

ARMISTICE: Una experiencia desarrollando software de gestión con Erlang*

Laura Castro, Víctor M. Gulías, David Cabrero, Carlos Abalde, Carlos Varela

Departamento de Computación, Universidad de A Coruña
Campus de Elviña s/n, 15071 A Coruña

email: {laura,gulias,carlos,cvarela}@lfcia.org, cabrero@udc.es

Resumen En este trabajo presentamos nuestra experiencia en el desarrollo e implementación de una aplicación vertical clásica, un sistema de información para la gestión de riesgos, empleando el lenguaje funcional concurrente Erlang. Hemos encontrado las técnicas tradicionales poco satisfactorias debido en gran parte a las interacciones propias de la arquitectura Cliente/Servidor desplegada y debido a la gran complejidad de la lógica de negocio presente en la aplicación. Así, la naturaleza del problema sugiere un diseño iterativo. Al mismo tiempo el uso de abstracciones (patrones funcionales) y la composicionalidad (composición tanto funcional como concurrente) han resultado ser factores clave a la hora de reducir el esfuerzo necesario para adaptar el sistema a los continuos cambios de los requerimientos. A pesar de nuestros temores iniciales, la distancia entre los paradigmas clásicos de la ingeniería del software y los paradigmas funcionales ha sido cubierta con éxito.

Palabras clave: Programación funcional, computación distribuida, programación concurrente, patrones de diseño, lógica de negocio, arquitectura cliente/servidor

1. Introducción

En este artículo presentamos nuestra experiencia en el diseño e implementación de una aplicación cliente/servidor en tres capas a la que hemos llamado ARMISTICE por *Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures*. La aplicación es un sistema para la gestión de riesgos diseñado para su uso en una multinacional y su despliegue está previsto para finales del 2003. La principal novedad del sistema es el uso de un lenguaje de programación declarativo, el lenguaje funcional concurrente Erlang [AVWW96]. Para cumplir los requisitos de fiabilidad y flexibilidad, la parte servidor de la aplicación deber poder ser distribuida, bien en un cluster de componentes de consumo o bien a lo largo de una red privada virtual, siguiendo la distribución de atribuciones propia de un holding de compañías de ámbito internacional. Además, la propia lógica de negocio de la aplicación tiene un gran nivel de complejidad y debe manejar diferentes tipos de objetos heterogéneos que representan riesgos, reglas de negocio que modelan las exposiciones de dichos riesgos, y restricciones de aplicabilidad sobre las pólizas que los cubren. El uso de Erlang ha simplificado

* Financiado parcialmente por el MCyT TIC 2002-02859 y la empresa Alfa21 Outsourcing S.L. (FUAC proyecto I+D 2/79)

notablemente el desarrollo de la parte servidor. Al mismo tiempo supone la aplicación del lenguaje a un dominio original y novedoso.

El uso de abstracciones (patrones funcionales) y la composicionalidad (tanto funcional como concurrente) han resultado ser factores clave a la hora de reducir el esfuerzo necesario para adaptar el sistema a los continuos cambios de los requerimientos. A pesar de nuestros temores iniciales, la distancia entre los paradigmas clásicos de la ingeniería del software y los paradigmas funcionales ha sido cubierta con éxito.

El resto del artículo se estructura de la siguiente manera: en primer lugar se introduce el dominio de aplicación y los principales requerimientos. A continuación, la sección 3 presenta una breve introducción al lenguaje de programación Erlang y su adecuación al proyecto. La sección 4 está dedicada a la arquitectura general del sistema. La sección 5 se centra en el uso de Erlang para el desarrollo del servidor desde el punto de vista de la incorporación de las tecnologías orientadas a objetos. Finalmente presentamos las conclusiones.

