Environment-Independent Methodology for Accessing External Data Sources

LAURA M. CASTRO, VÍCTOR M. GULÍAS, CARLOS ABALDE, JAVIER PARÍS
Department of Computer Science
University of A Coruña
Campus de Elviña s/n - A Coruña
SPAIN
{lcastro,gulias,cabalde,javierparis}@udc.es http://www.madsgroup.org

Abstract: - Software engineering is not a static field. Hardware is evolving, and so needs to do software development. Someone walking into a computer store today and buying a personal computer, will most likely end up owning a machine with more than one CPU. And that machine will most likely end up on a network, connected to a lot of other machines and devices. We are talking about paralelism and distribution, two features that threaten to make the software development process harder. To cope with these new parameters, new tools are claiming a place in the vangard of software creation. At the same time, there are also well-known established components, such as our traditional databases, that we still use (and need to use) the same way they have been used for many years. In this article, an environment-independent methodology to combine these two different worlds is be presented, showing that past and future can work together if we properly use abstraction and high-level software engineering tools.

Key-Words: - Software engineering, database access, design patterns, functional programming.

1 Introduction

From its early days, data management has been one of the main purposes of computing. Almost all software applications need to pay special attention to management, storage and retrieval of the information needed to carry out the task they are designed for. This is why databases are essential components in most software systems, whether they be critical or not, regardless of their purpose, scope, or scale.

Database Management Systems (DBMS) are pieces of software which provide structured storage to data, and also a query language to inspect and get the information back. There are multiple well-known DBMSs, both in the proprietary and the free software worlds. The most commonly known are perhaps Oracle [1], DB2 [2], Microsoft SQL Server [3], PostgreSQL [4], MySQL [5] or Sybase [6], and they have been for years very successful products. All of them implement the relational database model [7] and have long market experience behind them, which also means they grant stability and reliability enough to be trusted by developers and software companies.

On the other hand, in the field of software development new challenges appear every day, and so do new initiatives, new ideas, new tools and new solutions. Nowadays, we live in a world where many things happen at the same time and where information and knowledge is spread all over. This paralelism and distribution reaches not only the hardware, but also the software process as well, and as previously known development environments do not seem to adjust to this new conditions very well, new ones are beginning to stand out, striving for recognition to take their place.

Erlang/OTP is a functional language and a set of libraries that were created as a platform to develop robut applications meant to run over a net of computers. It was originated inside Ericsson Telecommunications in the early nineties and its initial aim was to be a tool to program telephone switches. But it turned out to be a programming environment that helped out to speed up the development and to reduce the maintenance effort while generating highly reliable robust pieces of software. This was the key for it to step out from the telephony world and start being used for other purposes.

Nowadays, Erlang/OTP has proved to be a very suitable framework, perfectly valid to develop almost any kind of software application, especially when robustness, reliability, high-avaliability, maintenance ease and transparent distributabion are essential requisites [8, 9, 10, 11, 12]. As it became more and more popular, the set of libraries and

utilities included in the Erlang/OTP distribution increased enormously. In modern releases, it even provides its own DBMS, called *Mnesia* [13]. Amongst the most innovative properties of Mnesia, which is actually a distributed DBMS with a hybrid object-relational data model [14], we find a very high level of fault-tolerance, together with dynamic distribution and reconfiguration. But still, the great influence and major importance of traditional DBMSs is too hard to overcome in many cases. Even more if data is already saved in such storage, i.e. when migrating an application, or building a new one that will work with, or share, already-modeled, pre-existing data.

For these reasons, in its way to a place in the industry world, Erlang/OTP faces the challenge of getting on with traditional DBMSs. In this article we take a close look to it, in an effort to understand how these two agents can work together in the best possible way, not only with regard to the technological solutions available, but also bearing in mind storage and retrieval strategies. For the former point of view, we have explored and compared two of the most important approaches, standard vs. native communication solutions. For the latter, we show that well-known database access patterns are of the most valuable help, and how to apply those patterns, essentially identified with the object-oriented world, on a functional environment.

2 External Data Sources

A database is a structured collection of pieces of information, stored in a computer system and accessible to be questioned, either by a human or by a computer program, through a special query language.

The nature of the pieces of information stored by a database, which can be seen or considered as its structural description, is called *database schema* [15]. Depending on how the database schema is organised or modelled, different *database models* are possible. Some of the most relevant database models are:

- *Hierarchical model*, where the database schema is a tree-like structure with one root and a number of branches or subdivisions (figure 1).
- *Network model*, where data relationships are represented in a many-to-many way, allowing arbitrary graphs and not only trees (like in the hierarchical model), thus resembling a network (figure 2).

