(Draft) Actas de las III Jornadas de Ingeniería del Software (JIS'98). 1998, p. 411-422

Automatizando el paso de Diseño Orientado a Objeto a Codificación mediante el uso de Metainformación basada en facets

F.J. Zarazaga, S. Comella, J. Nogueras, J. Valiño

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior, Universidad de Zaragoza
María de Luna 3, 50015 Zaragoza
{ javy , juanv @posta.unizar.es}
{scomella, jnog@ebro.cps.unizar.es}

Resumen

En este trabajo se presenta un esquema de representación que, integrando metainformación en elementos de C++, facilita el paso a la generación de código, a partir de un diseño orientado a objeto. Para ello se hace uso de un conjunto de librerías basadas en facets que proporcionan las herramientas necesarias, mediante herencia y relaciones de uso, para representar y gestionar dicha metainformación. El modo de proceder en el manejo de dichas librerías es expuesto e ilustrado mediante un ejemplo La bondad de la aproximación ha sido demostrada en el desarrollo de un sistema de información de uso comercial.

Palabras Clave: sistemas de información, orientación a objetos, C++, persistencia de objetos, interfaces gráficos de usuario, metainformación

1. Introducción

A medida que los sistemas de información son más avanzados, los clientes requieren mayor variedad de servicios lo que conduce a su vez a mayores y más complejos sistemas. Para desarrollar nuevos servicios con rapidez, el software debe construirse de acuerdo a esquemas de representación y diseños que proporcionen una potente capacidad expresiva y resulten fáciles de comprender, extender y reutilizar. Uno de los caminos a seguir para incrementar la capacidad expresiva se encuentra en el uso de metainformación de los componentes de los modelos de objetos.

La metainformación es el conocimiento sobre la estructura y semántica de un modelo que lo hace autocontenido. Su manejo permite utilizar los elementos descriptivos del modelo para su propio desarrollo. En modelos de objetos, la metainformación más relevante se encuentra en el ámbito de clase, atributo y relación. Dichas metainformaciones pueden ser utilizadas como datos por parte de herramientas encargadas de generar el código fuente que implemente dicho modelo. Esto permite un desarrollo más rápido de nuevos servicios ya que posibilitan una simplificación del proceso de codificación partiendo de los diseños realizados.

C++ (Stroustrup 97) es un lenguaje de programación que, tomando como base el lenguaje C, añade utilidades específicas para la programación orientada a objetos. Su compatibilidad con el muy popular lenguaje C (lo que facilita un cambio no traumático), su potencia y su eficiencia han hecho de él el lenguaje orientado a objetos más empleado en las aplicaciones profesionales.

Aproximaciones similares para dotar a un lenguaje de programación orientado a objeto de capacidades de representación de metainformaciones han sido llevadas a cabo con CLOS (Lassila90, Lassila95). Sin embargo, la realización de la implementación en C++ resulta algo más compleja al no disponer el lenguaje de algunas utilidades de programación entre los que cabe destacar los siguientes:

1. posibilidad de manejar los programas (tipos, clases, código) como datos;
2. posibilidad de crear nuevas clases en tiempo de ejecución;
3. muy fácil manipulación de estructuras de datos dinámicas;
4. posibilidad de trabajo de forma no tipada, frente a la programación fuertemente tipada de C++ (con las que obtiene sus ventajas de eficiencia de ejecución); …

Estas características de C++ hacen que el soporte para las utilidades de frames sean mucho más dificultosas de desarrollar de forma adecuada.

En este trabajo se presenta la metodología de trabajo a aplicar para proceder a la generación del código en C++, que implemente los modelos orientados a objeto generados en la fase de diseño, mediante la utilización de un conjunto de librerías que trabajan a partir de la metainformación que dichos modelos poseen sobre sí mismos. Estas librerías reciben el nombre de librerías de facets. En la siguiente sección se presentan dichas librerías, mientras que en la sección 3 se muestra el modo en el que trabaja el programador con ellas. En el siguiente punto se presenta un ejemplo de aplicación de las mismas. El trabajo finaliza con una sección de conclusiones.

2. Las librerías de facets.

Las librerías de facets se apoyan en el formalismo procedente de la Inteligencia Artificial denominado genericamente frame y que se encuentra organizado en base a una estructura en tres niveles, *frame-slot-facet*[1] (Masini et al. 91), donde se establece un paralelismo entre frame y objeto, y slot y atributo.

