. ODABA table and attribute names created might be truncated later on, when exceeding the name size allowed by the target system.

ODABA table names are the corresponding table names for ODABA data types. ODABA table names for M:N relationships are composed from several property and type names. Table names might be converted in to target system table names (e.g. Oracle table names), when exceeding the maximum length.

ODABA attribute names are property names. Property names in complex attributes are preceded by the property name for the attribute. Thus, ODABA attribute names may contain a number of ODABA property names separated by dots.

Depending on the features of the target database system, comments are stored to the table and column definitions and/or written to the table definition file.

The examples are generated from the schema definition below describing an update register for documents and presentations.

```
//****************************** Schema
*************************************************
// type     example
// date     10-03-20 17:47:25.82
// dbsource  - ODABA Version 10.0
//*****************************************************************
**************
UPDATE SCHEMA example
{

//****************************** Enumeration
*************************************************
// type     UpdateTypes
// date     10-03-20 17:47:25.95
// dbsource  - ODABA Version 10.0
//*****************************************************************
**************
UPDATE ENUM UpdateTypes
{
   change                                          = 3,
   create                                          = 2,
   delete                                          = 1,
   other                                           = 0,
};

//****************************** Class
*************************************************
// type     Document
// date     10-03-20 17:47:26.10
// dbsource  - ODABA Version 10.0
```

```
//************************************************************************
**************
NEW CLASS Document PERSISTENT VERSION=0 TYPE ID=1756  GUID
       :   PUBLIC UpdateObject GUID OWNER updateObject
               VERSION=0
               ORDERED_BY ( ik UNIQUE )
(
  KEY { IDENT_KEY ik( id_number ); };
  ALIGNMENT = 0;
)
{
};

//******************************** Class
*****************************************************
// type     Notice
// date     10-03-20 17:47:27.06
// dbsource  - ODABA Version 10.0
//************************************************************************
**************
NEW CLASS Notice PERSISTENT VERSION=0 TYPE ID=1753  GUID
(
  KEY { IDENT_KEY ik( id_number ); };
  ALIGNMENT = 0;
)
{
   ATTRIBUTE {
           PROTECTED INT(10,0) id_number
               VERSION=0;
   };
   REFERENCE {
           PROTECTED MEMO(4000) OWNER text [1]
               VERSION=0;
   };
};

//******************************** Class
*****************************************************
// type     Presentation
// date     10-03-20 17:47:27.18
// dbsource  - ODABA Version 10.0
//************************************************************************
**************
NEW CLASS Presentation PERSISTENT VERSION=0 TYPE_ID=1757  GUID
       :   PUBLIC UpdateObject GUID OWNER updateObject
               VERSION=0
               ORDERED BY ( ik UNIQUE )
(
  KEY { IDENT KEY ik( id number ); };
  ALIGNMENT = 0;
)
{
};

//******************************** Class
*****************************************************
// type     Update
// date     10-03-20 17:47:27.21
// dbsource  - ODABA Version 10.0
```

```
//************************************************************************
**************
NEW CLASS Update PERSISTENT VERSION=0 TYPE ID=1754  GUID
       :   PUBLIC Notice GUID UPDATE Notice
               VERSION=0
               BASED_ON   Notice
(
  KEY { IDENT_KEY ik( id_number ); };
  ALIGNMENT = 0;
)
{
   ATTRIBUTE {
           PROTECTED STRING(100) title
               VERSION=0;
           PROTECTED UpdateTypes update types
               VERSION=0;
   };
   REFERENCE {
           PROTECTED Notice UPDATE OWNER notices
               VERSION=0;
   };
   RELATIONSHIP {
           PROTECTED Update UPDATE related_updates
               VERSION=0
               INVERSE    referenced in
               BASED ON   Update
               ORDERED BY ( ik UNIQUE );
           PROTECTED Update UPDATE SECONDARY referenced_in
               VERSION=0
               INVERSE    related updates
               BASED ON   Update
               ORDERED BY ( ik UNIQUE );
           PROTECTED UpdateObject UPDATE WEAK_TYPED SECONDARY object [1]
               VERSION=0
               INVERSE    updates
               BASED ON   *
               ORDERED BY ( ik UNIQUE );
   };
};

//******************************** Class
*************************************************
// type     UpdateObject
// date     10-03-20 17:47:27.35
// dbsource  - ODABA Version 10.0
//************************************************************************
**************
NEW CLASS UpdateObject PERSISTENT VERSION=0 TYPE ID=1755  GUID
(
  KEY { IDENT_KEY ik( id_number ); };
  ALIGNMENT = 0;
)
{
   ATTRIBUTE {
           PROTECTED INT(10,0) id number
               VERSION=0;
   };
   RELATIONSHIP {
           PROTECTED Update UPDATE updates
```

```
                VERSION=0
                INVERSE    object
                BASED ON   Update;
   };
};

UPDATE EXTENT Document UPDATE MULTIPLE_KEY OWNER Document
                VERSION=0
                ORDERED BY ( ik UNIQUE LARGE );

UPDATE EXTENT Notice UPDATE MULTIPLE_KEY OWNER Notice
                VERSION=0
                ORDERED_BY ( ik UNIQUE LARGE );

UPDATE EXTENT Presentation UPDATE MULTIPLE KEY OWNER Presentation
                VERSION=0
                ORDERED_BY ( ik UNIQUE LARGE );

UPDATE EXTENT Update UPDATE MULTIPLE KEY OWNER Update
                VERSION=0
                ORDERED BY ( ik UNIQUE LARGE );
};
```

