# Contents

1 **Introduction** .......................................................... 4
   1.1 Scope of This Document ........................................... 4
   1.2 Before Reading This Document .................................... 4
   1.3 SQL DDL and Matisse Modeler ..................................... 4

2 **Creating a Schema with ODL** ........................................ 5
   2.1 Overview of ODL Programming ...................................... 5
   2.2 Specifying Namespaces with ODL .................................. 7
   2.3 Specifying Classes with ODL ....................................... 8
   2.4 Specifying Attributes with ODL ................................... 9
   2.5 Specifying Relationships with ODL ............................... 13
   2.6 Specifying Indexes with ODL ...................................... 15
   2.7 Specifying Entry-Point Dictionaries with ODL .................. 16
   2.8 Specifying SQL Methods with ODL ............................... 17

3 **Using the mt_sdl utility** ............................................. 18
   3.1 Loading a schema .................................................. 18
   3.2 Exporting a schema ............................................... 18
   3.3 Generating class files ............................................. 18
   3.4 Updating an application .......................................... 19
   3.5 Preprocessing a schema file ..................................... 20

4 **Extending the Meta-Schema with ODL** ............................... 21
   4.1 Matisse Meta-Schema .............................................. 21
   4.2 Modifying the Meta-Schema ....................................... 21
   4.3 Extending Meta-Schema Classes ................................... 21
   4.4 Adding an Attribute to a Meta-Class ............................. 22
   4.5 Adding a Relationship to a Meta-Class ........................... 22
   4.6 Extending Matisse MtAttribute ................................... 23
   4.7 Extending Matisse MtRelationship ............................... 24
   4.8 Extending Matisse MtClass ....................................... 25

5 **Sharing Properties with ODL** ..................................... 27
   5.1 Shared Properties .................................................. 27
   5.2 Defining a Shared Attribute ..................................... 27
   5.3 Defining a Shared Relationship ................................. 28
   5.4 Defining a Shared Index ........................................... 28
   5.5 Defining a Shared Entry-Point Dictionary ........................ 29

Index ............................................................................. 30
1 Introduction

1.1 Scope of This Document

This document is a guide to using Matisse’s Object Definition Language (ODL) to create Matisse database schemas. It covers all aspects of ODL programming and use of the mt_sdl utility that are the same regardless of which language binding(s) you use.

General information about designing Matisse database schemas is covered in Getting Started with MATISSE. Binding-specific aspects of Matisse ODL are covered in the C++, Java, PHP, Python and Eiffel binding programmer’s guides.

1.2 Before Reading This Document

Throughout this document, we presume that you have read Getting Started with MATISSE, which provides an overview of Matisse schema elements and a discussion on schema design.

1.3 SQL DDL and Matisse Modeler

You may use SQL DDL (Data Definition Language) or Matisse modeling tool to create or update Matisse database schema. For more information, refer to Matisse SQL Programmer’s Guide or Matisse Modeler User’s Guide.

Note that you need to use an ODL file to generate persistent stub classes for the C++, Java, PHP, Python or Eiffel bindings. However, you can generate an ODL file from the Matisse database using the mt_sdl utility, or the Matisse Enterprise Manager, after you create the database schema using SQL DDL or the Matisse Modeler. For more information about mt_sdl, read 3 Using the mt_sdl utility.
2 Creating a Schema with ODL

2.1 Overview of ODL Programming

To create a schema in Matisse ODL, you create an .odl file, a text file containing a set of ODL declarations. You can then use the mt_sdl utility or the Matisse Enterprise Manager to load the schema in a newly initialized database and to generate source code for use with the C++, Java, PHP, Python or Eiffel binding.

The following is the ODL code for the schema diagrammed at right:

```odl
interface Person : persistent {
    attribute String firstName;
    attribute String lastName;
    attribute String<500> Nullable comment;
    attribute Integer Nullable age = NULL;
    attribute Image Nullable photo = NULL;
    relationship Person spouse[0, 1] inverse Person::spouse;
    relationship List<Person> children inverse Person::father;
    readonly relationship Person father[0, 1] inverse Person::children;
};

interface Employee : Person : persistent {
    attribute Date hireDate;
    attribute Numeric(10, 2) salary;
    relationship Manager reportsTo inverse Manager::team;
};

interface Manager : Employee : persistent {
    relationship List<Employee> team[1, -1] inverse Employee::reportsTo;
    relationship Employee assistant[0, 1];
};
```

```odl
interface Person : persistent {
    attribute String firstName;
    attribute String lastName;
    attribute String<500> Nullable comment;
    attribute Integer Nullable age = NULL;
    attribute Image Nullable photo = NULL;
    relationship Person spouse[0, 1] inverse Person::spouse;
    relationship List<Person> children inverse Person::father;
    readonly relationship Person father[0, 1] inverse Person::children;
};

interface Employee : Person : persistent {
    attribute Date hireDate;
    attribute Numeric(10, 2) salary;
    relationship Manager reportsTo inverse Manager::team;
};

interface Manager : Employee : persistent {
    relationship List<Employee> team[1, -1] inverse Employee::reportsTo;
    relationship Employee assistant[0, 1];
};
```
Default cardinality for a relationship declared using `relationship List<ClassName>` is 0, -1, while for a relationship declared using `relationship ClassName` default cardinality is 1, 1. Thus the cardinality does not need to be declared for the `children` or `reportsTo` relationships. See *Specifying Relationships with ODL* on page 13 for more information.
2.2 Specifying Namespaces with ODL

Define a namespace

module Namespace_name {...};

Associated classes, indexes and entry-point dictionaries are declared within the brackets.

