

Programación de un caso de uso en Fisterra

Sergio Villar Senín

Igalia S.L. Ingeniería en informática y software libre
Gutenberg, 34B 2º, Polígono de A Grela – 15008 A Coruña
svillar@igalia.com

Resumen Fisterra es un proyecto de *software libre* cuyo objetivo es la creación de una plataforma para el desarrollo de aplicaciones de gestión empresarial. En la actualidad se encuentran disponibles a disposición de la comunidad el **framework** Fisterra y una implementación específica de un ERP para empresas de distribución conocida como Fisterra Distribution. El presente taller está dirigido a aquellas personas interesadas en utilizar la plataforma Fisterra y en especial para todas aquellas que deseen realizar desarrollos con esta tecnología. El taller se plantea, desde una orientación eminentemente práctica, con el objetivo de desarrollar un caso de uso típico de un ERP consistente en una ventana con un listado, y otra con un detalle de una de las entradas correspondientes a dicho listado. Para ello en un primer momento se realizará una instalación de la biblioteca y de la aplicación Fisterra Distribution. Posteriormente tras proporcionar una visión general sobre la arquitectura, necesaria para entender el desarrollo del caso de uso, se procederá a la presentación del caso de uso que se implementará. Debido a las limitaciones en el tiempo, el caso de uso a desarrollar será lo más sencillo posible y en muchas cuestiones aprovechará desarrollos previos a fin de poder tocar todos los pasos en el proceso de desarrollo sin entrar en complejidades técnicas. El taller será impartido tomando como base el sistema Debian GNU/Linux y las herramientas que este nos proporciona.

1. Introducción a Fisterra

1.1. El proyecto Fisterra

Fisterra ¹[1,2,3] es un proyecto de Software Libre que tiene como objetivo la creación de una plataforma de desarrollo de aplicaciones de gestión empresarial, y la implementación sobre ésta de soluciones verticales.

1.2. Diseño de la Arquitectura

Fisterra se articula según una arquitectura multicapa:

1. Cliente: aplicaciones GTK de ventanas, siguiendo el patrón MVC en dos capas.

¹ <http://www.fisterra.org>

2. Servidor: se compone de un conjunto de módulos que implementan los servicios de la interfaz y la lógica de negocio llamados EGBs² manejando objetos persistentes, los conocidos como *barnacles*[4].
3. `middleware` que conecta las diferentes capas. En concreto un despliegue de `Fisterra` necesita
 - servidor de BD. Sobre el se construye además un servicio de replicación.
 - servidor de nombres
 - servidor de autenticación (como LDAP)

2. Instalación y configuración

2.1. Instalación de `Fisterra Base`

En primera instancia deberemos instalar el paquete Debian correspondiente a `Fisterra Base`. Para ello incluimos el servidor `apt.igalia.com` en nuestro fichero `/etc/sources.lst` y lo instalamos con el comando `apt-get install libfisterra`.

2.2. Instalación de `Fisterra Distribution`

Posteriormente procederemos a descargar la versión en desarrollo de `Fisterra Distribution` del CVS mediante el comando

```
cvs -z3 -d :pserver:anonymous@cvs.igalia.com:/var/publiccvs  
co fisterra-distribution
```

por ejemplo en nuestro directorio `home`

2.3. Instalación de los paquetes de `ORBit` de `Igalia`

Mientras los desarrolladores de `ORBit` no acepten el parche de `lgalia` para el servicio de nombres, será necesario instalar los siguientes paquetes desde servidor `apt` de `lgalia`: `liborbit2`, `liborbit2-dev`, `orbit2` y `orbit2-nameserver`. También es posible la descarga desde la dirección

<http://community.igalia.com/twiki/bin/view/Fisterra/Orbit2Packages>

2.4. Configuración de `Fisterra Distribution`

Configuración del cliente y el servidor Podemos dejar los XML de configuración de cliente y servidor tal y como se proporcionan para nuestras pruebas, si se desea conocer más en detalle cada uno de los campos consúltese la documentación.