Memo and blob properties

Two tables are created in order to store the MEMO and BLOB type properties. Each row in the table contains a local unique identifier (LOID) for the MEMO or BLOB property and the CLOB or BLOB field in order to store the large character or binary object.

References to MEMO or BLOB properties are stored as links to the SYS__MEMO or SYS__BLOB table. Within the current table, a link attribute to the MEMO or BLOB table is defined with the odaba property name.

```
-- oracle
  CREATE TABLE "SYS__MEMO"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL PRIMARY KEY USING INDEX
TABLESPACE "example INDEX",
    "SYS__ENTRY"    CLOB
  )  LOB ("SYS__ENTRY") STORE AS ( STORAGE ( INITIAL 4M) NOCACHE
NOLOGGING );

  CREATE TABLE "SYS  BLOB"
  (
    "SYS__LOID"     NUMERIC(20,0) NOT NULL PRIMARY KEY USING INDEX
TABLESPACE "example_INDEX",
    "SYS  ENTRY"    BLOB
  )  LOB ("SYS  ENTRY") STORE AS ( STORAGE ( INITIAL 4M) NOCACHE
NOLOGGING );

-- ...

  ALTER TABLE "Notice" ADD ( "text" NUMERIC (20,0) REFERENCES "SYS  MEMO"
);
```

Enumerations

For each enumeration a table with the enumeration name will be created. Enumerator values (code and title) are stored to the table. Hierarchical enumerations are stored as flat ones. Details as constraint, enumerator type or detailed description are not stored to the relational database, since those information is still available via the ODABA dictionary.

Enumerator names and values are used as being defined in ODABA.

```
-- oracle
  CREATE TABLE "UpdateTypes"
  (
    "code"     NUMERIC(5,0) NOT NULL ,
    "name"     VARCHAR(40) ,
    PRIMARY KEY ("code") USING INDEX TABLESPACE "example INDEX"
  );

INSERT INTO "UpdateTypes" VALUES ('1', 'delete');
INSERT INTO "UpdateTypes" VALUES ('2', 'create');
INSERT INTO "UpdateTypes" VALUES ('3', 'change');
INSERT INTO "UpdateTypes" VALUES ('0', 'other');
```

Complex data typed

Instance data is stored in tables having the same name as the complex data type defined in the ODABA object model. All tables get an additional property SYS__LOID, which holds the local object identity for each instance. All relational tables are indexed by SYS__LOID. For each complex ODABA data type a table will be created.

```
-- oracle
  CREATE TABLE "Update"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS__LOID") USING INDEX TABLESPACE "example_INDEX"
  );
```

Extents

Being a member of an extent is an additional information, which cannot be directly stored in the relational database. Thus, each extent creates an additional reference attribute in the data type referenced in the extent (owner reference, since extents are usually the owner of the instances).

The name for the owner reference is the is the extent name succeeded by two underscores (Notice__). Thus, it becomes possible to distinguish between notices stored in the Notice extent and those stored locally for an Update instance.

```
-- oracle
  ALTER TABLE "Notice" ADD ( "Notice__" NUMERIC(20,0) );
```

Inheritance

When a type inherits exclusive from its base type, properties defined in the base type(s) are considered as properties of the type. Thus, Attributes of exclusive inherited base types might become attributes in any number of tables.

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

```
-- oracle, shared base type
  ALTER TABLE "Update" ADD ( "Notice" NUMERIC(20,0) REFERENCES "Notice"
);
```

Attributes

Attributes are defined as table columns using the attribute name. Complex attributes are provided as resolved attribute paths including dots, which are part of the attribute path (`address.city`). Such names, usually require name quotes, which depend on target system. When referring to attribute names in exclusive base types, names are not prefixed.

When an attribute defines a fixed array, a column will be created for each array element. Each array element except the first will be extended by the element position in the array (e.g. name_1).

