



UNIVERSIDADE DA CORUÑA

ARMISTICE: An Experience Developing Management Software with Erlang

David Cabrero, Carlos Abalde, Carlos Varela, Laura Castro

PLI'03: Principles, Logics, and Implementations of High-Level Programming Languages

2nd ACM SIGPLAN Erlang Workshop

Uppsala, Sweden, 25-29 August 2003

MADS Group - LFCIA Lab - Computer Science Department

University of A Coruña, SPAIN

`cabrero@udc.es, {carlos, cvarela, laura}@lfcia.org`

Outline

- Introduction
- Architecture design
- Erlang and OO
- Conclusions

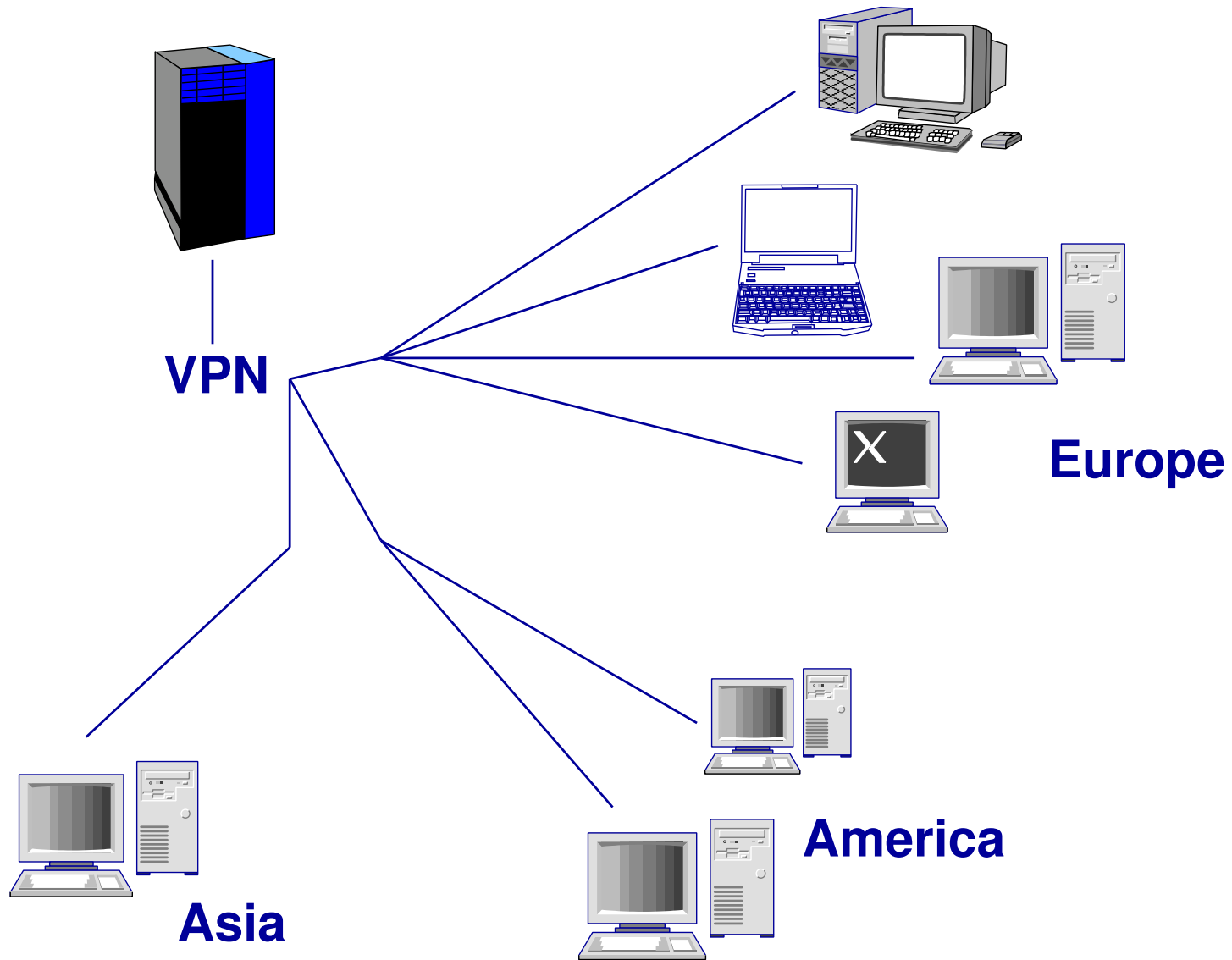


Introduction

