Matisse® Objective-C Programmer's Guide

January 2017

Matisse Objective-C Programmer's Guide

PDF generated 7 January 2017

Contents

1 Introduction 5
Scope of This Document 5
Before Reading This Document 5
Additional Documentation for the Objective-C Binding 5

2 Instructions for Objective-C Examples 6
Before Running the Examples 6
Compiling the Examples 6
Generating Class Documentation 7

3 Connection and Transaction 8
Building the Examples 8
Read Write Transaction 8
Read-Only Access 9
Version Access 9
Other Options 11

4 Working with Objects 13
Running ObjectsExample 13
Creating Objects 13
Listing Objects 14
Deleting Objects 14
Comparing Objects 15

5 Working with Values 16
Running ValuesExample 16
Setting and Getting Values 16
Removing Values 17
Streaming Values 17

6 Working with Relationships 19
Running RelationshipsExample 19
Setting and Getting Relationship Elements 19
Adding and Removing Relationship Elements 20
Listing Relationship Elements 21
Counting Relationship Elements 21

7 Working with Indexes 22
Running IndexExample 22
Index Lookup 22
Index Lookup Count 23
Index Entries Count 23

8 Working with Entry-Point Dictionaries 24
Running EPDictExample 24

Entry-Point Dictionary Lookup 24
Entry-Point Dictionary Lookup Count 24

9 Working with SQL 25
Running SQLExample 25
Retrieving Values 25
Retrieving Objects from a SELECT statement 27

10 Optimization 29
Running LoadExample 29
Creating Multiple Objects 29
Other Operations on Multiple Objects. 31
Eliminating Instances from the Client Cache 31
Error Handling 31

11 Handling Namespaces 33
Connection with Factory 33

12 Working with Database Events 35
Running EventsExample 35
Events Subscription 35
Events Notification 36
More about MtEvent 37

13 Building your Application 38
Discovering the Matisse Objective-C Classes 38
Generating Stub Classes 38
Extending the generated Stub Classes 39
Compiling the application 39
Running the application 40

Appendix A: Example Schema 41

Appendix B: Generated Methods 43

1 Introduction

Scope of This Document

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

Aspects of Matisse programming that the Objective-C 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 Objective-C programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

Additional Documentation for the Objective-C Binding

Getting Started with Matisse and the sample code and example applications discussed in this document are available for download at:

```
http://www.matisse.com/developers/documentation/
```

The HTML-format *Matisse Objective-C Binding API Reference* is installed with Matisse at:

```
%MATISSE_HOME%/docs/objc/api/index.html
```

2 Instructions for Objective-C Examples

Before Running the Examples

Before running this and the following examples, you must do the following:

- Install Matisse.
- Install Xcode to get the Objective-C compiler. Makefiles are provided for use with GNU GCC.
- Set the `MATISSE_HOME` environment variable to the top-level directory of the Matisse installation.
- Download and extract the Objective-C sample code from the Matisse Web site:

  ```
  http://www.matisse.com/developers/documentation/
  ```

  The sample code files are grouped in subdirectories by chapter number. For example, the code snippets from the following chapter are in the `chap_3` directory.
- Create and initialize a database. You can simply start the Matisse Enterprise Manager, select the database 'example' and right click on 'Re-Initialize'.
- From a Unix shell prompt, change to the `chap_x` subdirectory in the directory where you installed the examples.
- If applicable, load the ODL file into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema'. For example you may import `chaps_4_5/objects.odl` for the Chapter 4 demo.
- Generate Objective-C class files.

  ```
  mt_sdl stubgen --lang objC --sn examples.objc_examples.chaps_4_5 -f objects.odl
  ```

Compiling the Examples

A makefile is provided which can build the supplied applications from a command line tool. Each makefile will compile all sources and link all applications in the directory.

- With GNU GCC, run:

  ```
  make
  ```

These makefiles will not set up any databases needed by the applications, but will generate any source required from the ODL file.

> **NOTE:** There is no Xcode project provided for each example, but you can easily create a Command Line Tool project for each example and list the Objective-C example files as well as the `matisseObjC.m` file installed in the `$MATISSE_HOME/include/objc` directory.

Generating Class Documentation

You can generate an API reference for a set of generated Objective-C classes with doxygen, the open-source tool used to generate the Matisse Objective-C binding API documentation.

3 Connection and Transaction

All interaction between client Objective-C 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 Objective-C application may interact with the database using the schema-specific methods generated by `mt_sdl` (see *Generated Methods* on page 43). The following sample applications show a variety of ways of connecting with a Matisse database.

Note that in this chapter there is no ODL file as you do not need to create an application schema.

Building the Examples

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chap_3` directory in your installation (under `objc_examples`).
3. Build the application.

Read Write Transaction

The following code extracted from `chap_3/Connect.m` connects to a database, starts and commits a transaction, and closes the connection:

```
@try {
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname];

    [db open];

    [db startTransaction : MT_MAX_TRAN_PRIORITY];

    printf("Successful connection and open transaction to %s\n",
       [[db description] UTF8String]);

    [db commit];

    [db close];

  }
  @catch (MtException *e) {
    printf("ERROR: %s: (code=%d) %s\n",
       [[e name] UTF8String],
       [e errorCode],
       [[e message] UTF8String] );
    NSLog(@"Stack trace: %@\n", [e callStackSymbols]);
  }
  @finally {
    [db release];
    [hostname release];
    [dbname release];
  }
```

Read-Only Access

The following code extracted from `chap_3/VersionConnect.m` connects to a database in read-only mode, suitable for reports:

```
@try {
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname];

    [db open];

    [db startVersionAccess];

    printf("Successful connection and version access to %s\n",
       [[db description] UTF8String]);

    [db endVersionAccess];

    [db close];

  }
  @catch (MtException *e) {
    printf("ERROR: %s: (code=%d) %s\n",
       [[e name] UTF8String],
       [e errorCode],
       [[e message] UTF8String] );
    NSLog(@"Stack trace: %@\n", [e callStackSymbols]);
  }
  @finally {
    [db release];
    [hostname release];
    [dbname release];
  }
```

Version Access

The following code extracted from `chap_3/VersionNavigation.m` illustrates methods of accessing various versions of a database.

```
// simple function which opens an iterator on the list of version names
void listVersions(MtDatabase* db, NSString *msg)
{
  int i;
  NSString *vername;
  MtVersionEnumerator* viter = [db versionEnumerator];

  printf("%s\n%s has the following versions:\n",
     [msg UTF8String],
     [[db description] UTF8String]);
  i = 0;
  while ((vername = [viter nextObject]) != nil) {
    printf("%d: %s\n", (i+1), [vername UTF8String]);
    i++;
  }
  printf("%d named version(s)\n", i);
```

```
  // close when done.
  [viter close];
}


  @try {
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname];

    [db open];

    [db startTransaction : MT_MAX_TRAN_PRIORITY];

    listVersions(db, @"Versions before regular commit:");

    [db commit];

    [db startTransaction : MT_MIN_TRAN_PRIORITY];

    listVersions(db, @"Versions after regular commit:");

    versionname = [db commit: @"today_"];

    printf("Commit and preserve version named %s\n",
       [versionname UTF8String]);

    [db startVersionAccess : versionname];

    listVersions(db, @"Versions after named commit:");

    printf("Version access to %s\n",
       [versionname UTF8String]);

    [db endVersionAccess];

    [db startTransaction : MT_MIN_TRAN_PRIORITY];
    printf("Delete version named %s\n",
       [versionname UTF8String]);
    [db removeVersionAccess : versionname];


    listVersions(db, @"Versions after remove version:");

    [db commit];

    [db close];

  }
  @catch (MtException *e) {
    printf("ERROR: %s: (code=%d) %s\n",
       [[e name] UTF8String],
       [e errorCode],
       [[e message] UTF8String] );
    NSLog(@"Stack trace: %@\n", [e callStackSymbols]);
  }
  @finally {
    [db release];
    [hostname release];
    [dbname release];
  }
```

