Una aproximación para el incremento de la capacidad expresiva de objetos miembro y el RTTI de C++ basada en el uso intensivo de plantillas

Juan Valiño

Tesis doctoral

Junio de 2000

Director: Dr. Pedro R. Muro Medrano

Departamento de Informática e
Ingeniería de Sistemas

Universidad de Zaragoza

Una aproximación para el incremento de la capacidad expresiva de objetos miembro y el RTTI de C++ basada en el uso intensivo de plantillas

Juan Valiño

Tesis doctoral

Junio de 2000

Director: Dr. Pedro R. Muro Medrano

Departamento de Informática e
Ingeniería de Sistemas

Universidad de Zaragoza

–Hoy "el súper" ha hecho astillas su mesa de caoba.

–¡Hala!, ¿y con qué ***objeto****?.*

–¡Con qué ***objeto*** *va a ser!. ¡Con mi cabeza!.*

Escribo estas líneas porque este trabajo no habría llegado a término sin la colaboración desinteresada de algunas buenas gentes. Y de ello quiero dar fe.

Quiero nombrar a Pedro Muro, director de esta tesis, y a Javier Zarazaga por su paciencia, sus certeros “metaconsejos” y por apoyarme en los momentos en que, ahogándome en un mar de angustias, me asía a un bote de coca-cola. Un saludo también para el resto de miembros de un grupo, que me recuerda a cierta aldea gala y cuyo único defecto es no tener establecido un sistema de representación mínima del 25%. Qué menos.

Un abrazo afectuoso para Julio y el resto de sus compañeros cómplices de la mística matemática, suma directa de mentes preclaras y generosas.

Un recuerdo muy especial a mis compañeros de Huesca, mágico lugar, protegido por la distancia de las pequeñas y tristes guerrillas. Con Arturo, hombre cabal y atento. Con Fernando, adalid de la sensatez arriesgada, del pensamiento libre y la amistad permanente.

Mi familia ha estado en todo momento a mi lado con crítica cero, afecto constante, apoyo creciente, confianza infinita.

Bien por Manuel, el hombre suave, de cabeza brillante y sensible. Bien por Paco, rey del viento y del trueno, primero en dicción, primero en nobleza, segundo en talento. Y mejor por Cuca, destructora implacable de problemas reales e imaginarios. Y un abrazo para los respectivos consortes, en especial al de la recien citada, un hombre extraordinario, de sobresalientes cualidades físicas y síquicas.

Entre todos han logrado transformar mi pesimismo vocacional en entusiasmo grande.

Desde luego ha sido un placer escribir estas líneas sobre un folio apoyado en una pila de compañeros entintados, en vez de hacerlo sobre la chirriante madera de la mesa. La pluma corre suave y tranquila y el texto ha quedado más pinturero.

Buenos días.

Índice

Introducción — **15**

Aproximaciones a la metainformación en los lenguajes de programación — **19**

1.1. La metainformación. Ventajas. — 20
1.1.1. Problemas planteados por la ausencia de metainformación — 20
1.1.2. Implementaciones abiertas — 22
1.1.3. Otras ventajas de la metainformación — 23
1.2. Lenguajes reflexivos — 26
1.2.1. Conceptos — 26
1.2.2. Lenguajes orientados a objetos reflexivos — 28
1.2.3. Modelos de jerarquía de metaclases — 30
1.2.4. Metaobject Protocol — 33
1.2.5. Lenguajes basados en frames — 34
1.3. Ejemplos de lenguajes con propiedades reflexivas — 38
1.3.1. Smalltalk — 38
1.3.2. CLOS — 41
1.3.3. Java — 44
1.4. Metainformación en C++ — 48
1.4.1. Posibilidades de metainformación en C++ — 50
1.4.2. Soporte para la metainformación en el estándar del lenguaje — 54
1.4.3. Aproximaciones para añadir metainformación a C++ — 57
1.5. Las plantillas en C++ como herramienta de metaprogramación — 62
1.5.1. Plantillas como metafunciones — 63
1.5.2. Metaprogramación lógica — 67
1.5.3. Generación del árbol de una expresión — 68
1.5.4. Programación funcional — 70
1.5.5. Control de generación de código — 71
1.5.6. Polimorfismo en tiempo de compilación — 72
1.5.7. Limitaciones de las plantillas para metaprogramación — 74
1.6. Conclusiones — 75

Infraestructura para dar soporte a metainformación en C++ — **77**

2.1. Objetivos del esquema de metainformación — 78
2.1.1. Información de atributos — 79

2.1.2. Acceso a información de las clases 83
2.1.3. Acceso a información global del sistema 85
2.1.4. La librería de facets 85
2.2. Metainformación de atributos. 88
2.2.1. Problemas para incluir información en los atributos 89
2.2.2. Atributos enriquecidos 91
2.2.3. Clases raíz de facets y atributos enriquecidos 93
2.2.4. Facets y atributos básicos 95
2.2.5. Macros para declarar atributos y facets básicos 101
2.3. Metainformación de métodos 102
2.3.1. Macros de facets de métodos 105
2.4. Metainformación de clases 106
2.4.1. Metaobjetos estáticos como descriptores de clases 106
2.4.2. Macros para la inclusión del metaobjeto en las clases 111
2.4.3. Ampliación de la información de las clases 112
2.5. Información global del modelo 117
2.5.1. Clase raíz del modelo 117
2.5.2. Lista de clases del sistema 118
2.5.3. Lista de objetos del modelo 119
2.6. Herencia de facets 121
2.6.1. Herencia con modificación del valor del facet 121
2.6.2. Herencia con modificación del tipo del facet 124
2.6.3. Herencia de facets con modificación del tipo del facet y reaprovechamiento de valores 126
2.6.4. Herencia y construcción de la lista completa de facets 130
2.6.5. Herencia de facets en casos de herencia múltiple 131
2.6.6. Herencia de facets y herencia virtual 134
2.7. Conclusiones 135
Extensión de la jerarquía de facets para trabajar con asociaciones 139
3.1. Aproximaciones para implementar asociaciones 140
3.1.1. Utilización de punteros en las clases participantes 141
3.1.2. Implementación mediante herencia de una clase asociación 143
3.1.3. Implementación por medio de campos asociación 145
3.1.4. Implementación interna del lenguaje 147
3.2. Implementación de asociaciones mediante la librería de facets 148
3.2.1. Uso de atributos enriquecidos para trabajar con asociaciones 148
3.2.2. Roles 153
3.2.3. Asociaciones 158

3.3. Asociaciones en memoria ____ 160
3.3.1. Desarrollo de código con asociaciones en memoria ____ 165
3.3.2. Macros para manejar asociaciones en memoria ____ 167
3.3.3. Visibilidad y control de acceso ____ 168
3.3.4. Uso de las asociaciones ____ 170
3.4. Herencia ____ 171
3.4.1. Inconsistencias en asociaciones especializadas ____ 175
3.4.2. Asociaciones y herencia múltiple o virtual ____ 175
3.5. Otras clases de asociaciones ____ 177
3.6. Conclusiones ____ 184