2. Aproximación al dominio de aplicación

El caso de estudio que presentamos, ARMISTICE, es el resultado de una petición expresa realizada por parte de un gran grupo empresarial, con sede en la provincia de La Coruña. Dicha compañía se decantó por un desarrollo a medida tras examinar las diferentes alternativas existentes en el mercado sin encontrar ninguna que se ajustase a sus necesidades, debido a la gran complejidad del dominio de aplicación. Una vez confirmado el desarrollo de la aplicación, nos encontramos con la problemática recurrente en la ingeniería del software:

- *Falta de conocimiento específico sobre el dominio de la aplicación.* La complejidad del dominio (seguros, riesgos, exposiciones, ...) impone la necesidad de un esfuerzo considerable de comunicación con el usuario experto. En este sentido, la elicitación de conocimiento se ha favorecido con el empleo de una metodología de desarrollo iterativa.
- La comunicación con el *experto en el dominio* también es problemática debido a su *falta de conocimiento sobre la Ingeniería del Software*. En este sentido, el uso de tecnologías orientadas a objetos, en concreto diagramas de clases UML, ha resultado ser muy efectivo.
- Necesidad de un *almacenamiento persistente*. Hoy en día los gestores de bases de datos relacionales SGBDR representan la tecnología más madura al respecto. Este requerimiento fue explicitado por el usuario, aunque se consideraron otras alternativas como la base de datos distribuida asociada a Erlang: Mnesia [mne].

Adicionalmente, nos encontramos con que la aplicación debía cumplir ciertos requerimientos tanto para cumplir la funcionalidad básica de su dominio de aplicación, como para ofrecer un valor añadido sobre las alternativas existentes en el mercado:

- *Flexibilidad y escalabilidad del servidor.* Tras una primera fase de despliegue de la aplicación, se espera que en una segunda fase aumente considerablemente el número de clientes asociados al servidor. En esta situación debe ser posible escalar

hacia arriba el servidor, añadiendo nuevos nodos servidor, probablemente en un cluster de componentes de consumo.

- *Fiabilidad del servidor.* Además de la corrección del código, se desea la inclusión de código tolerante a fallos y nodos de respaldo.
- *Multiplataforma.* Tanto el cliente como el servidor deben ser multiplataforma, aunque el sistema operativo preferido para el cliente es Microsoft Windows®. En lo que se refiere al servidor, el sistema operativo preferido es Linux, aunque no necesariamente sobre arquitecturas x86.
- *Distribución de clientes y servidores a través de una Red Privada Virtual.* La aplicación estará distribuida a lo largo de la Red Privada Virtual del usuario, a nivel mundial. Para paliar la pérdida de rendimiento asociada a una red de banda ancha como la descrita, es deseable la posibilidad de emplear proxys de aplicación, servidores replicados, ...
- *Middleware multiprotocolo.* El middleware se basará en protocolos de llamadas a procedimientos remotos (RPC) estándar y sencillos. De esta forma se facilita, en una segunda fase, la adición de nuevas plataformas cliente, probablemente PDAs.

2.1. El dominio de aplicación

Tal y como hemos indicado, ARMISTICE es un sistema de información de gestión de riesgos. Este sistema surge de la necesidad de una gran compañía internacional de gestionar las reclamaciones de accidentes que ocurren en sus tiendas y recursos ubicados en diversos países de Europa, América, Asia y África. Esta gestión incluye tanto la notificación de accidentes como el seguimiento de las actividades derivadas de su tratamiento (pagos, facturas, reparaciones, ...). A mayores, para dar soporte a esta gestión es necesaria la realización de otras actividades como el modelado de pólizas contratadas para dar cobertura a los posibles siniestros o los procesos involucrados en el sistema de ayuda a la decisión que se utilizará para la selección de las garantías más adecuadas en el momento en que un usuario introduce información sobre un nuevo incidente acaecido. Por *garantía* entendemos la parte específica (cláusula) de una póliza de seguros que cubre los daños que puedan producirse sobre una *situación de riesgo*, esto es, un elemento asegurado, cuando un peligro o conjunto de peligros actúan sobre ella.

El análisis del dominio nos lleva a identificar tres grandes subsistemas. Estos subsistemas son interdependientes, cada uno de ellos se apoya en los anteriores como elementos básicos para su funcionamiento y ofrece a los siguientes la funcionalidad propia de los objetos de negocio que define. A grandes rasgos, dichos subsistemas son:

- *Subsistema de Riesgos.* Define la noción de *exposición* basada en *peligros* y *entidades*. Maneja tanto la información como la metainformación relacionadas con estos conceptos.
- *Subsistema de Pólizas.* En base a las exposiciones definidas, modela las *pólizas de seguros*. Para el modelado de las *cláusulas* que componen una póliza se aplica el concepto de *restricción*. Así cada cláusula queda representada como una restricción expresada en base a las elementos que componen las distintas exposiciones. Además, mediante un lenguaje de fórmulas, se expresan los cálculos complejos de *franquicias* y *límites* contratados para cada póliza.

