The EiffelMatisse library: An O-O database handle for EiffelStore

Draft

Book identification

The EiffelMatisse library: An O-O database handle for EiffelStore, ISE Technical Report TR-EI-57/MT.

Publication history

First published:March 1996. **This version is a draft and may not be distributed without ISE's explicit authorization.**

Author

Sami Ben Grine (original French version).

Translated into English, with minor changes, by Bertrand Meyer.

Software credits

The Matisse library was developed by Sami Ben Grine, drawing from the results of an earlier Matisse-Eiffel interface by Hubert Divoux.

COMPANY PRESENTATION

ISE (Interactive Software Engineering) is dedicated to improving software quality and productivity through advanced methods, tools and languages, based on sound scientific principles and on the systematic application of object technology.

The company provides a complete line of development tools as well as on-site consulting, library development services, and a training program on all aspects of O-O technology: analysis, design, implementation techniques, graphics, library building, Eiffel language, project management, large system design etc. ISE is also the original designer of the Eiffel method and language and a member of NICE, the Nonprofit International Consortium for Eiffel.

Obtaining more information

To keep informed of the latest developments in Eiffel and ISE products and services, or for a list of international ISE Eiffel 3 distributors, please contact:

Interactive Software Engineering Inc.
270 Storke Road, Suite 7, Goleta, CA 93117 USA
Telephone 805-685-1006, Fax 805-685-6869
E-mail: <*info@eiffel.com*> for general information.
<*training@eiffel.com*> for information on training courses and the Eiffel Education Partnership program (open to universities and other educational institutions).
<*eiffelworld@eiffel.com*> for information on the Eiffel World quarterly newsletter and subscriptions.
<*announce-request@eiffel.com*> to be included in the ISE Eiffel e-mail list (for ISE Eiffel users).
<*tools@tools.com*> for information about the TOOLS conferences (Technology of Object-Oriented Languages and Systems: USA, Europe, Pacific) and Eiffel user conferences.
<*support@eiffel.com*> for customer support.

On-line access: FTP and WWW (Mosaic)

ISE maintains an up-to-date repository of information about products and services, the TOOLS conferences, news of interest to the Eiffel community, Frequently Asked Questions, common problems, known bugs and workarounds, new releases. This on-line server may be accessed through FTP or WWW (World-Wide Web, accessible through Mosaic and similar tools) at the following addresses:

ftp://ftp.eiffel.com
http://www.eiffel.com

Welcome to Eiffel

Welcome to the premier multi-platform object-oriented environment for the quality-conscious software developer, featuring:

- Full implementation of the acclaimed Eiffel language, as described in *Eiffel: The Language* and following the recommendations of the Nonprofit International Consortium for Eiffel (NICE).
- Multi-platform, portable code generation.

- Full implementation of object-oriented principles — not a hybrid.
- Seamless integration of existing code written in C.
- C code generation for portability and efficiency.
- The first fully object-oriented environment: all tools represent objects, not functions.
- Browsing, documentation, debugging, fast compilation and much more.
- Automatic memory management (no more pointer chasing), automatic recompilation (no more Make files), automatic inlining, automatic application of static binding.
- Static typing, assertions, multiple inheritance, constrained and unconstrained genericity, exception handling, object persistence... all in a clear, readable language that you can learn in a few days.
- Thousands of reusable software components: fundamental data structures and algorithms, graphics, client-server communication, lexical analysis, parsing, database access, object editing, ...
- Graphical interface builder.
- Supports the entire software lifecycle, from analysis and design to implementation and maintenance. Enables a seamless, reversible approach to object-oriented software construction.
- Extensive literature and textbooks available from Prentice Hall and other major publishers.
- The solution of choice whenever the name of the game is **quality** and **productivity**.