Contenedores de objetos ____ 187

4.1. Programación genérica y contenedores estándar ____ 189
4.1.1. Programación genérica ____ 189
4.1.2. La librería de contenedores estándar STL ____ 196
4.1.3. Problemas en el uso de la STL ____ 199
4.2. Ampliación de la librería de contenedores ____ 201
4.2.1. Extensión de los contenedores estándar ____ 201
4.2.2. Vistas ____ 204
4.2.3. Contenedores virtuales ____ 209
4.2.4. Posibilidades de implementación de expresiones funcionales ____ 211
4.3. Aplicación a la librería de facets. ____ 212
4.3.1. Contenedores en memoria ____ 212
4.3.2. Contenedores en base de datos ____ 215
4.4. Conclusiones ____ 223

Conclusiones ____ 227

Bibliografía ____ 233

Introducción

A medida que los sistemas de información son más avanzados, los clientes requieren mayor variedad de servicios de información lo que conduce a su vez a mayores y más complejos sistemas. Para desarrollar nuevos servicios con rapidez, el software debe construirse basándose en esquemas de representación y en diseños que posean una potente capacidad expresiva, faciliten la utilización de paradigmas de programación orientados a objeto y resulten fáciles de extender y reutilizar. Por otra parte, el desarrollo de aplicaciones informáticas para una utilización industrial impone a su vez requisitos adicionales de eficiencia, de disponibilidad de compiladores, entornos y herramientas de desarrollo, de fiabilidad y de conexión o interacción con otros componentes de software (GUI, bases de datos, acceso a puertos, etc.). Estos requisitos hacen aconsejable utilizar un lenguaje eficiente, extendido y con buenas y probadas herramientas de desarrollo y depuración. C++ es un lenguaje que cumple muchos de estos requisitos pero carece de la flexibilidad y potencia representativa con la que cuentan otros lenguajes con menor tradición industrial como Lisp. En esta tesis se plantea dotar a C++ de infraestructuras adecuadas para reducir algunas de estas limitaciones mediante la adición de características que aumenten su capacidad expresiva y que faciliten la reutilización de la información subyacente en los propios modelos de objetos. Esta tesis trata de aunar por tanto las ventajas de dos mundos tradicionalmente contrapuestos: el de los lenguajes eficientes, de naturaleza estática y compilada como C++ y Ada, y el de los lenguajes más flexibles y con mayor capacidad expresiva y potencia representativa como Lisp y Smalltalk.

Una de las técnicas más prometedoras en este sentido consiste en capacitar al sistema para que pueda disponer y gestionar su metainformación, es decir, la información sobre su propia estructura y comportamiento. El funcionamiento del sistema queda parametrizado en términos de sí mismo lo que proporciona un notable grado de flexibilidad para personalizar su comportamiento de acuerdo a las necesidades concretas de la aplicación particular que se esté desarrollando. La separación de los aspectos del dominio de la aplicación y del dominio del comportamiento del sistema que la gobierna, conlleva importantes ventajas para el desarrollo del software a la hora de reutilizarlos de forma independiente.

C++, al igual que otros lenguajes de desarrollo de propósito general como Ada o Eiffel, ofrece un conjunto muy limitado de características para trabajar con metainformación. Se ha comprobado con Java que incorporar estas posibilidades al lenguaje, (aunque sea de forma limitada) permite abordar de forma sencilla problemas complejos como la persistencia y el envío de objetos por un canal de comunicación.

El trabajo de la tesis partirá de las infraestructuras disponibles en C++ como es el RTTI (*Runtime Type Information*) ampliándolas de modo sustancial para el soporte de metainformación. La aproximación adoptada está inspirada en técnicas provenientes de representación del conocimiento de IA y tiene muy presente aspectos de ingeniería de software. En este sentido, la solución propuesta permite acercar los modelos de objetos al código generado por el programador y elimina en muchos puntos aspectos relacionados con el diseño de bajo nivel y su consiguiente codificación en C++. En este contexto, la tesis ha puesto especial interés en el tratamiento de las relaciones entre objetos del modelo aprovechándose de la infraestructura básica de metainformación y de unas clases avanzadas de contenedores de objetos. Para realizar esto último se ha partido de la librería estándar de contenedores de C++ simplificando su sintaxis, facilitando su uso y aumentando su seguridad y su potencia.

Como se verá, las soluciones adoptadas hacen uso generalizado de las plantillas de C++. De este modo, el entorno de metainformación construido circula dentro de parámetros marcados por la filosofía del lenguaje: comprobación de tipos en tiempo de compilación y eficiencia. La localización de soluciones dentro del propio lenguaje evita tener que incluir una capa por encima del mismo que haría más complicado el mantenimiento y desarrollo de los programas.

Para poder llevar a cabo la explicación de todo este trabajo la tesis se ha organizado de la siguiente forma:

- En el capítulo 1 se estudian las ventajas que proporciona el concepto de metainformación en los lenguajes de programación, se analizan las diferentes aproximaciones posibles para realizar su soporte mediante técnicas de reflexión y se hace una comparación de las posibilidades reflexivas de algunos lenguajes concretos para pasar finalmente al estudio de las capacidades de C++ en este sentido y las posibles alternativas para mejorar dichas capacidades.

- En el capítulo 2 se precisan los objetivos concretos que se desean conseguir dentro del marco de la metainformación. En primer lugar se muestran los problemas que surgen al trabajar con metainformación dentro del lenguaje C++. A continuación se presenta un patrón (llamado atributo enriquecido) para dar cabida a metainformación en los programas mediante el uso de plantillas. Finalmente se describe una librería, llamada *librería de facets*, que proporciona un marco para trabajar con los más importantes aspectos de metainformación presentes en los programas (clases, atributos, objetos y jerarquía de clases). La librería hace uso del patrón que se acaba de citar.

- El capítulo 3 muestra cómo la librería de facets permite incorporar al lenguaje el concepto de asociación que aparece en diseño orientado a objetos permitiendo acortar la distancia que separa el diseño de la implementación. Se explica el uso de atributos enriquecidos para implementar diversos tipos de asociaciones en memoria central y cómo esa misma idea puede aplicarse para crear asociaciones con almacenamiento de valores en memoria secundaria.

- El capítulo 4 está dedicado al uso de las plantillas para implementar una serie de contenedores necesarios para la librería de facets. Usando como base la librería estándar de C++ se muestra el diseño y el uso de una serie de contenedores que van a facilitar la implementación de asociaciones de cardinalidad múltiple y de asociaciones con persistencia en bases de datos relacionales. Como se verá, estos contenedores son de propósito general con lo que sus posibilidades de aplicación van más allá de la implementación de asociaciones.

- Se finaliza la tesis con un capítulo de conclusiones y líneas abiertas de investigación.

Capítulo 1

Aproximaciones a la metainformación en los lenguajes de programación

A medida que los sistemas de información son más avanzados, los clientes requieren mayor variedad de servicios de información lo que conduce a su vez a mayores y más complejos sistemas de información. Para desarrollar nuevos servicios con rapidez, el software debe construirse basándose en esquemas de representación y en diseños que proporcionen una potente capacidad expresiva y resulten fáciles de comprender, extender y reutilizar. Una de las técnicas más prometedoras consiste en capacitar al sistema para disponer y gestionar información sobre sí mismo dotando a los sistemas de desarrollo de software de propiedades reflexivas. Con ello se consigue que una misma herramienta sirva para describir tanto la aplicación, como el sistema que la gobierna. El funcionamiento del sistema queda parametrizado en términos de sí mismo lo que proporciona un notable grado de flexibilidad para personalizar su comportamiento de acuerdo a las necesidades concretas de la aplicación particular que se esté desarrollando. De este modo se consigue establecer una separación de los aspectos del dominio de la aplicación y del dominio del comportamiento del sistema que la gobierna. Esto nos proporciona un modelo de abstracción que conlleva importantes ventajas para el desarrollo del software a la hora de reutilizarlos de forma independiente.