ODABA data types are converted to appropriate types in the target system. Generated column types may differ for different target systems. Enumeration values are converted into link columns, e.g. columns referring to the enumeration table.

```
// oracle
  ALTER TABLE "Update" ADD ( "title" VARCHAR(100) );
  ALTER TABLE "Update" ADD ( "update type" NUMERIC(5,0) REFERENCES
"UpdateTypes" );
```

Collections and references

Collections and singular references allowing asynchronous updates create a collection identity (LOID) in order to identify the collection object. The collection objects play an important role, when updating collections. Essential parts of the application logic interface (ALI) are based on collection events. For using this features, collection attributes are stored in table instances as well, even though the are not of interest for relational queries.

Collection attributes will be created for all references/relationships, which are multiple or weak-typed or can be updated asynchronously.

Collection attributes are stored with their property (reference or relationship) name preceded by two underscores (__related_updates).

```
-- oracle
  ALTER TABLE "Update" ADD ( "__related_updates" NUMERIC(20,0) );
  CREATE TABLE "Update__related_updates"
  (
    "SYS  LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
    "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
    PRIMARY KEY ("SYS__LOID","SYS__REF") USING INDEX TABLESPACE
"example_INDEX"
  );
```

References

References (and owning relationships) define a 1:N relationship and create an owner attribute in the table defined by the referenced type. The column name is constructed from the property (reference or relationship) name and the current type name (e.g. notices__Update).

```
-- oracle
  ALTER TABLE "Notice" ADD ( "notices  Update" NUMERIC(20,0) REFERENCES
"UpdateRegister" );
```

Notes: Usually, owning relationships are defined as primary relationships. When this is not the case, the inverse relationship will create a M:N table, in addition. This is not a problem, but leads to (unnecessary) redundancy.

Relationships

For not owning relationships, a mapping table for related instances will be created, when the relationship is primary. The table name will be constructed from the object name and the relationship name (e.g. Update__related_updates). The two columns contain the local unique identifiers for the rows in the target tables as being referenced in the column statement.

When exceeding the naming limits for the target system, a unique table name will be created.

```
-- oracle
  CREATE TABLE "Update  related updates"
  (
    "SYS__LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
    "SYS__REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
    PRIMARY KEY ("SYS__LOID","SYS__REF") USING INDEX TABLESPACE
"example INDEX"
  );
```

Weak-typed collections

Weak typed collections (references or relationships) create mapping tables for each data type which inherits directly or indirectly exclusive from the referenced type. Mapping tables are created for references or primary relationships, only. The table names for weak-typed collection are constructed from the current table name, the property name and the target table name separated by double underscore (Update__object__Document).

```
-- oracle, simulated by setting object in Update to primary
  CREATE TABLE "Update  object"
  (
    "SYS__LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
    "SYS__REF"  NUMERIC(20,0) NOT NULL REFERENCES "UpdateObject",
    PRIMARY KEY ("SYS  LOID","SYS  REF") USING INDEX TABLESPACE
"example INDEX"
  );
  CREATE TABLE "Update__object__Document"
  (
    "SYS__LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
    "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Document",
    PRIMARY KEY ("SYS  LOID","SYS  REF") USING INDEX TABLESPACE
"example_INDEX"
  );
  CREATE TABLE "Update__object__Presentation"
  (
    "SYS  LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
    "SYS__REF"  NUMERIC(20,0) NOT NULL REFERENCES "Presentation",
    PRIMARY KEY ("SYS__LOID","SYS__REF") USING INDEX TABLESPACE
"example INDEX"
  );
```

Generic attributes

Generic attributes are considered as multiple references and will create a collection attribute in the current table and an owner attribute in the referenced type for the generic attribute (see References).

ODL example

SQL example (oracle)

