Matisse® C++ Programmer's Guide

January 2017

Matisse C++ Programmer's Guide

Copyright © 2017 Matisse Software, Inc. All Rights Reserved.

This manual and the software described in it are copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without prior written consent of Matisse Software, Inc. This manual and the software described in it are 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)(l)(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 7 January 2017

Contents

1 Introduction

Scope of This Document

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

Aspects of Matisse programming that the 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 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 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 C++ Binding API Reference* is installed with Matisse at:

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

2 Instructions for C++ Examples

Before Running the Examples

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

- Install Matisse.
- Install a C++ compiler. Makefiles are provided for use with GNU GCC, Microsoft Visual C++ and Solaris C++. With other compilers, you will need to create your own makefiles.
- Set the `MATISSE_HOME` environment variable to the top-level directory of the Matisse installation.
- Download and extract the 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 or on MS Windows from a 'Command Prompt' window, 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 C++ class files.

  ```
  mt_sdl stubgen --lang cxx -f objects.odl
  ```

Compiling the Examples

Three sets of makefiles are supplied for several platforms 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:

  ```
  gmake -f Makefile.gcc
  ```
- With Microsoft Visual C++, run:

  ```
  nmake /f Makefile.win32
  ```

  If this fails with the error message, `'cl' is not recognized as an internal or external command`, enable the compiler's command-line option by running the batch file `VCVARSALL.BAT`, which you should find in a Visual C++ directory.

- With Solaris C++, run:

```
make -f Makefile.sun
```

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

Generating Class Documentation

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

3 Connection and Transaction

All interaction between client 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 C++ application may interact with the database using the schema-specific methods generated by `mt_sdl` (see *Generated Methods* on page 38). 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 `cxx_examples`).
3. Build the application.

Read Write Transaction

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

```
try
    {
      MtDatabase db(av[1], av[2]);

      // open, select and start access to the database
      db.open();
      db.startTransaction();

      // read/write access
      std::cout << "Successful connection and open transaction to "
         << db << std::endl;

      db.commit();
      db.close();
    }
  catch (MtException &e)
    {
      std::cerr << e << std::endl;
    }
  catch (...)
    {
      std::cerr << "Unknown exception" << std::endl;
    }
```

Read-Only Access

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

```
try
    {
      MtDatabase db(av[1], av[2]);

      // open, select and start access to the database
      db.open();
      db.startVersionAccess();

      // version connect implies read-only access
      std::cout << "Successful connection and version access to "
         << db << std::endl;

      db.endVersionAccess();
      db.close();
    }
  catch (MtException &e)
    {
      std::cerr << e << std::endl;
    }
  catch (...)
    {
      std::cerr << "Unknown exception" << std::endl;
    }
```

Version Access

The following code extracted from `chap_3/VersionNavigation.cpp` 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)
{
    MtStringIterator viter = db.versionIterator();

    std::cout << db << " has these versions:" << std::endl;
    while (viter.hasNext())
    {
        std::string vname = viter.next();
        std::cout << "\t" << vname << std::endl;
    }

    // close when finished.
    viter.close();
}

} // end anon namespace

try
    {
```

```
      MtDatabase db(av[1], av[2]);

      // open, select and start access to the database
      db.open();

      db.startTransaction();
      std::cout << "\nVersions before regular commit:" << std::endl;
      listVersions(db);
      db.commit();

      db.startTransaction();
      std::cout << "\nVersions after regular commit:" << std::endl;
      listVersions(db);

      // Providing a version name when committing a transaction
      // creates a named version of the database that must be
      // explicitly destroyed. The version name is the string
      // passed in argument (which disginguishes versions created
      // by this client) plus a unique ID number appended by the
      // server (which distinguishes this version from others
      // created by this client).
      std::string vername = db.commit("test");
      std::cout << "\nCommit to version named: " << vername << std::endl;

      db.startVersionAccess();
      std::cout << "\nVersions after named commit:" << std::endl;
      listVersions(db);
      db.endVersionAccess();

      // A saved version can be accessed read-only
      db.startVersionAccess(vername);
      std::cout << "\nSuccessful access within version: " << vername
        << std::endl;
      db.endVersionAccess();

      db.close();
    }
  catch (MtException &e)
    {
      std::cerr << e << std::endl;
    }
  catch (...)
    {
      std::cerr << "Unknown exception" << std::endl;
    }
```

Other Options

This source code extracted from `chap_3/VersionNavigation.cpp` shows how to enable the local client-server memory transport and to set or read various connection options and states.

