The ABEL Persistence Library Tutorial

Roman Schmocker, Pascal Roos, Marco Piccioni

Last updated:

November 20, 2013

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

ABEL is shipped with EiffelStudio in the *unstable* directory. You can get the latest code from the SVN directory [1].

If you want to modify the sample code used in this tutorial, just check out the tutorial code from SVN [2].

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

create
  make
```

[1] https://svn.eiffel.com/eiffelstudio/trunk/Src/unstable/library/persistency/abel

[2] https://svn.eiffel.com/eiffelstudio/trunk/Src/unstable/library/persistency/abel/sample/tutorial_api

```
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. 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_QUERY* [*G*] class is used to describe a read operation for 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

create
  make

feature {NONE} -- Initialization

  make
      -- Initialization for `Current'.
    local
      factory: PS_IN_MEMORY_REPOSITORY_FACTORY
    do
      create factory.make
      repository := factory.new_repository

      create criterion_factory
      explore
    end

  repository: PS_REPOSITORY
    -- The main repository.

  end
```

```
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.
  local
    p1, p2, p3: PERSON
    transaction: PS_TRANSACTION
  do
      -- Create persons
    create p1.make (...)
    create ...

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

      -- Now we can insert all three persons.
    transaction.insert (p1)
    transaction.insert (p2)
    transaction.insert (p3)

      -- Don't forget to commit.
    transaction.commit
  end
```

Listing 2.1: *Insertion code.*

2.2 Querying

A query for objects is done by creating a *PS_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
  local
    query: PS_QUERY[PERSON]
  do
    -- First create a query for PERSON objects.
    create query.make

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

    -- Iterate over the result.
    across
      query as person_cursor
    loop
      print (person_cursor.item)
    end

    -- Don't forget to close the query.
    query.close
  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.
  local
    query: PS_QUERY[PERSON]
    transaction: PS_TRANSACTION
    berno: PERSON
  do
    print ("Updating Berno Citrini's age by one.%N")

      -- Create query and transaction.
    create query.make
    transaction := repository.new_transaction

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

      -- Search for Berno Citrini
    across
      query as cursor
    loop
      if cursor.item.first_name ~ "Berno" then
        berno := cursor.item

          -- Change the object.
        berno.celebrate_birthday

          -- Perform the database update.
        transaction.update (berno)
      end
    end

    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.
  local
    bob: PERSON
    transaction: PS_TRANSACTION
  do
    create bob.make ("Robert", "Baratheon")
    transaction := repository.new_transaction
      -- Error: Bob was not inserted / retrieved before.
    transaction.update (bob)
    transaction.commit
  end

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

    joff.celebrate_birthday

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

```
    transaction.prepare

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

      -- 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 a *PS_QUERY[G]* object 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_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
create
  default_create

feature -- Creating a criterion

  new_uniform 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 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
  -- 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_more_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_more_than (?, 5)]]
  end

age_more_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 $age <= 20$, you can just create a criterion object for each of the constraints and combine them:

```

composite_search_criterion : 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[[ agent age_more_than (?, 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[[ 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 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).

4.1 Inserting objects with dependencies

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

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

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

  age: INTEGER
      -- The child's age.

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

feature -- Element Change

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

$$\boxed{BabyDoe} \xrightarrow{father} \boxed{JohnDoe} \xrightarrow{father} \boxed{GrandpaDoe}$$

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.
  local
    c1, c2, c3: CHILD
    transaction: PS_TRANSACTION
  do
      -- Create the object graph.
    create c1.make ("Baby", "Doe")
```

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

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

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

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

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

    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

Tuple queries

Consider a scenario in which you just want to have a list of all first names of `CHILD` objects in the database. Loading every attribute of each object of type `CHILD` might lead to a very bad performance, especially if there is a big object graph attached to each `CHILD` object.

To solve this problem ABEL allows queries which return data in `TUPLE` objects. Tuple queries are executed by calling `execute_tuple_query (a_tuple_query)` in either `PS_REPOSITORY` or `PS_TRANSACTION`, 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.

5.1 Tuple queries and projections

The `projection` feature in a `PS_TUPLE_QUERY` defines which attributes shall be included in the result `TUPLE`. Additionally, the order of the attributes in the projection array is the same as the order of the elements in the result tuples.

By default, a `PS_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`. 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.

5.2 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.

5.3 Example

```
explore_tuple_queries
    -- See what can be done with tuple queries.
  local
    query: PS_TUPLE_QUERY [CHILD]
    transaction: PS_TRANSACTION
    projection: ARRAYED_LIST [STRING]
  do
      -- Tuple queries are very similar to normal queries.
      -- I.e. you can query for CHILD objects by creating
      -- a PS_TUPLE_QUERY [CHILD]
    create query.make

      -- It is also possible to add criteria. Agent criteria
      -- are not supportedfor tuple queries however.
      -- Lets search for CHILD objects with last name Doe.
    query.set_criterion (criterion_factory
      [["last_name", criterion_factory.equals, "Doe"]])

      -- The big advantage of tuple queries is that you can
      -- define which attributes should be loaded.
      -- Thus you can avoid loading a whole object graph
      -- if you're just interested in e.g. the first name.
    create projection.make_from_array (<<"first_name">>)
    query.set_projection (projection)

      -- Execute the tuple query.
    repository.execute_tuple_query (query)

      -- The result of the query is a TUPLE containing the
      -- requested attribute.

    print ("Print all first names using a tuple query:%N")
```

```
  across
    query as cursor
  loop
      -- It is possible to downcast the TUPLE
      -- to a tagged tuple with correct type.
    check
      attached {TUPLE [first_name: STRING]}
        cursor.item as tuple
    then
      print (tuple.first_name + "%N")
    end
  end
end
```

Listing 5.1: *Using tuple queries.*

Chapter 6

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
  MY_PRIVATE_VISITOR

inherit
  PS_DEFAULT_ERROR_VISITOR
    redefine
      visit_transaction_aborted_error,
      visit_connection_setup_error
    end

feature -- Status report

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

feature -- Visitor features

  visit_transaction_aborted_error (tae:
      PS_TRANSACTION_ABORTED_ERROR)
      -- Visit a transaction aborted error
    do
      shall_retry := True
    end

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

```
    end

  feature {NONE} -- Pseudocode

   notify_user_of_abort
      -- Notify the user that the operation has been aborted
    do
    end

end

class
 EXAMPLE

feature

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

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

Listing 6.1: *Sample error handling using a visitor.*