Other Options

This source code extracted from `chap_3/VersionNavigation.m` shows how to set or read various connection options and states.

```
BOOL isReadOnly(MtDatabase* db)
{
  // see enum MtDataAccessMode in matisseCtx.h
  return ([db option: MT_DATA_ACCESS_MODE] == MT_DATA_READONLY);
}

void printState(MtDatabase* db)
{
  if (! [db isConnectionOpen]) {
    printMsg(db, @"not connected");
  } else {
    if ([db isTransactionInProgress]) {
      printMsg(db, @"read-write transaction underway");
    } else if ([db isVersionAccessInProgress]) {
      printMsg(db, @"read-only version access underway");
    } else {
      printMsg(db, @"no transaction underway");
    }
  }

  if (isReadOnly(db)) {
    printMsg(db, @"data access is read-only");
  } else {
    printMsg(db, @"data access is read-write");
  }
}


  accessMode = MT_DATA_DEFINITION;

  @try {
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname];

    [db setOption: MT_DATA_ACCESS_MODE : accessMode];

    [db open : username password: passwd];

    if (isReadOnly(db)) {
      [db startVersionAccess];
    } else {
      [db startTransaction : MT_MAX_TRAN_PRIORITY];
    }

    printState(db);

    if (isReadOnly(db)) {
      [db endVersionAccess];
    } else {
      [db rollback];
    }

    [db close];
```

```
}
@catch (MtException *e) {
  printf("ERROR: %s: (code=%d) %s\n",
     [[e name] UTF8String],
     [e errorCode],
     [[e message] UTF8String] );
  NSLog(@"Stack trace: %@\n", [e callStackSymbols]);
}
@finally {
  [db release];
  [hostname release];
  [dbname release];
}
```

4 Working with Objects

Running ObjectsExample

This sample program creates two objects (one `Person` and one `Employee`), 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. Note that because `FirstName` and `LastName` are not nullable, they *must* be set when creating an object.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chaps_4_5` directory in your installation (under `objc_examples`).
3. Load `objects.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `objects.odl`.
4. Generate Objective-C class files.

   ```
   mt_sdl stubgen --lang objC --sn examples.objc_examples.chaps_4_5 -f objects.odl
   ```

5. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).
6. Run the application:

   ```
   ObjectsExample host database
   ```

Creating Objects

This section illustrates the creation of objects. The stubclass provides a default constructor which is the base constructor for creating persistent objects.

```
[db startTransaction : MT_MAX_TRAN_PRIORITY];

 // create a new Person
Person* p = [[Person alloc] init: db];
// modify attributes
[p setFirstName : @"John"];
[p setLastName : @"Smith"];
[p setAge : 42];

// create a new Employee
Employee* e = [[Employee alloc] init: db];
// set attributes
[e setFirstName : @"Jane"];
[e setLastName: @"Jones"];

[db commit];
```

Listing Objects

This section illustrates the enumeration of objects from a class. The `instanceEnumerator()` static method defined on a generated stubclass allows you to enumerate the instances of this class and its subclasses. The `instanceNumber()` method returns the number of instances of this class.

```
// list all Persons
printf("\nEnumerating objects...\n");
printf("%d Persons in the database %s\n", [Person instanceNumber : db],
   [[db description] UTF8String]);

// open an iterator on all the instances of Person and all
// subclasses; if you wish to exclude subclasses, use
// [Person ownInstanceEnumerator] instead
Person* x;
MtObjectEnumerator* piter = [Person instanceEnumerator:db];
while ((x = (Person*)[piter nextObject]) != nil) {
  // show attributes and name of class
  printf("%s %s is a %s\n",
      [[x firstName] UTF8String],
      [[x lastName] UTF8String],
      [[[x getMtClass ] mtFullName] UTF8String]);
}
[piter close];
```

Deleting Objects

This section illustrates the removal of objects. The remove() method delete an object.

```
[db startTransaction : MT_MAX_TRAN_PRIORITY];
printf("\nRemoving created objects...\n");
 // NOTE: does not remove the object sub-parts
 [p remove];
 [e remove];

[db commit];
```

To remove an object and its sub-parts, you need to override the `-[MtObject deepRemove]` method in the stubclass to meet your application needs. For example the implementation of `deepRemove` in the `Person` class that contains a reference to a `PostalAddress` object is as follows:

```
- (void)deepRemove {
  if ([self address]) {
    [[self address] deepRemove];
  }
  [super deepRemove];
}
```

```
Person *p;
...
 [p deepRemove];
```

The `removeAllInstances` method defined on `MtClass` delete all the instances of a class.

```
[[Person getClass:db] removeAllInstances];
```

Comparing Objects

This section illustrates how to compare objects. Persistent objects must be compared with the `isEqual` method. You can't compare persistent object with the == operator.

```
Person *p1;
Person *p2;
...
if ([p1 isEquals:p2])
    printf("Same objects");
```

5 Working with Values

Running ValuesExample

This example is generated by the makefile used in `ObjectsExample`, and it uses the same database. It creates an object, manipulates its values in various ways as described in the source-code comments, imports the data in the file `smiley.gif` to an attribute value, creates a new file from the stored data, then removes the object.

