

ABEL Tutorial

Roman Schmocker
Reviewed by: Marco Piccioni

July 23, 2012

Contents

Chapter 1

Getting started

1.1 Setting it up

In this tutorial, we are 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 you don’t need to inherit from any specific class to make them persistent.

```
class PERSON

3 create
  make

6 feature {NONE} -- Initialization

  make (first, last: STRING)
9    -- Create a new person
  do
    first_name := first
12    last_name := last
    age:= 0
  end

15 feature -- Basic operations

18  celebrate_birthday
    -- Increase age by 1
  do
21    age:= age+1
  end
```

```

24 feature -- Access

    first_name: STRING
27     -- First name of person
    last_name: STRING
    -- Last name of person
30
    age: INTEGER
    -- The person's age
33
end

```

Listing 1.1: The PERSON class

There are three very important classes in ABEL:

- The *CRUD_EXECUTOR* is, as the name suggests, responsible to execute CRUD commands. It is the core interface in ABEL and completely transparent to the actual storage backend.
- The *OBJECT_QUERY* [*G*] class is used to describe a read operation. You can execute such a query in the *CRUD_EXECUTOR*. The result will be objects of type *G*.
- The deferred class *REPOSITORY* provides an abstraction to the actual storage mechanism. Every *CRUD_EXECUTOR* is attached to a specific *REPOSITORY*.

1.2 Initialization

To start using the library, we first need to create a *REPOSITORY*. For this tutorial we will use a simple *IN_MEMORY_REPOSITORY*, which simulates a relational database but stores all values in memory. Although this repository will not give us any persistency, we use it here because initialization is very easy.

```

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 listings of Eiffel features are inside the *TUTORIAL* class, if they are not enclosed in another class declaration.

If you want to set up ABEL using a real persistence mechanism you can read section ??, “Advanced Initialization.”

Chapter 2

Basic operations

2.1 Querying

A query for objects is done by creating a *OBJECT_QUERY* [*G*] object and executing it in the *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 *OBJECT_QUERY.result_cursor*. Having an iteration cursor as a result has several advantages, e.g. support for lazy loading or the across syntax, as you will see in the next example:

```
simple_query: LINKED_LIST [PERSON]
-- Query all person objects 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
```

Usually the result of such a query is very big, and you are probably only interested in objects that meet a certain criteria, e.g. all persons of age 20. ABEL has a mechanism to support this kind of result filtering. You can read more about it in section ??.

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

2.2 Inserting and Updating

Inserting and updating an object is done through `CRUD_EXECUTOR.insert` (or `CRUD_EXECUTOR.update`, respectively):

```
simple_insert_and_update (a_person:PERSON)
  -- Insert 'a_person' into the current repository
3  do
    executor.insert (a_person)
    a_person.celebrate_birthday
6  executor.update (a_person)
  end
```

2.3 Deleting

Deletion is done through the `CRUD_EXECUTOR.delete` feature, like shown in the following example:

```
delete_person (name:STRING)
  -- Delete the person called 'name'.
3  local
    query: PS_OBJECT_QUERY[PERSON]
  do
6    -- First retrieve the person from the database
    create query.make
    executor.execute_query (query)
9    across query as query_result
    loop
      if query_result.item.last_name.is_equal (name) then
12
        -- Now delete him
        executor.delete (query_result.item)
15    end
    end
  end
```

Another way to delete objects is described in section ??.

2.4 Recognizing 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:

```
1  failing_update
   -- Try and fail to update a new person object
   local
4    a_person:PERSON
   do
       create a_person.make ("Albo", "Bitossi")
7      executor.update (a_person)
       -- Results in a precondition violation
   end
10
   failing_delete (name:STRING)
   -- Try and fail to delete a new person object
13  local
       a_person:PERSON
   do
16    create a_person.make ("Albo", "Bitossi")
       executor.delete (a_person)
       -- Results in a precondition violation
19  end
```

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

The `CRUD_EXECUTOR.is_persistent` feature 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, queries to the database are done by creating a new *OBJECT_QUERY* and letting it be executed by the *CRUD_EXECUTOR*. The generic parameter *G* of the *OBJECT_QUERY* instance 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 section ??).

3.2 Criteria

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

3.2.1 Predefined Criteria

Predefined criteria take 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 objects that don't satisfy the criterion.

Most supported operators are pretty self-describing (see Listing ??). The likely exception is the **like**-operator, which does pattern-matching on strings. You can give the **like**-operator a pattern as a value which 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 of a basic type, like strings or numbers, and not every type of attribute supports every 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.

3.2.3 Creating criteria objects

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

The main functions 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     -- This function creates a new criterion according to
    -- the tuple in the argument. The tuple should either
    -- contain a single PREDICATE or three values with
12    -- type [STRING, STRING, ANY]
```

```

15  new_agent (a_predicate: PREDICATE [ANY, TUPLE [ANY]]):
      PS_CRITERION
      -- creates an agent criterion

18  new_predefined (object_attribute: STRING;
      operator: STRING; value: ANY): PS_CRITERION
      -- creates a predefined criterion

21  feature -- Operators

24  equals: STRING = "="

      greater: STRING = ">"

27  greater_equal: STRING = ">="

30  less: STRING = "<"

      less_equal: STRING = "<="

33  like_string: STRING = "like"

36 end

```

Listing 3.1: The *CRITERION_FACTORY* interface

To create a new criterion, you basically have two possibilities. The first one is more traditional, by using *CRITERION_FACTORY.new_agent* or *CRITERION_FACTORY.new_predefined*.