```
class AdvancedConnect
{
private:
  MtDatabase db;
```

```
public:
  // constructor creates the MtDatabase
  AdvancedConnect(const std::string& host, const std::string& dbname)
    : db(host,dbname)
  {
  }

  void run()
  {
    if (getenv("MT_MEM_TRANS") != NULL)
      db.setOption(::MT_MEMORY_TRANSPORT, 1);

    // possible values are 0=RW, 1=RO, 2=DD (RW + Schema Modification)
    if (getenv("MT_DATA_ACCESS") != NULL)
      {
     int da_opt = atoi(getenv("MT_DATA_ACCESS"));
     db.setOption(::MT_DATA_ACCESS_MODE, da_opt);
      }

    if (getenv("dbuser") != NULL)
      {
     std::string user = getenv("dbuser") ? getenv("dbuser") : "";
     std::string passwd = getenv("dbpasswd") ? getenv("dbpasswd") : "";
     db.open(user, passwd);
      }
    else
      {
     db.open();
      }

    start(isReadOnly());
    printState();

    // do other work here ...

    end();
    db.close();
  }

  void start(bool readonly)
  {
    if (readonly)
      db.startVersionAccess();
    else
      db.startTransaction();
  }

  void end(void)
  {
    if (db.isVersionAccessInProgress())
      db.endVersionAccess();
    else if (db.isTransactionInProgress())
      db.commit();
    else
      std::cerr << "No transaction/version access in progress"
         << std::endl;
  }
```

```
  bool isMemoryTransportOn()
  {
    return (db.getOption(::MT_TRANSPORT_TYPE) == ::MT_MEM_TRANSPORT);
  }

  bool isReadOnly()
  {
    return (db.getOption(::MT_DATA_ACCESS_MODE) == 1);
  }

  void printState()
  {
    if (!db.isConnectionOpen())
      {
     dbmsg("not connected");
      }
    else
      {
     if (db.isTransactionInProgress())
       dbmsg("read-write transaction underway");
     else if (db.isVersionAccessInProgress())
       dbmsg("read-only version access underway");
     else
       dbmsg("no transaction underway");
      }
    std::string msg = "MEMORY_TRANSPORT is ";
    msg += (isMemoryTransportOn() ? "on" : "off");
    dbmsg(msg);
    msg = "Access is ";
    msg += (isReadOnly() ? "readonly" : "readwrite");
    dbmsg(msg);
  }

  void dbmsg(const std::string& msg)
  {
    std::cout << db << ": " << msg << std::endl;
  }
};
```

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 `cxx_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 C++ class files.

   ```
   mt_sdl stubgen --lang cxx -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.

```
// open, select and start access to the database
db.open();
db.startTransaction();

// create a new Person
Person& p = Person::create(db);
// modify attributes
p.setFirstName("John");
p.setLastName("Smith");

// create a new Employee
Employee& e = Employee::create(db);
// set attributes
e.setFirstName("Jane");
e.setLastName("Jones");

db.commit();
```

Listing Objects

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

```
// list all Persons
 std::cout << std::endl << Person::getInstanceNumber(db)
          << " Persons in the database" << std::endl;
 // open an iterator on all the instances of Person and all
 // subclasses; if you wish to exclude subclasses, use
 // Person::ownInstanceIterator() instead
 MtObjectIterator<Person> piter = Person::instanceIterator(db);
 while (piter.hasNext())
 {
   // use object reference
   Person &x = piter.next();
   // show attributes and name of class
   std::cout << "\t" << x.getFirstName() << " " << x.getLastName()
             << " is a " << x.getMtClass().getMtName() << std::endl;
 }
```

Deleting Objects

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

```
db.startTransaction();

 // remove created objects
 std::cout << "\nRemoving created objects" << std::endl;
 p.remove();
 e.remove();

 // list again to show deletion, similar to above
 std::cout << "\nAfter deletion:" << std::endl;
 std::cout << Person::getInstanceNumber(db)
          << " Persons in the database" << std::endl;
 MtObjectIterator<Person> piter2 = Person::instanceIterator(db);
 while (piter2.hasNext())
 {
   Person &x = piter2.next();
   std::cout << "\t" << x.getFirstName() << " " << x.getLastName()
             << " is a " << x.getMtClass().getMtName() << std::endl;
 }

 db.commit();
```

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 `matisse.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::create(db);

    // setting string attributes
    e.setComment("setting values");
    e.setFirstName("John");
    e.setLastName("Jones");

    // setting numbers
    e.setAge(42);

    // setting Date
    matisse::MtTimestamp ts("2002-02-02");
    e.setHireDate(ts);