To launch the application:

```
ValuesExample host database
```

Setting and Getting Values

This section illustrates the set, update and read object property values. The stubclass provides a set and a get method for each property defined in the class.

```
// create a new Employee
Employee* e = [[Employee alloc] init: db];
// setting string attributes
[e setComment: @"setting values"];
[e setFirstName: @"John"];
[e setLastName: @"Jones"];

// setting numbers
[e setAge: 42];

// setting Date
NSDate *ts = [NSDate dateWithString : @"2012-05-06 00:00:00 +0000" ];
[e setHireDate : ts];

// setting Numeric
NSDecimalNumber* num = [NSDecimalNumber decimalNumberWithString : @"2000.50"];
[e setSalary: num];

// getting values
printf("%s\n", [[e comment] UTF8String]);
printf("Employee: %s %s\n",
    [[e firstName] UTF8String],
    [[e lastName] UTF8String]);

// suppress output if no value set
// use generated isAgeNull method to check if value is null
printf("Age: ");
if (! [e isAgeNull]) {
  printf(" %d years old\n", [e age]);
} else {
  printf("age not set\n");
}
printf("Number of Dependents: %d (default value: %s)\n",
```

```
        [e dependents],
        [e isDependentsDefaultValue] ? "Yes" : "No");

    printf("Salary: %s\n", [[[e salary] description] UTF8String]);
    printf("Hired on: %s\n", [[[e hireDate] description] UTF8String]);


    // changing values (getting and setting)
    [e setDependents: ([e dependents] + 2)];
```

Removing Values

This section illustrates the removal of object property values. Removing the value of an attribute will return the attribute to its default value.

```
    Employee *e;

    // Removing value returns attribute to default
    [e removeAge];
```

Streaming Values

This section illustrates the streaming of blob-type values (`MT_BYTES`, `MT_AUDIO`, `MT_IMAGE`, `MT_VIDEO`). The stubclass provides streaming methods (`setPhotoElements`, `photoElements`) for each blob-type property defined in the class. It also provides a method (`photoSize`) to retrieve the blob size without reading it.

```
   // setting blob
   NSFileHandle *fdr = [NSFileHandle fileHandleForReadingAtPath: @"matisse.gif"];
    MtSize maxbufsiz = 1024; // samll buffer size for demo

   MtSize numelts = [p photoSize];

   [p3 setPhotoElements: nil : 0 : MT_BEGIN_OFFSET : YES];
    NSData* databuffer = [fdr readDataOfLength: maxbufsiz];
    while ([databuffer length] > 0) {
      [p3 setPhotoElements: databuffer : (unsigned int)[databuffer length] :
MT_CURRENT_OFFSET : YES];
      databuffer = [fdr readDataOfLength: maxbufsiz];
    }
    [fdr closeFile];

    NSString* filename = @"out3.gif";
    NSFileManager *filemgr;
    filemgr = [NSFileManager defaultManager];
    if ([filemgr fileExistsAtPath: filename ] == YES) {
      if ([filemgr removeItemAtPath: filename error: NULL]  == YES) {
        NSLog (@"Removed %s file", [filename UTF8String]);
      }
    }
    [filemgr createFileAtPath: filename contents: nil attributes: nil];
```

```
MtSize bufsiz = 1024; // samll buffer size for demo
MtSize imgsiz = 0;

NSMutableData* valimagebuffer3 = [NSMutableData dataWithLength : bufsiz];

NSFileHandle *file;
file = [NSFileHandle fileHandleForWritingAtPath: filename];

[file seekToFileOffset: 0];
numelts = [p photoElements: valimagebuffer3 : 0 : MT_BEGIN_OFFSET];
do {
  numelts = [p photoElements: valimagebuffer3 : bufsiz : MT_CURRENT_OFFSET];
  if (numelts < bufsiz) [valimagebuffer3 setLength : numelts];
  [file writeData: valimagebuffer3];
  imgsiz += numelts;
} while (numelts == bufsiz);

[file closeFile];
printf("save buffers for a total of %d elements into out3.gif\n", imgsiz);
```

6 Working with Relationships

Running RelationshipsExample

This example creates several objects, manipulates the relationships among them in various ways as described in the source-code comments, then removes the objects.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chaps_6_7_8` directory in your installation (under `objc_examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `examples.odl`.
4. Generate Objective-C class files.

   ```
   mt_sdl stubgen --lang objC --sn examples.objc_examples.chaps_6_7_8 -f examples.odl
   ```

5. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).
6. Launch the application:

   `RelationshipsExample` *`host database`*

Setting and Getting Relationship Elements

This section illustrates the set, update and get object relationship values. The stubclass provides a set and a get method for each relationship defined in the class.

```
// create a Manager
Manager* m1 = [[Manager alloc] init:db];

// set the successor object for reportsTo (in this case it
// refers to itself, i.e., this manager is the big boss at
// the top of the reporting hierarchy)
[m1 setReportsTo:m1];

// create another Manager
Manager* m2 = [[Manager alloc] init:db];

[m2 setReportsTo:m1]; // the manager

// create a new Employee
Employee* e = [[Employee alloc] init: db];

[e setReportsTo:m2];

// specify an assistant for each of the two managers
[m1 setAssistant:e];
[m2 setAssistant:e];
```

```
printf("successor list:\n");
MtSize i;
MtObjectArray* mgrs = [e assistantOf];
for (i = 0; i < [mgrs count]; i++) {
  Manager* m = (Manager*)[mgrs objectAtIndex:i];
  // show attributes and name of class
  printf("%s is %s's assistant\n",
      [[e firstName] UTF8String],
      [[m firstName] UTF8String]);
}
printf("successor list to array:\n");
for (Manager* mm in [mgrs toArray]) {
  // show attributes and name of class
  printf("%s is %s's assistant\n",
      [[e firstName] UTF8String],
      [[mm firstName] UTF8String]);
}
```

Adding and Removing Relationship Elements

This section illustrates the adding and removing of relationship elements. The stubclass provides a append, a remove and a clear method for each relationship defined in the class.

```
Person* c1 = [[Person alloc] init: db];

Person* c2 = [[Person alloc] init: db];

NSArray* childarray = [NSArray arrayWithObjects:c1, c2, nil];
MtObjectArray* children = [[MtObjectArray alloc] initWithArray:db :childarray];
[m2 setChildren:children];

// the following code shows how to add and remove successors
Person* c3 = [[Person alloc] init: db];

// append to existing list
[m2 appendChildren:c3];

// use of MtObjectArray for specifying a list to remove
NSArray* childarray2 = [NSArray arrayWithObjects:c3, c2, nil];
MtObjectArray* children2 = [[MtObjectArray alloc] initWithArray:db :childarray2];
// remove
[m2 removeNumChildren:children2];
printf("\nRemove two successors:\n");

 // another signature supports a single removal
 //  i.e.    [m2 removeChildren:c2];

// clear all the successors
[m2 clearChildren];
```

Listing Relationship Elements

This section illustrates the listing of relationship elements for one-to-many relationships. The stubclass provides an enumerator method for each one-to-many relationship defined in the class.

```
Manager* x;
MtObjectEnumerator* it = [e assistantOfEnumerator];
while ((x = (Manager*)[it nextObject]) != nil) {
  // show attributes and name of class
  printf("%s is %s's assistant\n",
     [[e firstName] UTF8String],
     [[x firstName] UTF8String]);
}
[it close];
```

Counting Relationship Elements

This section illustrates the counting of relationship elements for one-to-many relationships. The stubclass provides an get size method for each one-to-many relationship defined in the class.

```
printf("Now %s has %d children\n",
   [[m2 firstName] UTF8String], [m2 childrenSize]);

// an alternative to get the relationship size
// but the object oid list is loaded before you can get the count
printf("Now %s has %d children\n",
   [[m2 firstName] UTF8String], [[m2 children] count]);
```

7 Working with Indexes

Running IndexExample

This example is generated by the makefile used in `RelationshipsExample`, and it uses the same database. It first creates some `Person` objects in the database and lists their names; then, using the `PersonName` index, checks whether the database contains an entry for a person matching the specified name; then deletes the objects.

