The ABEL Persistence Library Tutorial

Written by: Roman Schmocker
Reviewed by: Marco Piccioni

Last updated:

July 26, 2012

Contents

Chapter 1

Introducing ABEL

ABEL (A Better EiffelStore Library) is an object-oriented persistence library written in Eiffel and aiming at seamlessly integrating various kinds of data stores.

1.1 Setting things up

We are assuming you have checked out the ABEL code from the EiffelStudio SVN repository[1], and have EiffelStudio installed. Launch it and in the initial window choose "persistence_tutorial_project". If it is not there just push "Add project" and navigate to the location where you downloaded ABEL, and look for the *persistence_tutorial_project.ecf* project file in *abel/apps/sample/tutorial/*. You can then load the project and you are ready to go.

For convenience, in this tutorial we will use a simple in-memory repository simulating a relational database but storing all values in memory. To set up ABEL using a full-fledged relational back-end (like MySQL or SQLite) please read Chapter 5.

1.2 Getting started

We will be using `PERSON` objects to show the usage of the API. In the source code below you will see that ABEL handles objects "as they are", meaning that to make them persistent you don't need to add any dependencies to their class source code.

```
class PERSON
```

[1]https://svn.eiffel.com/eiffelstudio/branches/eth/eve/Src/library/abel

```
create
  make

feature {NONE} -- Initialization

  make (first, last: STRING)
      -- Create a newborn person.
    require
      first_exists: not first.is_empty
      last_exists: not last.is_empty
    do
      first_name := first
      last_name := last
      age:= 0
    ensure
      first_name_set: first_name = first
      last_name_set: last_name = last
      default_age: age = 0
    end

feature -- Basic operations

  celebrate_birthday
      -- Increase age by 1.
    do
      age:= age + 1
    ensure
      age_incremented_by_one: age = old age + 1
    end

feature -- Access

  first_name: STRING
    -- The person's first name.

  last_name: STRING
    -- The person's last name.

  age: INTEGER
    -- The person's age.

invariant
```

```
  age_non_negative: age >= 0
  first_name_exists: not first_name.is_empty
  last_name_exists: not last_name.is_empty
end
```

Listing 1.1: *The PERSON class*

There are three very important classes in ABEL:

- The deferred class `PS_REPOSITORY` provides an abstraction to the actual storage mechanism.
- The `PS_CRUD_EXECUTOR` class is, as the name suggests, responsible to execute CRUD (Create Read Update Delete) commands. Every `PS_CRUD_EXECUTOR` object works with a `PS_REPOSITORY`.
- The `PS_OBJECT_QUERY [G]` class is used to describe a read operation over objects of type `G`. You can execute such a query in the `PS_CRUD_EXECUTOR`. The result will be objects of type `G`.

To start using the library, we first need to create an object of type `PS_REPOSITORY`. In this case we will be creating a more specific object of type `PS_IN_MEMORY_REPOSITORY`.
As a second step, we need to create an object of type `PS_CRUD_EXECUTOR`, useful to execute CRUD operations. To create it, we will pass as an argument to its creation feature the previously created repository.

```
class TUTORIAL

create
  make

feature {NONE} -- Initialization

  make
    -- Set up a simple in-memory repository.
    local
      repository: PS_IN_MEMORY_REPOSITORY
    do
      create repository.make_empty
      create executor.make (repository)
    end

feature

```

```
  executor: PS_CRUD_EXECUTOR
    -- The CRUD executor used throughout the tutorial.

end
```

Listing 1.2: *The TUTORIAL class*

We will use this class throughout the tutorial. You can assume that the Eiffel features listed in this tutorial are located inside the *TUTORIAL* class, if they are not enclosed in another class declaration.
We encourage you to test the features shown in this tutorial by calling them from feature *explore* in class *TUTORIAL*.