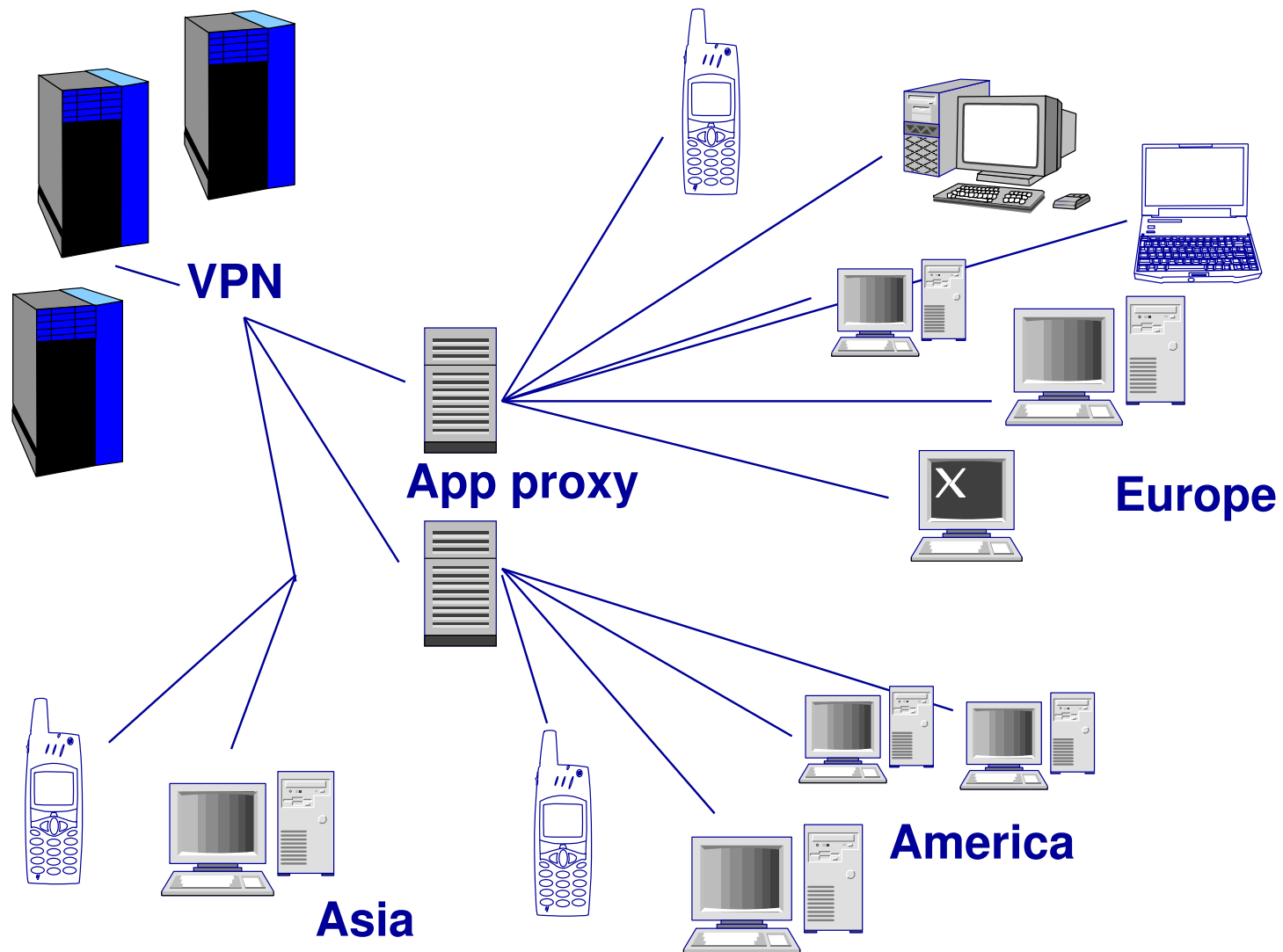
- This work is about our experience developing **ARMISTICE** (Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures).
- Risk management information system (Insurances)
 - For a large international company (holding enterprise)
 - Distributable, complex business logic, heterogeneous data
 - Applications in the market don't fit the expectations
- Requirements:
 - Client/Server
 - Scalability
 - Reliability
 - Multiplatform
 - Distribution across Virtual Private Network
 - Multiprotocol Middleware: standard RPC protocols