To run the application:

```
IndexExample host database firstName lastName
```

Index Lookup

This section illustrates retrieving objects from an index. The stubclass provides a lookup and a enumerator method for each index defined on the class.

```
// The lookup function must return a Person* to allow for nil
// to represent no match
Person *found = [Person lookupPersonName:db :lastName :firstName];
if (found != nil) {
  printf("found %s %s\n",
      [[found firstName] UTF8String],
      [[found lastName] UTF8String]);
} else {
  printf("Nobody found\n");
}

// instead of searching, open an iterator within the specified
// criteria; an Index can specify upto 4 criteria and this API
// would change accordingly (see examples.odl for specification)
NSString *fromFirstName = @"Fred";
NSString *fromLastName = @"Jones";
NSString *toFirstName = @"John";
NSString *toLastName = @"Murray";

MtObjectEnumerator* it = [Person personNameEnumerator:db :fromLastName :fromFirstName
:toLastName :toFirstName :nil :MT_DIRECT :MT_MAX_PREFETCHING];
while ((x = (Person*)[it nextObject]) != nil) {
  printf("%s %s\n",
      [[x firstName] UTF8String],
      [[x lastName] UTF8String]);
}
[it close];
```

Index Lookup Count

This section illustrates retrieving the object count for a matching index key. The `getObjectNumber`() method is defined on the `MtIndex` class.

```
NSArray* key = [NSArray arrayWithObjects:  fromLastName,  fromFirstName, nil];
unsigned int num = [[Person getPersonNameIndex:db] getObjectNumberWithKeys:key :nil];
printf("\n%d object(s) retrieved\n", num);
```

Index Entries Count

This section illustrates retrieving the number of entries in an index. The `getIndexEntriesNumber`() method is defined on the `MtIndex` class.

```
unsigned int count = [[Person getPersonNameIndex:db] getIndexEntriesNumber];
printf("\n%d entries in the index\n", count);
```

8 Working with Entry-Point Dictionaries

Running EPDictExample

This example is generated by the makefile used in `RelationshipsExample`, and it uses the same database. It first creates some `Person` objects in the database and lists them; then, using the `commentDict` entry-point dictionary, counts the number of objects with `Comments` fields containing the search string passed at the command line; then deletes the objects.

To run the application:

```
EPDictExample host database search_string
```

Entry-Point Dictionary Lookup

This section illustrates retrieving objects from an entry-point dictionary. The stubclass provides access to lookup methods and enumerator methods for each entry-point dictionary defined on the class.

```
Person* x;
 int hits = 0;
 MtObjectEnumerator* piter = [Person commentDictEnumerator:db :searchString :nil
:MT_MAX_PREFETCHING];
 while ((x = (Person*)[piter nextObject]) != nil) {
   // show attributes and name of class
   printf("%s %s says '%s'\n",
      [[x firstName] UTF8String],
      [[x lastName] UTF8String],
      [[x comment] UTF8String]);
   hits++;
 }
 [piter close];
 printf("\n%d comment fields contained '%s' \n", hits, [searchString UTF8String]);
```

Entry-Point Dictionary Lookup Count

This section illustrates retrieving the object count for a matching entry-point key. The `getObjectNumber`() method is defined on the `MtEntryPointDictionary` class.

```
MtSize num = [[Person getCommentDictDictionary:db] getObjectNumber:searchString :nil];
printf("\n%d matching object(s)\n", num);
```

9 Working with SQL

Running SQLExample

This example executes two SQL queries and displays their results.

The first query (`select name, id, boss.name from Employee where id > 2`) uses standard SQL syntax and returns "column" (attribute/relationship) and "row" (object) names and attribute values in the familiar table format.

The second query (`select Ref(Employee), Ref(boss) from Employee where id > 2`) uses Matisse's object extensions and returns object IDs (OIDs), which in turn are used to get the names and values.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chap_9` directory in your installation (under `objc_examples`).
3. Load `sql_eg.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `sql_eg.odl`.
4. Generate Objective-C class files.

   ```
   mt_sdl stubgen --lang objC --sn examples.objc_examples.chap_9 -f sqlschema.odl
   ```

5. Load the sample data into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `sql_eg.sql`. This will make the SQL statement appear in the Query Editor window. Then click 'Execute Query'.
6. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).
7. Launch the application:

   ```
   SQLExample host database
   ```

Retrieving Values

You use the `ResultSet` object, which is returned by the `executeQuery` method, to retrieve values or objects from the database. Use the `next` method combined with the appropriate `getString`, `getInt`, etc. methods to access each row in the result.

The following code demonstrates how to retrieve string and integer values from a `ResultSet` object after executing a `SELECT` statement.

```
dbNamespace = @"examples.objc_examples.chap_9";

// The third argument to MtDatabase init is given so that the connection
// object can find the mapping between the Objective-C class Person and
// the schema class Person defined in the "examples.objc_examples.chap_9"
```

```
    // namespace.
    //
    MtDynamicObjectFactory *factory = [[MtDynamicObjectFactory alloc] init: @"" :
dbNamespace];
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname andFactory: factory];

    [db open];

    [db startVersionAccess];

    // Set the SQL CURRENT_NAMESPACE to 'dbNamespace' so there is
    // no need to use the full qualified names to acces the schema objects
    [db setSqlCurrentNamespace:dbNamespace];

    printf("Querying: %s\n", [[db description] UTF8String]);
    printf("Sql Default Namespace: %s\n", [[db sqlCurrentNamespace] UTF8String]);


    // create a statement, execute some sql and iterate through row/cols
    MtStatement* stmt = [[MtStatement alloc] init:db];
    NSString *query = @"SELECT e.name, e.id, e.boss.name FROM Employee e WHERE e.id > 2";
    printf("Sql Default Namespace: %s\n", [[db sqlCurrentNamespace] UTF8String]);
    printf("Query: %s\n", [query UTF8String]);

    MtResultSet* res = [stmt executeQuery:query];

    unsigned int i;
    // list names and types
    // column index is 1-based
    for (i = 1; i <= [res getColumnCount]; i++) {
      printf("column #%d '%s' type is: %s\n", i,
          [[res getColumnName:i] UTF8String],
          [[res getColumnTypeName:i] UTF8String]);
    }
    // since we know that we asked for string and integer, we can dump
    //  all the results
    // use the next method to move through rows
    while ([res next]) {
      NSString* bnam = [res getString:3];
      BOOL isnull = [res wasNull];
      printf("Employee name: %s, id: %d, boss: %s (%d)\n",
          [[res getString:1] UTF8String],
          [res getInteger:2],
          (!isnull ? [bnam UTF8String]  : "[no boss]"),
          isnull);
    }

    // always remember to close the statement when done
    [stmt close];
   [db endVersionAccess];

    [db close];
```

Retrieving Objects from a SELECT statement

You can retrieve Objective-C objects directly from the database without using the Object-Relational mapping technique. This method eliminates the unnecessary complexity in your application, i.e., O/R mapping layer, and improves your application performance and maintenance.