- *Subsistema de Siniestros*. A partir de las pólizas, sigue la evolución un *siniestro*, desde que se da de alta. Proporciona una guía útil para el seguimiento de tareas relacionadas, el cálculo de límites y franquicias implicados y la estimación de gastos asociados.

3. El lenguaje funcional distribuido Erlang

Erlang [AVWW96] es un lenguaje de programación funcional concurrente y distribuido desarrollado por Ericsson para aplicaciones de telecomunicaciones. El lenguaje no tiene construcciones que induzcan efectos colaterales a un almacenamiento implícito, salvo las comunicaciones entre threads (*procesos*, en terminología de Erlang). Erlang evalúa las expresiones de forma voraz, al contrario que otros lenguajes funcionales como HASKELL que usa evaluación perezosa.

Los valores en Erlang (i.e., expresiones no reducibles) van desde los números y *átomos* (constantes simbólicas, en la sintaxis de Erlang comienzan por minúscula) hasta las estructuras de datos complejas (listas, tuplas) y valores funcionales, que son tratados como ciudadanos de primera clase. Una función se define como un conjunto de ecuaciones, cada una estableciendo un conjunto de restricciones propio basados principalmente en la estructura de los argumentos (*pattern-matching*). El flujo de control iterativo se lleva a cabo mediante el empleo de la recursión. Las listas, la estructura de datos más importante en los lenguajes funcionales, se escriben `[a , b , c]` siendo `[]` la lista vacía, y una lista cuyo primer elemento es `X` y el resto de elementos `Xs` se nota como `[X | Xs]`. Por tanto, para definir una función que comprueba la pertenencia a una lista, podemos escribir:

```
member(X,[]) -> false;
member(X,[X|Xs]) -> true;
member(X,[Y|Ys]) -> member(X,Ys).
```

Las funciones se agrupan en módulos, pudiéndose exportar un subconjunto de estas funciones, declarando tanto el nombre como la aridad de la función, para su uso en otros módulos.

```
-module(mymodule).
-export([member/2]).
```

Por lo tanto, `mymodule:member(3, [1,2,3])` se evalúa al átomo `true`, mientras que la expresión `mymodule:member(5, [1,2,3])` se reduce a `false`. A falta de un sistema de tipado estático, como en muchos lenguajes funcionales modernos, las listas pueden contener valores heterogéneos como `[a , [], 1]`. Además de las listas, los programadores Erlang pueden usar *tuplas*, similares a las estructuras de C, que se construyen con una longitud arbitraria, pero finita, escribiendo `{A,B,...,C}`. Las listas y tuplas pueden contener cualquier valor Erlang válido, desde números y átomos hasta listas y tuplas o incluso valores funcionales. Los *registros* suponen una ayuda sintáctica para acceder a los elementos de una tupla por su nombre en lugar de su posición. Un registro `myrec`, con los campos `x` y `y`, se define como sigue:

```
-record(myrec, {x, y}).
```

`R#myrec.x` se usa para acceder al campo `x` del registro `R`, mientras que la operación `R#myrec{y=5}` es una actualización que construye un nuevo registro igual a `R`, pero con `y` ligado a 5.

Como otros lenguajes funcionales modernos, Erlang presenta características interesantes como funciones de orden superior (parámetros funcionales o resultados funcionales), creación de funciones on-the-fly, tipado dinámico, comprensión de listas y un potente colección de librerías que componen la Open Telecom Platform (OTP). La OTP incluye una base de datos distribuida, interfaces gráficos, un compilador ASN.1, un CORBA object request broker, e interfaces COM y Java, entre otros. Estas características han hecho de Erlang uno de las excepciones más notables en cuanto al éxito en aplicaciones del mundo real [Wad98].

3.1. Erlang concurrente

Lo que diferencia a Erlang de otros lenguajes funcionales es su soporte para la concurrencia y distribución. Las primitivas de concurrencia de Erlang recuerdan los cálculos formales como el CCS de Milner [MPW92] o CSP de Hoare [Hoa85]. La idea es sencilla: la programación concurrente es mucho más sencilla en un lenguaje funcional que en uno imperativo debido a la ausencia de efectos colaterales (sólo creación y comunicación de procesos).