CONTENTS

1 INTRODUCTION 1

1.1 Delivery contents 1
1.2 Limitations of the current release 2
1.3 Using the library 2
1.4 Error handling 3
1.5 Constants 4

2 ENTITIES 4

2.1 Class *MT_CLASS* *4*
2.2 Class *MT_OBJECT* *4*
2.3 Class *MT_ATTRIBUTE* *5*
2.4 Class *MT_INFO* *6*
2.5 Class *MT_RELATIONSHIP* *6*
2.6 Class *MT_PROPERTY* *6*
2.7 Classes *MT_ATTACH*, *MT_ROUTINE*, *MT_SERVICE_FUNCTION* *6*
2.8 Classes *MT_INDEX* and *MT_INDEX_CRITERION* *7*
2.9 Class *MT_ENTRY_POINT* *7*
2.10 Class *MT_MESSAGE* *7*

3 THE HIGH-LEVEL INTERFACE 7

3.1 Managing the database and the transactions 7
3.2 Accessing the database 8
3.3 Updating the database 11
3.4 Version streams 11
3.5 Repository and storage 12

4 OBJECT PERSISTENCE SUPPORT 12

4.1 Storing into flat files 13
4.2 Selective storing 13
4.3 Storing into a database 13
4.4 Managing shared objects 13

5 EXTENSIONS 13

5.1 Automatic generation of ODL files 13

YOUR ROADMAP TO ISE EIFFEL DOCUMENTATION

OVERVIEW

Note: the precise references to the document mentioned here, such as [Environment], appear on the next page.

Extracts from some of the documents are also included in the delivery. Check the subdirectory **doc** of the delivery.

WHAT YOU MUST ABSOLUTELY HAVE READ TO USE ISE EIFFEL:

If you are a total Eiffel beginner:

- Read appendix D of [Environment], a general introduction to the concepts of Eiffel.

Before using the environment:

- Read the Preface and chapters 1, *Getting started*, and 2, *A Guided Tour*, of [Environment].
- Read chapter 1 of [Library], *Introduction to the Base libraries*, to know what libraries are available.

TO PROCEED TO MORE ADVANCED USES:

- Read the rest of [Environment].
- Read the chapters of [Library] corresponding to the libraries that you may want to use. To understand library principles, read chapters 2 and 3 of [Library].
- Refer to [Reference] or to [Language] for specific language questions.
- Read [Method] and [Language] to understand the approach in depth.

FOR SPECIFIC APPLICATIONS:

- If you are going to develop graphical (GUI) applications, read [EiffelBuild]; if you need to use EiffelVision directly, not just through EiffelBuild, read [EiffelVision-intro].
- To learn about how to produce high-level system architecture diagrams, and produce systems through graphical input, read [EiffelCase].
- To develop client-server applications or interprocess communication, read [EiffelNet].
- If your software needs to perform numerical computation, read [EiffelMath].
- To use Eiffel with the Matisse database management system, read [EiffelMatisse].

IF YOU ARE IN CHARGE OF INSTALLING ISE EIFFEL:

- Read appendices B and C of [Environment].

YOUR ROADMAP TO EIFFEL DOCUMENTATION:

THE REFERENCES

All the documents listed are available from ISE., including the Prentice Hall books (also available from the publisher and from technical bookstores).

The Official Documentation

[Language] *Eiffel: The Language*. Published by Prentice Hall. The complete language description.

[Reference] Eiffel: The Reference: A short language description, extracted from [Language]. Appendix K is an overview of the Eiffel approach. Report number TR-EI-41/ER.

[Library] *Reusable Software: The Base Object-Oriented Component Libraries*. A detailed description of the EiffelBase, EiffelLex and EiffelParse libraries, preceded by an in-depth examination of library design principles. TR-EI-44/LI. Also available as a book published by Prentice Hall.

[Environment] *ISE Eiffel 3: The Environment*. A detailed description of the EiffelBench development environment. TR-EI-39/IE. Also available as a book published by Prentice Hall.

