

Matisse® Eiffel Programmer's Guide

6th Edition

July 2006



MATISSE Eiffel Programmer's Guide

Copyright ©1992–2006 Matisse Software Inc. All Rights Reserved.

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

Printed in USA.

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)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

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

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

PDF generated 10 July 2006

1	Introduction	4
1.1	Scope of This Document	4
1.2	Before Reading This Document	4
1.3	Concepts and Features	4
1.4	Additional Documentation	5
2	Connecting	6
3	Retrieving Persistent Objects	7
3.1	MT_STORABLE	7
3.2	Retrieving objects	7
3.3	Accessing Objects	10
3.4	Class Invariants	13
3.5	Attribute Types	15
3.6	Collection Classes for Successors	16
3.7	Caching Objects	18
4	Modifying Persistent Objects	19
4.1	Setting Attribute Values	19
4.2	Setting Relationship Successors	19
4.3	Removing Attribute Values	20
4.4	Deleting Objects	20
5	Creating Persistent Objects	21
5.1	Automatic Promotion	21
5.2	Explicit Promotion	21
6	Executing SQL Statements	22
6.1	MT_STATEMENT	22
6.2	MT_RESULT_SET	23
	Appendix A: Sample Application Schema	26
	Appendix B: mt_odl	27
	Appendix C: Troubleshooting	28
	Error Messages	28
	Other Problems	29
	Appendix D: Custom Collection Classes	30
	Defining a New Collection Class	30
	Specifying a Collection Class for Relationships	32
	Appendix E: Class Hierarchies and Client Relations	34

1 Introduction

The Matisse-Eiffel binding provides a persistent system in which you can build an Eiffel application reading, creating and storing Matisse objects with the minimum possible effort. This document describes the Matisse-Eiffel binding and gives the basic idea of the way to use it.

1.1 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.

1.2 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*.

1.3 Concepts and Features

Convenience vs. optimizability For any database binding, there is a trade-off between convenience and optimizability. It is convenient to make storage and retrieval of persistent objects completely automatic, but for best performance a developer needs to be able to optimize an application by specifying in detail which objects are retrieved and stored, and when. This binding strikes a balance between those two goals.

Replica objects When a Matisse object is read, a “replica object” is created dynamically in Eiffel. (In the future, we will provide a proxy mechanism to access Matisse objects directly.)

Schema During a session, the binding reads the Matisse meta-schema and the application schema. These schemas are used to retrieve and store persistent objects efficiently. (No specific Eiffel type information is stored in the database)

Persistence not dependent on Eiffel collection classes	Matisse does not require a class library corresponding to the Eiffel collection classes (such as LIST, ARRAY and TABLE) in order to make Eiffel objects persistent (that is, to store them in a database). See <i>Collection Classes for Successors</i> on page 16 for more details.
SQL	SQL statements can be executed. The results are returned either as objects or as values (such as integers or strings).
Access to multiple databases	The binding has an Eiffel class whose instances represent database sessions. These instances manage the database schema and caching of persistent objects. This enables access to multiple databases from a single application program; these databases can have the same database schema and/or class names.
Access from multiple applications	Multiple application programs can access the same Matisse database simultaneously. These applications may use any of the Matisse language bindings or interfaces (C, C++, Eiffel, Java, JDBC, ODBC, Perl, PHP, Python, SQL, and XML).
High performance	Persistent objects are retrieved only when needed by an application, and are cached in the Eiffel environment. Only updated attributes and relationships are propagated to the database. There is no extra work to be done at transaction commit.
Transparent persistence	If a transient Eiffel object participates in a relationship with a persistent object in the Eiffel environment, the transient object becomes persistent automatically. This is called "persistence by reachability." All updates made on persistent objects are automatically propagated to the database. In addition, you can explicitly promote a transient object into a persistent object by using the procedure "persist."
Evolutive development cycle.	Matisse includes utilities for automatically generating Eiffel classes for a database schema defined in the Matisse ODL language or with Rational Rose, and updating those classes as the schema evolves. For details, see <i>Getting Started with Matisse</i> .

1.4 Additional Documentation

Getting Started with Matisse, installation guides, a server administration guide, and other documentation are available on our Web site:

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

2 Connecting

To retrieve persistent objects from the database, you must first establish a database connection and open a transaction or a version access (read-only transaction). The following is typical code to establish such a connection.

```
start_my_appliation is
  local
    db_session : MATISSE_APPL
      -- Representing a database session
  do
    create db_session.make("hostname","dbname")
      -- Setting a host name of the database server and
      -- database name.

    db_session.connect
      -- Connecting to the database
    db_session.start_transaction
      -- Start a new transaction

    ... -- Your application source code here

    db_session.commit_transaction
      -- Committing the transaction
    db_session.disconnect
      -- Disconnecting from the database
  end
```

NOTE: Most of the other example programs in this document assume that a database connection has been already established and a transaction or version access opened.

3 Retrieving Persistent Objects

After establishing a database connection and opening a transaction or version access, you can retrieve objects from the database using indexes, entry-point dictionaries, SQL statements, or class names. Once you have retrieved objects, you may retrieve other objects using relationships.

An Eiffel object is created corresponding to each retrieved Matisse object; we call these “replica objects.” Replica objects are cached in the Eiffel environment.

The database schema (see [Sample Application Schema](#) on page 26) and example code below are taken from the sample application included with the Eiffel binding.

3.1 MT_STORABLE

Persistent objects inherit from the class `MT_STORABLE`. Since instances of the classes `BOOK`, `PUBLISHER`, and `AUTHOR` are to be persistent, these classes inherit from `MT_STORABLE`. For example, from the definition of class `BOOK`:

```
class
  BOOK

inherit
  MT_STORABLE

feature
  .....
end -- class BOOK
```

3.2 Retrieving objects

Retrieving objects using entry-point dictionaries

Using an entry-point dictionary allows you to retrieve objects based on the results of a keyword search. An entry-point dictionary is a B-tree of keywords derived from the values of a string attribute. For details, see *Getting Started with Matisse*.

