

Matisse[®] Perl Programmer's Guide

August 2009



Matisse Perl Programmer's Guide

Copyright ©1992–2009 Matisse Software Inc. All Rights Reserved.

Matisse Software Inc.
930 San Marcos Circle
Mountain View, CA 94043
USA

Printed in USA.

This manual is copyrighted. Under the copyright laws, this manual may not be copied, in whole or in part, without prior written consent of Matisse Software Inc. This manual is provided under the terms of a license between Matisse Software Inc. and the recipient, and their use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and international patents.

TRADEMARKS: Matisse and the Matisse logo are registered trademarks of Matisse Software Inc. All other trademarks belong to their respective owners.

PDF generated 21 August 2009

Contents

1	Introduction	5
	Scope of This Document	5
	Before Reading This Document	5
	Required Environment	5
2	Installing Matisse Perl Binding	6
	Download Matisse Perl Binding	6
	Before Installing Matisse Perl Binding	6
	How to Install Matisse Perl Binding	6
	How to Use the Binding	7
	Test Programs for Checking the Binding	7
	How to Execute the Tests	7
3	Connecting	10
	Transaction	10
	Cancelling Transaction	11
	Read-Only Access	11
4	Working with Objects and Values	13
	Before Running Examples	13
	Working with Objects	13
	objects.pl Source	13
	Working with Values	14
	values.pl Source	14
5	Working with Relationships	17
	test.odl Source	17
	Running Relationships Example	17
	relationships.pl Source	17
6	Working with Indexes	20
	Running indexes Example	20
	indexes.pl Source	20
7	Working with Entry-Point Dictionaries	23
	Running EPDict Example	23
	EPDict.pl Source	23
8	Using SQL Statement in Perl Code	25
	Running queryBySQL Example	25
	queryBySQL.pl Source	25
9	Working with Versions	28
	versions.pl Source	28

1 Introduction

Scope of This Document

This document is intended to help Perl programmers to build Matisse-Perl binding and learn the aspects of Matisse design and programming that are unique to the Matisse-Perl binding.

Aspects of Matisse programming that the Perl binding shares with other interfaces, such as basic concepts and schema design, are covered in *Getting Started with Matisse*.

Future releases of this document will add more advanced topics. If there is anything you would like to see added, or if you have any questions about or corrections to this document, please send e-mail to support@matisse.com.

Before Reading This Document

Throughout this document, we presume that you already know the basics of Perl programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

Required Environment

- Matisse 8.3.x or higher
- Perl 5.8.x or higher
Path to executable must NOT include space(s) if you are using Windows.
- C compiler (gcc on Unix, Linux, Visual C++ on MS Windows)

2 Installing Matisse Perl Binding

Download Matisse Perl Binding

Download and extract the Matisse Perl binding package 8.3 or later from the Matisse Web site. It is recommended to install the binding in a subdirectory (MATISSE-Perl) of Matisse directory.

<http://www.matisse.com/developers/downloads/>

Before Installing Matisse Perl Binding

Make sure the following environmental variables are appropriately set so that the compiler/linker can find paths.

- For Windows:
To set environment variables: Control Panel-> System->Advanced->Environmental Variables
 - Path for Perl library is included in the LIB environmental variable.
 - Path for windows.h is included in the INCLUDE environmental variable.
 - Path for nmake.exe is included in the PATH environmental variable.
 - MATISSE_HOME variable is set to the top-level directory of the Matisse installation.
- For Unix:
MATISSE_HOME: set to the root directory of the Matisse installation, e.g., /usr/local/matisse. Note that if you executed mt_env.sh or mt_env.csh in the Matisse installation directory, the environment variable is already set.

How to Install Matisse Perl Binding

Follow the instructions below to install the Matisse Perl binding.

- Change current directly to MATISSE-Perl directory. Create the Makefile from the Makefile template.

```
% perl Makefile.PL
```

This should create the appropriate Makefile based on supplied environment variables and system configuration.

- Install the binding

```
% make (nmake on Windows)
```

Note: gnu make may be used instead of make.

A successful make will create the necessary local module hierarchy containing the linked dynamic library. If you have root access, you can install the binding into the default perl location by running:

```
% make install (nmake install on Windows)
```

Or else, you can override the install locations and install where you have permissions, but then you will have to append the location of locally installed modules before running.
(See Perl documentation concerning MakeMaker)

How to Use the Binding

In your perl program add the following:

```
use Matisse;
```

Execute the script with appropriate module search options. If the binding is not installed, you will need to specify the `-Mlib` options as well.

Test Programs for Checking the Binding

There are basic test programs to quickly and simply test your compiled binding. More advanced examples, which would be used as the basis for your own applications, will be shown later in this document.