Introduction > Deployment



Introduction > Deployment (next)



Introduction > The challenge

- Software engineering methods closer to other programming languages: specifically UML and OO analysis and design
- Managing objects and relational databases with Erlang

Introduction > Why Erlang ?

Why we mixed Erlang with OO inside an information management application ?

- ✓ We want it to work, so we used a declarative language
- ✓ We work with Erlang, so we wanted to try it and see the result
- × It is this week's GOL
- × It is on hype
- × It was a requirement
- ✓ Behaviours are good, design patterns are also good, so let's put them together



Introduction > Why Erlang ?

Why we mixed Erlang with OO inside an information management application ?

- ✓ We want it to work, so we used a declarative language
- ✓ We work with Erlang, so we wanted to try it and see the result
- × It is this week's GOL
- × It is on hype
- × It was a requirement
- ✓ Behaviours are good, design patterns are also good, so let's put them together
- ✓ Somebody told us it was no possible



Introduction > The domain

- The system has to manage claims for accidents that happen in shops and resources all over the world, including:
 - Accident management.
 - Activity tracking (payments, invoices, repairs, . . .)
 - Modelling of the contracted policies.
 - Supporting decision to select the most suitable warranties.

- Knowledge elicitation (ala expert systems way)
 - Hopefully, UML class diagrams were quite useful

- Subsystems
 - Risks Subsystem. Metainformation about risk situations and their groups.
 - Policies Subsystem. Modelling of policies. We have defined a language of logic-expressions and formulas representing the conditionals of policies.
 - Accident Subsystem. Daily used by users all over the world.

Architecture design > 3 tiers client/server

Storage Relational Database (PostgreSQL)

- Mature technology.
- “Well suited” to our data.
- It was a requirement.

Client Java/Swing

- “Thin” Client: mostly GUI, no business logic.
- Multiplatform (Windows, Linux, . . .).