To retrieve objects using an entry-point dictionary, create an instance `ep` of class `MtEntryPointDictionary` (we recommend using a mixture of lower case and upper case for greater readability) with two arguments, an entry-point dictionary name and a class name. The class name is used to filter the selected objects by the class, that is, only instances of that class (including subclasses) will be retrieved. The class name can be `Void`. Then call the function `retrieve_objects` on the object `ep` with a single argument, entry-point value. The function returns an array of objects accessed through the entry point.

The following sample function returns an array of objects of class `BOOK`:

```
find_books_from_title(book_title: STRING) : ARRAY[MT_STORABLE] is
  local
    ep: MTENTRYPOINTDICTIONARY
```

```

do
  create ep.make_from_name("BookTitleDictionary", Void)
  Result ?= ep.retrieve_objects(book_title)
end

```

Retrieving objects using indexes

Using an index allows you to retrieve objects as an ordered set whose property values are within bounds. A Matisse index is an ordered table with one to four columns corresponding to attributes in a schema class, and a row for each of the class's objects to hold the corresponding attribute values. For details, see *Getting Started with Matisse*.

To use an index, first create an instance of class `MTINDEX` with one argument, the index name. Then specify a start value and an end value for each criterion. After specifying these values, call the function `open_stream` on the instance; this returns a stream object, an instance of class `MT_STREAM`. You can retrieve objects through this stream object.

The following sample procedure prints all the objects of class `BOOK` whose titles are between a start value and an end value specified by the arguments.

```

find_and_print_books_using_index(start_v, end_v: STRING) is
  local
    an_index: MT_INDEX
    index_stream: MT_STREAM
    criterion: MT_INDEX_CRITERION
  do
    create an_index.make("BookTitleIndex")
    criterion := an_index.criteria.item(1)
    criterion.set_start_value(start_v)
    criterion.set_end_value(end_v)
    index_stream := an_index.open_stream
  from
    index_stream.start
  until
    index_stream.exhausted
  loop
    print(index_stream.next_object)
    index_stream.forth
  end
  index_stream.close
end
end

```

Retrieving objects using SQL statements

You can retrieve objects using a Matisse SQL `SELECT` statement into a result set object, and then retrieve each object in the result set. Matisse SQL is a subset of SQL 99, with enhancements to support unique features such as direct navigation of relationships; see the *Matisse SQL Programmer's Guide* for details.

In order to execute a SQL `SELECT` statement, first create an instance of `MT_STATEMENT`. Then call the function `execute_query` with the SQL statement string. This returns a result set object, an instance of class `MT_RESULT_SET`. The result set object allows you to retrieve objects.

The following procedure executes an SQL statement and prints selected objects.

```

exec_a_sql_query is

```

```

local
  stmt: MT_STATEMENT
  result_set: MT_RESULT_SET
do
  stmt := db_session.create_statement
  -- Create a statement
  -- db_session is an instance of MATISSE_APPL

  result_set := stmt.execute_query ("SELECT REF(p) FROM PersonP")
  -- Execute an SQL statement and return a result set.

  from
    result_set.start
  until
    result_set.exhausted
  loop
    print (result_set.get_object(1))
    result_set.forth
  end

  result_set.close
  stmt.close
  -- Free the external resources related to those objects
end

```

Retrieving objects using class names

You may retrieve all the instances of a class (and any subclasses) through its name. First, create an instance of class `MTCLASS` with one argument, a class name. Then call either the function `all_instances` on the instance, which returns an array containing all the instances of the class; or the function `open_stream`, which returns a stream you can use to iterate through the instances.

The following sample procedure prints all the instances of a class specified by the argument.

```

print_all_instances_of_class(a_class_name: STRING) is
  local
    a_class: MTCLASS
    all_instances: ARRAY[MT_STORABLE]
    i: INTEGER
  do
    create a_class.make_from_name(a_class_name)
    all_instances := a_class.all_instances
    from
      i := all_instances.lower
    until
      i > all_instances.upper
    loop
      print(all_instances.item(i))
      i := i + 1
    end
  end
end

```

3.3 Accessing Objects

When a replica object is created, it contains neither attribute values nor relationship successors (see *Getting Started with Matisse* for discussion of these and other Matisse components). You may load them either transparently or explicitly, as shown below.

Accesses through transparent loading

Transparent loading uses functions in the Eiffel classes generated by the `mt_odl` utility (see [Appendix B: `mt_odl`](#) on page 27). The names of these functions are derived from the attribute or relationship name. The following is from the code generated from the ODL in [Appendix A: *Sample Application Schema*](#) on page 26.

```
class
    BOOK
    ... Some code here ...

feature -- Access
    title: STRING is
        do
            if is_obsolete or else title_ = Void then
                title_ := mt_get_string_by_position (field_position_of_title)
            end
            Result := title_
        end

    price: REAL is
        do
            if is_obsolete or else price_ = Real_default_value then
                price_ := mt_get_real_by_position (field_position_of_price)
            end
            Result := price_
        end

    written_by: LINKED_LIST [AUTHOR] is
        do
            if is_persistent then
                written_by_.load_successors
            end
            Result := written_by_
        end

    publisher: PUBLISHER is
        do
            if is_obsolete or else publisher_ = Void then
                publisher_ :=
                    mt_get_successor_by_position (field_position_of_publisher)
            end
            Result := publisher_
        end
end -- class BOOK
```

The function to access attribute value, such as `title` or `price`, checks if the value is already read from the database or if the object is obsolete, i.e., the corresponding database object may have been updated. If not yet read or obsolete, it reads the value from the database and assigns the value to the corresponding field, such as `title_` or `price_`, then returns it. If the object is not persistent, the function just returns the value of its field.