² *Enterprise-GNOME-Barnacle*

Configuración y compilación Si vamos a desarrollar un caso de uso de Fistera Distribution deberemos configurar las variables de localización de los ficheros para que se referan a los directorios donde nos hemos descargado el código. Para ello ejecutamos el comando `./local_autogen.sh` desde el directorio raíz de la instalación. Si todo ha ido bien tan solo resta ejecutar `make` para compilar el código y tener los ejecutables listos para nuestro primer test.

Creación de la BD Deberemos asimismo crear la BD necesaria para la capa de persistencia de objetos. Para ello necesitaremos la BD PostgreSQL. En primer lugar crearemos un usuario de la BD llamado `fistera` con el comando `createuser`. Una vez hecho esto creamos la BD con el comando `createdb -h localhost -U fistera fistera`. Una vez creada la BD creamos la estructura de tablas: `psql -h localhost -U fistera fistera <database/data_definition.sql` Por último con el fin de poder probar la aplicación añadiremos una serie de tuplas de ejemplo con `./src/server/tests/testInsertDatabaseSamples`

2.5. Probando la instalación

Para comprobar que hemos realizado todos los pasos correctamente ejecutaremos la aplicación Fistera distribution. En un primer paso comprobaremos que el servicio de nombres de Orbit está levantado. Si es así procederemos a arrancar en un primer momento el servidor de sesiones y posteriormente el servidor Fistera. Para ello ejecutamos en un terminal en el directorio `src/server` los comandos

```
./fistera2-distribution-session-server\&  
./fistera2-distribution-server}
```

Una vez levantado el servidor ejecutamos en otro terminal

```
./fistera2-distribution-client
```

desde el directorio `src/client`. Si todo va bien nos aparecerá una pantalla de autenticación que solicita un *login* y una *password* con unos valores por defecto; simplemente deberemos aceptar y nos saldrá la pantalla principal de Fistera Distribution que podemos ver en la figura 1.

3. Desarrollo paso a paso de un caso de uso

Esta es la parte destinada a la codificación de un caso de uso típico en un ERP basado en la tecnología Fistera. Si bien sería deseable mostrar un desarrollo desde cero, la limitación de tiempo hara necesario el empleo de código perteneciente a Fistera Distribution.

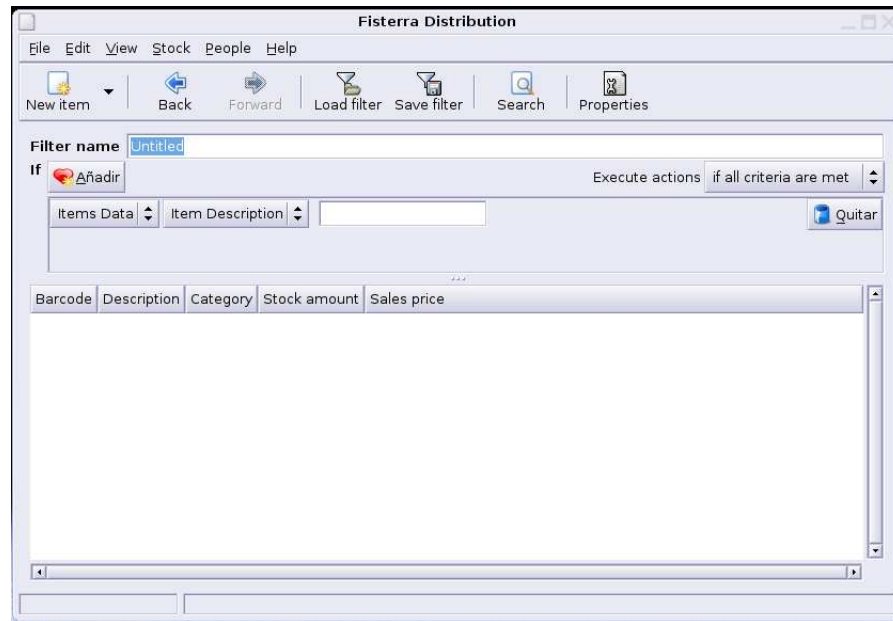


Figura 1. Pantalla principal de Fisterra distribution con un listado de artículos