Middleware XML-RPC

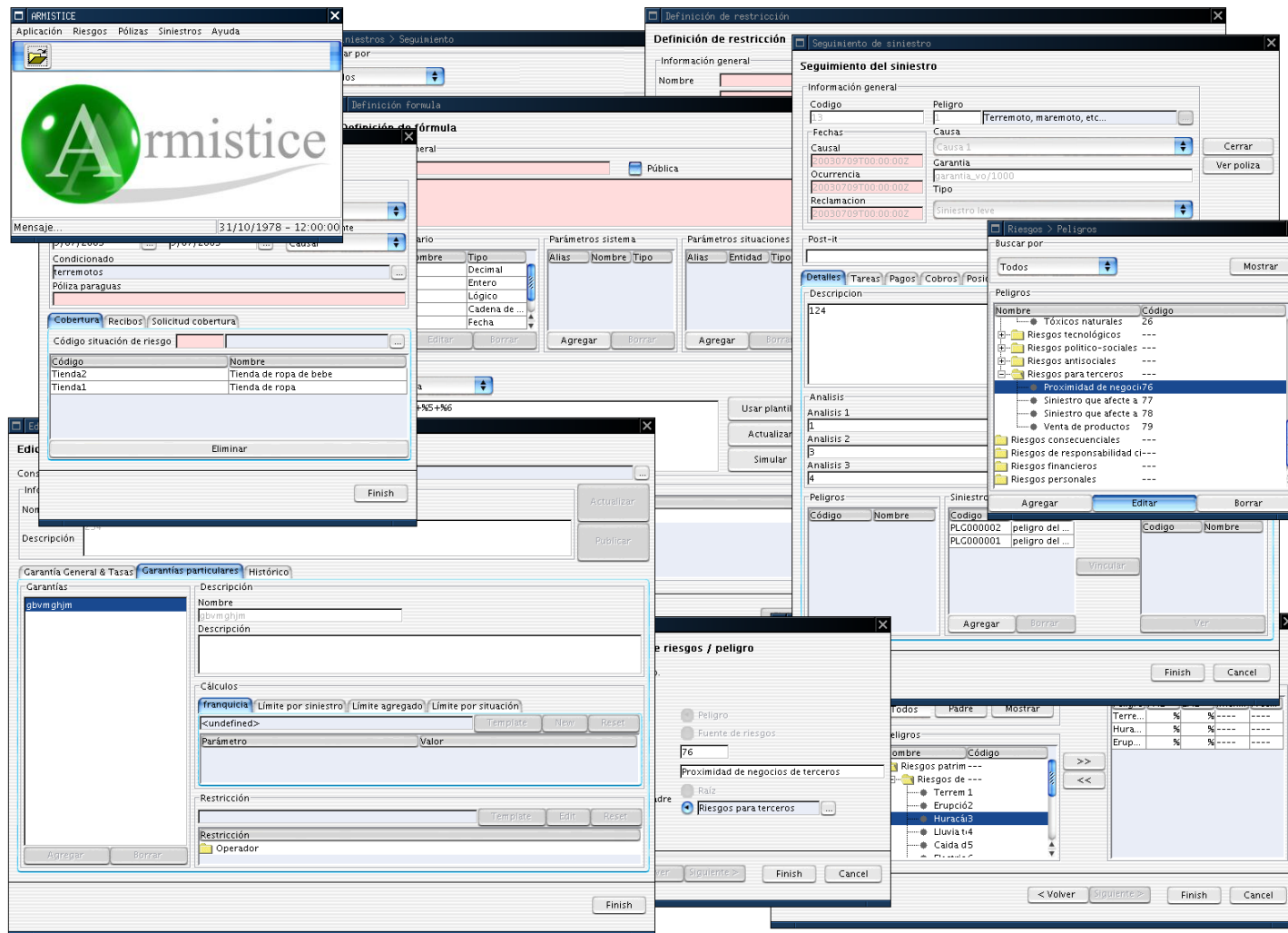
- Known and simple standard.
- Transport over HTTP.
- Replacements are available, i.e.: SOAP