Below, you will find the complete set of table definitions created from the example (update register) described in the ODL example.

```
--
-- SQL Schema : ODABA Dictionary for example
-- Target DB  : Oracle
-- Version    : 1.0
-- Date       : 10-03-20 Time: 19:00:55.76
--

  CREATE TABLE "SYS__MEMO"
  (
```

```
    "SYS__LOID"     NUMERIC(20,0) NOT NULL PRIMARY KEY USING INDEX
TABLESPACE "example_INDEX",
    "SYS  ENTRY"    CLOB
  )  LOB ("SYS  ENTRY") STORE AS ( STORAGE ( INITIAL 4M) NOCACHE
NOLOGGING );

  CREATE TABLE "SYS__BLOB"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL PRIMARY KEY USING INDEX
TABLESPACE "example INDEX",
    "SYS__ENTRY"    BLOB
  )  LOB ("SYS__ENTRY") STORE AS ( STORAGE ( INITIAL 4M) NOCACHE
NOLOGGING );

  CREATE TABLE "UpdateTypes"
  (
    "code"     NUMERIC(5,0) NOT NULL ,
    "name"     VARCHAR(40) ,
    PRIMARY KEY ("code") USING INDEX TABLESPACE "example INDEX"
  );

INSERT INTO "UpdateTypes" VALUES ('3', 'change');
INSERT INTO "UpdateTypes" VALUES ('2', 'create');
INSERT INTO "UpdateTypes" VALUES ('1', 'delete');
INSERT INTO "UpdateTypes" VALUES ('0', 'other');

  CREATE TABLE "Document"
  (
    "SYS__LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS  LOID") USING INDEX TABLESPACE "example INDEX"
  );

  CREATE TABLE "Notice"
  (
    "SYS__LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS  LOID") USING INDEX TABLESPACE "example INDEX"
  );

  CREATE TABLE "Presentation"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS  LOID") USING INDEX TABLESPACE "example INDEX"
  );

  CREATE TABLE "Update"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS  LOID") USING INDEX TABLESPACE "example INDEX"
  );

  CREATE TABLE "UpdateObject"
  (
    "SYS  LOID"     NUMERIC(20,0) NOT NULL,
    PRIMARY KEY ("SYS  LOID") USING INDEX TABLESPACE "example INDEX"
  );

-- Extent references
    ALTER TABLE "Document" ADD ( "Document__" NUMERIC(20,0) );
    COMMENT ON COLUMN "Document"."Document__" IS 'Owner: Document';
```

```
    ALTER TABLE "Notice" ADD ( "Notice__" NUMERIC(20,0) );
    COMMENT ON COLUMN "Notice"."Notice__" IS 'Owner: Notice';
    ALTER TABLE "Presentation" ADD ( "Presentation  " NUMERIC(20,0) );
    COMMENT ON COLUMN "Presentation"."Presentation  " IS 'Owner:
Presentation';
    ALTER TABLE "Update" ADD ( "Update__" NUMERIC(20,0) );
    COMMENT ON COLUMN "Update"."Update__" IS 'Owner: Update';

-- Structure Definition Document
    ALTER TABLE "Document" ADD ( "id number" NUMERIC(10,0) );
    COMMENT ON COLUMN "Document"."id_number" IS 'Attribute:
Document.updateObject.id_number';
    ALTER TABLE "Document" ADD ( "__updates" NUMERIC(20,0) );
    COMMENT ON COLUMN "Document"."  updates" IS 'Collection:
Document.updateObject.updates';
        CREATE TABLE "Document__updates"
        (
          "SYS  LOID" NUMERIC(20,0) NOT NULL REFERENCES "Document",
          "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
          PRIMARY KEY ("SYS  LOID","SYS  REF") USING INDEX TABLESPACE
"example INDEX"
        );

-- Structure Definition Notice
    ALTER TABLE "Notice" ADD ( "id number" NUMERIC(10,0) );
    COMMENT ON COLUMN "Notice"."id number" IS 'Attribute:
Notice.id number';
    ALTER TABLE "Notice" ADD ( "text" NUMERIC (20,0) REFERENCES
"SYS__MEMO" );
    COMMENT ON COLUMN "Notice"."text" IS 'Reference: Notice.text';

-- Structure Definition Presentation
    ALTER TABLE "Presentation" ADD ( "id_number" NUMERIC(10,0) );
    COMMENT ON COLUMN "Presentation"."id_number" IS 'Attribute:
Presentation.updateObject.id_number';
    ALTER TABLE "Presentation" ADD ( "  updates" NUMERIC(20,0) );
    COMMENT ON COLUMN "Presentation"."  updates" IS 'Collection:
Presentation.updateObject.updates';
        CREATE TABLE "Presentation__updates"
        (
          "SYS  LOID" NUMERIC(20,0) NOT NULL REFERENCES "Presentation",
          "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
          PRIMARY KEY ("SYS  LOID","SYS  REF") USING INDEX TABLESPACE
"example_INDEX"
        );

-- Structure Definition Update
    ALTER TABLE "Update" ADD ( "Notice" NUMERIC(20,0) REFERENCES "Notice"
);
    COMMENT ON COLUMN "Update"."Notice" IS 'Reference: Update.Notice';
    ALTER TABLE "Update" ADD ( "title" VARCHAR(101) );
    COMMENT ON COLUMN "Update"."title" IS 'Attribute: Update.title';
    ALTER TABLE "Update" ADD ( "update types" NUMERIC(5,0) REFERENCES
"UpdateTypes" );
    COMMENT ON COLUMN "Update"."update types" IS 'Attribute:
Update.update_types';
    ALTER TABLE "Update" ADD ( "__notices" NUMERIC(20,0) );
    COMMENT ON COLUMN "Update"."__notices" IS 'Collection:
Update.notices';
```

```
    ALTER TABLE "Notice" ADD ( "notices__Update" NUMERIC(20,0) REFERENCES
"Update" );
    COMMENT ON COLUMN "Notice"."notices  Update" IS 'Owner: notices';
    ALTER TABLE "Update" ADD ( "  related updates" NUMERIC(20,0) );
    COMMENT ON COLUMN "Update"."__related_updates" IS 'Collection:
Update.related_updates';
        CREATE TABLE "Update__related_updates"
        (
          "SYS  LOID" NUMERIC(20,0) NOT NULL REFERENCES "Update",
          "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
          PRIMARY KEY ("SYS__LOID","SYS__REF") USING INDEX TABLESPACE
"example_INDEX"
        );
    ALTER TABLE "Update" ADD ( "  referenced in" NUMERIC(20,0) );
    COMMENT ON COLUMN "Update"."  referenced in" IS 'Collection:
Update.referenced_in';
    ALTER TABLE "Update" ADD ( "__object" NUMERIC(20,0) );
    COMMENT ON COLUMN "Update"."  object" IS 'Collection: Update.object';

-- Structure Definition UpdateObject
    ALTER TABLE "UpdateObject" ADD ( "id number" NUMERIC(10,0) );
    COMMENT ON COLUMN "UpdateObject"."id_number" IS 'Attribute:
UpdateObject.id_number';
    ALTER TABLE "UpdateObject" ADD ( "  updates" NUMERIC(20,0) );
    COMMENT ON COLUMN "UpdateObject"."  updates" IS 'Collection:
UpdateObject.updates';
        CREATE TABLE "UpdateObject  updates"
        (
          "SYS__LOID" NUMERIC(20,0) NOT NULL REFERENCES "UpdateObject",
          "SYS  REF"  NUMERIC(20,0) NOT NULL REFERENCES "Update",
          PRIMARY KEY ("SYS  LOID","SYS  REF") USING INDEX TABLESPACE
"example INDEX"
        );
```

