1 Introduction

Scope of This Document

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

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

Before Running the Examples

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

- Install Matisse 9.1.0 or later.
- Install the Eiffel version 7.0 or later for your operating system.
- Download and extract the Matisse Eiffel binding source code and sample code from the Matisse Web site:
  
  http://www.matisse.com/developers/documentation/

  The sample code files are grouped in subdirectories by category. For example, the code snippets from the following chapter are in the connect directory.

- Build the Matisse Eiffel binding from the source code. Follow the building instructions as detailed in the README file.

- 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 category 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 readwrite/objects.odl for the Chapter 3 demo.
• Generate Eiffel class files:
  
  ```
  mt_sdl stubgen --lang eiffel -f objects.odl
  ```

• Open the Eiffel project for instance `connect.ecf` in Eiffel Studio and compile it.

• In Eiffel Studio or in a command line windows run the built application.
2 Connection and Transaction

All interaction between client Eiffel 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 `MT_DATABASE` class. Once the connection is established, your Eiffel application may interact with the database using the schema-specific methods generated by `mt_sdl`. The following sample code shows 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 5.
2. Change to the `connect` directory in your installation (under `examples`).
3. Open the Eiffel project for instance `connect.ecf` in Eiffel Studio and compile it.
4. In Eiffel Studio or in a command line windows run the built application.

Read Write Transaction

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

```eiffel
local
db: MT_DATABASE
do
create db.make(host, dbname)
db.open()
db.start_transaction({MT_DATABASE}.Mt_Min_Tran_Priority)
print("connection and read write access to: " + db.out + "\N")
db.commit()
db.close()
end
```

Read-Only Access

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

```eiffel
local
db: MT_DATABASE
do
create db.make(host, dbname)
db.open()
```
db.start_version_access(Void)

print("connection and read only access to: " + db.out + "%N")

db.end_version_access()

db.close()

end

Version Access

The following code illustrates methods of accessing various versions of a database.

list_versions (db: MT_DATABASE)
local
    iter: MT_VERSION_ITERATOR
    ver : STRING

do
    iter := db.version_iterator
    from
    iter.start
    until
    iter.exhausted
    loop
        ver := iter.item
        print("   " + ver + "%N")
        iter.forth
    end
end

version_navigation (host, dbname:STRING)
local
    impossible: BOOLEAN
    dev.ex: DEVELOPER_EXCEPTION
    db: MT_DATABASE
    vername : STRING

do
    if impossible = False then
        print("%NTest Version Navigation Connect:%N")
        create db.make(host, dbname)
        db.open()
        db.start_transaction([MT_DATABASE].Mt_Max_Tran_Priority)
        print("Version list before regular commit:%N")
        list_versions (db)
        db.commit ()
        db.start_transaction([MT_DATABASE].Mt_Max_Tran_Priority)
        print("Version list after regular commit:%N")
        list_versions (db)
        vername := db.commit_and_save("Snapshot_")
        print("Commit to version named: " + vername + ">%N")
db.start_version_access(Void)
print("Version list after named commit:%N")
list_versions (db)

db.end_version_access()

db.start_version_access(vername)
print("Sucessful access within version: " + vername + "%N")

db.end_version_access()

db.close()

rescue
   dev_ex ?= (create {EXCEPTION_MANAGER}).last_exception
   if dev_ex /= Void then
      print("%NException occurred on " + db.out + "%N")
      print("%NERROR message: " + dev_ex.message + "%N")
      if {MT_EXCEPTIONS}.c_matisse_exception_code = {MT_EXCEPTIONS}.MATISSE_NOSUCHDB then
         print("Unable to connect to: " + db.out + "%N")
         print("Make sure the database is started%N")
      end
   end
end
impossible := True
retry
end

**Specific Options**

This example shows how to enable the local client-server memory transport and to set or read various connection options and states.

local
db: MT_DATABASE
do
   create db.make(host, dbname)
   if read_only = True then
      db.set_data_access_mode({MT_DATABASE}.mt_data_readonly)
   else
      db.set_data_access_mode({MT_DATABASE}.mt_data_modification)
   end
   db.open()
end
c connected := db.is_connection_open()
if connected then
   if db.data_access_mode () = {MT_DATABASE}.Mt_Data_Readonly then
      db.start_version_access(Void)
   else
      db.start_transaction({MT_DATABASE}.Mt_Max_Tran_Priority)
   end
print("connection and ")
if db.is_version_access_in_progress() = True then
  print("read only access to: ")
else
  print("read write access to: ")
end
print(db.out)
print("%N")

if db.is_transaction_in_progress() = True then
  db.rollback()
else
  db.end_version_access()
end
end

db.close()
end

More about MT_DATABASE

As illustrated by the previous sections, the MT_DATABASE class provides all the methods for database connections and transactions. The reference documentation for the MT_DATABASE class is included in the Matisse Eiffel Binding API documentation located from the Matisse Eiffel binding installation root directory in docs/eiffel/api/index.html.
3 Working with Objects and Values

This chapter explains how to manipulate objects with the object interface of the Matisse Eiffel binding. The object interface allows you to directly retrieve objects from the Matisse database without Object-Relational mapping, navigate from one object to another through the relationship defined between them, and update properties of objects without writing SQL statements.

The object interface can be used with Matisse Eiffel SQL interface as well. For example, you can retrieve objects with SQL, then use the object interface to navigate to other objects from these objects, or update properties of these objects using the accessor methods defined on these classes.

Running the Examples on Objects

This sample program creates objects from 2 classes (Person and Employee), lists all Person objects (which includes both objects, since Employee is a subclass of Person), deletes 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 5.

2. Change to the readwrite directory in your installation (under examples).

3. Load objects.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on ‘Schema->Import ODL Schema’, then select chap_3/objects.odl for this demo.