For example, defining Person and Company classes in MyCompany.MyApp namespace and Employee and Manager in sub-namespace HR:

module MyCompany
{
    module MyApp
    {
        interface Person : persistent
        {
            attribute String<16> name;
        };
        interface Company : persistent
        {
            attribute String<32> name;
        };
        module HR
        {
            {
            };
            interface Manager: Employee: persistent
            {
            };
        };
    };
};
2.3 Specifying Classes with ODL

Define a class

```
interface Class_name : persistent {...};
```

Associated attributes, relationships, indexes and entry-point dictionaries are declared within the brackets, separated by semicolons.

Extend a class

```
interface Class_name : Superclass_name : persistent {...};
```

Extends an existing class (Superclass_name) to create a subclass. The subclass inherits all of the properties of the superclass.

For the multiple inheritance, enumerate the superclasses separated with a comma:

```
interface Class_name : Superclass1, Superclass2 : persistent {...};
```
2.4 Specifying Attributes with ODL

Basic declaration (name and type)  

attribute type Attribute_name

Specifies an attribute of type, where type is the ODL name for one of Matisse’s scalar (Boolean, Byte, Char, Date, Double, Float, Integer, Interval, Long, Numeric, Short, String, or Timestamp) or streamable (Audio, Bytes, Image, Text, Video) data types. For additional information, see the Matisse Data Type Reference.
### Table 1  Scalar types and media types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>True or false</td>
</tr>
<tr>
<td>Byte</td>
<td>A single byte. Valid range is from 0 to 255.</td>
</tr>
<tr>
<td>Char</td>
<td>A single byte character.</td>
</tr>
<tr>
<td>Date</td>
<td>Year, month, and day values of date</td>
</tr>
<tr>
<td>Double</td>
<td>A double-precision floating-point number, 64-bit approximation of a real number.</td>
</tr>
<tr>
<td>Float</td>
<td>A single-precision floating-point number, 32-bit approximation of a real number.</td>
</tr>
<tr>
<td>Integer</td>
<td>4-byte signed integer.</td>
</tr>
<tr>
<td>Interval</td>
<td>A period of time in days, hours, minutes, seconds, and micro-seconds.</td>
</tr>
<tr>
<td>Long</td>
<td>8-byte signed integer.</td>
</tr>
<tr>
<td>Numeric</td>
<td>A fixed-point decimal number having precision $p$ and scale $s$. Default precision and scale is 19 and 2. Precision sets the total number of digits and scale sets the number of those digits to the right of the decimal point.</td>
</tr>
<tr>
<td>Short</td>
<td>2-byte signed integer.</td>
</tr>
<tr>
<td>String</td>
<td>A variable length character string, optionally having the maximum length $n$. The maximum length is 2147483648 (2Gbytes). If $n$ is not specified, the maximum size is 2147483648.</td>
</tr>
<tr>
<td>String UTF16</td>
<td>A variable length Unicode character string, optionally having the maximum number of characters $n$. The maximum number of characters is 1073741824 (1G). If $n$ is not specified, the maximum is 1073741824.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Date and time consisting of year, month, day, hour, minute, second, and micro-second.</td>
</tr>
</tbody>
</table>
List types

<table>
<thead>
<tr>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;type&gt;</td>
</tr>
<tr>
<td>List&lt;type, max_elements&gt;</td>
</tr>
</tbody>
</table>

Defines the type as the LIST variation of the specified scalar type. It optionally specifies the maximum number of elements allowed in a list. For additional information, see the Matisse Data Type Reference.

The following is the list of available list types:

- List<Boolean>
- List<Date>
- List<Double>
- List<Float>
- List<Integer>
- List<Interval>
- List<Long>
- List<Numeric>
- List<Short>
- List<String>
- List<String UTF16>
- List<Timestamp>

Nullable flag

<table>
<thead>
<tr>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Nullable</td>
</tr>
</tbody>
</table>

Allows null values. If you specify Nullable and do not specify a default value, the attribute value will default to NULL. For example, in the case of a DateOfDeath attribute, a null would be appropriate for a living person.
**Default value**

\[
\text{attribute type } \text{Attribute_name} = \text{default_value}
\]