In the sample schema defined in the Appendix-A, the maximum cardinality of the relationship `written_by` is unlimited while that of the relationship `publisher` is one. When we define the class `BOOK` in Eiffel that corresponds to the database schema, we usually use one of the collection classes as an Eiffel type for the relationship `written_by` and use just the class `PUBLISHER` as an Eiffel type for the relationship `publisher`.

Here, we call the relationship whose maximum cardinality is one *single-successor relationship*, and call the relationship whose maximum cardinality is more than one *multiple-successor relationship*.

The function to access relationship successors, such as `written_by` or `publisher`, checks if the successors are read from the database, or if the object is obsolete. If not yet read or obsolete, it reads the successors from the database and creates replica objects. The function has two slightly different succeeding behaviors according to the relationship's cardinality

- **Single-successor relationship:** The function to access the single-successor relationship, like `publisher`, then assigns the replica object to the corresponding field, `publisher_`, and returns it.
- **Multiple-successor relationship:** The function to access the multiple-successor relationship, like `written_by`, then assigns the replica objects to the container object, like `written_by_`, and returns the container object.

If the object is not persistent, the function just returns the object of its field without accessing the database.

As long as you use these accessing functions to access attribute values or relationship successors, you don't have to care about when and how the values or successors are read from the database. They are read from the database only once when they are needed first. The later accesses to the same attribute or relationship do not read the database.

NOTE: We recognize that this approach violates the Eiffel principle of command-query separation. We believe this is justified by the performance advantages that result from not having to read the database every time the value is accessed in Eiffel. It also simplifies programming, since only a single call is required.

Accesses through explicit loading

This approach does not use the Eiffel classes generated by `mt_odl`. You must call the routines discussed below yourself, and manually keep track of when values and successors are read from the database. This is more work, but it gives you the flexibility of complete control over when and which values or successors are accessed and loaded. These routines are defined in class `MT_STORABLE`, so you can call them on an instance of any class that inherits from it (that is, on any persistent object).

```
class
  MT_STORABLE
```

```

feature

    mt_load_all_values
        -- Load values of all attributes of the current object
    mt_load_all_successors
        -- Load successors of all relationships of the current object
    mt_load_all_properties
        -- Load both attributes and relationships
end

```

The procedures `mt_load_all_values`, `mt_load_all_successors` and `mt_load_all_properties` read all values or all successors and assign them to each field of the target replica object. With these procedures, you do not have to worry about attribute names or relationship names. Once these procedures are called on an object, you can directly access the Eiffel attributes, such as `title`, `price`, `written_by` and `publisher` in the class `BOOK`.

The class `MT_STORABLE` also provides get functions, such as `mt_get_integer_by_name`, which return the value or successors of a specific property directly from the database, without assigning the returned value or successors to the field of the target. These functions do *not* read values or successors that are loaded and cached in the Eiffel environment. Some of them are listed below:

```

feature
    mt_get_value_by_name (attr_name: STRING): ANY
        -- Return the value of the attribute specified by 'attr_name'
    mt_get_value_by_position (index: INTEGER): ANY
        -- Return the value of the attribute field-positioned at 'index'
    mt_get_integer_by_name (attr_name: STRING): INTEGER
    mt_get_integer_by_position (index: INTEGER): INTEGER
    mt_get_string_by_name (attr_name: STRING): STRING
    mt_get_string_by_position (index: INTEGER): STRING

    mt_get_successor_by_name (rs_name: STRING): MT_STORABLE
        -- Return the first successor object of Current through
        -- the relationship specified by 'rs_name'
        -- This is usually used to get a successor of 'single-relationship'
    mt_get_successor_by_position (index: INTEGER): MT_STORABLE
        -- Same as 'mt_get_successor_by_name' except that relationship
        -- is specified by field position
    mt_get_successors_by_name (rs_name: STRING): MT_RS_CONTAINABLE [MT_STORABLE]
        -- Return an array of successor objects of Current through
        -- the relationship specified by 'rs_name'
    mt_get_successors_by_position (index: INTEGER): MT_RS_CONTAINABLE [MT_STORABLE]
        -- Same as 'mt_get_successors_by_name' except that relationship
        -- is specified by field position

    ... more functions ...
end

```

When you use these functions, you specify an attribute or relationship by giving its name or field position as an argument. The name of an attribute or relationship should be same as the one used to define the Eiffel class. The field position of an attribute or relationship is an index used to obtain internal object properties by using some functions defined in class `INTERNAL`. (If a class is generated by the automatic code generation tool, `mt_odl`, then the functions to get the field position are provided in the class; their names start with `field_position_of`. See [Accesses through transparent loading](#) on page 10 for an example.)

These get functions are provided for very specific purposes such as when you need access only one attribute of a replica object without loading all attribute values, or without assigning the returned value to the replica object. The following sample procedures print all properties of a book using the procedure `mt_load_all_properties`.

```
print_properities_of_book(book_title: STRING) is
  local
    a_book: BOOK
    an_ep: MENTRYPOINTDICTIONARY
    books: ARRAY[MT_STORABLE]
    i: INTEGER
  do
    create an_ep.make_from_name("title", "BOOK")
    books ?= an_ep.retrieve_objects(book_title)
    from i := books.lower
    until i > books.upper
    loop
      a_book ?= books.item(i)
      a_book.mt_load_all_properties
      print(a_book.title)
      print(a_book.price)
      print(a_book.publisher)
      print(a_book.written_by)
      i := i + 1
    end
  end
end
```