4. Generate Eiffel class files:
   `mt_sdl stubgen --lang eiffel -f objects.odl`

5. Open the Eiffel project for instance readwrite.ecf in Eiffel Studio and compile it.

6. In Eiffel Studio or in a command line windows run the built application.

Creating Objects

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

```
make_person (a_db: MT_DATABASE)
   -- Default make feature provided as an example
   -- You may delete or modify it to suit your needs.
   do
      make_from_mtclass (a_db.get_mtclass ("Person"))
   end

   db.start_transaction({MT_DATABASE}.Mt_Min_Tran_Priority)

   create p.make_person (db)
   p.set_firstname("John");
   p.set_lastname("Smith")
```
p.set_age(42)
create a.make_Postaladdress (db)
a.set_city("Portland")
a.set_postalcode("97201")
p.set_address(a)
print("%NPerson John Smith from Portland created.%N")

create e.make_employe (db)
e.set_firstname("Jane");
e.set_lastname("Jones")
-- Age is nullable we can leave it unset
create salary.make_from_string ("$5000.00")
e.set_salary(salary)
create hiredate.make (2010,10,10)
e.set_hiredate(hiredate)
print("%NEmployee Jane Jones created.%N")

db.commit()
Listing Objects

This section illustrates the enumeration of objects from a class. The `create_instance_iterator()` method defined on `MTCLASS` allows you to enumerate the instances of this class and its subclasses. The `instance_number()` method returns the number of instances of this class.

```plaintext
local
    db: MT_DATABASE
    p: PERSON
    cnt: INTEGER
    personCls: MTCLASS
    addressCls: MTCLASS
    p_iter1, p_iter2: MT_OBJECT_ITERATOR[PERSON]
    o_iter3: MT_OBJECT_ITERATOR[MTOBJECT]
do
    create db.make(host, dbname)
db.open()

    p_iter1.make_empty_iterator()
p_iter1.start
    until p_iter1.exhausted loop
        p := p_iter1.item
        print("   " + p.firstname + " " + p.lastname + " from ")
        if (p.address /= Void) then
            print(p.address.city)
        else
        end
    end
end
```

```plaintext
if db.num_preallocated () < 2 then
    n := db.preallocate(nb_prealloc)
end
i := i + 1
end

if db.is_transaction_in_progress() then
    db.commit()
end
```
print("???")
end

print(" is a " + p.mtclass.mtname + "%N")
p_iter1.forth
end

p_iter1.close

print("%NList all person(s) - Solution 2:%N")
-- Solution 2: Create an iterator with the class
create p_iter2.make_class_instance_iterator (db, personCls, {MT_DATABASE}.Mt_Max_Prefetching)

from
p_iter2.start
until
p_iter2.exhausted
loop
p := p_iter2.item
print("   " + p.firstname + " " + p.lastname + " from ")
if (p.address /= Void) then
    print(p.address.city)
else
    print("???")
end

print(" is a " + p.mtclass.mtname + "%N")
p_iter2.forth
end

p_iter2.close

print("%NList all person(s) - Solution 3:%N")
-- Solution 3: Create an iterator from the class with the MTOBJECT
-- base type
o_iter3 := personCls.instance_iterator ({MT_DATABASE}.Mt_Max_Prefetching)
from
o_iter3.start
until
o_iter3.exhausted
loop
p ?= o_iter3.item
if (p /= Void) then
    print("   " + p.firstname + " " + p.lastname + " from ")
    if (p.address /= Void) then
        print(p.address.city)
    else
        print("???")
    end
    print(" is a " + p.mtclass.mtname + "%N")
end
o_iter3.forth
end

o_iter3.close

db.end_version_access()
db.close()
The `create_own_instance_iterator()` method allows you to enumerate the own instances of a class (excluding its subclasses). The `own_instance_number()` method returns the number of instances of a class (excluding its subclasses).

```pascal
local
  db: MT_DATABASE
  p: PERSON
  cnt: INTEGER
  personCls: MTCLASS
  p_iter1, p_iter2: MT_OBJECT_ITERATOR[PERSON]
do
  create db.make(host, dbname)
db.open()
  db.start_version_access(Void)
  personCls := db.get_mtclass("Person");
  cnt := personCls.own_instance_number()
  print("%N" + cnt.out + " Person(s) (excluding subclasses) in the database.%N")
  print("%NList all person(s) (excluding subclasses) - Solution 1:%N")
  -- Solution 1: Create an iterator from the class
  create p_iter1.make_empty_iterator()
  personCls.create_own_instance_iterator (p_iter1,
  {MT_DATABASE}.Mt_Max_Prefetching)
  from
    p_iter1.start
  until
    p_iter1.exhausted
   loop
    p := p_iter1.item
    print("   " + p.firstname + " " + p.lastname + " from ")
    if (p.address /= Void) then
      print(p.address.city)
    else
      print(???")
    end
    print("  is a "+ p.mtclass.mtname + "%N")
    p_iter1.forth
  end
  p_iter1.close
  print("%NList all person(s) (excluding subclasses) - Solution 2:%N")
  -- Solution 2: Create an iterator with the class
  create p_iter2.make_class_own_instance_iterator (db, personCls,
  {MT_DATABASE}.Mt_Max_Prefetching)
  from
    p_iter2.start
  until
    p_iter2.exhausted
   loop
    p := p_iter2.item
    print("   " + p.firstname + " " + p.lastname + " from ")
    if (p.address /= Void) then
      print(p.address.city)
    else
      print(???")
    end
    print("  is a "+ p.mtclass.mtname + "%N")
    p_iter2.forth
  end
  p_iter2.close
```

The `create_own_instance_iterator()` method allows you to enumerate the own instances of a class (excluding its subclasses). The `own_instance_number()` method returns the number of instances of a class (excluding its subclasses).
print(p.address.city)
else
  print("???")
end

print(" is a "+ p.mtclass.mtname + "%N")

p_iter2.forth
end
p_iter2.close

db.end_version_access()
db.close()

Deleting Objects

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

```effel
// Remove created objects
...
// NOTE: does not remove the object sub-parts
p.remove()
```

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

```effel
--
-- Overrides MTOBJECT.deep_remove() to remove the Address object if any.
--
deep_remove()
  -- Delete the current object and the Address from the database.
  local
    adrs: POSTALADDRESS
  do
    adrs := Current.address ()
    if adrs /= Void then
      -- be careful of cyclic calls
      -- when using deep_remove() on navigation
      adrs.deep_remove ()
    end
  end
  remove ()
end
```

The remove_all_instances() method defined on MTCLASS delete all the instances of a class.

```effel
personCls := db.get_mtclass("Person");
personCls.remove_all_instances()
```
Comparing Objects

This section illustrates how to compare objects. Persistent objects must be compared with the is_equal() method. You can’t compare persistent object with the = operator.

```java
...
  if(p1.is_equal(p2))
    print("Same objects\n");
```

Running the Examples on Values

This example shows how to get and set values for various Matisse data types including Null values, and how to check if a property of an object is a Null value or not.

This example uses the database created for Objects Example. It creates objects, then manipulates its values in various ways.

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.

```java
local  
  db: MT_DATABASE
  p: PERSON
  e: EMPLOYEE
  a: POSTALADDRESS
  salary: DECIMAL
  hiredate: DATE

do

  -- Setting strings
  p.set_firstname("John");
  p.set_lastname("Smith")

  -- Setting numbers
  p.set_age(42)

  -- Setting Numerics
  create salary.make_from_string ("85000.00")
  e.set_salary(salary)

  -- Setting Dates
  create hiredate.make (2010,10,10)
  e.set_hiredate(hiredate)

  -- Setting an attribute of type int to NULL
  p.set_null(p.get_age_attribute ())

end
```
-- Getting String values
print("%NComment:" + e.comment () + "%N")
-- suppresses output if no value set
if not e.is_age_null () then
    print("Age:" + e.age ().out + "%N")
else
    print("Age: NULL")
    if e.is_age_default_value () then
        print(" (default value)")
    end
    print("%N")
end

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.

