

# The ABEL Persistence Library Tutorial

Roman Schmocker, Pascal Roos, Marco Piccioni

Last updated:

November 19, 2013

# 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 . . . . .	7
2.4	Deleting . . . . .	9
2.5	Dealing with Known Objects . . . . .	9
<b>3</b>	<b>Advanced Queries</b>	<b>11</b>
3.1	The query mechanism . . . . .	11
3.2	Criteria . . . . .	11
3.2.1	Predefined Criteria . . . . .	11
3.2.2	Agent Criteria . . . . .	12
3.2.3	Creating criteria objects . . . . .	12
3.2.4	Combining criteria . . . . .	14
<b>4</b>	<b>Dealing with references</b>	<b>16</b>
4.1	Inserting objects with dependencies . . . . .	16
4.2	Going deeper in the Object Graph . . . . .	19
<b>5</b>	<b>Error handling</b>	<b>20</b>

# 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

ABEL is shipped with EiffelStudio in the *unstable* directory. You can get the latest code from the SVN directory <sup>1</sup>.

If you want to modify the sample code used in this tutorial, just check out the tutorial code from SVN <sup>2</sup>.

### 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
```

```
3 create  
   make
```

---

<sup>1</sup><https://svn.eiffel.com/eiffelstudio/trunk/Src/unstable/library/persistence/abel>

<sup>2</sup>[https://svn.eiffel.com/eiffelstudio/trunk/Src/unstable/library/persistence/abel/sample/tutorial\\_api](https://svn.eiffel.com/eiffelstudio/trunk/Src/unstable/library/persistence/abel/sample/tutorial_api)

```

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. It can only be used for read operations.
- The *PS\_TRANSACTION* class represents a transaction and can be used to execute read, insert and update operations. Any *PS\_TRANSACTION* object is bound to 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\_EXECUTOR*. The result will be objects of type *G*.

To start using the library, we first need to create a *PS\_REPOSITORY*. For this tutorial we are going to use an in-memory repository to avoid setting up any external database. Each ABEL backend will ship a repository factory class to make initialization easier. The factory for the in-memory repository is called *PS\_IN\_MEMORY\_REPOSITORY\_FACTORY*.

```
class START

3 create
  make

6 feature {NONE} -- Initialization

  make
  -- Initialization for 'Current'.
  local
    factory: PS_IN_MEMORY_REPOSITORY_FACTORY
12 do
    create factory
    repository := factory.new_repository
15
    create criterion_factory
    explore
18 end

  repository: PS_REPOSITORY
21 -- The main repository.
```

**end**

24

**end**

***Listing 1.2: The START class***

We will use `criterion_factory` later in this tutorial. The feature `explore` will guide us through the rest of this API tutorial and show the possibilities in ABEL.

# Chapter 2

## Basic operations

### 2.1 Inserting

You can insert a new object using feature *insert* in *PS\_TRANSACTION*. As every write operation in ABEL needs to be embedded in a transaction, you first need to create a *PS\_TRANSACTION* object. Let's add three new persons to the database:

```
insert_persons
-- Populate the repository with some person objects.
3  local
    p1, p2, p3: PERSON
    transaction: PS_TRANSACTION
6  do
    -- Create persons
    create p1.make (...)
9  create ...

    -- We first need a new transaction.
12 transaction := repository.new_transaction

    -- Now we can insert all three persons.
15 transaction.insert (p1)
    transaction.insert (p2)
    transaction.insert (p3)
18
    -- Don't forget to commit.
    transaction.commit
21 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\_REPOSITORY* or *PS\_TRANSACTION*. The generic parameter *G* denotes the type of objects that should be queried.

After a successful execution of the query, you can iterate over the result using the **across** syntax. The feature *print\_persons* below shows how to get and print a list of persons from the repository:

```
print_persons
  -- Print all persons in the repository
3  local
    query: PS_OBJECT_QUERY[PERSON]
  do
6    -- First create a query for PERSON objects.
    create query.make

9    -- Execute it against the repository.
    repository.execute_query (query)