[EiffelBuild] *Building Graphical Applications with EiffelBuild.* A presentation of EiffelBuild. TR-EI-43/UI.

[EiffelCase] *EiffelCase: Engineering Object-Oriented Software, Forward and Backward*. A description of EiffelCase. TR-EI-53/EA.

[EiffelMath] *EiffelMath: A Numerical Library for Numerical Object-Oriented Computation.* A presentation of EiffelMath, the numerical computation library of ISE Eiffel 3. TR-EI-54/EM.

[EiffelNet] *Client -Server Programming with Eiffel: The EiffelNet Library.* A description of how to build client-server applications and inter-process communication with EiffelNet. TR-EI-47/EN.

[EiffelVision-intro] *Introduction to EiffelVision*. A general presentation of the EiffelVision graphical and user interface library. TR-EI-40/EV.

[EiffelMatisse] ** *The EiffelMatisse library: An O-O database handle for EiffelStore.* A presentation of the interface between Eiffel and the Matisse OODB. TR-EI-57/MT.

Important Additional Literature

*** This book*

[Method] *Object-Oriented Software Construction*. The complete presentation of the underlying method. Published Prentice Hall.

[BON] *Seamless Object-Oriented Software Architecture* by Kim Waldén and Jean-Marc Nerson. Published by Prentice Hall. A presentation of O-O analysis and design based on systematic concepts close to those of Eiffel. Introduces the Business Object Notation (BON), the basis for EiffelCase.

1 INTRODUCTION

The EiffelMatisse library enables Eiffel systems to use a Matisse database. From Eiffel, users can access:

- The entire Matisse application programming interface (API) in Eiffel form, in which all the C calls are hidden.
- A set of database operations for selection and modification of objects stored in the database.
- Persistence services on object networks.

This manual describes the available facilities.

1.1 Delivery contents

The EiffelMatisse delivery consists of a set of Eiffel clusters:

- *matisse_entities*: contains classes describing Matisse entities such as classes, attributes, relations and indexes. This cluster contains the *matisse_externals* and *matisse_streams* subclusters.
- *matisse_externals*: the lowest-level layer, encapsulating C calls.
- *matisse_streams*: the encapsulation of stream management services for getting objects from a database.
- *matisse_interface*: the highest-level cluster, providing the EiffelStore interface.
- *matisse_sr*: object storing and retrieval.

In addition to these Eiffel clusters, the delivery includes a directory *Clib* that contains the requisite C code, and provides a utility called *Build_a_Clib* to compile it.

The Ace control file (Assembly of Classes in Eiffel) of your system should use the following elements:

```
        -- Matisse for EiffelStore
    cluster
        matisse_interface:
            "$EIFFEL3/library/store/dbms/oodbms/matisse/matisse_interface";
        matisse_entities:"$EIFFEL3/library/store/dbms/oodbms/matisse/matisse_entities";
        matisse_entities_externals:
            "$EIFFEL3/library/store/dbms/oodbms/matisse/matisse_entities/externals";
        matisse_entities_streams:
            "$EIFFEL3/library/store/dbms/oodbms/matisse/matisse_entities/streams";
```

external

object:"*-L$(EIFFEL3/library/store)/spec/$(PLATFORM)*";

"*-lsupport*";

"*-lsys_time*";

"*$(EIFFEL3/library/store)/dbms/oodbms/matisse/Clib/mt_support.o*";

"*$(MTS_ROOT)/lib/csupport.o*";

"*$(MTS_ROOT)/lib/libmatisseDS.a*"

This assumes that the environment variables *EIFFEL3* and *MTS_ROOT* denote the installation directories of Eiffel and Matisse, and that EiffelStore has been installed in the expected location *$EIFFEL3/library/store*.

The delivery also includes a directory *examples* containing various examples enabling you to become familiar with the mechanisms offered.