3.1. Descripción del caso de uso

Desarrollaremos un caso de uso que nos permita realizar una gestión muy sencilla de clientes. El caso de uso para el ejemplo tiene los siguientes requisitos:

- La primera ventana deberá ser un listado de todos los clientes que muestre los siguientes atributos:
 - NIF
 - Nombre
 - Ciudad de residencia
 - Teléfono de contacto

Además los resultados del listado podrán filtrarse por cada uno de esos atributos o cualquier combinación de estos. Presentación de los requisitos del caso de uso.

- Al hacer doble click sobre una de las líneas se nos lanzará la segunda ventana que mostrará en detalle los datos del cliente. Además de los datos que muestra el listado esta ventana mostrará:
 - Dirección principal
 - Dirección secundaria
 - Provincia/Estado
 - País
 - Apartado de correos
 - Código Postal

Esta ventana debe servir tanto para visualizar y modificar datos de clientes ya existentes como para crear nuevos.

3.2. Haciendo una consulta a una BD en Fistera

Para la ventana del listado necesitaremos crear tres ficheros XML que nos describirán los filtros del listado, las columnas del listado y las consultas a realizar en la BD.

XML con la descripción de los filtros Los filtros se encuentran en el directorio `src/server/filters`. Abrimos el fichero `client-filter-context.xml`. Como podemos comprobar define un filtro y las distintas opciones por las que podemos filtrar. Falta por especificar un filtro por el teléfono. Insertamos el siguiente código:

```
<option value="client_phone">
  <title>Telephone number</title>
</option>
```

XML con la descripción de las columnas del listado Se encuentra en el fichero `client-scheme.xml`. Añadiremos lo siguiente para listar el teléfono.

```
<column type="G_TYPE_STRING" name="Phone" DBname="phone"
  visible="TRUE"/>
```

Cada uno de estos `tags` especifica una columna. Como se puede comprobar se indican además el tipo de datos, el título de la columna, el nombre en BD de esa columna y por último si es visible o no.

XML con la descripción de las consultas a la BD Se encuentra en el fichero `client-listing-source.xml`. Se divide en tres partes principales. En la primera se especifica el `mapping` entre los filtros y las columnas de la BD. Aquí introduciremos el `mapping` correspondiente al teléfono del cliente de forma que al final tendremos algo así como

```
<where-aliases>
  <alias name="ac_type_actor" real="ac.type_actor"/>
  <alias name="client_name" real="ac.name"/>
  <alias name="client_nif" real="ac.legal_id"/>
  <alias name="client_town" real="ad.town"/>
  <alias name="client_phone" real="ad.phone"/>
</where-aliases>
```

En la segunda parte se especifican la parte `FROM` de la consulta pudiendo especificar asimismo condiciones de `JOIN`. No es necesario modificar nada. En la tercera parte se define la parte `WHERE` de la consulta. Debido a que podemos necesitar

listar columnas con el mismo nombre de diferentes tablas podemos definir alias para utilizarlos en el XML que define el esquema, es decir, las columnas del listado. Aquí será necesario añadir el atributo *phone*:

```
<select>
  <attribute name="code" table="ac"/>
  <attribute name="legal_id" table="ac" alias="legal_id"/>
  <attribute name="name" table="ac" alias="name"/>
  <attribute name="town" table="ad" alias="town"/>
  <attribute name="phone" table="ad" alias="phone"/>
</select>
```