C++ es un lenguaje que soporta la programación orientada a objetos. Como es habitual en este tipo de lenguajes, C++ incluye el concepto de clase. Una clase de C++ consta de atributos que describen la información y el estado de los objetos en una estructura de datos a dos niveles (objeto-slot). Las clases constan así mismo de métodos que proporcionan servicios y gestionan el acceso a los atributos a través de envío de mensajes. Sin embargo no existe un soporte directo para una infraestructura a tres niveles (objeto-slot-facet) donde los facets almacenan las informaciones sobre el slot y que es la requerida para soportar la idea de frame desde el punto de vista de programación. Resulta necesario por tanto enriquecer la estructura de atributos de C++ para que pueda contener y gestionar el meta-conocimiento relacionado con él.

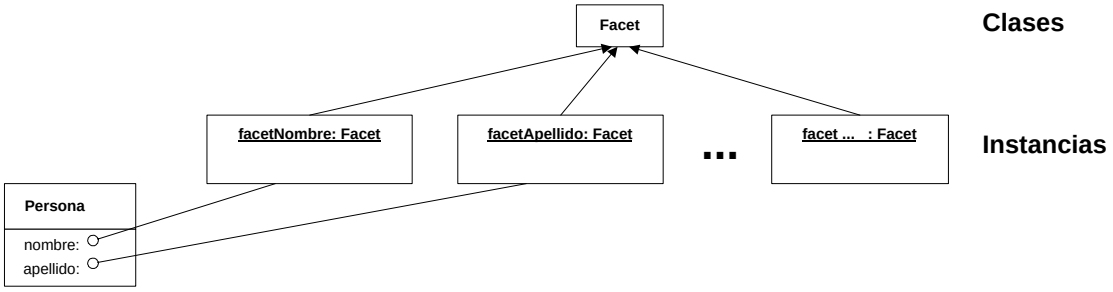


Figura 1: Estructura a tres niveles (objeto-slot-facet).

La solución adoptada por librerías de facets (Valiño et all 97) para soportar slots consiste en separar la información que es particular de cada instancia (facet con el valor específico) con la que es común a toda la clase. Así se organizará la estructura para almacenar la información del slot con una zona para guardar el valor específico para cada instancia y otra que incluye una referencia a una variable con la información común, es decir a una variable de una clase *facet*. La Figura 1 muestra esta situación en la que se aprecia en este caso la utilización de la misma instancia de facet por todas las instancias de la clase. Las librerías están integradas, básicamente, por tres grupos de elementos:

[1] En este trabajo utilizaremos la terminología inglesa de frames y slots pues creemos que está muy extendida y no encontramos traducciones al castellano suficientemente adecuadas y aceptadas.

- Un conjunto de clases que, a nivel de diseño, pueden ser considerados como tipos básicos pero que no son soportados como tales por C++, con las cuales trabajan las librerías de forma totalmente análoga a estos. Algunas de estas clases son *String* (representa una cadena de caracteres), *Fecha*, *Bool*, *Enum* (con un mayor poder de expresión que el *enum* de C), etc.
- Unas jerarquías de clases que son las que deben ser utilizadas, mediante herencia o mediante relaciones de uso, para el trabajo con las librerías. De esta forma tenemos unas jerarquías de facets (Figura 2), clases que deben actuar como padres de todas las clases del modelo (*Object*) y de todas las relaciones (*Relation*), además de la clase que encapsula a cada uno de los atributos (*Atribute*<*Facet*>).