1.2 Limitations of the current release

Out of 93 routines in the C interface of Matisse, 16 do not currently have an Eiffel equivalent. (The list of routines and their C counterparts is given below.) The unsupported functions include 6 user functions, four calls applying to procedural attachments, and three others. At the lowest level, then, the library currently covers 83% of the C Matisse API.

In addition, some of the Matisse base types (*MTU8*, *MTU16*, *MTU32 and MTS16*) do not have a counterpart in the Eiffel run-time, and automatic type conversions for arrays are limited to one-dimensional arrays.

1.3 Using the library

Common database operations require a connection with the Matisse server. The EiffelStore interface transparently takes care of the corresponding services: connecting, disconnecting, transaction management. It is also possible to handle these functions directly by calling the low-level primitives through the classes *MT_DB_CONTROL_MAT_EXTERNAL* et *MT_STATUS_MAT* of the *matisse_external* cluster.

The rest of this discussion will assume that a session has been opened for a Matisse database. The following example shows how to achieve this. To use the example, insert your application-specific actions into the body of the *actions* routine.

class *SIMPLE* ***inherit***

MATISSE_CONST

creation

make

```
feature {NONE}
    make is
            -- Execute an ordinary Matisse session.
        do
                -- Step 1 – Choose a Matisse server and a database:
            !! appl.login ("venus","COMPANY", 0, 0)
                -- Step 2 – Choose a mode of operation:
            appl.set_mode (OPENED_TRANSACTION, Void)
                -- Step 3 – Attach the EiffelStore handle:
            appl.set_base
                -- Step 4 – Start the Matisse session:
            !! session.make
                -- Step 5 – Connect to the database:
            session.connect
                -- Step 6 – Application-specific actions:
            actions
                --Step 7 – Disconnect:
            session.disconnect
        end
feature -- Database operations
        actions is
            -- Execute application-specific processing.
        do
                ... Your operations here ...
        end
feature {NONE} -- Implementation
    appl: MATISSE_APPL
    session: DB_CONTROL
end -- class SIMPLE
```

1.4 Error handling

Each time an operation of the Matisse API is executed, it produces and stores an error code accessible through class *DB_STATUS_MAT*. The class provides features for accessing the last error code, the last error message, the object that caused the last error, and other properties of the error. Note that each Matisse call resets this information; for example if you execute a failed database operation and then a disconnect, the error code will be that of the disconnect, not the operation that preceded it.

Here too it is possible to go directly to the C level rather than using *DB_STATUS_MAT*.

1.5 Constants

Some Matisse operations require the caller to pass some constants. The class *MATISSE_CONST.* provides constant attributes for that purpose: Matisse base types, connection parameters, possible result codes for Matisse requests, and internal EiffelStore codes.

2 ENTITIES

Every object present in a Matisse database has a unique integer identifier, which is used by the Eiffel side to distinguish objects. Eiffel objects representing Matisse entities are instances of a set of classes which all have an integer attribute denoting the identifier of an associated database object. All database accesses go through such objects.

The corresponding cluster is *matisse_entities*. The following sections introduce its classes, whose names all begin with *MT_*: *MT_CLASS*, *MT_OBJECT*, *MT_ATTRIBUTE*, *MT_INFO*, *MT_RELATIONSHIP*, *MT_PROPERTY*, *MT_ATTACH*, *MT_ROUTINE*, *MT_SERVICE_FUNCTION*, *MT_INDEX*, *MT_INDEX_CRITERION*, *MT_ENTRY_POINT* and *MT_MESSAGE*.

2.1 Class *MT_CLASS*

Class *MT_CLASS* represents the notion of Matisse class, of an instance of *MtClass* in the current database. Through an Eiffel instance of *MT_CLASS* you can create new objects in the database, delete objects, get all the instances of a class, its attributes, its relations, its inverse relations, its parents, its heirs and other class properties.