To retrieve objects, use `REF` in the select-list of the query statement and the `getObject` method returns an object. The following code example shows how to retrieve `Person` objects from a `ResultSet` object.

```
    dbNamespace = @"examples.objc_examples.chap_9";

    // The third argument to MtDatabase init is given so that the connection
    // object can find the mapping between the Objective-C class Person and
    // the schema class Person defined in the "examples.objc_examples.chap_9"
    // namespace.
    //
    MtDynamicObjectFactory *factory = [[MtDynamicObjectFactory alloc] init: @"" :
dbNamespace];
    db = [(MtDatabase *)[MtDatabase alloc] init : hostname : dbname andFactory: factory];

    [db open];

    [db startVersionAccess];

    // Set the SQL CURRENT_NAMESPACE to 'dbNamespace' so there is
    // no need to use the full qualified names to acces the schema objects
    [db setSqlCurrentNamespace:dbNamespace];

    printf("Querying: %s\n", [[db description] UTF8String]);
    printf("Sql Default Namespace: %s\n", [[db sqlCurrentNamespace] UTF8String]);

    // to get a list of objects, use the Ref operator

    MtStatement* stmt2 = [[MtStatement alloc] init:db];
    NSString *query2 = @"SELECT Ref(Employee), Ref(boss) FROM Employee WHERE id > 2;";
    printf("Sql Default Namespace: %s\n", [[db sqlCurrentNamespace] UTF8String]);
    printf("Query: %s\n", [query2 UTF8String]);

    MtResultSet* res2 = [stmt2 executeQuery:query2];

    // list names and types
    // column index is 1-based
    for (i = 1; i <= [res2 getColumnCount]; i++) {
      printf("column #%d '%s' type is: %s\n", i,
          [[res2 getColumnName:i] UTF8String],
          [[res2 getColumnTypeName:i] UTF8String]);
    }
    Employee *e;
    Manager *m;
    // still use the ResultSet, but get the object and coerce to the
    // class we expect
    while ([res2 next]) {
      e = (Employee *)[res2 getMtObject:1];
      m = (Manager *)[res2 getMtObject:2];
      BOOL isnull = [res2 wasNull];
      printf("Employee name: %s, id: %d, boss: %s (%d)\n",
```

```
        [[e name] UTF8String],
        [e id],
        (m ? [[m name] UTF8String] : "[no boss]"),
        isnull);
    }


   [db endVersionAccess];

    [db close];
```

10 Optimization

Running LoadExample

This example creates several objects, manipulates the relationships among them in various ways as described in the source-code comments, then removes the objects.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chap_10` directory in your installation (under `objc_examples`).
3. Load `objects.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `examples.odl`.
4. Generate Objective-C class files.

   ```
   mt_sdl stubgen --lang objC --sn examples.objc_examples.chap_10 -f objects.odl
   ```

5. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).
6. Launch the application:

   ```
   LoadExample host database
   ```

Creating Multiple Objects

When an application needs to create several objects, it is more efficient to have the server allocate multiple objects in one call rather than one at a time inside of a loop. To accomplish this we recommend that you use the preallocate method defined on MtDatabase which provide a substantial performance optimization.

```
    [db startTransaction];

    // Optimize the objects loading
    // Preallocate OIDs so objects can be created in the client workspace
    // without requesting any further information from the server
    [db preallocate:DEFAULT_ALLOCATOR_CNT];
    Employee *e = nil;
    Manager *m = nil;
    NSString* fname;
    NSString* lname;
    NSDate *ts;
    NSDecimalNumber* num;
    PostalAddress *a;

    for (i = 0; i < SAMPLE_OBJECT_CNT; i++) {
      if (!m || (i % 10 == 0)) {
         m = [[Manager alloc] init: db];
         // set attributes
         fname = [[NSString alloc] initWithFormat:((i % 2 == 0) ?  @"Franck%d" :
@"Phil%d"), (i+1)];
```

```
        lname = ((i % 2 == 0) ? @"Willianson" : @"Jefferson");

        [m setFirstName:fname];
        [m setLastName:lname];
        [m setAge:(21 + (i % 44))];
        ts = [NSDate dateWithString:@"2012-06-06 00:00:00 +0000" ];
        [m setHireDate:ts];
        num = [NSDecimalNumber decimalNumberWithString:@"12345.00"];
        [m setSalary:num];

        a = [[PostalAddress alloc] init:db];
        [a setCity:((i % 2 == 0) ? @"Portland" : @"Stuttgart")];
        [a setPostalCode:((i % 2 == 0) ? @"04179" : @"24100")];

        [m setAddress:a];
        printf("%s object created\n", [[m description] UTF8String]);

      }
      // create a new Employee
      e = [[Employee alloc] init:db];
      // set attributes
     fname = [[NSString alloc] initWithFormat:((i % 2 == 0) ?  @"Marty%d" : @"Gerry%d"),
(i+1)];
      lname = ((i % 2 == 0) ? @"Paulson" : @"Jackson");
      [e setFirstName:fname];
      [e setLastName:lname];
      [e setAge:(21 + (i % 44))];
      ts = [NSDate dateWithString:@"2012-06-06 00:00:00 +0000" ];
      [e setHireDate : ts];
      num = [NSDecimalNumber decimalNumberWithString:@"12345.00"];
      [e setSalary: num];

      a = [[PostalAddress alloc] init:db];
      [a setCity:((i % 2 == 0) ? @"Portland" : @"Stuttgart")];
      [a setPostalCode:((i % 2 == 0) ? @"04179" : @"24100")];

      [e setAddress:a];

      [e setReportsTo:m];
      printf("%s object created\n", [[e description] UTF8String]);

      if (i % OBJECT_PER_TRAN_CNT == 0) {
        [db commit];
        [db startTransaction];
      }
      // check the remaining number of preallocated objects.
      if ([db numPreallocated] < 2) {
        [db preallocate:DEFAULT_ALLOCATOR_CNT];
      }
    }

    if ([db isTransactionInProgress]) {
      [db commit];
    }
```

Other Operations on Multiple Objects

Generally speaking, data transferred between client and server is optimized by the client cache to avoid unnecessary round trips. For example, when an object instance is first accessed, all the basic attribute information (not including relationships or streamable attributes) for that instance is transferred to the client as well, and this information will stay in the cache during the transaction. With this in mind, when an application needs to access several objects in succession, it can optimize the data transferred between client and server by preloading all the instances into the client cache with a single server operation, so that each instance access will not require a separate server access. This is accomplished using the `load` method on an `MtObjectArray`. For example, to access all the `Employee` instances which are successors to a particular `Manager` object's `team` relationship:

```
MtObjectArray* team = [x team];
printf("Team of %d members:\n", [team count]);
// load in the client cache all the team members at once
[team load];
for (i = 0; i < [team count]; i++) {
   y = (Employee*)[team objectAtIndex:i];
   printf("    %s %s is a %s\n",
          [[y firstName] UTF8String],
          [[y lastName] UTF8String],
          [[[y getMtClass] mtFullName] UTF8String]);
}
// unload from the client cache all the team members at once
[team unload];
```

As a convenience, there is also an [`MtObjectArray remove`] method which can be used to remove all the instances without having to write a similar loop calling `[MtObject remove]`.