Chapter 2

Basic operations

2.1 Inserting

You insert an object in the repository using feature `insert` in class *PS_CRUD_EXECUTOR*:

```
simple_insert (a_person: PERSON)
  -- Insert 'a_person' into the current repository.
  do
    executor.insert (a_person)
  end
```

Listing 2.1: *Insertion query.*

2.2 Querying

A query for objects is done by creating a `PS_OBJECT_QUERY [G]` object and executing it using features of *PS_CRUD_EXECUTOR*. The generic parameter *G* denotes the type of objects that should be queried.

After a successful execution of the query, you can find the result in the iteration cursor `result_cursor` in class *PS_OBJECT_QUERY*. The feature `simple_query` below shows how to get a list of persons from the repository:

```
simple_query: LINKED_LIST [PERSON]
  -- Query all persons from the current repository.
  local
    query: PS_OBJECT_QUERY [PERSON]
  do
    create Result.make
```

```
    create query.make
    executor.execute_query (query)

    across query as query_result
    loop
      Result.extend (query_result.item)
    end
  end
```

Listing 2.2: *A simple query.*

Usually the result of such a query is very big, and you are probably only interested in objects that meet certain criteria, e.g. all persons of age 20. You can read more about it in Chapter 3.

Please note that ABEL does not enforce any kind of order on a query result.

2.3 Updating

Updating an object is done through feature *update* in *PS_CRUD_EXECUTOR*:

```
simple_update (a_person: PERSON)
  -- Update 'a_person' into the current repository.
  do
    a_person.celebrate_birthday
    executor.update (a_person)
  end
```

Listing 2.3: *Update query.*

The object to update needs to be previously known to ABEL through an insert or a successful query (see Section 2.5).

2.4 Deleting

Deletion is done through the feature *delete* in *PS_CRUD_EXECUTOR*, like in the following example:

```
delete_person (name: STRING)
  -- Delete the person called 'name'.
  local
    query: PS_OBJECT_QUERY [PERSON]
  do
    -- First retrieve the person from the database.
```

```
    create query.make
    executor.execute_query (query)
    across query as query_result
    loop
      if query_result.item.last_name.is_equal (name) then

        -- Now delete the person.
        executor.delete (query_result.item)
      end
    end
  end
```

Listing 2.4: *Deletion query.*

Another way to delete objects is described in Section 3.3.

2.5 Dealing with Known Objects

ABEL keeps track of objects that have been inserted or queried. This is important because in case of an update or delete, the library internally needs to map the object in the current execution of the program to its specific entry in the database.

Because of that, you can't update or delete an object that is not yet known to ABEL. As an example, the following two functions will fail:

```
failing_update
  -- Try and fail to update a new person object
  local
    a_person: PERSON
  do
    create a_person.make ("Albo", "Bitossi")
    executor.update (a_person)
      -- Results in a precondition violation
  end

failing_delete (name:STRING)
  -- Try and fail to delete a new person object
  local
    a_person:PERSON
  do
    create a_person.make ("Albo", "Bitossi")
    executor.delete (a_person)
      -- Results in a precondition violation
```

```
end
```

Listing 2.5: Failing updates and deletes.

Please note that there's another way to delete objects, described in Section 3.3, which doesn't have this restriction.

The feature *is_persistent* in *PS_CRUD_EXECUTOR* can tell you if a specific object is known to ABEL and hence has a link to its entry in the database.

Chapter 3

Advanced Queries

3.1 The query mechanism

As you already know from Section 2.2, queries to a database are done by creating an object of type `PS_OBJECT_QUERY[G]` and using it from within a `PS_CRUD_EXECUTOR`. The actual value of the generic parameter `G` determines the type of the objects that will be returned, including any conforming type (e.g. descendants of `G`).

ABEL will by default load an object completely, meaning all objects that can be reached by following references will be loaded as well (see also Chapter 4).

3.2 Criteria

You can filter your query results by setting criteria in the query object, using feature `set_criteria` in `PS_OBJECT_QUERY`. There are two types of criteria: predefined and agent criteria.

3.2.1 Predefined Criteria

When using a predefined criterion you pick an attribute name, an operator and a value. During a read operation, ABEL checks the attribute value of the freshly retrieved object against the value set in the criterion, and filters away objects that don't satisfy the criterion.

