The ABEL Persistence Library Tutorial

Roman Schmocker, Pascal Roos, Marco Piccioni

Last updated:

July 24, 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

We are assuming you have checked out the ABEL code from the EiffelStudio SVN repository[1], and have EiffelStudio installed. Launch EiffelStudio and in the initial window choose "tutorial_project". If it is not there just choose "Add project" and navigate to the location where you downloaded ABEL, and look for the *tutorial_project.ecf* project file in *abel/apps/sample/-tutorial/*. As the abel project file referenced by the tutorial project file [2] includes references to drivers that you might not have installed yet (e.g. MySQL-related drivers), you can comment out these references from the abel project file until your compilation succeeds [3]. You can then load and compile the project. To be able to compile the ABEL tutorial you don't need particular dependencies, because it is using an in-memory database simulating a relational database. If you want to experiment with ABEL's support for full-fledged relational back-ends (like MySQL or SQLite), you need to install the databases and the appropriate drivers first (also check the readme files when they are there).

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

[2] You can find the ABEL project file in *abel/libraries/ethz/src/abel/eiffelstore2.ecf*

[3] For example if some missing mysql references prevent your compilation from completing without errors you can try commenting out line *<library name="mysql" location="../mysqli/mysqli.ecf"/>* in *eiffelstore2.ecf*

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

feature {NONE} -- Initialization

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

feature -- Basic operations

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

feature -- Access

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

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

  age: INTEGER
    -- The person's age.

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

Listing 1.1: *The PERSON class*

There are three very important classes in ABEL:

- The deferred class `PS_REPOSITORY` provides an abstraction to the actual storage mechanism.
- The `PS_EXECUTOR` class is responsible to execute CRUD (Create Read Update Delete) commands. Every `PS_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_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 using a more specific object of type `PS_RELATIONAL_REPOSITORY`, and even more specifically a `PS_IN_MEMORY_REPOSITORY`, which simulates a relational repository while storing the data in memory. ABEL provides support for creating all kinds of `PS_REPOSITORY` objects through the factory class `PS_REPOSITORY_FACTORY`, so that is what we are going to use.
As a second step, we need to create an object of type `PS_EXECUTOR`. To do so, we will pass the previously created repository as an argument to its creation feature.

```
class TUTORIAL

create
  make

feature -- Tutorial exploration features
```

```
  explore
      -- Tutorial code.
    local
      in_memory_repo: PS_RELATIONAL_REPOSITORY
    do
      print ("---o--- ABEL Tutorial ---o---")
      io.new_line
      in_memory_repo := repo_factory.
         create_in_memory_repository
      create executor.make (in_memory_repo)
    end

feature -- Access

  repo_factory: PS_REPOSITORY_FACTORY
    -- Repository factory.

  executor: PS_EXECUTOR
    -- The executor of database operations used throughout
       the tutorial.

end
```

Listing 1.2: *The TUTORIAL class*

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

Chapter 2

Basic operations

2.1 Inserting

You insert an object in the repository using feature *execute_insert* in class
PS_EXECUTOR. Let's add three new persons to the database in feature *explore*:

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
  do
    print ("---o--- ABEL Tutorial ---o---")
    io.new_line
    in_memory_repo := repo_factory.
        create_in_memory_repository
    create executor.make (in_memory_repo)
    -- Insert 3 new persons in the database
    create p1.make ("Albo", "Bitossi")
    p1.celebrate_birthday
    executor.execute_insert (p1)
    create p2.make ("Berno", "Citrini")
    p2.celebrate_birthday
    p2.celebrate_birthday
    p2.celebrate_birthday
    executor.execute_insert (p2)
    create p3.make ("Dumbo", "Ermini")
    executor.execute_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_EXECUTOR`. The generic parameter `G` denotes the type of objects that should be queried.

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

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

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

Listing 2.2: *A simple query.*

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

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
  do
    -- Same code as before
    -- Query the database and print result
    print_result (simple_query)
```

```
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 `execute_update` in `PS_EXECUTOR` . Let's update the `age` attribute of Berno Citrini by celebrating his birthday:

```
explore
  -- Tutorial code.
 local
  in_memory_repo: PS_RELATIONAL_REPOSITORY
  p1, p2, p3: PERSON
 do
  -- Same code as before
  -- Update an existing person in the database and print
     the result again
  p2.celebrate_birthday
  executor.execute_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 `execute_delete` in `PS_EXECUTOR`. Let's now delete Albo Bitossi from the database:

```
explore
  -- Tutorial code.
 local
  in_memory_repo: PS_RELATIONAL_REPOSITORY
```