The second possibility uses some syntactic sugar: The criterion is created with two brackets after the factory object, of which one is an overloaded operator and the other a tuple definition. It can be used for both types of criteria, and it is up to you to choose which approach you like best.

```

      create_criteria_traditional : PS_CRITERION
3  -- Create a new criteria using the traditional approach
      do

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

9

```

```

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

15
create_criteria_double_bracket : PS_CRITERION
-- Create a new criteria using the double bracket syntax
18 do

-- for predefined criteria
21 Result:= factory[["age", factory.less, 5]]

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

27
age_less_than (person: PERSON; age: INTEGER): BOOLEAN
-- An example agent
30 do
Result:= person.age < age
end

```

3.2.4 Combining criteria

If you want to set multiple criterion objects, you can combine them using the standard Eiffel keywords. 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
3  -- Create a criterion object that searches for an Albo
   Bitossi which is not 20 years old
local
   first_name_criterion:PS_CRITERION
6   last_name_criterion: PS_CRITERION
   age_criterion: PS_CRITERION
do
9   first_name_criterion:=
   factory[["first_name", factory.equals, "Albo" ]]

```

```

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

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

18     Result := first_name_criterion and last_name_criterion
        and not age_criterion

        -- or a bit shorter
21     Result := factory[[ "first_name", "=", "Albo" ]]
        and factory[[ "last_name", "=", "Bitossi" ]]
        and not factory[[ "age", "=", 20 ]]
24 end

```

ABEL supports the three standard logical operators **AND**, **OR** and **NOT**. Their precedence is 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. When you call `CRUD_EXECUTOR.execute_deletion_query`, 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 ??.

```

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

```

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

So far, we've only looked at queries that return objects. However, in ABEL there is a second option to query data which returns tuples as a result. Consider an example where you just want to have a list of all last names of persons in the database. To load 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, you can instead send a *TUPLE_QUERY* to the executor. 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 *TUPLE_QUERY.projection*.

```
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 returned
    last_name_index:= query.attribute_index ("last_name")

12 from
    executor.execute_tuple_query (query)
until
15    query.result_cursor.after
loop
    single_result:= query.result_cursor.item
18    print (single_result [last_name_index] )
    end
end
```

3.4.1 Tuple queries and projections

By default, a *TUPLE_QUERY* will only return attributes which are of a basic type, so no references are followed during a retrieve. You can change this default by calling *TUPLE_QUERY.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. It is ok to 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, but 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
9  query.set_projection (<<"last_name">>)

    -- Only return persons with age=20
12 query.set_criterion (factory [{"age", "=", 20}])

    from
15  executor.execute_tuple_query (query)
until
    query.result_cursor.after
18 loop
    -- As we only have the last_name in the tuple,
    -- its index has to be 1
21  print (query.result_cursor.item [1] )
end
end
```

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 that are of a basic type.

However, in Eiffel there's also the fact that 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 sloppily defined as all 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  do
        first_name := first
        last_name := last
15  age := 0
    end

18 feature -- Access

    celebrate_birthday
```



```

21      -- Increase age by 1
      do
        age:= age+1
24      end

      feature -- Status report
27
        first_name: STRING
          -- First name of child
30      last_name: STRING
          -- Last name of child

33      age: INTEGER
          -- The child's age

36      feature -- Parents

        mother: detachable CHILD
39      -- 'Current's mother

        father: detachable CHILD
42      -- 'Current's father

        set_mother (a_mother: CHILD)
45      -- Set the mother
        do
          mother:= a_mother
48      end

        set_father (a_father: CHILD)
51      -- Set the father
        do
          father:= a_father
54      end
      end
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 if their parent attribute is attached as well it has to repeat this procedure. The good news for you is 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 behaviour, as objects are stored completely that way, but it also has some side effects:

- 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

By default, ABEL does not follow references during an update. 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)

6  do
    a_child.father.celebrate_birthday

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

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

4.2 Going deeper

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 will cause a big performance impact.

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. You can see how this is done in section ?? in the technical documentation, where the whole concept of an object graphs and its depth is described more detailed.

Chapter 5

Advanced Initialization

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

To use them, you currently have to assemble the parts that are needed. For MySQL, you need to create a *MYSQL_DATABASE* and *MYSQL_STRINGS* object. You need them to create a *GENERIC_LAYOUT_SQL_BACKEND*, which you need in turn to create the *RELATIONAL_REPOSITORY*.

The following little factory class show the process for either a MySQL or 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

```

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

Chapter 6

Transaction handling

Every CRUD operation in ABEL is 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 "halfway inserted" 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 *TRANSACTION* and using the **_within_transaction* features in *CRUD_EXECUTOR* instead of the normal ones. For your convenience there is a factory method for transactions built into the 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
        print ("Commit has not been successful")
24    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 `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 `TRANSACTION.rollback`.

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 the feature `REPOSITORY.set_transaction_isolation_level`. The default transaction isolation level of ABEL is defined by the actual storage backend.

Please note: Not every backend supports all isolation levels. Therefore a backend can also use a higher isolation level than you actually instruct it to use, but it is not allowed to use a lesser 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 *ERROR* object. ABEL distinguishes between two different kinds of errors.

- **Fatal errors:** Irrecoverable errors happening in a scenario like a broken connection or an integrity constraint violation in the database. The usual measure is to rollback the current transaction and raise an exception. If you catch the exception in a rescue clause and manage to resolve the problem, you can continue using ABEL.
- **Resolvable failures:** Those are not really visible to the user, because no exception is raised when they occur. A typical example is a conflict between two transactions. ABEL will detect the failure and, in case of implicit transaction management, retry.

If you use explicit transaction management, it 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 set of classes with the common ancestor *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 resolvable failure.

- *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 *CRUD_EXECUTOR.error* or *TRANSACTION.error* or - due to the fact that the *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

30 class
    TUTORIAL

33 feature

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

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

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

```