1 Foreword

This document describes an Eiffel library providing facilities for interfacing an Eiffel application with various data persistency mechanisms.

The library is designed as a virtual layer on top of different implementation schemes, ranging from object paging facilities to uniform interfaces to different types of data base management systems.

This document groups together reference information about the architectural and class design as well as samples and principles to guide the use, integration and extension of it.

2 Introduction

The aim of ***EiffelStore*** is to provide a consistent set of library classes for writing Eiffel applications that handle persistent data.

What is a persistent data? A persistent data represents some information of which life span extends beyond the execution time of the application that uses it.

Software applications manipulate persistent data of different nature: flat files, repositories, data dictionaries or databases.

Because of the specific representation used in each data storage, it is quite common to end up writing dedicated software pieces dealing with stored data. This task requires the use of specialized languages and some knowledge on the external storage format used. In the database field for instance, it is needed to design the way the external data will be stored and then queried using a dedicated language.

In an object-oriented world, the impedance mismatch between the application data and the persistent data is considerably reduced. The ultimate goal is to offer an execution model where transient and persistent objects are manipulated in a similar way.

The objective of the ***EiffelStore*** is twofold: first interfacing with existing databases and second extending predefined persistency mechanisms.

Regarding the coupling to existing databases, the most frequent situation concerns Eiffel applications that must manipulate existing data, derived from a legacy system or simply maintained by systems that may not easily migrate to object based representations.

Ideally, persistency should require minimal changes at the language level, and consequently at the application developer's level, but rather put the burden on the run-time system. In addition to in-built memory allocation and management facilities, one may expect at some point a virtual object space manager that would transparently page in and out clusters of objects. An implementation trade-off is a way of transforming Eiffel classes and objects into representations compatible with existing object-oriented database management systems.

The Eiffel4 kernel library with class *STORABLE* already supplies an in-built mechanism that makes possible the storage and retrieval of object networks into and from flat files. From a reference to a persistent root, the transitive closure of objects accessible from that root is stored on file. The description of the system of classes that lead to the stored objects is currently not part of the stored information.

EiffelStore handles data persistency as the ability to store and retrieve Eiffel objects from and to external sources regardless of the external format used.

The ***EiffelStore*** interface consists of several levels, ranging from a general set of primitives available through a wide range of object management systems, to more specific primitives customized to different external data formats.

The presentation of ***EiffelStore*** is structured as follows:

- A first section introduces the global architecture with the different clusters and the structuring in abstraction levels.
- Further sections explain how to use the library, based on some sample outlines.
- A final section lists the comprehensive interface description of the visible classes, grouped by cluster.

3 Class layers

3.1 INTERFACING WITH EIFFELSTORE

EiffelStore is a set of self-contained Eiffel classes that uses kernel and library classes of ***ISE-Eiffel4***. In addition, it requires a self-contained date and management cluster that comes with the delivered library classes and some external low-level C libraries.

An Eiffel application using ***EiffelStore*** refers to classes grouped into layers representing different levels of abstraction. The two main layers are introduced below:

1. The ***Interface*** layer encapsulates general primitives that manage the object manipulation session, and deal with high-level persistency primitives. It links an Eiffel application with primitives handling access to data base objets regardless of their external representation.

2. The ***Implementation*** layer customizes high-level routines to various DBMS handles. It allows the mapping between Eiffel objects and persistent data originating from various external formats. This layer consists of classes that give access to implementation details. Supporting a new DBMS often means writing a new set of classes without changing the ***Interface*** layer.

At session start up time, a user's application class selects a class name corresponding to the data base management systems to be used. The coupling is made using either an inheritance or a client-supplier relationship between the application class and ***EiffelStore***.

Classes bridging user's applications with ***EiffelStore*** are named *XXX_APPL*, where *XXX* stands for a data storage mean, such as *ORACLE* or *SYBASE* for instance. Class *XXX_APPL* is called the *handle* class between ***EiffelStore*** and its associated DBMS.

In any case, classes *XXX_APPL* are only to be looked at. Whenever an application needs to interface multiple handles, it will simply link to several *XXX_APPL* handle classes.

3.2 INTERFACE LAYER

The *Interface* layer groups high-level visible classes. Theses classes are in charge of the session management, and persistent data access and change.

At any time, inside any possible layer, persistent objects are stored in so-called *repositories*. The notion of repository is defined by class *REPOSITORY*.

In some cases, the use of the *REPOSITORY* class is totally transparent to the application using ***EiffelStore***, in other cases it is not.

For instance, whenever the application needs to access information on *the data container* in addition to the data itself, then it will be using class *REPOSITORY*. It will be same whenever an application creates a new repository, ore more generally changes the database schema. In this case too, class *REPOSITORY* is used.

When ***EiffelStore*** is hooked up to relational database, repositories are mapped to relational tables defined in the database schema. Therefore, it may very well be that case that primitives offered by class *REPOSITORY* are not useful for selecting or changing the data they contain. Therefore it will not be necessary to create and use *REPOSITORY* objects.

Session management primitives are grouped together into class *DB_CONTROL*. Four basic primitives control the data management system: ***connect, disconnect*** to trigger the activation of the underlying data server and ***commit, rollback*** to validate or discard the effect of data modifications in the working space since the last physical update occurred.

Persistent data manipulation capabilities are grouped into classes focusing on a specific capability.

- Class *DB_CHANGE* modifies data repositories. This class is used when performing data insertion or deletion for instance.
- Class *DB_SELECTION* accesses external persistent data using a query mechanism. This class is simply used to access or read the data. No date change is expected when using this class. After making a selection, it is possible to transform the result into an Eiffel object.
- Class *DB_RESULT* encapsulates primitives representing the format and the value of the data retrieved with *DB_SELECTION*.
- Class *DB_STORE* prepares the storing of Eiffel objects into repositories. In the case the external format is limited to predefined data types, only object attributes mapping these types are stored. For instance, in the case ***EiffelStore*** is coupled with an object-oriented database, the stored object is considered as a persistent root. If the underlying database is relational, only Eiffel types supported in the relational base will be considered for storage.

3.3 IMPLEMENTATION LAYER

Classes implementing the coupling between interface classes and RDBMS libraries are grouped in the DBMS cluster. Inside the DBMS cluster, each handle is implemented by a set of classes with a name referring to the specific handle. These classes make external calls to library functions supported by the database server. Supporting a new handle means rewriting an equivalent set of classes making external function calls to a different C library.

4 RDBMS coupling

This section describes more specifically the interface of ***EiffelStore*** with relational databases. Regardless of the selected or supported handle, the set of available primitives remain the same as long as the application interfaces with classes originating from the *interface* cluster and not the *implementation* cluster. Classes belonging to the *interface* cluster will be detailed below.

When starting a session, a user's application selects the relational data base management system to use, either by inheritance or with a client-supplier relationship (see the first programming example in next section).

This mechanism conveniently decouples the ***EiffelStore*** uniform interface level from several specific implementation levels, also called ‘‘*handles*’’.

The connection between the interface definition level and one or several selected handles remains under the programmer's control.

We will review in the sequel different possible uses of the available primitives. This is an insight of their capabilities:

- Accessing existing data bases from the Eiffel object side to update or query data base entities with an SQL-like interface;
- Creating a dynamic object model from a subset of an external data repository to manipulate the relational data from inside an Eiffel application;
- Making selected Eiffel objects persist into a relational data base to take advantage of all possible manipulation features offered by a RDBMS.

4.1 INTERFACE WITH THE DB SERVER

Whenever an Eiffel application wants to invoke SQL queries on RDB tables from the Eiffel side, ***EiffelStore*** offers a set of primitives that rely on the use of SQL expressions. Since SQL is standardized but usually offers extensions specific to a RDBMS, ***EiffelStore*** simply passes SQL expressions to the server. The implementation technique relies on so-called ‘‘dynamic SQL statements’’ since queries are built and interpreted at execution time.

Most relational servers provide the following options, switchable from class *DB_CONTROL*:

- Execute a statement stored in a buffer: *execute immediate* option;
- Encode a statement stored in a buffer and execute it several times: *prepare* and *execute* options;

- Obtain information on a table at run time: *prepare* and *describe* options.

EiffelStore provides two operating mode: the *immediate execution* mode and the *non-immediate (or delayed) execution* mode.

These modes are set using primitives *set_immediate* or *unset_immediate* defined in class *DB_CONTROL*. The default is non-immediate execution mode.

The *immediate* mode is most useful for SQL statements that are executed only once and do not acquire information from the database. Statements including a search condition for instance, do not apply for this mode.

The *non-immediate* execution mode is implicitly used with SQL statements involving search conditions. In this case, an SQL statement is wrapped up automatically by ***EiffelStore***, (doing a *prepare*), then appropriate descriptors are set up with the cache buffer filled in, user defined operations applied on selected objects, retrieved objects translated into C structures, etc.