NOTE: The creation of a `MtObjectArray` does not populate the client cache, nor does it create Objective-C object instances. It only retrieves an array of OID (unique object identifiers). The cache and instances are affected only by access.

Eliminating Instances from the Client Cache

`[MtObjectArray unload]` can be used to remove objects from the client cache (both those that were loaded explicitly with `[MtObjectArray load]` and those loaded automatically by access). For example, if a version access (or transaction) is used for continued access to large numbers of objects, it can unload the objects from the Matisse client cache after access is complete.

Error Handling

Example `LoadExample.m` demonstrates how to break a complex update up into a series of short transactions so that any exceptions resulting from invalid data will not roll back valid data.

```
  @try {

    MtDynamicObjectFactory *factory = [[MtDynamicObjectFactory alloc] init:@""
:@"examples.objc_examples.chap_10"];
    db = [(MtDatabase *)[MtDatabase alloc] init:hostname :dbname andFactory:factory];

    [db open];

    [db startTransaction];

    // Optimize the objects loading
    [db preallocate:DEFAULT_ALLOCATOR_CNT];
    Employee *e = nil;
    Manager *m = nil;

    for (i = 0; i < SAMPLE_OBJECT_CNT; i++) {
      if (!m || (i % 10 == 0)) {
        m = [[Manager alloc] init: db];
        ...
      }
      // create a new Employee
      e = [[Employee alloc] init:db];
      ...

      if (i % OBJECT_PER_TRAN_CNT == 0) {
        [db commit];
        [db startTransaction];
      }
      // check the remaining number of preallocated objects.
      if ([db numPreallocated] < 2) {
        [db preallocate:DEFAULT_ALLOCATOR_CNT];
      }
    }

    if ([db isTransactionInProgress]) {
      [db commit];
    }

    [db close];

  }
  @catch (MtException *e) {
    printf("ERROR: %s: (code=%d) %s\n",
        [[e name] UTF8String],
        [e errorCode],
        [[e message] UTF8String] );
    NSLog(@"Stack trace: %@\n", [e callStackSymbols]);
    if ([db isTransactionInProgress]) {
      [db rollback];
    }
    if ([db isConnectionOpen]) {
      [db close];
    }
  }
  @finally {
    [db release];
}
```

11 Handling Namespaces

The `mt_sdl` utility with the `stubgen` command allows you to generate Objective-C source code for the schema classes defined in the ODL file. The `-sn <namespace>` options define the mapping between the schema class namespace and the Objective-C class. When your persistent classes are defined in a specific database namespace, you need to give this information to the `Connection` object so that it can find these classes when returning objects.

For example, to generate the Objective-C classes from the schema classes in defined in the `objc_examples.chap_3` namespace.

```
mt_sdl stubgen --lang objC --sn objc_examples.chap_3 -f examples.odl
```

To generate the Objective-C classes from the schema classes in defined in the root namespace.

```
mt_sdl stubgen --lang objC -f example.odl
```

Connection with Factory

Using MtDynamicObjectFactory

For example, the persistent classes are defined in the `com.company.project.module` database namespace. In this case, you need to pass an `MtDynamicObjectFactory` object as the additional argument for the `MtDatabase` constructor. Assuming that the schema classes are defined in the root namespace:

```
MtDynamicObjectFactory *factory = [[MtDynamicObjectFactory alloc] init:@"" :
@"com.company.project.module"];
MtDatabase *db = [(MtDatabase *)[MtDatabase alloc] init:hostname :dbname andFactory:
factory];
```

Now assuming that the schema classes are defined in the root namespace, you can use the default factory:

```
MtDatabase *db = [[MtDatabase alloc] init:hostname :dbname];
```

Using MtCoreObjectFactory

This factory is the basic MtObject-based object factory. This factory is the most appropriate for application which does use generated stubs. This factory is faster than the default Object Factory used by MtDatabase since it doesn't use reflection to build objects.

```
MtCoreObjectFactory *factory = [[MtCoreObjectFactory alloc] init:];
MtDatabase *db = [(MtDatabase *)[MtDatabase alloc] init:hostname :dbname andFactory:
factory];
```

Implementing the MtObjectFactory protocol

The `MtObjectFactory` protocol describes the mechanism used by `MtDatabase` to create the appropriate Objective-C object for each Matisse object. Implementing the `MtObjectFactory` protocol requires to define the `getObjcClass` method which returns Objective-C class corresponding to a

Matisse Class Name, the `getDatabaseClass` method which returns database class name corresponding to the Objective-C class name and the `getObjectInstance` method which returns a Objective-C object based on an oid.

```
@interface MtCoreObjectFactory : NSObject<MtObjectFactory> {
}

@end

@implementation MyAppFactory

-(Class) getObjcClass: (NSString*) mtClsName {
  Class objcClass = [MtObject class];
  return objcClass;
}

-(MtObject*) getObjectInstance : (MtDatabase*) db : (MtOid) mtOid {
  if (mtOid == 0) {
    return nil;
  }
  return [[MtObject alloc] initWithOid : db : mtOid];
}
-(NSString*) getDatabaseClass: (NSString*) objcClsName {
  return objcClsName;
}

@end
```

12 Working with Database Events

This section illustrates Matisse Event Notification mechanism. The sample application is divided in two sections. The first section is event selection and notification. The second section is event registration and event handling.

Running EventsExample

This example creates several events, then manipulates them to illustrate the Event Notification mechanism.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `events` directory in your installation (under `objc_examples`).
3. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).
4. Launch the application:

```
Note that to run the example, you need to open at least 2 command line windows.

EventsExample localhost example N

EventsExample localhost example S
```

Events Subscription

This section illustrates event registration and event handling. Matisse provides the `MtEvent` class to manage database events. You can subscribe up to 32 events (`MtEvent.EVENT1 to MtEvent.EVENT32`) and then wait for the events to be triggered.

```
 const int TEMPERATURE_CHANGES_EVT = MT_EVENT1;
 const int RAINFALL_CHANGES_EVT = MT_EVENT2;
 const int HIMIDITY_CHANGES_EVT = MT_EVENT3;
 const int WINDSPEED_CHANGES_EVT = MT_EVENT4;

[db open];

 subscriber = [[MtEvent alloc] init:db];

  // Subscribe to all 4 events
MtEvents eventSet = TEMPERATURE_CHANGES_EVT |
  RAINFALL_CHANGES_EVT |
  HIMIDITY_CHANGES_EVT |
  WINDSPEED_CHANGES_EVT;

[subscriber subscribe:eventSet];

MtEvents triggeredEvents;

 // Wait 1000 ms for events to be triggered
```

```
  // return false if not event is triggered until the timeout is reached
  if ([subscriber wait:1090  :&triggeredEvents]) {
   printf("Events (#%d) triggered:\n"\
        "%sChange in temperature\n"\
        "%sChange in rain fall\n"\
        "%sChange in humidity\n"\
        "%sChange in wind speed\n", i,
        (((triggeredEvents & TEMPERATURE_CHANGES_EVT) > 0) ? "" : "No "),
        (((triggeredEvents & RAINFALL_CHANGES_EVT) > 0) ? "" : "No "),
        (((triggeredEvents & HIMIDITY_CHANGES_EVT) > 0) ? "" : "No "),
        (((triggeredEvents & WINDSPEED_CHANGES_EVT) > 0) ? "" : "No "));
  } else {
     printf("No Event received after ~1 sec\n");
  }
  i++;
}

printf("Unsubscribe to 4 Events\n");
// Unsubscribe to all 4 events
[subscriber unsubscribe];

[db close];
```