    // setting Numeric
    matisse::MtNumeric num(2958.33);
    e.setSalary(num);

    // getting
    std::cout << std::endl << e.getComment() << std::endl;
    std::cout << "\tEmployee: " << e.getFirstName() << " "
              << e.getLastName() << std::endl;

    // suppress output if no value set
    // use generated isAgeNull() method to check if value is null
    if (!(e.isAgeNull()))
      std::cout << "\t" << e.getAge() << " years old" << std::endl;
    std::cout << "\tNumber of Dependents: " << e.getDependents()
              << std::endl;
    std::cout << "\tSalary: $" << e.getSalary() << std::endl;
    std::cout << "\tHired on: " << e.getHireDate() << std::endl;

    // changing values (getting and setting)
```

```
e.setDependents(e.getDependents() + 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`(), `getPhotoElements()`) for each blob-type property defined in the class. It also provides a method (`getPhotoSize()`) to retrieve the blob size without reading it.

```
// setting blob
int bufSize = 15;       // small buff size for demo purposes
int actualBytes;
int totalBytes = 0;
// use auto_ptr to clean up when done
std::auto_ptr<char> tmpBytes(new char[bufSize]);

// binary file on disk
std::ifstream is("matisse.gif", std::ios::in | std::ios::binary);
if (!is)
{
  std::cerr << "matisse.gif does not exists" << std::endl;
}
else
{
  // initial call to truncate data
  e.setPhotoElements((::MtByte*)tmpBytes.get(), 0,
                     MT_BEGIN_OFFSET, true);
  while (!is.eof())
  {
    actualBytes = (is.read(tmpBytes.get(), bufSize)).gcount();
    e.setPhotoElements((::MtByte*)tmpBytes.get(), actualBytes,
                       MT_CURRENT_OFFSET, true);
    totalBytes += actualBytes;
  }
  is.close();
}
std::cout << "transferred " << totalBytes
          << " bytes from matisse.gif to object"
          << std::endl;

// read blob and store to file, just reverse the process from above
std::ofstream os("new.gif", std::ios::binary);
```

```
totalBytes = 0;
if (!os)
{
  std::cerr << "Failure opening new.gif" << std::endl;
}
else
{
  // reset the steam
  e.getPhotoElements((::MtByte*)tmpBytes.get(), 0, MT_BEGIN_OFFSET);
  do {
    actualBytes = e.getPhotoElements((::MtByte*)tmpBytes.get(),
                                     bufSize, MT_CURRENT_OFFSET);
    os.write(tmpBytes.get(), actualBytes);
    totalBytes += actualBytes;
  } while (actualBytes == bufSize);
  os.close();
}
std::cout << "transferred " << totalBytes
          << " bytes from object to new.gif"
          << std::endl;
```

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 `cxx_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 C++ class files.

   ```
   mt_sdl stubgen --lang cxx -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::create(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::create(db);

m2.setReportsTo(m1);    // the manager

// create an employee
Employee& e = Employee::create(db);

e.setReportsTo(m2);

// specify an assistant for each of the two managers
m1.setAssistant(e);
m2.setAssistant(e);
```

```
// get the entire list of sucessors (class Manager)
MtObjectArray<Manager> list = e.getAssistantOf();
// access list as an array using operator[]
for (int i=0; i < (int)list.size(); i++)
  std::cout << "\t" << e.getFirstName() << " is "
            << list[i].getFirstName() << "'s assistant" << std::endl;

// create a few extra persons
Person& c1 = Person::create(db);

Person& c2 = Person::create(db);

// a std::vector of Type* can be converted to from
// MtObjectArray<T> which can be used as argument to set a list
// of successors
std::vector<Person*> cVec(2);
cVec[0] = &c1;
cVec[1] = &c2;
MtObjectArray<Person> children(db,cVec);
m2.setChildren(children);
```

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& c3 = Person::create(db);

// append to existing list
m2.appendChildren(c3);

// use of MtObjectArray<> for specifying a list to remove
std::vector<Person*> rVec(2);
rVec[0] = &c2;
rVec[1] = &c3;
MtObjectArray<Person> children2(db, rVec);
// remove
m2.removeChildren(children2);

// another signature supports a single removal
// m2.removeChildren(c2);
// m2.removeChildren(c3);

// 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 iterator method for each one-to-many relationship defined in the class.

```
std::cout << "\nAdd successors and iterate through children.."
          << std::endl;

MtObjectIterator<Person> piter = m2.childrenIterator();
while (piter.hasNext())
{
  Person &p = piter.next();
  std::cout << "\t\t" << p.getFirstName() << std::endl;
}
```