-- Removing value returns attribute to default
e.remove_age();
e.remove_dependents();

-- Getting again to show effect of removing value
print("%NComment:" + e.comment () + "%N")
if not e.is_age_null () then
    print("Age:" + e.age ().out + "%N")
else
    print("Age: NULL")
    if e.is_age_default_value () then
        print(" (default value)")
    end
    print("%N")
end

Streaming Values

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

Retrieving an Object from its Oid

This section illustrates a very commonly used feature in the binding. Using the Object Identifier (OID) is very efficient for retrieving one object from the database. The example below illustrates how to view an image stored into the database using the object Identifier to quickly retrieve the object.
local
db: MT_DATABASE
p: PERSON
buffer: ARRAY [NATURAL_8]
person_oid: INTEGER
do
create db.make(host, dbname)
db.open()

db.start_version_access(Void)

p ?= db.upcast(person_oid)
buffer := p.photo ()

db.end_version_access()

db.close()
4 Working with Relationships

One of the major advantages of the object interface of the Matisse Eiffel binding is the ability to navigate from one object to another through a relationship defined between them. Relationship navigation is as easy as accessing an object property.

Running the Examples on Relationships

This example creates several objects, then manipulates the relationships among them in various ways.

1. Follow the instructions in Before Running the Examples on page 5.
2. Change to the retrieve directory (under examples).
3. Load examples.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on ‘Schema->Import ODL Schema’, then select retrieve/examples.odl for this demo.
4. Generate Eiffel class files:
   ```
   mt_sdl stubgen --lang eiffel -f examples.odl
   ```
5. Open the Eiffel project for instance rshp.ecf in Eiffel Studio and compile it.
6. In Eiffel Studio or in a command line windows run the built application.

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.

```
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.

create c3.make_person (db)

-- add successors
m.append_children(c3)

-- remove successors
m.remove_children(c3)

-- clearing all successors (this only breaks links, it does
-- not remove objects)
m.clear_children();

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.

-- Iterate when the relationship is large is always more efficient
iter := m.children_iterator ()
from
    iter.start
until
    iter.exhausted
loop
c := iter.item
   print("   " + c.firstname + " " + c.lastname + ">
iter.forth
end
iter.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.

-- Get the relationship size without loading the Java objects
-- which is the fast way to get the size
cnt := m.children_size()
print("   " + m.firstname() + " has " + cnt.out + " children");

-- an alternative to get the relationship size
-- but the Eiffel objects are loaded before you can get the count
cnt := m.children().count
print("   " + m.firstname() + " has " + cnt.out + " children");
5 Working with Indexes

While indexes are used mostly by the SQL query optimizer to speed up queries, the Matisse Eiffel binding also provides the index query APIs to look up objects based on a key value(s). The stubclass defines both lookup methods and iterator methods for each index defined on the class.

Running the Examples on Indexes

Using the PersonName index, it checks whether the database contains an entry for a person matching the specified name. The application will list the names in the database, indicate whether the specified name was found, and return results within a sample range (defined in the source) using an iterator.

1. Follow the instructions in Before Running the Examples on page 5.
2. Change to the retrieve directory (under examples).
3. Load examples.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on “Schema->Import ODL Schema”, then select retrieve/examples.odl for this demo.
4. Generate Eiffel class files:
   ```
   mt_sdl stubgen --lang eiffel -f examples.odl
   ```
5. Open the Eiffel project for instance index.ecf in Eiffel Studio and compile it.
6. In Eiffel Studio or in a command line windows run the built application.

Index Lookup

This section illustrates retrieving objects from an index. The stubclass provides accessors to the name of each index defined on the class.

1. Lookup

   ```
   local
   db: MT_DATABASE
   f_name, l_name: STRING
   p: PERSON
   filter: MTCLASS
   idxCls: MTINDEX
   key: ARRAY[ANY]
   do
     idxCls := db.get_mtindex({PERSON}.personname_name);
     f_name := "John"
     l_name := "Murray"
     print("
   ```

   `create key.make(1, 2)`
   `key.put(l_name, 1)`
   `key.put(f_name, 2)`
p ?= idxCls.\texttt{lookup} (key, Void)

if (p /= Void) then
  print(" found exactly one Person: " + p.firstname + ",")
else
  print(" nobody found")
end

2. Iterate

local
  db: MT\_DATABASE
  i: INTEGER
  from\_f\_name, from\_l\_name, to\_f\_name, to\_l\_name: STRING
  p: PERSON
  idxCls: MTINDEX
  start\_key, end\_key: ARRAY[ANY]
  p\_iter: MT\_OBJECT\_ITERATOR[PERSON]
  o\_iter: MT\_OBJECT\_ITERATOR[MTOBJECT]
do

  idxCls := db.\texttt{get\_mtindex}({PERSON}.\texttt{personname\_name});
  from\_f\_name := "Fred"
  from\_l\_name := "Jones"
  to\_f\_name := "John"
  to\_l\_name := "Murray"
  print("Lookup from Person " + from\_f\_name + " " + from\_l\_name + " to " + to\_f\_name + " " + to\_l\_name + ",")

  create start\_key.make(1, 2)
  start\_key.put(from\_l\_name, 1)
  start\_key.put(from\_f\_name, 2)

  create end\_key.make(1, 2)
  end\_key.put(to\_l\_name, 1)
  end\_key.put(to\_f\_name, 2)

  print("Solution 1")
  -- Solution 1: Create an iterator from the index
  create \texttt{p\_iter.make\_empty\_iterator} ()
  idxCls.\texttt{create\_iterator} (p\_iter, start\_key, end\_key, Void, {MTINDEX}.Mt\_Direct,
  {MT\_DATABASE}.Mt\_Max\_Prefetching)
i := 0
from
  p\_iter.start
until
  p\_iter.exhausted
loop
  p := p\_iter.item

  print(" + p.firstname + " + p.lastname + ",")
i := i + 1

  p\_iter.forth
end
p\_iter.close
print(" + i.out + " Person(s) found")
print("Solution 2\n")
-- Solution 2: Create an iterator from the index with the MTOBJECT
-- base type
o_iter := idxCls.iterator (start_key, end_key, Void, {MTINDEX}.Mt_Direct,
{MT_DATABASE}.Mt_Max_Prefetching)
i := 0
from o_iter until o_iter.exhausted loop
p := o_iter.item
if (p /= Void) then
   print("   " + p.firstname + " " + p.lastname + "\n")
i := i + 1
end
end
o_iter.forth
end
o_iter.close

Index Lookup Count

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

local
db: MT_DATABASE
f_name, l_name: STRING
i: INTEGER
filter: MTCLASS
idxCls: MTINDEX
key: ARRAY[ANY]
do
idxCls := db.get_mtindex({PERSON}.personname_name);
f_name := "John"
l_name := "Murray"
print("Looking for Person '" + f_name + " " + l_name + '"\n")
create key.make(1, 2)
key.put(l_name, 1)
key.put(f_name, 2)
i := idxCls.object_number (key, Void)
print(" " + i.out + " objects retrieved\n")

Index Entries Count

This section illustrates retrieving the number of entries in an index. The index_entries_number() method is defined on the MTINDEX class.
local
  db: MT_DATABASE
  i: INTEGER
  idxCls: MTINDEX
do
  idxCls := db.get_mtindex({PERSON}.personname_name);
  i := idxCls.index_entries_number ()
  print("" + i.out + " entries in the index\n")
6 Working with Entry-Point Dictionaries

An entry-point dictionary is an indexing structure containing keywords derived from a value, which is especially useful for full-text indexing. While the entry-point dictionary can be used with SQL query using ENTRY_POINT keyword, the object interface of the Matisse Eiffel binding also provides APIs to directly retrieve objects using the entry-point dictionaries.

Running the Examples on Dictionaries

Using the commentDict entry-point dictionary, the example retrieves the Person objects in the database with Comments fields containing a specified character string.

1. Follow the instructions in Before Running the Examples on page 5.
2. Change to the retrieve directory (under examples).
3. Load examples.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on ‘Schema->Import ODL Schema’, then select retrieve/examples.odl for this demo.
4. Generate Eiffel class files:
   
   \texttt{mt\_sdl stubgen --lang eiffel -f examples.odl}

5. Open the Eiffel project for instance epdict.ecf in Eiffel Studio and compile it.
6. In Eiffel Studio or in a command line windows run the built application.

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.

1. Lookup