Explicitly defines a default value (which must be compatible with \textit{type}). Syntax for \textit{default_value} varies with the type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Default value syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>TRUE</td>
</tr>
<tr>
<td>Byte</td>
<td>Byte(value)</td>
</tr>
<tr>
<td>Char</td>
<td>'a'</td>
</tr>
<tr>
<td>Date</td>
<td>Date(&quot;YYYY-MM-DD&quot;)</td>
</tr>
<tr>
<td>Double</td>
<td>Double(value)</td>
</tr>
<tr>
<td>Float</td>
<td>Float(value)</td>
</tr>
<tr>
<td>Integer</td>
<td>value</td>
</tr>
</tbody>
</table>
| Interval           | Interval("[-]days HH:MM:SS.MMMMMM")  
|                    | Note that there is a space after the \textit{days} value. |
| Long               | Long(value)           |
| Numeric            | Numeric(0.00) (The only default value allowed is 0.) |
| Short              | Short(value)          |
| String             | "string"             |
| String UTF16       | UTF16("cycling")     
|                    | UTF16("\u00E9lo")    
|                    | UTF16("\u81EA\u884Cu8F66") |
| Timestamp          | Timestamp("YYYY-MM-DD HH:MM:SS.MMMMMM") |
| Audio              | Audio()              |
| Bytes              | Bytes()              |
| Image              | Image()              |
| Text               | Text("string")       |
| Text UTF16         | Text UTF16("cycling")  
|                    | Text UTF16("\u00E9lo")    
|                    | Text UTF16("\u81EA\u884Cu8F66") |
| Video              | Video()              |

\[
\text{attribute type nullable } \text{Attribute_name} = \text{NULL}
\]

Sets the default value to NULL.

\[
\text{attribute List<type> } \text{Attribute_name} = \text{List<type>\{ default_value_1, default_value_2, \ldots \}}
\]

Sets a list of default values.
attribute List<type> Attribute_name = List<type>{}

Sets the default value as an empty list.

To change a default value, revise the ODL file and use mt_sdl to update the database (see Loading a schema on page 18).

### Scalar default value examples

- attribute Boolean Attribute_name = TRUE
- attribute Byte Attribute_name = Byte(0)
- attribute Char Attribute_name = 'A'
- attribute Date Attribute_name = Date("1997-07-29")
- attribute Double Attribute_name = Double(0.000000)
- attribute Float Attribute_name = Float(0.000000)
- attribute Integer Attribute_name = 0
- attribute Interval Attribute_name = Interval("-29 12:00:00")
- attribute Long Attribute_name = Long(0)
- attribute Numeric Attribute_name = Numeric(0)
- attribute Short Attribute_name = Short(0)
- attribute String<32> Attribute_name = "ascii string"
- attribute String<32> UTF16 Attribute_name = UTF16("v\u00E9lo")
- attribute Text Attribute_name = Text("ascii string")
- attribute Text UTF16 Attribute_name = Text(UTF16("\u81EA\u884C\u8F66")
- attribute Timestamp Attribute_name = Timestamp("2000-12-31 12:00:00")

### List default value examples

- attribute List<Boolean> Attribute_name = List<Boolean> { TRUE, FALSE }
- attribute List<Byte> Attribute_name = List<Byte> { 11, 21 }
- attribute List<Double> Attribute_name = List<Double> { 11.100000, 21.100000 }
- attribute List<Float> Attribute_name = List<Float> { 11.100000, 21.100000 }
- attribute List<Integer> Attribute_name = List<Integer> { 11, 21 }
- attribute List<Interval> Attribute_name = List<Interval> { Interval("-19 12:00:00"), Interval("-29 12:00:00") }
- attribute List<Long> Attribute_name = List<Long> { 11, 21 }
- attribute List<Numeric> Attribute_name = List<Numeric> { Numeric(0.00, 0.00) }
- attribute List<Short> Attribute_name = List<Short> { 11, 21 }
- attribute List<String> Attribute_name = List<String> { "a","b","c" }
- attribute List<String> UTF16 Attribute_name = List<String> { UTF16("cycling"), UTF16("v\u00E9lo"), UTF16("\u81EA\u884C\u8F66") }
- attribute List<Timestamp> Attribute_name = List<Timestamp> { Timestamp("2000-12-31 12:00:00"), Timestamp("2001-01-01 12:00:00") }

## 2.5 Specifying Relationships with ODL

### Basic declaration (name and default cardinality)

There are three ways to declare a relationship:

relationship Successor_class Direct_name
relationship List <Successor_class> Direct_name
relationship Set <Successor_class> Direct_name

You may also define an inverse relationship for all these declarations:

relationship Successor_class Direct_name
    inverse Successor_class::Inverse_name
relationship List <Successor_class> Direct_name
    inverse Successor_class::Inverse_name
relationship Set <Successor_class> Direct_name
    inverse Successor_class::Inverse_name

When you declare a relationship with List or Set, the default cardinality is [0, -1] (any number of successors); without List or Set, the default cardinality is [1, 1] (one successor).