En este capítulo se analiza, en primer lugar, este modelo de abstracción tratando de identificar las ventajas que proporciona. Naturalmente, y al igual que ocurre con la implementación de diseños orientados a objetos, es interesante que el lenguaje utilizado disponga de estructuras que permitan implementar los conceptos de este paradigma de forma sencilla. Los lenguajes pueden dar soporte a la metainformación mediante estructuras que permiten interrogar sobre aspectos del propio lenguaje. Los lenguajes que poseen esta cualidad se denominan *reflexivos*. Dentro de este capítulo se estudian los distintos modelos de

implementación de la reflexión en estos lenguajes y se describen las capacidades reflexivas de algunos lenguajes de programación. Por su riqueza en estos aspectos se ha hecho especial hincapié en lenguajes como Smalltalk y CLOS, los cuales resultan muy adecuados para la experimentación en este terreno y constituyen un referente a tener en cuenta a la hora de realizar una implementación de aspectos reflexivos en otros lenguajes. También se analizan brevemente las capacidades reflexivas de los lenguajes basados en *frames* ya que la aproximación que se presenta en este trabajo está parcialmente inspirada en ellos. Además, se hace un análisis de las capacidades reflexivas de Java por su popularidad y por seguir una filosofía más parecida a C++ que los lenguajes anteriores. Por último, se analiza más detalladamente la reflexión en C++ y se estudian las posibilidades intrínsecas del lenguaje y en particular las que puede proporcionarle el uso de las plantillas (*templates* en C++). También se muestran otras aproximaciones para incrementar estas posibilidades.

1.1. La metainformación

Un sistema informático resuelve problemas y actúa sobre un cierto conjunto de elementos que constituyen su *dominio* o nivel base. Sin embargo, al analizar un producto informático, se observa que el esfuerzo de desarrollo no recae en su totalidad en el dominio del problema. Muchas tareas se ocupan del propio sistema independientemente de su dominio. Se pueden citar, por ejemplo, la administración de la memoria y otros recursos, la coordinación de los diferentes componentes, la gestión de la interfaz gráfica, la gestión de las comunicaciones, etc. Todos estos elementos del sistema que tratan del funcionamiento del propio sistema constituyen su metadominio o metanivel.

La importancia de la parte de metanivel no debe ser subestimada. De hecho, gran parte del esfuerzo del paso de la fase de diseño a la fase de desarrollo en una aplicación se emplea en la implementación de características no relacionadas directamente con el dominio del problema.

1.1.1. Problemas planteados por la ausencia de metainformación

En muchos desarrollos no hay una separación de los componentes base y meta del dominio ya que ni siquiera se tiene conciencia de esta división. El nivel base y el metanivel se encuentran entremezclados en el producto per-

diéndose todas las ventajas que proporciona la separación en módulos y la abstracción de conceptos. En efecto, si se desea cambiar una funcionalidad de metanivel hay que buscar a mano los lugares donde se encuentra implementada y realizar los cambios pertinentes. Esto ocurre así porque esta información se encuentra dispersa en el código en vez de estar localizada y definida en un lugar concreto. Por ejemplo, resulta muy complicado en un sistema de desarrollo tradicional modificar el sistema de almacenamiento de los datos de memoria volátil a un sistema persistente. Por otro lado, el uso de un espacio separado para el metanivel permite al programador reusar y ampliar las características de dicho metanivel. Además, el desarrollador de aplicaciones que no desea modificar el metanivel, sino simplemente utilizarlo, puede centrarse en el dominio de la aplicación sin verse perturbado por los detalles de comportamiento.

Por otro lado, la dispersión o la ausencia de la información estructural del metanivel obliga a realizar a mano tareas en las que una adecuada estructuración de la metainformación permitiría automatizar en gran medida. Así, el hecho de que un componente del sistema posea autoconocimiento de su estructura facilita la construcción de una interfaz que permita a dicha componente cooperar e interactuar con el resto de componentes que conforman la aplicación. Veámoslo con un ejemplo. Considérese una aplicación cuyo dominio incluya el personal de una empresa. Una característica estructural de metanivel son los campos de información de cada persona.

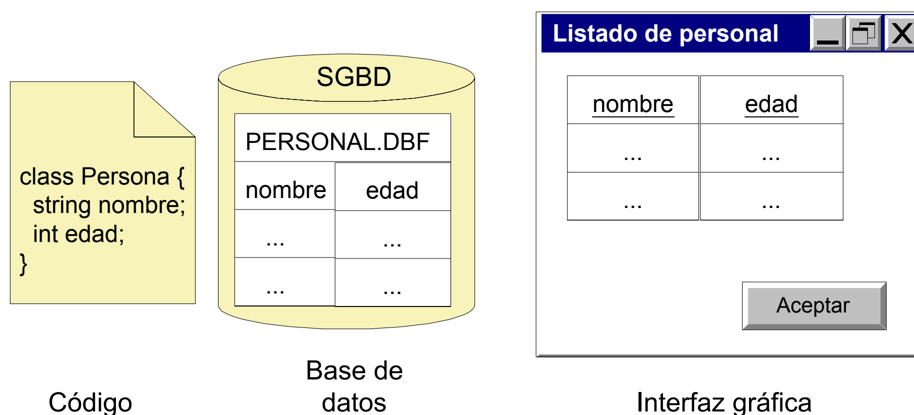


Figura 1. Redundancia de información

En la figura 1 se ve como esos metadatos aparecen repetidos en varios subsistemas de la aplicación (código fuente, tabla relacional, interfaz gráfica). Si esta información estuviera disponible para el programador, la construcción de estos subsistemas se podría automatizar en gran medida parametrizándola por medio de estos elementos.

1.1.2. Implementaciones abiertas

Un ejemplo de las ventajas de la separación de los niveles base y meta lo constituye la idea de *implementación abierta* [Kizcales 97] cuya aplicación permite optimizar las librerías y adecuarlas a cada uso concreto de las mismas. A continuación se explica brevemente la idea subyacente en este concepto.

La abstracción de la funcionalidad de un módulo software permite ocultar los aspectos internos de implementación presentando al usuario únicamente una interfaz como medio de acceso a sus servicios. Esta característica es muy productiva desde el punto de vista de la Ingeniería del Software ya que permite aumentar el grado de independencia de los distintos módulos que conforman el programa. De este modo, la realización de cambios en el código produce efectos en una zona localizada del programa. Los sistemas son así más fáciles de mantener y ampliar.

Sin embargo, una librería puede usarse en muchos contextos y de muchas formas. En algunos casos concretos las prestaciones del sistema pueden sufrir una degradación importante debida a la forma de implementar la librería. La idea básica de las implementaciones abiertas se apoya en el hecho de que la implementación de una librería puede ser óptima para un determinado uso del módulo pero ineficiente en otros casos. El usuario se encuentra, en estos casos, indefenso al tener vedado el acceso a los mecanismos de implementación. [Kizcales 96] muestra un ejemplo gráfico de este problema: un usuario desea programar una tabla de hoja de cálculo haciendo uso de un módulo gráfico que implementa un mecanismo básico de visualización por medio de ventanas. Sin embargo, la implementación del sistema de ventanas consume muchos recursos del sistema, siendo totalmente ineficiente utilizar una ventana para cada una de las celdas de la hoja de cálculo. Un sistema con una implementación abierta permitiría al usuario diseñar su propio sistema de "pequeñas ventanas".

