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 an approach to wrap every existing persistence library under a simple and yet powerful API. The programming interface of ABEL is completely transparent to the actual storage mechanism, and it supports the CRUD operations, transactions, and some advanced features like result filtering based on some criteria.

ABEL has a flexible framework that allows to adapt the API to basically any existing persistence solution. The framework has a reusable object-relational mapper at its core and is open for extensions and customization.

Contents

Chapter 1

Introduction

1.1 Introduction

The Eiffel language [2] [8] has a lot of different persistence libraries, and every solution has its own advantages and drawbacks. A database such as MySQL [3] for example is quite fast, but it comes at the expense of the object-relational impedance mismatch [7]. 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 for example the database from MySQL to Oracle [4] 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 interfaces for backend abstraction.

Chapter 2

API tutorial

2.1 Getting started

2.1.1 The setup

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

create
  make

feature {NONE} -- Initialization

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

feature -- Basic operations

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

```
    end

feature -- Access

    first_name: STRING
      -- First name of person
    last_name: STRING
      -- Last name of person

    age: INTEGER
      -- The person's age

end
```

Listing 2.1: *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*.

2.1.2 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

create
  make

feature {NONE} -- Initialization
```

```
  make
    -- 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 2.5, "Advanced Initialization."

2.2 Basic operations

2.2.1 Query

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
    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. You can read more 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
```

```
      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 2.3.3.

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, 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 2.3.3, 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.

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 *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 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 of criteria: predefined and agent criteria.

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 2.3). 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.

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 of 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 with
    -- 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 = ">"
27
    greater_equal: STRING = ">="

30  less: STRING = "<"

    less_equal: STRING = "<="
33
    like_string: STRING = "like"

36 end
```

Listing 2.3: *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
    Result := factory[[agent age_less_than (?, 5)]]
  end

age_less_than (person: PERSON; age: INTEGER): BOOLEAN
  -- An 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 ]]
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**.

2.3.3 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 2.2.3.

```
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.

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 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
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:= query.attribute_index ("last_name")

  from
    executor.execute_tuple_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 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.

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
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_tuple_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 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
  CHILD

create
  make
```

```
  feature {NONE} -- Initialization

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

  feature -- Access

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

  feature -- Status report

    first_name: STRING
      -- First name of child
    last_name: STRING
      -- Last name of child

    age: INTEGER
      -- The child's age

  feature -- Parents

    mother: detachable CHILD
      -- 'Current's mother

    father: detachable CHILD
      -- 'Current's father

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

```
    set_father (a_father: CHILD)
51    -- Set the father
      do
        father:= a_father
54    end
  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 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.

2.4.1 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
require
  child_persistent: executor.is_persistent (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 3.2.2 in the technical documentation, where the whole concept of an object graphs and its depth is described more detailed.

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 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 [5] database:

```
class
  REPOSITORY_FACTORY

feature -- Connection details

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

  db_name:STRING = "tutorial"
  db_host:STRING = "127.0.0.1"
  db_port:INTEGER = 3306

  sqlite_filename: STRING = "tutorial.db"

feature -- Factory methods

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

  create_sqlite_repository: PS_RELATIONAL_REPOSITORY
    -- Create an SQLite repository
    local
      database: PS_SQLITE_DATABASE
      sqlite_strings: PS_SQLITE_STRINGS
      backend: PS_GENERIC_LAYOUT_SQL_BACKEND
    do
      create database.make (sqlite_filename)
      create sqlite_strings
      create backend.make (database, strings)
      create Result.make (backend)
    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.

2.6 Transaction handling

Every CRUD operation in ABEL is executed within a transaction. Transactions are created and commited 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
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
    print ("Commit has not been successful")
  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*.

2.6.1 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.