3.3. Creación del *builder* del listado

En este punto tenemos definido ya la consulta y no tenemos que hacer nada más del lado servidor. La ventana estará manejada por un controlador. Dicho controlador será creado por un *builder*. El correspondiente a esta ventana se encuentra en el directorio `src/client/configuration` bajo el nombre de `f_configuration_controller_client_listing_builder.c`. El cliente usa un objeto de GTK llamado *GTKUIManager* que se encarga de construir menús y *toolbars* en base a unos XML. Posteriormente se encarga de asociar acciones y grupos de acciones a estos *widgets*. El controlador empieza definiendo esas acciones.

```
static GtkActionEntry action_entries [] = {
  {"Properties", GTK_STOCK_PROPERTIES, "_Properties", "<alt>Return", N_("Client properties"),
   GTK_SIGNAL_FUNC(f_configuration_controller_client_listing_on_modify_toolbar_button_clicked)},
  {"New", GTK_STOCK_NEW, "_New client", "<control>N", N_("Create a new client"),
   GTK_SIGNAL_FUNC(f_configuration_controller_client_listing_on_add_toolbar_button_clicked)},
  {"ToolNew", GTK_STOCK_NEW, "_New client", "<control>N", N_("Create a new client"),
   GTK_SIGNAL_FUNC(f_configuration_controller_client_listing_on_add_toolbar_button_clicked)},
  {"Search", GTK_STOCK_FIND, "_Search", "<control>F", N_("Search with the filter"),
   GTK_SIGNAL_FUNC(f_configuration_controller_client_listing_on_search_button_clicked)},
  {"LoadFilter", F_STOCK_LOAD_FILTER, "_Load filter", NULL, N_("Load a filter"),
   GTK_SIGNAL_FUNC(f_control_on_load_rule_button_clicked)},
  {"SaveFilter", F_STOCK_SAVE_FILTER, "_Save filter", NULL, N_("Save a filter"),
   GTK_SIGNAL_FUNC(f_control_on_save_rule_button_clicked)},
  {"NewOrder", GTK_STOCK_INDEX, "_Add order", NULL, N_("Add an order to this client"),
   GTK_SIGNAL_FUNC(f_configuration_controller_client_listing_on_add_order_toolbar_button_clicked)}
}
```

El método principal que debemos implementar en cada *builder* es el método *create* heredado del padre de todos los *builders*. En este caso ese método está implementado en la función `f_configuration_controller_client_listing_builder_create`. Esta función sigue una plantilla común a todos los *builders*.

- En primer lugar se crea el controlador correspondiente y se accede a los contenedores donde colocaremos los *widgets* a través de la librería glade.

```
FControl *controller = F_CONTROL(f_configuration_controller_client_listing_new());
GladeXML *glade_xml = glade_xml_new(f_control_def_get_file(controller_def),
                                     f_control_def_get_widget (controller_def), NULL);
GtkWidget *window = glade_xml_get_widget(glade_xml,
                                         f_control_def_get_widget (controller_def));
GtkWidget *filter_container = glade_xml_get_widget(glade_xml,"filter_container");
GtkWidget *tree_view_container = glade_xml_get_widget(glade_xml,"tree_view_container");
```

- Posteriormente se crean las fachadas que el controlador necesitará para invocar métodos del servidor. Se instancian también los modelos para el `widget` que representa el filtro y el que representa el listado. Dichos modelos son posteriormente asignados a sus `widgets` correspondientes.
- Tanto las fachadas como los `widgets` son pasadas al controlador. Asimismo se le pasan las acciones que deberá manejar el *UI Manager*
- El siguiente paso es conectar las señales que deberá gestionar el controlador con las funciones que se encargarán de manejarlas. En nuestro caso gestionaremos un `doble-click` sobre una línea y una pulsación con el botón derecho para mostrar un *pop-up*.
- Tan solo nos queda añadir los `widgets` del filtro y el listado a sus contenedores y liberar la memoria innecesaria.

