

# The ABEL Persistence Library Tutorial

**Written by:** Roman Schmocker

**Reviewed by:** Marco Piccioni

**Last updated:**

July 28, 2012

# Contents

<b>1</b>	<b>Introducing ABEL</b>	<b>2</b>
1.1	Setting things up . . . . .	2
1.2	Getting started . . . . .	2
<b>2</b>	<b>Basic operations</b>	<b>6</b>
2.1	Inserting . . . . .	6
2.2	Querying . . . . .	7
2.3	Updating . . . . .	8
2.4	Deleting . . . . .	8
2.5	Dealing with Known Objects . . . . .	9
<b>3</b>	<b>Advanced Queries</b>	<b>10</b>
3.1	The query mechanism . . . . .	10
3.2	Criteria . . . . .	10
3.2.1	Predefined Criteria . . . . .	10
3.2.2	Agent Criteria . . . . .	11
3.2.3	Creating criteria objects . . . . .	11
3.2.4	Combining criteria . . . . .	13
3.3	Deletion queries . . . . .	14
3.4	Tuple queries . . . . .	15
3.4.1	Tuple queries and projections . . . . .	16
3.4.2	Tuple queries and criteria . . . . .	16
<b>4</b>	<b>Dealing with references</b>	<b>17</b>
4.1	Updates . . . . .	20
4.2	Going deeper in the Object Graph . . . . .	20
<b>5</b>	<b>Advanced Initialization</b>	<b>21</b>
<b>6</b>	<b>Transaction handling</b>	<b>23</b>
6.1	Transaction isolation levels . . . . .	24



# 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<sup>1</sup>, 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
```

---

<sup>1</sup><https://svn.eiffel.com/eiffelstudio/branches/eth/eve/Src/library/abel>

```

3 create
  make

6 feature {NONE} -- Initialization

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

feature -- Basic operations
24
  celebrate_birthday
    -- Increase age by 1.
27 do
    age := age + 1
  ensure
30   age_incremented_by_one: age = old age + 1
end

33 feature -- Access

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

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

  age: INTEGER
42   -- The person's age.

invariant

```

```

45  age_non_negative: age >= 0
    first_name_exists: not first_name.is_empty
    last_name_exists: not last_name.is_empty
48 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

3  create
    make

6  feature {NONE} -- Initialization

    make

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

    feature

18

```

```
    executor: PS_CRUD_EXECUTOR
    -- The CRUD executor used throughout the tutorial.
21
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*. Let's add three new persons to the database in feature *explore*:

```
explore
  -- Tutorial code.
3  local
    p1, p2, p3: PERSON
  do
6    -- Insert 3 new persons in the database
    create p1.make ("Albo", "Bitossi")
    p1.celebrate_birthday
9    executor.insert (p1)
    create p2.make ("Berno", "Citrini")
    p2.celebrate_birthday
12   p2.celebrate_birthday
    p2.celebrate_birthday
    executor.insert (p2)
15   create p3.make ("Dumbo", "Ermini")
    executor.insert (p3)
  end
```

*Listing 2.1: Insertion code.*



## 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.
3  local
    query: PS_OBJECT_QUERY [PERSON]
  do
6    create Result.make
    create query.make
    executor.execute_query (query)
9
    across query as query_result
  loop
12   Result.extend (query_result.item)
  end
end
```

**Listing 2.2:** A simple query.

We now add in feature `explore` the code to print the linked list returned by feature `simple_query`:

```
explore
-- Tutorial code.
3  local
    p1, p2: PERSON
  do
6    -- Same code as before
    -- Query the database and print result
    print_result (simple_query)
9  end
```

**Listing 2.3:** Printing the query result.

Feature `print_result` takes the linked list result of the query and prints all its elements. 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`. Let's update the `age` attribute of Berno Citrini by celebrating his birthday:

```
explore
  -- Tutorial code.
3  local
    p1, p2: PERSON
  do
6    -- Same code as before
    -- Update an existing person in the database and print
      the result again
    p2.celebrate_birthday
9    executor.update (p2)
    print_result (simple_query)
  end
```

*Listing 2.4: Printing the query result.*

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 feature `delete` in `PS_CRUD_EXECUTOR`. Let's now delete Albo Bitossi from the database:

```
explore
  -- Tutorial code.
3  local
    p1, p2: PERSON
  do
6    -- Same code as before
    -- Delete Dumbo Ermini from the database and print the
      result again
    executor.delete (p3)
9    print_result (simple_query)
  end
```

*Listing 2.5: Deleting an object.*

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
2  -- Try and fail to update a new person object
   local
       a_person: PERSON
5  do
       create a_person.make ("Bob", "Barath")
       executor.update (a_person)
8  -- Results in a precondition violation
   end

11 failing_delete
   -- Try and fail to delete a new person object
   local
14   a_person:PERSON
   do
       create a_person.make ("Cersei", "Lannis")
17   executor.delete (a_person)
       -- Results in a precondition violation
   end
```

*Listing 2.6: 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
3 create
  default_create

6 feature -- Creating a criterion

  new alias "[]" (tuple: TUPLE [ANY]): PS_CRITERION
9   -- Creates a new criterion according to a 'tuple'
  -- containing either a single PREDICATE or three
  -- values of type [STRING, STRING, ANY].
12
```

```

    new_agent (a_predicate: PREDICATE [ANY, TUPLE [ANY]]):
        PS_CRITERION
        -- Creates an agent criterion.
15
    new_predefined (object_attribute: STRING;
        operator: STRING; value: ANY): PS_CRITERION
18    -- Creates a predefined criterion.