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.

```
int cnt = m2.getChildrenSize();
std::cout << "\tNow " << m2.getFirstName() << " has "
          << cnt << " children" << std::endl;
// an alternative to get the relationship size
// but the c++ objects are loaded before you can get the count
cnt = m2.getChildren().size();
```

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 iterator method for each index defined on the class.

```
// The lookup function must return a Person* to allow for NULL
// to represent no match
Person *found = Person::lookupPersonName(db, lastName, firstName);

// 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)
std::string fromFirstName = "Fred";
std::string toFirstName = "John";
std::string fromLastName = "Jones";
std::string toLastName = "Murray";

MtObjectIterator<Person> ppIter = Person::personNameIterator(db,
  fromLastName, fromFirstName, toLastName, toFirstName);
while (ppIter.hasNext())
{
    Person &p = ppIter.next();
    std::cout << "\t" << p.getFirstName() << " " << p.getLastName()
             << std::endl;
}
```

Index Lookup Count

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

```
MtValue vals[MT_MAX_CRITERIA] = {newString(fromLastName), newString(fromFirstName)};
MtSize num = Person::getPersonNameIndex(db).getObjectNumber(vals);
std::cout << "\n" << num << " object(s) retrieved" << std::endl;
```

Index Entries Count

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

```
MtSize count = Person::getPersonNameIndex(db).getIndexEntriesNumber();
std::cout << "\n" << count << " entries in the index " << std::endl;
```

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 iterator methods for each entry-point dictionary defined on the class.

```
// set the search string from command line
std::string searchString = av[3];

// open an iterator on the number of Persons that match
MtObjectIterator<Person> pIter = Person::commentDictIterator(db,
                                                    searchString);
while (pIter.hasNext())
{
  Person &p = pIter.next();
  std::cout << "\t" << p.getFirstName() << " " << p.getLastName()
            << std::endl;
}
```

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);
std::cout << "\n" << num << " object(s) retrieved" << std::endl;
```

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 `cxx_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 C++ class files.

   ```
   mt_sdl stubgen --lang cxx -f sql_eg.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.

```
db.open();
 db.startVersionAccess();
 // Set the SQL CURRENT_NAMESPACE to 'examples.cxx_examples.chap_9' so there is
 // no need to use the full qualified names to acces the schema objects
 db.setSqlCurrentNamespace("examples.cxx_examples.chap_9");
```

```
// create a statement, execute some sql and iterate through row/cols
MtStatement stmt(db);
std::string sqlStr =
  "SELECT e.name, e.id, e.boss.name FROM Employee e WHERE e.id > 2";
std::cout << "Sql Default Namespace: " << db.getSqlCurrentNamespace() << std::endl;
std::cout << "Query: " << sqlStr << std::endl;

MtResultSet res = stmt.executeQuery(sqlStr);

// list names and types
// column index is 1-based
for (unsigned int i = 1; i <= res.getColumnCount(); i++)
{
  std::cout << "column " << i << " '" << res.getColumnName(i)
            << "' type is: "
            << res.getColumnTypeName(i) << std::endl;
}
// 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())
{
  std::cout << "Employee name: " << res.getString(1) << ", id: "
            << res.getInteger(2) << ", boss : " << res.getString(3)
            << std::endl;
}

// always remember to close the statement when done
stmt.close();
db.endVersionAccess();
db.close();
```

Retrieving Objects from a SELECT statement

You can retrieve 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.

```
db.open();
db.startVersionAccess();
// Set the SQL CURRENT_NAMESPACE to 'examples.cxx_examples.chap_9' so there is
// no need to use the full qualified names to acces the schema objects
db.setSqlCurrentNamespace("examples.cxx_examples.chap_9");

// to get a list of objects, use the Ref operator
MtStatement stmt(db);
std::string sqlStr = "SELECT Ref(Employee), Ref(boss) FROM Employee WHERE id > 2;";
std::cout << "Sql Default Namespace: " << db.getSqlCurrentNamespace() << std::endl;
std::cout << "Query: " << sqlStr << std::endl;

MtResultSet res = stmt.executeQuery(sqlStr);
```

```
// list names and types
// column index is 1-based
for (unsigned int i = 1; i <= res.getColumnCount(); i++)
{
  std::cout << "column " << i << " '" << res.getColumnName(i)
            << "' type is: "
            << res.getColumnTypeName(i) << std::endl;
}
// still use the ResultSet, but get the object and coerce to the
// class we expect (note:  using dynamic_cast<> will throw
// std::bad_cast if the result type is not acceptable)
while (res.next())
{
  // show the class name of the reference
  std::cout << "[class = "
            << res.getMtObject(1)->getMtClass().getMtName()
            << "], ";
  // Note:  using dynamic_cast with pointer conversion returns
  // NULL on failure. If you would prefer an exception
  // (std::bad_cast), use object reference to make it an
  // assertion, e.g.:
  // Employee* e = &dynamic_cast<Employee&>(*(res.getMtObject(1)));
  Employee *e = dynamic_cast<Employee *>(res.getMtObject(1));
  Manager *m = dynamic_cast<Manager *>(res.getMtObject(2));
  if (e == NULL)
  {
    std::cout << "Empty Employee from Query!" << std::endl;
    continue;
  }
  std::cout << "Employee name: " << e->getName()
            << ", id: " << e->getId() << " boss: "
            << ((m != NULL) ? m->getName() : "[no boss]")
            << std::endl;
}

// remember to the close statement when done
stmt.close();

// always remember to close the statement when done
stmt.close();
db.endVersionAccess();
db.close();
```