Most of the supported operators are pretty self-describing (see class `CRITERION_FACTORY` in Section 3.2.3). An exception could be the **`like`** operator, which does pattern-matching on strings. You can provide the **`like`** operator with a pattern as a value. The pattern can contain the wildcard characters `*` and `?`. The asterisk stands for any number (including zero) of undefined characters, and the question mark means exactly one undefined character.

You can only use attributes that are strings or numbers, but not every type of attribute supports every other operator. Valid combinations for each type are:

- Strings: =, like
- Any numeric value: =, <, <=, >, >=
- Booleans: =

Note that for performance reasons it is usually better to use predefined criteria, because they can be compiled to SQL and hence the result can be filtered in the database.

3.2.2 Agent Criteria

An agent criterion will filter the objects according to the result of an agent applied to them.

The criterion is initialized with an agent of type *PREDICATE* [*ANY*, *TUPLE* [*ANY*]]. There should be either an open target or a single open argument, and the type of the objects in the query result should conform to the agent's open operand. For an example see Section 3.2.3.

3.2.3 Creating criteria objects

The criteria instances are best created using the *CRITERION_FACTORY* class.

The main features of the class are the following:

```
class
  PS_CRITERION_FACTORY
create
  default_create

feature -- Creating a criterion

  new alias "[]" (tuple: TUPLE [ANY]): PS_CRITERION
    -- Creates a new criterion according to a ‘tuple’
    -- containing either a single PREDICATE or three
    -- values of type [STRING, STRING, ANY].

```

```
  new_agent (a_predicate: PREDICATE [ANY, TUPLE [ANY]]):
      PS_CRITERION
    -- Creates an agent criterion.

  new_predefined (object_attribute: STRING;
    operator: STRING; value: ANY): PS_CRITERION
    -- Creates a predefined criterion.

feature -- Operators

  equals: STRING = "="

  greater: STRING = ">"

  greater_equal: STRING = ">="

  less: STRING = "<"

  less_equal: STRING = "<="

  like_string: STRING = "like"

end
```

Listing 3.1: *The CRITERION_FACTORY class interface*

Assuming you have an object `f: PS_CRITERION_FACTORY`, to create a new criterion you have two possibilities:

- The ”traditional” way
  - `f.new_agent (agent an_agent)`
  - `f.new_predefined (an_attr_name, an_operator, a_val)`
- The ”syntactic sugared” way
  - `f[[an_attr_name, an_operator, a_value]]`
  - `f[[agent an_agent]]`

caption=The CRITERION_FACTORY interface

```
  create_criteria_traditional : PS_CRITERION
    -- Create a new criteria using the traditional approach.
```

```
  do
    -- for predefined criteria
    Result:=
      factory.new_predefined ("age", factory.less, 5)

    -- for agent criteria
    Result :=
      factory.new_agent (agent age_less_than (?, 5))
  end

create_criteria_double_bracket : PS_CRITERION
  -- Create a new criteria using the double bracket syntax.
  do
    -- for predefined criteria
    Result:= factory[["age", factory.less, 5]]

    -- for agent criteria
    Result := factory[[agent age_less_than (?, 5)]]
  end

age_less_than (person: PERSON; age: INTEGER): BOOLEAN
  -- An example agent
  do
    Result:= person.age < age
  end
```

Listing 3.2: *Different ways of creating criteria.*

3.2.4 Combining criteria

You can combine multiple criterion objects by using the standard Eiffel logical operators. For example, if you want to search for a person called "Albo Bitossi" with $age \neq 20$, you can just create a criterion object for each of the constraints and combine them:

```

search_albo_bitossi : PS_CRITERION
  -- Combining criterion objects.
  local
    first_name_criterion:PS_CRITERION
    last_name_criterion: PS_CRITERION
    age_criterion: PS_CRITERION
  do
```

```
    first_name_criterion:=
      factory[[ "first_name", factory.equals, "Albo" ]]

    last_name_criterion :=
      factory[[ "last_name", factory.equals, "Bitossi" ]]

    age_criterion :=
      factory[[ "age", factory.equals, 20 ]]

    Result := first_name_criterion and last_name_criterion
         and not age_criterion

    -- using double brackets for compactness.
    Result := factory[[ "first_name", "=", "Albo" ]]
      and factory[[ "last_name", "=", "Bitossi" ]]
      and not factory[[ "age", "=", 20 ]]
  end
```