Naming

As long as possible, the target system uses the ODABA names.

Names supported in relational databases differ. Thus, Oracle supports not more than 30 characters for table and column names, while MS SQL Server allows table and column names up to 128 characters. Since table names may consists of three ODABA names plus underline characters and column names do not have any limit, names have to be truncated in order to provide unique names.

Moreover, name mapping is required later on for data access and also for documenting the mapping between ODABA and the target SQL database. Thus, when creating table definitions for a selected target system, a name translation table will be created, which provides the mapping between table and column names constructed by ODABA and table and column names in the target system. Translation tables are stored in the dictionary in following collections:

- SDB_SQLTarget('Oracle').names (Oracle mapping)
- SDB_SQLTarget('MSSQL').names (MS SQL Server mapping)
- SDB_SQLTarget('MySQL').names (MySQL mapping)

Name mappings can be viewed also in the ClassEditor Objects/SQL Targets.

Limitations

Running ODABA with a relational database underneath causes some restrictions. The first and most important one is, that the relational data storage might be accessed by SQL tools in order to perform queries, but not in order to update the database. All update operations must pass through the object relation mapper (ORM). Otherwise, the ORM database might become inconsistent. In detail, following restrictions have to be taken into account:

- Since relational databases usually do not support namespaces for tables, data model definitions running with relational data storage must not define persistent namespaces. Instead, type names should be prefixed or marked in any other way. In the model definition, one may define object types in modules or namespaces, but those must not be marked as active namespaces, i.e. type names must be unique within the dictionary.
- In order to guarantee proper maintenance of inverse relationships, ODABA supports update-able relationships. In a relational database, update-able relationships behave similar as many to many relations. This means that queries against the relational database must include an additional join operation when referring to singular update-able links.
- ODABA supports VOID type collections, i.e. collections, which may contain instances of any type. Theoretically, void collections could be supported, but this would require creating link tables between the type defining the VOID collection and all other defined types. This seems not to make much sense and has not been implemented. We suggest using weak-typed collections, instead.
- Property names in exclusive base types must be unique in order to avoid naming conflicts.
- Names of complex attributes are resolved. In case of deep nesting, this might exceed name length limits in the target system. The mapping tools care about creating proper names, but those might be difficult to read. Hence, attribute nesting and name length should be selected in a way that meets the target system requirements.
- Instance versioning is not yet supported for relational storage.
- Extension properties are not yet available.
- Many databases do not support different text encoding methods for text fields, i.e. one should use STRING (uses default encoding) always in order to avoid conflicts.

Other limitations are of minor importance. There are several features that require specific ODABA storage. Thus, when using work spaces, all workspace data is

stored in ODABA databases and is accessible via SQL only, when the workspace data has been consolidated to the root base.

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

SYS__LOID values (identities) are the base for all links and instance identification and must not change after being created.