The following example shows how to obtain the attributes of class *CAR*, assumed to have been defined on the Matisse side:

```
...
matisse_car: MT_CLASS
matisse_car_attributes: ARRAY [MT_ATTRIBUTES]
...
!! matisse_car•make ("Car")
matisse_car_attributes:= matisse_car•attributes
```

2.2 Class *MT_OBJECT*

Class *MT_OBJECT* represents Matisse objects of an arbitrary kind. It serves to manipulate instances of user classes. It is possible to create an instance on the Eiffel side, or retrieve it from the database, and then remove it on the Matisse side.

When an object exists you can access its successors and predecessors in one of the relations that apply to it, add new successors (at the beginning of the list, at the end of the list, or after a given object), and find out what has been modified since the beginning of the current transaction.

You can also obtain other practical information such as the generating class and whether it belongs to the meta-schema.

The following example shows how to create a new instance of the Matisse class *Car*.

...
your_car: *MT_OBJECT*
...
!! *your_car*•*make_from_class* (*matisse_car*)
io•*put_boolean* (*your_car*•*is_predefined_msp*)

2.3 Class *MT_ATTRIBUTE*

Attributes in Matisse are not specific to a class; instead, they are instances of the *MtAttribute* class, whose counterpart on the Eiffel side is *MT_ATTRIBUTE*. This makes it possible to treat attributes as full-fledged entities. The principal operations on an attributes are:

- Finding out the attribute type.
- Get the value of the corresponding field for a certain object.
- Change the value of the corresponding field for a certain object.

These operations require establishing a correspondence between Eiffel's and Matisse's base type, as given by the following table:

Eiffel type	Matisse type
INTEGER	*MTS32*
DOUBLE	*MTDOUBLE*
REAL	*MTFLOAT*
CHARACTER	*MTCHAR*, *MTASCIICHAR*
STRING	*MTSTRING*, *MTASCIISTRING*
LINKED_LIST [*basic_type*]	*MTbasic_type_LIST*
ARRAY [*basic_type*]	*MTbasic_type_ARRAY*

The following example shows how to modify the *model* field of an instance of class *Car*:

model: *MT_ATTRIBUTE*
...
!! *model*•*make* (*"model"*)'
model•*set_value* (*your_car*, *MtString*,*"P911"*)

Here now is how to read that field for a certain instance:

model_name: *STRING*
...
model_name?= *model*•*value* (*your_car*)

The value of an object field is always a reference to an Eiffel object. So you must be careful when dealing with Eiffel's basic types, which are available under the form *basic_type_REF*.

2.4 Class *MT_INFO*

An instance of class *MT_INFO* provides information about the connected database. Here for example is how to obtain various information on the server:

Here is an example showing how to obtain information about the server:

info: *MT_INFO*
...
!! *info*
mbo:= *info*•*max_buffered_objects*
mcn:= *info*•*max_index_criteria_number*
mkl:= *info*•*max_index_key_length*
trb:= *info*•*total_read_bytes*
twr:= *info*•*total_write_bytes*

2.5 Class *MT_RELATIONSHIP*

Class *MT_RELATIONSHIP* is the counterpart of the Eiffel notion of reference. Although indispensable to obtain objects, it exports few operations since Eiffel features apply to these objects, not the references themselves.

As an example, here is how to read the list of vehicles in a carmaker's catalog, assuming the schema has defined the corresponding relation:

carmaker_catalog: *MT_OBJECT*
model_relation: *MT_RELATIONSHIP*
models: *ARRAY* [*MT_OBJECT*]
...
!! *model_relation*•*make* (*"all_models"*)
models:= *carmaker_catalog*•*successors* (*model_relation*)

2.6 Class *MT_PROPERTY*

A Matisse "property" is an attribute or a relation. Correspondingly, the Eiffel class *MT_PROPERTY* has heirs *MT_ATTRIBUTE* and *MT_RELATIONSHIP*.

This makes it possible from Eiffel to call the specific "check" routine to find out whether an object is valid.