Listing 3.3: *Combining criteria.*

ABEL supports the three standard logical operators **AND**, **OR** and **NOT**. The precedence rules are the same as in Eiffel, which means that **NOT** is stronger than **AND**, which in turn is stronger than **OR**.

3.3 Deletion queries

As mentioned previously, there is another way to perform a deletion in the repository from within *PS_CRUD_EXECUTOR*. By calling *execute_deletion_query* instead of *delete*, ABEL will delete all objects in the database that would have been retrieved by executing the query normally. You can look at the following example and compare it with its variation in Section 2.4.

```
delete_person (name:STRING)
  -- Delete 'name' using a deletion query.
  local
    deletion_query: PS_OBJECT_QUERY [PERSON]
    criterion:PS_PREDEFINED_CRITERION
  do
    create deletion_query.make
    create criterion.make ("last_name", "=", name)
    deletion_query.set_criterion (criterion)
    executor.execute_deletion_query (deletion_query)
  end
```

Listing 3.4: *Using a deletion query.*

It depends upon the situation if you want to use deletion queries or a direct delete command. Usually, a direct command is better if you already have the object in memory, whereas deletion queries are nice to use if the object is not yet loaded from the database.

3.4 Tuple queries

Consider a scenario in which you just want to have a list of all last names of persons in the database. Loading every object of type *PERSON* might lead to a very bad performance, especially if there is a big object graph attached to each person object.

To solve this problem, *PS_CRUD_EXECUTOR* allows to query data while returning tuples as a result. You can do this by calling feature

```
execute_tuple_query (a_tuple_query),
```

where *a_tuple_query* is of type *PS_TUPLE_QUERY* [*G*]. The result is an iteration cursor over a list of tuples in which the attributes of an object are stored. The order of these attributes is the one defined in feature *projection* in *PS_TUPLE_QUERY* [*G*].

```
print_all_last_names
  -- Print the last name of all PERSON objects.
  local
    query: PS_TUPLE_QUERY [PERSON]
    last_name_index: INTEGER
    single_result: TUPLE
  do
    create query.make
    ---- Find out at which position in the tuple the
       last_name is.
    last_name_index := query.attribute_index ("last_name")
    from
      executor.execute_tuple_query (query)
    until
      query.result_cursor.after
    loop
      single_result:= query.result_cursor.item
      print (single_result [last_name_index] )
    end
  end
```

Listing 3.5: *Using tuple queries.*

3.4.1 Tuple queries and projections

By default, a *TUPLE_QUERY* object will only return values of attributes which are of a basic type, so no references are followed during a retrieve. You can change this default by calling *set_projection*, which expects an array of names of the attributes you would like to have. If you include an attribute name whose type is not a basic one, ABEL will actually retrieve and build the attribute object, and not just another tuple.

3.4.2 Tuple queries and criteria

You are restricted to use predefined criteria in tuple queries, because agent criteria expect an object and not a tuple. You can still combine them with logical operators, and even include a predefined criterion on an attribute that is not present in the projection list. These attributes will be loaded internally to check if the object satisfies the criterion, and then they are discarded for the actual result.

```
print_last_names_of_20_year_old
  -- Print the last name of all PERSON objects with age =
    20.
  local
    query: PS_TUPLE_QUERY [PERSON]
  do
    create query.make
    -- Only return the last_name of persons
    query.set_projection (<<"last_name">>)
    -- Only return persons with age = 20
    query.set_criterion (factory [["age", "=", 20]])
    from
      executor.execute_tuple_query (query)
    until
      query.result_cursor.after
    loop
      -- As we only have the last_name in the tuple,
      -- its index has to be 1.
      print (query.result_cursor.item [1] )
    end
  end
```