La figura 2 muestra un esquema de la relación entre el usuario y una librería con implementación abierta. La librería proporciona dos interfaces, una de

dominio base y otra de dominio meta y aporta una implementación por defecto que el usuario puede sustituir por otra sin que esto afecte al código base.

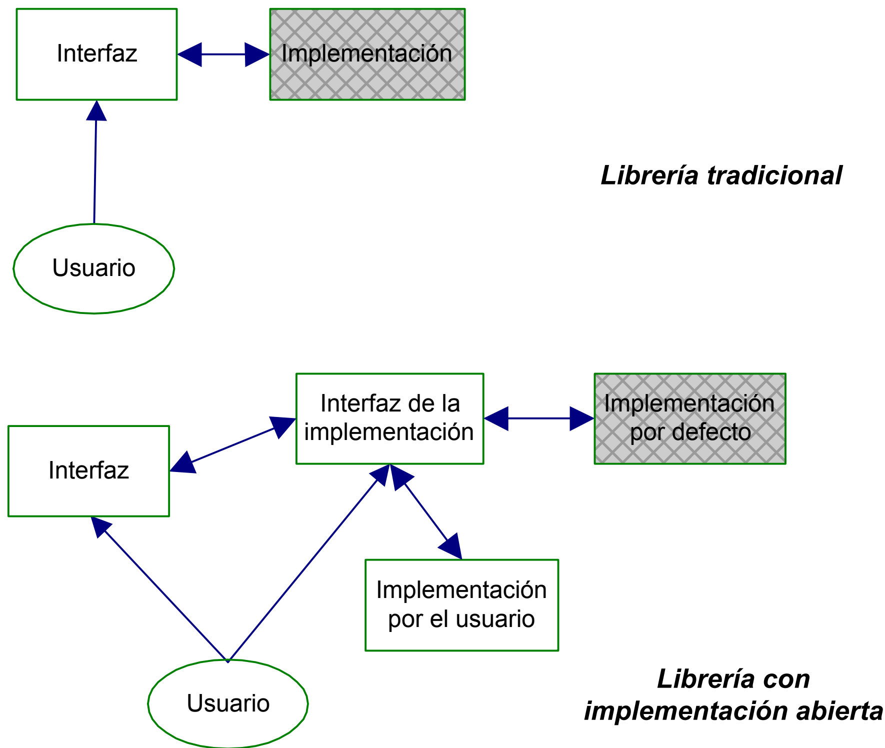


Figura 2. Implementaciones abiertas

Una consecuencia de las implementaciones abiertas es que se difumina la distinción entre diseñador y usuario de la librería proporcionando al usuario dos interfaces, una interfaz primaria para implementar los servicios del dominio o nivel base y una metainterfaz que le permite controlar la implementación sobre la que se asienta la interfaz básica. Ambas interfaces son independientes: el cambio a un sistema de implementación diferente se realiza sin tener que modificar el código del programa.

1.1.3. Otras ventajas de la metainformación

Las posibilidades de la metainformación de los programas son muy prometedoras y van más allá de las implementaciones abiertas:

- *Persistencia y envío de datos por red*. Para almacenar información en bases de datos o para enviar datos a otro nodo de una red o procesador es necesario que el sistema conozca la estructura interna de esos datos. Por ejemplo, [Lee 98] utiliza las capacidades de metainformación de Java para implementar un sistema de persistencia.

- *Objetos distribuidos*. En arquitecturas distribuidas orientadas a objeto aparece el concepto de objeto próximo (*proxy*) como representación local de un objeto remoto. El objeto próximo se encarga de realizar la llamada remota al objeto real al que representa. Obsérvese que la implementación de objetos próximos pertenece al nivel meta de la aplicación pues tiene que ver con la funcionalidad del sistema y no con el dominio de la aplicación. [Ledoux 97] muestra cómo la metainformación facilita la tarea de implementar ORBs (*Object Request Broker*) como CORBA [OMG 95].

- *Concurrencia y sincronización*. Los lenguajes concurrentes orientados a objetos [Scaife 96] se benefician de un modo especial de las técnicas de reflexión ya que éstas proporcionan un mecanismo para especificar políticas diferentes de concurrencia. [Foote 89] enumera ejemplos de implementación de ciertos tipos de concurrencia y programación distribuida (objetos futuros, activos y remotos) mediante interceptación del mecanismo general de disparo de mensajes.

- *Herramientas de depuración dentro del propio lenguaje*. La construcción de un depurador se facilita mucho si el lenguaje proporciona metainformación referente a las variables de cada módulo con sus nombres, tipos y valores. La metainformación de las funciones permitirá modificar estos valores durante la depuración.

- *Implementación de patrones*. Un patrón [Gamma 96] [Buschmann 96] [Martin 98] es una solución a un problema que aparece de forma recurrente. A la hora de implementar un patrón hay que adoptar una serie de criterios o normas de actuación que provocan que el patrón no tenga una representación explícita y distinta en el código produciéndose en muchos casos redundancias en dicho código. El razonamiento en el metanivel puede ayudar a definir el patrón como componente básico del lenguaje mediante directivas incluidas en el código fuente. Un metaprograma trata entonces estas directivas generando el código adecuado. Este es el sistema empleado en [Tatsubori 98] y [Soukup 95].

- *Recolección de datos para estadísticas* (*profiling*) *y trazado* (*tracing*). Se puede modificar el comportamiento de las funciones de modo que a su entrada se recoja el valor de sus parámetros y a su salida el valor devuelto junto con el tiempo empleado, guardando estos valores en estructuras adecuadas. Con estos datos se pueden realizar análisis de tiempos y llamadas y se puede volcar en un fichero la secuencia de llamadas a funciones que hayan ocurrido durante la ejecución del programa.

- *Generadores automáticos de documentación del código fuente*. El lenguaje puede permitir añadir datos a los diferentes componentes sintácticos, que informen sobre su estructura, uso, etc. Se puede procesar esta información para generar automáticamente un informe con la documentación completa o parcial de los diferentes módulos del programa, (Java).

- *Navegadores* (*browsers*) de la estructura de los elementos del programa. Esto se puede conseguir si el lenguaje proporciona estructuras que contengan la información relativa a estos elementos.

- *Componentes con autoinformación que permita su reutilización*. Los componentes son piezas de software que pueden utilizarse en otros programas. Para realizar programas basados en componentes existen herramientas gráficas que permiten diseñar la aplicación incorporando estos elementos gráficamente (moviendo iconos). Estos componentes se configuran en tiempo de programación asignando valores a sus atributos y proporcionando el esqueleto del código asociado a sus diferentes métodos. La herramienta gráfica debe extraer esta información del propio componente ya compilado y para poder hacerlo necesita que dicho componente guarde información de sí mismo (atributos, métodos, etc.). Un ejemplo de programación basada en componentes lo constituyen los *Beans* de Java [Sun 99].

- *Intérpretes del lenguaje gracias a la metainformación del sistema*. Para ello las funciones, variables y resto de estructuras deben tener información sobre su nombre y la forma de acceder a ellas.

