Chair of
Software Engineering

A Better Eiffelstore Library

Bachelor Thesis

By: Roman Schmocker
Supervised by: Marco Piccioni
Prof. Dr. Bertrand Meyer

Student Number: 09-911-215

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

Abstract

ABEL is a persistence library that unifies existing solutions under a simple and transparent API. The API fully supports the CRUD operations, transactions, and has some advanced features like result filtering based on some criteria.

The library contains a powerful framework with an object-relational mapping layer at the core that allows to adapt ABEL to basically any existing persistence solution.

Contents

Chapter 1

Introduction

1.1 Introduction

The Eiffel language features a lot of different persistence libraries, and every solution has its own advantages and drawback. A database such as MySQL for example is quite fast, but it comes at the expence of the object-relational impedance mismatch. A serialization library on the other hand is easier to handle for the programmer, but its performance rapidly decreases for large data sets.

It is very hard to switch from one persistence solution to another, because all have their own interface. Even changing the database from MySQL to Oracle is usually very hard to achieve, if only because their SQL dialects are different.

To overcome such problems we have developed a new library called ABEL, which is the acronym for "A better Eiffelstore library". ABEL tries to unify existing persistence libraries unter a simple and yet powerful API, which is completely transparent to the actual storage mechanism.

1.2 Overview

This thesis is basically splitted into two parts: The API tutorial and the technical documentation.

In the first part you will be introduced to the basic operations of the API, like the CRUD (Create, Read, Update, Delete) operations or transaction handling.

The second part is an introduction to the general architecture of ABEL and some selected topics like the object-relational mapping layer or the main backend abstraction interfaces.

Chapter 2

API tutorial

2.1 Getting started

2.1.1 The setup

n this little 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.

```
1 class PERSON

  create
4   make

  feature {NONE} -- Initialization
7
    make (first, last: STRING)
       -- Create a new person
10    do
        first_name := first
        last_name := last
13      age:= 0
      end

16  feature -- Basic operations

    celebrate_birthday
19      -- Increase age by 1
      do
        age:= age+1
```

```
22    end

  feature -- Status report
25
    first_name: STRING
      -- First name of person
28  last_name: STRING
      -- Last name of person

31  age: INTEGER
      -- The person's age

34 end
```

Listing 2.1: The PERSON class

There are 3 very important classes in ABEL:

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

2.1.2 Initialization

To start using the library, we therefore 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 persistence, it is chosen here because the initialization is very easy.

```
1 class TUTORIAL

  create
4  make

  feature {NONE} -- Initialization
7
```

```
  make
    -- Initialize ABEL, set up a simple in-memory repository
    local
      repository:PS_IN_MEMORY_REPOSITORY
    do
      create repository.make
      create executor.make (repository)
    end

feature

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

end
```

Listing 2.2: 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" 2.5.

2.2 Basic operations

2.2.1 Query

A query for objects is done by creating a new OBJECT_QUERY [G] object and execute it with the CRUD_EXECUTOR. The generic parameter G of the OBJECT_QUERY instance denotes the type of objects that should be queried.

After a successful execution of the query, you can get the result through 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
    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
```

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, and you can read about it in section 2.3.

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

2.2.2 Insert and update

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
  do
    executor.insert (a_person)
    a_person.celebrate_birthday
    executor.update (a_person)
  end
```

2.2.3 Deletion

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

```
delete_person (name:STRING)
  -- Delete the person called 'name'.
  local
    query: PS_OBJECT_QUERY[PERSON]
  do
    -- First retrieve the person from the database
    create query.make
    executor.execute_query (query)
    across query as query_result
```

```
    loop
11    if query_result.item.last_name.equals (name) then

        -- Now delete him
14      executor.delete (query_result.item)
      end
    end
17  end
```

There is another way to delete objects which uses a query and some matching criteria. You can read about it in section 2.3.3.