How to Execute the Tests

- Create and initialize a database. You can simply start the `mt_emgr` tool, select the database 'example' and right-click on 're-initialize'. For more options, please refer to the *Getting Started with Matisse* document.

- To load a schema, use an `odl` file in the test directory.

```
mt_sdl -d <database> import -odl test/test.odl
```

- To instantiate objects, use `test.sql` in the test directory.

```
mt_sql -d <database> < test/test.sql
```

- To check the instantiated objects:

```
mt_sql -d <database>
```

You will see `sql>` prompt. Run any SQL statement. For example:

```
select * from Person;
```

To exit from the `sql>` prompt, type:

```
quit
```

Now you are ready to run four OO tests and two SQL tests in the `test` subdirectory. You may run the tests without `-Mlib` option after the binding is successfully installed. The following instructions assumes that you are in `MATISSE-Perl` directory.

- **ootest1.pl**
A test to create an instance of MtClass.

```
perl -Mblib test/ootest1.pl <host> <database>
```

- **ootest2.pl**
A test to list class names.

```
perl -Mblib test/ootest2.pl <host> <database>
```

Expected Results:

A list of classes including Manager, Employee, and Person

- **ootest3.pl**
Before running ootest3, you need to invoke code generator for each class. To do so, you may use gen.pl which generates the codes, but the generator may also be invoked as a one-liner, e.g.

```
perl -Mblib -MMatisse::SourceGen -e 'Matisse::SourceGen::genFromDb(host=>"<host>",database=>"<database>");'
```

Use double quotation " for Windows, instead of single quotation '. Again, the above should be typed in ONE line, without any new line characters.

Expected Results:

Three package files, Manager.pm, Employee.pm, and Person.pm, will be generated.

Class Codes are Generated from Database

To generate class codes using gen.pl of Perl binding, an online database loaded with the schema *must* exist, as it will generate code for all classes *defined in the database*. This is unique to Perl binding compared to other language bindings where mt_sdl is used to generate class codes from the schema, hence, an active database is not required.

To run ootest3.pl

```
perl -Mblib test/ootest3.pl <host> <database>
```

Expected Results:

A list of five Person objects with attributes -- name, age, comments, and vote for each person. Run the ootest3.pl a few more times. You will see that the program appended an 'X' to each person's name per run. This demonstrates that the objects are appropriately updated each time.

- **ootest4.pl**
If you have just run ootest3.pl, run SQL statements in test.sql again so that you will have clean objects without appended 'X's.

To run the test:

```
perl -Mblib test/ootest4.pl <host> <database>
```

Expected Results:

```
Connecting to <database>@<host> (1)
Employee Dick, id=101
Employee Sally, id=2
  has boss Dick
Employee Ralph, id=1
  has boss Dick
Manager Dick
..has staff: Ralph Sally
```

Disconnecting from <database>@<host>(1)

If you skip running test.sql, you will see 'X's appended to each person's name.

- **sqltest.pl**

A test for executeQuery method with "SELECT MtName FROM MtClass".

To run the test:

```
perl -Mblib test/sqltest.pl <host> <database>
```

Expected Results:

A list of classes including Person, Employee, and Manager.

- **sqltest2.pl**

A test for getObject method.

To run the test:

```
perl -Mblib test/sqltest2.pl <host> <database>
```

Expected Results:

A list of five objects -- the class name followed by 'name' for each object.

3 Connecting

All interaction between client Perl applications and Matisse databases takes place within the context of transactions (either explicit or implicit) established by database connections, which are transient instances of the MtDatabase class. Once the connection is established, your Perl application may interact with the database using the schema-specific methods generated by gen.pl. The following sample code shows a variety of ways of connecting with a Matisse database.

Before running the examples in this section, you need to initialize a database. To do so, start the mt_emgr tool, select the database 'example' and right-click on 're-initialize'.

To run an example program in examples directory, type the following if you are in MATISSE-Perl directory.

```
perl examples/<filename> <host> <database>
```

Transaction

The following code connects to a database, starts a transaction, commits the transaction, and closes the connection.

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/transaction.pl <host> <database>
#
# How to access Matisse database

use Matisse;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

# Create a new connection handle with the host and database
my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();          # establish the connection
$db->select();        # select as current database
$db->startTransaction(); # open a transaction context

#
# management of your data
#

print "\nConnection and read-write access to ", $db->getName(), "\n\n";

$db->commit();        # commit the changes
$db->unselect();      # unselect the currently selected database
$db->close();         # close the connection

__END__
```

Whenever the access control is enabled, you may want to connect to the database using a specific username and password. These can be specified when calling the `open` method.