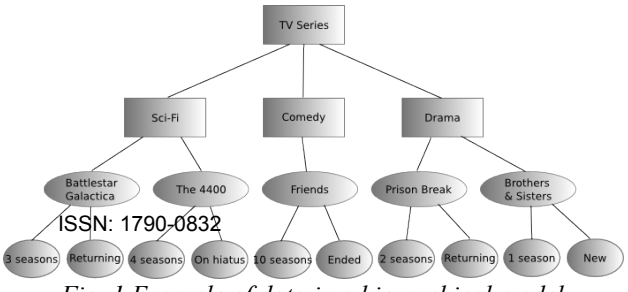

Fig. 1 Example of data in a hierarchical model

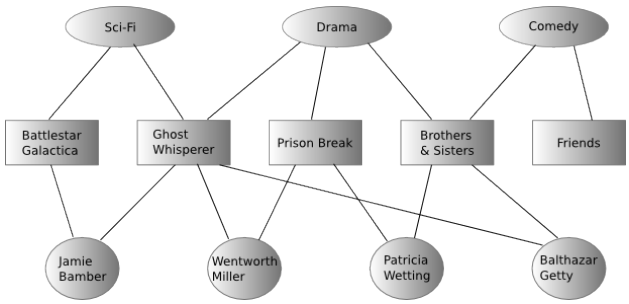

Fig. 2: Example of data in a network model

- *Relational model*, where information, and relationships amongst information items, are modelled according to the set theory [16] (figure 3).
- *Objectual model*, where schema components are objects, comparable to the concept of "object" in Object-Oriented Programming [17] (figure 4).

CAST

Marcia Cross	*Desperate Housewives*
Felicity Huffman	*Desperate Housewives*
Wentworth Miller	*Prison Break*
Patricia Wetting	*Prison Break*
Patricia Wetting	*Brothers and Sisters*
Calista Flockhart	*Brothers and Sisters*

CATEGORY

Drama	*Desperate Housewives*
Comedy	*Desperate Housewives*
Drama	*Prison Break*
Sci-Fi	*The 4400*

Fig. 3: Example of data in a relational model

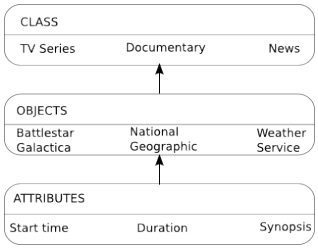

Fig. 4: Example of data in an objectual model

From these main models, the relational model was the first formal database model [18]. It has a strong mathematical background which includes set theory and two-value predicate logic. In the relational model, stored data is operated upon by means of relational algebra [19]. Thanks to this formalism and its rigorousness, relational databases became popular in the eighties and even though other database models where formalised afterwards, and many other new formal models have appeared since, none of them has been able to take its place as the most commonly used database model. For this reason, we will only focus on this specific type of databases.

2.1 Access Types

In order to inspect the collection of data stored in a relational database (RDB), a person (or a computer application) must use, as previously mentioned, a particular query language. The query language for relational databases is an ANSI [20] and ISO [21] standard called Structured Query Language (SQL [22]). To proceed, a human user would type some SQL-queries in an interactive environment. After being introduced, those queries would be processed by the DBMS, which would retrieve the matching data, and the results would appear on the screen. If this process is to be carried out by an external software application (that may need to query the database in order to obtain some data to later on process it, generate some results, or perform some task), the interaction mechanism needs to be slightly different.

There are several ways in which a computer application can submit SQL-queries to a relational database:

- *Using a standard API*, such as Open Database Connectivity (ODBC [23]). An API (stands for Application Programming Interface) is a source code interface provided to support standard and independent requests for services. The use of standard APIs makes it possible for two independent parties (computer programs, application components, system agents) to interact in a clean, pluggable and reliable way (figure 5).

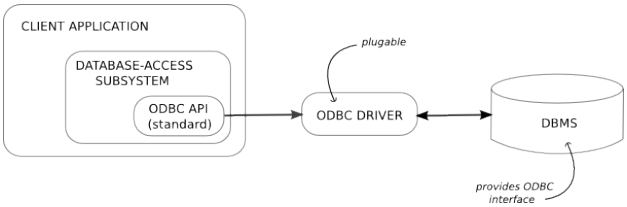

Fig. 5: ODBC-based application architecture outline

- *Using a database-native API*. Even though using a database-native API is generally more efficient than using just a generic standard API (and may offer special features/services), this is also a solution that binds the program to a specific database, thus compromising (or at least restricting) the future flexibility and maintainability of the system (figure 6).