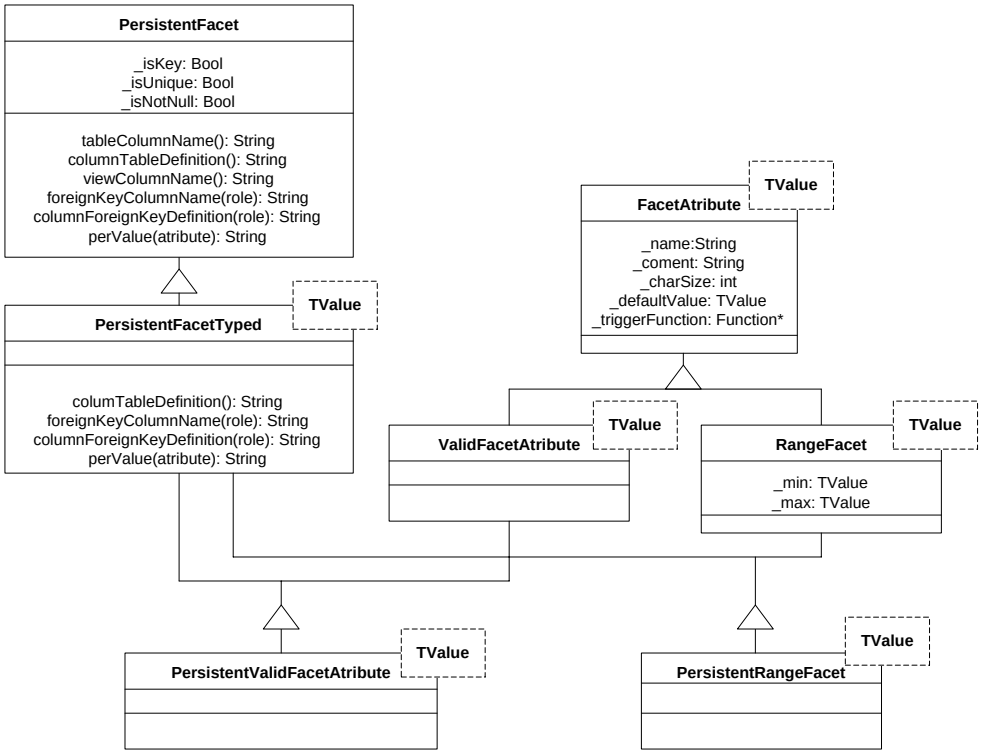


Figura 2: Jerarquía de facets

- Una serie de macros necesarias para mejorar la sencillez del uso de las librerías ya que el código que ocultan resulta muy complejo y enrevesado debido a que C++ no ofrece facilidades para soportar el concepto de frame. Estas macros simplifican en gran medida el trabajo con los facets, tal y como se podrá ver en la siguiente sección.

3. Representación de la metainformación con facets

En el proceso de modelización de un sistema se van identificando toda una serie de componentes de metainformación vinculados al mismo. Estos componentes, tal y como se

ha visto en el punto de introducción, afectan a las clases, atributos y relaciones. En la siguiente figura se puede observar un ejemplo en el que se presentan dos clases con sus atributos, y una relación entre ellas. Se han identificado una serie de metainformaciones que se han reflejado en el propio modelo mediante notas, ya que UML (Muller 97) no da soporte para estas informaciones de forma explícita en sus diagramas.