Listing 3.6: *Using tuple queries with criteria.*

Chapter 4

Dealing with references

In ABEL, a basic type is an object of type *STRING*, *BOOLEAN*, *CHARACTER* or any numeric class like *REAL* or *INTEGER*. The *PERSON* class only has attributes of a basic type. However, an object can contain references to other objects. ABEL is able to handle these references by storing and reconstructing the whole object graph (an object graph is roughly defined as all the objects that can be reached by recursively following all references, starting at some root object).

Let's look at the new class *CHILD*:

```
class
 CHILD

create
 make

feature {NONE} -- Initialization

 make (first, last: STRING)
   -- Create a new child.
  require
   first_exists: not first.is_empty
   last_exists: not last.is_empty
  do
   first_name := first
   last_name := last
   age := 0
  ensure
   first_name_set: first_name = first
   last_name_set: last_name = last
```

```
      default_age: age = 0
    end

  feature -- Access

    celebrate_birthday
      -- Increase age by 1.
    do
      age := age + 1
    ensure
      age_incremented_by_one: age = old age + 1
    end

  feature -- Status report

    first_name: STRING
      -- The child's first name.

    last_name: STRING
      -- The child's last name.

    age: INTEGER
      -- The child's age.

  feature -- Parents

    mother: detachable CHILD
      -- The child's mother.

    father: detachable CHILD
      -- The child's father.

    set_mother (a_mother: CHILD)
      -- Set a mother for the child.
    do
      mother := a_mother
    ensure
      mother_set: mother = a_mother
    end

    set_father (a_father: CHILD)
      -- Set a father for the child.
    do
```

```
      father := a_father
    ensure
      father_set: father = a_father
    end

invariant
  age_non_negative: age >= 0
  first_name_exists: not first_name.is_empty
  last_name_exists: not last_name.is_empty
end
```

Listing 4.1: *The CHILD class.*

This adds in some complexity: instead of having a single object, ABEL has to insert a `CHILD`'s mother and father as well, and it has to repeat this procedure if their parent attribute is also attached. The good news are that the examples above will work exactly the same.

However, there are some additional caveats to take into consideration. Let's consider a simple example with `CHILD` objects "Baby Doe", "John Doe" and "Grandpa Doe". From the name of the object instances you can already guess what the object graph looks like:

Now if you insert "Baby Doe", ABEL will by default follow all references and insert every single object along the object graph, which means that "John Doe" and "Grandpa Doe" will be inserted as well. This is usually the desired behavior, as objects are stored completely that way, but it also has some side effects we need to be aware of:

- Assume an insert of "Baby Doe" has happened to an empty database. If you now query the database for `CHILD` objects, it will return exactly the same object graph as above, but the query result will actually have three items, as the object graph consists of three single `CHILD` objects.

- After you've inserted "Baby Doe", it has no effect if you insert "John Doe" or "Grandpa Doe" afterwards, because they have already been inserted by the first statement.

4.1 Updates

ABEL does not follow references during an update by default, so for example the following statement has no effect on the database:

```
celebrate_fathers_birthday (a_child: CHILD)
  -- Increase age of 'a_child's father.
  require
    child_persistent: executor.is_persistent (a_child)
  do
    a_child.father.celebrate_birthday

    -- This won't have any effect
    executor.update (a_child)

    -- however, it works that way
    executor.update (a_child.father)
  end
```

Listing 4.2: *References are not followed by default during updates.*

Section 4.2 will tell you how do change the default settings.

4.2 Going deeper in the Object Graph

ABEL has no limits regarding the depth of an object graph, and it will detect and handle reference cycles correctly. You are welcome to test ABEL's capability with very complex objects, however please keep in mind that this may impact performance significantly.