The sample code above can be rewritten by using the get functions as follows,

```
print_properities_of_book_using_get_functions(book_title: STRING) is
  local
    a_book: BOOK
    an_ep: MENTRYPOINTDICTIONARY
    books: ARRAY[MT_STORABLE]
    i: INTEGER
  do
    create an_ep.make_from_name("title", "BOOK")
    books ?= an_ep.retrieve_objects(book_title)
    from i := books.lower
    until i > books.upper
    loop
      a_book ?= books.item(i)
      print(a_book.mt_get_string_by_name("title"))
      print(a_book.mt_get_real_by_name("price"))
      print(a_book.mt_get_successor_by_name("publisher"))
      print(a_book.mt_get_successors_by_name("written_by"))
      i := i + 1
    end
  end
end
```

3.4 Class Invariants

As discussed above, when a replica object is first created, neither attribute values nor relationship successors are loaded. This creates a potential problem: if, for example, the class `BOOK` were to use the attribute `title` in its invariant clause, it could cause an error. To avoid this situation, we recommend

using `implies`, so that the invariant clause returns `true` if the object is not persistent, or its properties are not loaded yet. You can write a class invariant as follows:

When using transparent loading

When you are using transparent loading, you can write a class invariant as follows:

```
class BOOK
.....
invariant
  title_not_void: is_persistent implies title /= Void
  publisher_not_void: is_persistent implies publisher /= Void
```

If an object is not persistent, the succeeding expressions, `title /= Void` or `publisher /= Void`, are not evaluated. Note that this approach assumes that the current database session is set properly whenever this invariant clause is evaluated.

When using explicit loading

When you are using explicit loading, you can write a class invariant as follows:

```
class BOOK
.....
invariant
  title_not_void: attributes_loaded implies title /= Void
  publisher_not_void: relationships_loaded implies publisher /= Void
```

- The value of `attributes_loaded` becomes `true` after the procedure `mt_load_all_values` is called.
- The value of `relationships_loaded` becomes `true` after the procedure `mt_load_all_successors` is called.
- The values of both `attributes_loaded` and `relationships_loaded` become `true` after the procedure `mt_load_all_properties` is called.

3.5 Attribute Types

The following table shows the Matisse data types and Eiffel types to which they correspond.

Matisse data type	Eiffel type
MT_BOOLEAN	BOOLEAN
MT_CHAR	CHARACTER
MT_DATE	DATE
MT_TIMESTAMP	DATE_TIME
MT_INTERVAL	DATE_TIME_DURATION
MT_BYTE	INTEGER_8
MT_SHORT	INTEGER_16
MT_INTEGER	INTEGER
MT_LONG	INTEGER_64
MT_NUMERIC	DECIMAL
MT_FLOAT	REAL
MT_DOUBLE	DOUBLE
MT_STRING, MT_TEXT	STRING
MT_BYTES, MT_AUDIO, MT_IMAGE, MT_VIDEO	ARRAY[INTEGER_8]
MT_SHORT_LIST	LINKED_LIST[INTEGER_16]
MT_INTEGER_LIST	LINKED_LIST[INTEGER]
MT_LONG_LIST	LINKED_LIST[INTEGER_64]
MT_NUMERIC_LIST	LINKED_LIST[DECIMAL]
MT_FLOAT_LIST	LINKED_LIST[REAL]
MT_DOUBLE_LIST	LINKED_LIST[DOUBLE]
MT_STRING_LIST	LINKED_LIST[STRING]
MT_DATE_LIST	LINKED_LIST[DATE]
MT_TIMESTAMP_LIST	LINKED_LIST[DATE_TIME]
MT_INTERVAL_LIST	LINKED_LIST[DATE_TIME_DURATION]
MT_BYTE_ARRAY	ARRAY[INTEGER_8]
MT_SHORT_ARRAY	ARRAY[INTEGER_16]
MT_INTEGER_ARRAY	ARRAY[INTEGER]
MT_FLOAT_ARRAY	ARRAY[REAL]
MT_DOUBLE_ARRAY	ARRAY[DOUBLE]
MT_STRING_ARRAY	ARRAY[STRING]

Note that multiple dimensional array for all array data types, such as MT_INTEGER_ARRAY, are not supported.

3.6 Collection Classes for Successors

The ISE Eiffel library provides a very rich set of collection classes such as `ARRAY`, `LIST`, and `TREE` etc. Matisse, however, uses the “relationship” mechanism to refer to a collection of objects. It would be possible to build collection classes equivalent to the ones defined in the Eiffel library, but we prefer a simple mechanism that does not assume the existence of those collection classes.

There are several reasons for this choice. The first is conceptual: a container object itself should not be persistent whenever a relationship can be used instead. The second reason is simplicity: we do not want to assume the existence of the equivalent collection class library in Matisse as a part of the basic binding mechanism. The third reason is compatibility with other interfaces like Java or SQL. Finally, this choice provides better performance: if we were to build the equivalent collection class library in the Matisse database, it would require more database accesses to reach the actual successor object.

MT_RS_CONTAINABLE and MT_LINEAR_COLLECTION

These are deferred classes, whose descendant classes can contain multiple-successor relationship objects. The following is an example of a descendant class.

```
class
  MT_ARRAY[G]

inherit
  ARRAY[G]

  MT_RS_CONTAINABLE[G]

  MT_LINEAR_COLLECTION[G]

creation
  .....
```

The binding defines three collection classes, `MT_ARRAY`, `MT_LINKED_LIST` and `MT_ARRAYED_LIST`, that inherit from the classes `MT_RS_CONTAINABLE`, `MT_LINEAR_COLLECTION`, and each of the three frequently used collection classes, `ARRAY`, `LINKED_LIST` and `ARRAYED_LIST`. `MT_ARRAY`, `MT_LINKED_LIST` and `MT_ARRAYED_LIST` are used in `mt_odl`-generated Eiffel functions that relate to one-to-many and many-to-many relationships (that is, relationship descriptors with a maximum cardinality greater than one). The behavior of each of these classes is identical to that of its ancestor class, except that it manages successor objects through a relationship.

When a replica object is created, an empty collection object is created for such relationship and assigned to the corresponding field of the replica object. For example, when a replica object of class `BOOK` is created, an instance of class `MT_ARRAY` is created and assigned to the field `written_by_`, but the instance of class `MT_ARRAY` contains no successor.