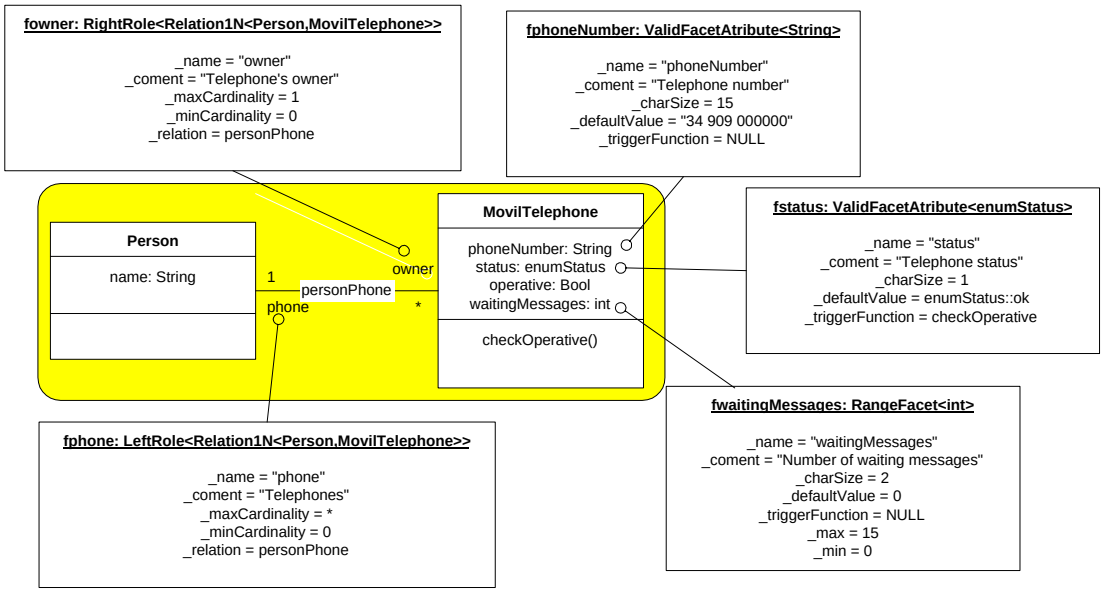


Figura 3: Ejemplo con metainformaciones

El siguiente paso consiste en llevar estas informaciones a la fase de implementación. Para ello se puede hacer uso de las librerías de facets presentadas en el punto anterior. A continuación se presenta el modo en el que esto se puede llevar a cabo.

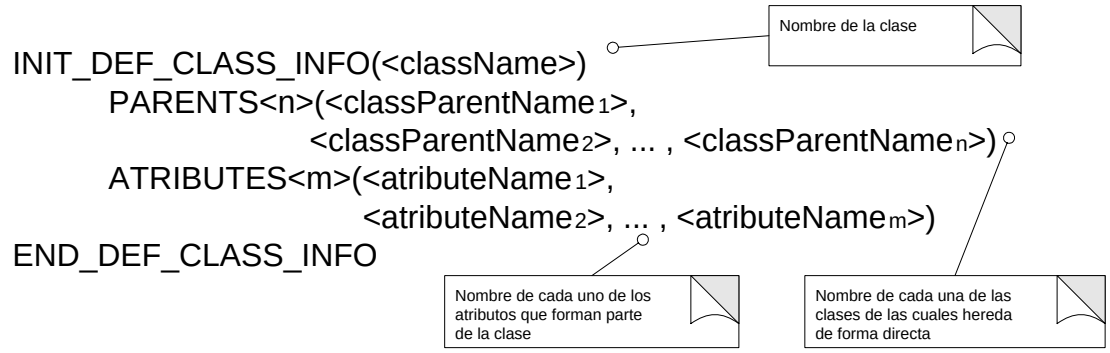
3.1. Implementación de la metainformación de clases

Para poder manejar metainformaciones relativas a la clase haciendo uso de las librerías de facets, la clase a implementar debe encontrarse dentro de la jerarquía de herencia de una clase suministrada por las librerías denominada *Object*. Además será necesario proceder a la declaración de una serie de servicios y atributos que son los encargados de proveer a la clase de metainformación sobre sí misma. Para ello, bastará con incluir la siguiente macro dentro de la declaración de la clase:

DEC_CLASS_INFO

Para llevar a cabo la definición de dichos servicios y atributos, se hará uso de un grupo de cuatro macros que, utilizadas dentro de un fichero compilable, realizan dichas definiciones

y la inicialización de los atributos correspondientes. Este grupo de macros es el que se puede observar a continuación.



Cuando se quiere que los atributos y/o relaciones vinculados a la clase sean persistentes, la clase debe encontrarse dentro de la jerarquía de herencia de *PersistentClass* (que hereda de *Object*). Además, la macro *DEC_CLASS_INFO* se debe sustituir por *DEC_CLASS_INFO_PER* (que entre otras cosas hace una llamada a la anterior), y la macro *INIT_DEF_CLASS_INFO* por *INIT_DEF_CLASS_INFO_PER*.

3.2. Implementación de la metainformación de atributos

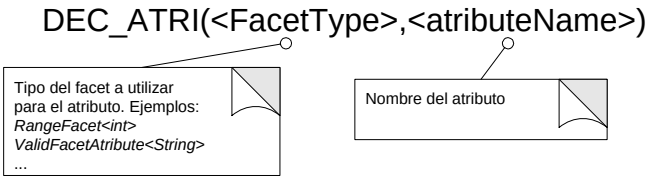
En el proceso de implementación de los atributos de una clase, cada uno ellos se debe declarar dentro del cuerpo de la clase, y definir en un fichero compilable. El primer paso a la hora de declarar un atributo consiste en elegir el tipo del facet a utilizar. Este tipo debe ser el que mejor se ajuste a las diversas metainformaciones que sobre dicho atributo se quiere manejar. En la versión actualmente operativa de las librerías de facets se incluyen dos tipos de facets:

- ValidFacetAtribute: Se trata del tipo más sencillo de facet desarrollado. Las informaciones descriptivas del atributo que puede recoger son el nombre, valor por defecto, tamaño máximo del valor del atributo (medido en número de caracteres), una breve descripción, y una función de disparo.
- RangeFacet: Este tipo de facet es usado para describir informaciones de atributos cuyos valores deben encontrarse dentro de un rango de valores determinado por un máximo y un mínimo. Cuando se trata de asignar un valor fuera de este rango, el facet se encarga de generar una excepción que puede ser capturada y manejada por la aplicación. Esto permite, entre otras cosas, que el programador deje al bajo nivel del modelo el control de los valores introducidos.