The general Eiffel template for starting the connection to the database server and controlling the session is given in figure 4.1.

```
class SESSION
feature
        db_handle: ORACLE_APPL
        session_control: DB_CONTROL

        make is
                do
                        !!db_handle.login (...)
                        db_handle.set_base

                        !!session_control.make
                        session_control.connect
                        -- ....
                        -- <Various Data Queries and Manipulations>
                        -- ....
                        session_control.commit
                        session_control.disconnect
                rescue
                        session_control.roll_back
                        session_control.disconnect
                end
                ...
end -- class SESSION
```

Figure 4.1 Session start-up

In the code excerpt of figure 4.1, the **rescue** clause is used to recover from unexpected events. Under such condition a *roll_back* operation is invoked and the session terminates. Another possibility would be to program a **retry** operation and start over again.

Feature *set_base* must be called before creating or using any data base operation. After a call to *set_base*, the base becomes active.

One may also handle concurrently multiple relational bases. The example of figure 4.2 illustrates such situation.

```
class MULTIPLE
creation
        make
feature -- Status report

        oracle_session: ORACLE_APPL
                -- One ORACLE connection

        sybase_session: SYBASE_APPL
                -- One SYBASE connection

feature -- Initialization

        make (name, password: STRING) is
                        -- Handle multiple database connections.
                do
                        !!oracle_session.login_and_connect (name, password)
                        !!sybase_session.login_and_connect (name, password)
                end

feature -- Element change

        oracle_activate is
                        -- Make session talk to ORACLE server.
                do
                        oracle_session.set_base
                end

        sybase_activate is
                        -- Make session talk to SYBASE server.
                do
                        sybase_session.set_base
                end

end -- class MULTIPLE
```

Figure 4.2 Multiple sessions

4.2 REPOSITORIES

When dealing with relational bases, repositories map relational tables. These tables either already exist in a so-called database schema or can be created by ***EiffelStore***.

Using repositories is needed to check for instance the presence or the absence of a certain table in a schema. This is illustrated in figure 4.3.

It is important to note that, from a RDBMS standpoint, loading a repository usually requires a *costly* operation since it goes through the entire set of existing tables of the data base schema, which may eventually be quite big.

Therefore, class *REPOSITORY* should mostly be used to map Eiffel objets and table fields in one way or the other:

- Whenever a query result must be transformed into an Eiffel object. In this case, it may be useful to rely on class *REPOSITORY* to check the compatibility between the table column names and types and the object attribute names and types; although it is not absolutely necessary if the application knows exactly what it does.

```
class SCHEMA
feature

        repository: DB_REPOSITORY

        make is
                do
                        !!repository.make ("VIEW_NAME.TABLE_NAME")
                        repository.load
                        if not repository.exists then
                                io.putstring ("Table does not exist.")
                        end
                end

end -- class SCHEMA
```

Figure 4.3 Repository use

- Whenever an Eiffel object is flattened into a table row. In this case, it is required to create and load an instance of class *REPOSITORY* first.

To sum up, whenever ***EiffelStore*** updates, queries or modifies a RDB schema without any Eiffel object transformation, class *REPOSITORY* may not be used.

4.3 SELECTION QUERIES

When dealing with a RDB, one manipulates tables. Tables correspond to the notion of ‘‘repository’’ in ***EiffelStore***, even if, as we just said earlier, they may not be always activated.

RDB tables fields can potentially be transformed into Eiffel class instances. It is usually assumed that the application is aware of the type of objects that will be manipulated. This is usually the case when transforming a relational schema into an object schema to compute results or display information. This is usually not the case when invoking session management, deletion or insertion primitives.

The result of a selection translates, in the most general case, into two-dimension data in the range [0-m, 0-n].

As an example, assume that a relational base contains two tables, an *occupancy* table and a *faculty* table defined and set as given in figure 4.4.

An SQL query performing a **select** and a **join** on two tables is written as follows:

select *occupancy.room_number, faculty.teacher* **from**
occupancy, faculty **where** *occupancy.course = faculty.course;*

This results into a temporary table of figure 4.5, of dimension [2,2], of which rows

Room	Course
101	Maths
107	Chemistry
110	Administration
102	Restaurant

Course	Teacher
Physics	Peter
Maths	Bob
Chemistry	Jack

Figure 4.4 Relational tables

are accessible through a selection process detailed further on.

On the Eiffel side, any SQL statement involving variables such as a *field* name, a *table* name, a *row* to insert or a *seach* condition, can be expressed directly with values or, in a more generic manner using *bind* variables.

Room	**Teacher**
101	Bob
107	Jack

Figure 4.5 Selection result

A bind variable reference, in an ***EiffelStore*** SQL expression, is used with the following syntax:

<BindVariableReference> ::= **:***<Identifier>*

All bind variable names are prepended with a semicolon character. SQL string expressions that may contain bind variables, are passed by Eiffel to the relational base. These expressions are first parsed. Then references to bind variables are replaced with their effective value.

Each bind variable must be first declared and associated to a value or a reference. A bind variable owns a value if it is mapped to an Eiffel entity declared of type *INTEGER, REAL, DOUBLE, BOOLEAN* or *CHARACTER*. A bind variable owns a reference if it is mapped to an Eiffel entity declared of type *ABSOLUTE_DATE, STRING* or class.

Example of figure 4.6 illustrates how to map and use a bind variable.

```
selection: DB_SELECTION
!!selection.make
io.readline
selection.set_map_name (io.laststring, "author_string")
selection.query
  ("select * from BOOK_COLLECTION where AUTHOR = :author_string")
selection.unset_map_name ("author_string")
 ...
```

Figure 4.6 Binding variables

In the example of figure 4.6, bind variable *author_string* is mapped to the last input string refered to by *io.laststring*. Before the SQL expression is sent to the server, *:author_string* is replaced by its associated value declared between single quotes. Namely, the server gets passed in an SQL expression similar to:

```
SELECT * FROM BOOK_COLLECTION WHERE AUTHOR = 'EINSTEIN';
```

Whenever a bind variable is attached to an object reference, the substitution of the bind variable consists of a list of fields values separated with a comma.

4.4 DATA MANIPULATION

After making a selection, several options are offered to deal with the resulting table. An application may chose among the following cases:

- Step sequentially through the resulting rows;
- Iterate through all resulting rows until a certain condition is met;
- Iterate through all resulting rows until a certain condition is met and then apply an operation of each row;
- Store each resulting row into a data structure;
- Transform resulting rows into Eiffel objects.

The simplest way to make a SQL query is to invoke primitive *query* on an instance of class *DB_SELECTION* and then to pass in as argument a SQL expression string, as outlined in figure 4.7.

```
selection: DB_SELECTION
!!selection.make
selection.query ("select OCCUPANCY.ROOM_NUMBER, FACULTY.TEACHER %
      %from OCCUPANCY, FACULTY where %
      % OCCUPANCY.COURSE = FACULTY.COURSE")
```

Figure 4.7 Querying the base

The result of the query, applied on a *DB_SELECTION* object, returns a reference to a *cursor*, declared of type *DB_RESULT*.

A *cursor* corresponds to an elementary selection result. Referring to our previous example, it corresponds either to row {*101-Bob*} or to row {*107-Jack*}.

The primitive used to fetch the data and load it into the cursor is *load_result*. This routine automatically creates a cursor is it is still set to *Void* (as the postcondition says). The *cursor* attribute may be set using primitive *reset_cursor*. This is needed for primitives that require as precondition, the existence of a cursor.

The typical sequence of actions used to select and iteratively execute some action each time a cursor is loaded, as long as there is a resulting row, is given in figure 4.8.

Class *DB_RESULT* is rather sketchy: at the interface level, there is very little to say about the general format of the data returned from the base; whichever this database is. In practise, in the case the application needs to access table format information, one must use an instance of class *DB_TUPLE*. It then becomes possible

```
...
selection: DB_SELECTION
!!selection.make
...

selection.query ("...")
selection.load_result

-- <cursor is now loaded and 'is_exiting' set to true>
...
```

Figure 4.8 Loading selection cursor

to access more specific features such as column names and the row dimension. This is outlined in figure 4.9.

In addition to making selection and getting cursor values, an Eiffel application may need to store intermediate selection results into a data structure on which parameterized operations can be applied.

```
...
selection: DB_SELECTION
tuple: DB_TUPLE
...
!!tuple
tuple.copy (selection.cursor)
if tuple.count >= 2 and then tuple.column_name (2).is_equal ("COURSE") then
    ...
end
...
```

Figure 4.9 Accessing tuple values

This usually happens when a result is pulled off, changed or used to produce some computation with no intention to propagate the modifications back into the data base.

In such case, it suffices to pass a reference to a storage structure to a *DB_SELECTION* instance. The storage data structure must conform to *LINKED_LIST* [*DB_RESULT*], as illustrated in figure 4.10.

```
selection: DB_SELECTION
!!selection.make
...
container: LINKED_LIST [DB_RESULT]
!!container.make
selection.set_container (container)
selection.query ("...")
selection.load_result
check
    selection.is_exiting
end
...
```

Figure 4.10 Storing selection result in a container list

If no reference is passed, the *container* attribute remains set to *Void* and no result storage is made (this is the default).

It is also possible to control the data loading operation using attribute *stop_condition*, which is set with primitive *set_action* before *load_result* is called.