If you define and inverse relationship, the relationships must be declared in each of the two classes involved. For example:

interface Employee: persistent {
    relationship Department MemberOf inverse Department::Members;
};

interface Department: persistent {
    relationship List <Person> Members inverse Person::MemberOf;
};

Cardinality relationship ... Direct_name [min_#_succs, max_#_succs] ...

Cardinality is specified on the direct relationship part. To explicitly set cardinality for both ends of a relationship, specify it for each of the direct declarations.

When List or Set is used, cardinality if not set explicitly is [0, -1] (minimum zero, maximum unlimited): any number of successors, or none, may be specified. Thus the following two declarations are equivalent:

relationship List<Person> Members
relationship List<Person> Members [0, -1]

With List or Set, you may explicitly set any minimum and maximum, for example:

relationship List ... Direct_name [0, 2] ...
relationship List ... Direct_name [3, 10] ...
relationship List ... Direct_name [2, -1] ...

Without List or Set, cardinality if not set explicitly is [1, 1]: one and only one successor must be specified. Thus the following two declarations are equivalent:

relationship Department MemberOf inverse ...
relationship Department MemberOf [1, 1] inverse ...

The only other cardinality supported without List or Set is [0, 1]: a single successor, or none, may be specified.

Whenever the minimum cardinality is greater than 0, the relationship must always have a successor. In this case, any transaction that creates an object or clears all successors must set a successor before it can be committed.

**Order**

When you declare a relationship with List, the order of successors of the relationship is preserved. On the other hand, if you declare a relationship with Set, the order of successors need not be preserved.

**“Read-only”**

readonly relationship ...

When readonly is specified, the direct relationship can be used to get or query successors, but cannot be used to update or delete.

**Multiple successor classes**

relationship List <Successor_class_1, Successor_class_2>

A relationship may have more than one successor class: for example, the class Person might have a relationship Owner / OwnedBy with the classes Car and House.

Usually a more straightforward and flexible way to represent such real-world correspondences is with inheritance. For example, Person would have a relationship Owner / OwnedBy with the class Property, which would have subclasses Car and House.

Note that this is not an “n-ary” relationship: that is, it cannot be used to create links between Successor_class_1 objects and Successor_class_2 objects.

### 2.6 Specifying Indexes with ODL

**Index criteria**

mt_index Index_name criteria

{Class_name::Attribute1_name sort_order},
{Class_name::Attribute2_name sort_order},
{Class_name::Attribute3_name sort_order},
{Class_name::Attribute4_name sort_order};

An index may include up to four of its class’s attributes (which may be inherited from superclasses rather than declared in the class) that are referred as criteria for the index. For example:

interface Person : persistent
{
    attribute String<16> firstName;
    attribute String<16> lastName;
    mt_index personName
        criteria {person::lastName MT_ASCEND},
               {person::firstName MT_ASCEND};
sort_order must be either MT_ASCEND or MT_DESCEND.

The maximum size allowed for a key is 256 bytes. Thus the sum of the sizes for the attributes that are indexed cannot exceed 256 bytes.

Unique key

mt_index  Index_name
  unique_key TRUE
  criteria
    { Class_name::Attribute1_name sort_order} ...

If set to TRUE, each entry in the index must be unique, allowing them to be used as primary keys. The default is FALSE, so if this option is not specified the index may contain duplicate entries. unique_key must be specified before criteria.

2.7 Specifying Entry-Point Dictionaries with ODL

NOTE: Use entry-point dictionaries with attributes of type String only. For other data types, use indexes.

Basic declaration

mt_entry_point_dictionary  Dictionary_name
  entry_point_of  Attribute_name

Creates an entry-point dictionary for the attribute. Options must be specified in the order shown in this example for an attribute comment):

interface Document : persistent
{
  attribute String comment;
  mt_entry_point_dictionary commentDict
    entry_point_of comment
    unique_key FALSE
    case_sensitive FALSE
    make_entry_function "make-full-text-entry";
};

Unique key

mt_entry_point_dictionary  Dictionary_name
  entry_point_of  Attribute_name unique_key TRUE

If set to TRUE, each entry in the dictionary must be unique, allowing them to be used as primary keys. The default is FALSE, so if this option is not specified the dictionary may contain duplicate entries.

Case sensitivity

mt_entry_point_dictionary  Dictionary_name


entry_point_of Attribute_name case_sensitive TRUE

If set to TRUE, dictionary lookups are case-sensitive. Otherwise, they are case-insensitive. The default is FALSE, so if this option is not specified lookups are case-insensitive.

Function

mt_entry_point_dictionary Dictionary_name

entry_point_of Attribute_name make_entry_function "function_name"

function_name specifies the function to be used to generate the dictionary. Two such functions are included with Matisse:

- "make-entry" (default): generates one entry point per object based on the attribute value
- "make-full-text-entry": generates a separate entry point for each word in the string

If no function is specified, Matisse will use the default method, make-entry.