To overcome this problem, you can either use simple object structures, or you can tell ABEL to only load or store an object up to a certain depth. The default ABEL's behavior with respect to the object graph can be changed by using feature `default_object_graph` in class `PS_REPOSITORY` and passing an appropriate object of type `PS_DEFAULT_OBJECT_GRAPH_SETTINGS`.

Chapter 5

Advanced Initialization

The in-memory repository we've used so far doesn't store data permanently. This is acceptable for testing or for a tutorial, but not in a real application. Therefore, ABEL ships with support for a MySQL database and an SQLite database.

To use them, you have to assemble the needed parts. Let's focus on MySQL: you will need to create a *PS_MYSQL_DATABASE* and a *PS_MYSQL_STRINGS* object. Then you will use them to create a *PS_GENERIC_LAYOUT_SQL_BACKEND* , which you will need in turn to create the *PS_RELATIONAL_REPOSITORY*.

The following little factory class shows the process for both a MySQL and an SQLite database:

```
class
  REPOSITORY_FACTORY

feature -- Connection details

  username:STRING = "tutorial"
  password:STRING = "tutorial"

  db_name:STRING = "tutorial"
  db_host:STRING = "127.0.0.1"
  db_port:INTEGER = 3306

  sqlite_filename: STRING = "tutorial.db"

feature -- Factory methods

  create_mysql_repository: PS_RELATIONAL_REPOSITORY
    -- Create a MySQL repository
```

```
    local
      database: PS_MYSQL_DATABASE
      mysql_strings: PS_MYSQL_STRINGS
      backend: PS_GENERIC_LAYOUT_SQL_BACKEND
    do
      create database.make (username, password, db_name,
          db_host, db_port)
      create mysql_strings
      create backend.make (database, mysql_strings)
      create Result.make (backend)
    end

  create_sqlite_repository: PS_RELATIONAL_REPOSITORY
    -- Create an SQLite repository
    local
      database: PS_SQLITE_DATABASE
      sqlite_strings: PS_SQLITE_STRINGS
      backend: PS_GENERIC_LAYOUT_SQL_BACKEND
    do
      create database.make (sqlite_filename)
      create sqlite_strings
      create backend.make (database, sqlite_strings)
      create Result.make (backend)
    end
end
```

Listing 5.1: *Setting up a MySQL and a SQLite repository*

All examples from this tutorial work exactly the same, no matter if you use the *PS_IN_MEMORY_REPOSITORY* or any of the provided database repositories.

Chapter 6

Transaction handling

Every CRUD operation in ABEL is by default executed within a transaction. Transactions are created and committed implicitly, which has the advantage that - especially when dealing with complex object graphs - an object doesn't get inserted halfway in case of an error.

As a user, you also have the possibility to use transactions explicitly. This is done by manually creating an object of type *PS_TRANSACTION* and using the `*_within_transaction` features in *PS_CRUD_EXECUTOR* instead of the normal ones. For your convenience there is a factory method `new_transaction` in class *PS_CRUD_EXECUTOR*.

Let's consider an example where you want to update the age of every person by one:

```
update_ages
  -- Increase everyone's age by one.
  local
    query: PS_OBJECT_QUERY [PERSON]
    transaction: PS_TRANSACTION
  do
    create query.make
    transaction := executor.new_transaction

    executor.execute_query_within_transaction (query,
        transaction)

    across query as query_result
    loop
      query_result.item.celebrate_birthday
      executor.update_within_transaction
        (query_result.item, transaction)
```

```
    end

    transaction.commit

    -- The commit may have failed
    if transaction.has_error then
      if attached transaction.error.message as msg then
        print ("Commit has failed. Error: " + msg)
      end
    end
  end
```

You can see here that a commit can fail in some situations, e.g. when a write conflict happened in the database. The errors are reported in the `PS_TRANSACTION.has_error` attribute. In case of an error, all changes of the transaction are rolled back automatically.

You can also abort a transaction manually by calling feature `rollback` in class `PS_TRANSACTION`.

