ABEL Technical Documentation

Roman Schmocker
Reviewed by: Marco Piccioni

November 20, 2013

Contents

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 read and write operations of the API have a look at the ABEL tutorial.

This is an introduction to the general architecture of ABEL.

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:

- *TRANSACTION*: Represents a transaction and can be used for read and write operations.
- *QUERY*: Collects information like the Criteria, Projection and the type of the object to be retrieved (through its generic parameter).
- *REPOSITORY*: Hides the actual storage mechanism, and can be used for read operations or for creating new transactions.
- *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.

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_BACKEND*.

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 graph traversal layer. It is responsible to take an object graph apart into its pieces and generate a more suitable representation for the next layer. During retrieval this layer is responsible for reconstructing the object graph from the data.

On the next level there is the *BACKEND* layer. Its task is to map the objects, represented as string-value pairs in class *BACKEND_OBJECT*, to a specific storage engine like 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. It is for example responsible for the fact that an update to a newly created object fails

Another important data structure is the *KEY_POID_TABLE*, which maps the object's *object_identifier* to the primary key of the corresponding entry in the database.

2.2.3 Transactions

Every operation 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.

Chapter 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* and the database wrapper classes.

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 *DEFAULT_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*.

3.2 BACKEND

Another important interface is the deferred class *BACKEND*. This layer only deals with *BACKEND_OBJECT* or *BACKEND_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 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 4

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.

4.1 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" [2].

The ER-model in figure 4.1 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.

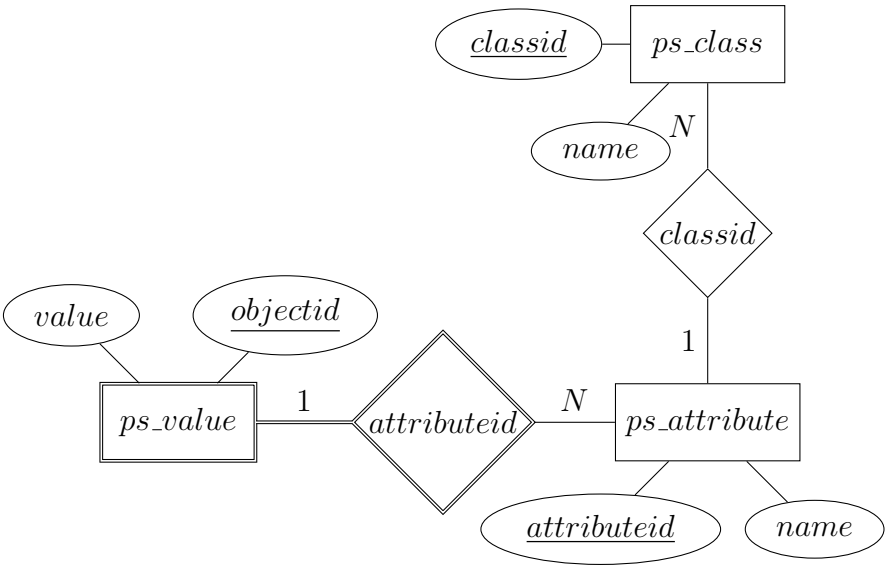

Figure 4.1: *The ER-Model of the generic database layout.*

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 `BACKEND_OBJECT`, the attribute foreign key which is stored inside the class `METADATA_TABLES_MANAGER`, and the object primary key which is part of the `BACKEND_OBJECT` as well.

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.

Bibliography

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

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