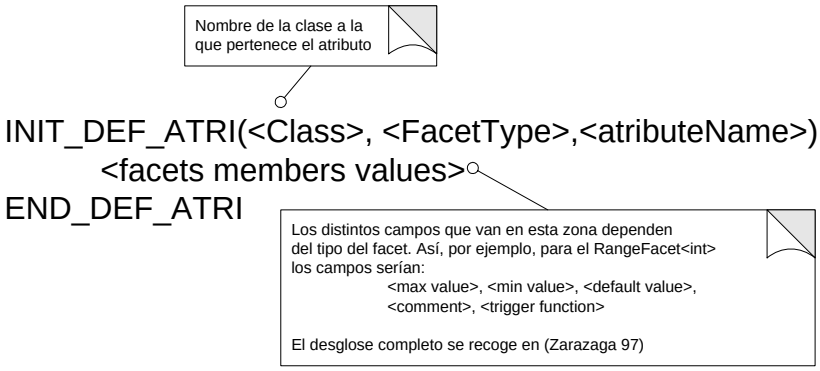
Para ambos tipos de facets, existe el correspondiente heredero que recoge las características de persistencia permitiendo definir como persistentes a los atributos a los que modela.

Los diferentes tipos de facets van instanciados por el tipo de los valores de los atributos, así, si se quiere manejar un facet del tipo *RangFacet* para un atributo cuyos valores van a ser enteros, se deberá utilizar un *RangFacet<int>*.

Para proceder a la declaración y definición de un atributo, se han construido una macros que simplifican en gran medida el proceso. La macro que declara el atributo es la siguiente:



En la macro de definición del atributo, se inicializan los valores de los distintos campos de los facets.



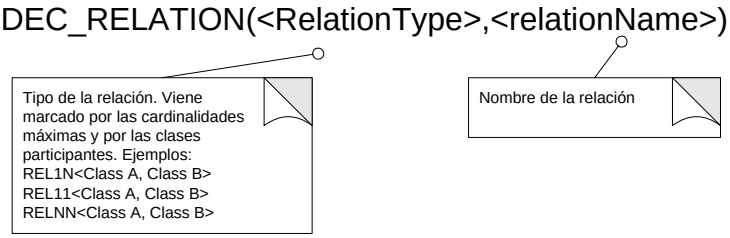
En el caso de atributos persistentes, los cambios introducidos en las macros son mínimos. Así, la macro para declarar un atributo pasa de ser *DEC_ATRI* a *DEC_ATRI_PER*. Para la definición, pasa de ser *INIT_DEF_ATRI* a *INIT_DEF_ATRI_PER*, además de incluir más miembros en el facet para saber si el atributo forma parte de la clave, si puede tomar valores no nulos, etc.

Una vez que los atributos han sido declarados y definidos, el sistema ha generado una serie de funciones de acceso dotadas de una sintaxis natural que enmascara, tal y como se puede ver a continuación, las operaciones intermedias de validación y chequeo de valores.

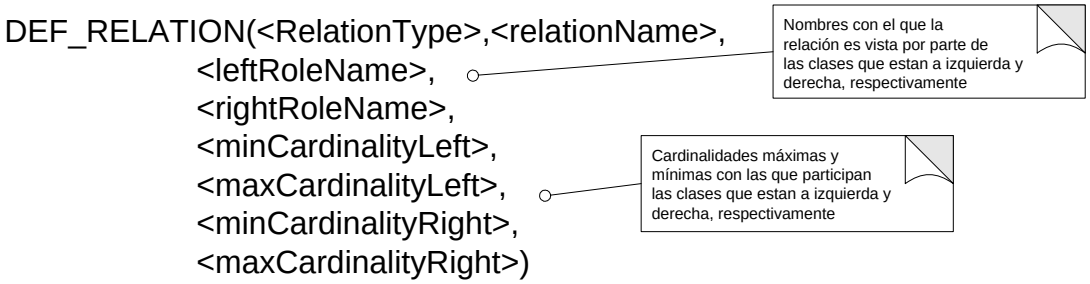
```
<classInstance>.<atributeName> = <value>;
variable = <classInstance>.<atributeName>;
```