12   -- Iterate over the result.
    across
      query as person_cursor
15   loop
      print (person_cursor.item)
    end
18
    -- Don't forget to close the query.
    query.close
21 end
```

*Listing 2.2: Print all PERSON objects.*

In a real database the result of a query may be 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\_TRANSACTION*. Like the insert operation, an update needs to happen within a transaction.



Note that in order to *update* an object, we first have to retrieve it.

Let's update the *age* attribute of Berno Citrini by celebrating his birthday:

```
update_berno_citrini
-- Increase the age of Berno Citrini by one.
3  local
    query: PS_OBJECT_QUERY[PERSON]
    transaction: PS_TRANSACTION
6  berno: PERSON
do
    print ("Updating Berno Citrini's age by one.%N")
9
    -- Create query and transaction.
    create query.make
12   transaction := repository.new_transaction

    -- As we're doing a read followed by a write, we
15   -- need to execute the query within a transaction.
    transaction.execute_query (query)

18   -- Search for Berno Citrini
    across
        query as cursor
21   loop
        if cursor.item.first_name ~ "Berno" then
            berno := cursor.item
24
            -- Change the object.
            berno.celebrate_birthday
27
            -- Perform the database update.
            transaction.update (berno)
30   end
end

33   query.close
    transaction.commit
end
```

**Listing 2.3:** Update Berno Citrini's age.

To perform an update the object first needs to be retrieved or inserted within the same transaction. Otherwise ABEL cannot map the Eiffel object to its database counterpart (see also Section 2.5).

## 2.4 Deleting

ABEL does not support explicit deletes any longer, as it is considered dangerous for shared objects. Instead of deletion it is planned to introduce a garbage collection mechanism in the future.

## 2.5 Dealing with Known Objects

Within a transaction ABEL keeps track of objects that have been inserted or queried. This is important because in case of an update, 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 an object that is not yet known to ABEL. As an example, the following functions will fail:

```

    failing_update
      -- Trying to update a new person object.
3    local
      bob: PERSON
      transaction: PS_TRANSACTION
6    do
      create bob.make ("Robert", "Baratheon")
      transaction := repository.new_transaction
9      -- Error: Bob was not inserted / retrieved before.
      transaction.update (bob)
      transaction.commit
12    end

    update_after_commit
15    -- Update after transaction committed.
    local
      joff: PERSON
18    transaction: PS_TRANSACTION
    do
      create joff.make ("Joffrey", "Baratheon")
21    transaction := repository.new_transaction
      transaction.insert (joff)
      transaction.commit
24
      joff.celebrate_birthday

27    -- Prepare can be used to restart a transaction.
```

```

transaction.prepare

30      -- Error: Joff was not inserted / retrieved before.
transaction.update (joff)

33      -- Note: After commit and prepare, 'transaction'
      -- represents a completely new transaction.
end

```

**Listing 2.4:** *Common pitfalls with update.*

The feature *is\_persistent* in *PS\_TRANSACTION* 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 executing it against a *PS\_TRANSACTION* or *PS\_REPOSITORY*. The actual value of the generic parameter *G* determines the type of the objects that will be returned. At the moment descendants of *G* will not be loaded, but this behaviour may change in the future.

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\_criterion* 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 *PS\_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_uniform 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 = "<"

          less_equal: STRING = "<="
30      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 sugar” way
  - *f*[[*an\_attr\_name*, *an\_operator*, *a\_value*]]
  - *f*[[**agent** *an\_agent*]]

caption=The *PS\_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)

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

24  age_more_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* ≤ 20, you can just create a criterion object for each of the constraints and combine them:

```

1  composite_search_criterion : 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[[ agent age_more_than (?, 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[[ agent age_more_than (?, 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**.



# 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).

### 4.1 Inserting objects with dependencies

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

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

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

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

39 feature -- Element Change

    celebrate_birthday
42     -- Increase age by 1.
    do
      age := age + 1
45    ensure
      age_incremented_by_one: age = old age + 1
    end
48

    set_father (a_father: CHILD)
      -- Set a father for the child.
51    do
      father := a_father
    ensure
54     father_set: father = a_father
    end

57 invariant
    age_non_negative: age >= 0
    first_name_exists: not first_name.is_empty
60    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 inserting "Baby Doe", an insert of "John Doe" or "Grandpa Doe" will result in a precondition violation, because they have already been inserted as references of "Baby Doe".