Para crear un thread nuevo usamos la primitiva interna `spawn`. Dado un módulo `M`, una función `F` con aridad `N` exportada por dicho módulo, y una lista de argumentos `[A1, A2, ..., AN]`, la expresión `spawn(M, F, [A1, A2, ..., AN])` crea un nuevo proceso que calcula `M : F(A1, A2, ..., An)`. Una vez evaluado, `spawn` devuelve el *identificador de proceso*, `Pid`, del proceso ligero recién creado. Para permitir la comunicación entre procesos, se establecen dos primitivas de paso asíncrono de mensajes:

- *Envío asíncrono:*

```
Pid ! Msg
```

`Msg` es enviado al proceso `Pid` sin bloquear el proceso que lo envía. Si `Pid` existe, el mensaje se almacena en su *mailbox*. Cualquier valor Erlang válido puede ser enviado a otro proceso, incluyendo estructuras de datos complejas que contengan tuplas, listas, funciones, o identificadores de procesos.

- *Mailbox pattern matching:*

```
receive
    Pat1 ->Expr1;
    ...
    PatM ->ExprM
end
```

La construcción anterior busca en la mailbox del proceso un mensaje que encaje con alguno de los patrones `Pat1, ..., PatM` de forma secuencial. Si no existe tal mensaje, el proceso se bloquea hasta que llega. El resultado es la evaluación de `Expri` suponiendo que el patrón encajado sea `Pati`.

Usando estos constructores concurrentes, podemos definir un *servidor* como una función recursiva que recibe una petición de un cliente y manda una respuesta de vuelta. El *estado* del servidor se incluye explícitamente como un parámetro de la función. Por ejemplo, el siguiente código define un proceso que añade `Inc` al parámetro de la petición del cliente.

```

-module(inc_server).
-export([loop/1]).

loop(Inc) ->
  receive
    {From, X} -> From ! X+Inc,
                loop(Inc)
  end.

```

Para crear el servidor con $Inc=1$, debemos evaluar `spawn(inc_server, loop, [1])`. Obsérvese que, como parte de la petición, el cliente incluye su propio identificador para recibir la respuesta del servidor. También, nótese el uso del *operador secuencial*: E_1, E_2 evalúa E_1 (quizás realizando alguna comunicación), descarta el valor calculado y a continuación evalúa E_2 . El API del cliente puede ser definido como:

```

client(X) ->
  ServerPid ! {self(), X},
  receive
    Response -> Response
  end.

```

donde *ServerPid* es el identificador del proceso servidor, y *self* es una primitiva interna que devuelve el identificador del proceso en ejecución.

Podemos usar las funciones de orden superior para generalizar el patrón de servidor básico y evitar la repetición de la misma estructura en diferentes lugares. En este caso, el algoritmo para calcular la respuesta dada una petición se encapsula en una función $F : Request \times State \rightarrow Response \times State$, proporcionando al mismo tiempo el estado de la siguiente iteración.

```

-module(server).
-export([start/2, loop/2]).

start(F, State) -> spawn(server, loop, [F, State]).

loop(F, State) ->
  receive
    {From, Req} ->
      {Resp, NewState} = F(Req, State),
      From ! Response,
      loop(F, NewState)
  end.

```

Podemos especializar este esqueleto para construir el servidor `inc_server`:

```

server:start(fun (X,Inc) -> {X+Inc,Inc} end, 1).

```

3.2. Erlang distribuido

La extensión natural de Erlang a un entorno distribuido es permitir a los procesos residir en más de una máquina virtual Erlang (*nodos* en terminología Erlang) posiblemente ejecutándose en diferentes nodos físicos. Para crear un proceso en un nodo remoto debemos usar `spawn(Node, M, F, Args)`, añadiendo como parámetro extra *Node* que nos indicará la localización del nuevo proceso. Una característica positiva es que las construcciones concurrentes son semánticamente equivalentes en este entorno distribuido (aunque las comunicaciones entre procesos son menos eficientes).