To load all the successor objects from the database, the procedure `load_successors` is called on the collection object. Once successors are loaded, further calls of `load_successors` on the same collection object have no effect. The procedure `load_successors` is used in other routines internally, for example, `written_by` from class `BOOK`, and `mt_load_all_successors` and `mt_get_successors_by_name` from class `MT_STORABLE`. Usually you do not need to use this procedure explicitly.

Alternatively, you may call the procedure `open_stream` on the collection object to iterate through successor objects using the routines `start`, `forth` and `item`. The successors are not read from the server unless a read operation is performed on the stream, and reading the stream does not put the successors in the collection object. Here is an example:

```
print_all_authors_of_book(book_title: STRING) is
  -- Using stream
  local
    a_stream: MT_STREAM
    an_author: AUTHOR
    a_book: BOOK
    an_ep: MTENTRYPOINTDICTIONARY
  do
    create an_ep.make_from_name("title", "BOOK")
    a_book := an_ep.retrieve_first(book_title)
    if a_book /= Void then
      a_stream := a_book.written_by.open_stream
      from
        a_stream.start
      until
        a_stream.exhausted
      loop
        print(a_stream.item)
        a_stream.forth
      end
      a_stream.close
    end
  end
end
```

Collection class and application

You can think of applications developed with the Matisse-Eiffel binding as having two layers, the “persistent classes layer” and the “application layer.” The former is composed of classes defined in an ODL file and stored in the database. The latter is an Eiffel application (including GUI and so on) that uses these persistent classes.

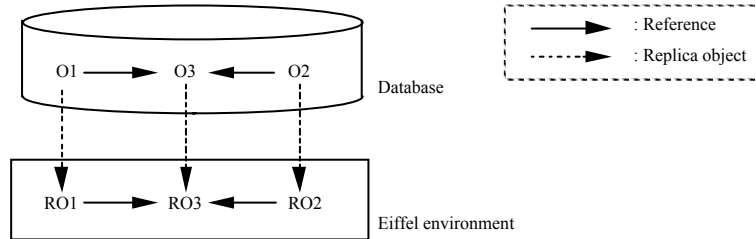
This binding uses `MT_ARRAY`, `MT_LINKED_LIST` and so on to define the persistent classes. But the interfaces of access functions (such as `written_by`) in the persistent classes are defined using the standard classes such as `ARRAY` or `LINKED_LIST`. Hence the application layer does not require `MT_ARRAY` or `MT_LINKED_LIST`, but can use on the standard Eiffel collection library.

Defining Your Own Collection Class

If you wish to store objects in classes not supported by the Matisse-Eiffel binding, such as `QUEUE`, `SORTED_LIST`, or `STACK`, you must define your own collection class. For instructions, see [Custom Collection Classes](#) on page 30.

3.7 Caching Objects

The objects retrieved from the database are cached in the Eiffel environment to maintain the object identity within a transaction or version access. For example, suppose that, in the database, object O1 refers to object O3, and object O2 refers to object O3, as in this diagram:



If you retrieve O1 and then navigate the relationship to retrieve O3, replica objects RO1 and RO3 are created in the Eiffel environment. If you then retrieve O2 and then navigate the relationship to retrieve O3, the replica object RO2 is created in the Eiffel environment, but RO3 is retrieved from the Eiffel environment's object cache.

Object identity is maintained within database connection boundaries. That is, it is maintained across transactions or version (read-only) accesses within the same database connection unless you explicitly clear the object cache. For example, say you start a transaction, retrieve an object P1, and commit the transaction; the replica object RP1 remains in the object cache. If you then start another transaction without disconnecting from the database, and retrieve the same object P1 from the database, replica object RP1 will be retrieved from the object cache.

When a retrieved object is not referred to by any other objects, it would normally be a candidate for garbage collection, but since replica objects are referred to by the binding's cache to maintain object identity this does not happen. (If your Eiffel environment provides a "weak reference" facility, you can modify the object cache source code to use it, in which case the garbage collector will reclaim such objects.) The class `MT_STORABLE` includes the procedure `remove_from_cache` to remove a replica object from the object cache.

The object cache is created per database connection. The cache is flushed when the connection is closed.

4 Modifying Persistent Objects

4.1 Setting Attribute Values

The Eiffel classes generated by `mt_odl` include procedures to update attribute values. The following code is taken from the `book.e` class generated by `mt_odl` from the ODL definition shown in [Sample Application Schema](#) on page 26:

```
set_title (new_title: STRING) is
  do
    title_ := clone (new_title)
    mt_set_string (field_position_of_title)
  end

set_price (new_price: REAL) is
  do
    price_ := new_price
    mt_set_real (field_position_of_price)
  end
```

Modifications made through these and similar procedures are immediately stored in the database. No further action is required.

4.2 Setting Relationship Successors

The Eiffel classes generated by `mt_odl` include procedures to update single-successor relationships. The following code is taken from the `book.e` class generated by `mt_odl` from the ODL definition shown in [Sample Application Schema](#) on page 26:

```
set_publisher (new_publisher: PUBLISHER) is
  do
    check_persistence (new_publisher)
    publisher_ := new_publisher
    mt_set_successor (field_position_of_publisher)
  end=
```

The procedure `set_publisher` first calls the procedure `check_persistence` with the argument `new_publisher`. The procedure `check_persistence` promotes the argument object and its all dependents into persistent objects if these objects are not persistent yet. We call this “persistence by reachability.”

Modifications made through this and similar procedures are immediately stored in the database. No further action is required.