2.1.1 Implementing an access package

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

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

Access packages to relational databases support two ways of accessing columns. One is by column or attribute name. Since several systems support internal column numbers when accessing data from within a program, columns might be accessed via column number, also. In order to access columns by number, the GetColumnNumber() function has to be overloaded in order to provide the proper column number for each table column/attribute.

When using column numbers, GetMemo() and InsertMemo() have to be overloaded as well, since those functions are referring to column names rather than to column numbers.

The access logic is mainly managed by the SQL_RootBase and SQLTable base class. In order to provide package (RDB) specific functionality, those classes have to be overloaded in corresponding OR-Mappers.

Implementing an access package

Implementing an OR-Mapper requires overloading the following functions in SQL_RootBase:

- Close
- Debug
- GetRootBase
- GetMemo
- InsertMemo
- LinkInstance
- LinkOwner
- Open
- RBType
- [StartCommit]
- [StopCommit]
- UnlinkInstance
- UnlinkOwner
- UpdateMemo
- destructor

Moreover, following functions have to be overloaded in the SQLTable class:

- Debug
- DeleteRow
- FinishInsertRow
- FinishSelectRow
- FinishUpdateRow
- GetColumn
- InsertRow
- SelectRow
- SetColumn
- UpdateColumn
- UpdateRow
- destructor

Instance operations are introduced by a row function (SelectRow(), InsertRow(), UpdateRow(), DeleteRow()), which usually locate the requested row for the operation. After locating a row in a table, several column function calls are made (GetColumn(), SetColumn(), UpdateColumn()) in order to read, create or update column values. Column values are provided as character data. Finally, the Finish...() functions are called in order to indicate the end of row processing. The functions usually have to be be overloaded in order to perform final row processing.

Apart from updating object attribute values, link information will be updated before and after updating attribute values in instances. In order to update link information, LinkInstance() or LinkOwner() and UnlinkInstance() or UnlinkOwner() have to be implemented in order to maintain M:N or 1:M relationships. Both functions are called only ones for a table row in order to create or delete a parent (reference) or relationship link.

In case of parent (owner) links, the link value has to be updated in the attribute passed to the function. In case of a relationship (instance) link, a mapping row has to be inserted into the M:N relationship table.

Data conversion

Data but also table column names require conversion. In order to convert column and table names properly, the base class SQL_RootBase provides a name conversion function Name(). From the name and the database specific maximum name length, the function constructs an appropriate database specific table or attribute name, which correspond to the name generated as table or attribute name when generating the table definition. All functions receiving table or attribute names, receive the original ODABA type or property names, which have to be converted to table or attribute names.

Attribute values are always passed in string formats (ASCII or Latin1) with a terminating 0. Following data formats are passed:

- string - ASCII string (latin1)
- integer - "[-]n*[.n*]" (decimal point according precision definition)
- float - "[-]n*[.n*][E[-]n*]"
- time - "hh:mm:ss,hs"
- date - "yy-mm-dd"
- datetime - "yy-mm-dd hh:mm:ss,hs"
- guid - "A-xxxxxxxx-xxxxxxxx-xxxx-xxxx-xxxxxxxx"

Values have to be passed in both directions referring to the same format, i.e. the access package will obtain values in the format above when updating columns and has to return values in an appropriate format when reading values.

Transaction management

Transaction management is mainly organized on ODABA level, i.e. a request of storing instances to the database is submitted by ODABA only, when committing a transaction. Thus, all update requests are send to the root base in the commit phase.

There are, however, RDB specific requirements passed to SQL_RootBase while a transaction is running. Thus, LinkInstance() and UnlinkInstance() requests are sent while running a transaction and will be cached by SQL_RootBase.

When committing a transaction, StartCommit() is called in order to indicate the beginning of the commit request. StopCommit() indicates, that committing data has been finished. StartCommit() is called in order to maintain table links. All links to be removes are reset here. When reimplementing or overloading the function, one has to take into account, that links have to be removed, before instances can be stored to the transaction. Between StartCommit() and StopCommit() all updated requests are submitted by ODABA calling UpdateInstance() or UpdateMemo(). After storing instances, StopCommit() is called, which will setup new links created during the transaction. This can be done, after instances have been created within the database.

As long as StartCommit() has not been called, the access package can assume, that access is read-only. This is true also after StopCommit().

By default, link requests are submitted in the EndCommit() function. The function reads all link and unlink requests from the cache (link_cache) and call LinkInstance() or UnlinkInstance() in order to handle the request. Those functions must be overloaded in the appropriate access package.

For write optimization, it might, however, be more efficient processing the link cache in the access package. in this case, outstanding link requests must be written to database before terminating the commit phase. Link requests can be obtained from the link cache (link_cache.RemoveHead()).