2.8 Specifying SQL Methods with ODL

Basic declaration

mt_method "SQL_DDL_statement"

SQL_DDL_Statement is either a CREATE METHOD statement or a CREATE STATIC METHOD statement. This generates a SQL method in the database for a class. For more information about SQL methods, refer to Matisse SQL Programmer's Guide.

When you use a double quotation mark (" within mt_method, it needs to be preceded by a backslash (\).

Example

The following example defines a method which concatenates the first name and the last name of a person:

interface Person : persistent {
    attribute String firstName;
    attribute String lastName;
    mt_method "CREATE METHOD fullName()
    RETURNS STRING
    FOR \"Person\"
    BEGIN
        RETURN CONCAT(firstName, lastName);
    END;";
};
3 Using the mt_sdl utility

Once you have defined a schema in an ODL file, you may use either the Matisse Enterprise Manager or the mt_sdl utility to load it into a Matisse database and/or generate schema-class source code for your application.

3.1 Loading a schema

First, create and initialize a database as described in the Matisse Server Administration Guide. Then use the following command to load the schema defined in your ODL file into the new database:

```
mt_sdl -d db_name@host_name import --odl -f odl_filename
```

3.2 Exporting a schema

The mt_sdl utility with the export command allows you to export the database scheam in ODL. With the -n <namespace> option, you only export the database schema under the provided namespace.

```
$ mt_sdl export -h
MATISSE Schema Definition Language x64 Version 9.1.0.0 (64-bit Edition) - Apr 29 2013.
(c) Copyright 2013 Matisse Software Inc. All rights reserved.

Usage:
mt_sdl [OPTIONS] export -f <schema file> {-o | -d} [-n <namespace>] [-h]
   -f, --file=... Specify the schema definitions script file to be generated from the database.
   -o, --odl       Generate the ODL class definitions from a database schema.
   -d, --ddl       Generate the SQL DDL script from a database schema.
   -n, --ns=...    Export only the schema objects under the provided namespace
   -h, --help      Display this help and exit.
```

3.3 Generating class files

The following command generates source code files for the classes defined in the schema:

```
$ mt_sdl stubgen -h
MATISSE Schema Definition Language x64 Version 9.1.0.0 (64-bit Edition) - Apr 29 2013.
(c) Copyright 2013 Matisse Software Inc. All rights reserved.

Usage:
mt_sdl stubgen { -l cxx [-s <namespace>] [-n <namespace>] |
   -l java [-s <namespace>] [-n <package>] [-m] |
   -l php [-s <namespace>] [-n <namespace>] |
   -l python [-s <namespace>] |
   -l eiffel [-s <namespace>] } -f <ODL file> [-h]
   -f, --file=... Specify the ODL class definitions file.
```
Using the mt_sdl utility

Depending on whether you choose cxx, eiffel, java, python, or php a .h, .e, .java, .py, or .php file is created in the current directory for each class defined in the specified ODL file. For C++, a .cpp source file with the same name as the ODL file is also created; Specifying the --sn ns_name and --ln ns_name option associates the classes with the specified Java package or PHP or C++ namespace. With -psm option, the SQL method calls are generated.

The portion of the class files that you should not modify are bracketed by the following comments:

```
// BEGIN generated code
// END of generated code
```

Add any additional methods required by your application after the END comment or before BEGIN.

mt_sdl places some sample user methods after the generated code. You may modify or delete these methods as appropriate. If you delete these methods, they will not be replaced by mt_sdl if it is run again to update the class with schema modifications. If you modify them, be sure to modify the preceding comment accordingly.

See the programmer’s guide for the C++, Java, Objective C, PHP, Python and Eiffel bindings for more information on mt_sdl output.

For the class file generation of other languages (C#, VisualBasic), please refer to the documentation of each language binding.

### 3.4 Updating an application

If you modify an ODL schema definition, you may run

mt_sdl again to update the database and/or source code with the changes, with the following limitations:

- You cannot rename attributes or relationships, indexes, or entry-point dictionaries. (You can rename them with SQL DDL.)

For classes that have instances (objects):
You cannot change inheritance, for example, you cannot change `interface Employee : Person` to `interface Manager : Employee : Person`.

You cannot change an attribute descriptor's type.

You cannot remove the default value from a non-nullable attribute.

You cannot make the cardinality of a relationship descriptor more restrictive. For example, you can change `[1, 1]` to `[0, 1]` or `[1, -1]`, but you cannot change `[0, -1]` to `[1, -1]` or `[0, 1]`.

If you need to make any of the above changes:

1. Use `mt_xml` to export all of the data.
2. Initialize the database.
3. Use `mt_sdl` to generate source code and load the updated schema.
4. Use `mt_xml` to load the previously exported data.
5. For any renamed classes, copy any user methods you added to the `mt_sdl`-generated source code from the old generated file to the new name, then delete the old file.

See the `Matisse XML Programming Guide` for instructions on using `mt_xml`.

### 3.5 Preprocessing a schema file

The Matisse ODL files support pre-processing directives (for example - `#include`) that allow you to include other ODL files.