```
    p1, p2, p3: PERSON
  do
    -- Same code as before
    -- Delete Dumbo Ermini from the database and print the
       result again
    executor.execute_delete (p3)
    print_result (simple_query)
  end
```

Listing 2.5: *Deleting an object.*

The object to delete needs to be previously known to ABEL through an insert or a successful query (see Section 2.5). A way to delete objects that always works (because ABEL queries for them in advance) is described in Section 3.3.

2.5 Dealing with Known Objects

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

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

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

failing_delete
  -- Try and fail to delete a new person object
  local
    a_person:PERSON
  do
    create a_person.make ("Cersei", "Lannis")
    executor.execute_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_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_EXECUTOR*. The actual value of the generic parameter *G* determines the type of the objects that will be returned, including any conforming type (e.g. descendants of *G*).

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

3.2 Criteria

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

3.2.1 Predefined Criteria

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

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

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

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

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

3.2.2 Agent Criteria

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

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

3.2.3 Creating criteria objects

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

The main features of the class are the following:

```
class
  PS_CRITERION_FACTORY
create
  default_create

feature -- Creating a criterion

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

```

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

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

feature -- Operators

  equals: STRING = "="

  greater: STRING = ">"

  greater_equal: STRING = ">="

  less: STRING = "<"

  less_equal: STRING = "<="

  like_string: STRING = "like"

end
```

Listing 3.1: *The CRITERION_FACTORY class interface*

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

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

caption=The CRITERION_FACTORY interface

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

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

    -- for agent criteria
    Result :=
      factory.new_agent (agent age_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**.

We can now add the necessary code to feature *explore*:

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
  do
    -- Same code as before
    -- Search for Albo Bitossi with age <= 20
    print_result (query_with_composite_criterion)
  end
```

Listing 3.4: *Invoking the code that searches for Albo Bitossi*

Where feature *query_with_composite_criterion* looks like the following:

```
query_with_composite_criterion: LINKED_LIST [PERSON]
  -- Query using a composite criterion.
  local
    query: PS_OBJECT_QUERY [PERSON]
  do
```

```
    create Result.make
    create query.make
    query.set_criterion (composite_search_criterion)
    executor.execute_query (query)

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

***Listing 3.5:** Invoking the code that searches for Albo Bitossi*

As you may have noticed, it is very simple to set criteria on a query.

3.3 Deletion queries

As mentioned in Section 2.4, there is another way to perform a deletion in the repository from within `PS_EXECUTOR`. By calling `execute_deletion_query` instead of `execute_delete`, ABEL will delete all objects in the database that would have been retrieved by executing the query normally.

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

***Listing 3.6:** Using a deletion query.*

We can now add the necessary code to feature `explore`:

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
  do
```

```
    -- Same code as before
    -- Delete Albo Bitossi using a deletion query
    delete_person_with_deletion_query ("Bitossi")
    print_result (simple_query)
  end
```

Listing 3.7: *Invoking the code that searches for Albo Bitossi*

Using a deletion query instead of a direct delete command depends upon the situation. 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.

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

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

feature -- Status report

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

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

    age: INTEGER
        -- The child's age.

feature -- Parents

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

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

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

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

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

Listing 4.1: *The CHILD class.*

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

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

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

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

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

Here is the code in feature *explore* that tests what we have stated above:

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
    c1, c2, c3: CHILD
  do
    -- Same code as before
    print ("Insert 3 children in the database")
    create c1.make ("Baby", "Doe")
    create c2.make ("John", "Doe")
    create c3.make ("Grandpa", "Doe")
    c1.set_father (c2)
    c2.set_father (c3)
    executor.execute_insert (c1)
    io.new_line
    print ("Query the database for children and print
        result")
    print_children_result (query_for_children)
    print ("Inserting John Doe has no effect")
    executor.execute_insert (c2)
    print_children_result (query_for_children)
  end
```

Listing 4.2: *Inserting objects having references to other objects.*

You can find the code for *query_for_children* and *print_children_result* in the ABEL repository. You will notice it is very similar to the corresponding routines seen before (the only thing that changes is the kind of linked list that is passed as an argument).

4.2 Updating objects with dependencies

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

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
```

```
    c1, c2, c3: CHILD
  do
    -- Same code as before
    print ("Updating John Doe has no effect")
    if attached {CHILD} c1.father as dad then
      dad.celebrate_birthday
    end
    executor.execute_update (c1)
    print_children_result (query_for_children)
  end
```

Listing 4.3: *References are not followed by default during updates.*

Section 4.3 will tell you how do change the default settings.

4.3 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

Transaction handling

Every CRUD operation in ABEL is by default executed within a transaction. Transactions are created and committed implicitly. This is convenient when dealing with complex object graphs, because 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_EXECUTOR* instead of the normal ones. For your convenience there is a factory method *new_transaction* in class *PS_EXECUTOR*.

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