No such procedures are generated for multiple-successor relationships. Instead, you must create a procedure like the following, which would be added to the class `PUBLISHER` in the sample application:

```
add_books (books: ARRAY[BOOK]) is
  local
    i: INTEGER
  do
    from i := books.lower
```

```

until i > books.upper
loop
  published_books.extend(books.item(i))
  -- Adding successor object.
  -- If books.item(i) is transient, it is promoted into persistent object.
  i := i + 1
end
end

```

4.3 Removing Attribute Values

The procedures `mt_remove_value_by_name` and `mt_remove_value_by_position` defined in class `MT_STORABLE` allow you to remove attribute values from replica objects. The procedure `mt_remove_value_by_name` takes an argument `attribute_name` of type `STRING` to specify the attribute by its name, while `mt_remove_value_by_position` takes an argument `position` of type `INTEGER` to specify the attribute by its field position. Once the procedure is called on an object, the specified attribute value of the object becomes unspecified in the database. However, the procedure does not change the corresponding field of the replica object.

```

mt_remove_value_by_name(attribute_name: STRING)
mt_remove_value_by_position(position: INTEGER)

```

The following procedure is defined in class `BOOK`. It removes the value of the attribute `price`.

```

remove_price is
do
  price := 0
  mt_remove_value_by_position(field_position_of_price)
end

```

4.4 Deleting Objects

The procedure `mt_remove` defined in class `MT_STORABLE` allows you to delete an object from the database. The corresponding replica object is also removed from the object cache maintained by the Matisse-Eiffel binding; however, the replica object itself still remains in the Eiffel environment until the garbage collector collects it. If an object is not persistent, the procedure `mt_remove` does nothing. This procedure must be called inside a transaction.

The following code is a procedure using `mt_remove`.

```

delete_book(book_title: STRING) is
local
  ep: MENTRYPOINTDICTIONARY
  a_book: BOOK
do
  create ep.make_from_name("title", "BOOK")
  a_book := ep.retrieve_first(book_title)
  if a_book = Void then
    print("The book is not found%N")
  else
    a_book.mt_remove
  end
end
end

```

5 Creating Persistent Objects

Transient objects instantiated from the classes generated by `mt_odl` become persistent automatically when referenced by a persistent object, or may be explicitly promoted to persistent objects.

5.1 Automatic Promotion

When transient object O1 (an instance of a class generated by `mt_odl`) is referenced from another object that is already persistent, object O1 will automatically be promoted into a persistent object. Here is an example:

```
create_new_book(new_book_title: STRING; a_publisher: PUBLISHER)
  require
    a_publisher_is_persistent: a_publisher.is_persistent
  local
    new_book: BOOK
  do
    create new_book.make(new_book_title)
      -- Some more settings on new_book.
      -- The object 'new_book' is transient.
    a_publisher.published_books.extend(new_book)
      -- Now, the object 'new_book' is promoted into a persistent object.
  end
end
```

5.2 Explicit Promotion

The procedure `persist` allows you to explicitly promote a transient object instantiated from a class generated by `mt_odl` into a persistent object. This procedure is defined in class `MATISSE_APPL`. Here is an example:

```
create_new_author(name: STRING; db_session: MATISSE_APPL) : AUTHOR
  do
    create Result.make(clone(name))
    .... -- Some more operations on Result
    db_session.persist(Result)
      -- db_session represents a database session.
  end
end
```

6 Executing SQL Statements

You can execute any SQL statements in Eiffel and get the results. The typical procedure to execute an SQL statement is:

1. Create a statement object from a database session object.
2. Execute a SQL statement using the statement object.
3. Get the result.
 - When a SELECT query is executed, the result is the set object, which represents a pseudo-table.
 - When an UPDATE, DELETE, or INSERT statement is executed, the result is the number of objects affected.
 - Otherwise, use the `statement_info` routine to get whatever information is needed.
4. In the case of a SELECT query, close the result,
5. Close the statement object.

For more information about Matisse SQL, see the *Matisse SQL Programmer's Guide*.

6.1 MT_STATEMENT

The only type of SQL statement supported in this release of the Matisse-Eiffel binding is `MT_STATEMENT`. In a future release we will implement `MT_PREPARED_STATEMENT` and `MT_CALLABLE_STATEMENT`.

An object of the `MT_STATEMENT` class is created using a database session object, an instance of `MATISSE_APPL`. For example:

```
feature
    db_session: MATISSE_APPL

feature
    test_sql is
        local
            stmt: MT_STATEMENT
            result_set: MT_RESULT_SET
            a_person: PERSON
        do
            db_session.start_transaction
            stmt := db_session.create_statement
            result_set := stmt.execute_query ("SELECT REF(p) FROM Person p");
            from
                result_set.start
            until
                result_set.exhausted
            loop
```

```

        a_person := result_set.get_object(1) -- get an object at the first
column in the select-list
        print (a_person.firstname); print (" ")
        print (a_person.lastname); print ("%N")
        result_set.forth
    end
    result_set.close
    stmt.close
end

```

The `MT_STATEMENT` object can execute any SQL statement. Which function you should use to execute it depends on the statement's type:

- `execute_query` This is used to execute a `SELECT` query. It returns an `MT_RESULT_SET` object, which allows you to traverse on the result pseudo-table.
- `execute_update` This is used to execute `UPDATE`, `DELETE`, or `INSERT` statement. It returns the number of objects affected by the statement.
- `execute` This can execute any SQL statement. It is useful when you don't know what kind of SQL statement you are going to execute.
 - It returns a boolean value `True` if a result set object is prepared as the result, i.e., a `SELECT` query is executed. The function `result_set` returns the result set object.
 - Otherwise it returns `False`. If `UPDATE`, `DELETE`, or `INSERT` statement was executed, the function `update_count` returns the number of objects affected. The type of SQL statement executed can be obtained by using `statement_type`. For other kinds of SQL statements, use the `statement_info` function to get the information.

6.2 MT_RESULT_SET

An instance of this class represents a pseudo-table that is returned by a `SELECT` statement. You can get the selected objects or values that are listed in the statement's `SELECT` list.