- *Grabación del estado del sistema*. Se puede guardar el estado del sistema en un momento dado y recuperarlo posteriormente. Esto puede servir, por ejemplo, para realizar tareas de depuración retrocediendo en la ejecución del programa o para guardar el estado de un programa en un soporte persistente y recuperarlo en una subsiguiente ejecución. [Kasbekar 98] utiliza

una aproximación más sofisticada para conseguir este objetivo mediante el uso de metainformación relativa a la dependencia de datos.

1.2. Lenguajes reflexivos

1.2.1. Conceptos

La constatación de la importancia de la metainformación y su adecuada representación en los sistemas informáticos ha provocado el nacimiento de lenguajes que contienen estructuras dentro del propio lenguaje que permiten acceder a la metainformación de los programas. Son los llamados *lenguajes reflexivos*. Varios trabajos [Maes 87] [Smith 90] [Cointe 96] [Bobrow 93] se ocupan de clarificar los conceptos relacionados con la reflexión en los lenguajes de programación.

Un lenguaje reflexivo es aquél que cuenta con elementos del propio lenguaje que permiten representar la estructura, el comportamiento y el estado del programa durante su ejecución. Estos lenguajes permiten acceder a los elementos y al estado de un programa por medio de estructuras del propio lenguaje. Bobrow llama *reification* a dicha codificación. Los elementos que contienen la información del programa deben poder observarse y manipularse dentro del propio lenguaje y serán por tanto elementos de primer orden dentro de dicho lenguaje. Hay muchas nociones que intervienen en la metainformación de un programa y cualquiera de ellas es candidata a ser codificada mediante estos metadatos: tipos de datos, campos, mensajes, bloques de código, expresiones, árboles sintácticos, etc. [Foote 90] muestra una lista exhaustiva de tales elementos en el contexto de los lenguajes orientados a objetos.

En la codificación que se acaba de describir se pueden distinguir dos aspectos:

- Estructurales o de descripción de los elementos que integran el sistema. Por ejemplo, una estructura que tenga acceso a todas las variables del programa puede servir para guardar el estado de dicho programa con vistas a recuperarlo en un momento posterior.

- Funcionales o de comportamiento, por ejemplo el mecanismo de comunicación en un sistema distribuido, el sistema empleado para realizar las llamadas a funciones, la gestión de la memoria, etc.

Se dice que un lenguaje tiene capacidades de *introspección* si permite únicamente acceder a la información de su estado sin poder modificarla (es decir, cuando los datos que describen su estado son constantes). Por el contrario, se habla de que el lenguaje posee capacidades de *intervención* (*intercession*) si permite modificar su propia estructura o comportamiento mediante la manipulación de los datos que lo describen. Nótese que, en el caso de la intervención, el funcionamiento del sistema está causalmente conectado a los datos que lo describen, de modo que una modificación de estos datos se refleja automáticamente en el comportamiento del programa.

Las ideas de reflexión aparecen en muchos paradigmas y lenguajes de programación. Un estudio comparativo puede verse en [Demers 95]. En algunos lenguajes, los programas pueden ser tratados como datos, de este modo, el usuario puede dinámicamente construir programas (es decir, realizar metaprogramación) que le permiten crear nuevos modelos de computación. La metaprogramación no tiene por qué ir unida a la reflexión ya que el metaprograma no tiene por qué estar escrito en el mismo lenguaje que el programa sobre el que actúa. La reflexión, sin embargo permite muchas veces realizar metaprogramación utilizando el propio lenguaje.

Los lenguajes que funcionan en modo interpretado permiten añadir características reflexivas de un modo más sencillo que si son compilados ya que el intérprete debe contener y construir gran cantidad de metainformación para poder llevar a cabo su tarea. Para dotar a dicho lenguaje de capacidades reflexivas basta con hacer pública al programador esa información que utiliza internamente para realizar el proceso de interpretación. Si el intérprete está disponible como una función más del lenguaje (que toma como parámetro un bloque de código fuente) se dice que estamos en presencia de un intérprete metacircular. Para ello es necesario, naturalmente, que los programas puedan tratarse como datos.

La metainformación en los lenguajes compilables se necesita para realizar la compilación pero suele ser eliminada del código ejecutable salvo cuando se compile con opciones de depurado. [Kakkad 98] es una arquitectura que proporciona metainformación extraída del código compilado con opciones de depuración.

1.2.2. Lenguajes orientados a objetos reflexivos

El paradigma de orientación a objetos resulta muy apropiado para implementar características de reflexión. La metainformación se puede modelar por medio de objetos y clases del propio lenguaje que reciben el nombre de metaobjetos y metaclases. Los metaobjetos definen, implementan y dirigen el comportamiento y la estructura de la aplicación en sí. Se consigue de este modo una separación natural de los conceptos de nivel base y nivel meta. La funcionalidad en el metanivel queda distribuida en una jerarquía de objetos y clases permitiendo su manejo de forma independiente. Asimismo, la herencia, el encapsulado y la abstracción permiten la reutilización y el mantenimiento de la parte de metanivel del mismo modo que ocurre con el nivel básico. Toda la potencia de la orientación a objetos queda por tanto al servicio de la implementación del metanivel.

Cada objeto del lenguaje tiene asociado un metaobjeto que lo describe y que determina su comportamiento. Los modelos para estructurar esta relación entre un objeto y su metaobjeto se pueden clasificar en tres tipos (figura 3).

- *Identidad metaobjeto clase*. Es el modelo más simple. Consiste en utilizar las clases como sistema de centralización de la metainformación. Cada clase determina las propiedades y el comportamiento de sus instancias (los objetos pertenecientes a la clase), la representación de éstas en memoria, la forma de crear nuevas instancias, las acciones a realizar cuando se llame a un método para una instancia, etc. Las clases son así los metaobjetos de sus instancias.

- *Metaobjeto por clase*. En este caso, el comportamiento también se estructura en el ámbito de las clases, es decir, todas las instancias de una clase se rigen por el mismo sistema de funcionamiento. La diferencia consiste en que la clase se ocupa únicamente de almacenar la información de su interfaz, es decir, la lista de atributos y métodos. Los aspectos de implementación antes citados, como la organización en memoria de los atributos, se guardan en un metaobjeto especial. Hay una correspondencia uno a uno entre este metaobjeto y la clase. Smalltalk80 utiliza esta aproximación.