2.7 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
  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_CRUD_EXECUTOR
    -- 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
```

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 previous 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 the type of the object to be retrieved (through its generic parameter).
- *TRANSACTION*: Represents a transaction, and is responsible to internally propagate errors.
- *CRITERION*: Its descendants provide a filtering function for retrieved objects, and it has features 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 which the backend needs.

The class diagram provides an overview over the frontend. Note that this diagram only shows the most important classes and their relations.

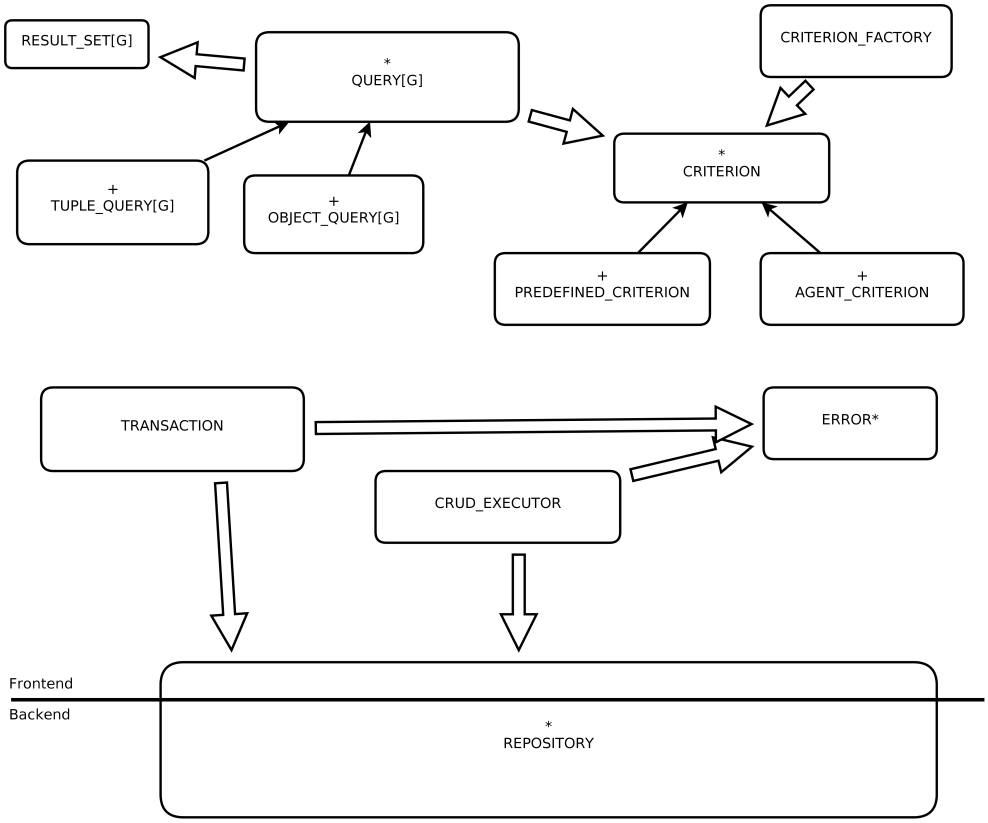

Figure 3.1: *The main frontend classes and their relations.*

3.1.2 Backend

The frontend needs a repository which is specific to a persistence library, and the backend part provides a framework to implement these repositories (in cluster *framework*).

There are also some predefined repositories inside the backend (cluster *backends*), like the *IN_MEMORY_REPOSITORY*.

The framework layers

The framework is built of several layers, with each layer being 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, as 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, and also to build an object graph from the pieces during retrieval. This layer is described more precisely in section 3.2.

On the next level there is the *BACKEND_STRATEGY* layer. Its task is to map the object graph pieces to a specific storage engine, e.g. a database with some table layout.

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

Important data structures

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

Another important data structure 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 inside a transaction, 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 its actions.

Class diagram

To visualize the whole structure, there is a class diagram that shows the most important classes and concepts of the backend.

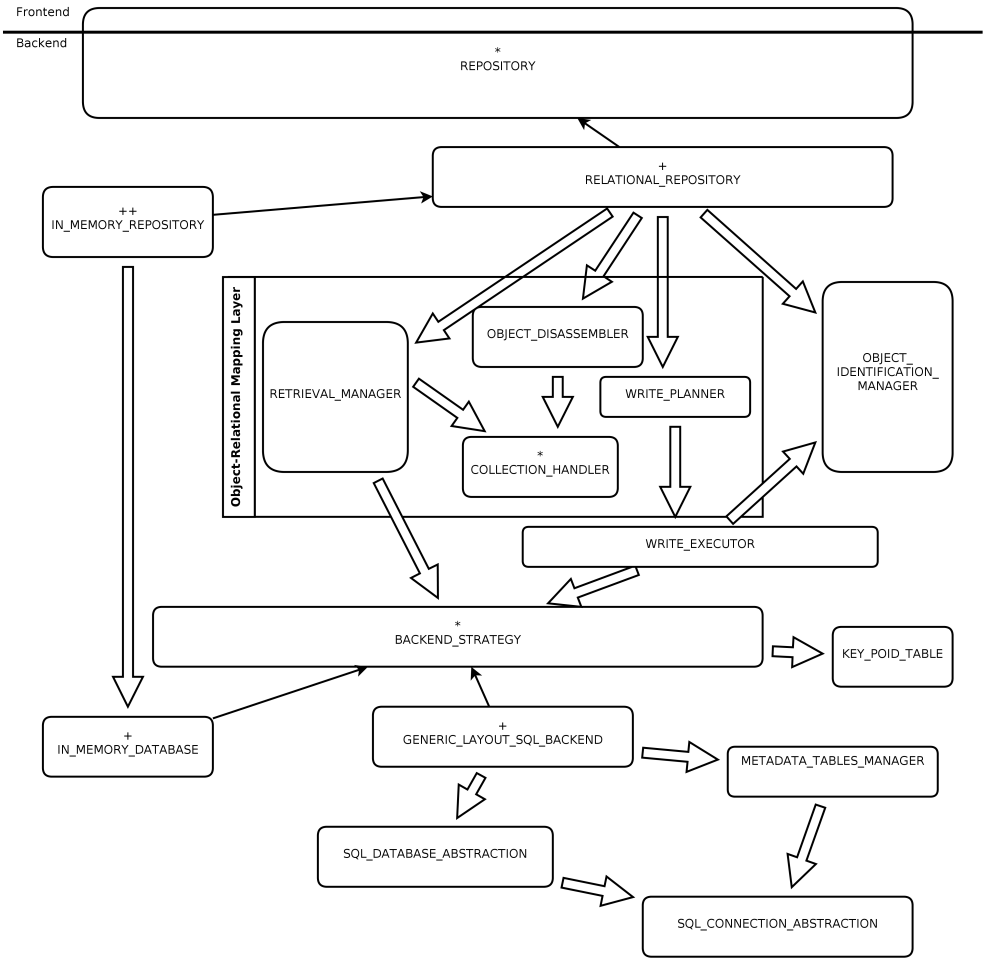

Figure 3.2: *The main backend classes and their relations.*

3.2 Object-relational Mapping

The object-relational mapping layer (abbreviated as ORM layer) lies between the `REPOSITORY` and the `BACKEND_STRATEGY`. It consists of four main classes doing the actual work, and a set of helper classes to represent an object graph.

All helper classes are in *framework/object_graph_representation*. Besides representation of an object graph, they are also used to describe a write operation in the *BACKEND_STRATEGY*. The most important ones are:

- *BASIC_ATTRIBUTE_PART* represents an object of a basic type
- *COLLECTION_PART* represents a collection, for example an instance of *SPECIAL*