In our main tutorial class *START* we have the following two features that show how to deal with object graphs. You will notice it is very similar to the corresponding routines for the flat *PERSON* objects.

```
insert_children
-- Populate the repository with some children objects.
3  local
    c1, c2, c3: CHILD
    transaction: PS_TRANSACTION
6  do
    -- Create the object graph.
    create c1.make ("Baby", "Doe")
```

```

9      create c2.make ("John", "Doe")
      create c3.make ("Grandpa", "Doe")
      c1.set_father (c2)
12     c2.set_father (c3)

      print ("Insert 3 children in the database.%N")
15     transaction := repository.new_transaction

      -- It is sufficient to just insert "Baby Joe",
18     -- as the other CHILD objects are (transitively)
      -- referenced and thus inserted automatically.
      transaction.insert (c1)
21     transaction.commit
      end

24     print_children
      -- Print all children in the repository
      local
27     query: PS_QUERY[CHILD]
      do
      create query.make
30     repository.execute_query (query)

      -- The result will also contain
33     -- all referenced CHILD objects.
      across
      query as person_cursor
36     loop
      print (person_cursor.item)
      end
39     query.close
      end

```

*Listing 4.2: Dealing with object graphs.*

## 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.

# Chapter 5

## Error handling

As ABEL is dealing with I/O 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 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 with some examples (the *PS\_* prefix is omitted for brevity):

- *CONNECTION\_SETUP\_ERROR*: No internet link, or a deleted serialization file.
- *AUTHORIZATION\_ERROR*: Usually a wrong password.
- *BACKEND\_ERROR*: An unrecoverable error in the storage backend, e.g. a disk failure.
- *INTERNAL\_ERROR*: Any error happening inside ABEL.
- *PS\_OPERATION\_ERROR*: For invalid operations, e.g. no access rights to a table.
- *TRANSACTION\_ABORTED\_ERROR*: A conflict between two transactions.
- *MESSAGE\_NOT\_UNDERSTOOD\_ERROR*: Malformed SQL or JSON statements.
- *INTEGRITY\_CONSTRAINT\_VIOLATION\_ERROR*: The operation violates an integrity constraint in the database.
- *EXTERNAL\_ROUTINE\_ERROR*: An SQL routine or triggered action has failed.

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

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 *PS\_TRANSACTION.last\_error* or - due to the fact that *PS\_ERROR* inherits from *DEVELOPER\_EXCEPTION* - by performing an object test on *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:

```

class
3  MY_PRIVATE_VISITOR

inherit
6  PS_DEFAULT_ERROR_VISITOR
    redefine
        visit_transaction_aborted_error,
9     visit_connection_setup_error
    end

12 feature -- Status report

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

feature -- Visitor features
18
    visit_transaction_aborted_error (tae:
        PS_TRANSACTION_ABORTED_ERROR)
        -- Visit a transaction aborted error
21    do
        shall_retry := True
    end

24    visit_connection_setup_error (cse:
        PS_CONNECTION_SETUP_ERROR)
        -- Visit a connection setup error
27    do
        notify_user_of_abort
        shall_retry:=False

```

```

30     end

    feature {NONE} -- Pseudocode
33
        notify_user_of_abort
            -- Notify the user that the operation has been aborted
36    do
        end

39 end

class
42     EXAMPLE

    feature

45
        my_visitor: MY_PRIVATE_VISITOR
            -- A user-defined visitor to react to an error.

48
        do_something_with_error_handling
            -- Perform some operations. Deal with errors in case of
            a problem.
51    local
        transaction: PS_TRANSACTION
    do
54        -- Some complicated operations
    rescue
        my_visitor.visit (executor.last_error)
57    if my_visitor.shall_retry then
        retry
    else
60        -- The exception propagates upwards, and maybe
        -- another feature can handle it
        end
63    end
end
end

```

*Listing 5.1: Sample error handling using a visitor.*