3.3. Implementación de la metainformación de relaciones

Las relaciones se crean entre dos clases del modelo. Para ello es necesario declararlas y definirlas. La macro utilizada para declarar una clase es la siguiente:



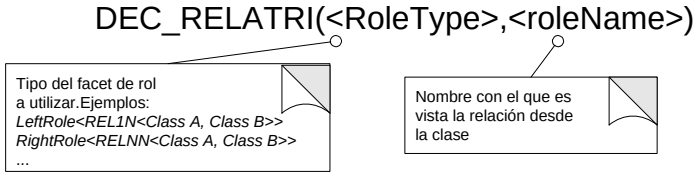
En la definición de la relación se deben incluir las cardinalidades máximas y mínimas para ambos lados de la misma, además del nombre con el que será vista por parte de cada una de las clases participantes. La macro a utilizar es la siguiente:



Las relaciones pueden ser, al igual que los atributos de las clases, persistentes o no. Una relación será persistente cuando sus informaciones (instancias de las clases participantes relacionadas entre sí) se guardan en la base de datos. La persistencia de las relaciones se indica en el tipo de la relación. Para ello se dispone de un conjunto de seis macros que nos dan todas las posibilidades de relaciones por cardinalidad y persistencia. Estas macros son:

- *REL11NP(A,B), REL1NNP(A,B), RELNNNP(A, B)*: relaciones no persistentes entre la clase A y la clase B, con cardinalidades 1:1, 1:n, n:n respectivamente.
- *REL11P(A, B), REL1NP(A, B), RELNNP(A, B)*: relaciones persistentes entre la clase A y la clase B, con cardinalidades 1:1, 1:n, n:n respectivamente.

A la hora de hacer visible la relación desde cada una de las clases participantes, el modo de operar es similar al utilizado con los atributos. Se declara el atributo de relación en la clase mediante la siguiente macro:



Se puede ver que la estructura de la macro es casi igual a la utilizada para declarar un atributo dentro de una clase.

A continuación, se define el atributo dentro de un fichero compilable. Para ello, es necesario utilizar la macro:

DEF_RELATRI(<RoleType>,<roleName>)

Como se puede observar, las macros de declaración y definición de una relación son prácticamente iguales.

Las librerías generan un conjunto de operadores de inserción, eliminación y recuperación de intancias de la relación que permiten un trabajo cómodo y natural con las relaciones. A continuación se puede ver la sintaxis utilizada:

```
<classInstance>.<roleName>.insert(<classInstance>);
<classInstance>.<roleName>.erase(<classInstance>);
variable = <classInstance>.<roleName>;
```