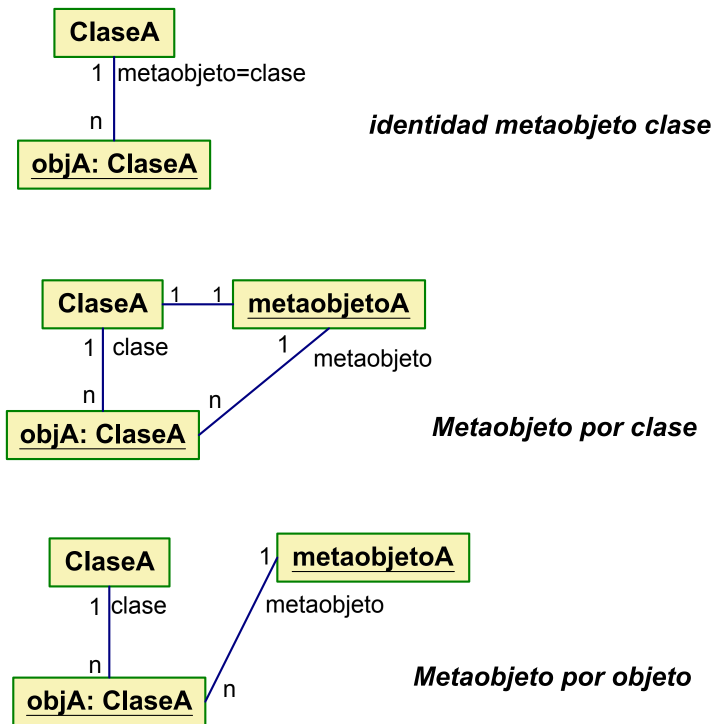


Figura 3. Modelos de coordinación entre los niveles base y meta

También se debe usar esta aproximación en el caso de lenguajes en los que las clases no son *ciudadanos de primera clase* (es decir, aquellos en los que las clases no se pueden crear, modificar o pasar como parámetros en tiempo de ejecución). En estos lenguajes, las clases tienen un comportamiento predeterminado que no se puede modificar, y "desaparecen" como tales en tiempo de compilación o ven reducida drásticamente su representación en el código binario. Por ello se hace necesario crear un objeto para cada clase que mantenga en tiempo de ejecución esa información estática de las clases que desaparece en tiempo de compilación. C++ [Stroustrup 97], Ada95 [Barnes 95] [Taft 97], Eiffel [Meyer 92] y otros lenguajes fuertemente tipificados deben utilizar esta aproximación. Los metaobjetos pertenecerán a clases que reciben el nombre de *metaclases*. En los casos más simples habrá un única metaclase a la que pertenecerán todos los metaobjetos.

- *Metaobjeto por objeto*. Un modelo más flexible pero más costoso se describe en [Ferber 89]. En este caso la metainformación se localiza en

el ámbito de los objetos y no de las clases, es decir, cada objeto tendrá su propio metaobjeto el cual determinará su comportamiento. Este modelo es apropiado, por ejemplo, si se desea guardar para cada objeto el historial de los mensajes recibidos.

Los dos primeros modelos son los más extendidos. En cualquiera de los casos, los metaobjetos, al ser objetos, deben ser instancias de otra clase que será su metaclase. Las instancias de una clase son objetos finales mientras que las instancias de una metaclase son clases.

1.2.3. Modelos de jerarquía de metaclases

Existen tres modelos típicos de jerarquía de metaclases en los lenguajes orientados a objetos:

- *Metaclase única* (figura 4). Es el modelo más sencillo y es el adoptado en Java, C++ y Smalltalk76 [Ingalls 78]. Todas las clases son instancias de una única metaclase (Class en el ejemplo). Los atributos de esta metaclase se convierten en atributos estáticos (atributos de clase) dentro de la clase[1]. Por ejemplo, si la metaclase tiene un atributo nombre, entonces cada instancia, o sea, cada clase tendrá un atributo nombre estático, es decir único. Del mismo modo, la metaclase puede tener un atributo que almacene la lista de atributos de la clase o la lista de métodos. La metaclase suele tener también un método new que se convertirá en un new estático para cada clase y que servirá para crear instancias. Si se considera la metaclase como una clase más, la metaclase será una instancia de la propia metaclase y por tanto tendrá un new estático que le servirá para crear clases (siempre que el lenguaje permita la creación dinámica de clases).

[1] En orientación a objetos, una clase puede tener atributos normales con un valor diferente para cada objeto de la clase (por ejemplo el nombre en una clase Persona) y atributos estáticos que tienen el mismo valor para todos los objetos de la clase (por ejemplo la longitud máxima del nombre en la clase Persona).

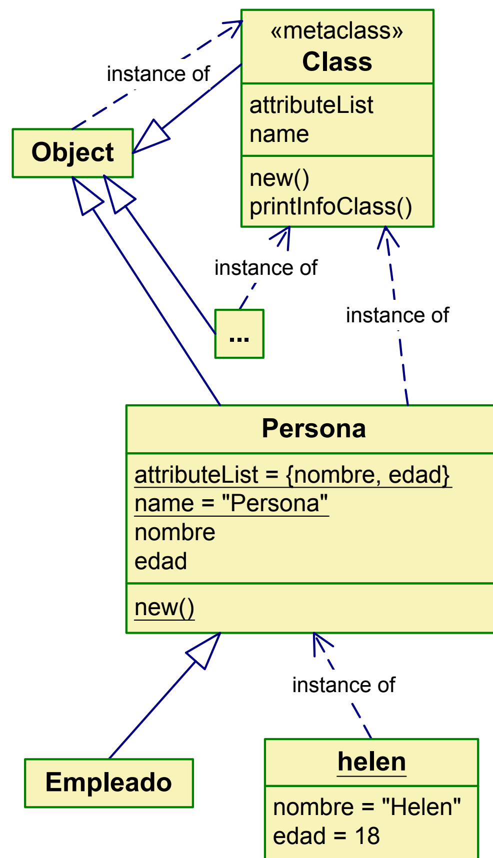


Figura 4. Metaclase única

En algunos lenguajes, como Java, existe una clase raíz del árbol de herencia que suele recibir el nombre de Object. Al ser Object una clase será instancia de Class. Al ser Class una clase heredará de Object.

Este modelo tiene la desventaja de que todas las clases tienen la misma estructura. No se pueden abstraer propiedades estáticas comunes de clases. Por ejemplo, si un grupo de clases almacena un contador con el número de instancias que se creen, este contador habrá que incluirlo clase por clase ya que no se puede declarar una metaclase que con este atributo.

- *Metametaclase única* (figura 5). Este modelo añade un grado más de libertad al permitir la existencia de varias metaclases que agruparán propiedades comunes de grupos específicos de clases. Todas estas metacla-

ses serán instancias de una única metametaclase. La metametaclase será la única clase cuyas instancias son metaclases. *Loops* [Bobrow 83] sigue este esquema.

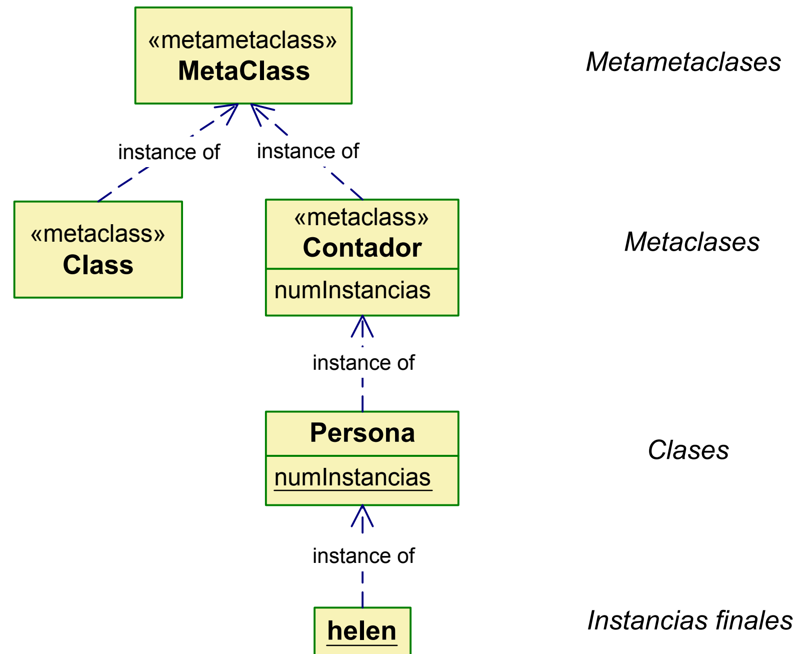


Figura 5. Metametaclase única

- *Modelo uniforme* (figura 6). Éste es el modelo más libre y el que permite más flexibilidad. En él no hay límite en la jerarquía de metaclases ya que las metaclases y las clases se tratan uniformemente. La única diferencia entre una clase y una metaclase es que en ésta última el método `new` crea clases en vez de instancias finales. ObjVLisp [Cointe 87] y SOM [Danforth 94] utilizan esta aproximación.