Create, delete and update instance

New entries are usually created via an update request. In order to distinguish new instances from old instances, the data position (acb::GetPosition()) can be checked. In case the position is 0, the instance is considered as new instance. In order to mark the instance as existing after creating is, the position should be set to a positive value (loid is suggested).

2.2 XML database

ODABA provides features for accessing XML files like an ordinary ODABA database. The idea is not maintaining persistent data in an XML file, but opening the possibility accessing XML data by the same means as accessing an ODABA database.

2.2.1 XML schema attribute extensions

ODABA schema definitions require some ODABA specific schema extensions. Schema extensions are available at www.odaba.com/OXMLExtensions.xsd. Using this schema extensions allow providing complete schema definitions via an XML schema.

A summary of ODABA XML schema extensions is given in the definition below.

Rules:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema targetNamespace="http://www.w3.org/XML/1998/namespace"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:attribute name="alignment" type="xs:integer"/>
<xs:attribute name="assignment" type="xs:string"/>
<xs:attribute name="baseCollection" type="xs:string"/>
<xs:attribute name="complete" type="xs:boolean" default="false"/>
<xs:attribute name="dataType" type="xs:string"/>
<xs:attribute name="deleteEmpty" type="xs:boolean" default="false"/>
<xs:attribute name="dependent" type="xs:boolean" default="false"/>
<xs:attribute name="descending" type="xs:boolean" default="false"/>
<xs:attribute name="dimension" type="xs:integer"/>
<xs:attribute name="distinct" type="xs:boolean" default="false"/>
<xs:attribute name="elementType">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="BaseType"/>
      <xs:enumeration value="Attribute"/>
      <xs:enumeration value="Reference"/>
      <xs:enumeration value="Relationship"/>
      <xs:enumeration value="Key"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="guid" type="xs:boolean" default="false"/>
<xs:attribute name="ignoreCase" type="xs:boolean" default="false"/>
<xs:attribute name="intersect" type="xs:boolean" default="false"/>
<xs:attribute name="inverse" type="xs:string"/>
<xs:attribute name="keyComponents" type="xs:string"/>  <!-- one or more key
components separated by comma, e.g. "name(descending),first_name" -->
<xs:attribute name="identKey" type="xs:boolean" default="false"/>
<xs:attribute name="multipleKey" type="xs:boolean" default="false"/>
<xs:attribute name="noCreate" type="xs:boolean" default="false"/>
<xs:attribute name="notEmpty" type="xs:boolean" default="false"/>
<xs:attribute name="orderKeys" type="xs:string"/>  <!-- one or more order
keys separated by comma, e.g. "key_name1(unique),keyname2" -->
<xs:attribute name="owner" type="xs:boolean" default="false"/>
<xs:attribute name="precision" type="xs:integer"/>
<xs:attribute name="privilege">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="private"/>
      <xs:enumeration value="public"/>
      <xs:enumeration value="protected"/>
```

```
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="referenceLevel">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="byValue"/>
      <xs:enumeration value="byReference"/>
      <xs:enumeration value="byPointer"/>
      <xs:enumeration value="byPointerPointer"/>
      <xs:enumeration value="generic"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="secondary" type="xs:boolean" default="false"/>
<xs:attribute name="size" type="xs:integer"/>
<xs:attribute name="static" type="xs:boolean" default="false"/>
<xs:attribute name="subSet" type="xs:string"/>
<xs:attribute name="superSet" type="xs:string"/>
<xs:attribute name="transient" type="xs:boolean" default="false"/>
<xs:attribute name="update" type="xs:boolean" default="false"/>
<xs:attribute name="version" type="xs:integer"/>
<xs:attribute name="virtual" type="xs:boolean" default="false"/>
<xs:attribute name="weakTyped" type="xs:boolean" default="false"/>
</xs:schema>
```

2.3 File access via property handle

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 property handle functionality for the external files. 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, which are used for explicit data exchange.

Property handle file extent

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

The path to the file to be accessed is set in an option with the extent name.

The file type to be accessed has to be defined in the extent definition /access type) as AT_BIN (flat files) or AT_EXTERN for extended self delimiter (ESDF), comma separated (CSV), object interchange format (OIF) and xml files (XML). One more type supported is directory access (AT_DIR) .

Accessing external files via extents is limited in the sense that specific settings as head line option indicating self describing files or special delimiters are not supported. More flexible file access is provided via the openExtern() function, which allows opening a collection based on an external file without referring to definitions in the directory.

Open extern

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

In order to access external files that do not have a file schema definition at all, ad-hoc schemata can be created for semi-structured files as XML or OIF. In this case,

the file is analyzed and a file schema is derived from property names passed with the data.