Server Erlang

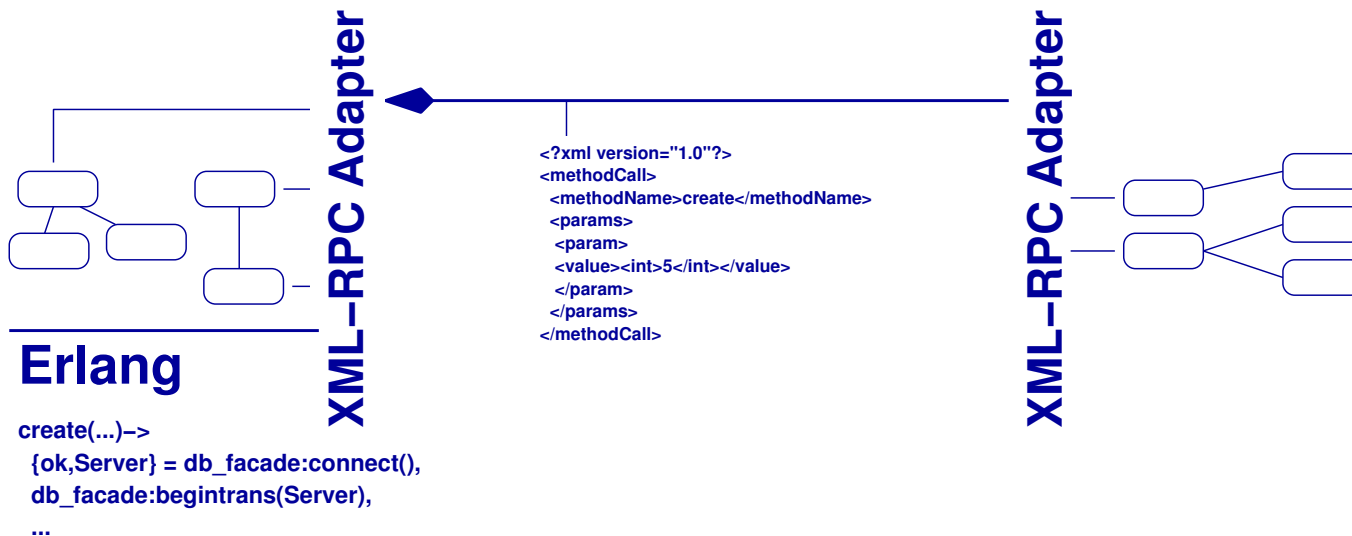
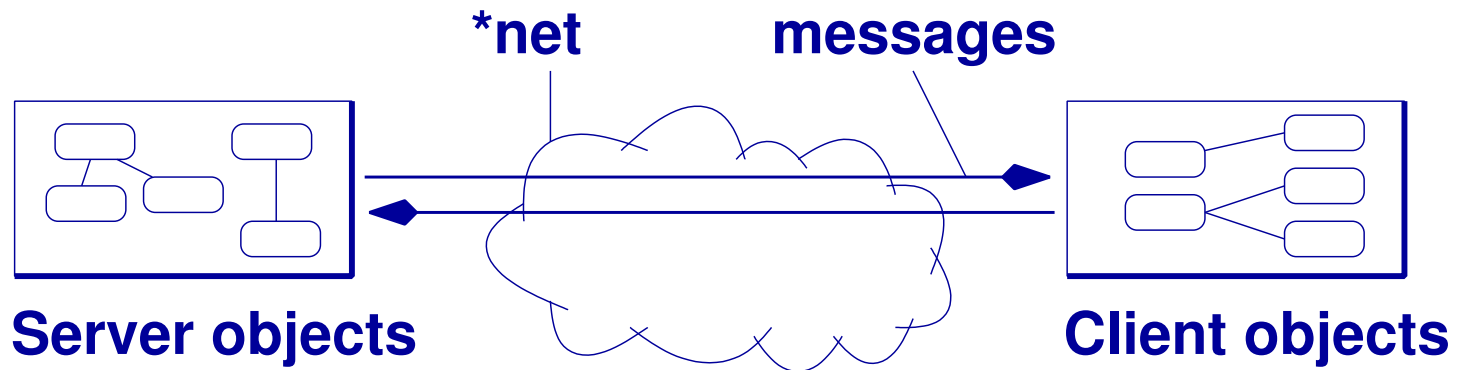
- We already said why