```
update_ages
  -- Increase the age of all persons by one.
  local
    query: PS_OBJECT_QUERY [PERSON]
    transaction: PS_TRANSACTION
  do
    create query.make
    transaction := executor.new_transaction

    executor.execute_query_within_transaction (query,
       transaction)

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

```
    end

    transaction.commit

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

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

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

As usual, here is the code for feature `explore`:

```
explore
    -- Tutorial code.
  local
    in_memory_repo: PS_RELATIONAL_REPOSITORY
    p1, p2, p3: PERSON
    c1, c2, c3: CHILD
  do
    -- Same code as before
    print ("Celebrating the birthday for all PERSON
       objects in the repository")
    update_ages
    print_result (simple_query)
  end
```

Listing 5.1: *Testing an update with explicit transaction.*

5.1 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 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 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_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 relevant features are shown:

```
class
 MY_PRIVATE_VISITOR
inherit
 PS_ERROR_VISITOR

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

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

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

class
  TUTORIAL

feature

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

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


  do_something_with_error_handling
    -- Perform some operations. Deal with errors in case of
       a problem.
    do
      -- Some complicated operations
    rescue
      my_visitor.visit (executor.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.*

Chapter 7

CouchDB Support

ABEL does not only work with an in-memory database. It is also able to store objects in other database, both relational (like MySQL and SQLite) and non-relational like CouchDB, always using the same API.

7.1 What is CouchDB

CouchDB is a free, open-source document-oriented database [1]. CouchDB stores objects on a persistent database using JSON documents. JSON is a textual notation similar to XML that stores Eiffel objects like this:

```
{
  "firstname": "Albo",
  "lastname": "Bitossi",
  "age": 0
}
```

Listing 7.1: *Sample Eiffel Object in JSON*

7.2 Setting up CouchDB

Before we can start using CouchDB from within Eiffel we have to set it up either on a local machine or get hold of a database on the internet. To install CouchDB locally visit `www.couchdb.com` and download the appropriate package.

Once installed, CouchDB should be running in the background and is accessible trough a browser by accessing *127.0.0.1:5984/_utils* To work with

[1] `http://couchdb.apache.org`

CouchDB in Eiffel we have created another tutorial which you can get at *abel/apps/sample/tutorial-couchdb/*. Look for the *tutorial_project.ecf* and open it with EiffelStudio.

7.3 Getting started with CouchDB

On the surface there is not much difference between using the in-memory database and CouchDB. You may notice that all we changed in the tutorial is the call to the repo_factory.

```
explore
    -- Tutorial code.
  local
    p1, p2, p3: PERSON
    c1, c2, c3: CHILD
    couchdb_repo: PS_RELATIONAL_REPOSITORY
  do
    print ("---o--- CouchDB Tutorial ---o---")
    io.new_line
    couchdb_repo := repo_factory.create_cdb_repository ("
        127.0.0.1", 5984)
    create executor.make (couchdb_repo)
    ...
```

Listing 7.2: *The CouchDB Tutorial*

Instead of using *repo_factory.create_in_memory_repository* we now use *repo_factory.create_cdb_repository("127.0.0.1", 5984)*. Whereby the first argument of this method denotes the URL where the database is located (In this case we use the localhost) and the second argument is the used port (we use the default CouchDB port which is 5984). If for some reason your couch is not located on your own machine, you might have to adjust these values to point to the correct location.

If you compare the output of this tutorial to the output you got when using the in-memory database you might notice that nothing changed. On the surface both these databases provide the same services. Namely storing Eiffel objects.

7.4 Beneath the surface

Using CouchDB, Eiffel can store objects on a persistent database that can also be accessed by other programs. If not deleted, the data will persist after your program has ended. To accomplish this, ABEL will convert Eiffel objects to JSON documents, whereby each attribute will get its own *"name": "value"* pair. The resulting document for a person will look similar to *Listing 7.1*. After running the tutorial, the stored objects can also be explored by visiting *127.0.0.1:5984/_utils*.

You will notice that for both person and child a sub-database was created. The person database will only contain person-objects and the child database will only contain child-objects. If you don't want your data to remain in the database after the program has ended, insert a *couchdb_repo.wipe_out* at the end of the feature explore

7.5 Limitations

CouchDB is not meant to be a relational database: it can nicely store objects as JSON Documents, that can then be searched by key. CouchDB was mainly developed for the world wide web. For its basic API it uses cURL which is really easy to use but for its more advanced features like map-reduce it uses JavaScript. Map-reduce would come in handy when querying for objects in the database but it is not yet integrated and therefore for queries rather than using the inbuilt map-reduce of CouchDB ABEL uses an Eiffel function to accomplish the same.

For more information on CoachDB see the online documentation.