Routine *set_action* takes as input argument a descendant class of *ACTION*. The descendant class is free to redefine routine *execute* invoked each time a row is processed, and routine *status* used to force an exit condition while the resulting rows are sequentially stepped through.

Note that attribute *stop_condition* refers to a descendant of class *ACTION* at execution time.

The code of figure 4.11 outlines how to put this to work.

```
class CONTROL_ACTION
inherit
	ACTION
		redefine
			execute, status
		end
feature

	execute is
		do
			...
		end

	status: BOOLEAN is
		do
			...
		end

	...
end -- class CONTROL_ACTION

class QUERY_CONTROL
feature
	...
	cursor: DB_RESULT
	selection: DB_SELECTION
	some_action: CONTROL_ACTION
	...
	make is
		do
			!!cursor; !!action
			!!selection.make
			selection.set_action (some_action)
			selection.query ("...")
			if selection.is_ok then
				selection.load_result
			end
		end
	...
end -- QUERY_CONTROL
```

Figure 4.11 Action call back on row value fetch

4.5 MODIFICATION QUERIES

To modify table contents, drop tables or perform any other operation that does not require access to a result, one should use class *DB_CHANGE*.

Class *DB_CHANGE* provides a routine *modify* taking as input argument any SQL string used to update the base.

Example given in figure 4.12 lists different invocations of primitive *modify* of class *DB_CHANGE*.

```
modification: DB_CHANGE
!!modification.make
modification.modify ("update view_name.TABLE_NAME %
        %set COLUMN_I = xxx, COLUMN_J = yyyy %
        %where COLUMN_A = zzzz and COLUMN_B = tttt")
modification.modify ("insert into view_name.TABLE_NAME %
        %values ('AAA', 23.4) ")
modification.modify ("delete view_name.TABLE_NAME")
...
```

Figure 4.12 Table modifications

4.6 TABLE RECORDS AND EIFFEL OBJECTS

Record fields of relational tables can be mapped to Eiffel class attribute fields.

Conversely, it is possible to create in the relational schema a new table mirroring the structure of an object.

The mapping between table rows and Eiffel objects is limited though to a set of predefined types corresponding to those supported to most relational bases. What may change from one server to another is the naming convention of these predefined types. ***EiffelStore*** integrates the different conventions adopted by supported bases and manages the transformation between a column table type and an Eiffel object attribute type, using a double matching rule:

- A table column and an object attribute must match in name.
- A table column and an object attribute must be compatible in type.

The set of Eiffel types (class names) usable for data conversion between table rows and object fields are the following:

INTEGER, REAL, DOUBLE, BOOLEAN, CHARACTER, STRING, ABSOLUTE_DATE.

Tables involved in the data transformation can be part of a persistent or a transient relational table; namely may correspond to rows resulting from either **select** or **join** operations.

To transform the selection result pointed to by *cursor* in class *DB_SELECTION* into Eiffel object instances, it suffices to invoke feature *object_convert* and pass as input argument a reference to an Eiffel object to be filled in.

Then after a selection is completed, routine *cursor_to_object* maps the resulting column values into Eiffel object fields.

The number of attributes of the Eiffel object must be greater or equal to the dimension of the row (number of columns).

Each candidate attribute is set with the value of the column having the exact same name as the Eiffel class attribute.

Assuming we have in our schema a relational table defined as in figure 4.13.

The Eiffel class outlined in figure 4.14 is then a good candidate class for having its instances receive the values of the table record fields, because two of its attributes match the table column names and types.

In addition, to ensure that an instance of class *ASSIGNMENT* will match our relational table, one may use a sanitary check as outlined in figure 4.15.

```
Name                             Type
-------------------------------- ----
ROOM_NUMBER                      NUMBER(13)
TEACHER_NAME                     CHAR(30)
```

Figure 4.13 Table ASSIGNMENT

```
class ASSIGNMENT
feature

    room_number: INTEGER
    teacher_name: STRING
    nb_attendees: INTEGER

    get_attendees is
            -- Update 'nb_attendees'.
        do ...
        end

    set_room_number (n: INTEGER) is
        do
            room_number := n
        ensure
            room_number = n
        end

    set_teacher_name (last_name: STRING) is
        require
            last_name /= Void
        do
            teacher_name := last_name
        ensure
            teacher_name = last_name
        end
    ...
end -- class ASSIGNMENT
```

Figure 4.14 Eiffel class used for mapping a selection result

```
one_assignment: ASSIGNMENT
repository: DB_REPOSITORY

!! one_assignment
!! repository.make ("ASSIGNMENT")
repository.load
if repository.conforms (one_assignment) then
    <... Proceed as before ...>
else
    < ... Error handling ...>
end
```

Figure 4.15 Mapping validity check

Once again, it is important to note that loading a repository often requires operations that may degrade system performance. Therefore, it is the designer's responsibility to evaluate the appropriateness of the controls suggested in figure 4.15.

The conversion procedure is straightforward. Attributes *teacher_name* and *room_number* defined in class *ASSIGNMENT* are assigned with table column values, whilst attribute *nb_attendees* remains untouched. This is outlined in figure 4.16.

```
selection: DB_SELECTION
 ...
!!selection.make
selection.object_convert (one_assignment)
selection.query ("select * from ASSIGNMENT")
selection.load_result
selection.cursor_to_object
 ...
```

Figure 4.16 Object conversion

Class *ASSIGNMENT* template can also be automatically generated on the standard output to avoid tedious and error-prone manual programming, specially when table rows become rather complex. The generated Eiffel code can then be enriched with additional primitives and later participate to the coupling of table values and object fields.

The program excerpt of figure 4.17 prints out a class description that may be reused and adapted inside an Eiffel system.

```
repository: DB_REPOSITORY
!!repository.make ("ASSIGNMENT")
repository.load
repository.generate_class
 ...
```

Figure 4.17 Class generated from table description

An example of the generated class test, resulting from the invocation of routine *generate_class* is given in figure 4.18. The default output is the standard output.

```
class ASSIGNMENT
feature

   room_number: INTEGER
   teacher_name: STRING

end
```

Figure 4.18 Result of class generation

4.7 STORING AN EIFFEL OBJECT INSIDE A TABLE

Whenever table rows must be added with column values corresponding to Eiffel object attribute values, the simplest way is to rely on class *DB_STORE*.

In this case, it is requested to declare and load a repository mapping the relational table name.

First, an instance of *DB_STORE* must be linked to a repository object of which column set or subset conforms to an Eiffel object. Then, a flattened representation of the Eiffel object is inserted into the table, as long as the number of object fields with names identical to table column names remains greater or equal to the dimension of the table.

The repository name attached to the storage handler must be identical to the table name of the relational schema. The repository must also be loaded before any insertion begins.

A typical sequence of operation corresponding to a table row insertion with column values mapping an Eiffel object attribute values is given in figure 4.19.

```
...
storage: DB_STORE
assignment_table: DB_REPOSITORY
one_assignment: ASSIGNMENT

!!assignment_table.make ("ASSIGNMENT")
            -- 'ASSIGNMENT' is the RDB table name.
...
assignment_table.load
...
!!store.make
        -- One may store if and only if the
        -- store is attached to a repository.
storage.set_repository (assignment_table)
...
        -- Following instruction may be invoked, for instance,
        -- after 'one_assignment' attributes have been changed.
storage.put (one_assignment)
...
```

Figure 4.19 Table row insertion

4.8 STORED PROCEDURES

Whenever a request is sent to a database server, it is first needed to parse the SQL expression before it is sent to the server. In database applications organized in client-server architectures, the parsing of the SQL expression is often left to the client whereas the execution of the request itself remains under the control of the server.

In the case the same request is repeatedly executed, with only changes in expression values, reparsing over and over again the same SQL statement means a waste of time.

Therefore, some database servers now offer the ability to store precompiled SQL statements on the server side. These precompiled statements can be invoked by the client in a way that resembles a routine call. These routine are often named *stored procedures* and are callable by their name with possible arguments. Stored procedures are defined by a name, a signature and a body which is the SQL statement to be executed whenever the stored procedure is invoked. They are treated like usual data since they can be queried, stored or deleted.

EiffelStore provides a common interface to the definition and invocation of stored procedure with class *DB_PROC*. Features defined in class *DB_PROC* are only applicable in the case the ***EiffelStore*** handle is hooked up to a database server that supports such capability.

On the Eiffel side, stored procedure are defined with a name. This name maps a stored procedure name existing in the database server. Instances of class *DB_PROC* lead to objects that potentially represent a stored procedure. To make the linkage between an effective stored procedure and a corresponding Eiffel object, one must invoke routine *load*. The only condition for *load* to succeed is that the session is active. The boolean function *exists* test the existence of the stored procedure. The program excerpt given in figure 4.20 illustrates these different conventions.

```
...
query_procedure: DB_PROC
...
        -- Create an Eiffel object mapping a stored procedure
        -- of name ‘‘query_client’’
!!query_procedure.make ("query_client")

        -- Load stored procedure with ‘name’ equal to "query_client"
query_procedure.load
        -- Test of presence
if query_procedure.exists then
        <process as usual>
else
        <print error message>
end
...
```

Figure 4.20 Database stored procedure and EiffelStore linkage

Stored procedure are entered in the server base using the *store* primitive that takes as argument a SQL expression. The SQL expression may contain formal arguments defined by a name prepended with the semicolon character. The linkage between the formal argument name and its effective value is achieved using the same scheme introduced earlier regarding bind variables.