3.4. Creación del controlador del listado

Ya hemos visto en el *builder* unas cuantas funciones que debe implementar el controlador para atender a las posibles interacciones del usuario con la ventana. Nos centraremos en dos de esos métodos particularmente interesantes.

El método *search* maneja la pulsación sobre el botón buscar. La función `f_configuration_controller_client_listing_on_search_button_clicked` lo implementa. Veamos el código en detalle.

- Primero obtenemos los *widgets* y los modelos, invocando además un método para que el `widget` de filtrado se actualice con los filtros seleccionados por el usuario.

```
filter_widget = F_WIDGET_FILTER(
    f_configuration_controller_client_listing_get_filter_widget(controller));
tree_view_widget = F_WIDGET_TREE_VIEW(
    f_configuration_controller_client_listing_get_tree_view_widget(controller));

f_widget_filter_fill_model (filter_widget);

filter_model = f_widget_filter_get_model(filter_widget);
tree_view_model = f_widget_tree_view_get_model(tree_view_widget);
```

- Obtenemos los filtros formateados en XML, y se los pasamos a la fachada de listado que había creado el *builder* para que el módulo de listados realice una consulta en la BD.

```
xml_rule = f_widget_filter_model_get_xml_rule(filter_model);

listing_facade = f_configuration_controller_client_listing_get_listing_facade(controller);
xml_tuples = f_facade_listing_get_data (listing_facade, "client", "client", xml_rule);
if (f_common_facade_show_exception_widget_and_clean(F_COMMON_FACADE(listing_facade)))
    return;
```

En caso de error se muestra una ventana y se sale.

- Tan solo nos queda rellenar el `widget` de listado con las tuplas recibidas de la consulta.

```
f_widget_filter_set_model (filter_widget, filter_model);
f_widget_tree_view_model_set_tuples (tree_view_model, xml_tuples);
f_widget_tree_view_load_model (tree_view_widget);
```

El método *on_row_activated* se encarga de atender el evento de *double-click* sobre una línea. Como ya se especificó este evento debe levantar la ventana de detalle. El código necesario para hacer esto es el siguiente:

- La primera tarea es crear una instancia del controlador de la ventana (que veremos más adelante) y obtener una referencia al controlador de la ventana principal para que este pueda actualizar la lista de ventanas abiertas con el fin de poder navegar entre ellas.

```
controller = F_CONTROL(user_data);
shell_controller = F_CLIENT_SHELL_CONTROL(f_control_get_creator(controller));
new_controller = f_control_main_controller_get_controller(
    shell_controller->main_controller,
    "f_configuration_controller_client_detail_modify");
```

- Posteriormente obtendremos una referencia al *widget* que muestra un cliente en la ventana de detalle.

```
client_widget =
f_configuration_controller_client_detail_modify_get_client_widget(
    F_CONFIGURATION_CONTROLLER_CLIENT_DETAIL_MODIFY(new_controller));
```

Invocamos el método del servidor que nos recupera la información del cliente mediante la fachada. Este método nos devuelve un modelo para el *widget* anteriormente citado.

```
facade = f_configuration_controller_client_listing_get_configuration_facade(
    F_CONFIGURATION_CONTROLLER_CLIENT_LISTING(controller));
client_model = f_configuration_facade_get_enterprise_model_from_code(facade,actor_code);
```

Asignamos el modelo al *widget* y lo mostramos

```
f_widget_enterprise_view_set_model(F_WIDGET_ENTERPRISE_VIEW(client_widget),client_model);
f_widget_enterprise_view_load_model (F_WIDGET_ENTERPRISE_VIEW(client_widget));
gtk_widget_show_all (client_widget);
```

En este punto disponemos de una ventana de listado plenamente operativa que incluso tiene ya el soporte para mostrar una ventana de detalle. Lo siguiente será definir el servicio CORBA que invocábamos en la función que atendía el evento del *double-click*.