4. Ejemplo de paso de diseño orientado a objeto a codificación

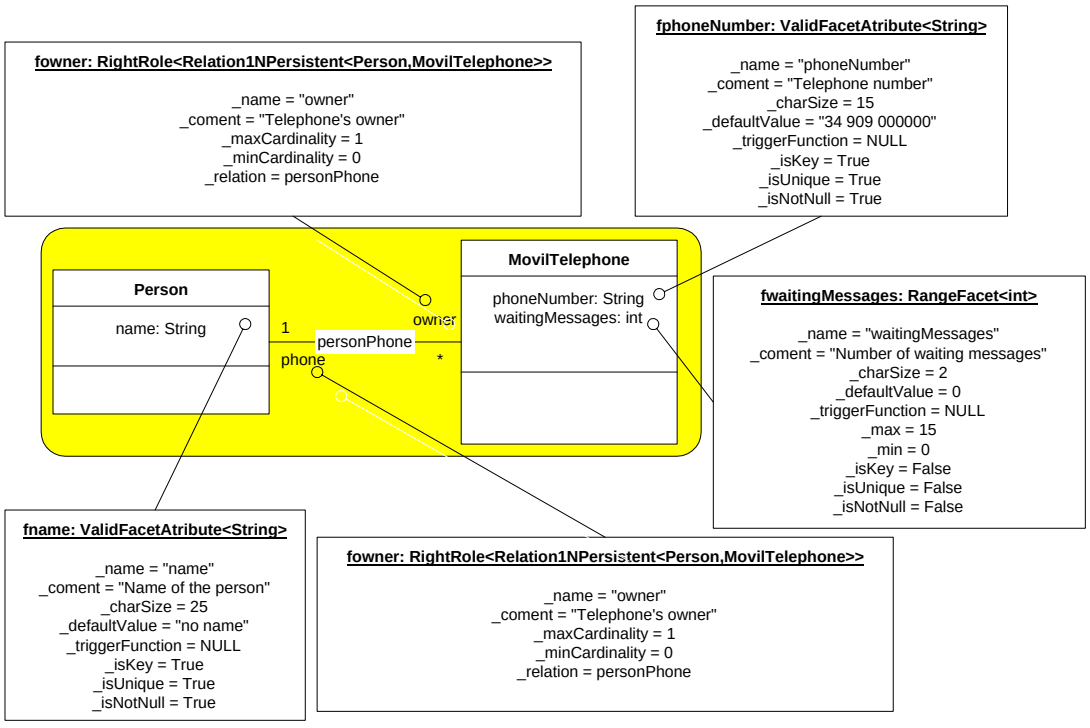


Figura 4: Ejemplo de aplicación

En este punto se va a presentar un pequeño ejemplo de aplicación de las librerías de facets, haciendo uso de las macros anteriormente descritas. Pare ello se parte de una versión algo más simple del modelo expuesto en la Figura 3, a la que se le añaden informaciones

concernientes a la persistencia de los atributos y la relación. El modelo final utilizado para el ejemplo (Figura 4) consta de dos clases, *Person* y *MovilTelephone*, que mantienen una relación entre ambas, *personPhone*, con cardinalidad 1:n.

Utilizando las macros expuestas en el punto anterior, se procede a codificar el modelo y se obtiene el código fuente que se puede observar en la Figura 5.

```
DEC_RELATION(REL1NP(Person, MovilTelephone), personPhone)
DEF_RELATION(REL1NP(Person, MovilTelephone), personPhone, phone, owner, 0, 1, 0, *)

*****************************************************************************

class TelephoneNumber : public PersistentClass
{
          DEC_CLASS_INFO_PER
          DEC_ATRI_PER(ValidFacetAtribute<String>, phoneNumber)
          DEC_ATRI_PER(RangeFacet<int>, waitingMessages)
          DEC_RELATRI(REL1NP(Person, MovilTelephone), owner)
}

DEF_ATRI_PER(MovilTelephone, ValidFacetAtribute<String>, phoneNumber)
          "35 909 000000", 15, "Telephone number", NULL,
          True, True, True
END_DEF_ATRI
DEF_ATRI_PER(MovilTelephone, RangeFacet<int>, waitingMessages)
          15, 0, 0, "Number of waiting messages", NULL,
          False, False, False
END_DEF_ATRI
DEF_RELATRI (REL1NP(Person, MovilTelephone), owner)

INIT_DEF_CLASS_INFO_PER(MovilTelephone)
          Parents1(PersistentClass)
          Atributes3(phoneNumber,waitingMessages,owner)
END_DEF_CLASS_INFO
*****************************************************************************

class Person : public PersistentClass
{
          DEC_CLASS_INFO_PER
          DEC_ATRI_PER(ValidFacetAtribute<String>, name)
          DEC_RELATRI(REL1NP(Person, MovilTelephone), phone)
}

DEF_ATRI_PER(Person, ValidFacetAtribute<String>, name)
          "no name", 25, "Name of the person", NULL,
          True, True, True
END_DEF_ATRI
DEF_RELATRI(REL1NP(Person, MovilTelephone), phone)

INIT_DEF_CLASS_INFO_PER(Person)
          Parents1(PersistentClass)
          Atributes2(name, phone)
END_DEF_CLASS_INFO
```

Figura 5: Ejemplo de código fuente

La implementación de las informaciones descriptivas de los atributos y relaciones persistentes permite la generación automática de las tablas de base de datos necesarias para llevar a cabo dicha persistencia. La generación de dichas tablas se basa en la construcción, por parte de las librerías, de un modelo relacional (Date 95) apropiado para el almacenamiento de los atributos y relaciones. Así, el código mostrado en la Figura 5 conlleva la generación de un modelo relacional correspondiente al siguiente diagrama de entidad-relación .