3.3. Behaviours

El uso de un lenguaje funcional, en particular la identificación de abstracciones funcionales, es un factor clave en la simplificación del desarrollo. Además de las abstracciones funcionales clásicas, `map` o `fold`, es posible definir patrones de más alto nivel. Una fuente importante de tales abstracciones, provenientes del mundo de la Orientación a Objetos es la colección de patrones de diseño del *libro de los cuatro* [GHJV95]. Erlang, a través, de las librerías OTP, proporciona su propio conjunto de patrones llamados *behaviours*. Estos patrones se enfocan en el uso originario de Erlang (aplicaciones de telecomunicaciones) e incluyen, entre otros, *servidor genérico*, *supervisor*, *máquina de estado finito*, ... Usar un *behaviour* es tan simple como incluir al comienzo del módulo la declaración `-behaviour` e implementar las funciones del API correspondiente:

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

handle_call({f}, {msg}, State) -> ...
handle_call({g, Arg}, {PID,_}, State) -> ...
```

3.4. Adecuación al dominio de aplicación

La lógica de negocio de la aplicación ha sido programada usando un estilo funcional convencional. Por ejemplo, el siguiente fragmento de la construcción de un árbol de objetos usa la conocida abstracción `map`:

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

Tal y como hemos indicado, el uso de las abstracciones funcionales ha supuesto un factor clave en la simplificación del desarrollo. Para ilustrar esta idea, mostramos uno de los casos de uso más importantes: el soporte de ayuda a la decisión en la introducción de una nueva pérdida o daño para el cual hemos de determinar qué póliza lo cubre.

El sistema debe buscar las pólizas que sean *relevantes*, esto es, que puedan cubrir el recurso asegurado frente al riesgo que produjo el desperfecto. Para poder hacerlo, las cláusulas de una póliza (garantías) se modelan como restricciones (utilizando un lenguaje de expresiones) que pueden ser fácilmente definidas usando estructuras de datos Erlang. Una versión simplificada sería la siguiente:

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

Al buscar una póliza, cada una de sus cláusulas se compara con el evento contingente que produjo el daño. El *pattern-matching* es muy útil para definir esta operación. En este caso simplificado, la información del siniestro se reduce al peligro principal:

```

%% 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(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) when list(OpB) -> OpB;
do_and(false, OpB) when list(OpB) -> false;
do_and(OpA, OpB) when list(OpA), list(OpB) -> OpA ++ "AND" ++ OpB;
do_and(OpA, OpB) -> do_or(OpB, OpA).

```

Representando las cláusulas de una póliza como listas de restricciones, podemos filtrar las cláusulas relevantes para un evento contingente usando la comprensión de listas:

```

relevant_clauses(Danger, {Policy, Clauses}) ->
  [ ClauseID || {ClauseId, ClauseConstraint} <- Clauses,
    relevant(Danger, ClauseConstraint) /= false ].

```

Otro aspecto importante a la hora de emplear Erlang es la tolerancia a fallos. En el caso de ARMISTICE colocaremos los procesos principales en un árbol de supervisión. Cuando un proceso en el árbol termina de forma abrupta, el nodo padre es informado y deberá tomar las acciones pertinentes para que el sistema se recupere del fallo, por ejemplo creando un nuevo proceso. El árbol de supervisión es un behaviour Erlang por lo que su implementación es directa:

```

-module(armistice_sup).
-behaviour(supervisor).