Getting objects

To select objects using a `SELECT` statement, use `REF()` in the `SELECT` list. The function `get_object` returns an Eiffel object for the current row in the `MT_RESULT_SET` object. For example:

```

feature
    db_session: MATISSE_APPL

feature
    exec_sql1 is
        local
            sql_stmt: MT_STATEMENT
            result_set: MT_RESULT_SET
            a_book: BOOK
        do
            db_session.start_transaction

```

```

sql_stmt := db_session.create_statement
    -- Create a new statement object.

result_set := sql_stmt.execute_query
("SELECT REF(c) FROM Book c WHERE title LIKE 'A%')
    -- Execute a query.

from
    result_set.start
        -- Set the cursor to the initial position.

until
    result_set.exhausted
loop
    a_book ?= result_set.get_object (1)
        -- Get the object. The argument of get_object is the position
        -- of the column in the SELECT list.

        result_set.forth
            -- Move the cursor to the next position.
end

result_set.close
sql_stmt.close
    -- Free the external resources

db_session.commit_transaction
end

```

Getting values

When you are selecting values such as integers or strings in the SELECT list, you can use routines such as `get_integer` or `get_string` to get these values in the current row of the `MT_RESULT_SET` object. For example:

```

...
result_set := sql_stmt.execute_query
("SELECT title, price FROM Book WHERE title LIKE 'A%')
    -- Execute a query.

from
    result_set.start
        -- Set the cursor to the initial position.
until
    result_set.exhausted
loop
    a_title := result_set.get_string (1)
        -- Get the string value. The argument of get_string is the position
        -- of the column in the SELECT list.

    a_price := result_set.get_real (2)
        -- Get the real value.

    result_set.forth
        -- Move the cursor to the next position.
end

```

...

Appendix A: Sample Application Schema

This section describes the sample application schema that is used throughout this document. The schema is defined in the Matisse ODL format (see *Getting Started with Matisse* for more information).

```
interface BOOK : persistent {
    attribute String title;
    mt_entry_point_dictionary BookTitleDictionary
    entry_point_of title
    make_entry_function "make-entry";
    attribute Float price;
    relationship List<AUTHOR> written_by
    inverse AUTHOR::books;
    relationship PUBLISHER publisher
    inverse PUBLISHER::published_books;
    mt_index BookTitleIndex
    criteria {BOOK::title MT_ASCEND 32};
};

interface PERSON : persistent {
    attribute String name;
    mt_entry_point_dictionary PersonNameDictionary
    entry_point_of name
    make_entry_function "make-entry";
    attribute Long birth_year;
};

interface AUTHOR : PERSON : persistent {
    relationship List<BOOK> books
    inverse BOOK::written_by;
};

interface PUBLISHER : persistent {
    attribute String name;
    mt_entry_point_dictionary PublisherNameDict
    entry_point_of name;
    relationship List<BOOK> published_books
    inverse BOOK::publisher;
};
```

Appendix B: mt_odl

The following command generates Eiffel source code files from an ODL file like the one shown above:

```
mt_odl -eiffel odl_filename
```

A .e file is created in the current directory for each class defined in the specified ODL file. For example, from the ODL definition above, mt_odl would create book.e, person.e, author.e and publisher.e.

mt_odl also creates an additional file, mt_container_types.e, which is used by the binding.

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

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

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

For more information about ODL, see *Getting Started with Matisse* and the *Matisse ODL Programmer's Guide*.

Appendix C: Troubleshooting

Error Messages

The following are common error messages you may encounter and their likely causes.

INVALIDATTRIBUTE

An attribute (date, number, string, etc.) value in the Eiffel code is `Void` but the corresponding attribute description in the ODL file does not have the `Nullable` flag set.

INVALIDSUCCESSORS and other messages containing "SUCC"

This error message indicates one of the following problems relationship successors:

- A relationship specified in the ODL file as having a minimum cardinality greater than zero such as `[1, 1]` or `[1, -1]` is void in the Eiffel object you are trying to commit.
- You specified a relationship with cardinality `[1, 1]` in the ODL file but one Eiffel object is the successor to two different Eiffel objects.
- The number of successors is invalid. Check that any `MT_LINKED_LIST` declared in the ODL file to contain at least a certain number of elements does in fact contain at least that many objects before calling commit.
- An `MT_LINKED_LIST` contains void objects.

NOTRANSACTION

This error message indicates one of the following problems:

- The application is attempting to retrieve objects, load values, or execute a SQL statement outside of a `start_version_access ... end_version_access` or a `start_transaction ... commit` pair.
- The application is attempting to persist objects or commit a transaction outside of a `start_transaction ... commit` pair.
- An invariant is not preceded by `is_persistent implies`.

Other Problems

Run-time Panic with the 'Matisse binding: no make_procedure found' error

The panic is happening because a Matisse C library could not find the routine “make” for `MT_LINKED_LIST` (or other Matisse specific container classes), i.e., the class was not visible to the C program.

You need to include `MT_LINKED_LIST` (or other Matisse specific container classes) in the visible clause in your Ace file or in the Visible classes in the matisse cluster in System Configuration, and recompile your project (you need to freeze it).

Run-time error: 'Class <your class name> does not exist or is outside the Eiffel system'

You get this error when you are retrieving objects from your database, but the class of the objects is not “really” included in your Eiffel project. Open the project using EiffelStudio, and find the class in the ‘Clusters’ window. The class oval should be gray. You need to make it blue, for example, by declaring a variable of the class somewhere in the project.

Eiffel assertion violation

Make sure all invariants are preceded by `is_persistent implies` if they are using persistent fields.

Eiffel application appears to hang mysteriously

Your application may be trying to access an object on which another process has obtained a write lock.

Another another process may have an open transaction (not read-only version access) that is accessing an object which you have modified and are trying to commit.