The program excerpt given in figure 4.21 outlines the definition of a procedure body and the binding of a formal argument to an Eiffel entity. Primitive *set_arguments* takes two manifest arrays as arguments. The first argument is a set of strings corresponding to formal names used in SQL statement expressions. The second argument associates the formal name with a value taken by Eiffel program entities.

The execution of a store procedure takes as input argument a reference to a *DB_SELECTION* or *DB_CHANGE* object. The argument should be a

```
last_name: STRING
 ...
query_procedure.store ("select pname, paddress, vmodel, acqudate %
            %from CUSTOMER_TABLE, VEHICLE_TABLE %
            %where CUSTOMER_TABLE.ccode = VEHICLE_TABLE.ccode %
            %and CUSTOMER_TABLE.pname = :key")

query_procedure.set_arguments (<<"key">>, <<last_name>>)
 ...
```

Figure 4.21 Stored procedure body definition

DB_SELECTION object in the case the stored procedure is a selection query whereas the argument should be a *DB_CHANGE* object in all other cases where no result is expected.

```
customer: CLIENT
 ...
selection: DB_SELECTION
!!selection.make
selection.set_map_name (customer.last_name, "key")
 ...
query_procedure.execute (selection)
selection.load_result
      -- All standard operations are now accessible from 'selection'

 ...
selection.unset_map_name ("key")
 ...
```

Figure 4.22 Execution of a stored procedure

In conclusion, using a stored procedure replaces the use of the *query* operation defined in class *DB_SELECTION*. In the case the stored procedure has arguments, in addition to the definition of their name and types, one must map their effective values using routines *set_map_name* and *unset_map_name*.

4.9 ERROR HANDLING

Error handling features are accessible through class *DB_CONTROL* and introduced in one of its ancestor class.

The internal integer error code returned by the server is given by feature *error_code*. The corresponding error message or warning message string is retrieved with feature *error_message* or feature *warning_message* respectively. Error codes and messages are database specific.

Since each query updates the error code if any, it is possible to systematically test the success or failure of the last executed transaction with the boolean function *is_ok*. If an error occurs, ***EiffelStore*** does not reset the error code and *is_ok* function keeps returning **false**. It is the application responsibility to invoke *reset* after performing some error handling operation.

In addition to this, the boolean function *is_connected* checks whether the last RDB server connection succeeded or not.

The *disconnect* operation, once called, automatically clears the last error code.

5 DBMS specifics

5.1 ORACLE SPECIFICS

Class *ORACLE_APPL* makes the connection between the end user application and the ***EiffelStore*** library.

The interface is consistent with ORACLE RDBMS V6/V7

To run with the *Oracle* interface, ***EiffelStore*** needs a server to be started.

Users must set their environment variable with $ORACLE_HOME telling the access path to Oracle installation.

When setting up your working environment with ORACLE, the following environment variables may need to be set:

ORACLE_HOME, ORACLE_SID, ORAKITPATH, ORATERMPATH, ORA_SERVER

Then, check your path as follows (C-Shell):

```
set path = ($path $ORACLE_HOME/bin)
```

Then start the Oracle database server. A excerpt of such session is given below (from an Oracle V6 session):

```
$ % su oracle
$ oracle% sqldba
$ SQL*DBA: Version 6.0.33.1.1 - Production on Wed Jul 29 17:53:21 1992
$
$ Copyright (c) Oracle Corporation 1979, 1989.  All rights reserved.
$
$ ORACLE RDBMS V6.0.33.1.1 - Production
$ PL/SQL V1.0.32.3.1 - Production
$
$ SQLDBA>startup
$ ORACLE instance started.
$ Database mounted.
$ Database opened.
$ Total System Global Area        776264 bytes
$             Fixed Size         26348 bytes
$          Variable Size        307548 bytes
$       Database Buffers        409600 bytes
$           Redo Buffers         32768 bytes
$ SQLDBA>exit
$ oracle%^D
```

To shutdown the Oracle database server, proceed as follows:

```
$ %su oracle
$ oracle% sqldba
$
$ SQL*DBA: Version 6.0.33.1.1 - Production on Wed Jul 29 17:53:21 1992
$
$ Copyright (c) Oracle Corporation 1979, 1989.  All rights reserved.
$
$ ORACLE RDBMS V6.0.33.1.1 - Production
$ PL/SQL V1.0.32.3.1 - Production
$
$ SQLDBA> shutdown
$ Database closed.
$ Database dismounted.
$ ORACLE instance shut down.
$ SQLDBA> exit
$ oracle% ^D
```

Now you need to check that your Eiffel program works properly in connection with your base.

Under Oracle, a special account is already set up for you usually:

```
Name: scott
Password: tiger
```

Following is an excerpt of an interactive session.

```
$ user% sqlplus scott/tiger
$
$ SQL*Plus: Version 3.0.9.1.2 - Production on Wed Jul 29 18:02:58 1992
$
$ Copyright (c) Oracle Corporation 1979, 1989.  All rights reserved.
$
$
$ Connected to:
$ ORACLE RDBMS V6.0.33.1.1 - Production
$ PL/SQL V1.0.32.3.1 - Production
$
$ SQL> ....
```

Then, you may type in a few SQL queries and statements simply to check the effect of your Eiffel program.

To couple the Oracle RDBMS with EiffelStore, you need to add the following external object libraries in your ACE file:

With Oracle V6:

```
$EIFFELSTORE/spec/$PLATFORM/libsupport.a
$EIFFELSTORE/spec/$PLATFORM/liboracle.a
$EIFFELSTORE/spec/$PLATFORM/libsys_time.a
$ORACLE_HOME/rdbms/lib/libpro.a
$ORACLE_HOME/rdbms/lib/libpro.a
$ORACLE_HOME/rdbms/lib/libsql.a
$ORACLE_HOME/rdbms/lib/libocic.a
$ORACLE_HOME/rdbms/lib/libsqlnet.a
$ORACLE_HOME/rdbms/lib/libora.a
$ORACLE_HOME/rdbms/lib/osntab.o
```

With Oracle V7:

```
$EIFFELSTORE/spec/$PLATFORM/libsupport.a
$EIFFELSTORE/spec/$PLATFORM/liboracle.a
$EIFFELSTORE/spec/$PLATFORM/libsys_time.a
$ORACLE_HOME/lib/libsql.a
$ORACLE_HOME/lib/osntab.o
$ORACLE_HOME/lib/libsqlnet.a
$ORACLE_HOME/lib/libora.a
$ORACLE_HOME/lib/libpls.a
$ORACLE_HOME/lib/libnlsrtl.a
$ORACLE_HOME/lib/libcv6.a
$ORACLE_HOME/lib/libcore.a
```

In case your system has no $ORACLE_HOME/rdbms/lib/osntab.o, you may generate it anywhere you want as follows:

```
genosntab > osntab.c
cc -c osntab.c
```

or just take a look to the Oracle Clib Makefile.

5.2 SYBASE SPECIFICS

Class *SYBASE_APPL* makes the connection between the end user application and EiffelStore library.

The interface is consistent with SYBASE V4 and V10

To run with the *Sybase* interface, *EiffelStore* needs a server to be started.

Users must set their environment variable with $SYBASE telling the path to the Sybase installation directory.

Then, update your path as follows:

```
set path = ($path $SYBASE/bin)
```

First check that you have a Sybase SQL server installed. This is done in Sybase, using the utility **sybinit** that configures SQL Server and other Sybase products.