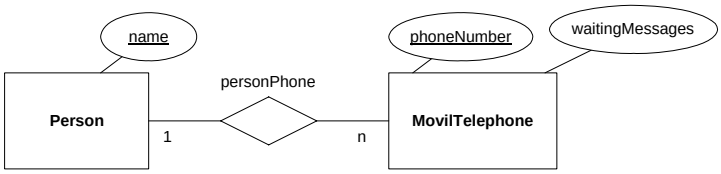


Figura 6: Modelo Entidad-Relación

El sistema construye automáticamente las tablas necesarias para implementar el diagrama de la Figura 6. De esta forma, para el presente ejemplo, se generan dos tablas dentro de la base de datos relacional. Una de ellas es la correspondiente a la clase *Person* que cuenta con una única columna. La otra es la vinculada a la clase *MovilTelephone*. Esta última, tal y como se puede observar en la Figura 7, consta de dos columnas que se corresponden con los dos atributos de la clase, más una tercera correspondiente a la implementación de la persistencia de la relación personPhone

Table: MovilTelephone		
phoneNumber (Char(15), Key, Unique, NotNull)	**waitingMessages (Int(2))**	**owner (Foreign key Person)**
"909 10 10 10"	2	"John Malkovich"
"907 55 55 55"	0	"Silvester Stallone"
"907 55 55 56"	1	"Silvester Stallone"

Figura 7: Tabla de base de datos

Del mismo modo que se ha utilizado la información contenida en los facets para generar las tablas de base de datos, se podría utilizar para la construcción de los interfaces de usuario ya que, por ejemplo, partiendo de las descripciones de los atributos de podrían generar las etiquetas de las cajas de edición.

5. Conclusiones

En este trabajo se ha presentado un esquema de representación escrito sobre C++ que facilita la automatización en la generación de código fuente en C++ partiendo del modelo de orientado a objeto, dicho esquema está basado en su soporte para metainformación.

Un conjunto de macros simplifican y facilitan la utilización de las librerías de clases. Dado que una macro es, básicamente, un sustituto de código fuente que puede ser parametrizable, en algunos casos, es necesario incluir informaciones redundantes acerca del modelo. La solución a estas redundancias vendría dada por un precompilador que hiciese una primera lectura del código escrito generando código C++ capaz de trabajar directamente con las librerías. No obstante, y dado que el volumen de redundancias no es

significativo, se ha considerado que la generación y posterior utilización de dicho precompilador supondría un trabajo excesivo frente a la mejora en los resultados.

Las librerías de facets han sido utilizadas, haciendo uso del método de trabajo aquí descrito, para el desarrollo de una aplicación de gestión de redes de radiotelefonía trunking. Dicha aplicación se encuentra instalada en tres redes ubicadas en Gran Canarias, Alicante y Viladecans (Barcelona).

Agradecimientos.

Este trabajo ha estado parcialmente financiado por la CICYT bajo el proyecto TAP95-0574.

Referencias

C.J. Date (1995): *"An introduction to database systems, Sixth edition"*, Addison-Wesley, 1995

O. Lassila (1990). *"Frames or Objects, or Both?"*, 1990, Workshop Notes from the 8th National Conference on Artificial Intelligence (AAAI-90): Object-Oriented Programming in AI, Boston (MA).

O. Lassila (1995), *"PORK Object System Programmers' Guide"*, CMU-RI-TR-95-12, The Robotics Institute, Carnegie Mellon University 1995.

G. Masini, A. Napoli, D. Colnet, D. Leonard, K. Tombe (1991): *"Object Oriented Languages"*. The APIC Series, Vol. 34. Academic Press. 1991.

P.A. Muller (1997): *"Modelado de objetos con UML"*, ed. Gestión 2000, 1997.

B. Stroustrup (1997): *"The C++ language, Third edition"*, Addison-Wesley. 1997.

J. Valiño, F.J. Zarazaga, S. Comella, J. Nogueras y P. Muro-Medrano (1997): *"Utilización de técnicas de programación basadas en frames para incrementar la potencia de representación en clases de C++ para aplicaciones de sistemas de información"*. VII Conferencia de la Asociación Española para la Inteligencia Artificial, CAEPIA'97. Málaga, España, Nov, 1997.

F.J. Zarazaga, *"Macros para el manejo de facets en códigos fuente"*. Documento interno de trabajo grupo IAAA, Enero 1997