```
openExtern (const odaba::ObjectSpace &cObjectSpace, odaba::String
sFilePath, odaba::String sDefinitionFile, odaba::String sFileType,
odaba::AccessModes eAccessMode, bool bHeadline )
```

2.3.1 Flat or binary files

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

There are several limitations in using binary files.

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

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

In order to access flat files, a file description is required in the data base or has to be passed explicitly to the Property::openExtern() function. In order to provide an external schema definition, any of the supported definition formats might be used (see Data Exchange schema)

When the flat file contains line breaks, those have to be defined explicitly in the record structure definition for the file.

One might define weak typed binary files, in which case the data has to be provided in ASCII format and records have to be terminated by line breaks.

2.3.2 Comma separated format

CSV is a simple exchange format separating properties (fields) by field delimiters. As field delimiter, tab (\t), newline (\n) or semicolon (;) might be used (but not comma. In case of values containing one of those characters, values have to be enclosed in string delimiters String delimiters are also required, when the value contains string delimiters itself. In this case, string delimiters within the value have to be escaped. (e.g. "character \" is a string delimiter"). The escape character (\) will be removed before storing data.

CSV files contain any number of records (instances of a collection) terminated by new line character (\n). New line characters within values enclosed in string delimiters are not counted as instance end. CSV instances support attributes and attribute arrays, but no references or collections.

Since CSV does not require any tags, it is an efficient way of exchanging flat data files. On the other hand, it requires fields being defined in a correct sequence. CSV files must not contain data of more than one extent or weak-typed collections. Typically, CSV is used to pass complex data within an application. Thus, the Key contains CSV structured instances when not requesting another format.

Furthermore, CSV files can be accessed in exportData() and importData() functions (Property and ObjectSpace) or when opening external files by Property::openExtern().

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

Limited access to CSV files is supported by defining CSV extents in the dictionary (access type AT_EXTERN). I this case, the file path is expected in an option with the extent name. No head lines and no external file description are supported for CSV extents.

In order to provide more flexible access to CSV files, a file description has to be passed explicitly to the Property::openExtern() function. In order to provide an external schema definition, any of the supported definition formats might be used (see Data Exchange schema)

2.3.3 ESDF format

The Extended Self Delimiter File format is an extension of the CSV format. ESDF files contain any number of records (instances of a collection) enclosed in parenthesis { ... }. Between properties and lines non or any number of line breaks might be defined. In contrast to CSV, ESDF supports complex attributes and reference collections with variable number of instances. Since ESDF does not provide property names, all properties are interpreted by position.

Since ESDF does not require any tags, it is an efficient way of exchanging large data files. On the other hand, it requires fields being defined in a correct sequence. ESDF files must not contain data of more than one extent or weak-typed collections. Typically, ESDF is used to pass complex data within an application. Thus, the Instance contains ESDF structured instances when not being defined with another format.

Furthermore, ESDF files can be accessed in exportData() and importData() functions (Property and ObjectSpace) or when opening external files by Property::openExtern().

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

Limited access to ESDF files is supported by defining ESDF extents in the dictionary (access type AT_EXTERN). I this case, the file path is expected in an option with the extent name. No head lines and no external file description are supported for ESDF extents.

In order to provide more flexible access to ESDF files, a file description has to be passed explicitly to the Property::openExtern() function. In order to provide an external schema definition, any of the supported definition formats might be used (see Data Exchange schema)

Specification

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

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

Rules:

```
StringData            := Headline | Data | CSV              // defined for
providing the bnf class name

Headline              := fields
fields                := field [ field_ext(*) ]
field_ext             := sep field
field                 := [fname] [size] [sub_fields] [dimension] [source]
fname                 := name | string
source                := '=' [path ref]
path ref              :=  null | 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

CSV                   := csv_items
csv_items             := [ cvalue ] [ csv_item_ext(*) ]
csv_item_ext          := sep [ cvalue ]
cvalue                := string | csv string

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

sep                   := ';' | '|' | '\t' | ','
 null                 :: 'NULL'

std_symbols          ::= class(BNFStandardSymbols)
spec_symbols         ::= class(BNFSpecialSymbols)
svalue               ::= ref(spec_value)
csv string           ::= ref(spec csv value)
name                 ::= ref(std name)
number               ::= ref(std_number)
string               ::= ref(std_stringn)
constant             ::= ref(std_constant)
```

Data exchange schema

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

When passing the exchange schema in the data file headline, the schema must be defined completely in the first line. When passing en exchange schema in a separate file, the data exchange definition may contain any number of line breaks.

Another way is defining the data exchange specification in the dictionary (resource database).

Delimiters

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

Field delimiter

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

String delimiter

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

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

Instance delimiter

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

Collection delimiter

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

2.3.4 Object Interchange Format (OIF)

2.3.5 ODABA XML format