3.5. Creación del servicio CORBA

CORBA emplea un lenguaje de definición de interfaces denominado IDL ³. Estos interfaces se encuentran en el directorio `src/common/com` en un fichero de nombre `f_com_corba_services.idl`. Si lo abrimos veremos el servicio que recupera los datos de un actor cualesquiera sea una persona o una empresa a partir de un código. Es código es el conocido como código de Barnacle, se trata de una clave que identifica unívocamente un objeto en un entorno *Fisterra*. Ese código lo obtenemos del listado.

```
boolean get_enterprise_model_from_code(in unsigned long long code,
    out f_data::CORBA_Barnacle actor,
    out f_data::CORBA_Barnacle address,
    out f_data::BarnacleList countries,
    in session::object session)
    raises (f_common::FisterraException);
```

³ *Interface Definition Language*

3.6. La capa de adaptación a CORBA

Como la arquitectura de Fistera soporta otros sistemas de comunicación existe una capa de adaptación al sistema de comunicaciones tanto en el cliente como en el servidor.

Capa de adaptación a CORBA en el cliente Implementada en el fichero `src/client/com/f_com_corba_model_delegate.c`. En el se encuentra la función `f_com_corba_model_delegate_get_enterprise_model_from_code` que realiza las siguientes acciones:

- invocar el servicio CORBA
- convertir los objetos CORBA a `DataObjects` y devolverlos a la función que lo invoca

El *Model* del patrón MVC oculta a la vista el modelo de datos subyacente. La vista solo conoce modelos que están compuestos de *DataObjects*. Es necesaria por tanto la existencia una capa de fachadas entre los controladores y la capa de adaptación a las comunicaciones que se encargue de realizar las llamadas a los servicios CORBA para solicitar los `DataObjects` necesarios para construir los modelos. En concreto la fachada que atiende este servicio está implementada en el fichero `src/client/configuration/f_configuration_facade.c` en la función `f_configuration_facade_get_enterprise_model_from_code` que hace lo siguiente:

- invocar el método correspondiente de la capa de adaptación a CORBA
- instanciar un modelo nuevo
- rellenarlo con los `DataObjects` y devolverlo

Capa de adaptación a CORBA en el servidor En el servidor esta capa se encuentra implementada en el fichero `f_com_corba_general_services.c`. La estructura de estos servicios es la misma en todos:

- comprobar que el usuario tiene autorización para invocar ese servicio

```
authorization =
_f_com_services_general_check_authorization(SERVICE_GET_ENTERPRISE_MODEL,
                                             session, ev);
if (!authorization) return FALSE;
```

- convertir los objetos CORBA en `DataObjects`. En este caso el cliente no nos envía ninguno, solo espera respuesta.
- invocar el método del EGB de servicio correspondiente

```
f_egb_enterprise = f_egb_enterprise_new();
f_egb_begin_transaction(F_EGB(f_egb_enterprise));
retval = f_egb_enterprise_get_model (f_egb_enterprise,
type_actor,
&f_do_actor,
&f_do_address,
&f_do_countries);
f_egb_end_transaction(F_EGB(f_egb_enterprise));
f_com_corba_set_exception(f_egb_get_exception(F_EGB(f_egb_enterprise)),ev);
f_egb_enterprise_destroy(f_egb_enterprise);
```

- convertir los `DataObjects` en objetos CORBA para transmitirlos al cliente

```
*actor = f_com_mapping_do_barnacle_to_corba_barnacle(F_DO_BARNACLE(f_do_actor));
*address = f_com_mapping_do_barnacle_to_corba_barnacle(F_DO_BARNACLE(f_do_address));
f_com_mapping_do_barnacle_list_to_corba_barnacle_list(f_do_countries, country_list);
```

3.7. Creación del método del EGB de servicio

En este punto se deberá crear el código del servidor que implementará el servicio CORBA que proporcionará los datos necesarios para la ventana de detalle. En este caso la capa de adaptación a CORBA invoca un método del EGB `f_egb_enterprise` situado en el directorio `src/server/egbs` que le devuelve los `DataObjects` necesarios. La implementación de este método es muy sencilla y se puede comprobar que tan sólo realiza llamadas a DAOs ⁴, los objetos encargados de acceder a la BD.