10 Optimization

Installing Sample Applications

To install the sample applications for this section and the next:

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Change to the `chaps_10_11` directory in your installation (under `cxx_examples`).
3. Load `company.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `company.odl`.
4. Generate C++ class files.

   ```
   mt_sdl stubgen --lang cxx -f company.odl
   ```

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

   ```
   load localhost example

   newman1 localhost example JOHN DOE000025 10 201

   newman2 localhost example JOHN DOE000061 5 301

   delete localhost example
   ```

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, use the `MtClass::createInstances(num)` method, which returns an `MtObjectArray<>`. For example, to create *`count`* instances of the class `Employee`:

```
MtObjectArray<Employee>
emps(Employee::getClass(db)).createInstances(count).typeless());
for (int i = 0; i < count; i++)
{
        Employee &e = emps[i];
        // set attributes for employee i
        e.setFirstName(...);
        // etc
}
// commit (or rollback) still required
// etc
```

Note that `Employee::getClass(db)` is a generated static method for `Employee` which returns an instance of `MtClass`. Since the `MtClass::createInstances()` method returns an `MtObjectArray<MtObject>` (an array of generic `MtObject`), it must be converted to typeless form to be used in the copy constructor for `MtObjectArray<Employee>`; this is accomplished by the `MtObjectArray<>::typeless()` method.

When creating a large quantity of objects of different classes, an application can still optimize client-server traffic by preallocating OIDs. A block of OIDs can be preallocated for object creation within a transaction; whenever an object is created, it will use a preallocated OID if available, avoiding traffic to the server. Note, however, that these OIDs are "wasted" if not used by the current transaction. See the Matisse C++ binding API reference for complete documentation of `MtDatabase::preallocate(int num)`.

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 `Department` object's `team` relationship:

```
MtObjectArray<Employee> emps = aDepartment.getTeam();  // get the array of successors
emps.load();  // load all the Employee instances
for (int i = 0; i < emps.size(); i++)
{
        Employee &e = emps[i];
        myid = e.getId();    // collect info from Employee instance
}
```

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 `e.remove()`.

Note that the creation of the `MtObjectArray<Employee>` does not populate the client cache, nor does it create 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 `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.

Clearing the C++ Instance Cache

A cache of all the C++ instances the C++ binding creates is maintained by `MtDatabase`. A C++ instance is a very small "stub" to a Matisse instance consisting only of references to the relevant OID and `MtDatabase` (plus any C++ overhead).

The C++ instance cache is populated automatically during object lookup. There is no explicit load method. The cache can be configured to be cleared automatically, for example at connection or transaction boundary. This policy can be set by `MtDatabase::setObjectCacheBoundary()`. Since these C++ stubs are very small, automatic clearing is usually adequate, but when necessary (for example, during a lengthy version access) the cache can be cleared explicitly by calling `MtDatabase::clearObjectCache()`. See the *Matisse C++ Binding API Reference* for additional discussion of these two methods.

The default clearing policy set during creation of a new `MtDatabase` is `NO_BOUND`, which means that the cache can only be cleared explicitly; either by a call to `clearObjectCache()` or as would happen during object cleanup when the `MtDatabase` is destroyed (in the `MtDatabase` destructor).

In either case (automatic or explicit), the C++ instance cache is cleared completely. There is no method for removing only selected C++ instances.

The client cache and C++ instance cache operate independently. That is, a C++ instance may be cached when the corresponding Matisse instance is not, and vice-versa.