2.2.4 Recognizing Objects

ABEL keeps track of objects that have been inserted or queried. This is important because in case of an update or delete, ABEL needs to internally 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 person that has never been
         inserted or queried before
    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 person that has never been
       inserted or queried before
13  local
      a_person:PERSON
    do
16    create a_person.make ("Albo", "Bitossi")
      executor.delete (a_person)
        -- Results in a precondition violation
19  end
```

The CRUD_EXECUTOR.is_persistent feature can tell you if a specific object is known to ABEL and therefore has a link to its entry in the database.

2.3 Advanced Queries

2.3.1 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 of the OBJECT_QUERY instance determines the type of the objects that will be returned, including every conforming types (e.g. descendants of that class).

ABEL will by default load an object completely, that means all objects that can be reached by following references will be loaded as well. There is however a mechanism in ABEL that ensures that an object will not be loaded twice in a single query (see also 2.4).

2.3.2 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: predefined and agent criteria.

Predefined Criteria

Predefined criteria take an attribute name, an operator and a value. You can only use attributes that are of a basic type, like strings or numbers. During retrieval, they check the object's own value against the value set in the criterion, and filter objects that don't satisfy the criterion.

The supported operators are pretty self-describing (see Listing 4), except for the "like"-operator. That does pattern-matching on strings. You can give the criterion 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.

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 therefore the result will be filtered in the database.

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.

Creating criteria objects

The criteria instances are best created using the CRITERION_FACTORY class.

The main functions to the class are the following:

```
class
  PS_CRITERION_FACTORY
create
  default_create

feature -- Creating a criterion

  new alias "[]" (tuple: TUPLE [ANY]): PS_CRITERION
      -- This function creates a new criterion according to
         the tuple in the argument.
      -- The tuple should either contain a single PREDICATE
         or three values of the form [STRING, STRING, ANY]



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

  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 2.3: The CRITERION_FACTORY interface

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

The second 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
    -- 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_less_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_less_than (?, 5)]]
  end


age_less_than (person: PERSON; age: INTEGER): BOOLEAN
  -- Just a little example agent
  do
    Result:= person.age < age
  end
```

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
  -- Create a criterion object that searches for an Albo
     Bitossi which is not 20 years old
  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[[ "age", factory.equals, 20 ]]

    Result := first_name_criterion and last_name_criterion
        and not age_criterion
```

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

2.3.3 Deletion queries

As already mentioned previously, there is another way to perform a delete 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 the delete section 2.2.3

```
delete_person (name:STRING)
  -- Delete '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", "=", name)
    deletion_query.set_criterion (criterion)
    executor.execute_deletion_query (deletion_query)
  end
```

It depends on 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 nicer to use if the object is not yet loaded from the database.

2.3.4 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 first an 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 iterator over a list of tuples in which the attributes of an object are collected. The order of these attributes is the one defined in TUPLE_QUERY.projection

```
print_all_names
  -- Print the last name of all PERSON objects in the
     database
local
  query: PS_TUPLE_QUERY[PERSON]
  last_name_index:INTEGER
  single_result: TUPLE
do
  create query.make
  -- Find out at which position in the tuple the
     last_name is returned
  last_name_index:= find_index_of_attribute("last_name")

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

Tuple queries and projections

By default, a TUPLE_QUERY will only return attributes of an object that 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 a list 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.

Tuple queries and criteria

As you can't use agents on tuples (as they expect an object and not a tuple), you are restricted to use predefined criteria in tuple queries. You can still combine them as usual. It is ok to include a 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_names_of_20_year_old
  -- Print the last name of all PERSON objects with age=20
  local
    query: PS_TUPLE_QUERY[PERSON]
  do
    create query.make

    -- Only return the last_name of persons
    query.set_projection (<<"last_name">>)

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

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

2.4 Dealing with references

In ABEL, a basic type is an object of type STRING, BOOLEAN, CHARACTER or any numeric object like REAL or INTEGER. The PERSON class has attributes that are of a basic type only, and those are stored together as a single unit in the database.

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 CHILD
```

```
4  create
     make

7  feature {NONE} -- Initialization

     make (first, last: STRING)
10      -- Create a new person
       do
         first_name := first
13       last_name := last
         age:= 0
       end
16
   feature -- Basic operations

19   celebrate_birthday
        -- Increase age by 1
       do
22       age:= age+1
       end

25 feature -- Status report

     first_name: STRING
28     -- First name of person
     last_name: STRING
       -- Last name of person
31
     age: INTEGER
       -- The person's age
34
   feature -- Parents

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

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

43   set_mother (a_mother: CHILD)
       do
         mother:= a_mother
```

```
46    end

    set_father (a_father: CHILD)
49    do
        father:= a_father
      end
52  end
```

Listing 2.4: 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 they too are instances of CHILD, then it has to repeat that again. The good news for you is that the examples above will work exactly the same way.