```eiffel
local
  db: MT\_DATABASE
  search\_string: STRING
  p: PERSON
  epdictCls: MTENTRYPOINTDICTIONARY

do
  epdictCls := db.get\_mtentrypointdictionary({PERSON}.commentdict\_name);
  search\_string := "knees"
  print("%NLooking for one Person with '" + search\_string + '" in the 'comment' text"%N)
  p ?= epdictCls.lookup (search\_string, Void)
  if (p /= Void) then
    print(" found exactly one Person: "+ p.firstname + ";")
```
else
    print(" nobody found\n")
end

2. Iterate

local
db: MT_DATABASE
i: INTEGER
search_string: STRING
p: PERSON
epdictCls: MTENTRYPOINTDICTIONARY
p_iter: MT_OBJECT_ITERATOR[PERSON]
o_iter: MT_OBJECT_ITERATOR[MTOBJECT]
do

epdictCls := db.get_mtentrypointdictionary({PERSON}.commentdict_name);
search_string := "knees"
print("%NLooking for Persons with " + search_string + " in the 'comment' text\n")

print("Solution 1\n")
-- Solution 1: Create an iterator from the EP dict
create p_iter.make_empty_iterator()
epdictCls.create_iterator (p_iter, search_string, Void,
{MT_DATABASE}.Mt_Max_Prefetching)
i := 0
from
    p_iter.start
until
    p_iter.exhausted
loop
    p := p_iter.item

    print("   " + p.firstname + " " + p.lastname + "\n")
i := i + 1
    p_iter.forth
end
p_iter.close
print("" + i.out + " Person(s) with 'comment' containing " + search_string + "\n")

print("Solution 2\n")
-- Solution 2: Create an iterator from the EP dict with the MTOBJECT
-- base type
o_iter := epdictCls.iterator (search_string, Void,
{MT_DATABASE}.Mt_Max_Prefetching)
i := 0
from
    o_iter.start
until
    o_iter.exhausted
loop
    p ?= o_iter.item
    if (p /= Void) then
        print("   " + p.firstname + " " + p.lastname + "\n")
i := i + 1
Entry-Point Dictionary Lookup Count

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

```plaintext
local
    db: MT_DATABASE
    i: INTEGER
    search_string: STRING
    epdictCls: MTENTRYPOINTDICTIONARY

do
    epdictCls := db.get_mtentrypointdictionary({PERSON}.commentdict_name);
    search_string := "knees"
    print("%NLooking for Persons with " + search_string + " in the 'comment' text\n")
    i := epdictCls.object_number (search_string, Void)
    print("" + i.out + " Person(s) with 'comment' containing " + search_string + ">
    end
```

7 Working with SQL

Running the Examples on SQL

This sample program demonstrates how to manipulate objects via the Matisse Eiffel SQL interface. It creates objects (Person Employee and Manager) and it executes SELECT statements to retrieve objects. It also shows how to create SQL methods and execute them.

1. Follow the instructions in Before Running the Examples on page 5.
2. Change to the sql directory in your installation (under examples).
3. Load examples.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on ‘Schema->Import ODL Schema’, then select sql/examples.odl for this demo.
4. Generate Eiffel class files:
   
   ```
   mt_sdl stubgen --lang eiffel -f examples.odl
   ```
5. Open the Eiffel project for instance sql.ecf in Eiffel Studio and compile it.
6. In Eiffel Studio or in a command line windows run the built application.

Executing a SQL Statement

After you open a connection to a Matisse database, you can execute statements (i.e., SQL statements or SQL methods) using a MT_STATEMENT object. You can create a statement object for a specific MT_DATABASE object using the create_statement method.

You can create more specific Statement objects for different purposes:

- **MT_STATEMENT** - It is specifically used for the SQL statements where you don't need to pass any value as a parameter
- **MT_PREPARED_STATEMENT** - It is a subclass of the statement class. The main difference is that, unlike the statement class, prepared statement is complied and optimized once and can be used multiple times by setting different parameter values.
- **MT_CALLABLE_STATEMENT** - It provides a way to call a stored procedure on the server from a Eiffel program. Callable statements also need to be prepared first, and then their parameters are set using the set methods.
- **MT_RESULT_SET** - It represents a table of data, which is usually generated by executing a statement that queries the database. A ResultSet object maintains a cursor pointing to its current row of data.

**NOTE:** With the Matisse Eiffel SQL interface you usually don’t need to use the Eiffel stub classes unless you want to retrieve objects from a SQL statement or from the execution of a SQL method.
Creating Objects

You can also create objects into the database without the Eiffel stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```java
// Create an instance of Statement
stmt := db.create_statement()

// Set the relationship 'Spouse' between these two Person objects
commandText := "SELECT REF(p) FROM Person p WHERE FirstName = 'James' AND LastName = 'Watson' INTO p1;"
stmt->execute(commandText)
commandText := "UPDATE Person SET Spouse = p1 WHERE FirstName = 'Elizabeth' AND LastName = 'Watson';"
inserted := $stmt.execute_update(commandText)

// Clean up
stmt.close()
```

Updating Objects

You can also create objects into the database without the Eiffel stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```java
// Create an instance of Statement
stmt := db.create_statement()

// Set the relationship 'Spouse' between these two Person objects
commandText := "SELECT REF(p) FROM Person p WHERE FirstName = 'James' AND LastName = 'Watson' INTO p1;"
stmt->execute(commandText)
commandText := "UPDATE Person SET Spouse = p1 WHERE FirstName = 'Elizabeth' AND LastName = 'Watson';"
inserted := $stmt.execute_update(commandText)

// Clean up
stmt.close()
```
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.