  El modelo uniforme, aunque muy flexible, introduce un problema nuevo: la compatibilidad de metaclases. Si `B` deriva de `A`, ¿la metaclase de `B` debe derivar de la metaclase de `A`?. Sea `a` un objeto de `A` y `f` un método de la metaclase de `A`. Si no se exige derivación, una llamada del tipo `a.metaclass().f()` podría no ser correcta si en vez de `a` usamos un objeto de `B` puesto que la metaclase de `B` no tiene por qué tener el método `f`. Si se exige derivación entonces cualquier propiedad que tenga la clase `A`

se propagará a sus clases derivadas. Esto no siempre es deseable; por ejemplo la clase A puede tener la propiedad de que sólo se puede crear una instancia pero sus clases derivadas no tienen por qué tener esta restricción. En [Bouraqadi 98] se analiza con mayor detalle este problema.

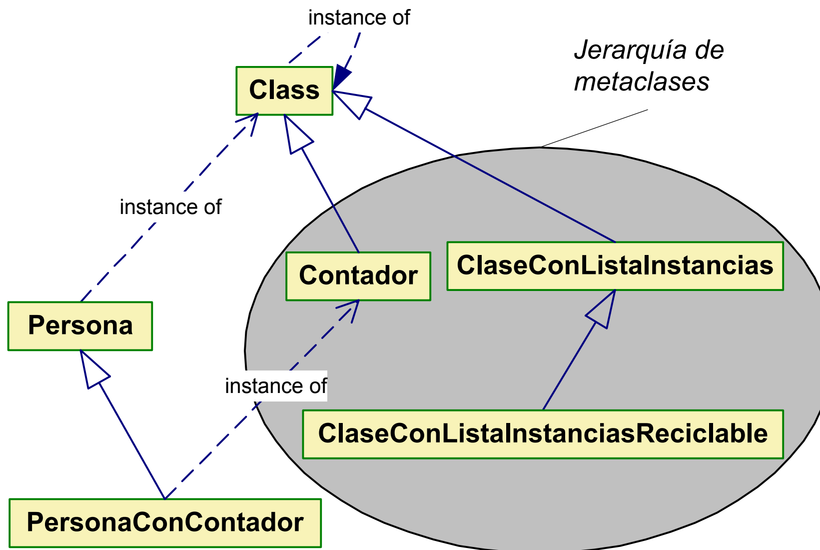


Figura 6. Modelo uniforme

1.2.4. Metaobject Protocol

En el contexto de los lenguajes orientados a objetos se llama *Metaobject Protocol* (MOP) a un sistema que proporciona una interfaz al usuario del lenguaje para modificar su comportamiento e implementación por medio de metaobjetos. Generalmente un MOP proporciona metaobjetos que determinan el comportamiento del sistema por defecto. La herencia permite al usuario personalizar dicho comportamiento. La exportación del modelo interno del lenguaje permite modificar el propio lenguaje haciendo de su traductor una implementación abierta en el sentido visto en 1.1.2. Por ejemplo, un lenguaje que incorpore un sistema de gestión de memoria basado en reciclaje de basura puede resultar altamente ineficiente para ciertos tipos de programas. El hecho de permitir desactivar la tarea recolectora de basura en ciertos momentos críticos es un ejemplo de cómo el control de los mecanismos del lenguaje puede ayudar al

desarrollo del software. La utilización del MOP podrá permitir al usuario establecer un mecanismo de gestión de memoria más adecuado a sus necesidades.

Gracias a los MOP un usuario puede acceder a territorios que le estaban vedados ya que el diseño y la implementación de los lenguajes requieren un alto grado de especialización y avanzados conocimientos. El comportamiento del lenguaje se puede personalizar introduciendo variaciones sobre el modelo inicial. Características que en otro caso requerirían la definición de un nuevo lenguaje pueden añadirse a lenguajes existentes modificando la implementación de su comportamiento.

Un MOP es una herramienta poderosa cuyo uso requiere disciplina y rigor ya que el acceso a todo el sistema de implementación representa un peligro potencial. [Kizcales 91] hace un estudio de los conceptos relacionados con los MOPs presentando como ejemplo el MOP de CLOS.

1.2.5. Lenguajes basados en frames

La idea de autoconocimiento de un sistema aparece de forma natural en Inteligencia Artificial. Una de las aproximaciones tradicionales en este campo se ha centrado en los aspectos de representación de la información/conocimiento [Brachman 85]. Las ideas básicas de estos esquemas de representación [Minsky 81], que provienen de las redes semánticas [Brachman 79], consisten en proporcionar formas de expresar la estructura lógica del conocimiento para dar flexibilidad a la asociación de procedimientos con piezas de conocimiento específicas y controlar la accesibilidad relativa de diferentes hechos y descripciones. Estos formalismos, con la denominación genérica de *frames*, se basan en entidades conceptuales estructuradas con descripciones asociadas [Bobrow 77]. Estas entidades forman una red de unidades de memoria con diferentes tipos de conexiones, teniendo cada una implicaciones bien definidas para el proceso de recuperación. Los procedimientos pueden estar asociados directamente con la estructura interna de una entidad conceptual. Estos formalismos proporcionan buenos mecanismos de separación en módulos y reutilización del conocimiento (centralizando conocimiento o a través de relaciones entre entidades y utilizando mecanismos de inferencia como el de herencia) que ya han sido transferidos a entornos más especializados de sistemas de información con mayores requisitos de ingeniería del software a través de los lenguajes y métodos de programación orientados a objeto [Stefik 85].

Los lenguajes basados en frames tienen algunos aspectos en común con los lenguajes orientados a objetos: los frames, como las clases, son entidades genéricas que contienen *slots* (atributos). Cada slot tiene asociado ciertos elementos de información llamados facets. La organización de los frames se estructura así en tres niveles *frame-slot-facet* [Masini 91].