init([]) ->
  SupFlags = {one_for_one, 4, 3600},
  {ok, {SupFlags,
        [child(config, []), child(monitor, []),
         child(file_logger, []),
         child(dao_fd_proxy, [categoria_dao_fd_proxy, categoria_vol]),
         child(dao_fd_proxy, [tipo_dao_fd_proxy, tipo_vol]),
         child(http_frontend, []), child(db_facade, [])]}}}.

```

Como vemos en el ejemplo, el comportamiento de un supervisor es configurable. En este caso establecemos que reorganice los procesos tan pronto fallan, a menos que los fallos se produzcan más de cuatro veces en un intervalo de 3600 segundos.

4. Arquitectura de ARMISTICE

ARMISTICE es una aplicación cliente/servidor en 3 capas, tal y como se muestra en la figura 1. Puesto que no consideramos las interfaces de usuario gráficas en Erlang suficientemente maduras en plataformas MsWindows, el lenguaje elegido para la capa cliente fue Java [Coo00] (Swing), quedando Erlang restringido a la capa del servidor.

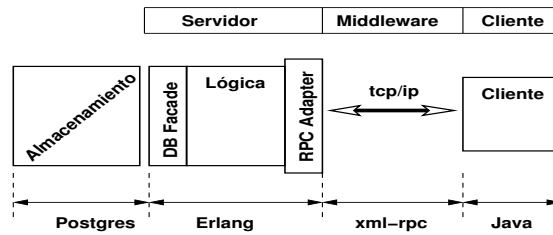


Figura 1. Arquitectura 3-tier de ARMISTICE

4.1. Cliente

El cliente es un cliente delgado como se presenta en [OHE99]: con ausencia total de lógica de negocio. Su funcionalidad se limita a representar información, validar la entrada de datos y realizar las correspondientes peticiones al servidor. La figura 2 muestra algunas instantáneas de las interfaces gráficas del cliente. A pesar de la (aparente) simplicidad de la capa cliente con respecto a la capa servidor, su desarrollo ha sido mucho más costoso en tiempo y recursos.

4.2. Middleware

Uno de los requerimientos iniciales era emplear un protocolo de middleware sencillo y estándar. En concreto, hemos optado por XML-RPC [xml]. Entre las ventajas de esta aproximación citaremos: bajo acoplamiento entre servidor y clientes, el transporte de datos se realiza sobre HTTP, facilitando el paso a través de firewalls, gateways VPN o la introducción de capas de encriptación como SSL.

4.3. Servidor

El servidor, a su vez, también está estructurado en capas:

- *El adaptador middleware.* Recibe peticiones de los clientes, siguiendo el protocolo XML-RPC, y las convierte al API de la capa de lógica de negocio.
- *La lógica de negocio.* La lógica de negocio propiamente dicha, independientemente de la comunicación con los clientes o la persistencia de los datos.
- *El acceso al almacenamiento persistente.* Comunicación con el almacenamiento, típicamente una base de datos, y la serialización y deserialización de los datos.

4.4. Almacenamiento

Como ya se ha mencionado anteriormente, se ha optado por una base de datos relacional. En concreto PostgreSQL, la cual se accede a través de una interface ODBC. Dicha interface forma parte de la librería OTP de Erlang.

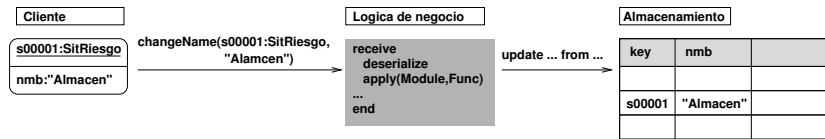


Figura 3. Uso de IDs de objetos

nombre del registro será el nombre de la clase y el nombre de sus elementos el de los atributos de la clase. Respecto a la herencia, representaremos el estado de un objeto como una lista de los estados específicos de cada una de sus clases ancestro. Así el estado de un objeto de la clase Cuadrado que deriva de la clase Figura viene dado por:

```
[ EstadoEspecificoCuadrado, EstadoEspecificoFigura ]
```

y una posible implementación del módulo Cuadrado:

```
-module(Cuadrado).
-record(?MODULE, {lado})

new(Lado) ->
  {ok, EstadoSuperclase } = figura:new(),
  {ok, [#?MODULE{lado = Lado} | EstadoSuperClase ]}.
```

Además necesitaremos un mecanismo de dispatch dinámico que maneje la invocación de métodos heredados. Para ello la lista de estados específicos contiene implícita la lista de ancestros del objeto. Recorreremos dicha lista buscando el método a invocar:

```
dispatch([], Metodo, RestoParametros) ->
  throw(undefined_method);
dispatch(Estado=[Especifico|Heredado], Metodo, RestoParametros) ->
  Clase = erlang:element(1, Especifico),
  case catch apply(Clase, Metodo, [Estado|RestoParametros]) of
    {'EXIT', {function_clause, [{Clase, Metodo, _} |_]}} ->
      dispatch(Heredado, Metodo, RestoParametros);
    {abstract_function} ->
      throw(abstract_function);
    {ok, Resultado} ->
      {ok, Resultado};
    _ ->
      throw(unexpected_behaviour)
  end.
```

Finalmente hemos de considerar si el estado de cada objeto se maneja como estado *implícito* o *explícito*. Como estado explícito, cada instancia de una clase se asocia con una estructura de datos como la que hemos definido anteriormente. De esta forma, cada invocación a un método requiere que el estado se pase de forma explícita como un parámetro más. Por ejemplo, para la clase `classA` con atributos `attributeA`, `attributeB`, y `attributeC`, y el método `methodOne`:

```
-module(classA).
-export([methodOne/n]).
-record(classA, {attributeA, attributeB, attributeC}).