2.7 Classes *MT_ATTACH*, *MT_ROUTINE*, *MT_SERVICE_FUNCTION*

One of Matisse's strong points is a flexible system of procedural attachments. The corresponding services (service functions, triggers, verification functions, entry points) are not yet supported because of a minor technical obstacle, which will be overcome in the next version.

It will be possible in that version to define procedural attachments from Eiffel and, as a result, to apply Eiffel operations as a result of requests made by the Matisse server. This facility has already been implemented on the Eiffel side, in class *MT_OBJECT*.

2.8 Classes *MT_INDEX* and *MT_INDEX_CRITERION*

Classes *MT_INDEX* and *MT_INDEX_CRITERION* make it possible to use an index that has already been defined in the Matisse database.

These classes are only useful for advanced uses. Note that index criteria are not used for queries because of the variable signature length of C functions. When you request the objects of an index, you will get all the corresponding instances, but in the requested order.

2.9 Class *MT_ENTRY_POINT*

Class *MT_ENTRY_POINT* makes it possible to access the database through a given predefined entry point. You can obtain the objects that are accessible from that entry point, or lock them.

2.10 Class *MT_MESSAGE*

Class *MT_MESSAGE* is not used in the EiffelStore interface, but makes it possible to obtain an object of type *MtMessage* through its name.

3 THE HIGH-LEVEL INTERFACE

The *matisse_interface* cluster provides the Matisse library's integration in the EiffelStore library, with the same interface as for other EiffelStore handles. The following paragraphs explain how to use Matisse from EiffelStore.

3.1 Managing the database and the transactions

To write an EiffelStore application using Matisse, you must first use the specific EiffelStore handle. This is done through the class *MATISSE_APPL* and its *login* creation procedure. That procedure has four arguments: host machine name, database name, priority of the transactions to come, maximum waiting time for queries. A zero value for these last two arguments means that the default values will be used. For the priority, you can also use constants *Mtmin_tran_priority* and *Mtmax_tran_priority* from *MATISSE_CONST*.

Then you should choose one of the following working modes, using procedure *set_mode* from class *MATISSE_APPL*:

- *NO_SELECTED_DATABASE*: minimal connection. No operations are possible on the database. This mode makes it possible to keep several contexts (database links).
- *SELECTED_DATABASE*: some operations are possible, but they include neither object reads nor object writes.
- *VERSION_ACCESS*: makes it possible to use a version in read mode, through its name, passed as the second argument. The list of available versions is available through instances of class *MT_TIME_STREAM*.
- *OPENED_TRANSACTION*: allows reading and writing on the latest version.

Finally you must use procedure *set_base* from class *MATISSE_APPL* to connect the handle.

Here is a summary of the required operations:

1 • Identification:

!! *appl*.*login* (*server_name*, *database_name*, *priority*, *maximum_waiting_time*)

2 • Choice of working mode:

appl.*set_mode* (*transaction_mode*, *version_if_necessary*)

3 • Handle connection:

appl.*set_base*

where *appl* is an instance of the class *MATISSE_APPL*.

This scheme governs the first access to the database. For subsequent ones it suffices to adapt the mode through the procedures *set_mode*, *set_wait*, *set_priority*.

The actual database connection is established through the *connect* procedure of class *DB_CONTROL*. The earlier call to *set_base* from class *MATISSE_APPL* took care of gathering all the information to make it available to an instance of the class *DB_CONTROL_MAT* dynamically connected to *DB_CONTROL*. In practice this means that *DB_CONTROL* automatically handles changes of working mode without any need for the software developer to call C routines directly:

1 • Connecting:

session.*connect*

2 • Operations on the database:

...

3 • Disconnecting:

session.*disconnect*

Disconnecting will put you in the *NO_SELECTED_DATABASE* mode; so does an error occurring during a transaction.

3.2 Accessing the database

All Matisse selections use the notion of stream; for each kind of selection you will use the corresponding kind of stream.