6.1 Transaction isolation levels

ABEL supports the four standard transaction isolation levels found in almost every database system:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

The different levels are defined in `TRANSACTION_ISOLATION_LEVEL`. You can change the transaction isolation level by calling feature `set_transaction_isolation_level` in class `PS_REPOSITORY`. The default transaction isolation level of ABEL is defined by the actual storage backend.

Please note that not every backend supports all isolation levels. Therefore a backend can also use a more restrictive isolation level than you actually instruct it to use, but it is not allowed to use a less restrictive isolation level.

Chapter 7

Error handling

As ABEL is dealing with IO and databases, runtime errors may happen. The library will in general raise an exception in case of an error and expose the error to the library user as an `PS_ERROR` object. ABEL recognizes two different kinds of errors:

- Irrecoverable errors: fatal errors happening in scenarios like a dropped connection or a database integrity constraint violation. The default behavior is to rollback the current transaction and raise an exception. If you catch the exception in a rescue clause and manage to solve the problem, you can continue using ABEL.
- Recoverable errors: exceptional situations typically not visible to the user, because no exception is raised when they occur. An example is a conflict between two transactions. ABEL will detect the issue and, in case of implicit transaction management, retry. If you use explicit transaction management, ABEL will just doom the current transaction to fail at commit time.

ABEL maps database specific error messages to its own representation for errors, which is a hierarchy of classes rooted at `PS_ERROR`. The following list shows all error classes that are currently defined.

If not explicitly stated otherwise, the errors in this lists belong to the first category (fatal errors).

- `CONNECTION_PROBLEM`: A broken internet link, or a deleted serialization file.
- `TRANSACTION_CONFLICT`: A write conflict between two transactions. This is a recoverable error.

- *UNRESOLVABLE_TRANSACTION_CONFLICT*: A write conflict between implicit transactions that doesn't resolve after a retry.

- *ACCESS_RIGHT_VIOLATION*: Insufficient privileges in database, or no write permission to serialization file.

- *VERSION_MISMATCH*: The stored version of an object isn't compatible any more to the current type.

- *INTERNAL_ERROR*: Any error happening inside the library, e.g. a wrong SQL compilation.

- *GENERAL_ERROR*: Anything that doesn't fit into one of the categories above.

If you want to handle an error, you have to add a **rescue** clause somewhere in your code.

You can get the actual error from the feature *PS_CRUD_EXECUTOR*.*error* or *PS_TRANSACTION*.*error* or - due to the fact that the *PS_ERROR* class inherits from *DEVELOPER_EXCEPTION* - by performing an object test on Eiffel's *EXCEPTION_MANAGER*.*last_exception*.

For your convenience, there is a visitor pattern for all ABEL error types. You can just implement the appropriate functions and use it for your error handling code.

The following code shows an example. Note that only some important features are shown:

```
class
 MY_PRIVATE_VISITOR
inherit
 PS_ERROR_VISITOR

feature
 shall_retry: BOOLEAN
  -- Should my client retry the operation?

 visit_access_right_violation (
  error: PS_ACCESS_RIGHT_VIOLATION)
  -- Visit an access right violation error.
  do
   add_some_privileges
   shall_retry := True
  end
```

```
  visit_connection_problem (error: PS_CONNECTION_PROBLEM)
    -- Visit a connection problem error.
    do
      notify_user_of_abort
      shall_retry:=False
    end

end

class
  TUTORIAL

feature

  my_visitor: MY_PRIVATE_VISITOR
    -- A user-defined visitor to react to an error.

  executor: PS_CRUD_EXECUTOR
    -- The CRUD executor used throughout the tutorial.


  do_something_with_error_handling
    -- Perform some operations. Deal with errors in case of
       a problem.
    do
      -- Some complicated operations
    rescue
      my_visitor.visit (executor.error)
      if my_visitor.shall_retry then
        retry
      else
        -- The exception propagates upwards, and maybe
        -- another feature can handle it
      end
    end
end
```

Listing 7.1: *Sample error handling using a visitor.*