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

Las principales ventajas son simplicidad, bajo consumo de recursos (sólo una lista de registros por cada instancia) y el hecho de que los objetos son inmutables (las estructuras de datos en Erlang son no destructivas). Por contra como desventajas destacaremos que no se garantiza el principio de encapsulación puesto que manejamos explícitamente el estado de los objetos. Esta es una característica bastante molesta, fuente de muchos errores de programación. El siguiente código ilustra esta idea:

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

En lo que respecta al uso de estado implícito, cada instancia de una clase se hace corresponder con un proceso Erlang, quedando su estado implícito en el estado del proceso. Destacaremos como ventajas que se garantiza la encapsulación, el paso de mensajes entre objetos es a todos los efectos un intercambio de mensajes, cada objeto posee identidad propia, dada por el PID (*Process Identifier*) del proceso correspondiente, y cada instancia se ejecuta concurrentemente con las demás. El siguiente ejemplo muestra la clase `classA` implementada según esta segunda aproximación:

```
-module(classA).
-export([new/0, methodOne/n]).
%% Uso:
%%   Object = classA:new(),
%%   Object ! {call, self(), methodOne, [... method parameters ...]}

-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}.
```

Como principal desventaja destacaremos el consumo excesivo de recursos, ya que cada instancia tiene asociada un proceso Erlang. Este problema es especialmente grave si en nuestro sistema existen muchos objetos de pequeño tamaño.

Nos resta, pues, decidir cual de las dos aproximaciones adoptar. Ya hemos indicado en apartados anteriores que el uso de ciertas abstracciones ha supuesto un factor clave en la simplificación del desarrollo de la aplicación. También recalcaremos que el uso de patrones de diseño [GHJV95,Mar02] es el objetivo y la implementación de un sistema objetual en Erlang, el medio. Así, emplearemos para cada patrón de diseño concreto la aproximación más conveniente. Ilustraremos esta idea mediante dos ejemplos.

En primer lugar, en nuestra aplicación, nos encontramos con el patrón *value_objects* [Mar02]. Esto nos lleva a manejar gran cantidad de objetos que mapean los objetos del almacenamiento. Este tipo de objetos es bastante simple ya que el único comportamiento se reduce a leer o establecer sus atributos. Dado este comportamiento simple y

el hecho de que es necesario crear una gran cantidad de instancias, en este caso hemos optado por implementar estos objetos usando un estado explícito.

En segundo lugar nos encontramos con la necesidad de implementar algunas clases según el patrón de diseño *singleton* [GHJV95]. Este es un patrón de creación que asegura la existencia de una única instancia de la clase, y proporciona un punto de acceso global a dicha instancia. Al precisar una única instancia eliminamos la desventaja del consumo de recursos y podemos adoptar la segunda aproximación implementando dicha instancia como un proceso Erlang. Además, para proveer un punto de acceso global a ella, emplearemos el mecanismo de registro de procesos de Erlang. Destacamos también que nuestro esfuerzo se reduce considerablemente estableciendo una correspondencia directa entre el patrón *singleton* y el behaviour *generic server* de Erlang.

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

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

5.1. Patrones de diseño en el acceso al almacenamiento

Al margen de la propia lógica de negocio, la capa de acceso al almacenamiento persistente es de vital importancia para el desarrollo del servidor. En esta capa se ha hecho un uso intensivo de los patrones de diseño presentados en [Mar02]. A continuación presentamos algunos ejemplos significativos. La figura 4 muestra las clases involucradas.