```
$db->open(username => "guest", password => "guestpw");
```

Multi-database access can be managed by opening several connections and switching from one connection to another. You can always retrieve the currently selected connection with the `MtDatabase::getCurrentMtDatabase()` method.

```
my $db1 = new Matisse::MtDatabase( host => "host1", database => "database1");
my $db2 = new Matisse::MtDatabase( host => "host2", database => "database2");

# Establish the connections
$db1->open();
$db2->open();

# Select a current database
$db2->select();
# ... do something with database 2
# Unselect the current database and select another one
$db2->unselect();
$db1->select();
# ... do something with database 1

$db1->close();
$db2->close();
```

Cancelling Transaction

When a transaction is started, it may be cancelled at any time by using the `rollback` method.

```
# open a transaction context
$db->startTransaction();

#
# change your data
#

# Rollback the changes
$db->rollback();
```

Read-Only Access

The following code connects to a database in read-only mode, suitable for reports.

```
# Copyright (c) 1992-2006 Matisse Software Inc. All Rights Reserved.
# This file (in both binary and source code form) is subject to the Supplemental
# License Terms governing use, modification and redistribution of Sample Code
# accompanying the Matisse(R) software.
#
# Syntax:
```

```
# perl readOnly.pl <host> <database>
#
# How to open and close read-only connection

use Matisse;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();           # establish the connection
$db->select();
$db->startVersionAccess(); # open read only connection

# read data

print "\nConnection and read only access to ", $db->getName(), "\n\n";

$db->endVersionAccess(); # close read only connection
$db->unselect();
$db->close();           # close the connection

__END__
```

4 Working with Objects and Values

Before Running Examples

Running sample programs in this section requires schema created by `test.odl` in the test directory. Initialize a database and change your current directory to `MATISSE-Perl`. Load `test.odl` into the database.

```
mt_sdl -d <database> import -odl test/test.odl
```

Generate source files for each class. Type the following in one line at the command prompt. Use double quotation “ instead of single quotation ‘ for Windows.

```
perl -MMatisse::SourceGen -e 'Matisse::SourceGen::genFromDb(host=>"<host>",database=>"<database>");'
```

Working with Objects

This sample program creates two objects; one `Person` and one `Employee` objects. Lists all `Person` objects (which includes both objects, since `Employee` is a subclass of `Person`), deletes both objects, then lists all `Person` objects again to show the deletion.

The sample program is in `examples` directory under `MATISSE-Perl` directory. To run the program:

```
perl examples/<filename> <host> <database>
```

objects.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/objects.pl <host> <database>
#
# This sample program demonstrates how to create, remove, and list objects.
# Run test.odl to load the schema before running this program.

use Matisse;
use Person;
use Employee;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

# create a new connection handle with the host and database
my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();          # establish the connection
$db->select();        # select as current database
$db->startTransaction(); # open a transaction context
```

```

# create a new Person object (instance of Person class)
my $person = new Person();
# set attribute values
$person->setName("John");
$person->setAge(30);

# create a new Employee object which extends Person class
my $emp = new Employee();
$emp->setName("Susan"); # Person attribute
$emp->setAge(35);      # Person attribute
$emp->setId(101);     # Employee attribute

# list all Person objects
listPersonObjects();

# remove created objects
$person->remove();
$emp->remove();

# list all objects again to show deletion
print "After removing the objects:\n";
listPersonObjects();

$db->commit();      # commit the changes
$db->unselect();   # unselect the currently selected database
$db->close();      # close the connection

sub listPersonObjects {
# the details for the codes in this subroutine is discussed later
print Person::getInstanceNumber($db), " Person(s) in the database\n";
my $iter = Person::instanceIterator($db);
while ($iter->hasNext())
{
my $o = $iter->next();
print " ", $o->getName(), ", age=", $o->getAge(), "\n";
}
print "\n";
}

__END__

```

Working with Values

This example uses a database created for `objects.pl`. It creates an object, then manipulates its values in various ways as described in the source code comments.