- *SINGLE_OBJECT_PART*: represets an Eiffel object that is neither a basic type nor a collection

All helper classes inherit from *OBJECT_GRAPH_PART*. They have a built-in 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 Y's 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 object graph representation.
- The *WRITE_PLANNER* is responsible to generate a total order on all write operations, taking care of the dependency issues.
- The *RETRIEVAL_MANAGER* builds objects from the parts that it gets from the backend, and takes care that the complete object graph gets loaded.
- 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 collections as well.

The graph visualizes the process and shows the intermediate representation of data in the object-relational mapping layer. You can see there that the object writing part is a bit more complex than the reading part, because of the dependency issue.

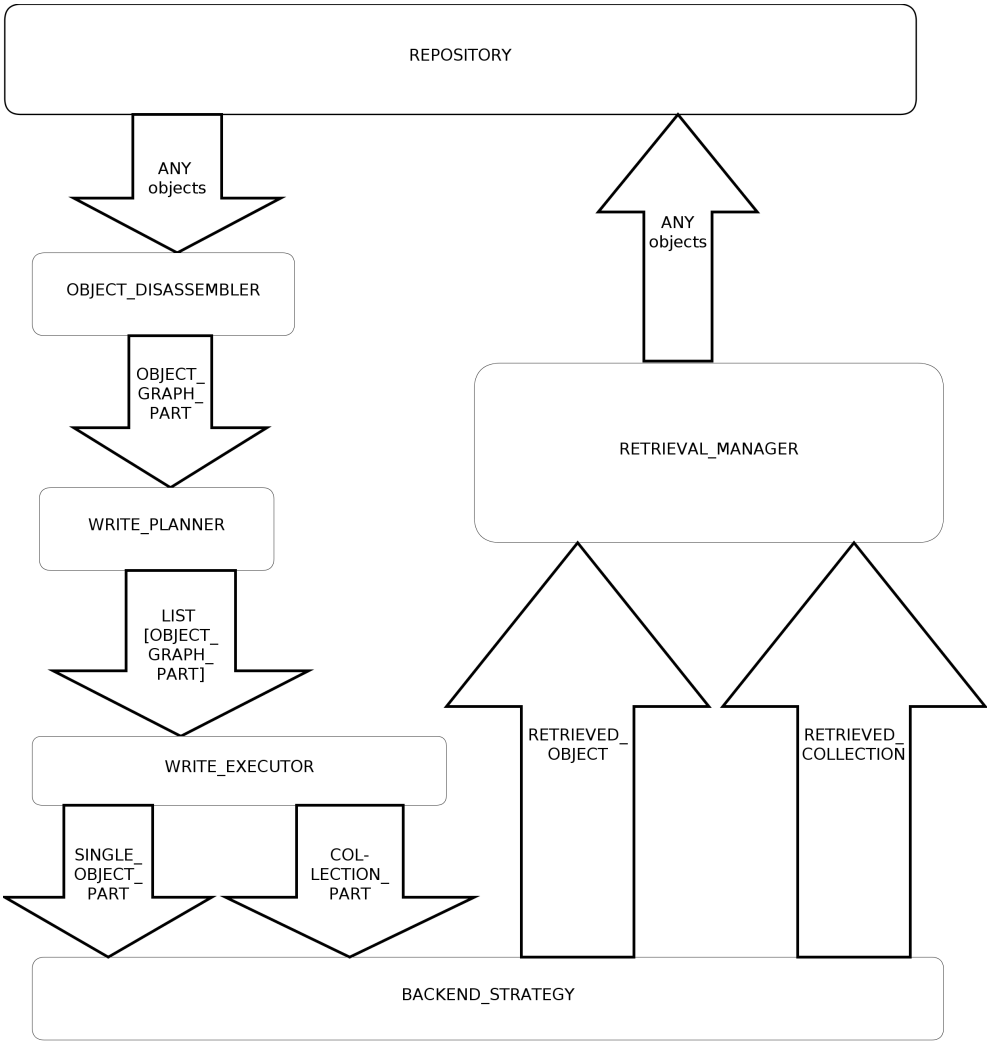

Figure 3.3: *The different intermediate representations of data.*

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 an M:N-relation table. 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, like a `LIST` or an `ARRAY`.

There are two types of collections you can create within a handler. The `RELATIONAL_COLLECTION` is intended for a case when you have a typical database layout, with tables for every specific class and relations stored either within the table of the referenced object (1:N-Relations) or inside

their own table (M:N-Relations).

The `OBJECT_COLLECTION` is intended for a scenario where you store collections in a separate table, having their own primary key, and with the collection owner using this key as a foreign key.

The following diagrams shows an example entity-relationship model for each type of collection:

Figure 3.4: *An ER-model where an OBJECT_COLLECTION can be used.*

Figure 3.5: *An ER-model where a RELATIONAL_COLLECTION with M:N mapping can be used.*

Figure 3.6: *An ER-model where a RELATIONAL_COLLECTION with 1:N mapping can be used.*

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

- `OBJECT_COLLECTION`s are 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 item to insert a single row in an M:N-relation table.
- A `RELATIONAL_COLLECTION` in a 1:N mapping mode actually isn't forwarded to the backend at all. Instead, each item in the collection

gets a new dependency to the collection owner. Again, this comes from the normal practice in database layouts for 1:N relations.

If you use one of the predefined backends, you usually don't have to care about collection handlers. They get important however if you want to adapt ABEL to a custom database layout, as you can see in section 3.5.2.

Please note that the framework itself does not provide any collection handler, and inserting a `SPECIAL` object without setting an appropriate handler will result in a runtime crash. However, there is a handler for `SPECIAL` shipped with the predefined backends, which is used for example by the `IN_MEMORY_REPOSITORY`.

3.2.2 Object graph settings

First, let's define the object graph more exactly, using graph theory. An object corresponds to a vertex in the graph, 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 the induced subgraph of the global object graph that 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, which is defined in `OBJECT_GRAPH_SETTINGS`. An operation will only handle an object 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 the 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 an 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 parameter of the individual operations in `REPOSITORY.default_object_graph`. Please keep in mind that this is a dangerous operation, because a partially loaded object will contain Void references even in a void-safe environment and may also violate its 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

The framework provides some very flexible interfaces to be able to support many different storage engines. The three main levels of abstraction are the `REPOSITORY`, the `BACKEND_STRATEGY` and the database wrapper classes.

3.3.1 REPOSITORY

The deferred class `REPOSITORY` provides the highest level of abstraction, as it deals with raw Eiffel objects including their complete object graph. It provides a good interface to wrap a persistence mechanism that provides a similarly high level of abstraction, like for example db4o [1].

The `RELATIONAL_REPOSITORY` is the main implementation of this interface. It uses the ORM layer and a `BACKEND_STRATEGY` and is therefore the default repository for persistence libraries which are wrapped through `BACKEND_STRATEGY`.

3.3.2 BACKEND_STRATEGY

Another important interface is the deferred class `BACKEND_STRATEGY`. This layer only deals with one object graph part at once, either a single object or a collection. It is responsible to map them to the actual persistence mechanism which is usually a specific layout in a database.

Its use however is not restricted to relational databases. The predefined IN_MEMORY_DATABASE backend 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* represents a database. Its main task is to acquire or release a *SQL_CONNECTION*.
- The *SQL_CONNECTION* represents a single connection. It has to forward SQL statements to the database and represent the result in an iteration cursor of *SQL_ROW*s. Another important task is to map database specific error messages to ABEL *ERROR* instances.
- The *SQL_ROW* 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 the 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 flexible abstraction mechanism, you can easily extend ABEL with features like client-side transaction management or ESCHER [9] integration.

The general pattern on how to do this is quite simple: You can implement the extension in a class that inherits from *BACKEND_STRATEGY* and forwards all calls to another *BACKEND_STRATEGY* that does the actual storage. Then the extension can do some processing on the intermediate result. That way you can add:

- 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 by using an *IN_MEMORY_DATABASE* alongside the actual backend.
- 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 nice thing is that you can add such extensions without adding much complexity to the core of ABEL, and it works for all possible implementations of *BACKEND_STRATEGY* at once.

3.5 Database adaption

The *BACKEND_STRATEGY* interface allows to adapt the framework to many database layouts. Shipped with the library is a backend that uses a generic database layout which can handle every type of object. It is explained in the next section.

But you can also adapt ABEL to your very own private database layout. An example on how to do this is shown in section 3.5.2.

3.5.1 The generic layout backend

The database layout is based upon metadata of the class. It is very flexible and allows for any type of objects to be inserted. The layout is a modified version from the suggestion in Scott W. Ambler's article "Mapping Objects to Relational Databases" [6].

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

The backend located in *backends/generic_database_layout* maps Eiffel objects to this database layout.

It is split into three classes:

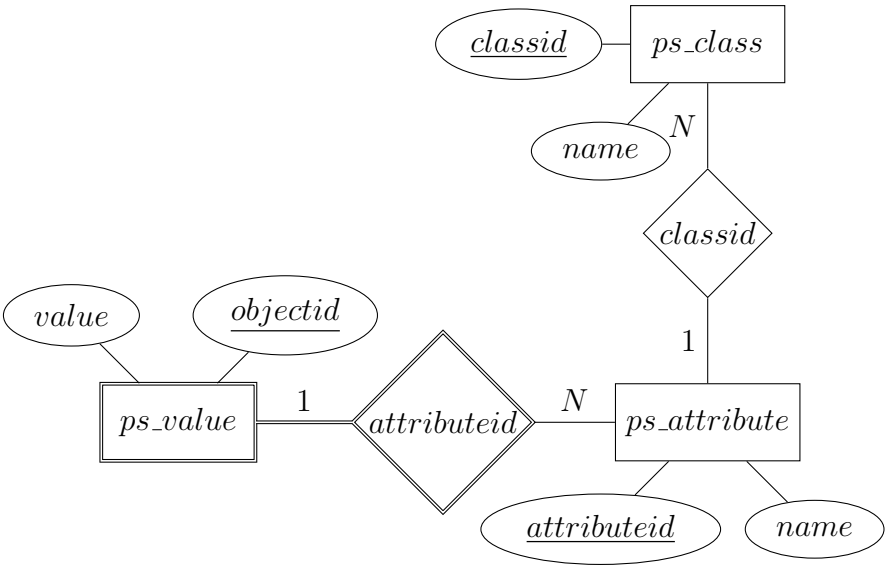

Figure 3.7: *The ER-Model of the generic database layout.*

- The `METADATA_TABLES_MANAGER` is responsible to read and write tables `ps_class` and `ps_attribute`.
- The `GENERIC_LAYOUT_SQL_BACKEND` is responsible to write and read the `ps_value` table. It is an implementation of `BACKEND_STRATEGY`.
- The `GENERIC_LAYOUT_SQL_STRINGS` collects all SQL statements. 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 table `ps_class` and `ps_attribute` 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.

Using the table manager, the `GENERIC_LAYOUT_SQL_BACKEND` has all information to perform a write operation: The attribute value inside the `SINGLE_OBJECT_PART`, the attribute foreign key which is provided by the `METADATA_TABLES_MANAGER`, and the object primary key which is either stored in the `KEY_POID_TABLE` or generated during an insert.

The retrieval operation is similar. First, the backend gets all attribute primary keys of a specific class from the table manager, and then it executes an SQL query to retrieve all values whose attribute foreign keys match the ones retrieved before. The backend does also sort the result by

the object primary key, such that attributes of the same object are grouped together.

3.5.2 Adaption to a custom database layout

Adapting ABEL to a custom database layout needs two steps:

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

Let's consider a very simple example with only two classes:

```
class
  PERSON

feature
  name:STRING
    -- Name of 'Current'
  items_owned: LINKED_LIST [ITEM]
    -- Items owned by 'Current'
end


class
  ITEM

feature
  value:INTEGER
    -- The value of 'Current'
end
```

Listing 3.1: *Example 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 *owner* to table *Person*.

In this setup, you only need a collection handler for *LINKED_LIST*. All *LINKED_LIST* instances are relationally mapped in 1:M mode, therefore, the implementation of the (only) collection handler is very simple:

```
class
  LINKED_LIST_HANDLER
inherit
```

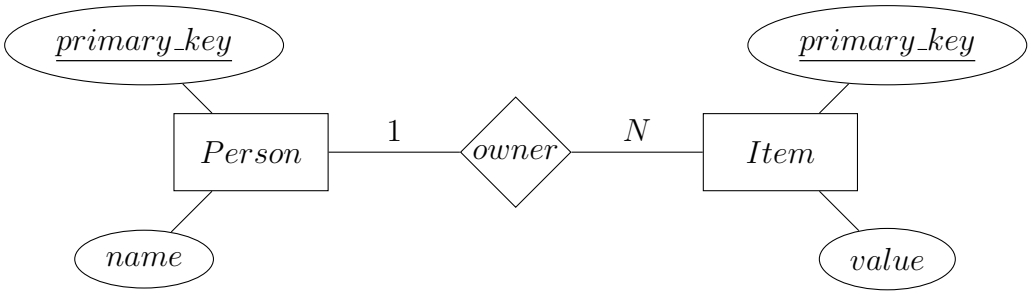

Figure 3.8: *The example ER-Model.*

```
  PS_COLLECTION_HANDLER [LINKED_LIST [detachable ANY]]
feature

  is_relationally_mapped (collection, owner_type:
     PS_TYPE_METADATA):BOOLEAN
    -- Is 'collection' mapped as a 1:N or M:N - Relation?
    do
      Result:= True
    end

  is_mapped_as_1_to_N (collection, owner_type:
     PS_TYPE_METADATA):BOOLEAN
    -- Is 'collection' mapped as a 1:N - Relation?
    do
      Result:= True
    end

  build_relational_collection (collection_type:
     PS_TYPE_METADATA; objects: LIST[detachable ANY]):
     LINKED_LIST[detachable ANY]
    -- Build a new LINKED_LIST
    do
      create Result.make
      Result.append (objects)
    end

end
```

Listing 3.2: *The collection handler for LINKED_LIST*

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 means that you don't need to implement the write operations for relational collections.

The following code listing shows the insert feature in pseudocode:

```
class
  MY_SIMPLE_BACKEND
inherit
  PS_BACKEND_STRATEGY
feature

  insert (object:PS_SINGLE_OBJECT_PART; tx:PS_TRANSACTION)
    -- Inserts the object into the database
    do
      if object is a PERSON object then
        database.execute_sql ("INSERT INTO person (name)
           VALUES " + object.attribute_value ("name"))
        key_mapper.add_entry (object, database.execute_sql ("
           Get autoincremented primary key of Person")
      else
        -- The ORM layer added 'items_owned' to ITEM
        foreign_key:= key_mapper.primary_key_of (object.
           get_value ("items_owned"))
        database.execute_sql ("INSERT INTO item (value, owner
           ) VALUES " + object.attribute_value ("value") +
           foreign_key)
        key_mapper.add_entry (object, database.execute_sql("
           Get autoincremented primary key of Item")
      end
    end

  key_mapper: PS_KEY_POID_TABLE
    -- Maps object identifiers to primary keys
end
```

Listing 3.3: *The collection handler for LINKED_LIST*

During a retrieval operation, you similarly have to select your values from the correct table. Please note that you have to implement the *retrieve_relational_collection* feature 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.

Below 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, and a test suite that is mostly 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` the framework is not able to get ancestors or descendants of a class. It therefore treats all classes as if they are flat, meaning that all inherited features are defined directly inside the class. That way ABEL can handle classes inside an inheritance hierarchy, but the drawback of this approach is that a query to objects of class `G` will not return descendants of `G`.

- If you adapt ABEL to a custom database layout, it can only handle object types that have corresponding tables in the database.

- If a custom database layout isn't well designed, e.g. if there are redundancies, you might get into trouble when trying to adapt ABEL to it.
- Some basic types are not fully supported:
  - *REAL* has a rounding error.
  - *STRING_32* is converted to *STRING_8*, which may distort them sometimes.
  - *CHARACTER* is converted to *INTEGER* for storage. This is usually fine, but in custom database layouts it 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 layer 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 useful.

Update Query With the current API you can only do an update if the object has been retrieved or inserted before. To select the object to be updated, you could also use a *QUERY* object, but additionally you need some 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:

- Compile *PREDEFINED_CRITERIA* to SQL in order to get smaller results from the database.
- 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 if you could just define a mapping from classes to tables and attribute names to column names, and the framework takes care of the rest. As an extension this mapping could be defined in an XML file.

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

- A serialization library
- An object database like 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 an extension that provides local transaction management by using multiversion concurrency control at the `BACKEND_STRATEGY` level.

ESCHER At the moment, there is an initial integration of ESCHER into the ABEL framework, but the functionality is limited to a simple version check and error reporting in case of a mismatch. This solution could be extended to support automatic conversion as well.

Bibliography

[1] db4o. http://db4o.com/.

[2] Eiffel ECMA-367 Standard. http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[3] MySQL. http://www.mysql.com/.

[4] Oracle. http://www.oracle.com/.

[5] SQLite. http://www.sqlite.org/.

[6] Scott W. Ambler. Mapping Objects to Relational Databases: O/R Mapping In Detail. http://www.agiledata.org/essays/mappingObjects.html.

[7] Scott W. Ambler. The Object-Relational Impedance Mismatch. http://www.agiledata.org/essays/impedanceMismatch.html.

[8] B. Meyer. *Touch of Class*. Springer, 2009.

[9] Marco Piccioni, Manuel Oriol, and Bertrand Meyer. Class schema evolution for persistent object-oriented software: Model, empirical study, and automated support. *IEEE Transactions on Software Engineering (to appear)*, 2013.