To use this capability, you must set the `MATISSE_CPP` environment variable to the location of a preprocessor.

**UNIX**

```
MATISSE_CPP=`which cpp`; export MATISSE_CPP
or with spaces in the path
export MATISSE_CPP='/opt/tools/my compiler/cpp' -DMYVAR=1
```

**MS-Windows**

```
set MATISSE_CPP=cl.exe -E
or
set MATISSE_CPP="C:\<install path>\cl.exe" -E
or
set MATISSE_CPP="C:\<install path>\cl.exe" /EP /C
or to use it from Matisse Enterprise Manager
SET MATISSE_CPP="C:\<install path>\vcvarsall.bat" x86 & & "C:\<install path>\Bin\cl.exe" -E
```
4 Extending the Meta-Schema with ODL

This section covers Matisse Meta-Schema extensions. While you are still able to generate persistent stub classes from the ODL file or from the database schema, no methods are generated to access or update your Meta-Schema extensions. These extensions can be accessed using the Matisse discovery (reflection) APIs.

This section is indented for advanced Matisse developers.

4.1 Matisse Meta-Schema

Matisse Meta-Schema relies on the three basic concepts of Class, Attribute and Relationship to be described. Matisse Meta-Schema is comprised of a set of classes which provides the necessary means to describe an application database schema with an Object-Oriented model.

4.2 Modifying the Meta-Schema

Matisse Meta-Schema can be extended to meet your schema description needs that go beyond Matisse Object-Oriented model. As in Matisse, Meta-Classes, Classes and Instances are objects, developers could modify the Meta-Schema with the same programming API used to create or modify classes and instances. However, to simplify Meta-Schema extensions, Matisse ODL includes specific features to describe the Meta-Schema extensions.

The Matisse Meta-Classes (i.e. MtClass, MtAttribute, MtRelationship, etc.) could be described in ODL like any of your application schema classes. However, it would be tedious to have to specify the full definition of the Meta-Schema classes to be extended. So only added properties need to be described when added to pre-defined Meta-Classes.

**NOTE:** Modifying the Meta-Schema is quite complex. So in order to limit invalid or inappropriate modifications, we recommend that you import the ODL file, export the schema in a new ODL file and then check that the exported ODL description matches to your original schema.

4.3 Extending Meta-Schema Classes

Matisse allows you to create your own Meta-Classes. To do this, you need to create a sub-class of a pre-defined Meta-Class. To specify that a class in your schema is an instance of your newly created Meta-Class, use the `mt_instance_of` ODL clause.

```java
// Meta-Schema Extensions
interface KClass : MtClass : persistent
{
}
```

You are then able to refer to the new Meta-Class in your class definition as follows:
4.4 Adding an Attribute to a Meta-Class

Matisse allows you also to add new attributes to a Meta-Class, whether it is your own or a Matisse one. Matisse ODL allows you to specify the newly created attribute by using the \texttt{mt\_property ODL} clause. For example, suppose you want to associate a comment with each class definition. To do this, you add an attribute \texttt{Description} of type \texttt{String} to the \texttt{MtClass} definition as follows:

```
// Meta-Schema Extensions
interface MtClass : persistent
{
    attribute String Description = String(NULL);
};
```

You are then able to use the new attribute in your class definition as follows:

```
// Application Classes
interface Person : persistent
{
    mt\_property Description = "My application comments for Person";

    attribute String FirstName = String(NULL);
    attribute String LastName = String(NULL);
};
```

4.5 Adding a Relationship to a Meta-Class

New relationships can also be added your own Meta-Class or a pre-defined one. The \texttt{mt\_property ODL} clause also enables to specify these relationships in your application class. For example, let's suppose that you need to keep a list of public attributes in your application class.

```
// Meta-Schema Extensions
interface MtClass : persistent
{
    relationship List<MtAttribute> PublicAttributes
        inverse MtAttribute::PublicAttributeOf;
};

interface MtAttribute : persistent
{
    relationship MtClass PublicAttributeOf [0, 1]
        inverse MtClass::PublicAttributes;
};
```

You are then able to use the new relationship in your class definition as follows:
// Application Classes
interface Person : persistent
{
   attribute String FirstName = String(NULL)
   mt_property PublicAttributeOf = {interface(Person)};
   attribute String LastName = String(NULL)
   mt_property PublicAttributeOf = {interface(Person)};

   mt_property PublicAttributes = {
      attribute(Person::FirstName),
      attribute(Person::LastName) 
   };
}

CAUTION: Both the relationship (PublicAttributes) and its inverse
(PublicAttributeOf) must be specified in the class definition.