feature -- Operators
21
    equals: STRING = "="

24    greater: STRING = ">"

    greater_equal: STRING = ">="
27
    less: STRING = "<"

30    less_equal: STRING = "<="

    like_string: STRING = "like"
33
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
3    -- Create a new criteria using the traditional approach.

```

```

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

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

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

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

24 age_less_than (person: PERSON; age: INTEGER): BOOLEAN
    -- An example agent
do
27   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:

```

1   search_albo_bitossi : PS_CRITERION
    -- Combining criterion objects.
4   local
      first_name_criterion: PS_CRITERION
      last_name_criterion: PS_CRITERION
7   age_criterion: PS_CRITERION
do

```

```

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

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

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

Result := first_name_criterion and last_name_criterion
        and not age_criterion

19
-- using double brackets for compactness.
Result := factory[[ "first_name", "=", "Albo" ]]
22     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.
3  local
    deletion_query: PS_OBJECT_QUERY [PERSON]
    criterion:PS_PREDEFINED_CRITERION
6  do
    create deletion_query.make
    create criterion.make ("last_name", "=", name)
9    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.
3  local
    query: PS_TUPLE_QUERY [PERSON]
    last_name_index: INTEGER
6    single_result: TUPLE
do
    create query.make
9    ---- Find out at which position in the tuple the
        last_name is.
    last_name_index := query.attribute_index ("last_name")
    from
12    executor.execute_tuple_query (query)
    until
        query.result_cursor.after
15    loop
        single_result:= query.result_cursor.item
        print (single_result [last_name_index] )
18    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.
3  local
    query: PS_TUPLE_QUERY [PERSON]
  do
6    create query.make
    -- Only return the last_name of persons
    query.set_projection (<<"last_name">>)
9    -- Only return persons with age = 20
    query.set_criterion (factory [{"age", "=", 20}])
    from
12    executor.execute_tuple_query (query)
    until
      query.result_cursor.after
15  loop
    -- As we only have the last_name in the tuple,
    -- its index has to be 1.
18    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 re-constructing 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
3  CHILD

create
6  make

feature {NONE} -- Initialization
9
    make (first, last: STRING)
        -- Create a new child.
12  require
        first_exists: not first.is_empty
        last_exists: not last.is_empty
15  do
        first_name := first
        last_name := last
18  age := 0
    ensure
        first_name_set: first_name = first
21  last_name_set: last_name = last
```

```

    default_age: age = 0
  end
24  feature -- Access

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

    feature -- Status report

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

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

42    age: INTEGER
      -- The child's age.

45    feature -- Parents

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

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

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

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

```

```

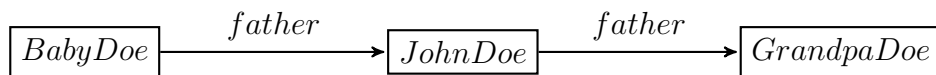
        father := a_father
66  ensure
    father_set: father = a_father
    end
69
invariant
    age_non_negative: age >= 0
72  first_name_exists: not first_name.is_empty
    last_name_exists: not last_name.is_empty
end

```

*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.
3   require
    child_persistent: executor.is_persistent (a_child)
  do
6     a_child.father.celebrate_birthday

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

    -- however, it works that way
12    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
3  REPOSITORY_FACTORY

feature -- Connection details
6
    username:STRING = "tutorial"
    password:STRING = "tutorial"
9
    db_name:STRING = "tutorial"
    db_host:STRING = "127.0.0.1"
12 db_port:INTEGER = 3306

    sqlite_filename: STRING = "tutorial.db"
15
feature -- Factory methods

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

```

21     local
        database: PS_MYSQL_DATABASE
        mysql_strings: PS_MYSQL_STRINGS
        backend: PS_GENERIC_LAYOUT_SQL_BACKEND
24     do
        create database.make (username, password, db_name,
            db_host, db_port)
        create mysql_strings
27     create backend.make (database, mysql_strings)
        create Result.make (backend)
    end
30
    create_sqlite_repository: PS_RELATIONAL_REPOSITORY
    -- Create an SQLite repository
33     local
        database: PS_SQLITE_DATABASE
        sqlite_strings: PS_SQLITE_STRINGS
36     backend: PS_GENERIC_LAYOUT_SQL_BACKEND
    do
        create database.make (sqlite_filename)
39     create sqlite_strings
        create backend.make (database, sqlite_strings)
        create Result.make (backend)
42     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.
3  local
    query: PS_OBJECT_QUERY [PERSON]
    transaction: PS_TRANSACTION
6  do
    create query.make
    transaction := executor.new_transaction
9
    executor.execute_query_within_transaction (query,
        transaction)

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

```

    end
18
    transaction.commit

21    -- The commit may have failed
    if transaction.has_error then
        if attached transaction.error.message as msg then
24            print ("Commit has failed. Error: " + msg)
        end
    end
27 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
3  MY_PRIVATE_VISITOR
  inherit
    PS_ERROR_VISITOR
6
  feature
    shall_retry: BOOLEAN
9    -- Should my client retry the operation?

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

```

```

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

    end

27     class
        TUTORIAL
30
        feature

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

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

39
        do_something_with_error_handling
        -- Perform some operations. Deal with errors in case of
        a problem.
42     do
        -- Some complicated operations
    rescue
45     my_visitor.visit (executor.error)
        if my_visitor.shall_retry then
            retry
48     else
        -- The exception propagates upwards, and maybe
        -- another feature can handle it
51     end
    end
end
end

```

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