Non-void objects appear as void references in debugger

The binding retrieves objects or values only when your program accesses them; otherwise, they retain their default value in Eiffel. For debugging purposes, you may want to use `mt_load_all_properties` in class `MT_STORABLE`.

Remember that features that are not declared persistent in the `.odl` file will not be stored in the database. That is, when you retrieve an object, any non-persistent features will have the default values (`void`, `0`, etc.) unless you redefine `post_retrieved` on that class.

Segmentation violation during call to commit or persist

If you changed the ODL file and regenerated Eiffel classes, you need to freeze your Eiffel application again.

Appendix D: Custom Collection Classes

If you wish to store objects in classes not supported by the Matisse-Eiffel binding, such as `QUEUE`, `SORTED_LIST`, or `STACK`, you must define your own collection class. The following are the relevant portions of `MT_RS_CONTAINABLE` and `MT_LINEAR_COLLECTION`:

```
deferred class
  MT_RS_CONTAINABLE[G->MT_STORABLE]

feature
  load_successors is
    do ...
    end

end -- class MT_RS_CONTAINABLE

deferred class MT_LINEAR_COLLECTION[G->MT_STORABLE]

feature -- Stream
  open_stream : MT_RELATIONSHIP_STREAM is
    do ...
    end

feature -- Status
  count : INTEGER is
    deferred
    end

feature -- Loading
  mt_put_at_loading(new: G; i: INTEGER) is
    deferred
    end

  mt_resize_at_loading(new_size: INTEGER) is
    deferred
    end

end -class MT_LINEAR_COLLECTION
```

As you can see, descendant classes of class `MT_LINEAR_COLLECTION` should redefine the “deferred” routines `count`, `mt_put_at_loading` and `mt_resize_at_loading`, which retrieve and store successor objects. To define a new descendant class of `MT_RS_CONTAINABLE` and `MT_LINEAR_COLLECTION` that can contain successor objects, redefine these routines as detailed in the following section.

Defining a New Collection Class

This example class will inherit from `LINKED_STACK`.

1. Define the creation procedure `make`.

This procedure initializes a collection object with a default size, so it needs no arguments. Use the creation procedure `make` that is inherited from the class `LINKED_LIST`. Note that `LINKED_STACK` is inheriting from `LINKED_LIST`.

2. Define the procedure `mt_put_at_loading`.

This procedure puts a successor object in a collection object when the procedure `load_successors` is called on the collection object.

The procedure `mt_put_at_loading` takes two arguments: `v` is the successor object to be added, and `i` is an integer representing the positional index of the successor object in the relationship. The procedure is called for all the successors in the relationship.

We think it is appropriate to push the successor objects onto the stack in the same order as that of the successors list. Hence, the procedure `mt_put_at_loading` just calls the procedure `extend` with the argument `v`.

3. Define the procedure `mt_resize_at_loading`.

This procedure resizes a collection object when the procedure `load_successors` is called on the collection object.

The procedure `mt_resize_at_loading` is called from the procedure `load_successors` with one argument, `new_size` of type `INTEGER`. `new_size` is the number of the successors in the relationship in question. The collection object should be resized (if necessary) so that it can hold all the successors.

Since the procedure `extend` of class `LINKED_STACK` extends the stack itself, the procedure `mt_resize_at_loading` of class `MT_LINKED_STACK` does not have to do anything.

4. Define the function `linear_representation`.

This function is used to get all the successor objects in a collection object in a linear order when the successor objects need to be stored to the database.

The function `linear_representation` is defined in one of the ancestor classes, `LINKED_STACK`. However, the order of the list returned by this function is the reverse of the original order of insertion. We need to redefine the function so that the order is the same as the original insertion order.

5. Define the function `count`

This function is inherited from the class `LINKED_LIST`. We do not have to redefine it.

The resulting class definition is as follows:

```
class
  MT_LINKED_STACK[G->MT_STORABLE]

inherit
  LINKED_STACK[G]
  redefine
    linear_representation
  end

  MT_RS_CONTAINABLE[G]
```

```

    MT_LINER_COLLECTION[G]

creation
  make

feature
  mt_put_at_loading(v: G; i: INTEGER) is
  do
    extend(v)
  end

  mt_resize_at_loading(new_size: INTEGER) is
  -- Do nothing
  do
  end

  linear_representation: ARRAYED_LIST [G] is
  -- Representation as a linear structure
  -- (order is the same of original order of insertion)
  local
    old_cursor: CURSOR
  do
    old_cursor := cursor;
    from
      create Result.make (count);
      start
    until
      after
    loop
      Result.put_front (ll_item);
      forth
    end;
    go_to (old_cursor)
  end

end -- class MT_LINKED_STACK

```

Specifying a Collection Class for Relationships

In this version of the Matisse-Eiffel binding, the Eiffel class for a Matisse multiple-successor relationship is defined as a constant in class `MT_CONTAINER_TYPES`. The default Eiffel class for a Matisse multiple-successor relationship is class `MT_LINKED_LIST`. If you want to use another class, then you must add a line in the function `Container_class_names`. The following code is an example of class `MT_CONTAINER_TYPES`.

```

class
  MT_CONTAINER_TYPES

feature {NONE} -- constants

  Container_class_names: HASH_TABLE[STRING, STRING] is
  -- Value is a container class name in upper case
  -- Key is a Matisse relationship name.
  once
    create Result.make(20)
    Result.put("MT_ARRAY", "BOOK__written_by")

```

```
        Result.put("MT_ARRAYED_LIST", "PUBLISHER__published_books")
    end
end -- class MT_CONTAINER_TYPES
```

This function returns a constant hash table whose key is a Matisse relationship name and whose value is an Eiffel class name. For example, if you want to use class `MT_ARRAY` for the relationship `AUTHOR__books`, you should add the following line in the function:

```
Result.put("MT_ARRAY", "AUTHOR__books")
```

Appendix E: Class Hierarchies and Client Relations