The value of a relationship always refer to a list of Attribute, Relationship, Class or a combination of the
three which can be expressed as follows:

• a list of attributes:
  attribute_list {"class1::att1", "class1::att2"}
  {attribute("class1::att1"), attribute("class1::att2"))

• a list of relationships:
  relationship_list {"class1::rel1", "class1::rel2"}
  {relationship("class1::rel1"), relationship("class1::rel2")}

• a list of classes:
  interface_list {"class1", "class2"}
  {interface("class1"), interface("class2")}

• a list of attributes and relationships:
  property_list {"class1::att1", "class1::rel1"}
  {attribute("class1::att1"), relationship("class1::rel1")}

• a list of attributes, relationships and classes:
  { attribute("class1::att1"),
    relationship("class1::rel1"),
    interface("class1") } 

4.6 Extending Matisse MtAttribute

Matisse enables you to define your own attribute description class by creating a sub-class of Matisse
MtAttribute. The mt_instance_of ODL clause allows you to specify that an attribute in your
schema is an instance of your newly created Meta-Class. For example, suppose that you need to define an
alias to each attribute in your application classes.

// Meta-Schema Extensions
interface KAttribute : MtAttribute : persistent
{
   attribute String Alias = String(NULL);
You are then able to use the new attribute Meta-Class in your class definition as follows:

```java
// Application Classes
interface Person : persistent {
    attribute String FirstName = String(NULL)
    mt_property Alias = "Person First Name"
    mt_instance_of KAttribute;
};
```

### 4.7 Extending Matisse MtRelationship

Matisse enables you to define your own relationship description class by creating a sub-class of Matisse MtRelationship. The `mt_instance_of ODL` clause allows you to specify that a relationship in your schema is an instance of your newly created Meta-Class. For example, suppose that you want to specify whether or not a relationship is indexed.

```java
// Meta-Schema Extensions
interface KRelationship : MtRelationship : persistent {
    attribute Boolean Indexed = FALSE;
    attribute String IndexKey = String(NULL);
};
```

You are then able to use the new relationship Meta-Class in your class definition as follows:

```java
// Application Classes
interface Document : persistent {
    relationship List<Topic> Topics
    inverse Topic::Documents
    mt_property Indexed = TRUE
    mt_property IndexKey = "Topic::Weight"
    mt_instance_of KRelationship;
};
```

```java
interface Topic : persistent {
    attribute String Name;
    attribute Integer Weight = 0;
    relationship List<Document> Documents
    inverse Document::Topics
    mt_instance_of KRelationship;
};
```
4.8 Extending Matisse MtClass

Matisse allows you to extend Matisse MtClass so you can enrich the description of your application classes. The `mt_instance_of ODL` clause allows you to specify that a class in your schema is an instance of your newly created Meta-Class. For example, suppose that you want to extend Matisse class description to support a description field and to list the properties that are public for the application.

```java
// Meta-Schema Extensions
interface KClass : MtClass : persistent
{
    attribute String Description = String(NULL);

    relationship List<KAttribute> PublicAttributes
        inverse KAttribute::PublicAttributeOf;
    relationship List<KRelationship> PublicRelationships
        inverse KRelationship::PublicRelationshipOf;
};

interface KAttribute : MtAttribute : persistent
{
    attribute String Alias = String(NULL);

    relationship KClass PublicAttributeOf [0, 1]
        inverse KClass::PublicAttributes;
};

interface KRelationship : MtRelationship : persistent
{
    attribute Boolean Indexed = FALSE;
    attribute String IndexKey = String(NULL);

    relationship KClass PublicRelationshipOf [0, 1]
        inverse KClass::PublicRelationships;
};
```

You are then able to use the new class Meta-Class in your application class definition as follows:

```java
// Application Classes
interface Document : persistent
{
    mt_instance_of KClass;

    attribute String Title
        mt_instance_of KAttribute
        mt_property PublicAttributeOf = {interface(Document)}
        mt_property Alias = "Document Title";

    attribute String CreationDate
        mt_instance_of KAttribute
        mt_property Alias = "Document Creation Date";

    relationship List<Topic> Topics
```
interface Topic : persistent
{
    mt_instance_of KClass;

    attribute String Name
    mt_instance_of KAttribute
    mt_property PublicAttributeOf = {interface(Topic)}
    mt_property Alias = "Topic Name";

    attribute Integer Weight = 0
    mt_instance_of KAttribute
    mt_property Alias = "Topic Weight";

    relationship List<Document> Documents
    inverse Document::Topics
    mt_instance_of KRelationship;

    mt_property Description = "My application comments about Topic";
    mt_property PublicAttributes = { attribute(Topic::Name) };
};
5 Sharing Properties with ODL

This section covers Matisse ability to share property definitions amongst classes without relying on inheritance. The term property is used to designate an attribute or a relationship.

This section is intended for advanced Matisse developers.

5.1 Shared Properties

In Matisse, properties are objects like any other in the database. Therefore they can exist outside of a class definition and can then be shared by multiple classes. If you consider that an object is a set of properties and that any given property has a unique meaning (described by set of characteristics), then a property can be shared by objects of various types.

In most object-oriented languages, the scope of a property is the class and its subclasses, so there is no ODL syntax to describe properties outside of a class definition. The `mt_attribute` and `mt_relationship` ODL clauses allow you to define respectively a shared attribute and a shared relationship outside of a class definition.