To select data, you may use either class *DB_SELECTION* or class *MT_STREAM* (or one of its descendants). In the first case, all of a stream's data are loaded in memory; in the second case, only a pointer to the stream is loaded.

Everything is set up as if the selections provided by the Matisse server were stored procedures; for that reason, the selection operations use instances of class *DB_PROC*.

To execute a selection operation, you will use the following scheme:

1 • Create the selection:

!! *selection*•*make*

2 • Create a query:

!! *query*•*make* (*query_type*)

3 • Set the selection values:

selection•*set_map_name* (*correspondence_name*, *value*)

4 • Execute the selection:

query•*execute* (*selection*)

During a selection, you may choose not to load the stream's objects into the final result, or to process them individually by associating an action to the instance of *DB_SELECTION*. In this latter case two specific operations are actually called:

- The procedure *start* is executed before the loading of the first object.
- The procedure *execute* is executed after the loading of each object.

You may stop the selection by setting the boolean value *found* in the corresponding action class.

The following table gives the list of possible selections with, for each selection: the query name; the equivalent function in Matisse's C API; the nature of the objects returned; the arguments to be passed to procedure *set_map_name* (in the form of a pair, whose first element is the type of the associated value or a constant name, and whose second element is the name of the association pair); and the operations that are permitted and prohibited during the opening of the stream.

Type	**Matisse counterpart**	**Functions**	**Arguments to** *set_map_name*	**Notes**
OCS	*OpenClassStream*	All instances of a class.	(*MT_CLASS*, *Class_map*)	Cannot create or delete instances.
OEPS	*OpenEPSteam*	Objects accessible through entry point.	(*STRING*, *Name_ map*) (*MT_ATTRIBUTE*, *Attribute_Map*) (*MT_CLASS*, *Class_map*)	Only: • Object creation. • Adding successors. • Removing successors/

Type	**Matisse counterpart**	**Functions**	**Arguments to** *set_map_name*	**Notes**
OIS	*OpenIndexStream*	All the index's objects in the chosen direction.	(*STRING*, *Name_map*) (*MT_CLASS*, *Class_map*) (*MtDirect*, *Direction_map*) ou (*MtReverse*, *Direction_map*)	Only: • Informations on indices
ORS	*OpenRelStream*	Successors in the relation.	(*MT_OBJECT*, *Object_map*) (*MT_RELATIONSHIP*, *Relation_ship_map*)	Only: • Object creation. • Modifying objects. • Removing values.
OOAS	*OpenObjAttStream*	Fields (attributes) of the object	(*MT_OBJECT*, *Object_map*)	Only: • Object creation. • Adding successors. • Removing successors/
OORS	*OpenObjRelStream*	Relations of the object	(*MT_OBJECT*, *Object_map*)	Only: • Object creation. • Modifying values. • Removing values.
OOIRS	*OpenObjIRelStream*	Inverse relations of the object	(*MT_OBJECT*, *Object_map*)	Only: • Object creation. • Modifying values. • Removing values.

For example, to select all instances of the Matisse class *Employee* using an action defined in class *CONTROL_ACTION*, an heir of class *ACTION*, you should execute:

```
selection: DB_SELECTION
your_class: MT_CLASS
query: DB_PROC
your_action: CONTROL_ACTION

!! selection.make
!!  query.make (Ocs)
!! your_class.make ("Employee")
selection.set_map_name (your_class, Class_map)
```

```
query.execute (selection)
!! your_action
selection.set_action (your_action)
selection.reset_cursor (your_cursor)
if selection.is_ok then
    selection.load_result
end
```

When using a container (that is to say, an instance of class *LINKED_LIST* [*DB_RESULT*] that has been associated with the selection to store the result into an object) you can obtain the objects through an assignment attempt. This work through the descendants of class *DATA*: *MT_OBJECT*, *MT_ATTRIBUTE* and *MT_RELATIONSHIP*.