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" into the repository, 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 through the first statement.

2.4.1 Updates

By default, ABEL does not follow references on updates. So for example the following statement has no effect to the database.

```
celebrate_fathers_birthday (a_child: CHILD)
  require
    a_child_is_in_database: executor.is_already_loaded (
       a_child)
  do
    a_child.father.celebrate_birthday

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

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

2.4.2 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.

TODO: insert section reference

2.5 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 the 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 REPOSITORY_FACTORY
```

```
3
  feature -- Connection details

6   username:STRING = "tutorial"
    password:STRING = "tutorial"

9   db_name:STRING = "tutorial"
    db_host:STRING = "127.0.0.1"
    db_port:INTEGER = 3306
12
    sqlite_filename: STRING = "tutorial.db"

15 feature -- Factory methods

    create_mysql_repository: PS_RELATIONAL_REPOSITORY
18    -- Create a MySQL repository
      local
        database: PS_MYSQL_DATABASE
21      mysql_strings: PS_MYSQL_STRINGS
        backend: PS_GENERIC_LAYOUT_SQL_BACKEND
      do
24      create database.make (username, password, db_name,
           db_host, db_port)
        create mysql_strings
        create backend.make (database, strings)
27      create Result.make (backend)
      end

30  create_sqlite_repository: PS_RELATIONAL_REPOSITORY
      -- Create an SQLite repository
      local
33      database: PS_SQLITE_DATABASE
        sqlite_strings: PS_SQLITE_STRINGS
        backend: PS_GENERIC_LAYOUT_SQL_BACKEND
36    do
        create database.make (sqlite_filename)
        create sqlite_strings
39      create backend.make (database, strings)
        create Result.make (backend)
      end
42 end
```

That's all. You can now use the repository to create a CRUD_EXECUTOR and start programming on it. All examples from this tutorial work exactly

the same, no matter if you use the IN_MEMORY_REPOSITORY or any of the database repositories.

2.6 Transaction handling

Every CRUD operation in ABEL is executed within a transaction. Transactions are created and commited implicitly, and have e.g. 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 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
  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

    -- It might be that the transaction has encountered an
        error at commit time:
    if transaction.has_error then
      print ("Commit has not been successful")
    end
```