To start the Sybase server, do the following:

```
startserver
```

Then set the system administrator account password. The default is *null*. You may first test try your installation using the **isql** interactive SQL to create, query or modify tables as follows:

```
isql –U sa –P
> select * from test
> go
```

To shut down the server, do the following:

```
isql -U sa -P
> shutdown
> go
```

To couple the Sybase RDBMS with EiffelStore, you need to add the following external object libraries in your ACE file:

```
$EIFFELSTORE/spec/$PLATFORM/libsupport.a
$EIFFELSTORE/spec/$PLATFORM/libsybase.a
$EIFFELSTORE/spec/$PLATFORM/libsys_time.a
$SYBASE/lib/libsybdb.a
```

6 Class interfaces

6.1 INTERFACE OF CLASS DB_CHANGE

```
indexing
        status: "See notice at end of class"
        date: "$Date$"
        revision: "$Revision$"
        access: change, modify, update, insert, delete
        product: eiffelstore
        database: all_bases

class interface
        DB_CHANGE

creation
        make

feature -- Basic operations

        execute_query
                        -- Execute modify with last_query.
                require -- from DB_EXPRESSION
                        last_query_not_void: last_query /= void

        modify (request: STRING)
                        -- Execute request to modify persistent objects.
                        -- When using the DBMS layer the request must be
                        -- SQL-like compliant.
                require
                        connected: is_connected;
                        request_exists: request /= void;
                        is_ok: is_ok
                ensure
                        last_query_changed: last_query = request

feature -- Status report

        is_mapped (key: STRING): BOOLEAN
                        -- Is key mapped to an Eiffel entity?
                        -- (from STRING_HDL)
                require -- from STRING_HDL
                        keys_exists: key /= void

        last_query: STRING
```

```
                        -- Last SQL statement used
                        -- (from DB_EXPRESSION)

        mapped_value (key: STRING): ANY
                        -- Value mapped with key
                        -- (from STRING_HDL)
                require -- from STRING_HDL
                        key_exists: key /= void;
                        key_mapped: is_mapped (key)
                ensure -- from STRING_HDL
                        result_exists: Result /= void

        is_connected: BOOLEAN
                        -- Has connection to the data base server succeeded?
                        -- (from DB_STATUS_USE)

        is_ok: BOOLEAN
                        -- Is last SQL statement ok ?
                        -- (from DB_STATUS_USE)

feature -- Status setting

        clear_all
                        -- Remove all mapped keys.
                        -- (from STRING_HDL)

        set_map_name (n: ANY; key: STRING)
                        -- Store item n whith key key.
                        -- (from STRING_HDL)
                require -- from STRING_HDL
                        argument_exists: n /= void;
                        key_exists: key /= void;
                        not_key_in_table: not is_mapped (key)

        set_query (query: STRING)
                        -- Set last_query with query.
                        -- (from DB_EXPRESSION)
                require -- from DB_EXPRESSION
                        query_not_void: query /= void
                ensure -- from DB_EXPRESSION
                        last_query_changed: last_query = query

        unset_map_name (key: STRING)
                        -- Remove item associated with key key.
                        -- (from STRING_HDL)
                require -- from STRING_HDL
                        key_exists: key /= void;
                        item_exists: is_mapped (key)

end -- class DB_CHANGE
```

6.2 INTERFACE OF CLASS DB_CONTROL

indexing

status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
product: *eiffelstore*
database: *all_bases*

class interface

DB_CONTROL

creation

make

feature -- Basic operations

begin
-- Start a new transaction.
require
connection_exists: *is_connected*

commit
-- Commit work.
require
connection_exists: *is_connected*;
transaction_exists: *transaction_count* > *0*

connect
-- Connect to database.
require
not_already_connected: ***not*** *is_connected*
ensure
not *is_ok* ***or else*** *is_connected*

disconnect
-- Disconnect from database.
require
connection_exists: *is_connected*
ensure
no_connection: ***not*** *is_connected*;
all_transaction_ended: *transaction_count* = *0*

raise_error
-- Prompt error code and error message on standard output.

rollback
-- Rollback work.
require
connection_exists: *is_connected*;
transaction_exists: *transaction_count* > *0*

feature -- Status report

transaction_count: *INTEGER*

-- Number of started transactions
require
connection_exists: *is_connected*

error_code: *INTEGER*
-- Error code of last transaction
-- (from *DB_STATUS_USE*)

error_message: *STRING*
-- SQL error message prompted by database server
-- (from *DB_STATUS_USE*)

immediate_execution: *BOOLEAN*
-- Are requests immediately executed?
-- (default is *no*).
-- (from *DB_EXEC_USE*)

is_connected: *BOOLEAN*
-- Has connection to the data base server succeeded?
-- (from *DB_STATUS_USE*)

is_ok: *BOOLEAN*
-- Is last SQL statement ok ?
-- (from *DB_STATUS_USE*)

is_tracing: *BOOLEAN*
-- Is trace option for SQL queries on?
-- (from *DB_EXEC_USE*)

trace_output: *FILE*
-- Trace destination file
-- (from *DB_EXEC_USE*)

warning_message: *STRING*
-- SQL warning message prompted by database server
-- (from *DB_STATUS_USE*)

feature -- Status setting

reset
-- Reset *is_ok* and *error_code* after error occurred
-- (from *DB_STATUS_USE*)
ensure -- from *DB_STATUS_USE*
is_ok: *is_ok*;
no_error: *error_code = 0*

set_immediate
-- Set queries to be executed with a
-- *EXECUTE IMMEDIATE* SQL statement.
-- (from *DB_EXEC_USE*)
ensure -- from *DB_EXEC_USE*
execution_status: *immediate_execution*

set_trace
-- Trace queries sent to database server.

```
                    -- (from DB_EXEC_USE)
            ensure -- from DB_EXEC_USE
                    trace_status: is_tracing

        unset_immediate
                    -- Set queries to be executed with a
                    -- PREPARE followed by a EXECUTE SQL statement.
                    -- (from DB_EXEC_USE)
            ensure -- from DB_EXEC_USE
                    execution_status: not immediate_execution

        unset_trace
                    -- Do not trace queries sent to database server.
                    -- (from DB_EXEC_USE)
            ensure -- from DB_EXEC_USE
                    trace_status: not is_tracing

end -- class DB_CONTROL
```

6.3 INTERFACE OF CLASS DB_EXPRESSION

indexing
status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
product: "*EiffelStore*"
database: "*All bases*"

deferred class interface
DB_EXPRESSION

feature -- Basic operations

execute_query
-- Execute *last_query*.
require
last_query_not_void: *last_query* /= *void*

feature -- Status report

is_mapped (*key*: *STRING*): *BOOLEAN*
-- Is *key* mapped to an Eiffel entity?
-- (from *STRING_HDL*)
require -- from *STRING_HDL*
keys_exists: *key* /= *void*

last_query: *STRING*
-- Last SQL statement used

mapped_value (*key*: *STRING*): *ANY*
-- Value mapped with *key*
-- (from *STRING_HDL*)
require -- from *STRING_HDL*
key_exists: *key* /= *void*;
key_mapped: *is_mapped* (*key*)
ensure -- from *STRING_HDL*
result_exists: *Result* /= *void*

feature -- Status setting

clear_all
-- Remove all mapped keys.
-- (from *STRING_HDL*)

set_map_name (*n*: *ANY*; *key*: *STRING*)
-- Store item *n* whith key *key*.
-- (from *STRING_HDL*)
require -- from *STRING_HDL*
argument_exists: *n* /= *void*;
key_exists: *key* /= *void*;
not_key_in_table: **not** *is_mapped* (*key*)

set_query (*query*: *STRING*)

```
			-- Set last_query with query.
		require
			query_not_void: query /= void
		ensure
			last_query_changed: last_query = query

	unset_map_name (key: STRING)
			-- Remove item associated with key key.
			-- (from STRING_HDL)
		require -- from STRING_HDL
			key_exists: key /= void;
			item_exists: is_mapped (key)

end -- class DB_EXPRESSION
```

6.4 INTERFACE OF CLASS DB_PROC

indexing
    status: "*See notice at end of class*"
    date: "*$Date$*"
    revision: "*$Revision$*"
    product: "*EiffelStore*"
    database: "*All bases*"

class interface
    DB_PROC

creation
    make

feature -- Basic operations

    drop
            -- Drop current procedure from server.
        require
            exists: *exists*
        ensure
            not_loaded: ***not*** *loaded*

    execute (*destination*: *DB_EXPRESSION*)
            -- Execute current procedure with *destination*
            -- be a DB_SELECTION or DB_CHANGE object mapping
            -- entity values with procedure parameter names
            -- if any.
        require
            destination_not_void: *destination* /= *void*;
            exists: *exists*

    load
            -- Load stored procedure *name*
        require
            is_connected: *is_connected*
        ensure
            loaded: *loaded*

    store (*sql*: *STRING*)
            -- Store current procedure with *sql* expression.
        require
            sql_not_void: *sql* /= *void*;
            not_exists: ***not*** *exists*;
            args: ***not*** *arguments_set* ***or*** *arguments_set*

feature -- Initialization

    make (*a_name*: *STRING*)
            -- Create an interface object to create
            -- and execute stored procedure.
        require
            a_name_not_void: *a_name* /= *void*
        ensure

```
			name_equal: name•is_equal (a_name)

feature -- Status report

	arguments_name: ARRAY [STRING]
			-- Argument names

	arguments_set: BOOLEAN
			-- Have arguments been set?
		ensure
			Result = (arguments_name /= void and arguments_type /= void)

	arguments_type: ARRAY [ANY]
			-- Argument types

	exists: BOOLEAN
			-- Does current procedure exist in server?
		require
			loaded: loaded

	immediate_execution: BOOLEAN
			-- Are requests immediately executed?
			-- (default is no).
			-- (from DB_EXEC_USE)

	is_tracing: BOOLEAN
			-- Is trace option for SQL queries on?
			-- (from DB_EXEC_USE)

	loaded: BOOLEAN
			-- Is current procedure loaded?

	name: STRING
			-- Procedure name

	text: STRING
			-- SQL text of current procedure
		require
			exists: exists
		ensure
			result_not_void: Result /= void

	trace_output: FILE
			-- Trace destination file
			-- (from DB_EXEC_USE)

	is_connected: BOOLEAN
			-- Has connection to the data base server succeeded?
			-- (from DB_STATUS_USE)

	is_ok: BOOLEAN
			-- Is last SQL statement ok ?
			-- (from DB_STATUS_USE)

feature -- Status setting
```