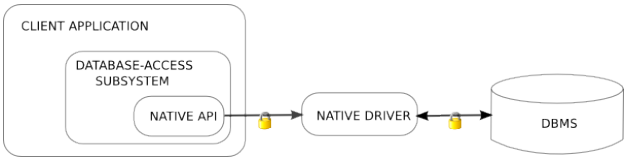

Fig. 6: Database-native driver alternative outline

- *Using sockets*, to directly connect to the database. This implies taking care of all low-level details of the application-to-database communication, which usually is too much work to even take this option under consideration, unless the system under development has very strong speed and efficiency requirements which cannot be fulfilled using an already-existing API (figure 7).

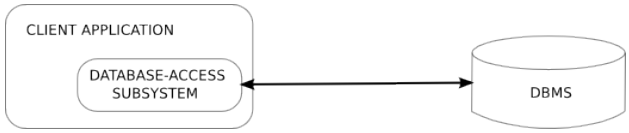

Fig. 7: Direct to-socket-communication solution outline

2.2 Erlang and ODBC

As previously said, ODBC (Open DataBase Connectivity) is a standard API which allows us to query a relational database. Thus, it is a way of communicating applications and database servers using a standard set of functions to open a connection with the database, send queries, and get the results back.

Each ODBC-compatible DBMS provides an implementation of the ODBC API, which internally translates ODBC calls into its own native database dialect. To be able to connect to a DBMS via ODBC, an ODBC driver is also needed, which is the piece of software client applications will link to, thus remaining fully independent from the DBMS. Actually, the fact that the clients only perform standard function calls makes them independent from the driver as well.

Erlang/OTP platform provides support for ODBC, hence allowing Erlang programmers to build applications that can communicate with a relational database via ODBC. And since the ODBC-support module is part of the standard set of libraries, OTP, such communication takes place in a transparent, Erlang-like style (see source code fragment 1).

```
Eshell V5.5.5  (abort with ^G)
% start ODBC Erlang application
1> application:start(odbc).
ok
% obtain connection reference to database
2> {ok, Ref} = odbc:connect("DSN=database-name;"
                            "UID=username;"
                            "PWD=apassword", []).
{ok,<0.39.0>}
% send query to database using a connection
3> odbc:sql_query(Ref, "SELECT * FROM atable").
{selected,["column_a", "column_b", "column_c"],
          [{"Patricia", "GMT+02:00", 3},
           {"Javier", "GMT+02:00", 3},
           {"John", "GMT+01:00", 1}]}
% close database connection
4> odbc:disconnect(Ref).
ok
% close ODBC Erlang application
5> application:stop(odbc).
ok
```

Table 1 Erlang/ODBC code sample

2.3 Erlang Native Drivers

As happens on many other programming environments and development tools, some native drivers that have appeared recently are available for Erlang/OTP. These Erlang drivers are database-specific, and what they allow is to directly connect to a given DBMS (PostgreSQL or MySQL, at the moment) from an Erlang environment. Nevertheless, the actual communication is, again, transparent for the Erlang client applications (see source code 2).

```
Eshell V5.5.5  (abort with ^G)
1> application:start(sql).
ok
2> application:start(psql).
ok
3> sql:q("SELECT * FROM atable").
[{"Patricia", "GMT+02:00", 3},
 {"Javier", "GMT+02:00", 3},
 {"John", "GMT+01:00", 1}]
4> application:stop(psql).
ok
5> application:stop(sql).
ok
```

Table 2: Native Erlang driver code sample

3 Database Access Patterns

From an application conception to its actual development, some initial work needs to be done. Apart from requirements elicitation and functionalities determination, the broad analysis task, as fas as persistence aspects are concerned, will usually involve some of (or all) the following steps:

- *Identification of the persistent business objects*. By persistent business objects in a system we understand those information or data elements whose life can extend during the running time of the application or even beyond, from one execution to the next. Once those relevant elements are recognized, their persistent parts or components (usually referred to as the object "state") need to be identified as well. These are the ones the relational database will take care of.
- *Identification of the persistence operations*. Persistence operations usually include creation, deletion, update, search and retrieval of the persistent elements (and/or their states) enumerated in the previous step.
- *Control flow definition*. A control flow is the sequence of operations that need to be executed in order to carry out some of the *use cases* of a system or application, i.e. a piece of its functionality, a service. Eventually, persistence operations will be involved in some of those use-case operations.
- *Functionality or service API implementation*. Once the control flow is clear and stated, related use-cases (services) should be grouped together, and one or more interaction or service APIs should be defined, to be the interaction points used by other parts of the system (or even third-party clients).

These four stages help us approach and organise the process of designing and building the storage and retrieval module/subsystem of an application. The actual implementation is the last task, for which some other formal tools are available to help developers to perform it in a more efficient and, ideally, error-free manner. One of this conceptual tools are *software design patterns*.

A *software pattern* is a "repeatable software solution to a particular kind of problem, which has proved to be both efficient and simple" [24]. Applying software design patterns during the design and development process of a system or application, helps to reduce the development effort, prevents the appearance of common or already-known errors, and increases its maintainability.

In the next sections some software design patterns closely related to database access are presented and explained.

3.1 Value Object Pattern

The *Value Object* (VO) pattern represents the abstraction of the state or value of a business object in the domain [24]. The business object properties are modeled as the VO internal attributes, protected with a restricted access. The interaction of the business object with the outside world, its functionalities, and the access to its properties is modeled as the VO operations, with the appropriate visibility in each case.

The structure of the VO pattern is shown in figure 8, using the UML modelling language [25, 26].

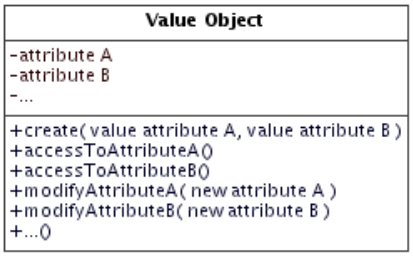

Fig. 8: Value Object pattern structure

We can use the VO pattern to abstract the persistent properties of a business concept as elements which will be saved on a permanent storage (RDB). Access to this data is restricted and conducted through function calls, which can be read-only ("access" functions/methods) or read-write ("update" functions/methods).

3.2 Data Access Object Pattern

The *Data Access Object* (DAO) pattern, which structure can be seen in figure 9, hides the interaction with the persistent storage (database) from the rest of the system or application, setting apart business logic from persistence logic [24].

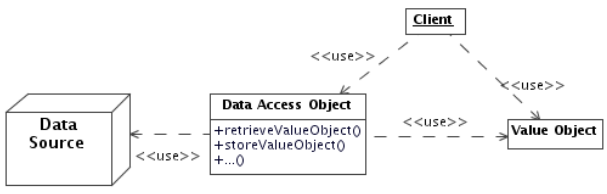

Fig. 9: Data Access Object pattern structure

A DAO is an element in the system that will provide an internal API to recover/store VO data, abstracting the external storage source (typically, and in our examples, a relational database, but it could be any other storage media). A DAO is the software piece of the system that will encapsulate all the datasource access details, for example, the invocation of a standard API to access the database using ODBC, or else the instructions to deal directly with a particular database by means of a native driver, or even the implementation of a customised access to the database.

We will usually model at least one DAO for each VO, in a one-to-one correspondence. Thus, the DAO will be responsible of all the persistence operations involving its associated VO. In case we have to deal with some complex VOs, composed by several basic VOs, we can define a DAO that makes use of other DAOs, like presented in figure 10.

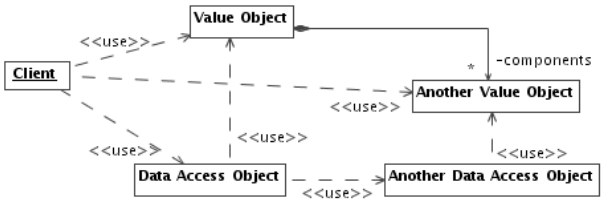

Fig. 10: Data Access Object for a complex VO

From a theoretical point of view, we can consider that the DAO pattern mixes functionalities commonly identified with two different software patterns:

- *Adaptor pattern* [24], since it provides an interface for persistence, regardless of the type of datasource and the access method we choose for it.
- *Factory pattern* [24], since it creates (recovers, and also stores) VOs, making this process datasource-independent for the rest of the system.

3.3 Facade Pattern

The *Facade* pattern provides a single interface to a set of components/functionalities in a system [24]. By using facades, clients do not need to know the structure, members, or any other internal details and complexities in a system: they just interact with the facade component, which has convenient methods for fulfilling their needs.

The use of facades as interfaces for the functionality of a system (or part of it) decouples it from its clients (or other parts of the system), allowing modifications to be transparent (since only facade internals are affected), and thus increasing modularity and maintainability. Apart from offering a neat access point to a set of logically related use

cases, a facade can represent workflow as well. This implies more flexibility (and simplicity) in developing the system, because different facades can be created (as many as needs), each of them presenting a single specific-purposed well-designed API, leading to a better structure and compartimentalisation, since modifications on those parts of the system isolated by facades will not affect the clients of the facades, but only the facades themselves.

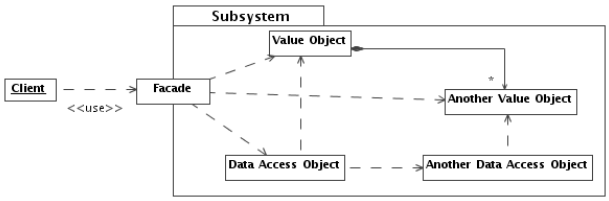

Fig. 11: Facade pattern structure (interaction with DAO and VO)

Thus, the combination of these software patterns (Value Object, Data Access Object and Facade) represents the perfect mechanism to model the persistence subsystem of an application. This will configurate, as we have already explained, a flexible, modular and reusable solution from which other components of the system or clients will remain as independent as possible, and future changes in the internal architecture will have minimum repercussion.

4 Erlang Implementation

Erlang is a distributed, concurrent, functional programming language. As a functional language, Erlang has no constructs inducing side effects (with the exception of thread communications -*processes*-). Values in Erlang (i.e. non-reducible expressions) range from numbers and *atoms* (symbolic lower-case constants) to complex data structures[1] (lists, tuples, and records -useful syntactic sugar for accessing tuples by name instead of by position, similar to C structures-) and functional values.

An Erlang function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments (*pattern-matching*). Iterative control flow is carried out by using recursion and expresssions are evaluated eagerly. Functions are grouped into *modules* (implementation files), and a subset of those functions can be exported, declaring both function name and arity, to be used from other modules.

Experience shows that using a functional language to develop complex systems whose logic involves complicated algorithms contributes greatly to reduce the development effort. The reason lies in that functional languages are more abstract, higher level languages, and thus they are closer to our reasoning level. Hence, it is easier to deal with more complex problems when approaching them from a functional point of view.

Erlang popularity is increasing these days as paralelism and distribution are more and more present in systems, equipment and requirements. Erlang/OTP was designed from scratch having in mind, as main objectives, distribution and reliability. Thus, the language provides, for instance, simple mechanisms to run code on several different nodes with almost no effort, and also powerful supervision capabilities [27]. Last but not least, its concurrency model, based on asynchronous message-passing, eases the developer task when specifically programming parallel behaviour for distributed environments [28]. Knowing this, it is not strange that Erlang has been recently referred to as “the next Java” [29].

Our challenge here is combining this innovative and convenient development platform with analysis and design solutions that seem to be only thought for the object-oriented world. We will show here that it is not a utopia to use software design patterns and then implement our application using a functional language.

The storage access use case we have presented here is one of the simplest scenarios of object-oriented design that we can try to implement in a non-object-oriented development environment. Figure 12 shows the typical architecture that will correspond to what we have explained in previous sections.

As we can see, the proposed structure is merely based on a set of different entities or objects, offering different operations, with and without internal state. Their interrelationships are just based on plain UML assotiations, and use dependencies. Despite being a typical UML diagram, most likely to be related to an object-oriented implementation, there is no inheritance and there is no polimorphism involved.

1 Since Erlang has no static type system either, lists and tuples/records can be heterogeneous and hold any valid value.

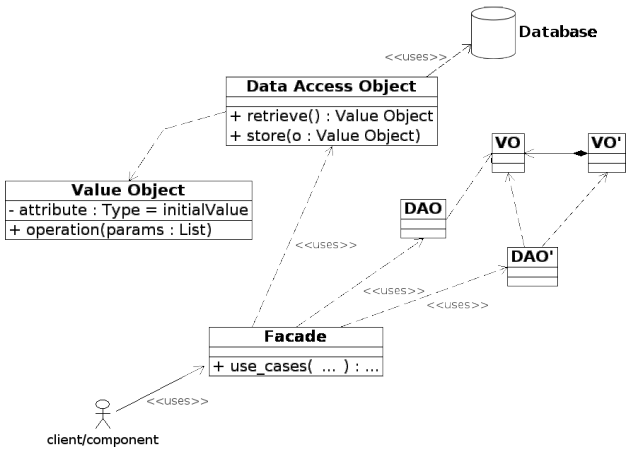

Fig. 12: Proposed architecture for database access

So we just need to define a model in our functional development environment to represent the concept of "object", its internal state (attributes) and its operations, to be able to implement such a design in Erlang/OTP. The model we propose here is leant on around Erlang modules, which we have mentioned at the beginning of this section.

Erlang modules are the implementation unit elements in this language (like classes in Java). We propose identifying each object/class in our design with an Erlang module. To represent object internal state, we suggest using Erlang records (similar to C structures) defined inside the corresponding Erlang modules. Finally, the object interface (operations) can be modelled precisely by the set of exported functions of each Erlang module.

Thus, following this ideas, each Value Object in the system, representing an application business object, will correspond with a module (see code fragment 3), with a record definition -if there are object attributes to implement- and a list of exported functions, including the appropriate access and update ones, as object interface.

```
-module(a_value_object).
-export([create/2, access_A/1, modify_A/2]).

% definition: structure representing VO
-record(a_value_object, {attribute_a, attribute_b}).

% method: returns new value object
create(ValueAttributeA, ValueAttributeB) ->
   #a_value_object{attribute_a = ValueAttributeA,
                   attribute_b = ValueAttributeB}.

% method: accesses to VO attribute value
access_A(AValueObject) ->
    AValueObject#a_value_object.attribute_a.

% method: changes VO attribute value
modify_A(AValueObject, NewAttributeA) ->
    AValueObject#a_value_object{attribute_a = NewAttributeA}.
```

Table 3: Erlang VO implementation example

As we previously outlined in section 3.2, for each class representing a business object (VO), a DAO module is required. Again, an Erlang module for each DAO will be implemented. Source code fragment 4 shows an abstract and generic view of an Erlang DAO.

```
-module(a_dao).
-export([retrieve_VO/2, store_VO/2]).

% method: retrieves value object from database
retrieve_VO(TxReference, VOKey) ->
   case odbc:sql_query(TxReference,
                       "SELECT attribute_a, attribute_b "
                       "FROM vo_table...") of
      {selected, ["attribute_a", "attribute_b"], [{A, B}]} ->
            value_object:create(A, B);
      _ -> {error, unknown_object}
   end.

% method: stores value object on database
store_VO(TxReference, AValueObject) ->
   ValueA = access_to_attribute_a(AValueObject),
   ValueB = access_to_attribute_b(AValueObject),
   case odbc:sql_query(TxReference,
                       "UPDATE vo_table SET ...") of
      {updated, 1} -> ok;
      _ -> {error, unknown_object}
   end.
```

Table 4: Erlang DAO implementation example

DAO modules take care of data persistence; in the previous source code example, the external data source is a relational database accessed via ODBC, but this details are hidden from and are transparent to DAO clients (no matter whether external ones or part of the same system). Furthermore, any change in storage nature or storage access method will only affect the DAOs and not their clients.

The only thing a DAO does need from their clients, as we can see in source code example 4, is a transaction identifier, in order to allow the invocation of different methods grouped in a single transaction, when needed. Transactionality implies that, given any error at any of the grouped steps, the whole process would be rolled back (cleanly undone) and the database information would remain intact, as if nothing (no operation) had happened. Of course, this kind of responsibility is not desirable to fall on clients, so a facade is very helpful here. This facade will not only present a set of logically related use-cases, but also encapsulate transactionality issues (see source code 5).

```
-module(a_facade).
-export([copy_object/1]).

% method: given a business object key, returns a copy
copy_object(VOKey) ->
   {ok, TxReference} = odbc:connect(" ... ", []),
   VO = a_dao:retrieve_VO(TxReference, VOKey),
   A = a_value_object:access_to_attribute_a(VO),
   B = a_value_object:access_to_attribute_b(VO),
   CopyVO = a_value_object:create(A, B),
   a_dao:store_VO(TxReference, CopyVO),
   odbc:commit(TxReference, commit),
   odbc:disconnect(TxReference),
   CopyVO.
```

Table 5: Erlang facade implementation example

In systems with a design like the one presented here, clients only need to invoke the use cases (functionalities) they want in each moment. Their requests will be translated by the facade in as many internal modules (DAOs and VOs) methods calls as needed, grouped to be transactional if appropriate, and finally performed in a clean, secure and transparent way. If anything changes in such system, clients are very unlikely to notice, since two different levels of abstraction (DAOs, for storage details, and facades, for use-case implementation details) protect them.

4.1 Performance Comparison

Almost all major relational database vendors and all free software DBMS provide ODBC support. On the other hand, native solutions exist in Erlang only for the latter: MySQL [30] and PostgreSQL [31] Erlang native drivers. We have conducted several performance tests in order to compare and contrast these alternatives. We have tested Erlang ODBC access to a PostgreSQL database, and Erlang native access to the same DBMS.

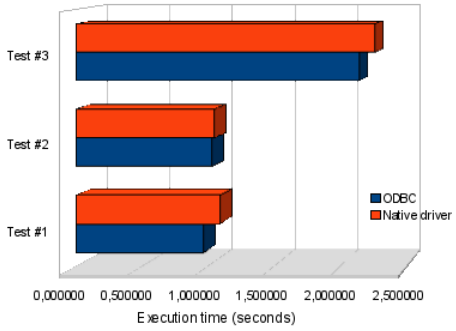

Fig. 13: Performance test results

Our experiments consisted in the execution of real use-case queries (taken from the risk management application ARMISTICE [32, 33, 34, 35]) a number of times, both using ODBC + PostgreSQL, and PostgreSQL Erlang native support, and the measurement of the time elapsed in each case.

ARMISTICE is an information system whose business logic has been developed using Erlang/OTP. Using a functional language such as Erlang was a key factor for success not only in implementing a software application to deal with such a complex business domain as insurance management, but also in reaching an abstraction level at the definition of the system which makes it applicable to different business fields. This innovative risk management system (RMIS) is meant to be a tool for both the expert and the daily non-expert users. An advanced profile will use ARMISTICE to specify a set of resources and their relevant properties of interest, as well as the insurance policies contracted to protect those resources from the consequences of potentially harmful events, whichever these might be for each particular case. On the other hand, the system is of assistance also to the kind of user that, without any expert knowledge regarding coverages and warranties, has to deal with incident reports, accident claims, and file trackings. In this case, ARMISTICE, which has been in production at an international company for a few years now, helps by retrieving and isolating just the relevant information for each scenario, according to the provided contextual data, and thus, giving valuable support to actions and decisions.

The selected queries involved complex read (SELECT) queries, performing in every case not only SELECT...FROM clauses with WHERE/GROUP BY filters, but also several SUBSELECTS and JOINS, all of them corresponding to three different ARMISTICE application use cases, namely:

- *Report of all risk situations* (objects of interest for the risk manager) belonging to a specific risk group (test #1). Risk situations are the largest group of persistent business objects in ARMISTICE, so inspecting and recovering all the risk situations belonging to a specific risk group, their attributes and values, is a database-intensive operation.
- *Report of all risk groups* (abstractions to group risk situations with the same attributes, i.e. shops, vehicles,...). A risk group defines the properties a certain (sub)set of risk situations must have, and also their default values. In our ARMISTICE study case, there are few risk groups in comparison with the great number of risk situations associated with each one, so this test (test #2) is much less database-intensive than the previous one.
- *Report of all the risk situations* belonging to a *specific risk group* that have *no coverage*. Risk situations in the system are usually covered against hazards by means of insurance policies (also defined and managed by the system, and stored on the same database). However, recent risk situations may still be uncovered if a coverage request has not been fulfilled and processed yet. Ideally, the set of uncovered risk situations should be empty, so this report is reviewed quite often (meaning this is a frequently invoked use case). On the other hand,

since creation of new risk situations and coverage requests are operations performed almost daily, the set of uncovered risk situations tends to be small, making this test (test #3) not too database-intensive in terms of amount of information involved (but being, as we previously noticed, database-intensive in terms of invocation frequence).

Results of these three tests in figure 12, compare and contrast ARMISTICE server performance when using ODBC-access to the database, and when using Erlang native PostgreSQL access. By executing these basic use cases we pretend to obtain some evidence about which datasource access method is preferable. As we can see, time measurements indicate that, the elapsed time falling in similar ranges, the first option turns out to be more efficient in all the cases. This might not be the expected result, since one should think that a native solution would take advantage of its intricsic properties to improve its performance in terms of effectiveness. All in all, in light of these results, other factors need to be considered and, in our opinion, each particular solution maturity level is the key to understanding these results. While ODBC access support to relational databases has been part of Erlang/OTP for quite a few years now, native access to PostgreSQL is justs a couple of years old now. This not only means that the first is more likely to have been more extensively used, but also debugged and fine-tuned. In consequence, it would be interesting to repeat these tests as soon as new releases of the Erlang/OTP native PostgresSQL driver appear, and check if there is any significant change in these measurements.

Of course, relevance of complex queries as performance indexes depends on the software application. In this case, results show that the ODBC alternative is better in most cases (which, as previously said, may be due to driver development and optimisation status), but further testing might also be performed to ensure convenience of each option for a particular environment, business model and application workload.

5 Conclusions

In this article we have analysed, studied and presented the scenario of accessing a relational database from an Erlang environment. Relational databases are the most commonly used kind of databases nowadays, and even though there are other, newer and more complex database models in the market, relational DBMSs are not likely to lose their predominance, at least in the next years to come.

As far as Erlang is concerned, it is undoubtly a very interesting development platform, most of all bearing in mind requirements that are more and more demanded day after day: robustness, fault tolerance, high availability, distributability, concurrency, reliability, ease of maintenance... Thus, we have explored different solutions available when combining the Erlang/OTP development environment with relational DBMS storage. However, other implementations of similar native solutions for this functional development platform are appearing, so extending the same experiments to cover also those options should be an interesting exercise.

Despite the specific case study we have presented here, the design methodology that has been explained can be applied regardless of the implementation stage details (development environment, programming paradigm, type of application, etc.). Nevertheless, only careful analysis of application workload, business use cases, and nature of both data operations and data properties, together with technology solution maturity, would provide enough judgement references, in each case, to choose not only between database access alternatives, but even between technological approaches [36].

In any case, we have illustrated, once more, how addressing the design and development of a new software product using abstraction and high-level software engineering tools (such as design patterns), gives a greater degree of confidence in the quality of the final outcome.

Acknowledgements

This work has been partially supported by MEyC TIN 2005-08986 and XUGA PGIDIT06PXIC10516 4PN.

References:

[1] Oracle, Oracle Database Management System, http://www.oracle.com.

[2] IBM, DB2 Database Management System, http://www.ibm.com/db2.

[3] Microsoft Corporation, Microsoft SQL Server, http://www.microsoft.com/sql.

[4] PostgreSQL Server, http://www.postgresql.org.

[5] MySQL, http://www.mysql.com.

[6] Sybase, Sybase Adaptive Server Enterprise, http://www.sybase.com.

[7] David Maier, *Theory of Relational Databases*, Computer Science Press, 1983.
[8] Lambdastream S.L., Video on Demand Kernel Architecture (VoDKA), http://www.lambdastream.com/lambda/products/VoDKA?l=EN.
[9] Igalia S.L., SERVAL: Internet Software VLAN Switch developed in Erlang, http://serval.igalia.com.
[10] Yaws, High Performance WebServer, http://yaws.hyber.org.
[11] Tsung, Multi-protocol distributed load testing tool, http://tsung.erlang-projects.org.
[12] Ejabberd, Jabber/XMPP instant messaging server, http://www.ejabberd.im.
[13] Mnesia, Distributed functional object-relational Erlang/OTP database, http://www.erlang.org/doc/apps/mnesia/index.html.
[14] Irina Astrova, Ahto Kalja, Storing OWL Ontologies in SQL3 Object-Relational Databases, *8th WSEAS International Conference on Applied Informatics and Communications* (AIC'08), 2008, pp. 99-103.
[15] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley, 2006.
[16] Karel Hrbacek and Thomas Jech, *Introduction to Set Theory*, CRC, 1999.
[17] Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison Wesley, 2001.
[18] Abraham Silberschatz, Henry F. Korth and S. Sudarshan, *Database Systems and Concepts*, McGraw-Hill, 2005.
[19] C. J. Date, *An Introduction to Database Systems*, Addison Wesley, 2003.
[20] ANSI, American National Standards Institute, http://www.ansi.org.
[21] ISO, International Standards Organization, http://www.iso.org.
[22] Kevin Kline, Daniel Kline and Brand Hunt, *SQL in a Nutshell*, O'Reilly, 2004.
[23] Roger E. Sanders, *ODBC 3.5 Developers Guide*, McGraw-Hill, 1998.
[24] Eric Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1996.
[25] Grady Booch, Ivar Jacobson, James Rumbaugh, *The Unified Modeling Language UML*, Addison Wesley, 1998.
[26] Artis Teilans, Arnis Kleins, Yuri Merkuryev and Andris Grinbergs, Design of UML Models and their Simulation using ARENA, *WSEAS TRANSACTIONS on COMPUTER RESEARCH*, Vol. 3, No. 1, 2008, pp. 67-73.
[27] Francesco Cesarini and Simon Thompson, *Erlang programming*, O'Reilly, 2008.
[28] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.
[29] Ralph Johnson, *Erlang, the next Java*, 2007.
[30] Mickael Remond, MySQL native Erlang driver, http://support.process-one.net/doc/display/CONTRIBS/Yxa.
[31] Erlang-Consulting, PostgreSQL Erlang native driver, http://www.erlang-consulting.com/aboutus/opensource.
[32] ARMISTICE, Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures, http://www.madsgroup.org/armistice, 2002.
[33] Víctor M. Gulías, Carlos Abalde, Laura M. Castro and Carlos Varela, Formalisation of a functional risk management system, *8th International Conference on Enterprise Information Systems*, INSTICC Press, 2006, pp. 516-519.
[34] Víctor M. Gulías, Carlos Abalde, Laura M. Castro and Carlos Varela, A new risk management approach deployed over a client/server distributed functional architecture, *18th International Conference on Systems Engineering*, IEEE Computer Society, 2005, pp. 370-375.
[35] David Cabrero, Carlos Abalde, Carlos Varela and Laura M. Castro, ARMISTICE: An experience developing management software with Erlang, *Principles Logics and Implementations of High-Level Programming Languages*, 2003.
[36] Jiri Kohout and Katerina Kucerova, Two views on data integration, *8th WSEAS International Conference on Applied Informatics and Communications* (AIC'08), 2008, pp. 506-514.