```
23  end
```

You can see here as well that a commit may fail, e.g. when a write conflict happened in the database. These errors are indicated 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.

2.6.1 Transaction isolation levels

ABEL supports the four standard transaction isolation levels found in most database systems:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

The different levels are defined in the "enum class" TRANSACTION_ISOLATION_LEVEL.

You can set a transaction isolation level using 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 need, but it is not allowed to use a lesser isolation level.

2.7 Error handling

As ABEL is dealing with IO and databases, there are several runtime errors that can happen. ABEL will in general raise exceptions if an error happens, and propagate errors through 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, then you can continue using ABEL.

- Resolvable failures: Those are not really visible to the user, because no exception is raised if 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 represents errors as objects. All specific error classes inherit from the deferred class ERROR. The following list denotes all errors that can happen at runtime. If not explicitly stated otherwise, the errors in this lists belong to the first category (Fatal errors).

- CONNECTION_LOST: Happens e.g. if the internet link breaks, or a serialization file gets exernally deleted.
- TRANSACTION_CONFLICT: Happens if there's a conflict between two transactions. This is a resolvable failure.
- ACCESS_RIGHT_VIOLATION: Can happen if you don't have write permission to the database.
-

Create all error types...

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 CRUD_EXECUTOR.error or TRANSACTION.error feature or - as ERROR 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.

Chapter 3

Technical documentation

3.1 Architecture overview

The ABEL library can be splitted into a Frontend and a Backend part. The frontend provides the main API, which is completely agnostic of the actual storage engine, whereas the backend provides a framework and some implementations to adapt ABEL to a specific storage engine. The boundary between Backend and Frontend can be drawn straight through the deferred class REPOSITORY.

3.1.1 Frontend

If you've read the previos part of the documentation, then you should be quite familiar now with the frontend. The main classes are:

- CRUD_EXECUTOR: Provides features for CRUD operations, and does some error handling for transaction conflicts.
- QUERY: Collects information like the Criteria, Projection, and (through its generic parameter) the type of the object to be retrieved. It doesn't do anything by itself.
- TRANSACTION: Represents an ABEL transaction, and is responsible to internally propagate errors. It provides the features commit and rollback, but internally it relies on REPOSITORY to execute them.
- CRITERION: Its descendants provide a filtering function for retrieved objects, and it has builtin functions to generate a tree of criteria using the overloaded logical operators.

You can see that the main objective of the frontend is to provide an easy to use, backend-agnostic API, and to collect information that the backend can consume.

Insert class diagram

3.1.2 Backend

The frontend needs a repository which is specific to a data storage engine, and the backend provides a framework to implement these repositories (in cluster Framework). There are also some already predefined repositories inside the backend (cluster Backends), like the IN_MEMORY_REPOSITORY.

The backend layers

The framework is built of several layers, each layer more specific to a persistence mechanism as it goes down.

The uppermost layer is the REPOSITORY class. It provides a very high level of abstraction which should be sufficient for all kinds of persistence mechanisms. It deals with normal Eiffel objects that may reference a lot of other objects.

One level below you can find the object-relational mapping layer. It is responsible to take an object graph apart into its pieces and generate a plan for the write operations. On the other hand it is responsible to build an object graph from its pieces during object retrieval. This layer is described more precisely in the next section 3.2.

On the next level there is the BACKEND_STRATEGY layer. It is responsible to map the object pieces generated before to a specific storage engine, e.g. a database with some table layout.

The lowest level of abstraction is only significant for databases that can understand SQL. It provides a set of wrapper classes that hide connection details and error handling and has a features to execute SQL and retrieve the result in a standardized way.