**NOTE:** shared properties are global to the database, their name must be unique.

**CAUTION:** Shared properties are not supported by SQL DDL.

5.2 Defining a Shared Attribute

Matisse allows you to define attributes shared by multiples classes. The `mt_attribute` ODL clause allows you to specify that an attribute is shared. The attribute is only defined once. When you need to refer to the attribute in a class definition, declare it by specifying its name after the clause `mt_attribute`. The unique definition of the attribute can be made within the definition of a class, or outside of it. For example, suppose that you want the attribute Comment to be part of many of your application classes, but you don’t want these classes to inherit from a common super-class containing the Comment attribute and/or that you don’t want to use multiple inheritance.

```java
// Application Classes

mt_attribute String Comment = String(NULL);

interface Document : persistent
{
    attribute String Title;
    mt_attribute Comment;
};

interface Topic : persistent
{
    attribute String Name;
    mt_attribute Comment;
}`
```
5.3 Defining a Shared Relationship

Matisse allows you to define relationships shared by multiples classes. The `mt_relationship` ODL clause allows you to specify that an attribute is shared. The relationship is only defined once. When you need to refer to the relationship in a class definition, declare it by specifying its name after the clause `mt_relationship`. For example, suppose you want to link on your front page the most popular items which happens to be of different kinds.

```c
// Application Classes
mt_relationship List<{News, Person, Location}> MostPopular
    inverse OnFrontPages;
mt_relationship Set<FrontPage> OnFrontPages inverse MostPopular;

interface FrontPage : persistent
{
    mt_relationship MostPopular;
};

interface News : persistent
{
    mt_relationship OnFrontPages;
};

interface Person : persistent
{
    mt_relationship OnFrontPages;
};

interface Location : persistent
{
    mt_relationship OnFrontPages;
};
```

5.4 Defining a Shared Index

Matisse allows you to define an index on shared attributes. This declaration must be made outside the scope of a class definition. The `mt_classes` clause is used to specify the indexed classes. The other characteristics of the definition are similar to the definition of an index inside a class definition. The definition of such an index is as follows:

```c
// Application Classes

mt_attribute String<64> Name = String(NULL);

interface Nation : persistent
{
```


```java
mt_attribute Name;
};

interface Topic : persistent
{
    mt_attribute Name;
};

interface Ship : persistent
{
    mt_attribute Name;
};

mt_index SharedNameIdx
    mt_classes { Nation, Topic, Ship }
    criteria { Name MT_ASCEND }; 

5.5 Defining a Shared Entry-Point Dictionary

Matisse allows you to define an Entry-Point Dictionary on a shared attribute of type String. This declaration must be made outside the scope of a class definition. The syntax for the description of an Entry-Point dictionary is identical to the one used to define a Entry-Point Dictionary inside a class description. The definition of an Entry-Point Dictionary on a shared attribute of type String is as follows:

    // Application Classes
    mt_attribute String Comment = String(NULL);
    mt_entry_point_dictionary commentDict entry_point_of Comment
        unique_key FALSE
        case_sensitive FALSE
        make_entry_function "make-full-text-entry";

    interface Document : persistent
    {
        attribute String Title;
        mt_attribute Comment;
    };

    interface Topic : persistent
    {
        attribute String Name;
        mt_attribute Comment;
    };
```
Index

**A**
- attribute: 9, 23
- attribute_list: 23

**B**
- Boolean: 13
- Byte: 13

**C**
- Cardinality: 14
- case_sensitive: 17
- Char: 13
- CREATE METHOD: 17
- criteria: 15

**D**
- Date: 13
- Default value: 12
- Define a class: 8
- Define a namespace: 7
- Double: 13

**E**
- entry_point_of: 16
- Extend a class: 8

**F**
- Float: 13

**I**
- Integer: 13
- interface: 8, 23
- interface_list: 23
- Interval: 13
- inverse: 14

**L**
List default value: 13
List types: 11
Long: 13

M
make_entry_function: 17
make-entry: 17
make-full-text-entry: 17
Matisse Meta-Schema extensions: 21
MATISSE_CPP: 20
module: 7
mt_attribute: 27
mt_classes: 28
mt_entry_point_dictionary: 16
mt_index: 15
mt_instance_of: 21, 23, 24, 25
mt_method: 17
mt_property: 22
mt_relationship: 27
mt_sdl: 18

N
ndex criteria: 15
Nullable: 11
Numeric: 13

O
ODL: 4
Order: 15

P
persistent: 8
property_list: 23

R
readonly: 15
relationship: 13, 23
relationship List: 14
relationship Set: 14
relationship_list: 23
S
shared attribute: 27
Shared Entry-Point Dictionary: 29
Shared Index: 28
Shared properties: 27
shared relationship: 27
Short: 13
SQL DDL: 4
String: 13

T
Text: 13
Timestamp: 13

U
unique_key: 16
UTF16: 13