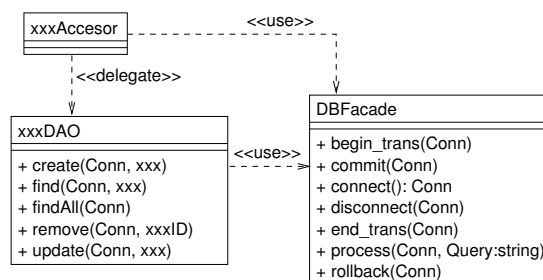


Figura 4. Patrones empleados en el acceso al almacenamiento

En primer lugar, el patrón *Facade* proporciona un interface unificado a un conjunto de interfaces de un subsistema, simplificando el acceso al mismo. Por ello, hemos implementado el módulo `db_facade` para el acceso al subsistema de acceso al almacenamiento. Esta clase es, además, un *singleton* por lo que queda:

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

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

Anteriormente comentábamos el uso de *value objects* para representar los objetos contenidos en el almacenamiento. Sin embargo, esto no es suficiente para desacoplar la lógica de negocio de la lógica de acceso al almacenamiento, y que los cambios en el almacenamiento sean transparentes al resto del sistema. Para ello empleamos el patrón *DAO (Data Access Object)*. Por cada clase que represente un objeto de negocio crearemos una clase DAO que encapsule la lógica de acceso al almacenamiento para los objetos correspondientes. Por ejemplo, para la clase `Calculo` crearemos `Calculo_Dao`:

```
-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.
```

Finalmente consideraremos la situación en que un *caso de uso* implica la colaboración de varios objetos de negocio, accediendo al almacenamiento. Si el proceso falla antes de finalizar el caso de uso, el almacenamiento podría quedar en un estado inconsistente, por lo que es necesario encapsular su lógica en una *transición*. Para ello son útiles los patrones *Session Facade* y *Business Delegate*. Por cada caso de uso tendremos una clase *accesor* encargada de gestionar la transición. Por ejemplo para el caso de uso actualizar un cálculo tenemos:

```
-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:remove(ConnRef, KeyOldFormula)
  end,
  {ok, NewNewCalculoVO} = calculo_dao:find(ConnRef, KeyCalculo),
  db_facade:endtrans(ConnRef),
  db_facade:disconnect(ConnRef),
  {ok, NewNewCalculoVO}.
```

6. Conclusiones

Hemos presentado nuestra experiencia empleando el lenguaje funcional Erlang para implementar una aplicación cliente/servidor. Creemos que es una experiencia positiva ya que el proyecto ha conseguido con éxito el desarrollo de una aplicación del mundo real empleando Erlang. La aplicación se encuentra actualmente en estado beta, esperándose su despliegue a finales del 2003.

Tanto el lenguaje como el paradigma de programación han sido factores clave para el éxito del proyecto. El uso de abstracciones en forma de patrones funcionales ha contribuido a reducir el esfuerzo de programación durante las fases de prototipado del proceso de desarrollo iterativo. Así mismo, la incorporación de técnicas de los paradigmas orientados a objetos se ha realizado sin grandes esfuerzos y, a su vez, también han sido factores claves en el éxito del desarrollo.

Sin embargo, como aspecto negativo de Erlang, destacaremos la falta de un sistema de tipado estático y la falta de soporte a las técnicas de diseño por contrato. Además, hemos encontrado que el componente de desarrollo más costoso es la capa de cliente (que coincide que no ha sido implementada usando un lenguaje declarativo). Probablemente un cliente genérico construido a partir de una especificación, tal y como se propone en [uim], reduciría el coste de desarrollo de este tipo de clientes ligeros.

7. Agradecimientos

Estamos especialmente agradecidos a Javier Losada por compartir su conocimiento como experto en el dominio de aplicación y por ser un usuario inusualmente agradable.

Referencias

- [AVWW96] Joe Armstrong, Robert Viriding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
- [Coo00] James W. Cooper. *Java Design Patterns: A Tutorial*. Addison-Wesley, Reading, MA, USA, 2000.
- [GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [Mar02] Floyd Marinescu. *EJB Design Patterns. Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., 2002.
- [mne] Erlang OTP documentation. <http://www.erlang.org/>.
- [MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.
- [OHE99] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. John Wiley & Sons, third edition edition, 1999.
- [uim] Uiml website. <http://www.uiml.org/>.
- [Wad98] Philip Wadler. Functional programming: An angry half dozen. *SIGPLAN Notices*, 33(2):25–30, February 1998. Functional programming column [NB. Table of contents on the cover of this issue is wrong.].
- [xml] Xml-rpc website. <http://www.xmlrpc.org/>.