Important data structures

The key data structure in the framework is the OBJECT_IDENTIFICATION_MANAGER: It maintains a weak reference to every object that has been queried or inserted before, and assigns a repository-wide unique number to it (called the object_identifier). It is, for example, responsible for the fact that the update fails in section “Recognizing Objects” 2.2.4.

Another important datastructure is the KEY_POID_TABLE, which maps the objects object_identifier to the primary key of the corresponding entry in the database.

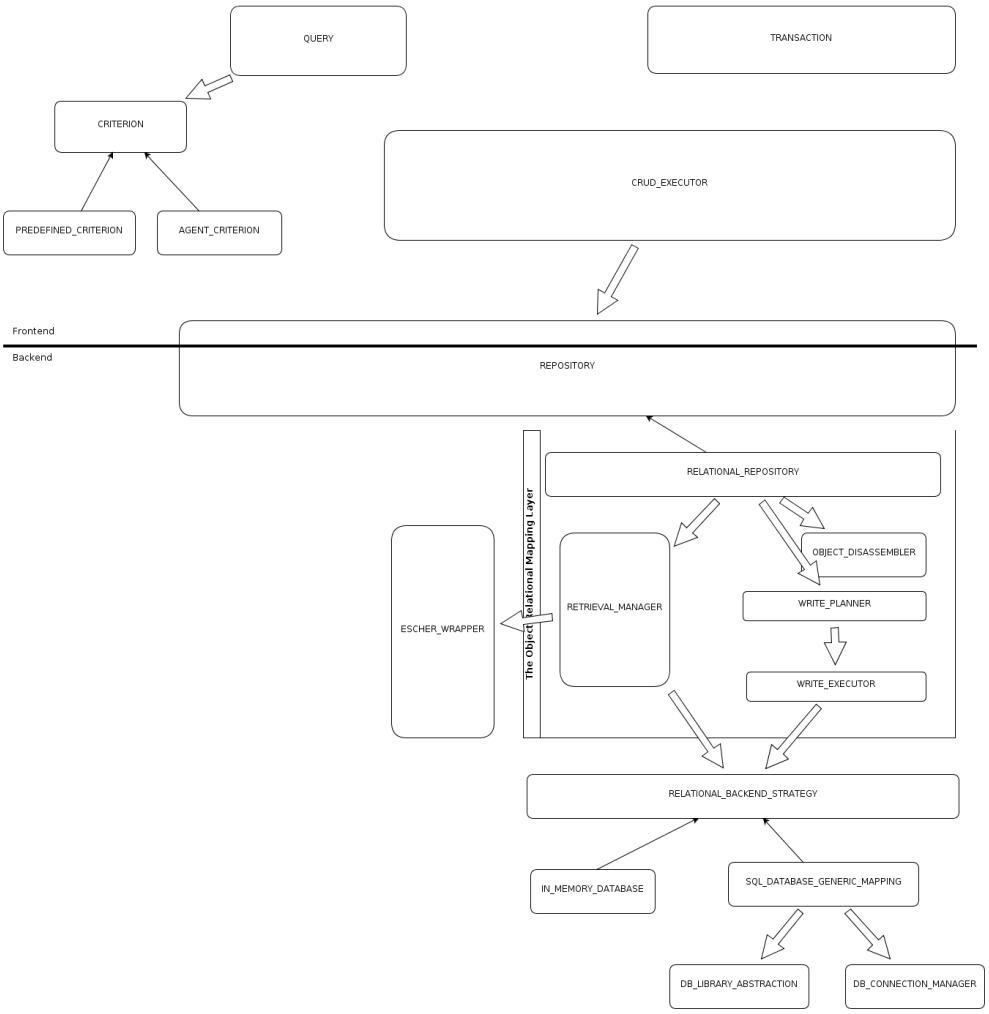
Transactions

Although not directly visible, transactions play an important role in the backend. Every operation internally runs in a backend, and almost every part in the backend is aware of transactions. For example, the two important data structures described above have to provide some kind of rollback mechanism, and ideally all ACID properties as well.

Another important task of transactions is error propagation within the backend. If for example an SQL statement fails because of some integrity constraint violation, then the database wrapper can set the error field in the current transaction instance and raise an exception. As the exception propagates upwards, every layer in the backend can do the appropriate steps to bring the library back in a consistent state, using the transaction with the error inside to decide on actions.

Class diagram

To visualize the whole structure, there is a class diagram that shows the most important classes and concepts. You can also see two implementations of the backend abstraction there.



3.2 Object-relational Mapping

The object-relational mapping layer lies between the REPOSITORY and the BACKEND_STRATEGY layer. It mainly consists of four main classes doing the actual work, and a set of helper classes that represent an object graph.

The object graph representation classes are all in folder "framework/object_graph_representation". Although their main purpose is to represent an object graph, they are also used to describe a write operation (the BACKEND_STRATEGY actually takes such objects as an argument) These are the most important ones:

- BASIC_ATTRIBUTE_PART represents an object of a basic type

- COLLECTION_PART represents a collection, e.g. an instance of SPECIAL.
- SINGLE_OBJECT_PART: represets an Eiffel object that is neither a basic type nor a collection.

All these classes inherit from a deferred class OBJECT_GRAPH_PART. They have a builtin iteration cursor, and they share the concept of a dependency. If an object graph part X is dependent on another part Y, then it means for example that Y has to be inserted first, because X needs its primary key as a foreign key in the database.

The four classes listed here are the ones that do the actual work:

- The OBJECT_DISASSEMBLER is responsible to create the explicit data structure for an object graph.
- The WRITE_PLANNER is responsible to generate a total order on all write operations, taking care of the dependency relations.
- The RETRIEVAL_MANAGER builds objects from the parts that it gets from the backend, and takes care that all referenced objects of a retrieved object get loaded as well.
- The COLLECTION_HANDLER, or rather its descendants, add collection handling support to the basic ORM layer. You need at least one handler for SPECIAL, but you can add handlers for other collection as well.

The object writing part is a bit more complex than the reading part, because of the dependency issue.

Add a little visualization of the different parts

3.2.1 Collection handling

You can extend the ORM algorithm to include collections. A collection is usually mapped differently from a normal object in the backend, e.g. through a M:N-relation table. By default you need at least one handler for SPECIAL, because of its peculiarity that it doesn't have a fixed amount of fields. But you can include any other collection, e.g. a LIST or an ARRAY.

There are two types of collections that you can create within a handler. The RELATIONAL_COLLECTION is intended for a case when you have a typical database layout, with tables for a specific class and relations stored

either with in the referenced object table (1:N-Relations) or inside their own table (M:N-Relations). The OBJECT_COLLECTION is intended for a scenario where you can store collections in a separate table, having their own primary key, and with the collection owner using this key as a foreign key.

Note that the choice of the collection has an effect in the object-relational mapping layer already:

- An OBJECT_COLLECTION is handled like a SINGLE_OBJECT_PART: The owner of the collection object depends on the collection, and the collection depends on all items that it references.
- A RELATIONAL_COLLECTION in an M:N mapping mode depends on both the collection owner and all items that it references, but the owner does not depend on its collection. This comes from the fact that you need both a foreign key of the owner and the collection items to insert a row in a M:N-relation table
- A RELATIONAL_COLLECTION in an 1:N mapping mode actually isn't forwarded to the backend at all. Instead, for each collection item there is a dependency added to the collection owner. Again, this comes from the normal practice in database layouts for 1:N relations.

It depends on the database layout of the backend which collection part you need to create.

3.2.2 Object graph settings

First, let's define the object graph more exactly, using graph theory. A vertex in the graph corresponds to an object, and a reference is a directed edge.

The (global) object graph is the web of objects and references as it is currently in main memory.

An object Y can be "reached" from another object X if there is a path between X and Y, i.e. Y is in the transitive closure of X.

The object graph of an object X is an induced subgraph of the global object graph which contains all vertices that can be reached from X.

The level of an object Y in the object graph of X is the length of the shortest path from X to Y.

Using these definitions we can now describe how ABEL handles object graphs, and how you can tweak the default settings to increase performance.

Every operation in ABEL has its own depth parameter (defined in OBJECT_GRAPH_SETTINGS), which has the following effect: Each operation will only handle the objects when the following condition holds: $level(object) < depth$

Now, let's put this in a context: You already know that the insert and retrieve features handle the complete object graph of an object. In fact, the depth for both functions is infinity by default.

On the other hand, the update or delete operations only handle first object they get, and don't care about the object graph. Their depth is defined as exactly 1, which means that only an object with a level of 0 satisfies the condition above. The only object with level 0 is in fact the root object of the object graph.

To fully understand the behaviour of ABEL, we also have to look at what happens when the algorithm reaches the "last" object, i.e. when the condition $level + 1 = depth$ holds. In that case the object with all basic attributes gets inserted/updated, but references only get written if the referenced object is already persistent. If it isn't persistent, then in a later retrieval operation the reference will be Void.

You can change the depth of the individual operations in REPOSITORY.default_object_graph. Please keep in mind that this is a dangerous operation, as a not fully retrieved or inserted object will contain Void references even in a void-safe environment, and it's also possible that they violate the invariant.

Apart from the depth, there are some other settings as well, i.e. what ABEL should do if it finds an already persistent object along the object graph of a new object to insert, or vice versa.

3.3 Backend abstraction

ABEL provides some powerful abstractions to be able to support many different storage engines. The three main levels of abstraction are the REPOSITORY class, the BACKEND_STRATEGY and the database wrapper classes.

3.3.1 REPOSITORY

The deferred class REPOSITORY is the highest level of abstraction. It deals with raw Eiffel objects and always deals with the complete object graph of such an object. It provides a good interface for persistence mechanism that provide a similarly high level of abstraction, like for example db4o .

reference db4o

At the moment, only the RELATIONAL_REPOSITORY implements REPOSITORY. The RELATIONAL_REPOSITORY uses the object-relational mapping layer and uses a generic BACKEND_STRATEGY to perform the operations at a lower level.

3.3.2 BACKEND_STRATEGY

The second important level of abstraction is the deferred class BACKEND_STRATEGY. This layer deals with one object graph part at once, either a single object or a collection. It is responsible to map these to the actual persistence mechanism, which is usually a specific layout in a database. Its use however is not restricted to relational databases. The IN_MEMORY_DATABASE for example implements this interface to provide a fake storage engine useful for testing, and it is planned to wrap the serialization libraries using this abstraction.

3.3.3 Database wrapper