change_name (*new_name*: *STRING*)
-- Change procedure name with *new_name*.
require
new_name_not_void: *new_name* /= *void*
ensure
new_name•is_equal (*name*);
not_loaded: ***not*** *loaded*

set_arguments (*args_name*: ***like*** *arguments_name*; *args_type*: ***like*** *arguments_type*)
require
args_name_not_void: *args_name* /= *void*;
args_type_not_void: *args_type* /= *void*;
same_count: *args_name•count* = *args_type•count*
ensure
arguments_name = *args_name*;
arguments_type = *args_type*;
arguments_set

set_no_arguments
-- No arguments for the current procedure
ensure
arguments_name = *void*;
arguments_type = *void*;
no_arguments: ***not*** *arguments_set*

invariant
load_and_exists: *loaded* **implies** (*exists* ***or not*** *exists*);

end -- class *DB_PROC*

6.5 INTERFACE OF CLASS DB_REPOSITORY

```
indexing
        status: "See notice at end of class"
        date: "$Date$"
        revision: "$Revision$"
        product: eiffelstore
        database: all_bases

class interface
        DB_REPOSITORY

creation
        make

feature -- Basic operations

        allocate (object: ANY)
                        -- Generate a database repository according to the
                        -- data representation of Eiffel object object.
                require
                        connected: is_connected;
                        obj_exists: object /= void;
                        is_ok: is_ok

        change_name (new_name: STRING)
                        -- Change repository name with new_name.
                require
                        is_ok: is_ok;
                        name_exists: new_name /= void
                ensure
                        new_name•is_equal (repository_name)

        generate_class
                        -- Generate an Eiffel class template mapping
                        -- the loaded data description.
                require
                        is_ok: is_ok;
                        rep_loaded: loaded

        load
                        -- Load persistent data description accessible through
                        -- repository_name.
                require
                        repository_name: repository_name /= void;
                        connected: is_connected
                ensure
                        loaded

feature -- Initialization

        make (name: STRING)
                        -- Create repository with name.
                require
                        name_exists: name /= void
```

```
        ensure
                repository_name•is_equal (name)

feature -- Status report

    conforms (object: ANY): BOOLEAN
                -- Do object attributes match the data description
                -- accessed through repository_name ?
        require
                parameter_not_void: object /= void;
                is_loaded: loaded;
                is_ok: is_ok

    exists: BOOLEAN
                -- Does repository repository_name exist?
        require
                is_ok: is_ok;
                rep_loaded: loaded

    loaded: BOOLEAN
                -- Is current repository data description
                -- retrieved from base?

    repository_name: STRING
                -- Name of repository

    is_connected: BOOLEAN
                -- Has connection to the data base server succeeded?
                -- (from DB_STATUS_USE)

    is_ok: BOOLEAN
                -- Is last SQL statement ok ?
                -- (from DB_STATUS_USE)

end -- class DB_REPOSITORY
```

6.6 INTERFACE OF CLASS DB_RESULT

indexing

status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
product: *eiffelstore*
database: *all_bases*

class interface

DB_RESULT

creation

make

feature -- Element change

copy (*other*: *DB_RESULT*)
-- Assign Current attributes with *other* attributes.
require
other /= *void*

fill_in
-- Fill in *data*.

feature -- Initialization

make
-- Create an interface object
-- to receive query result

feature -- Status report

data: *DB_DATA*
-- Loaded data

end -- class *DB_RESULT*

6.7 INTERFACE OF CLASS DB_SELECTION

```
indexing
        status: "See notice at end of class"
        date: "$Date$"
        revision: "$Revision$"
        access: perform_select, search, retrieve
        product: eiffelstore
        database: all_bases

class interface
        DB_SELECTION

creation
        make

feature -- Conversion

        cursor_to_object
                        -- Assign object attributes with cursor field values.
                require
                        cursor_exists: cursor /= void;
                        object_exists: object /= void

        no_object_convert
                        -- Do not transform cursor into an Eiffel object
                        -- while reading in selection results.
                ensure
                        object = void

        object: ANY
                        -- Eiffel object to be filled in by cursor_to_object

        object_convert (reference: ANY)
                        -- Set object with reference, reference to an Eiffel
                        -- object to be filled in with cursor field values.
                require
                        reference_exists: reference /= void
                ensure
                        object_set: object = reference

feature -- Name binding

        clear_all
                        -- Remove all mapped keys.
                        -- (from STRING_HDL)

        is_mapped (key: STRING): BOOLEAN
                        -- Is key mapped to an Eiffel entity?
                        -- (from STRING_HDL)
                require -- from STRING_HDL
                        keys_exists: key /= void

        mapped_value (key: STRING): ANY
                        -- Value mapped with key
```

```
                    -- (from STRING_HDL)
          require -- from STRING_HDL
                    key_exists: key /= void;
                    key_mapped: is_mapped (key)
          ensure -- from STRING_HDL
                    result_exists: Result /= void

     set_map_name (n: ANY; key: STRING)
                    -- Store item n whith key key.
                    -- (from STRING_HDL)
          require -- from STRING_HDL
                    argument_exists: n /= void;
                    key_exists: key /= void;
                    not_key_in_table: not is_mapped (key)

     unset_map_name (key: STRING)
                    -- Remove item associated with key key.
                    -- (from STRING_HDL)
          require -- from STRING_HDL
                    key_exists: key /= void;
                    item_exists: is_mapped (key)

feature -- SQL interface

     cursor: DB_RESULT
                    -- Cursor pointing to last fetched query result

     execute_query
                    -- Execute query with last_query.
          require -- from DB_EXPRESSION
                    last_query_not_void: last_query /= void

     exhausted: BOOLEAN
                    -- Is there any more resulting row?

     is_allocatable: BOOLEAN
                    -- Can Current be added to other concurrently opened selections?

     is_exiting: BOOLEAN
                    -- Is exit condition of load_result iteration loop met?
          ensure
                    Result implies
                              (not is_ok or else exhausted
                              or else (stop_condition /= void and then stop_condition•found))

     last_query: STRING
                    -- Last SQL statement used
                    -- (from DB_EXPRESSION)

     load_result
                    -- Iterate through selection results,
                    -- load container if requested and call action routine
                    -- for each iteration step until exit_condition is met.
          require
                    connected: is_connected;
```

is_ok: *is_ok*
ensure
cursor_not_void: *cursor* /= *void*;
exit_condition_met: *is_exiting*

next
-- Move to next element matching the query.
require
connected: *is_connected*

query (*s*: *STRING*)
-- Select stored objects using *s* and make
-- them retrievable using *load_result*.
require
connected: *is_connected*;
argument_exists: *s* /= *void*;
argument_is_not_empty: **not** *s*•*empty*;
is_ok: *is_ok*;
is_allocatable: *is_allocatable*
ensure
last_query_changed: *last_query* = *s*

reset_cursor (*c*: *DB_RESULT*)
-- Reset *cursor* with *c*.
require
arguments_exists: *c* /= *void*;
connected: *is_connected*
ensure
cursor_reset: *cursor* = *c*

set_action (*action*: *ACTION*)
-- Set *stop_condition* with *action*.
require
action_exists: *action* /= *void*
ensure
stop_condition_set: *stop_condition* = *action*

set_query (*query*: *STRING*)
-- Set *last_query* with *query*.
-- (from *DB_EXPRESSION*)
require -- from *DB_EXPRESSION*
query_not_void: *query* /= *void*
ensure -- from *DB_EXPRESSION*
last_query_changed: *last_query* = *query*

stop_condition: *ACTION*
-- Object providing an *execute* routine called
-- after each *load_result* iteration step

terminate
-- Clear database cursor.
require
connected: *is_connected*
ensure
is_allocatable: *is_allocatable*

```
wipe_out
            -- Clear selection results.
      ensure
            container_is_empty: container /= void implies container•empty;
            object_model_void: object = void;
            cursor_void: cursor = void

feature -- Storage

   container: LINKED_LIST [DB_RESULT]
            -- Stored cursors

   forth
            -- Move cursor to next element of container.
      require
            container_exists: container /= void
      ensure
            container_index_moved: container•index = old container•index + 1;
            cursor_updated: cursor = container•item

   set_container (one_container: like container)
            -- Make results of selection query persist in container.
      require
            container_exists: one_container /= void
      ensure
            container_set: container = one_container

   start
            -- Set cursor on first element of container.
      require
            container_exists: container /= void
      ensure
            container_on_first: container•isfirst;
            cursor_updated: cursor = container•item

   unset_container
            -- Do not store in container results of selection.
      require
            container_exists: container /= void
      ensure
            container_is_void: container = void

feature -- Transaction status

   immediate_execution: BOOLEAN
            -- Are requests immediately executed?
            -- (default is no).
            -- (from DB_EXEC_USE)

   is_connected: BOOLEAN
            -- Has connection to the data base server succeeded?
            -- (from DB_STATUS_USE)

   is_ok: BOOLEAN
```