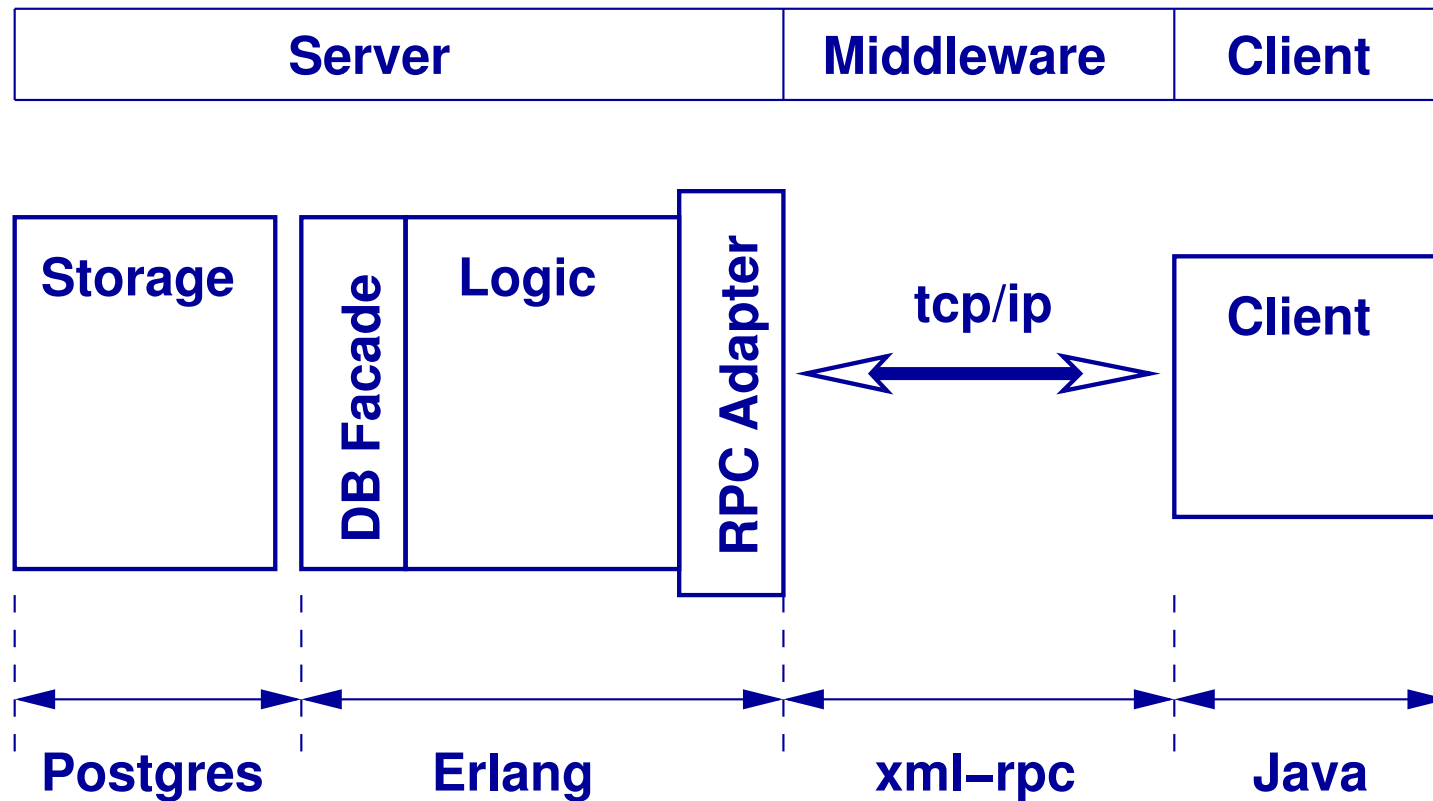
The client > screenshots



Erlang and OO > How ?



Erlang and OO > How ? (cont.)



Some patterns:

- Adapter. Convert the interface of a class into another interface clients expect.
- Facade. Provide a unified interface to a set of interfaces in a subsystem.

Erlang and OO > How ? (cont.)