Using Pointers

Within the binding, pointers to C++ instances are only used in a few cases where existence needs to be confirmed (for example, during an index lookup). All other cases use a C++ reference; this design helps avoid accumulation of dangling pointers to cache instances that no longer exist. However, use of pointers cannot be strictly enforced as it is only a design pattern, therefore, use caution when using pointers. For example, when the C++ instance cache clearing policy is `TIME_BOUND` (transaction), do not hold pointers across transaction boundaries.

11 Additional Topics

Arrays

The C++ binding defines two template classes to be used by API methods which pass or return arrays.

- `MtObjectArray<>` is used only for arrays of Matisse objects and provides an API which includes methods to access the individual objects using the `[]` operator as well as methods supporting conversions to/from `std::vector`. See *Other Operations on Multiple Objects* on page 28 for more information about `MtObjectArray<>` and `delete.cpp` for examples.

- `MtArray<>` is used for arrays of any type which can be used as an attribute value, both C++ primitives as well as Matisse-supplied types such as `MtTimestamp`. The `MtArray<>` class also provides element access via the `[]` operator, but additionally, it is derived from `MtValue` and therefore inherits methods to access the type, size, and so on. For example:

```
// a Department has a list of integers in the Days attribute
MtArray<int> days = thisDepartment.getDays();
for (int i = 0; i < days.getSize(); i++)
{
    std::cout << i << " : " << days[i] << std::endl;
}

// make a new array and set 'Days' attribute
// previously allocated array, 'myDays' to use..
int *myDays = ....; // can be allocated or stack based
// make new int array and let it borrow the pointer
MtArray<int> newDays = newIntegers(mySize, myDays, MtPointerBorrow);
for (i = 0; i < newDays.getSize(); i++)
newDays[i] = ...;
thisDepartment.setDays(newDays);
```

See the *Matisse C++ Binding API Reference* (`%MATISSEHOME%/docs/cxx/api/index.html`) for the full API and documentation on `MtValue` and `MtArray` instance creation.

Namespaces

The C++ binding defines and uses the `matisse` namespace; all core classes for the binding are defined under the space. Any user generated code, by default, contains no namespace specification and is therefore in the top level (`::`). If the `--ln` *`namespace`* option is passed to `mt_sdl`, the code will be generated with the namespace specification for *`namespace`*. The namespace specification affects how the scope of the classes as well as the search path used by the default object factory (`MtDynamicObjectFactory`) to create the proper C++ class based on the database object class.

If a class is in a namespace, then the header will put all definitions within a namespace `<name> { }` and in order to use that class in user applications, the class name must be fully qualified unless the module contains a `using namespace <name>`. For example, to refer to the `MtDatabase` class in a source file without a `using` clause, it would be referred to as `matisse::MtDatabase`.

For more information on namespaces, refer to a C++ reference such as *The C++ Programming Language* by Bjarne Stroustrup.