Events Notification

This section illustrates event selection and notification.

```
const int TEMPERATURE_CHANGES_EVT = MT_EVENT1;
const int RAINFALL_CHANGES_EVT = MT_EVENT2;
const int HIMIDITY_CHANGES_EVT = MT_EVENT3;
const int WINDSPEED_CHANGES_EVT = MT_EVENT4;

db = [(MtDatabase *)[MtDatabase alloc] init:hostname :dbname];

[db open];

notifier = [[MtEvent alloc] init:db];

MtEvents eventSet;

eventSet = 0;
eventSet |= RAINFALL_CHANGES_EVT;
eventSet |= WINDSPEED_CHANGES_EVT;

[notifier notify:eventSet];

[db close];
```

More about MtEvent

As illustrated by the previous sections, the `MtEvent` class provides all the methods for managing database events. The reference documentation for the `MtEvent` class is included in the Matisse Objective-C Binding API documentation located from the Matisse installation root directory in `docs/objc/api/index.html`.

13 Building your Application

This section describes the process for building an application from scratch with the Matisse Objective-C binding.

Discovering the Matisse Objective-C Classes

All the Objective-C binding classes are provided open-source in 2 files: **`matisseObjC.h`** and **`matisseObjC.m`**. located in `$MATISSE_HOME/include/objc`. The core classes defined in the **matisse** sub-directory. These classes manages the database connection, the object factories as well as the objects caching mechanisms. It also includes the Matisse meta-schema classes defined in the **matisse/reflect** sub-directory. It also includes the Matisse SQL implementation defined in the **matisse/sql** sub-directory

Matisse Client Server

The Matisse Objective-C binding is comprised of only one file:

1. **matisseObjC.m** contains all the Objective-C binding classes that links the binding to the **matisse** library.

Matisse Lite

Matisse Lite is the embedded version of Matisse DBMS. Matisse Lite is a compact library that implements the server-less version of Matisse. The Objective-C binding also link to the Matisse Lite library. The Matisse Lite Objective-C binding is comprised of only one file:

1. **matisseObjC.m** contains all the Objective-C binding classes that links the binding to the **matisselite** library.

The Matisse Objective-C API documentation included in the delivery provides a detailed description of all the classes and methods.

> **NOTE:** The Objective-C binding API for Matisse Client Server and for Matisse Lite are totally identical making your application working with either one without any code changes.

Generating Stub Classes

The Objective-C binding relies on object-to-object mapping to access objects from the database. Matisse `mt_sdl` utility allows you to generate the stub classes mapping your database schema classes. Generating Objective-C stub classes is a 2 steps process:

1. Design a database schema using ODL (Object Definition Language).

2. Generate the Objective-C classes from the ODL file:

```
mt_sdl stubgen --lang objC -f myschema.odl
```

A .h and .m files will be created for each class defined in the database. If you need to define these persistent classes from a specific database namespace, use -sn option. The following command generates classes from the namespace `com.company.project`:

```
mt_sdl stubgen --lang objC --sn com.company.project -f myschema.odl
```

When you update your database schema later, load the updated schema into the database. Then, execute the `mt_sdl` utility in the directory where you first generated the class files, to update the files. Your own program codes added to these stub class files will be preserved.

Extending the generated Stub Classes

You can add your own source code outside of the `BEGIN` and `END` markers produced in the generated stub class.

```
// BEGIN Matisse SDL Generated Code
// DO NOT MODIFY UNTIL THE 'END of Matisse SDL Generated Code' MARK BELOW
...
// END of Matisse SDL Generated Code
```

Compiling the application

Matisse Client Server

```
gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2 Person.m -o
Person.o

gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2
$MATISSE_HOME/include/objc/matisseObjC.m -o matisseObjC.o

gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2
ObjectsExample.m -o ObjectsExample.o

gcc -o ObjectsExample Employee.o Person.o PostalAddress.o PersonExtend.o
matisseObjC.o ObjectsExample.o -L$MATISSE_HOME/lib -lmatisse -framework Foundation
```

Matisse Lite

```
gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2 Person.m -o
Person.o

gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2
$MATISSE_HOME/include/objc/matisseObjC.m -o matisseObjC.o

gcc -Wall -c -I$MATISSE_HOME/include/objc -I$MATISSE_HOME/include -O2
ObjectsExample.m -o ObjectsExample.o
```

```
gcc -o ObjectsExample Employee.o Person.o PostalAddress.o PersonExtend.o
matisseObjC.o ObjectsExample.o -L$MATISSE_HOME/lib -lmatisselite -framework
Foundation
```

CAUTION: The order for the -I options is important and must list `$MATISSE_HOME/include/objc` first and then `$MATISSE_HOME/include` so the compiler does not use the header files for the C++ binding located in `$MATISSE_HOME/include`

Running the application

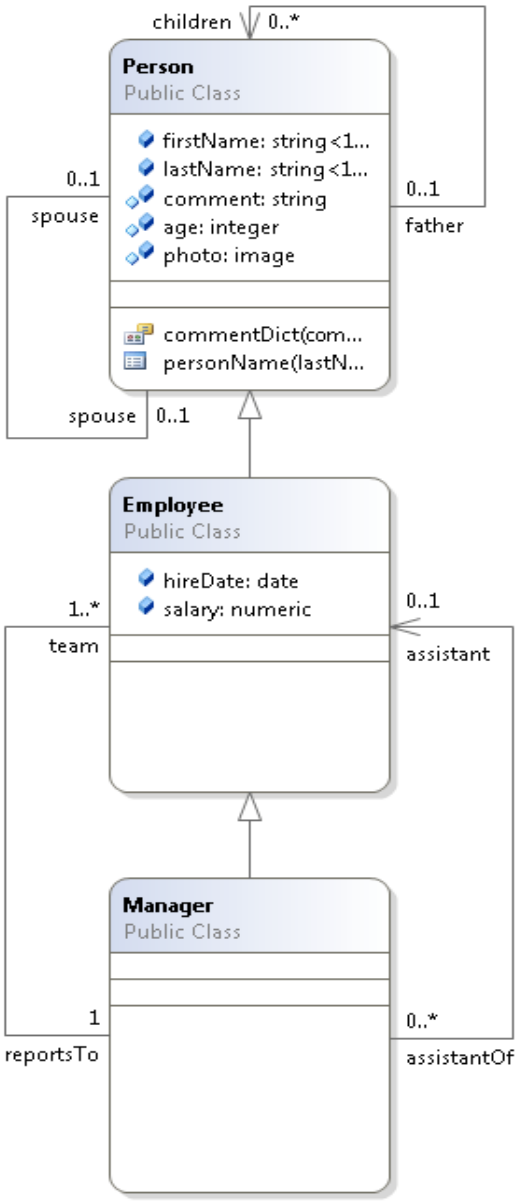
Matisse Client Server

```
ObjectSExample host database
```