The sample program is in `examples` directory under `MATISSE-Perl` directory. To run the program:

```
perl examples/<filename> <host> <database>
```

values.pl Source

```

# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the

```

```

# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl -Mlib examples/values.pl <host> <database>
#
# This sample program demonstrates how to manipulates attribute values in
# various ways. Requires a database and class files created for objects.pl.

use Matisse;
use Person;
use Employee;

unless (scalar(@ARGV) == 2) {die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();           # establish the connection
$db->select();         # select as current database
$db->startTransaction(); # open a transaction context

# create two new Employee object
my $emp1 = new Employee();
my $emp2 = new Employee();

# set String
$emp1->setName("Nancy");
$emp2->setName("Scott");

# set numeric
$emp1->setAge(29);
$emp1->setId(110);
$emp2->setId(120);

# retrieve an attribute value by get<Att> method
print "\nNew employee: ", $emp1->getName(), ", ", $emp1->getAge(), ", ", $emp1->getId(), "\n";

# change attribute values by set<Att> method
$emp1->setId(115);
# check name before making changes
if ($emp1->getName() eq "Nancy"){
    $emp1->setAge(30);
}

# show the changes
print "Updated data for ", $emp1->getName(), ": age=", $emp1->getAge(), " id=", $emp1->getId(), "\n\n";

# you will get an error if you attempt to retrieve a value which is null
# Check if the attribute is null, if you are not sure, by is<Att>Null method
if ($emp2->isAgeNull()){
    print "Age of ", $emp2->getName(), " is null\n\n";
}
else {
    print "Age of ", $emp2->getName(), " is ", $emp2->getAge(), "\n\n";
}

# set an attribute value that is currently null

```

```

$emp2->setAge(35);
# show the results
print "Updated data for ", $emp2->getName(), ": age=", $emp2->getAge(), " id=", $emp2->getId(), "\n\n";

# remove a nullable value by remove<Att> method
$emp2->removeAge();
# show the results
print "Scott's age is removed with remove<att> method\n";
if ($emp2->isAgeNull()){
    print "Age of ", $emp2->getName(), " is null\n\n";
}
else {
    print "Age of ", $emp2->getName(), " is ", $emp2->getAge(), "\n\n";
}

# Notes: if you attempt to remove an attribute value that is not nullable,
# you will get an error. e.g. $emp1->removeName() will generate an error

$db->commit();          # commit the changes

# read-only access
$db->startVersionAccess();
print "Read only access
Data for ", $emp2->getName(), " id=", $emp2->getId(), "\n\n";

# Since this transaction is read-only, you cannot set or change
# any attributes here. i.g. $emp2->setId(200) will cause an error.

# end of read only access
$db->endVersionAccess();

# start read-write access again to remove objects
$db->startTransaction();
$emp1->remove();
$emp2->remove();

$db->commit();
$db->unselect();
$db->close();

__END__

```

5 Working with Relationships

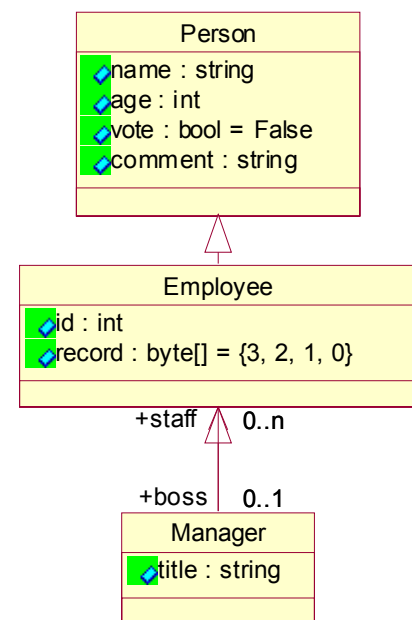
This example creates several objects and manipulates the relationships among them in various ways as described in the source code comments. The schema described in test.odl is depicted below. Manager may have any number (0 or more) of Employee as staff. An Employee is allowed to have 0 or 1 Manager as his/her boss. As shown in test.odl source below, note that the boss relationship is read only for Employee. Thus, the relationship is allowed to add/delete only from Manager object. When a relationship of staff is set from a Manager to an Employee, the relationship of boss for the Employee is automatically updated.

test.odl Source

```
interface Person : persistent {
  attribute String name;
  attribute Integer Nullable age;
  attribute String Nullable comment;
  attribute Boolean vote = MT_FALSE;
};

interface Employee : Person : persistent {
  attribute Integer id;
  readonly relationship Manager boss[0,1]
    inverse Manager::staff;
  attribute List<Byte> record = List<Byte>{3,2,1,0};
};

interface Manager : Employee : persistent {
  attribute String title;
  relationship List<Employee> staff
    inverse Employee::boss;
};
```



Running Relationships Example

Running relationships.pl requires schema created by test.odl in the test directory. See [Before Running Examples](#) to load test.odl and generate class files.