```java
cmd_text := "SELECT FirstName, LastName, Spouse.FirstName AS Spouse, Age FROM Person
WHERE LastName = ? LIMIT 10;"
pstmt := db.prepare_statement (cmd_text)

-- Set parameters
pstmt.set_string(1, "Watson")
print ("Executing: " + pstmt.stmt_text() + "%N")
result_set := pstmt.execute_query ()
colnum := result_set.column_count ()

print ("Total selected: " + result_set.total_num_objects().out + "%N")
print ("Total qualified: " + result_set.total_num_qualified().out + "%N")
print ("Total columns: " + colnum.out +"%N")

-- List column names
from
  i := 1
until
  i > colnum
loop
  print(result_set.column_name(i) + "      ")
  i := i + 1
end
rint("%N");
print("---%N");

-- List rows
from
  result_set.start
until
  result_set.exhausted
loop
  fname := result_set.get_string(1)
  lname := result_set.get_string(2)
  sfname := result_set.get_string(3)
  if not result_set.is_null(4) then
    age := result_set.get_integer(4).out
  else
    age := "NULL"
  end
  print(fname + " , " + lname + " , " + sfname + " , " + age + "%N");

  result_set.forth
end
```
Retrieving Objects from a SELECT statement

You can retrieve Eiffel 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.

```java
-- create SQL statement
stmt := db.create_statement()
query := "SELECT REF(p) FROM Person p WHERE LastName = 'Watson';"
result_set := stmt.execute_query(query)

print ("Total selected:  " + result_set.total_num_objects().out + "%N")
print ("Total qualified: " + result_set.total_num_qualified().out + "%N")
print ("Total columns:   " + result_set.column_count().out +"%N")

from
result_set.start
until
result_set.exhausted
loop
  p := result_set.get_object(1)
  print("  " + p.mtclass().mtname() + " " + p.firstname() + " " + p.lastname() + " married to " + p.spouse().firstname() + "%N");

result_set.forth
end
result_set.close()
stmt.close()
```

Retrieving Objects from a Block Statement

You can also retrieve a collection of Eiffel objects directly from the database by executing a SQL block statement.

The `getObject` method defined on a `MT_CALLABLE_STATEMENT` is used to return one object as well as an object collection. The following code example shows how to retrieve a collection of `Person` objects from a `MT_CALLABLE_STATEMENT`.
Executing DDL Statements

You can also create schema objects from a Eiffel application via SQL.

Creating a Class

You can create schema objects using the `executeUpdate` method as long as the transaction is started in the **DATA DEFINITION** mode.

```eiffel
create db.make(host, dbname)
db.open()
-- In order to execute DDL statements, the transaction needs to be
-- started in the "Data Definition" mode

db.set_data_access_mode({MT_DATABASE}.Mt_Data_Definition)
db.start_transaction()
-- Execute the DDL statement
stmt := db.create_statement ()
stmt.execute_update ("CREATE CLASS Manager UNDER Employee (bonus INTEGER)")
stmt.close ()
db.commit()
```

Creating a SQL Method

Creating a schema object using the `execute` method does not require to start a transaction. A transaction will be automatically started in the **DATA DEFINITION** mode.

```eiffel
create db.make(host, dbname)
db.open()
stmt := db.create_statement ()

-- The first method returns the number of Person objects which have a specified last name
commandText :=
"CREATE STATIC METHOD CountByLName(lname STRING)
  RETURNS INTEGER
  FOR Person
  BEGIN
    DECLARE cnt INTEGER;
    SELECT COUNT(*) INTO cnt FROM Person WHERE LastName = lname;
    RETURN cnt;
  END;"

stmt.execute (commandText)
stmt.close ()
db.commit()
```

Executing SQL Methods

You can call a SQL method using the `CALL` syntax, i.e., simply passing the SQL method name followed by its arguments as an SQL statement. You can also use the Callable Statement object, which allows you to explicitly specify the method’s parameters.
Executing a Method returning a Value

The following program code shows how to call the SQL method CountByLName of the Person class.

```java
-- Specify the stored method. we call a static method,
-- the name is consisted of class name and method name.
-- Use CALL syntax to call the method
commandText := "CALL Person::CountByLName(?)";

-- Create an instance of CallableStatement
stmt = db.prepare_call(commandText)

-- Set parameters
stmt.set_string(1, "Watson");

-- Execute the stored method
stmt.execute()

-- Get the returned value
count := stmt.get_integer(0)

-- Print it
print (count.out "objects found")

-- Clean up
stmt.close()
```

Executing a Method returning an Object

The following program code shows how to call the SQL method FindByName of the Person class.

```java
-- Specify the SQL method. Since we call a static method,
-- the name is consisted of class name and method name.
-- Use CALL syntax to call the method
commandText := "CALL Person::FindByName('Watson', 'James');"

-- Create an instance of CallableStatement
stmt := db.prepare_call(commandText)

-- Execute the stored method
stmt.execute()

-- Get the returned value
p ?= stmt.get_object(0)

// Print it
if p /= Void then
    print("Found: " + p.lastname() + " " + p.firstname() + "%N")
else
    print("no matching object found%N")

-- Clean up
stmt.close()
```
Catching a Method Execution Error

The following program code shows how to retrieve the execution stack trace of a SQL method when an error occurs.

Deleting Objects

You can delete objects from the database with a DELETE statement as follows:

```effel
db.start_transaction({MT_DATABASE}.Mt_Min_Tran_Priority)

print("%NDelleting all Persons...%N");
-- execute an update statement
query := "DELETE FROM Person"
cnt := stmt.execute_update (query)
stmt_type := stmt.statement_type ()
print("   " + stmt.stmt_type_to_string(stmt_type) + "' statement executed affecting " + cnt.out + " objects in the database.%N");

stmt.close()

db.commit()
```
8 Working with Class Reflection

This section illustrates Matisse Reflection mechanism. This example shows how to manipulate persistent objects without having to create the corresponding Eiffel stubclass. It also presents how to discover all the object properties.

Running the Examples on Reflection

This example creates several objects, then manipulates them to illustrate Matisse Reflection mechanism.

1. Follow the instructions in Before Running the Examples on page 5.
2. Change to the reflection directory (under examples).
3. Load examples.odl into the database. From the Enterprise Manager, select the database ‘example’ and right click on ‘Schema->Import ODL Schema’, then select reflection/examples.odl for this demo.
4. Open the Eiffel project for instance reflection.ecf in Eiffel Studio and compile it.
5. In Eiffel Studio or in a command line windows run the built application.

Creating Objects

This example shows how to create persistent objects without the corresponding Eiffel stubclass. The method get_mt<xyz>() defined on all Matisse Meta-Schema classes (i.e. MTCLASS, MTATTRIBUTE, etc.) allows you to access to the schema descriptor necessary to create objects. Each object is an instance of the MTOBJECT base class. The MTOBJECT class holds all the methods to update the object properties (attribute and relationships (i.e. set_string(), set_successors(), etc.).

```eiffel
local
  factory: MT_CORE_OBJECT_FACTORY
  db: MT_DATABASE
  personCls, employeeCls, managerCls: MTCLASS
  fnAtt, lnAtt, cgAtt, hdAtt, slAtt: MTATTRIBUTE
  tmRshp: MRELATIONSHIP
  p, e, m: MOBJECT
  salary: DECIMAL
  hiredate: DATE
  tmbrs: ARRAY[MOBJECT]
do
  -- Use the MT_CORE_OBJECT_FACTORY since there is need for
  -- dynamic object creation, it is all MOBJECT
  create factory.make
  create db.make_factory(host, dbname, factory)
  db.open()
  db.start_transaction({MT_DATABASE}.Mt_Min_Tran_Priority)

  personCls := db.get_mtclass("Person");
  employeeCls := db.get_mtclass("Employee");
```
managerCls := db.get_mtclass("Manager");
fnAtt := db.get_mtattribute("FirstName", personCls)
lnAtt := db.get_mtattribute("LastName", personCls)
cgAtt := db.get_mtattribute("collegeGrad", personCls)
hdAtt := db.get_mtattribute("hireDate", employeeCls)
slAtt := db.get_mtattribute("salary", employeeCls)
tmRshp := db.get_mtrelationship("team", managerCls)
print("%NCreating one Person...%N")
create p.make_from_mtclass (personCls)
p.set_string(fnAtt, "John");
p.set_string(lnAtt, "Smith")
p.set_boolean(cgAtt, False)

create e.make_from_mtclass (employeeCls)
e.set_string(fnAtt, "James");
e.set_string(lnAtt, "Roberts")
e.set_boolean(cgAtt, True)
create salary.make_from_string ("5123.25")
e.set_numeric(slAtt, salary)
create hiredate.make (2009,09,09)
e.set_date(hdAtt, hiredate)

create m.make_from_mtclass (managerCls)
m.set_string(fnAtt, "Andy");
m.set_string(lnAtt, "Brown")
m.set_boolean(cgAtt, True)
create salary.make_from_string ("5123.25")
m.set_numeric(slAtt, salary)
create hiredate.make (2008,08,08)
m.set_date(hdAtt, hiredate)
create tmbrs.make(1,2)
tmbrs.put(m, 1)
tmbrs.put(e, 2)
m.set_successors(tmRshp, tmbrs)

db.commit()
db.close()

Listing Objects

This example shows how to list persistent objects without the corresponding Eiffel stubclass. The instance_iterator() method defined on the MTCLASS object allows you to access all instances defined on the class.

local

  factory: MT_CORE_OBJECT_FACTORY
  db: MT_DATABASE
  cnt: INTEGER
  personCls: MTCLASS
  fnAtt, lnAtt, cgAtt: MTATTRIBUTE
  o_iter: MT_OBJECT_ITERATOR[MTOBJECT]
  p: MTOBJECT

do
  -- Use the MT_CORE_OBJECT_FACTORY since there is need for
-- dynamic object creation, it is all MTOBJECT
create factory
create db.make_factory(host, dbname, factory)
db.open()

db.start_version_access(Void)

personCls := db.get_mtclass("Person");
fnAtt := db.get_mtattribute("FirstName", personCls)
lnAtt := db.get_mtattribute("LastName", personCls)
cgAtt := db.get_mtattribute("collegeGrad", personCls)

cnt := personCls.instance_number()
print("%N" + cnt.out + " Person(s) in the database.%N")

o_iter := personCls.instance_iterator({MT_DATABASE}.Mt_Max_Prefetching)
from
    o_iter.start
until
    o_iter.exhausted
loop
    p := o_iter.item
    print("   " + p.mtclass.mtname + " #" + p.oid.out)
    print(" - " + p.get_string(fnAtt) + " " + p.get_string(lnAtt))
    print(" collegeGrad="+ p.get_boolean(cgAtt).out + "%N")

    o_iter.forth
end

o_iter.close

db.end_version_access()

db.close()

\section{Working with Indexes}

This example shows how to retrieve persistent objects from an index. The \texttt{MTINDEX} class holds all the methods retrieves objects from an index key.

personCls := db.get_mtclass("Person");
fnAtt := db.get_mtattribute("FirstName", personCls)
lnAtt := db.get_mtattribute("LastName", personCls)
cgAtt := db.get_mtattribute("collegeGrad", personCls)

idxCls := db.get_mtindex("personName");

-- Get the number of entries in the index
i := idxCls.index_entries_number()

print("" + i.out + " entries in the index%N")

f_name := "James"
l_name := "Roberts"
print("%NLooking for Person '" + f_name + " " + l_name + "'"%N")
-- Create the key
create key.make (1, 2)
key.put (l_name, 1)
key.put (f_name, 2)

-- lookup for the number of objects matching the key
i := idxCls.object_number (key, Void)
print(" " + i.out + " objects retrieved\n")

if i > 1 then
  o_iter := idxCls.iterator (key, key, Void, (MTINDEX).Mt_Direct,
  (MT_DATABASE).Mt_Max_Prefetching)
i := 0
from
  o_iter.start
until
  o_iter.exhausted
loop
  p := o_iter.item
  print(" found " + p.mtclass.mtname + " #" + p.oid.out)
  print(" - " + p.get_string(fnAtt) + " " + p.get_string(lnAtt))
  print(" collegeGrad=" + p.get_boolean(cgAtt).out + "\n")
  i := i + 1
from
  o_iter.forth
end
  o_iter.close
print("" + i.out + " Person(s) found\n")
else
  p := idxCls.lookup (key, Void)
  if (p /= Void) then
    print(" found exactly one Person: " + p.get_string(fnAtt) + " " +
p.get_string(lnAtt) + "\n")
  else
    print(" nobody found\n")
end

Working with Entry Point Dictionaries

This example shows how to retrieve persistent objects from an Entry Point Dictionary. The
MTENTRYPOINTDICTIONARY class holds the methods to retrieve objects from a string key.

personCls := db.get_mtclass ("Person");
fnAtt := db.get_mtattribute ("FirstName", personCls)
lnAtt := db.get_mtattribute ("LastName", personCls)
cgAtt := db.get_mtattribute ("collegeGrad", personCls)
epdictCls := db.get_mtentrypointdictionary ("collegeGradDict");
search_string := "True"
print("%NLooking for Persons with CollegeGrad=" + search_string + "%N")

i := epdictCls.object_number (search_string, Void)

if i > 1 then
  o_iter := epdictCls.iterator (search_string, Void, (MT_DATABASE).Mt_Max_Prefetching)
i := 0
  from
  o_iter.start
  until
  o_iter.exhausted
  loop
    p ?= o_iter.item

    print(" found exactly one Person:" + " #" + p.oid.out)
    print(" - " + p.get_string(fnAtt) + " " + p.get_string(lnAtt))
    print(" collegeGrad="+ p.get_boolean(cgAtt).out + "%N")
i := i + 1

  o_iter.forth
end
  o_iter.close
print("" + i.out + " Person(s) with collegeGrad=" + search_string + "%N")
else
    p := epdictCls.lookup (search_string, Void)
    if (p /= Void) then
        print(" found exactly one Person:" + " #" + p.oid.out)
        print(" - " + p.get_string(fnAtt) + " " + p.get_string(lnAtt))
        print(" collegeGrad="+ p.get_boolean(cgAtt).out + "%N")
    else
        print(" nobody found%N")
    end
end

Discovering Object Properties

This example shows how to list the properties directly from an object. The MTOBJECT class holds the attributes_iterator() method, relationships_iterator() method and inverse_relationships_iterator() method which enumerate the object properties.

personCls := db.get_mtclass("Person");
o_iter := personCls.instance_iterator ({MT_DATABASE}.Mt_Max_Prefetching)
  from
  o_iter.start
  until
  o_iter.exhausted
  loop
    p := o_iter.item

    print("- " + p.mtclass.mtname + " #" + p.oid.out + "%N")
print(" Attributes:%N")
att_iter := p.attributes_iterator()
from

att_iter.start
until
att_iter.exhausted
loop
  att := att_iter.item
  att_type := att.get_mtype()
  val_type := p.get_type(att)
  print("  " + att.mtname + "(type=" + att_type.out + ": ")
  if val_type = {MTTYPE}.Mt_Null then
    print("MT_NULL")
  else
    print(p.get_value(att).out)
  end
  print(" (valtype=" + val_type.out + ")")
  print("%N")
  att_iter.forth
end
att_iter.close

print("  Relationships:%N")
rel_iter := p.relationships_iterator()
from
  rel_iter.start
until
  rel_iter.exhausted
loop
  rel := rel_iter.item
  print("  " + rel.mtname + ": " + p.get_successor_size(rel).out + "%N")
  rel_iter.forth
end
rel_iter.close

print("  Inverse Relationships:%N")
rel_iter := p.inverse_relationships_iterator()
from
  rel_iter.start
until
  rel_iter.exhausted
loop
  rel := rel_iter.item
  print("  " + rel.mtname + ": " + p.get_successor_size(rel).out + "%N")
  rel_iter.forth
end
rel_iter.close

o_iter.forth
end
o_iter.close
Adding Classes

This example shows how to add a new class to the database schema. The connection needs to be open in the DDL ([MT_DATABASE].Mt_Data_Definition) mode. Then you need to create instances of MTCLASS, MTATTRIBUTE and MTRELATIONSHIP and connect them together.