Error Handling

Example applications `newman1.cpp` and `newman2.cpp` demonstrate 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. See the comments in `newman1.cpp` for more information and instructions on running the applications. See *Installing Sample Applications* on page 27 for installation instructions.

```
    for (int i = 0; i < newManagerCount; i++)
    {
      try
      {
        actualTransCount++;
        db.startTransaction();

        // find the employee by name lookup; in this simple
        // example, this lookup should be moved outside the
        // loop, since the search criteria do not change, and
        // left here for example only as we use the logic of
        // finding Employee to commit/rollback transaction.
        Employee *e =
          dynamic_cast<Employee*>(Person::lookupPersonName(db, asstLastName,
asstFirstName));

        if (e != NULL)
        {
          // create a new manager
          Manager &m = Manager::create(db);
          std::ostringstream os1;
          os1 << "JACK";

          std::ostringstream os2;
          os2 << "MORRISON" << std::setw(6) << setfill('0') << useId;

          // manager attributes
          m.setFirstName(os1.str());
          m.setLastName(os2.str());
          m.setId(useId);
          // set the assistant
          m.setAssistant(*e);

          db.commit();
          actualCreateCount++;
        }
        else
        {
          std::cerr << "could not find " << asstFirstName << " "
                    << asstLastName << std::endl;
          if (db.isTransactionInProgress())
          {
            db.rollback();
          }
          actualAbortCount++;
        }
```

```
      }
      // an exception during create will rollback transaction
      catch (MtException& mte)
      {
        if (db.isTransactionInProgress())
        {
          db.rollback();
        }
        actualAbortCount++;
        std::cerr << mte << std::endl;
        std::cerr << "At transaction #: " << i
                  << ", id: " << useId << std::endl;
      }
      useId++;           // update for the next employee id
    }

    try
      {
        actualTransCount++;
        db.startTransaction();

        MtStatement stmt(db);
        std::string selName = "sel1"; // selection name for
        // setting relationship

        // similar to non-SQL version, finding the employee
        // could be outside the loop, since it is static
        std::ostringstream sqs_emps;
        sqs_emps
          << "SELECT REF(e) FROM examples.cxx_examples.chaps_10_11.Employee e WHERE
e.firstName = '"
          << asstFirstName << "' and e.lastName = '" << asstLastName
          << "' into " << selName;

        stmt.execute(sqs_emps.str());

        // check
        MtResultSet res =  stmt.executeQuery("SELECT REF(s) FROM sel1 s");
        if (res.next())
        {
          Employee *e = dynamic_cast<Employee*>(res.getMtObject(1));
          std::cout << e->getFirstName() << " " << e->getLastName()
                    << std::endl;
        }
        else
        {
          std::cerr << "could not find " << asstFirstName << " "
                    << asstLastName << std::endl;
        }

        std::ostringstream sqs_man;

        // buld the insert command
        sqs_man << "INSERT INTO examples.cxx_examples.chaps_10_11.Manager(firstName,
lastName, id, assistant) Values("
                << "'JIM', 'JACKSON" << std::setfill('0') << std::setw(6)
                << useId << "',"
                << useId << "," << selName << ");";
```

```
    int res = stmt.executeUpdate(sqs_man.str());
    std::cout << "Update: " << sqs_man.str() << std::endl
              << "Result : " << res << std::endl;

    if (res != 1)
    {
      // if insert command didn't have an exception,
      // result would be 1, so we should not reach this
      std::cerr << "Should not be here!" << std::endl;
      // but let continue
    }
    db.commit();
    actualCreateCount++;
  }
  catch (MtSQLException &mtsqle)
  {
    std::cerr << mtsqle << std::endl;
    if (db.isTransactionInProgress())
    {
      db.rollback();
    }
    actualAbortCount++;
  }
  catch (MtException &mte)
  {
    std::cerr << mte << std::endl;
    if (db.isTransactionInProgress())
    {
      db.rollback();
    }
    actualAbortCount++;
  }
  useId++;
}
```

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 `cxx_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;

// Open the connection to the database
dbcon.open();

matisse::MtEvent &subscriber = *new matisse::MtEvent(dbcon);

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

subscriber.subscribe(eventSet);

::MtEvent triggeredEvents;
// Wait 1000 ms for events to be triggered
```

```
// return false if not event is triggered until the timeout is reached
if (subscriber.wait(1000, &triggeredEvents)) {
    cout << "Events (#" << i << ") triggered:" << endl;
    cout << (((triggeredEvents & TEMPERATURE_CHANGES_EVT) > 0) ? "" : "No ")
         << "Change in temperature" << endl;
} else {
   cout << "No Event received after ~1 sec\n" << endl;
}

cout << "Unsubscribe to 4 Events" << endl;
// Unsubscribe to all 4 events
subscriber.unsubscribe();

// Close the database connection
dbcon.close();
```

Events Notification

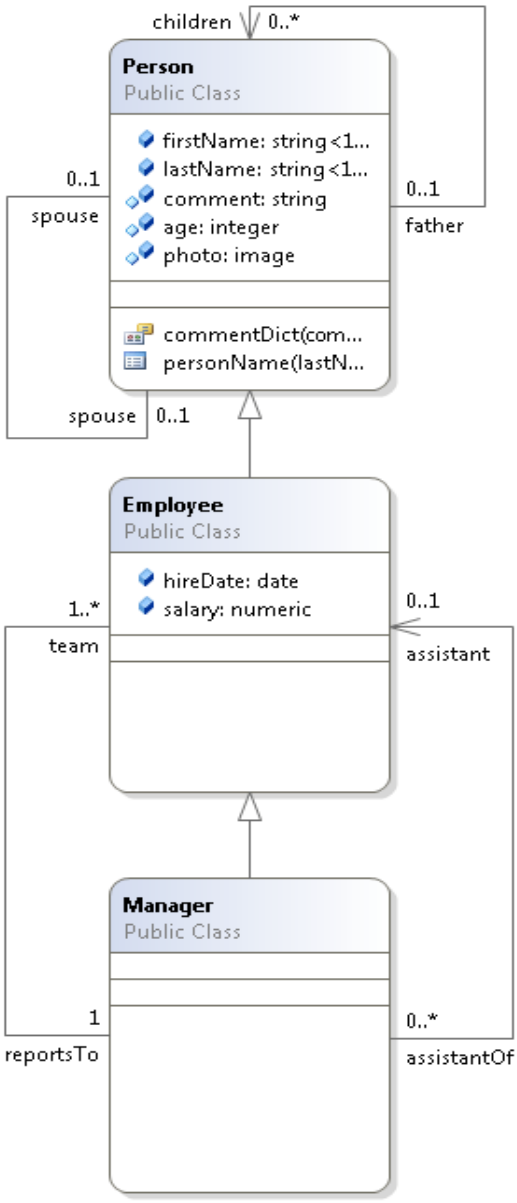
This section illustrates event selection and notification.