To run relationships.pl:

```
perl examples/relationships.pl <host> <database>
```

relationships.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/relationships.pl <host> <database>
#
```

```

# This sample program demonstrates how to access, modify, and navigate
# relationships.

use Matisse;
use Person;
use Employee;
use Manager;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();
$db->select();
$db->startTransaction();

# delete all objects that are currently exists in the database.

# create one Manager and three Employee objects
my $m1 = new Manager();
$m1->setName("Heather");
$m1->setAge(40);
$m1->setId(500);
$m1->setTitle("Director");

my $e1 = new Employee();
$e1->setName("Amy");
$e1->setAge(25);
$e1->setId(130);

my $e2 = new Employee();
$e2->setName("Bob");
$e2->setAge(24);
$e2->setId(131);

my $e3 = new Employee();
$e3->setName("Michael");
$e3->setAge(41);
$e3->setId(132);

# set a relationship: Manager (Heather) has two Employees (Amy and Bob) as staff
$m1->appendStaff($e1);
$m1->appendStaff($e2);

# show the results
print "\nNumber of staff for ", $m1->getName(), " is ", $m1->getStaffSize(), "\n";

# As a Manager may have multiple staff, use iterator to list all staff names,
# which is coded as a subroutine.
print "Staff list for ", $m1->getName(), ":\n";
listStaff($m1);

# 'prepend' adds a staff at the beginning of the list
$m1->prependStaff($e3);
# show the results
print "Michael added at the top\nStaff list for ", $m1->getName(), ":\n";
listStaff($m1);

```

```

# relationship of 'boss' for Employee is automatically updated.
# no need to use iterator this time, as an Employee is allowed to have
# only 0 or 1 boss,
print $e1->getName(), " has boss ", $e1->getBoss()->getName(), "\n";
print $e2->getName(), " has boss ", $e2->getBoss()->getName(), "\n";
print $e3->getName(), " has boss ", $e3->getBoss()->getName(), "\n\n";

# check if two employees have the same boss.
print "Do ", $e1->getName(), " and ", $e2->getName(), " have the same boss?\n";
if (($e1->getBoss()->equals($e2->getBoss())){
    print "Yes! \n\n";
}
else {
    print "No! \n\n";
}

# remove<Relationship> method removes the link, only, not the employee object
# remove Amy ($e1) from Heather($m1)'s staff
print "remove ", $e1->getName(), " from staff\n";
$m1->removeStaff($e1);

# show the results
print "Updated staff list for ", $m1->getName(), ": \n";
listStaff($m1);

# To remove ALL relationships of staff-boss, use clear<Rel> method.
# Similar to remove<Rel> method, clear<Rel> removes the link only, not objects.
$m1->clearStaff();
print "All staff removed from staff list\nStaff list for ", $m1->getName(), ":\n";
print "\nNumber of staff for ", $m1->getName(), " is ", $m1->getStaffSize(), "\n";
listStaff($m1);
print "\n";

$m1->remove();
$e1->remove();
$e2->remove();
$e3->remove();

$db->commit();
$db->unselect();
$db->close();

sub listStaff {
    my $m = shift;
    if ($m->getStaffSize() > 0)
    {
        my $iterator = $m->staffIterator();
        while ($iterator->hasNext())
        {
            my $e = $iterator->next();
            print " ", $e->getName(), " id=", $e->getId();
        }
        print "\n\n";
    }
}

__END__

```

6 Working with Indexes

The following example demonstrates two methods, lookup and iterator, that are created for every index defined for a database.

- Use lookup method for retrieving a single object. The method returns an object upon successful single match. Generates error if multiple matches were found. Returns 0 if matching object was not found.
- Use iterator for retrieving a range of objects

See the comments in the codes for deleted implementation.

Running indexes Example

Running `indexes.pl` requires schema created by `test.odl` in the test directory. See [Before Running Examples](#) to load `test.odl` and generate class files.

To run `indexes.pl`:

```
perl examples/indexes.pl <host> <database>
```

indexes.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/indexes.pl <host> <database>
#
# This sample program demonstrates how to use lookup method and iterator
# to retrieve object(s) with indexed attribute.
# The example requires a database with a schema created by test.odl

use Matisse;
use Person;
use Employee;
use Manager;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();           # establish the connection
$db->select();         # select as current database
$db->startTransaction(); # open a transaction context

# delete all existing Person objects using a SQL statment
my $stmt = new Matisse::MtStatement(database=>$db);
$stmt->executeQuery("DELETE FROM Person");
```

```

$stmt->close();

# create four Person objects
my $p1 = new Person();
$p1->setName("Carmen");
$p1->setAge(31);
my $p2 = new Person();
$p2->setName("Don");
$p2->setAge(32);
my $p3 = new Person();
$p3->setName("Fred");
$p3->setAge(33);
my $p4 = new Person();
$p4->setName("Steve");
$p4->setAge(34);

print Person::getInstanceNumber($db), " Person(s) available in $dbName\n";
my $iter = Person::instanceIterator($db);
while ($iter->hasNext())
{
    my $o = $iter->next();
    print " ", $o->getName(), " age=", $o->getAge(), "\n";
}
print "\n";

$db->commit();

$db->startVersionAccess();

# search for a person with specified name and age
print "Enter a name (case sensitive): ";
my $inputName = <STDIN>;
chomp $inputName;

# Lookup method must be used to retrieve a unique single object based on
# an indexed attribute.
# The method returns the object if a single match was successfully found.
# Returns 0 to represent no match. Generates error for multiple matches.
my $found = Person::lookupName($db, $inputName);
if ($found == 0){
    print "No match in $dbName\n";
}
else {
    print("Found ", $found->getName(), " age=", $found->getAge());
}

# To retrieve multiple objects, use iterator with specific criteria.
# An index can specify up to 4 criteria
# The following example retrieves all names between Don and Steve.

my $fromName = "Don";
my $toName = "Steve";
print "Index from \'$fromName\' to \'$toName\': ";

my $pIter = Person::nameIterator($db, $fromName, $toName);
while ($pIter->hasNext()){
    my $o = $pIter->next();
    print " ", $o->getName();
}

```

```
print "\n\n";

$db->endVersionAccess();

# start read-write access again to remove objects
$db->startTransaction();
$p1->remove();
$p2->remove();
$p3->remove();
$p4->remove();
$db->commit();
$db->unselect();
$db->close();

__END__
```

7 Working with Entry-Point Dictionaries

Entry-Point Dictionary is useful in searching a specific string in a database. In the following example, `commentDict` entry-point dictionary is used to count the number of `Person` objects that contain specified string in the `comments` field.

Running EPDict Example

Running `EPDict.pl` requires schema created by `test.odl` in the test directory. See [Before Running Examples](#) to load `test.odl` and generate class files.

To run `EPDict.pl`:

```
perl examples/EPDict.pl <host> <database>
```

EPDict.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/EPDict.pl <host> <database>
#
# This sample program demonstrates how to use entry-point dictionary function.
# It searches specific string in an attribute, and retrieves the objects where
# the string was found. Requires a database with a schema created by test.odl

use Matisse;
use Person;

unless (scalar(@ARGV) == 2) {die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();
$db->select();
$db->startTransaction();

# delete all existing Person objects
my $stmt = new Matisse::MtStatement(database=>$db);
$stmt->executeQuery("DELETE FROM Person");
$stmt->close();

# create four Employee objects with names and comments
my $e1 = new Employee();
$e1->setName("Carmen");
$e1->setComment("Software engineer, C, C++, Java");
my $e2 = new Employee();
$e2->setNeme("Don");
```

```

$e2->setComment("Database administrator, C, Perl");
my $e3 = new Employee();
$e3->setName("Fred");
$e3->setComment("Software engineer, GUI developer, Java, Java Script);
my $e4 = new Employee();
$e4->setName("Steve");
$e4->setComment("Software engineer, C, Smalltalk, Eiffel, Python);

print Employee::getInstanceNumber($db), " Employee(s) available in $dbName\n";
my $iter = Employee::instanceIterator($db);
while ($iter->hasNext())
{
    my $o = $iter->next();
    print $o->getName(), ": ", $o->getComment(), "\n";
}
print "\n";

$db->commit();

$db->startVersionAccess();

# Search for Employees with any words in the 'comment' attribute.
# For example, enter 'Java' to search Java programmers
print "Enter a string to search in '\comment\' (case sensitive): ";
my $searchString = <STDIN>;
chomp $searchString;
my $hits = 0;

print "Looking for $searchString\n";

# Open an iterator on the matching person objects
my $pIter = Person::commentDictIterator($db, $searchString);
while ($pIter->hasNext())
{
    my $o = $pIter->next();
    print " ", $o->getName();
    $hits++;
}

print "$dbName: $hits comment fields contained \"$searchString\"\n\n";
$db->endVersionAccess();

$db->startTransaction();
$e1->remove();
$e2->remove();
$e3->remove();
$e4->remove();
$db->commit();
$db->unselect();
$db->close();

__END__

```

8 Using SQL Statement in Perl Code

Unlike traditional object databases, Matisse is powerful in sense that you can use SQL statements for query. The following example describes how to use SQL statements in a Perl program.

Note: Matisse does not support Perl::DBI module.

Running queryBySQL Example

Running queryBySQL.pl requires schema created by test.odl in the test directory. See [Before Running Examples](#) to load test.odl and generate class files.

To run queryBySQL.pl:

```
perl examples/queryBySQL.pl <host> <database>
```

queryBySQL.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/queryBySQL.pl <host> <database>
#
# This sample program demonstrates how to use SQL statements in a Perl program

use Matisse;
use Person;
use Employee;
use Manager;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();           # establish the connection
$db->select();         # select as current database
$db->startTransaction(); # open a transaction context

# create one Manager and three Employee objects
my $m1 = new Manager();
$m1->setName("Heather");
$m1->setAge(40);
$m1->setId(500);
$m1->setTitle("Director");

my $e1 = new Employee();
$e1->setName("Amy");
```

```

$e1->setAge(25);
$e1->setId(130);

my $e2 = new Employee();
$e2->setName("Bob");
$e2->setAge(24);
$e2->setId(131);

my $e3 = new Employee();
$e3->setName("Michael");
$e3->setAge(41);
$e3->setId(132);

# to query using SQL, create a statement object first
my $stmt = new Matisse::MtStatement(database=>$db);

# query all class names in the database, store the results in $results
# which is a MtResultSet object
my $results = $stmt->executeQuery("SELECT MtName FROM MtClass");
# print the content of results
printClassNames($results);

# there are more than one way to print class names. This example
# uses getValue() method. getString() also works.
sub printClassNames{
    my $res = shift;
    while ($res->next()){
        my $val = $res->getValue(1);
        print $val->value(), "\n";
        # print $res->getString(1), "\n";
    }
    print "\n";
}

# query all names in Employee class
my $stmtA = new Matisse::MtStatement(database=>$db);
$results = $stmtA->executeQuery("SELECT Ref(Employee) FROM Employee");
printObjectInfo($results);

# print 'name' attribute value for the retrieved results. For each
# result, get the object or a reference, and then, get the
# specific attribute values.
sub printObjectInfo{
    my $res = shift;
    while ($res->next()) {
        my $p = $res->getObject(1);
        print $p->getMtClass()->getMtName(), " ", $p->getName(), "\n";
    }
    print "\n";
}

# count objects using COUNT
my $stmtB = new Matisse::MtStatement(database=>$db);
$results = $stmtB->executeQuery("SELECT COUNT(*) FROM Employee");
print "Querying all Employees including Managers\n";
printCountResults($results);
# The above SQL results will return the number of Employees and
# Managers which extend Employee

```

```

# adding ONLY to the SQL statement will eliminate Manager from the count
my $stmtC = new Matisse::MtStatement(database=>$db);
$results = $stmtC->executeQuery("SELECT COUNT(*) FROM ONLY Employee");
print "Querying Employees only, excluding Managers\n";
printCountResults($results);

# print the COUNT results
sub printCountResults{
    my $res = shift;
    while ($res->next()){
        my $val = $res->getValue(1); # returns MtValue
        print "Results for COUNT: ", $val->value(), "\n";
    }
    print "\n";
}

$results->close();
$stmt->close(); $stmtA->close(); $stmtB->close(); $stmtC->close();
$m1->remove(); $e1->remove(); $e2->remove(); $e3->remove();

$db->commit();
$db->unselect();
$db->close();

__END__

```

9 Working with Versions

For an introduction to Matisse version access and named versions, see *Getting Started with Matisse*. This example is a very simple demonstration of version access using `test.odl` schema in the test directory. If you have not loaded `test.odl`, initialize a database, and load the odl file, and generate a class file for each class. Should you need the instructions, see [Before Running Examples](#).

The run `versions.pl`, type

```
perl examples/versions.pl <host> <database>
```

The program demonstrates how to execute named commit and how to access specific version of database. You can view a list of versions in `mt_emgr`, by selecting the sub-node 'Database Snapshots' of your online database.