Here is for example how to get all the attributes of an object:

```
!! selection.make
!! query.make (Ooas)
selection.set_map_name (your_object, Object_map)
query.execute (selection)
!! container.make
selection.set_container (container)
selection.load_result
from
    container.start
until
    container.off
loop
    attribute ?= container.item.data
    container.forth
end
```

3.3 Updating the database

To insert values into the database you will use class *MT_ATTRIBUTE*. As noted, type conversions are automatic. There is no specific EiffelStore support for inserting values, as this would make things too complicated.

3.4 Version streams

Class *MT_TIME_STREAM* makes it possible to obtain the names of the database versions currently being used. To open the stream you may use the creation procedure *make*, without argument; to obtain the version names, in the form of instances of *STRING*, you may use the procedure *next_time*; to close the database, the procedure *close*.

3.5 Repository and storage

An EiffelStore "repository" corresponds to a Matisse class. This makes it possible to load the definition of a Matisse class, generate the code of the corresponding Eiffel schema, check that a Matisse class conforms to an Eiffel class (will it be possible to store all the fields) and use *DB_ STORER* to store objects made of basic fields only. (See the next section about objects having reference fields.)

The following example shows how to load a Matisse class definition, check its conformance to an Eiffel instance, generate the class text, and store an object:

```
        -- Using the repository:
!! repository.make ("db_repository_mat test")
repository.load
tester.make
b:= repository.conforms (tester)
repository.generate_class
        -- Using the storer:
!! tester.make
!! storer.make
storer.set_repository (repository)
storer.put (tester)
```

4 OBJECT PERSISTENCE SUPPORT

Support for object persistence is a key facility for Eiffel users, making it possible to store entire object structures in a transparent way. Until now, the mechanism relied on class *STORABLE*, providing very high level storage and retrieval facilities under several formats. For more flexibility, the EiffelMatisse library provides a new version of this class, entirely written in Eiffel.

The corresponding cluster, *matisse_sr*, contains four parts:

- Object storage: descendants of *STORER*.
- Object retrieval: descendants of *RETRIEVER*.
- Traversal: *TRAVERSAL*, *TRAVERSAL_ACTION*.
- Internal table management.

This new version of *store* and *retrieve* offers the following benefits:

- The cluster's organization is clearly defined, and each of the functions is easy to find. For example the algorithms for file storage are present in descendants of *FILE_STORER*, which implements standard features (providing a device of type *IO_MEDIUM*, object traversal, object marking, writing a file header etc.).
- Storage and retrieval are customizable. You can control what objects will be stored, change the output format, change the writing algorithms, and similarly for retrieval. To perform such customization it suffices to redefine the routines of class *TRAVERSAL_ACTION* in a

descendant of *EXT_STORABLE* so as to change the default action executed during the recursive marking of the graph of storable objects.

- Everything is written in Eiffel. This makes it possible to add new functionalities much more easily.
- The "basic" store is functionally identical to the standard *basic_store*.
- Small tests, using identical assumptions (no assertion checking, no inlining) seem to indicate a performance similar to that of the current C version.

Whereas actual storage support have little relevance for developers, what counts most is the ability to store and retrieve objects in the simplest possible way. This is why the library offers a single interface for all possible media. Whether for a file or for a database, it suffices to use the following scheme:

1 • Storage

your_object: *TYPE* -- Must be a descendant of *EXT_STORABLE*
...
your_object.*set_..._storer* -- *basic* or Matisse
your_object.*store* (*arguments*) -- File name or key

2 • Retrieval

your_object.*set_..._retriever* -- *basic* or Matisse
your_object ?= *your_object*.*retrieve* (*arguments*) -- Name or key

4.1 Storing into flat files

4.2 Selective storing

4.3 Storing into a database

4.4 Managing shared objects

5 EXTENSIONS

5.1 Automatic generation of ODL files