The last layer of abstraction is a set of wrappers to a database. It consists of three deferred classes:

- The SQL_DATABASE_ABSTRACTION represents a database. The main function is to acquire or release a SQL_CONNECTION_ABSTRACTION.
- The SQL_CONNECTION_ABSTRACTION represents a single connection. Its main responsibility is to forward SQL statements to the database and to represent the result in an iteration cursor of SQL_ROW_ABSTRACTIONs Another important task is to map database errors to ABEL ERROR instances.
- The SQL_ROW_ABSTRACTION represents a single row in the result of an SQL query.

The wrapper is very useful if you want to easily swap e.g. from a MySQL database to SQLite

However, keep in mind that its abstraction is not perfect. For example, the wrapper doesn't care about the different SQL variations, as it just forwards the statements to the database.

To overcome this problem, you can put all SQL statements in your implementation of BACKEND_STRATEGY into a separate class and generally stick to standard SQL as much as possible.

3.4 Extensions

Due to its very flexible abstraction mechanism, you can easily extend ABEL with features like transaction management or ESCHER integration. The pattern how to do this is quite simple: You can implement a BACKEND_STRATEGY which uses another instance of BACKEND_STRATEGY, but does some processing on the intermediate result. That way you can add:

reference as cher

- Filter support for some non-persistent attributes by removing them from the OBJECT_GRAPH_PART during a write, and adding a default value during retrieval.
- ESCHER support by checking on the version attribute during a retrieval, and calling the conversion function if necessary.
- Client-side transaction management by using a multiversion concurrency control mechanism and delaying write operations until you can definitely commit.
- Caching of objects
- An instance that does correctness checks, e.g. by routing the calls to two different backends and comparing if the results are the same.
- Anything else you can imagine...

The really nice thing is that you can do that without adding complexity to the core of ABEL, and for all possible implementations of BACKEND_STRATEGY at once.

3.5 Database adaption

The BACKEND_STRATEGY interface allows to adapt ABEL to a lot of database layout. Shipped with the library is a backend that uses a generic database layout which is suitable for all kinds of objects, which is explained in the next section. But you can also adapt ABEL to your very own private database layout, as described in section 3.5.2

3.5.1 The generic layout backend

The layout in the database is based upon metadata of the class. It is very flexible and allows for any type of objects to be inserted:

In fact, this is a simplified view. The real model uses another relation between value and class to determine the runtime type of a value, which is required in some special cases.

Add Entity-Relationship model

The generic layout backend, located in "backends/generic_database_layout", maps Eiffel objects to this layout.

It is split into three main classes:

- The METADATA_TABLES_MANAGER is responsible to read and write tables "ps_class" and "ps_attribute".
- The GENERIC_LAYOUT_SQL_BACKEND actually implements BACKEND_STRATEGY and is responsible to write and read the ps_value table
- The GENERIC_LAYOUT_SQL_STRINGS collects all SQL statements needed by the other classes. Its descendants adapt the statements to a specific database if there is an incompatibility.

The functionality of the metadata table manager is quite easy: It just caches both metadata tables in memory and provides features to get the primary key of an attribute or a class. If the class is not present in the database, then it will insert it and return the new primary key.

A write operation is now possible, as all required information is available: The attribute value in the SINGLE_OBJECT_PART, the attribute foreign key from the METADATA_TABLES_MANAGER, and the object primary key either stored in the KEY_POID_TABLE or generated during an insert.

The retrieval operations is similar. You can get all attribute primary keys of a specific class from the METADATA_TABLES_MANAGER and then execute an SQL query which returns all values whose attribute foreign key is in the set of primary keys retrieved before, sorted by the object primary key.

3.5.2 Adaption to a custom database layout

Adapting ABEL to a specific database layout needs two steps:

- Implement a BACKEND_STRATEGY for your layout
- Implement COLLECTION_HANDLERS for all collections that need to be mapped relationally

Let's consider a very simple example. You only have two classes PERSON and ITEM:

```
1 class PERSON

  feature
4   name:STRING

    items_owned: LINKED_LIST [ITEM]
7 end


10 class ITEM

   feature
13   value:INTEGER

   end
```

Listing 3.1: Application classes

In the database, there is table Person with columns primary_key and name, and table Item with columns primary_key, item, and a foreign key to Person (as it is usual in an 1:N relation)

In this setup, you only need a collection handler for LINKED_LIST. The collection is a RELATIONAL_COLLECTION and it is always 1:N mapped in this simple setup.

The implementation of BACKEND_STRATEGY is quite straightforward as well. You just have to distinguish between PERSON and ITEM objects and insert them in the corresponding table.

Please remember that the object-rlational mapping layer adds an attribute with name "items_owned" to the ITEM object, which is the default behaviour for 1:N relations. This especially menans that you don't need to implement the relational collection write operations.

During a retrieval operation, you similarly have to select your values from the correct table. Note that you need the retrieve_relational_collection here.

Chapter 4

Conclusions

4.1 Conclusions

In this thesis, we have developed a software library to access different persitence mechanism. The library features a simple yet powerful programming interface, which is completely agnostic of the actual backend.

Behind the API we developed a very flexible framework to adapt ABEL to a lot of existing storage engines. The framework includes a reusable object-relational mapping layer, a database library wrapper, and some interfaces for extension and customization.

Based on this framework we have developed an in-memory backend which is useful for testing along with a test suite that is independent of the actual backend.

Furthermore, we have developed a backend that uses a generic database layout for storing objects, working both for MySQL and SQLite databases.

4.2 Current limitations

- Due to a limitation in INTERNAL, inheritance is not properly supported. This especially means that a query of an object of class X will not return descendants of X.
- If you adapt ABEL to a custom layout, it can only handle object types that have corresponding tables in the database.
- If a custom database layout isn't well designed, i.e. if there are redundancies, you might get into trouble when adapting ABEL to it.
- Some basic types are not fully supported:

  - Reals have a rounding error
  - STRING_32 is converted to STRING_8, which distorts them sometimes.
  - CHARACTER is converted to INTEGER for storage. This is usually fine, but in custom database layouts this results in a type mismatch.

- The generic database layout backend doesn't support SPECIAL yet (or collections in general).
- The library completely lacks any performance optimization.
- The retrieval operation in the object-relational mapping doesn't support the depth parameter.
- The error representation is not complete.

4.3 Future work

Ordering At the moment, a query result has no defined order. A mechanism to enforce an order in the result set might be a useful addition to the library user.

Update Query With the current API you can only do an update if the object has been retrieved or inserted before. Another way to do update operations could use a QUERY instead, but you also need some new mechanism to be able to say which attributes should be updated with a new value.

Performance Currently there is no optimization in ABEL, and there is a lot that can be done in this area:

- Move the filtering of objects according to the CRITERIA in a query to the database, by compiling criteria to SQL.
- Add support for lazy loading in the generic database layout implementation by using SQL cursors instead of normal select statements.
- Use prepared statements and maybe even stored procedures instead of normal SQL statements.
- Optimize ABEL by trying to reduce network round trip times to a minimum.

- Finding and fixing performance bottlenecks in the code with the help of a profiler

Adaptor Framework Adaption to a specific layout is a tedious task at the moment, as everything has to be hardcoded. It would be much easier to just be able to define a mapping from classes to tables, attribute names to column names, and the framework takes care of the rest. As an extension this mapping could be defined as an XML file, which lifts the burden of the user to hack on ABEL itself.

Backends Extend ABEL to support more backends, e.g.

- A serialization library
- An object database, e.g. db4o
- EiffelStore
- A NoSQL database like CouchDB

Inheritance ABEL needs proper inheritance support as soon as the required features in INTERNAL get implemented.

Transaction management Some backends, e.g. the in-memory backend or the serialization library, don't support transactions. Therefore it would be nice to implement local transaction management for these libraries by using e.g. multiversion concurrency control at the BACKEND_STRATEGY level.

ESCHER At the moment, the behaviour of ABEL is undefined in case of a version mismatch between stored objects and their runtime type. An integration of ESCHER into the ABEL framework would help detecting a version mismatch and, if possible, correct it immediately.

Bibliography