3.8. Añadiendo la ventana de detalle al *main controller*

El cliente Fistera dispone de una factoría conocida como *main controller* que básicamente se encarga de instanciar los controladores correspondientes en función de un parámetro que recibe que es el nombre del controlador. La configuración de esta factoría se realiza mediante un XML en el directorio `src/client/etc` con el nombre `f_client_fistera_distribution.xml`. Veamos más en detalle el correspondiente al detalle de cliente:

- nombre con el que se instanciará el controlador y constructor

```
name="f_configuration_controller_item_detail_new"
builder="FConfigurationControllerItemDetailNewBuilder"
```

- fichero glade correspondiente a la ventana

```
file="f_widget_new.glade"
```

- opción del menú controlado por el `UI Manager` que levanta dicha ventana.

```
menu_option="New item"
action_name="NewItem"
```

- widget raíz del fichero glade

```
widget="detail_new"
```

- este último campo indica si la ventana será incrustada en la ventana principal o será flotante. Como es una ventana de detalle, no estará embebida en la ventana principal.

```
embedded="false"/>
```

⁴ *Data Access Objects*

3.9. Creación del glade de la ventana de detalle

Mediante el programa de generación de interfaces gráficas para GTK y GNOME glade se realizará el aspecto visual de la ventana y los `widget` `Fisterra` que compondrán la ventana de detalle. Ya vimos cual era el fichero de la ventana principal. Nuestro detalle estará compuesto por un único `widget` que mostrará toda la información. El glade de ese `widget` está en el directorio `src/client/glade` bajo el nombre de `f_widget_enterprise` y se puede ver en la figura 2.

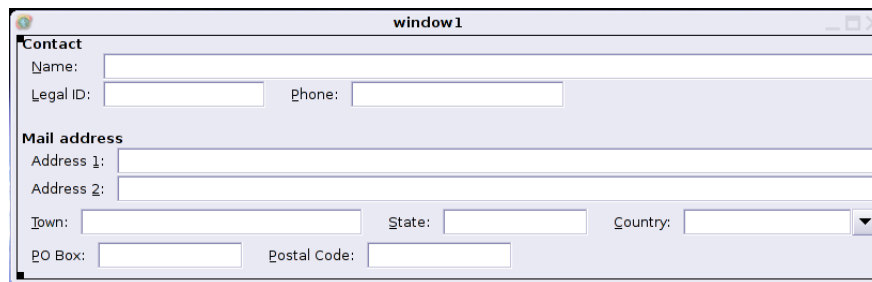


Figura 2. Glade del `widget` con los datos de cliente

3.10. Creación del builder de la ventana de detalle

El correspondiente a esta ventana se encuentra en `src/client/configuration` con el nombre `f_configuration_controller_client_detail_modify_builder.c`. Lo ya descrito para el constructor de la ventana de listado es aplicable aquí con la salvedad de que es más sencillo. Esto es debido a que al no estar esta ventana incrustada en la principal no necesita interacción alguna con el *UI Manager*.

3.11. Creación del controlador de la ventana de detalle

Este controlador es muy sencillo. Dispone de una serie de métodos para obtener o establecer el `widget` que contiene y aquellos necesarios para gestionar los eventos relacionados con las pulsaciones de botones. Revisemos en concreto el método que gestiona el click en el botón de modificar:

- En primer lugar fuerza al `widget` a que lea los datos que se ven por pantalla y actualiza el modelo del `widget` con ellos.

```
client_widget = F_WIDGET_ENTERPRISE_VIEW(  
    f_configuration_controller_client_detail_modify_get_client_widget(controller));  
f_widget_enterprise_view_fill_model(F_WIDGET_ENTERPRISE_VIEW(client_widget));  
client_model = f_widget_enterprise_view_get_model(client_widget);
```