```
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;

// Open the connection to the database
dbcon.open();

matisse::MtEvent &notifier = *new matisse::MtEvent(dbcon);

::MtEvent eventSet;

eventSet = 0;
eventSet |= TEMPERATURE_CHANGES_EVT;
eventSet |= RAINFALL_CHANGES_EVT;
eventSet |= HIMIDITY_CHANGES_EVT;

notifier.notify(eventSet);

// Close the database connection
dbcon.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 C++ Binding API documentation located from the Matisse installation root directory in `docs/cxx/api/index.html`.

Appendix A: Example Schema




```
module examples {

  module cxx_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 C++ class files generated by `mt_sdl`. Definitions are in *class*`.h`, inlines in *class*`.hpp`, and other source in *class*`.cpp`.

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

```
getInstanceNumber(const MtDatabase &db)
```

Open an iterator

```
MtObjectIterator<Person> instanceIterator(const MtDatabase &db)
```

Sample constructor

```
Person &create(const MtDatabase &db))
```

Sample ostream << operator

```
std::ostream &operator<<(std::ostream &o, const Person &obj)
```

This non-class method overloads the insertion operator for streams, which enables code such as:

```
cout << aPerson << endl
```

Get descriptor

```
MtClass& getClass(const MtDatabase &db)
```

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

Factory constructor

```
MtObject* newStub(const MtDatabase &db, ::MtOid oid)
```

This constructor is called by `MtObjectFactory`. It is public for technical reasons but is not intended to be called directly by user methods.

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

```
getFirstName()
```

Set value

```
setFirstName(const std::string & val)
```

Get descriptor `getFirstNameAttribute(const 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 `MtArray<unsigned char> getPhoto()`

Set value `setPhoto(const MtArray<unsigned char> & val)`

Get elements
```
getPhotoElements(::MtByte *value, unsigned int len, unsigned int
    offset=MT_CURRENT_OFFSET)
```

Set elements
```
setPhotoElements(::MtByte *value, unsigned int len, unsigned int
    offset=MT_CURRENT_OFFSET, bool discardAfter=MT_FALSE)
```

Count elements `getPhotoSize()`

Get descriptor `MtAttribute& getPhotoAttribute(const 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 `clearSpouse()`

Get descriptor `MtRelationship& getSpouseRelationship(const 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* getAssistant()`

Set successor `setAssistant(const 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<Employee> getTeam()`

Open an iterator
```
MtObjectIterator<Employee> teamIterator()
```

Count successors
```
getTeamSize()
```

Set successors
```
setTeam(const MtObjectArray<Employee> &succs)
```

Add successors

Insert one successor before any existing successors:
```
prependTeam(const Employee &succ)
```

Add one successor after any existing successors:
```
appendTeamp(const Employee &succ)
```

Add multiple successors after any existing successors:
```
appendTeam(const MtObjectArray<Employee> &succs)
```

Remove successors
```
removeTeam(const Employee &succ)
removeTeam(const MtObjectArray<Employee> &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 iterator methods is dependent on the number of criteria defined for the index (in this case, two, `lastName` and `firstName`).

Lookup
```
Person* lookupPersonName(const MtDatabase &db, const std::string &
    lastName, const std::string & firstName)
```

Open an iterator
```
MtObjectIterator<Person> personNameIterator(const MtDatabase &db,
    const std::string & fromLastName, const std::string &
    fromFirstName, const std::string & toLastName, const std::string
    & toFirstName, const MtClass* filterClass=NULL, ::MtDirection
    direction=MT_DIRECT, int numObjPerBuffer=MT_MAX_PREFETCHING)
```

Get descriptor
```
MtIndex& getPersonNameIndex(const 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(const MtDatabase &db, const std::string
    &value)
```

Open an iterator
```
MtObjectIterator<Person> commentDictIterator(const MtDatabase &db,
    const std::string &value, const MtClass* filterClass=NULL, int
    numObjPerBuffer=MT_MAX_PREFETCHING)
```

Get descriptor

```
MtEntryPointDictionary& getCommentDictDictionary(const MtDatabase
    &db)
```

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