-- Is last SQL statement ok ?
-- (from *DB_STATUS_USE*)

invariant

last_cursor_in_container: *container* **/=** *void* ***and then***
not *container*•*empty* ***implies*** *container*•*has* (*cursor*);

end -- class *DB_SELECTION*

6.8 INTERFACE OF CLASS DB_STORE

indexing

status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
access: *store*
product: *eiffelstore*
database: *all_bases*

class interface
DB_STORE

creation
make

feature -- Basic operations

force (*object*: *ANY*)
-- Insert *object* in repository attached to Current.
require
connected: *is_connected*;
object_exists: *object* /= *void*;
is_ok: *is_ok*;
owns_repository: *owns_repository*

put (*object*: *ANY*)
-- Insert *object* in repository attached to Current.
require
connected: *is_connected*;
object_exists: *object* /= *void*;
is_ok: *is_ok*;
owns_repository: *owns_repository*

feature -- Status report

immediate_execution: *BOOLEAN*
-- Are requests immediately executed?
-- (default is *no*).
-- (from *DB_EXEC_USE*)

is_tracing: *BOOLEAN*
-- Is trace option for SQL queries on?
-- (from *DB_EXEC_USE*)

owns_repository: *BOOLEAN*
-- Is Current linked to a repository?

trace_output: *FILE*
-- Trace destination file
-- (from *DB_EXEC_USE*)

is_connected: *BOOLEAN*
-- Has connection to the data base server succeeded?
-- (from *DB_STATUS_USE*)

```
is_ok: BOOLEAN
        -- Is last SQL statement ok ?
        -- (from DB_STATUS_USE)

feature -- Status setting

    set_repository (repository: DB_REPOSITORY)
            -- Set implementation repository with repository.
        require
            repository_not_void: repository /= void
        ensure
            owns_repository: owns_repository

end -- class DB_STORE
```

6.9 INTERFACE OF CLASS DB_TUPLE

indexing

status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
product: *eiffelstore*
database: *rdbms*

class interface

DB_TUPLE

creation

copy,
make

feature -- Element change

copy (*other*: *DB_RESULT*)
-- Assign Current attributes with *other* attributes.
-- (from *DB_RESULT*)
require -- from *DB_RESULT*
other /= *void*

fill_in
-- Fill in *data*.
-- (from *DB_RESULT*)

feature -- Initialization

make
-- Create an interface object
-- to receive query result
-- (from *DB_RESULT*)

feature -- Status report

column_name (*index*: *INTEGER*): *STRING*
-- Name of *index*-th item in Current tuple.

count: *INTEGER*
-- Number of columns in Current tuple

data: *DB_DATA_SQL*
-- Loaded data returned from last SQL query result

empty: *BOOLEAN*
-- Is Curren tuple empty?

item (*index*: *INTEGER*): *ANY*
-- Retrieved value at *index* position in *data*.

end -- class *DB_TUPLE*

6.10 INTERFACE OF CLASS HANDLE

indexing
status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
product: *eiffelstore*
database: *all_bases*

class interface
HANDLE

feature -- Status report

all_types: *DB_ALL_TYPES*
-- All data types available in active database
ensure
result_not_void: *Result* /= *void*

database: *DATABASE*
-- Active database accessed through the handle

execution_type: *DB_EXEC*
-- Immediate or non-immediate execution

process: *POINTER_REF*
-- Communication channel with database server
-- (single or multiple depending on RDBMS)

status: *DB_STATUS*
-- Status of active database

end -- class *HANDLE*

6.11 INTERFACE OF CLASS ORACLE_APPL

```
indexing
        status: "See notice at end of class"
        date: "$Date$"
        revision: "$Revision$"
        product: eiffelstore
        database: oracle

class interface
        ORACLE_APPL

creation
        login,
        login_and_connect

feature -- Initialization

        login (user_name, password: STRING)
                        -- Login to database server under user_name with password.
                require
                        arguments_exist: user_name /= void and password /= void

        login_and_connect (user_name, password: STRING)
                        -- Login under user_name with password
                        -- and immediately connect to Sybase database server,
                        -- using a temporary local DB_CONTROL object.

feature -- Status report

        is_logged_to_base: BOOLEAN
                        -- Is current handle logged to Oracle server?

feature -- Status setting

        set_base
                        -- Initialize or re-activate Oracle database server
                        -- after a handle change.
                require
                        is_logged_to_base: is_logged_to_base

end -- class ORACLE_APPL
```

6.12 INTERFACE OF CLASS SYBASE_APPL

indexing
    status: "*See notice at end of class*"
    date: "*$Date$*"
    revision: "*$Revision$*"
    product: *eiffelstore*
    database: *sybase*

class interface
    SYBASE_APPL

creation
    login,
    login_and_connect

feature -- Initialization

    login (*user_name*, *password*: *STRING*)
            -- Login to database server under *user_name* with *password*.
        require
            arguments_exist: *user_name* /= *void* ***and*** *password* /= *void*
        ensure
            is_logged_to_base: *is_logged_to_base*

    login_and_connect (*user_name*, *password*: *STRING*)
            -- Login under *user_name* with *password*
            -- and immediately connect to Sybase database server,
            -- using a temporary local DB_CONTROL object.
        require
            arguments_not_void: *user_name* /= *void* ***and*** *password* /= *void*

feature -- Status report

    is_logged_to_base: *BOOLEAN*
            -- Is current handle logged to Sybase server?

feature -- Status setting

    set_application (*application_name*: *STRING*)
            -- Set database application name with *application_name*.
        require
            argument_exist: *application_name* /= *void*
        ensure
            is_logged_to_base: *is_logged_to_base*

    set_base
            -- Initialize or re-activate Sybase database server
            -- after a handle change.
        require
            is_logged_to_base: *is_logged_to_base*

    set_hostname (*host_name*: *STRING*)
            -- Set database host name with *host_name*.
        require

argument_exist: *host_name* /= *void*

ensure

is_logged_to_base: *is_logged_to_base*

end -- class *SYBASE_APPL*

6.13 INTERFACE OF CLASS ABSOLUTE_DATE

indexing
status: "*See notice at end of class*"
date: "*$Date$*"
revision: "*$Revision$*"
access: *date*, *time*
product: *time_and_date*
system: *unix*

class interface
ABSOLUTE_DATE

creation
make

feature -- Arithmetics

infix *"+"* (*d*: *DATE*): *DATE*
-- Sum of current date and *d*

infix *"-"* (*d*: *DATE*): *DATE*
-- Difference between current date and *d*
require
large_enough: *d <= Current*
require else -- from *DATE*
large_enough: *Current >= d*
require else -- from *TIME*
Current >= t
require else -- from *ABSOLUTE_TIME*
large_enough: *t <= Current*

feature -- Change of state

change_date (*m*, *d*, *y*: *INTEGER*)
-- Set *year*, *month* and *day* with, *y*, *m*, *d* respectively;
-- time remains unchanged.
require -- from *DATE*
positive_year: *y >= 0*;
proper_month: *m > 0* ***and*** *m <= 12*;
proper_day: *d > 0* ***and*** *d <= 31*
ensure -- from *DATE*
year = y;
month = m;
day = d;
hour = ***old*** *hour*;
minute = ***old*** *minute*;
second = ***old*** *second*

change_time (*h*, *m*, *s*: *INTEGER*)
-- Set *hour, ‘minute* and *second* to *h*, *m*, *s* repectively.
-- (from *ABSOLUTE_TIME*)
require -- from *TIME*
proper_hour: *h >= 0* ***and*** *h < 24*;
proper_minute: *m >= 0* ***and*** *m < 60*;

proper_second: *s >= 0* ***and*** *s < 60*
ensure -- from *TIME*
hour = h;
minute = m;
second = s

from_string (*date_string*: *STRING*)
-- Set date according to *date_string* format
-- given as follows: "mm/dd/yy" or "mm/dd/yyyy"
-- with *mm* in range [1 .. 12].

load (*one_date*: *DATE*)
-- Load current with *one_date* date and time attributes.
-- (from *DATE*)
require -- from *DATE*
argument_non_void: *one_date /= void*
ensure -- from *DATE*
year = one_date•year;
month = one_date•month;
day = one_date•day;
hour = one_date•hour;
minute = one_date•minute;
second = one_date•second

seconds_to_time (*s*: *INTEGER*)
-- Set current time from *s* input seconds.
-- (from *TIME*)
require -- from *TIME*
s >= 0

set_local_date
-- Reset current date with local system date and time.