- llamada al método de la fachada que actualiza los datos del cliente

```

facade =
    f_configuration_controller_client_detail_modify_get_configuration_facade(controller);
f_widget_enterprise_model_set_type_actor (client_model, CLIENT);

```

- por último se comprueba si la llamada se ha realizado correctamente y se cierra la ventana

```

correct_execution = f_configuration_facade_update_enterprise (facade, client_model);
if (correct_execution)
    f_client_util_open_info_window ("The client has been sucesfully modified");
else
    f_client_util_open_error_window ("Error: Client modification failed");
f_configuration_controller_client_detail_modify_destroy (controller);

```

Con esto hemos terminado la codificación del caso de uso. Tan solo nos queda recompilar el código y ejecutar la aplicación para comprobar que realmente funciona.

4. Conclusiones y trabajo futuro

El desarrollo de un caso de uso típico de listado y ventana de detalle, es un proceso que se comprueba se puede llevar a cabo en poco tiempo si se tiene el adecuado conocimiento de la arquitectura. La tecnología de **Fisterra** nos solventa muchas cuestiones como el acceso a la BD, las transacciones, autorización y autenticación, haciendo que la tarea del desarrollador sea un proceso de ‘rellenar huecos’.

En la actualidad el **framework** **Fisterra** ha alcanzado una madurez importante y no ha sufrido modificaciones relevantes durante los últimos meses a excepción de la inclusión de características nuevas. En cuanto a **Fisterra Distribution** comentar que si bien el código que se encuentra en **CVS** actualmente es escaso, el equipo de **Fisterra** ha desarrollado una gran cantidad de código el año pasado y se está procediendo a la publicación del mismo. Con el fin de que sea de la máxima calidad se están reanalizando de forma pública los distintos casos de uso y las soluciones tecnológicas empleadas. Además en **Igalia** se es consciente de que la documentación es un factor vital para incorporar nuevos desarrolladores, es por eso que se está realizando un especial esfuerzo en este apartado. Es por ello que durante los próximos meses los cambios en **Fisterra Distribution** serán considerables.

A la vez que se desarrolla este código el equipo de **Fisterra** trabaja asimismo en nuevas características. Se encuentra disponible en **CVS** una serie de test que demostraron las posibilidades de usar **PHP** como cliente de **Fisterra** usando un **middleware SOAP**. Asimismo se encuentra en evaluación la posibilidad de desarrollar en **C#**. Por último señalar que tras la migración de **ORBit2 2** a **Windows** en teoría ya es posible correr un cliente **GTK** de **Fisterra** en este sistema operativo.

5. Agradecimientos

El autor del artículo quiere agradecer a José María Casanova Crespo, Xavier Castaño García, José Dapena Paz, Javier Fernández García-Boente, Eloy Froufe Pérez,

Alejandro García Castro, Alberto García González, José Juan González Alonso, Alejandro Piñeiro Iglesias, Xavier Rodríguez Calvar, Juan José Sánchez Penas y Javier Vázquez Lamas su colaboración en el desarrollo del proyecto **Fisterra**.

Referencias

1. García Castro, A., Dapena Paz, J.: Gnome for business appliances: A case study. Technical report, IV GUADEC, Dublin (2003)
2. Casanova Crespo, J.M., García Castro, A.: Hacia fisterra 2.0: aplicaciones de empresa para pymes. Technical report, Congreso Hispalinux (2003)
3. Casanova Crespo, J.M., García Castro, A., Sánchez Penas, J.J., Dapena Paz, J.: Fisterra 2: free software for enterprise management. Technical report, Open Source International Conference (OSIC) (2004)
4. Casanova Crespo, J.M., García Castro, A.: Fisterra barnacle: Persistencia de objetos en gobject. Technical report, I GUADEC-ES, Almendralejo (2004)