```plaintext
-- open connection in DDL mode
db.set_data_access_mode([MT_DATABASE].Mt_Data_Definition)
db.open()

db.start_transaction([MT_DATABASE].Mt_Min_Tran_Priority)

personCls := db.get_mtclass("Person");

-- Create new attributes
create cAtt.makeMtAttribute(db, "City", {MTTYPE}.Mt_String)
create pcAtt.makeMtAttribute(db, "PostalCode", {MTTYPE}.Mt_String)

-- Create a new Class
create attrs.make(1, 2)
attrs.force(cAtt, 1)
attrs.force(pcAtt, 2)
create addrCls.makeMtClassFull(db, "PostalAddress", attrs, Void)

-- List the new PostalAddress class to Person
create ad_rel.makeMtRelationship(db, "Address", addrCls, 0, 1)
personCls.addclsMtRelationship(ad_rel);

db.commit()
```

Deleting Objects

This example shows how to delete persistent objects. The MTOBJECT class holds remove() and deep_remove(). Note that on MTOBJECT.deep_remove() does not execute any cascading delete but only calls remove().

```plaintext
db.start_transaction([MT_DATABASE].Mt_Min_Tran_Priority)

personCls := db.get_mtclass("Person");

cnt := personCls.instance_number()
print("%N + cnt.out + " Person(s) in the database.%N"

print("%N Deleting all Persons...%N")
personCls.remove_all_instances()

cnt := personCls.instance_number()
print("%N + cnt.out + " Person(s) in the database.%N")

db.commit()
```
Removing Classes

This example shows how to remove a class for the database schema. The `deep_remove()` method defined on `MTCLASS` will delete the class and its properties and indexes. The connection needs to be open in `{MT_DATABASE}.Mt_Data_Definition` mode.

```effel
-- open connection in DDL mode
db.set_data_access_mode({MT_DATABASE}.Mt_Data_Definition)

db.open()

db.start_transaction({MT_DATABASE}.Mt_Min_Tran_Priority)

addrCls := db.get_mtclass("PostalAddress");

addrCls.deep_remove()

db.commit()
```
9 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 the Events Example

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 5.
2. Change to the events directory (under examples).
3. Open the Eiffel project for instance events.ecf in Eiffel Studio and compile it.
4. In Eiffel Studio or in a command line windows run the built application.

Events Subscription

This section illustrates event registration and event handling. Matisse provides the MT_EVENT class to manage database events. You can subscribe up to 32 events (MT_EVENT.Mt_Event1 to MT_EVENT.Mt_Event32) and then wait for the events to be triggered.