time_from_string (*time_string*: *STRING*)
-- Set time according *time_string*
-- that must be the format "hh:mm:ss"
-- (*hh* go from 0 to 23).
-- (from *ABSOLUTE_TIME*)

feature -- Change of state or reference

set (*tz*: *STRING*)
-- Set to local system time converted
-- to the specified timezone *tz*.
-- (from *ABSOLUTE_TIME*)
require -- from *ABSOLUTE_TIME*
time_zone_name_non_void: *tz /= void*

set_gmt
-- Set to local system time converted
-- to Greenwich Mean Time (GMT).
-- (from *ABSOLUTE_TIME*)

set_local_time
-- Set to local system time.

```
                        -- (from ABSOLUTE_TIME)

feature -- Comparison

    is_equal (other: like Current): BOOLEAN
                        -- Is current date and time equal to other?
                        -- (from DATE)
            ensure -- from DATE
                        (year = other•year) and (month = other•month) and (day = other•day)
                        and Current•time_is_equal (other)
            ensure then -- from TIME
                        (hour = other•hour) and (minute = other•minute) and (second = other•second)
            ensure then -- from COMPARABLE
                        trichotomy: Result = (not (Current < other) and not (other < Current))

    max (other: like Current): like Current
                        -- The greater of current object and other
                        -- (from COMPARABLE)
            require -- from COMPARABLE
                        other_exists: other /= void
            ensure -- from COMPARABLE
                        current_if_not_smaller: Current >= other implies Result = Current;
                        other_if_smaller: Current < other implies Result = other

    min (other: like Current): like Current
                        -- The smaller of current object and other
                        -- (from COMPARABLE)
            require -- from COMPARABLE
                        other_exists: other /= void
            ensure -- from COMPARABLE
                        current_if_not_greater: Current <= other implies Result = Current;
                        other_if_greater: Current > other implies Result = other

    three_way_comparison (other: like Current): INTEGER
                        -- If current object equal to other, 0;
                        -- if smaller, -1; if greater, 1
                        -- (from COMPARABLE)
            require -- from COMPARABLE
                        other_exists: other /= void
            ensure -- from COMPARABLE
                        equal_zero: (Result = 0) = is_equal (other);
                        smaller_negative: (Result = – 1) = (Current < other);
                        greater_positive: (Result = 1) = (Current > other)

    time_is_equal (other: like Current): BOOLEAN
                        -- Is Current time equal to other?
                        -- (from TIME)
            ensure -- from TIME
                        (hour = other•hour) and (minute = other•minute) and (second = other•second)
            ensure then -- from COMPARABLE
                        trichotomy: Result = (not (Current < other) and not (other < Current))

    infix ">=" (other: like Current): BOOLEAN
                        -- Is current object greater than or equal to other?
                        -- (from COMPARABLE)
```

```
            require -- from PART_COMPARABLE
                    other_exists: other /= void
            ensure -- from COMPARABLE
                    definition: Result = (other <= Current)

        infix ">" (other: like Current): BOOLEAN
                    -- Is current object greater than other?
                    -- (from COMPARABLE)
            require -- from PART_COMPARABLE
                    other_exists: other /= void
            ensure -- from COMPARABLE
                    definition: Result = (other < Current)

        infix "<=" (other: like Current): BOOLEAN
                    -- Is current object less than or equal to other?
                    -- (from COMPARABLE)
            require -- from PART_COMPARABLE
                    other_exists: other /= void
            ensure -- from COMPARABLE
                    definition: Result = (Current < other) or is_equal (other)

        infix "<" (other: like Current): BOOLEAN
                    -- Is current date less than other?
                    -- (from DATE)
            require -- from PART_COMPARABLE
                    other_exists: other /= void
            ensure -- from DATE
                    (year < other•year) or ((year = other•year)
                    and (month < other•month)) or ((year = other•year)
                    and (month = other•month) and (day < other•day))
                    or ((day = other•day) and Current time_less_than )
            ensure then -- from TIME
                    (hour < other•hour) or else ((hour = other•hour)
                    and ((minute < other•minute) or else ((minute = other•minute)
                    and (second < other•second))))
            ensure then -- from COMPARABLE
                    asymmetric: Result implies not (other < Current)

        "time_less_than" (other: like Current): BOOLEAN
                    -- Is current time less than other?
                    -- (from TIME)
            require -- from PART_COMPARABLE
                    other_exists: other /= void
            ensure -- from TIME
                    (hour < other•hour) or else ((hour = other•hour)
                    and ((minute < other•minute) or else ((minute = other•minute)
                    and (second < other•second))))
            ensure then -- from COMPARABLE
                    asymmetric: Result implies not (other < Current)

feature -- Creation

        make
                    -- Create a calendar date initialized
                    -- with today's date and time.
```

feature -- External representation

default_format: *STRING*
-- Default output format used for printing date and time
-- following UNIX-like conventions.
-- (from *DATE*)
ensure -- from *DATE*
Result•substring (*Result•count* – *4*, *Result•count* – *3*)•*is_equal* ("*%%D*")
ensure then -- from *TIME*
Result•substring (*Result•count* – *1*, *Result•count*)•*is_equal* ("*%%T*")

out: *STRING*
-- Time or date formatted according to output format
-- (from *TIME*)

output_format: *STRING*
-- Output format used by routine *out*
-- (from *TIME*)

set_locale (*lang*: *STRING*)
-- Change time and date conversion functions
-- so as to comply with the *.../lib/locale/LC_TIME/lang*
-- file definitions.
-- If *lang* = *Void* the value of the environment variable *LC_TIME*
-- is selected, if valid. If the value is invalid, *set_locale*
-- has no effect.
-- (from *TIME*)

set_output_format (*format_string*: *STRING*)
-- Change *default_format* to *format_string*.
-- (from *TIME*)
require -- from *TIME*
format_string /= *void*
ensure -- from *TIME*
output_format•is_equal (*format_string*)

feature -- Other properties

current_month: *INTEGER*
-- Elapsed months since January
-- (excluding current month)
ensure
Result >= *0* ***and*** *Result* <= *11*

current_year: *INTEGER*
-- Years since 1900 -- [0-99].
ensure
Result >= *0* ***and*** *Result* <= *99*

hours: *INTEGER*
-- Hours since midnight
-- (from *ABSOLUTE_TIME*)
ensure -- from *ABSOLUTE_TIME*
Result >= *0* ***and*** *Result* <= *23*

```
is_daylight: BOOLEAN
        -- Is it daylight savings time?
        -- (from ABSOLUTE_TIME)

leap_year: BOOLEAN
        -- Is current year a leap year?

minutes: INTEGER
        -- Minutes after the hour
        -- (from ABSOLUTE_TIME)
    ensure -- from ABSOLUTE_TIME
        Result >= 0 and Result <= 59

month_day: INTEGER
        -- Day of the month
    ensure
        Result >= 1 and Result <= 31

month_days: INTEGER
        -- Number of days in current month
    ensure
        Result >= 28 and Result <= 31

month_seconds: INTEGER
        -- Number of seconds in month
        -- (from DATE_CONSTANTS)

seconds: INTEGER
        -- Seconds after the minute
        -- (from ABSOLUTE_TIME)
    ensure -- from ABSOLUTE_TIME
        Result >= 0 and Result <= 59

time_to_seconds: INTEGER
        -- Time converted into seconds
        -- (from TIME)

weekday: INTEGER
        -- Elapsed days since last Sunday
        -- (excluding current day)
    ensure
        Result >= 0 and Result <= 6

year_day: INTEGER
        -- Elapsed days since January 1
        -- (excluding current day)
    ensure
        Result >= 1 and Result <= 365

year_days: INTEGER
        -- Number of days in current year
    ensure
        Result = 365 or Result = 366
```

```
year_seconds: INTEGER
        -- Number of seconds in year
        -- (from DATE_CONSTANTS)

feature -- State

    day: INTEGER
        -- (from DATE)

    hour: INTEGER
        -- (from TIME)

    minute: INTEGER
        -- (from TIME)

    month: INTEGER
        -- (from DATE)

    second: INTEGER
        -- (from TIME)

    year: INTEGER
        -- (from DATE)

invariant
    day >= 1 and day <= month_days;
    -- from COMPARABLE
    irreflexive_comparison: not (Current < Current);

end -- class ABSOLUTE_DATE
```

References

Index

Contents

1 Foreword 1

2 Introduction 2

3 Class layers 4
- 3.1 Interfacing with EiffelStore 4
- 3.2 Interface layer 4
- 3.3 Implementation layer 5

4 RDBMS coupling 6
- 4.1 Interface with the DB server 6
- 4.2 Repositories 8
- 4.3 Selection queries 9
- 4.4 Data manipulation 11
- 4.5 Modification queries 13
- 4.6 Table records and Eiffel objects 14
- 4.7 Storing an Eiffel object inside a table 17
- 4.8 Stored procedures 17
- 4.9 Error handling 19

5 DBMS specifics 21
- 5.1 Oracle specifics 21
- 5.2 Sybase specifics 23

6 Class interfaces 25
- 6.1 Interface of class DB_CHANGE 25
- 6.2 Interface of class DB_CONTROL 27
- 6.3 Interface of class DB_EXPRESSION 30
- 6.4 Interface of class DB_PROC 32
- 6.5 Interface of class DB_REPOSITORY 35
- 6.6 Interface of class DB_RESULT 37
- 6.7 Interface of class DB_SELECTION 38
- 6.8 Interface of class DB_STORE 43
- 6.9 Interface of class DB_TUPLE 45

6.10 Interface of class HANDLE 46
6.11 Interface of class ORACLE_APPL 47
6.12 Interface of class SYBASE_APPL 48
6.13 Interface of class ABSOLUTE_DATE 50

References 57

Index 59