ABEL Technical Documentation

Roman Schmocker
Reviewed by: Marco Piccioni

July 23, 2012

Contents

1 Introduction **2**

2 Architecture overview **3**
2.1 Front-end 3
2.2 Back-end 4
2.2.1 The framework layers 4
2.2.2 Important data structures 5
2.2.3 Transactions 5
2.2.4 Class diagram 6

3 Object-relational Mapping **7**
3.1 Collection handling 8
3.2 Object graph settings 10

4 Backend abstraction **12**
4.1 REPOSITORY 12
4.2 BACKEND 12
4.3 Database wrapper 13

5 Extensions **14**

6 Database adaption **15**
6.1 The generic layout backend 15
6.2 Adaption to a custom database layout 16

Chapter 1

Introduction

ABEL (A Better Eiffelstore Library) aims at providing a unified, easy to use object-oriented interface to different kinds of persistence stores, trying to be as back-end-agnostic as possible.

For the basic CRUD (Create, Read, Update, Delete) operations of the API and transaction handling, see the ABEL tutorial.

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

At the moment the supported back-ends are an in-memory database, MySQL, and SQLite.

Chapter 2

Architecture overview

The ABEL library can be split into a front-end and a back-end part. The front-end 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 front-end can be drawn straight through the deferred class *REPOSITORY*.

2.1 Front-end

If you've read the previous part of the documentation, then you should be quite familiar now with the front-end. 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 front-end 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 front-end. Note that this diagram only shows the most important classes and their relations.

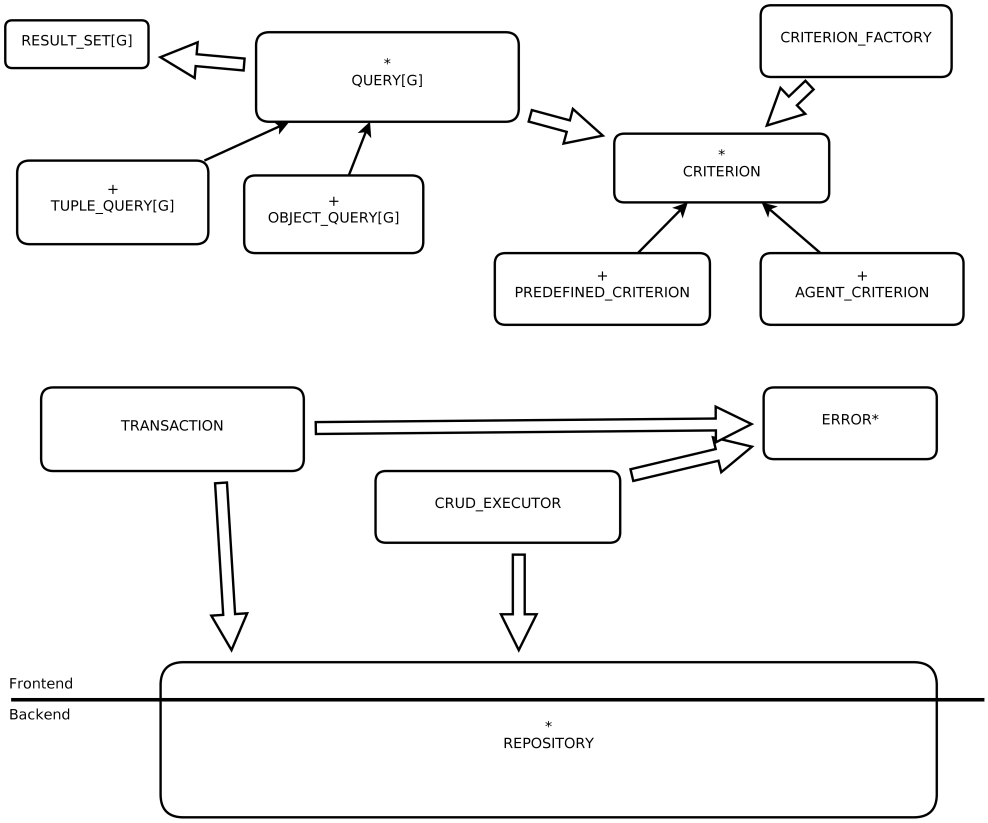

Figure 2.1: *The main front-end classes and their relations.*

2.2 Back-end

The front-end needs a repository which is specific to a persistence library, and the back-end part provides a framework to implement these repositories (in cluster `framework`).

There are also some predefined repositories inside the back-end (cluster `backends`), like the `IN_MEMORY_REPOSITORY`.

2.2.1 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.

On the next level there is the *BACKEND* 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.

2.2.2 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 **??**, "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.

2.2.3 Transactions

Although not directly visible, transactions play an important role in the back-end. Every operation internally runs inside a transaction, and almost every part in the back-end 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 back-end. 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 back-end 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.

2.2.4 Class diagram

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

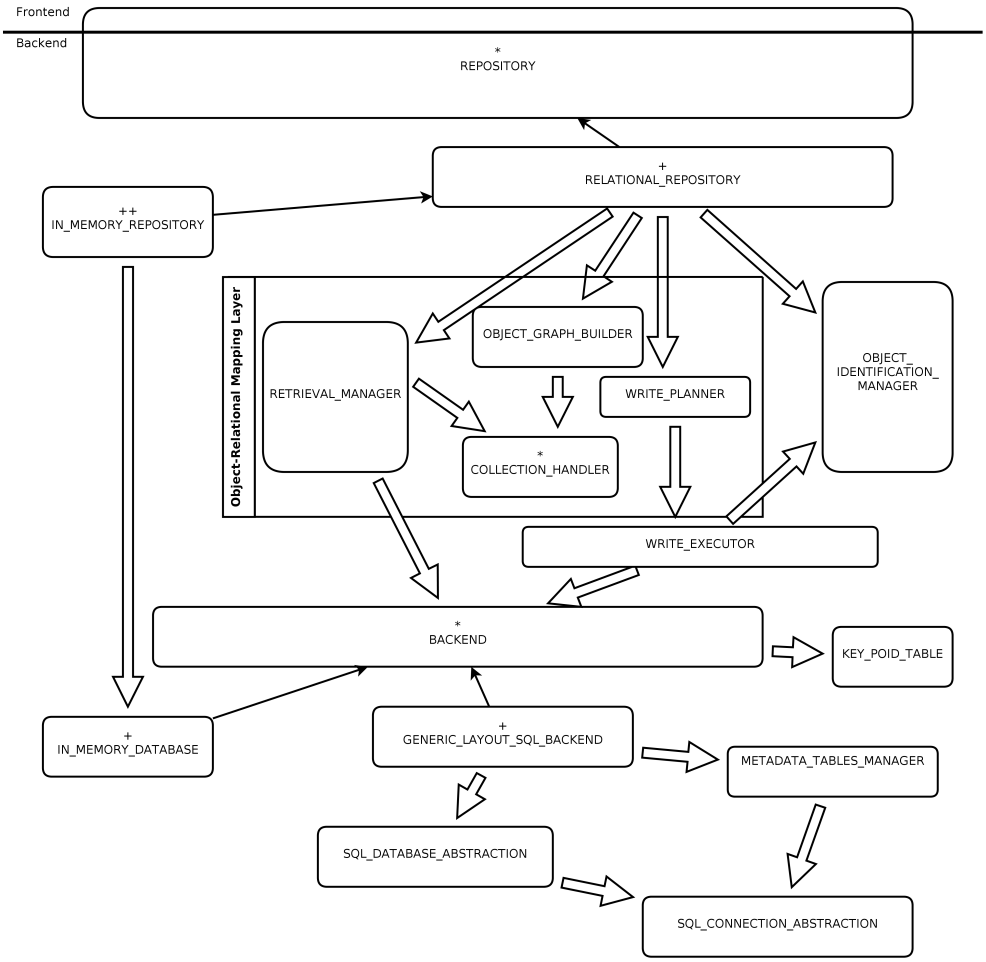

Figure 2.2: *The main back-end classes and their relations.*

Chapter 3

Object-relational Mapping

The object-relational mapping layer (abbreviated as ORM layer) lies between the *REPOSITORY* and the *BACKEND*. 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*. 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_GRAPH_BUILDER* 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.

3.1 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:

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

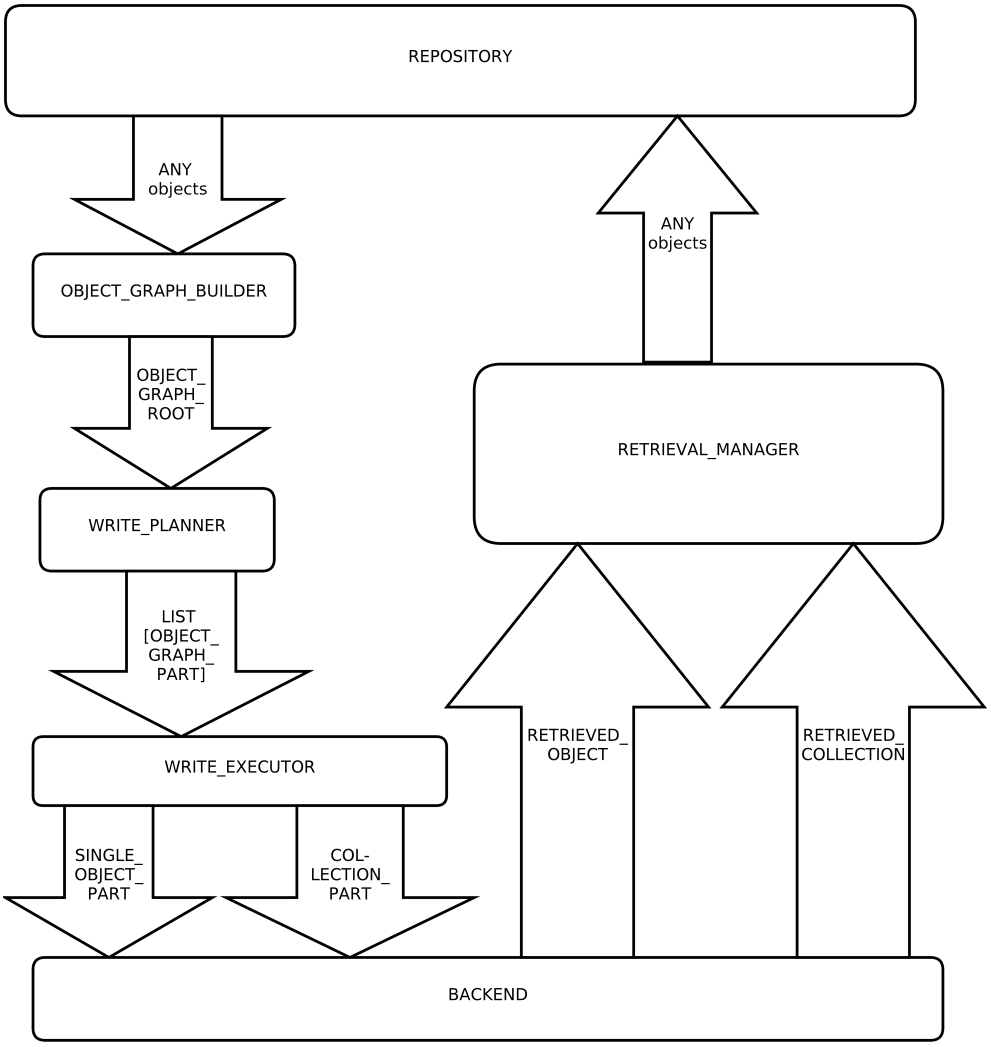

Figure 3.1: *The different intermediate representations of data.*

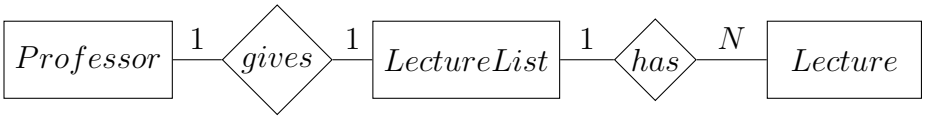

Figure 3.2: *An ER-model where an OBJECT_COLLECTION can be used.*

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

Figure 3.3: *An ER-model where a RELATIONAL_COLLECTION with M:N mapping can be used.*

Figure 3.4: *An ER-model where a RELATIONAL_COLLECTION with 1:N mapping can be used.*

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 6.2.

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 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.

Chapter 4

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* and the database wrapper classes.

4.1 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 [**?**].

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

4.2 BACKEND

Another important interface is the deferred class *BACKEND*. 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.

4.3 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* into a separate class, and generally stick to standard SQL as much as possible.

Chapter 5

Extensions

Due to its flexible abstraction mechanism, you can easily extend ABEL with features like client-side transaction management or ESCHER [**?**] integration.

The general pattern on how to do this is quite simple: You can implement the extension in a class that inherits from *BACKEND* and forwards all calls to another *BACKEND* 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* at once.

Chapter 6

Database adaption

The *BACKEND* 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 6.2.

6.1 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" [**?**].

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:

- 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*.
- The *GENERIC_LAYOUT_SQL_STRINGS* collects all SQL statements. Its descendants adapt the statements to a specific database if there is an incompatibility.

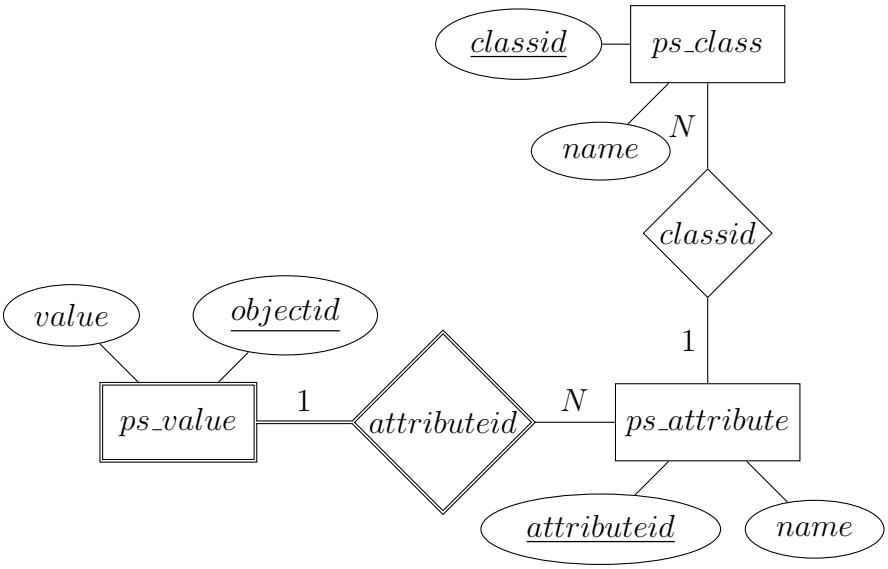

Figure 6.1: *The ER-Model of the generic database layout.*

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.

6.2 Adaption to a custom database layout

Adapting ABEL to a custom database layout needs two steps:

- Implement a `BACKEND` 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 6.1: *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`.

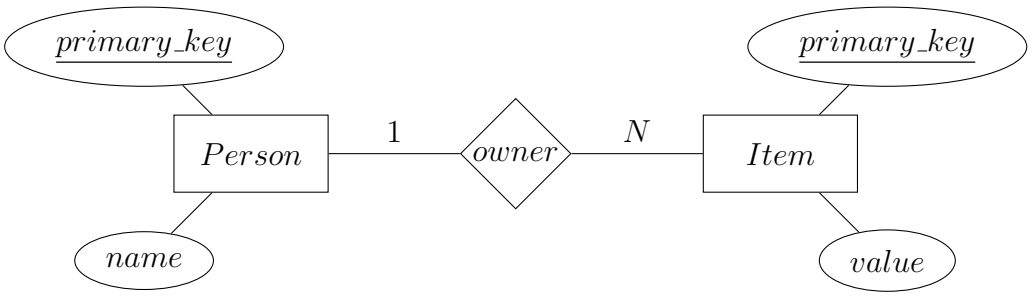

Figure 6.2: *The example ER-Model.*

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
  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 6.2: *The collection handler for LINKED_LIST*

The implementation of *BACKEND* 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
```

```
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 6.3: *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.