```eiffel
Temperature_ChangesEvt := MT_EVENT.Mt_Event1
Rainfall_ChangesEvt := MT_EVENT.Mt_Event2
Humidity_ChangesEvt := MT_EVENT.Mt_Event3
Windspeed_ChangesEvt := MT_EVENT.Mt_Event4

create db.make(host, dbname)
db.open()

create subscriber.make (db)

-- Subscribe to all 4 events
event_set := Temperature_ChangesEvt
event_set := event_set + Rainfall_ChangesEvt
event_set := event_set + Humidity_ChangesEvt
event_set := event_set + Windspeed_ChangesEvt

-- Subscribe
subscriber.subscribe(event_set)

-- Wait 1000 ms for events to be triggered
-- return 0 if not event is triggered until the timeout is reached
triggered_events := subscriber.wait(1000)
if triggered_events /= 0 then
    print ("Events triggered: " + triggered_events.out + ":%N")
else
    print ("No events triggered%N")
end
```
-- Unsubscribe to all 4 events
subscriber.unsubscribe()

db.close()

Events Notification

This section illustrates event selection and notification.

Temperature_ChangesEvt := {MT_EVENT}.Mt_Event1
Rainfall_ChangesEvt := {MT_EVENT}.Mt_Event2
Humidity_ChangesEvt := {MT_EVENT}.Mt_Event3
Windspeed_ChangesEvt := {MT_EVENT}.Mt_Event4

create db.make(host, dbname)
db.open ()

create notifier.make (db)

event_set := Temperature_ChangesEvt
event_set := event_set + Windspeed_ChangesEvt

-- Notify of some events
notifier.notify(event_set);

db.close ()

More about MT_EVENT

As illustrated by the previous sections, the MT_EVENT class provides all the methods for managing database events. The reference documentation for the MT_EVENT class is included in the Matisse Eiffel Binding API documentation located from the Matisse installation root directory in docs/eiffel/api/index.html.
10 Object Factories

You can generate your Eiffel class stubs with the \texttt{mt\_sd}l utility as follows:

\begin{verbatim}
mt.sd stubgen --lang eiffel -f examples.odl
\end{verbatim}

When your persistent classes are defined, you need a factory that is able to create an Eiffel object from a schema class name defined in the database. The \texttt{MT\_DYNAMIC\_OBJECT\_FACTORY} class does it for you and this is the default factory.

Connection with a Factory

Using \texttt{MT\_CORE\_OBJECT\_FACTORY}

This factory is the basic \texttt{MTOBJECT} -based object factory. This factory is the most appropriate for application which does not use generated stubs. This factory is faster than the default Object Factory used by \texttt{MT\_DATABASE} since it doesn't use reflection to build objects.

\begin{verbatim}
local
  factory: MT\_CORE\_OBJECT\_FACTORY
  db: MT\_DATABASE
do
  create factory.make
  create db.make\_factory(host, dbname, factory)
end
\end{verbatim}

Creating your Object Factory

Implementing the \texttt{MT\_OBJECT\_FACTORY} abstract class

The \texttt{MT\_OBJECT\_FACTORY} interface describes the mechanism used by \texttt{MT\_DATABASE} to create the appropriate Eiffel object for each Matisse object. Implementing the \texttt{MT\_OBJECT\_FACTORY} abstract class requires to define the \texttt{get\_eiffel\_class()} method which return Eiffel class identifier corresponding to a Matisse Class Name, the \texttt{get\_database\_class()} method which return Matisse class name corresponding to the Eiffel class name and the \texttt{get\_object\_instance()} method which return an Eiffel object based on an oid.

\begin{verbatim}
defered class
  MT\_OBJECT\_FACTORY

feature -- Abstract Methods
  get\_eiffel\_class (mtcls\_name: STRING): INTEGER deferred end
  get\_database\_class (mtcls\_name: STRING): STRING deferred end
  get\_object\_instance (db: MT\_DATABASE; mt\_oid : INTEGER): MTOBJECT deferred end
end
\end{verbatim}
11 Building your Application

This section describes the process for building an application from scratch with the Matisse Eiffel binding.

Discovering the Matisse Eiffel Classes

The Matisse Eiffel binding is comprised of 2 elements:

1. **matisse** cluster contains all the core classes. 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 **matisse\reflect** as well as all the SQL-related classes defined in the **matisse\sql**. These classes manages the execution of all types of SQL statements.

2. **matisseEIFFEL** library bridges Matisse client library and Eiffel

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

Generating Stub Classes

The Eiffel 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 Eiffel stub classes is a 2 steps process:

1. Design a database schema using ODL (Object Definition Language).
2. Generate the Eiffel code from the ODL file:

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

A .e file will be created for each class defined in the database. 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
```
Appendix A: Generated Public Methods

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

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.

Sample constructor

```e
make_person (a_db: MT_DATABASE)
```

For all attributes

The following methods are created for each attribute. For example, if the ODL definition for class Check contains the attributes Date and Amount, the Check.Eiffel file will contain the methods getDate and getAmount. These examples are taken from Person.firstName.

- Get value: `firstname, get_firstname () : STRING`
- Set value: `set_firstname (a_val: STRING)`
- Remove value: `remove_firstname ()`
- Check Null value: `is_firstname_null () : BOOLEAN`
- Check Default value: `is_firstname_default_value () : BOOLEAN`
- Get descriptor: `get_firstname_attribute () : MTATTRIBUTE`

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

For list-type attributes only

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

- Get elements: `get_photo_elements (buffer: ARRAY [NATURAL_8]; count, offset: INTEGER) : INTEGER`
- Set elements: `set_photo_elements (buffer: ARRAY [NATURAL_8]; buffer_size, offset: INTEGER; discard_after: BOOLEAN)`
- Count elements: `get_photo_size ()`

For all relationships

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

- Clear successors: `clear_spouse ()`
Get descriptor  
get_spouse_relationship (): MTRELATIONSHIP

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

For relationships where the maximum cardinality is 1

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

Get successor  
assistant, get_assistant (): EMPLOYEE

Set successor  
set_assistant (succ: EMPLOYEE)

For relationships where the maximum cardinality is greater than 1

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

Get successors  
team, get_team (): ARRAY[EMPLOYEE]

Open an iterator  
team_iterator (): MT_OBJECT_ITERATOR[EMPLOYEE]

Count successors  
team_size, get_team_size (): INTEGER

Set successors  
set_team (succs: ARRAY[EMPLOYEE])

Add successors  
Insert one successor before any existing successors:  
prepend_team (succ: EMPLOYEE)

Add one successor after any existing successors:  
append_team (succ: EMPLOYEE)

Add one successor after one specific successor:  
append_after_team (succ, after: EMPLOYEE)

Add multiple successors after any existing successors:  
append_num_team (succs: ARRAY[EMPLOYEE])

Remove successors  
remove_team (succ: EMPLOYEE)

Remove specified successors.

For indexes

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

Name  
personname_name: STRING = "personName"
For entry-point dictionaries

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

Name: commentdict_name: STRING = "commentDict"