Matisse Lite

```
ObjectsExample host database
```

Appendix A: Example Schema




```
module examples {

  module objc_examples {

    module chaps_6_7_8 {

      interface Person : persistent
      {
        attribute String<16> firstName;
        attribute String<16> lastName;
        attribute String Nullable comment;
```

```
        attribute Integer Nullable age;
        attribute Image Nullable photo = NULL;
        relationship Person spouse[0,1] inverse Person::spouse;
        readonly relationship Person father[0,1] inverse Person::children;
        relationship Set<Person> children inverse Person::father;
        mt_index personName
          criteria {person::lastName MT_ASCEND},
        {person::firstName MT_ASCEND};
        mt_entry_point_dictionary commentDict entry_point_of comment
          make_entry_function "make-full-text-entry";
      };

      interface Employee : Person : persistent
      {
        attribute Date hireDate;
        attribute Numeric salary;
        readonly relationship Set<Manager> assistantOf inverse Manager::assistant;
        relationship Manager reportsTo inverse Manager::team;
      };

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

    };
  };
};
```

Appendix B: Generated Methods

The following methods are defined in the Objective-C class files generated by `mt_sdl`. Definitions are in *`class`*`.h`, and other source in *`class`*`.m`.

For schema classes

The following methods are created for each schema class. These are class methods (also called static methods): that is, they apply to the class as a whole, not to individual instances of the class. These examples are taken from `Person`.

Count instances

```
+ (unsigned int)instanceNumber:(MtDatabase *)db;
+ (unsigned int)ownInstanceNumber:(MtDatabase *)db;
```

Open an enumerator

```
+ (MtObjectEnumerator*)instanceEnumerator:(MtDatabase *)db;
+ (MtObjectEnumerator*)ownInstanceEnumerator:(MtDatabase *)db;
```

Sample constructor

```
- (id)init:(MtDatabase *)db;
```

Sample description

```
- (NSString*)description;
```

Get descriptor

```
+ (MtClass *)getClass:(MtDatabase *)db;
```

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

Factory constructor

```
- (id)initWithOid:(MtDatabase *)db :(MtOid) oid;
```

This constructor is inherited from the `MtObject` class is called by `MtObjectFactory` implementation.

For attribute descriptors

The following methods are created for each attribute descriptor. For example, if the ODL definition for class `Check` contains attribute descriptors `Date` and `Amount`, the `Check.h` file will contain the methods `getDate` and `getAmount`. This and following examples are taken from `Person::firstName`.

For all attribute descriptors

Remove value

```
removeFirstName()
```

For scalar (non-list-type) attribute descriptors only

Get value

```
- (NSString*)firstName;
```

Set value

```
- (void)setFirstName:(NSString*)val;
```

Remove value

```
- (void)removeFirstName;
```

Check Null value

```
- (BOOL)isFirstNameNull;
```

Check Default value
```
- (BOOL)isFirstNameDefaultValue;
```

Get descriptor
```
+ (MtAttribute *)getLastNameAttribute: (MtDatabase *)db;
```

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

For list-type attribute descriptors only

The following methods are created for each list-type attribute descriptor. These examples are from `Person::photo`.

Get value
```
- (NSData*)photo;
```

Set value
```
- (void)setPhoto:(NSData*)val;
```

Remove value
```
- (void)removePhoto;
```

Check Null value
```
- (BOOL)isPhotoNull;
```

Check Default value
```
- (BOOL)isPhotoDefaultValue;
```

Get elements
```
- (unsigned int)photoElements:(NSData*)value :(unsigned int)len
    :(unsigned int)offset;
```

Set elements
```
- (void)setPhotoElements:(NSData*)value :(unsigned int)len
    :(unsigned int)offset :(BOOL)discardAfter;
```

Count elements
```
- (unsigned int)photoSize;
```

Get descriptor
```
+ (MtAttribute *)getPhotoAttribute: (MtDatabase *)db;
```

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

For all relationship descriptors

The following methods are created for each relationship descriptor. These examples are from `Person::spouse`.

Clear successors
```
- (void)clearSpouse;
```

Get descriptor
```
+ (MtRelationship *)getSpouseRelationship:(MtDatabase *)db;
```

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

For relationship descriptors where the maximum cardinality is 1

The following methods are created for each relationship descriptor with a maximum cardinality of 1. These examples are from `Manager::assistant`.

Get successor
```
- (Employee *)assistant;
```

Set successor
```
- (void)setAssistant:(Employee *)succ;
```

For relationship descriptors where the maximum cardinality is greater than 1

The following methods are created for each relationship descriptor with a maximum cardinality greater than 1. These examples are from `Manager::team`.

Get successors
```
- (MtObjectArray *)team;
```

Open an enumerator
```
- (MtObjectEnumerator *)teamEnumerator;)
```

Count successors
```
- (unsigned int)teamSize;
```

Set successors
```
- (void)setTeam:(MtObjectArray*)succs;
```

Add successors

Insert one successor before any existing successors:
```
- (void)prependTeam:(Employee *)succ;
```

Add one successor after any existing successors:
```
- (void)appendTeam:(Employee*)succ;
```

Add multiple successors after any existing successors:
```
- (void)appendNumTeam:(MtObjectArray *)succs;
```

Remove successors
```
- (void)removeTeam:(Employee *)succ;
- (void)removeNumTeam:(MtObjectArray *)succs;
```

Remove specified successors.

For index descriptors

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`. The number of attributes in the lookup and enumerator methods is dependent on the number of criteria defined for the index (in this case, two, `lastName` and `firstName`).

Lookup
```
+ (Person *)lookupPersonName:(MtDatabase *)db :(NSString*) lastName :
    (NSString*) firstName;
+ (MtObjectArray *)lookupObjectsPersonName:(MtDatabase *)db
    :(NSString*) lastName : (NSString*) firstName;
```

Open an enumerator
```
+ (MtObjectEnumerator *)personNameEnumerator:(MtDatabase *)db
    :(NSString*) fromLastName :(NSString*) fromFirstName
    :(NSString*) toLastName :(NSString*) toFirstName :(MtClass
    *)filterClass :(MtDirection)direction : (int)numObjPerBuffer;
```

Get descriptor
```
+ (MtIndex *)getPersonNameIndex:(MtDatabase *)db;
```

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

For entry-point descriptors

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

```
+ (Person *)lookupCommentDict:(MtDatabase *)db :(NSString *)value;
```

Open an enumerator

```
+ (MtObjectEnumerator *)commentDictEnumerator:(MtDatabase *)db
    :(NSString *)value :(MtClass *)filterClass
    :(int)numObjPerBuffer;
```

Get descriptor

```
+ (MtEntryPointDictionary *)getCommentDictDictionary:(MtDatabase
    *)db;
```

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