Explicit state

- The state of an object is a explicit data structure (record).

```
-module(classA).
```

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

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

```
methodOne(State, ... method parameters ...) ->
```

```
    ... do something ...,    % State is a classA record  
    {ok, NextState, Result}.
```

Implicit state

- The state of an object is the state of a process.

```
-module(classC).
```

```
-behaviour(gen_server).
```

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

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

Erlang and OO > Design Patterns

- Both **explicit state** and **implicit state** have advantages and disadvantages.
- Solution: each design pattern has a preferred approach. Examples:

Singleton pattern Use **implicit state**.

- Ensure a class only has once instance, and provide a global point of access to it.
- Example: database facade.

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

Value object pattern Use **explicit state**.

- Objects that map database objects.
- Behaviour limited to **get** and **set** methods.

```
-module(calculo_vo).  
-export([new/2, getID/1, getForm/1, getValues/1, setForm/2, setValues/2]).  
-export([metaInfo/0]).  
-record(calculo_vo, {oid, for, vals}).
```



Erlang and OO > Design Patterns (cont.)

More patterns: **Data Access Object (DAO)**

- Used to decoupling the business logic from the access logic to the DB.

```
-module(calculo_dao).
create(ServerPID, CalculoVO) ->
    {ok, Generator} = entity_identifier_generator_factory:getGenerator(),
    {ok, CalID} = erlang:apply(Generator, genID, [ServerPID, "calculos"]),
    {ok, CalFor} = calculo_vo:getForm(CalculoVO),
    {ok, CalVals} = calculo_vo:getValues(CalculoVO),
    % we assume value creation to be done by corresponding DAO
    Query = "INSERT INTO calculos (cal_oid, cal_for) VALUES ( " ++
        CalID ++ ", " ++ lib:a2l(CalFor) ++ " )",
    case db_facade:process(ServerPID, Query) of
        {updated, NRows} ->
            if
                (NRows /= 1) ->
                    {error, internal_error, internal_error};
                true ->
                    calculo_vo:new(CalID, CalFor, CalVals)
            end;
        {error, foreign_key_not_found, ErrMsg} ->
            {error, formula_not_found, CalFor};
        {error, ErrFunc, ErrMsg} ->
            {error, ErrFunc, ErrMsg}
    end.
end.
```



Erlang and OO > Design Patterns (cont.)

More patterns: **Accessor**

- Encapsulate a use case's business logic in one transaction.

```
-module(calculo_accessor).
```

```
update(KeyCalculo, KeyFormula) ->
    {ok, ConnRef} = db_facade:connect(),
    db_facade:begintrans(ConnRef),
    {ok, CalculoVO} = calculo_dao:find(ConnRef, KeyCalculo),
    {ok, KeyOldFormula} = calculo_vo:getForm(CalculoVO),
    {ok, NewCalculoVO} = calculo_vo:setForm(CalculoVO, KeyFormula),
    ok = calculo_dao:update(ConnRef, NewCalculoVO),
    ok = valor_dao:removeByCal(ConnRef, KeyCalculo),
    {ok, _} = valor_dao:create_skeletonByForm(ConnRef, KeyCalculo, KeyFormula),
    case KeyOldFormula of
        KeyFormula -> ok;
        _           -> ok = formula_dao:cremove(ConnRef, KeyOldFormula)
    end,
    {ok, NewNewCalculoVO} = calculo_dao:find(ConnRef, KeyCalculo),
    db_facade:endtrans(ConnRef),
    db_facade:disconnect(ConnRef),
    {ok, NewNewCalculoVO}.
```

Erlang and Erlang > The bussines logic

Searching for relevant policies.

- Clauses in a policy (warranties) are modelled as constraint expressions:

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

- Each clause is matched against the contingent event that has produced the damage.
- Pattern-matching is quite useful for defining this operation.

```
%% 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]).
```



Conclusions and Future Work

- A good experience implementing a client/server application with Erlang.
- Real-world application of the Erlang language.
- Successful mix of OO and Erlang.
- Developing the client tier (GUI) is much more expensive than its server counterpart.
- The customer has proposed a continuation of the project !



Conclusions and Future Work

ErlangWorkshop:thanks().