versions.pl Source

```
# Copyright (c) 1992-2006 Matisse Software Inc.
# This file (in both binary and source code form) is subject to the
# Supplemental License Terms governing use, modification and
# redistribution of Sample Code accompanying the Matisse Developers'
# Release version 8
#
# Syntax:
# perl examples/versions.pl <host> <database>
#
# This sample program shows how to execute named commit with version name
# and how to access a specific version of the database.

use Matisse;
use Person;
use Employee;

unless (scalar(@ARGV) == 2) { die "Usage: $0 <host> <database>"; }
my ($hostName, $dbName) = @ARGV;

my $db = new Matisse::MtDatabase(host=>$hostName, database=>$dbName);

$db->open();
$db->select();
$db->startTransaction();

# create one Employee object
my $e1 = new Employee();
$e1->setName("Becky");
$e1->setAge(41);
$e1->setId(401);

# show the created object
print "\n", $e1->getName(), " id=", $e1->getId(), " commit as ver1\n";

# do named comit (ver1)
my $versionName1 = $db->commit('ver1');

# change an attribute value, show the change, and then do named commit (ver2)
```

```

$db->startTransaction();
$e1->setId(402);
print $e1->getName(), " id=", $e1->getId(), " commit as ver2\n";
my $versionName2 = $db->commit('ver2');

# change the id again, add another object, show the changes, do named commit as ver3
$db->startTransaction();
$e1->setId(403);
print $e1->getName(), " id=", $e1->getId(), " commit as ver3\n";
my $e2 = new Employee();
$e2->setName("Jennifer");
$e2->setAge(31);
$e2->setId(301);
print $e2->getName(), " id=", $e2->getId(), " commit as ver3\n\n";
my $versionName3 = $db->commit('ver3');

# get an attribute value for a particular object from a particular version
$db->startVersionAccess($versionName1);
print "Accessing ", $versionName1, ": ", $e1->getName(), " id=", $e1->getId(), "\n\n";
$db->endVersionAccess($versionName1);

# list all objects in a particular version
$db->startVersionAccess($versionName2);
print "Accessing ", $versionName2, "\n";
print listEmployeeObjects(), "\n";
$db->endVersionAccess($versionName2);

$db->startVersionAccess($versionName3);
print "Accessing ", $versionName3, "\n";
print listEmployeeObjects(), "\n";
$db->endVersionAccess($versionName3);

$db->startTransaction();
$e1->remove();
$e2->remove();
$db->commit();

$db->unselect();
$db->close();

# list name and id of all employees
sub listEmployeeObjects {
    print Employee::getInstanceNumber($db), " Employee(s) in the database\n";
    my $iter = Employee::instanceIterator($db);
    while ($iter->hasNext())
    {
        my $o = $iter->next();
        print " ", $o->getName(), " id=", $o->getId(), "\n";
    }
    print "\n";
}

__END__

```

Appendix A: Generated Public Methods

The following methods are generated automatically in the .pm class files generated by .

For schema classes

The following methods are created for each schema class. These are class methods that are applied to the class as a whole, not to individual instances of the class. These examples are taken from Person.pm.

```
Count instances  getInstanceNumber($db) # $db: MtDatabase object

Open an iterator  instanceIterator($db)

Sample constructor  new Person()

Get descriptor    getClass($db)      # Returns MtClass object
```

For all attributes

The following methods are created for each attribute. For example, if the ODL definition for class Person contains the attributes name and age, the Person.pm file will contain the methods getName and getAge. These examples are taken from name attribute in Person.pm.

```
Get value  getName()

Set value  setName("myName")

Remove value  removeName()  # cannot remove attribute if not Nullable
```

For list-type attributes only

The following methods are created for each list-type attribute. These examples are from Person.record.

```
Get elements  getRecordElements($elements, $length, $offset)

Set elements  setRecordElements($elements, $length, $offset, $discardAfter)

Count elements  getRecordSize()
```

For all relationships

The following methods are created for each relationship. These examples are from Manager.staff.

```
Clear successors  clearStaff()
```

For relationships where the maximum cardinality is 1

The following methods are created for each relationship with a maximum cardinality of 1.

```
Get successor  getSpouse()
```

Set successor `setSpouse($p) # $p: Person object`

For relationships where the maximum cardinality is greater than 1

The following methods are created for each relationship with a maximum cardinality greater than 1. These examples are from `Manager.pm`, which includes `staff` relationship pointed to `Employee` class.

Get successors `getStaff()`

Open an iterator `staffIterator()`

Count successors `getStaffSize()`

Set successors `setStaff($succ) # $succ: MtObjectArray object`

Add successors Insert one successor before any existing successors:
 `prependStaff($emp) # $emp: Employee object`

 Add one successor after any existing successors:
 `appendStaff($emp)`

**Remove
successors** `removeStaff($emp)`

For indexes

The following methods are created for every index defined for a database. These examples are for the only index defined in the example, `Person::personName`.

Lookup `Person::lookupName($db, $searchName)`

Open an iterator `Person::nameIterator($db, $fromName, $toName)`

Get descriptor `Person::getNameIndex($db)`

Returns an `MtIndex` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

For entry-point dictionaries

The following methods are created for every entry-point dictionary defined for a database. These examples are for the only dictionary defined in the example, `Person::commentDict`.

Lookup `lookupCommentDict($db, $searchString)`

Open an iterator `commentDictIterator($db, $value)`

Get descriptor `getCommentDictDictionary($db)`

Returns an `MtEntryPointDictionary` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema