ARMISTICE: An Experience Developing Management Software with Erlang *

David Cabrero[1], Carlos Abalde[2], Carlos Varela[2], Laura Castro[2]
LFCIA Lab, Computer Science Department
University of A Coruña
Campus de Elviña s/n. 15071, A Coruña, Spain
[1]cabrero@udc.es, [2]{carlos, cvarela, laura}@lfcia.org

ABSTRACT

In this paper, some experiences of using the concurrent functional language Erlang to implement a classical vertical application, a risk management information system, are presented. Due to the complex nature of the business logic and the interactions involved in the client/server architecture deployed, traditional development techniques are unsatisfactory. First, the nature of the problem suggests an iterative design approach. The use of abstractions (functional patterns) and compositionality (both functional and concurrent composition) have been key factors to reduce the amount of time spent adapting the system to changes in requirements. Despite our initial concerns, the gap between classical software engineering and the functional programming paradigm has been successfully fullfiled.

Keywords

Functional programming, distributed computing, concurrent programming, design patterns, business logic, client/server architecture

1. INTRODUCTION

In this paper, we introduce some of our experiences designing and implementing a three-tier client/server application called ARMISTICE (*Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures*). This application is a risk management information system designed for a large company and it is intended to be deployed by late 2003. The main novelty of this system is that it has been partly developed using a declarative language, the concurrent functional language Erlang [4]. In order to achieve the reliability and flexibility required, the server side needs to be distributable, either on a cluster of off-the-shelf computers or inside a virtual private network, for the inherent distribution of franchises in a large holding enterprise. In addition, the logic of the application has a great level of complexity, managing different kind of heterogeneous business objects representing risks, rules modeling the exposures to dangers of such risks, and applicability constraints for insurances covering them. The use of Erlang has considerably simplified the development process of such server tier.

The use of abstractions (functional patterns) and compositionality (both functional and concurrent composition) have been key factors to reduce the amount of time spent adapting the system to changes in requirements. Despite our initial concerns, the gap between classical software engineering and functional programming has been successfully fullfiled.

This paper is structured as follows: First, some background knowledge is presented to introduce the application domain and the main requirements. Section 3 is devoted to introduce the design of our case study, the ARMISTICE system. Section 4 focuses on the use of Erlang to help simplifying some aspects of the project. Finally, we present the conclusions.

2. A GLIMPSE OF THE DOMAIN

The presented case study is the result of a request made by a large holding enterprise after exploring several applications available in the market without finding any suitable due to its particular and complex domain. When facing the development of the application, the usual issues of software engineering were identified. This isseus set the first constraints for the project:

- *Lack of specific knowledge about the application domain.* The complex domain (insurances, risks, exposures) suggests a great deal of effort communicating with the user expert. Thus, an iterative development approach was advised, and, consequently, rapid prototyping.
- The communication with the domain expert is also problematic because of *her lack of knowledge about software engineering.* In this sense, we found the use of object-oriented abstract models (i.e., UML artifacts) to be quite effective.

*This work has been partially supported by Spanish MCyT TIC 2002-02859 and by the company Alfa21 Outsourcing S.L. (FUAC R&D project 2/79)

Erlang Workshop '03, 29/08/2003, Uppsala, Sweden

- The application needs to save data in a *persistent storage*; nowadays the most mature technology is relational databases, which was a must for the user even though other alternatives were under consideration, such as Erlang's distributed database Mnesia [1].

2.1 Application Domain

The system can be classified as a risk management information system. The enterprise that requested its development needs an application to manage claims for accidents that happen in its shops and resources all over the world. This management includes *accident* management and tracking activities (payments, invoices, repairs...), and other lower level activities such as the modeling of the contracted *policies* or the processes involved in the helping decision system used to select the most suitable warranties when the user introduces information about a new accident. A *warranty* can be seen as a clause of the insurance policy that covers the damages that may suffer a *risk situation*, an insured resource, when some danger acts on it.

In the analysis and design stages, three big subsystems were identified: *risks subsystem*, *policies subsystem* and *claim subsystem*, respectively. Each one defines business objects for the next subsystem. Thus, the risk subsystem defines the notion of *exposure* based upon dangers and entities, the policies subsystem uses those exposures to define *insurance policies* based upon exposures and applicability constraints and, finally, the claim subsystem uses the insurance policy to track an *accident* affecting one or more of the covered exposures, providing useful guidance to forecast limits and franchises.

As it can be noted, the application domain is quite complex. A lot of entities, with complex relations among them, were identified in the analysis of the domain. However, in the following sections, we give a brief overview of the basic concepts involved in each subsystem, that may help to obtain a better understanding of the application domain and goals.

2.1.1 Risks Subsystem

This is the most abstract subsystem. Here, a set of basic concepts are modeled and managed. This concepts are used througout the whole application:sic to the working of the whole application: risk situations, risk groups, dangers, entities, entity categories, exposures....

This subsystem is very dynamic. The user can define a risk group by specifying the metainformation needed for each instance of a risk situation belonging to this group. The concrete data for a specific risk situation can change with the time but references to past values can be needed, thus the subsystem must keep track of any change in order to get back in time, if necessary.

2.1.2 Policies Subsystem

This subsystem is closer to the domain. Its main goal is the modeling of the policies contracted by the company to give support to the sinisters that might happen in its resources.

Basically, a policy is composed by a *conditional*. A conditional can be seen as a contract, composed by a set of clauses, where each clause is a set of *warranties* for the exposures and *formulas*.

The goal is the translation of the hard copy document (the policy) into a representation that could be manipulated by the system to carry out several operations. Some examples of operations are the process of deciding if a policy includes any useful warranty to cover a risk situation against a danger that has produced some damage, or the calculation of a policy price in a automatic or semi-automatic way. To make possible this kind of tasks, a representation of the conditionals based on formulas and a logic-expression language is defined.

On the other hand, in this subsystem are included other more traditional tasks such as the accounting of payments or the document management.

2.1.3 Accident Subsystem

The accident subsystem is the part of the system used daily by users all over the world (the previous subsystems are restricted to be used by a reduced group of expert users). Its goal is the management of a claim for a given accident.

When the user adds a new accident, there is a decision helping process to guide the user in the selection of the most suitable warranty (and policy) to cover the expenses related with the accident consequences. After that, the subsystem is responsible for tracking the accident and manage the related tasks.

2.2 Main Goals

The following requirements were needed to fulfill the final user expectations and to get a added value over other similar applications:

- *Server flexibility and scalability.* As the number of software clients grows, it is possible to add new nodes, maybe using a cluster of off-the-shelf components, to scale up server's performance.
- *Server reliability.* Fault-tolerant code and backup nodes improve availability of the service.
- *Multiplatform.* Both client and server tier should be multiplatform, even though the preferred OS for desktop clients is Microsoft Windows©. Regarding the server side, Linux is the preferred operating system, but not necessarily over x86 processors.
- *Distribution across Virtual Private Network.* The server tier will be distributable in several ways: proxy-servers, backup servers...
- *Multiprotocol middleware.* Middleware will be based on standard RPC protocols. This will simplify adding new clients for new platforms (for instance, PDAs) to the application.

3. THE ARMISTICE PROJECT

ARMISTICE is a three-tier client/server application as shown in figure 1. As we consider Erlang GUIs not mature enough at the moment of starting the project, the language selected for the client tier was Java [5] (Swing); thus, Erlang is restricted to the implementation of the server tier.

3.1 Client

As said before, the client tier is implemented using the Java programming language; this client tier is defined as a thin client, in the sense presented by [8] (absence of business logic), with very few responsibilities: data presentation, validation of input data, ... The interaction with the server,

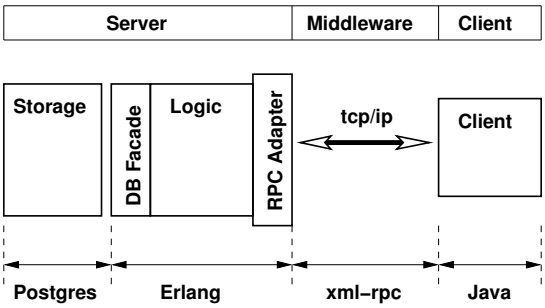

Figure 1: 3-tier architecture of ARMISTICE

which defines the business logic, is carried out throughout XML-RPC requests. Figure 2 shows a snapshot of a part of the graphical interface of this client tier.

Even though the (apparent) simplicity of the client tier with respect to the server tier, it should be pointed out that most of the development time and human resources are consumed by the graphical interface of this client tier.

Although Java/Swing was the selected technology for the implementation of the client tier, some other technologies were taken into account. In particular the use of a classic web interface and the use of some kind of interface definition language [2] were considered. The main advantage of the web interface was its simplicity: with this kind of interface, there are only a limited set of elements and types of interactions among components, then its development is easier. However, at the same time, this simplicity is a great handicap because it constrains the system usability.

On the other hand, interface definition languages were taken into account, but they were rejected because they seem to be insufficient for modeling the high number of dynamic elements and interaction possibilities among UI components.

3.2 Middleware

One of the initial requirements of the project is the definition of a middleware using a standard protocol for performing remote procedure calls. In particular, in our case study this protocol is XML-RPC [3]. The advantages of this approach are clear: low coupling among server and clients, and the transport of data using HTTP eases the communication through firewalls and existent security layers, such as SSL.

XML-RPC also has some potential drawbacks. Its simplicity could be a problem if compared with other middleware options such as SOAP or IIOP (Corba). But this does not really represent an actual problem because we do not need all the facilities provided by other more complex protocols. The client side is only a presentation layer, so facilities like object references are not needed. XML-RPC fits very well with the simplicity of our clients.

3.3 Server

The server tier, developed using the distributed functional language Erlang, is divided into three main subsystems: the *middleware adapter*, responsible for receiving client requests and converting them to the business logic API; the *business logic*; and the *storage accessor*, which is responsible for the communication with the database engine and for the serialization and deserialization of data into the persistent storage, a relational database.

3.3.1 Middleware adapter

This subsystem has the resposibility of receiving client requests (wrapped as XML-RPC messages), making the necessary adaptations to input data (data type conversions, format adaptations...) and building the response to the client. This response is built performing the necessary calls to the business logic and storage accessor subsystems.

This middleware adapter includes all the necessary logic to format the results to the client tier (Java presentation layer). The goal is to minimize the necessary data processing done by the client. Moreover, this subsystem has the responsibility of the management of the user sessions. Sessions are stored in a database (Mnesia) distributed over the cluster of nodes where the server is deployed. This kind of storage makes easier the state sharing between cluster nodes, and consequently the implementation of features like distribution and fault tolerance.

The use of a different user interface (client tier) implies the implementation of a new middleware layer, which must be adapted to the new interface particularities.

3.3.2 Business Logic

This subsystem implements all the business logic, which is independent of the server access peculiarities. Examples could be the modeling of policies, warranty search, statistics and global information management, etc. This kind of operations are based on the persistent server storage, accessed using the storage accessor subsystem API.

3.3.3 Storage Accessor

This subsystem is the nearest level to the persistent storage. It models the domain entities following a simple object representation in Erlang that simplifies the translation from the UML model to a non object-oriented implementation (see 4.1 section).

Besides the definition of the objects and the relations between them, Some APIs are defined at this level to allow the access to the persistent storage of them using the DAO and accessor patterns. A relational database was selected to the storage of the objects, and the access is done using ODBC.

4. BUILDING THE SERVER TIER USING A FUNCTIONAL LANGUAGE

The logic of the server tier has been programmed using a conventional functional style. For example, this is a fragment of the construction of a tree of objects that uses the well-known abstraction `map`:

```
load_treetable(Session, Oid) ->
  ...
  B = lists:map(fun(RiskSituationVO) ->
           {ok, _, Row} = treetable_row(RiskSituationVO),
           Row
      end,
      RiskSituations),
  {ok, {array, B} };
  ...
```

However, to simplify the mapping between objects identified at analysis and design, a method to define such mapping might be established.

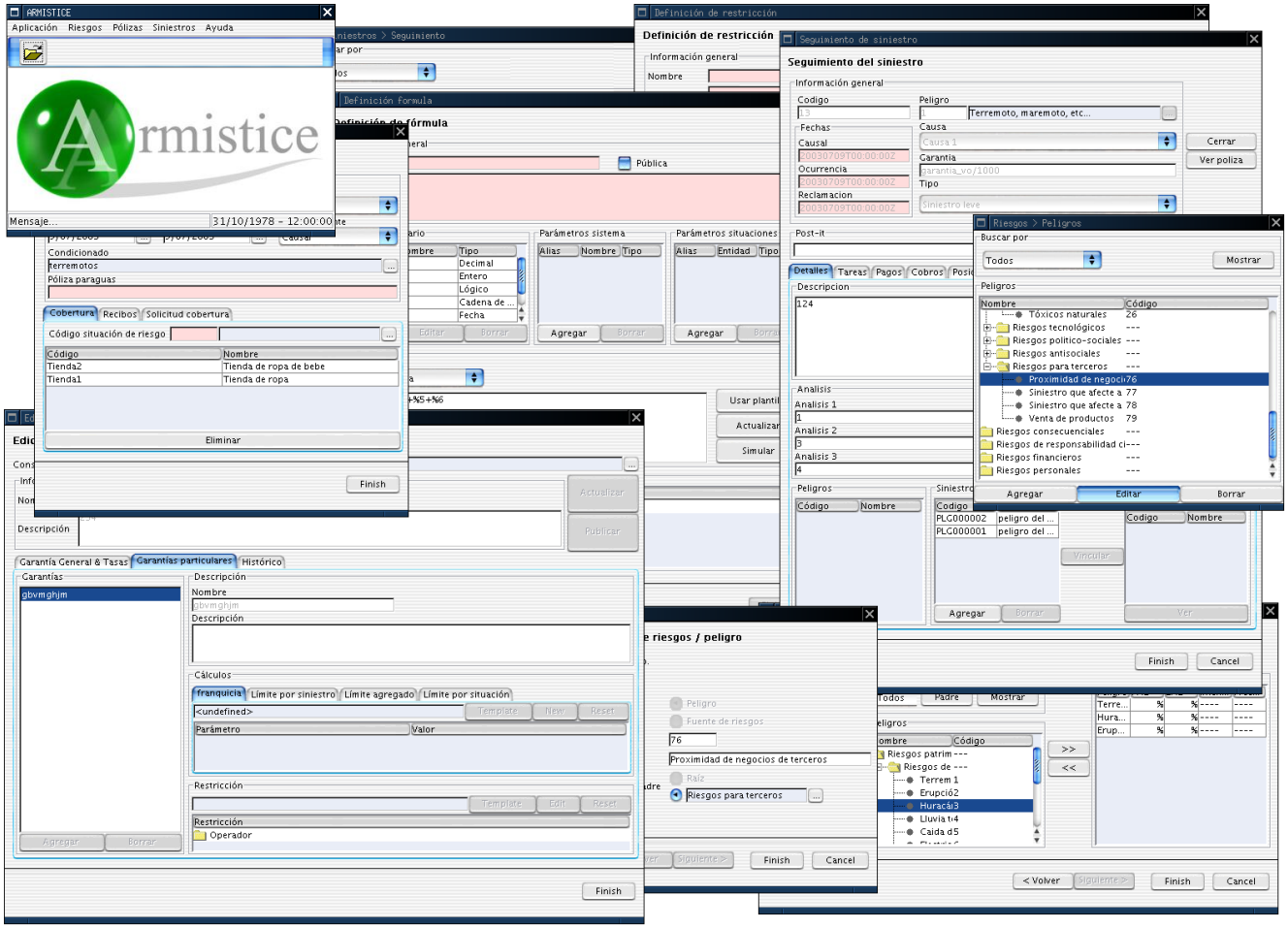

Figure 2: Some windows in the graphical interface (Client tier)

4.1 Object Handling

As seen, for model design purposes, the server must manage business *objects*. Even though, Erlang is not an object-oriented language, we can easily program its concepts in a number of different ways, such as the one presented in [4]. Some of these concepts like modularity, concurrency, dynamic memory handling..., are already included in Erlang. For the concept of object class, we usually choose to map each class to an Erlang module implementing the interface and behaviour of that class.

Regarding the implementation of the concept of object state we can remark two approaches representing those states:

- As explicit data structures.
- As Erlang processes.

With the first approach, each class instance is associated with a data structure that represents its members, both state and methods. In this case, if a method call is invoked over an object, a method (function) is evaluated with the actual data structure that models the object as an argument.

A simple example is the following. Suppose the class `classA` with the attributes `attributeA`, `attributeB`, and `attributeC`. The Erlang module skeleton that implements this class can be:

```
-module(classA).
-export([methodOne/n]).

-record(classA, {attributeA, attributeB, attributeC}).

methodOne(State, ... method parameters ...) ->
  ... do something ...,    % State is a classA record
  {ok, NextState, Result}.
```

Simplicity, low resource usage (one object is a record or a list of records) and the fact that objects are really inmutable (Erlang data structures are non-destructive) are the main advantages of the use of explicit data structures to implement objects in Erlang. However, the simplicity of this approach also has some disadvantages. Using records to implement object breaks the encapsulation principle because we have to explicitly manipulate object state between messages. This is a very annoying requirement and could be the source of many programming bugs. The next code section shows clearly this drawback:

```
{ok, StateTwo, Result} = classA:methodOne(StateOne, ...)
...
{ok, StateThree, OtherResult} = classA:methodTwo(StateOne,...)
```

Using the second approach, each class instance is mapped to an Erlang process. Then, the object state is implicitly encapsulated into the Erlang process state. From client's

point of view, an object is now modeled as a process PID (its *object identifier*), and sending a message to an object is, actually, sending a message to the process object. The following example shows our class `classA`, now implemented using the processes approach.

```
-module(classA).
-export([new/0, methodOne/n]).

-record(state, {attributeA, attributeB, attributeC}).

new() ->
  spawn(?MODULE, dispatch, [#state{}]).

dispatch(State) ->
  receive
    {call, Pid, Method, Args} ->
      {ok, NextState, Result} = ?MODULE:Method([State | Args]),
      Pid ! {response, Result},
      dispatch(NextState)
  end.

methodOne(State, ... method parameters ...) ->
  ... do something ...,
  {ok, NextState, Result}.
```

And it would be used as follows:

```
Object = classA:new(),
Object ! {call, self(), methodOne, [... method parameters ...])}
```

This second solution is probably a more natural way of mapping many of the object orientation principles: it provides object state encapsulation, object identity is unique (as long as processes identifiers are unique) and this kind of objects are really concurrent objects. However, there is a drawback, since this approach is more resource consuming than the former, specially if there are many small objects in the system.

Focusing in our system, we must deal with many *value objects* [7]. This kind of objects, which map database objects, are quite simple since their behaviour is mainly constrained to `get` and `set` methods. The simplicity, short life cycle and high number of instances in the system suggest the use of explicit data structures to implement them.

On the other hand, sometimes we find the necessity of implement some classes based on the *singleton* [6] design pattern (the database connection manager, monitoring processes...). This pattern ensures the existence of only one instance of the class at any time, and provides a global point of access for the whole system. Because these objects are large-grain and since they are long life cycle objects, they are implemented using the second solution (one object as one process). Moreover, the global access point is easily implemented using the Erlang process registration facilities. Then, our effort is notably reduced since we have established a direct mapping between the singleton pattern and the generic server Erlang behaviour.

```
-module(classC).
-behaviour(gen_server).

handle_call({methodA}, {Pid,_}, State) -> ...
handle_call({methodB, arg}, _, State) -> ...
```

4.2 Dealing with Object Representation across Tiers

Another issue to solve is the heterogeneity of the object representation across the application: client (Java), middleware (XML-RPC), server (Erlang) and storage (relational database). To manage this complexity, object identifiers are composed of two parts: the *object key* (persistent storage key) and its *class* (type). Thus, each subsystem has enough information to identify an object with its own representation, as shown in figure 3.

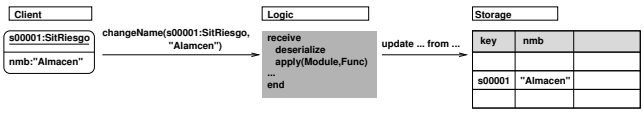

Figure 3: Use of object IDs

4.3 Searching for Relevant Insurances

One of the key use cases is the introduction into the system of a new loss or damage that might be covered by a insurance policy. The system must search for policies that are *relevant*, that is, that might cover the insured resource against the risk that has produced the damage. In order to accomplish that, clauses in a policy (warranties) are modeled as constraints expressions that can be nicely defined using Erlang data structures. A quite simplified view is shown as follows:

```
constraint ::= {danger, Danger}
             | {literal, String}
             | {negation, Constraint}
             | {all, [Constraint]}
             | {any, [Constraint]}
             | ...
```

When searching for the policy, each of its clauses is matched against the contingent event that has produced the damage. Pattern-matching is quite useful for defining this operation. In this simplified case all the information of the contingent event is summarized as the main danger that has caused it.

```
%% relevant: Danger x Constraint --> true | false | String

relevant(P, {literal, String}) -> String;
relevant(P, {danger, P}) -> true;
relevant(P, {danger, Q}) -> false;
relevant(P, {negation, C}) -> do_not(relevant(P, C));
relevant(P, {any, Cs}) -> lists:foldl( fun(A,B)-> do_or(A,B) end,
                                       false,
                                       [ relevant(P, C) || C <- Cs]);
relevant(P, {all, Cs}) -> lists:foldl(fun(A,B)-> do_and(A,B) end,
                                       true,
                                       [ relevant(P, C) || C <- Cs]).

do_not(Op) when atom(Op) -> not Op;
do_not(Op) when list(Op) -> "NOT " ++ Op.

do_or(OpA, OpB) when atom(OpA), atom(OpB) -> OpA or OpB;
do_or(true, OpB) when list(OpB) -> true;
do_or(false, OpB) when list(OpB) -> OpB;
do_or(OpA, OpB) when list(OpA), list(OpB) -> OpA ++ " OR " ++ OpB;
do_or(OpA, OpB) -> do_or(OpB, OpA).

do_and(OpA, OpB) when atom(OpA), atom(OpB) -> OpA and OpB;
do_and(true, OpB) -> OpB;
do_and(false, OpB) -> false;
do_and(OpA, OpB) when list(OpA), list(OpB) -> OpA ++ " AND " ++ OpB;
do_and(OpA, OpB) -> do_and(OpB, OpA).
```

Now, if the clauses of a given policy are represented as a list of constraints, we can filter out the clauses relevant for a given contingent event using a list comprehension:

```
relevant_clauses(Danger, {Policy, Clauses}) ->
       [ ClauseId || {ClauseId, ClauseConstraint} <- Clauses,
                     relevant(Danger, ClauseConstraint) ].
```

We can see the evaluation of a restriction as a simplification of a expression tree. For example, the following simple restricction,

```
{all , [{danger , earthquake}, {literal , "Intensity less than 3"}]}
```

when evaluated with the danger `'earthquake'` as input the result will be `"Intensity less than 3"`.

Of course, relevant checks are more complicated, including a richer language with more expressions types, more information from the contingent event, and the result is not a boolean value, but a richer one including certainty factors. Erlang is very well suited to define such rule-based algorithms.

4.4 Database Access Patterns

The access to the persistent storage is carried out using some well-known design patterns [7, 6] and the ODBC technology included in the Erlang OTP library. The *Facade* pattern provides a unified interface to a set of interfaces in a subsystem, simplifying the access to the database. In order to do so, a `db_facade` module is implemented as a generic server (a standard Erlang behaviour) which responds to a request to the storage.

```
-module(db_facade).
-behaviour(gen_server).

handle_call({connect}, {Pid, _}, State) -> ...
handle_call({commit, ConnRef}, _, State) -> ...
handle_call({rollback, ConnRef}, _, State) -> ...
```

The *DAO* pattern (*Data Access Object*) is used to decoupling the business logic from the access logic to the database. Hence, the data source can be changed easily. For each class representing a business object, a DAO module is implemented. For example, for the *RiskSituation* class this module defines access functions as presented:

```
-module(sit_riesgo_dao_fd).
%% Find RiskSituation object with cdg SitCdg
findByCdg(ServerPID, SitCdg) -> ...
%% Build list of RiskSituation objects
buildListOfSits(ServerPID,[]) -> [];
buildListOfSits(ServerPID,[ SitID | T ]) -> ...
```

The patterns *Session Facade* and *Business Delegate* allow the encapsulation of use cases logically related (searching for set of data, sequence of queries to the database, and so on). For example, the implementation of the accessor for the class *RiskSituation* looks like this:

```
-module(risk_situation).

create(RiskSituationVO, AttrProps, EntityProps) ->
    {ok, ServerPID} = db_facade:connect(),
    db_facade:begintrans(ServerPID),
    {ok, NewRiskSituationVO} = ?DAO:create(ServerPID, RiskSituationVO),
    {ok, KeyRiskSituation} = sit_riesgo_vo:getID(NewRiskSituationVO),
    KeysAttrProps =
        do_create_props(ServerPID,AttrProps, KeyRiskSituation),
  ...
    db_facade:endtrans(ServerPID),
    db_facade:disconnect(ServerPID),
    {ok, NewNewNewRiskSituationVO}.
```

4.5 Fault Tolerance Support

In order to get fault tolerance in the server tier, the main processes are placed into a supervision tree. Each tree node corresponds to an Erlang process. When a child process terminates, its parent is notified and it is supposed to take the appropriate actions to recover from the failure (for example, spawning a new process in a different node). At the root of the tree, a supervisor node is devoted to manage abnormal terminations. Tree supervision is an Erlang behaviour, so its implementation is straightforward.

Additionally, the internal state is replicated over the cluster of nodes and the persistent data is stored in a relational database that can be distributed (e.g. Oracle parallel server).

5. CONCLUSIONS

Some experiences of using the concurrent functional language Erlang to implement a client/server application have been presented. The project represents a real-world application of the Erlang language, currently in beta-testing stage, and we expect the system to be in production stage by the end of 2003.

Both the programming paradigm and the language have been a key success factor. The use of abstraction in the form of functional patterns helped reduce the programming effort to evolve the prototypes during the iterative development process. The available libraries and the built-in concurrent programming constructs of Erlang have simplified the development of a robust server. Despite our initial concerns, the need for integration within an object-oriented design was done rather smoothly. However, the lack of a static type system, a structured module system and design-by-contract support complicates the development at large and they should be considered interesting improvements to the Erlang language.

Nevertheless, the most expensive component of the system is the client tier (which, by the way, was not developed using a functional language). A generic client built from XML specifications, as shown in [2] could be very important to reduce the development cost of the client interface.

6. ACKNOWLEDGEMENTS

Special thanks to Javier Losada for sharing his expertise about the application domain and being such a nice unusual user.

7. REFERENCES

[1] Erlang otp documentation. http://www.elang.org/.

[2] Uiml website. http://www.uiml.org/.

[3] Xml-rpc website. http://www.xmlrpc.org/.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[5] J. W. Cooper. *Java Design Patterns: A Tutorial*. Addison-Wesley, Reading, MA, USA, 2000.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[7] F. Marinescu. *EJB Design Patterns. Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., 2002.

[8] R. Orfali, D. Harkey, and J. Edwards. *Client/Server Survival Guide*. John Wiley & Sons, third edition edition, 1999.