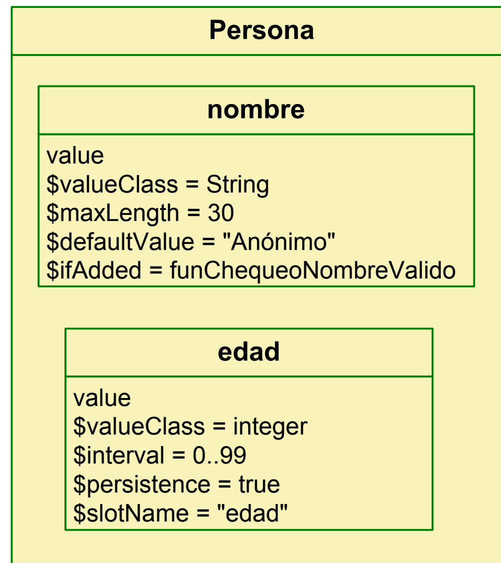


Figura 7. Ejemplo de frame

Los facets describen distintos aspectos de los atributos (slots) de las entidades, es decir conocimiento sobre el propio conocimiento de la entidad. Además del metaconocimiento específico de las entidades derivado del dominio de aplicación, existe de partida cierto metaconocimiento originado por su uso en un sistema de información. El conocimiento necesario para su persistencia (tipos de dato, nombre de tablas y de campos si la persistencia es en una base de datos relacional) sería un ejemplo de metainformación útil para la entidad. Los requerimientos de la interfaz gráfica de usuario proporcionan otra fuente de utilización de este metaconocimiento. En este sentido, con frecuencia es necesario adquirir información del usuario para completar la información de las entidades, y puede resultar útil ubicar en la propia entidad el conocimiento de cómo adquirir esta información: texto a salir por pantalla, comprobación de validez de datos de entrada, apariencia de salida (por ejemplo una lista desplegable con valores para un tipo de dato enumerado), etc.

La figura 7 ilustra parcialmente un ejemplo típico de frame con la organización de información en tres niveles y con una serie de facets para los slots nombre y edad.

Cada uno de los lenguajes basados en frames ofrece su propio conjunto de facets. A continuación se indican los tipos de facets más comunes. El desglose se ha dividido en dos categorías [Masini 91]:

- *Declarativas o estructurales*. Definen aspectos estructurales de los slots.
- *De procedimiento o comportamiento*. Especifican comportamiento asociado localmente a los slots y personalizan su acceso. Son utilizados normalmente para realizar efectos laterales como calcular o actualizar el valor del slot.

Características estructurales:

- *Descripciones taxonómicas*. Una característica esencial de los lenguajes de representación basados en frames consiste en sus construcciones para describir individuos y clases de individuos en el dominio de aplicación formando taxonomías. Un frame representando una clase puede contener descripciones prototípicas de los miembros de la clase, así como descripciones de la clase misma [Fickes 85]. Debe proporcionar, asimismo, los mecanismos necesarios para implementar la estrategia básica de inferencia por herencia.
- *Valores por defecto en los slots*. Los frames disponen de diversos mecanismos (valores propios o inferidos por herencia) para inferir el valor de los slots. Sin embargo, existen ocasiones en las que dichos mecanismos no proporcionan ningún valor útil. En estos casos los frames proporcionan todavía otro mecanismo de inferencia por defecto por el que se proporciona el valor a utilizar en esta situación.
- *Clase de valores y restricciones en el rango de los valores de los slots*. Los frames proporcionan también la posibilidad de especificar la clase de valores que puede almacenar el slot. Además, para un frame concreto puede que no sean válidos todos los posibles valores de la clase a la que pertenece el slot. Generalmente los frames proporcionan también la posibilidad de especificar el rango de valores correctos. En el caso de sistemas de información, en los que es frecuente que el usuario introduzca valores, resulta particularmente útil disponer de mecanismos automáti-

cos que permitan detectar el hecho para reaccionar adecuadamente y preservar, de esta forma, la integridad semántica del sistema.

- *Otros aspectos de los slots*. Además de ciertas propiedades estándar de los slots como las descritas anteriormente, los frames proporcionan la infraestructura para integrar otro tipo de conocimiento más dependiente del dominio de aplicación. Por ejemplo, dentro del contexto de sistemas de información, puede resultar útil centralizar en el atributo informaciones relacionadas con su apariencia en la interfaz gráfica de usuario o con el acceso al gestor de base de datos. Así por ejemplo, disponer de una propiedad con una cadena conteniendo el nombre textual del slot facilitaría la realización de tareas de entrada/salida (mostrando ese nombre al usuario le identificamos el slot) o el trabajo con su tabla de la base de datos donde tiene persistencia el objeto (nombre del campo dentro de la tabla). De esta forma el propio frame “sabe” cómo representarse o cómo darse persistencia ofreciendo así a los otros frames una funcionalidad más “inteligente”. Esta centralización del conocimiento relacionado con una entidad, además de constituir una de las recomendaciones de trabajo provenientes ya de la época de las redes semánticas, proporciona interesantes propiedades desde el punto de vista de ingeniería al limitar la duplicidad y facilitar la modificación del código.
- ***Relaciones entre entidades***. La centralización del conocimiento específico de una entidad en una estructura de datos propia y el establecimiento de asociaciones con cierta carga semántica con otras entidades con las que tiene algún tipo de relación, son utilidades importantes propuestas desde los orígenes de las redes semánticas [Woods 75] [Brachman 79].

Propiedades de comportamiento:

- Daemons *para lectura y escritura*. En esta forma de programación, cada acción es el resultado de un acceso a los datos. Los *daemons* son procedimientos ligados a slots que se invocan cuando se producen acciones de lectura o escritura del valor del slot. Los daemons pueden ser utilizados también para computar valores en el momento que se necesiten.
- *Encapsulado del acceso a los slots*. Los usuarios de un frame no deben tener acceso directo a los slots. Impidiendo el acceso exterior a la implementación de un slot se gana independencia entre los distintos frames y se posibilita la modificación de la implementación de la estructura de

datos del valor del slot sin que se vean afectados el resto de los objetos. Si bien esta es una cualidad más requerida por cuestiones de ingeniería del software, se puede emplear en los lenguajes basados en frames a través de los *daemons*.

- *Envío de mensajes*. La mayoría de los lenguajes basados en frames ofrecen una característica típica de la programación orientada a objetos que permite que los objetos representados por los frames respondan a mensajes.

Un análisis de los diferentes lenguajes basados en frames [Bobrow 77] [Fox 86] [Roberts 77] escapa de los propósitos de este trabajo. Hay que mencionar también los llamados lenguajes híbridos como BEEF [Lassila 90], LOOPS [Bobrow 83] y Kee [Kempf 87], que incorporan diferentes paradigmas como la orientación a objetos, los frames y la programación lógica. Estos lenguajes híbridos han demostrado ser muy adecuados en el ámbito de la Inteligencia Artificial donde la variedad de datos utilizados obliga a utilizar diferentes formalismos para representarlos. Posteriormente se hace un análisis de CLOS, un lenguaje híbrido basado en LISP, por ser el que incorpora las características más potentes de autorrepresentación por medio de un MOP.

1.3. Ejemplos de lenguajes con propiedades reflexivas

1.3.1. Smalltalk

Smalltalk tiene sus orígenes a principios de los años 70 con la aparición de Smalltalk-72 inspirado en Simula-67. El lenguaje fue evolucionando apareciendo nuevas versiones. Smalltalk-80 [Goldberg 83] es la última versión estable y ha alcanzado una popularidad importante, principalmente en entornos de investigación. Es un lenguaje orientado a objetos puro y uniforme. Todas las entidades, incluidos los elementos básicos como números y caracteres, son objetos. También son objetos los métodos (mensajes que se envían a los objetos) y las clases. Las clases, al ser objetos, deben ser instancias de otras clases que se denominan metaclases. Un amplio conjunto de clases determina el comportamiento de las estructuras del propio lenguaje.

Smalltalk-80 modifica de modo importante la estructura de metaclase única de Smalltalk-76. Cada clase de Smalltalk-80 tiene asociada una metaclase de la

que es su única instancia. Se sigue por tanto el modelo *metaobjeto por clase* en el que la metainformación no se almacena en la propia clase sino en un objeto asociado a ella de forma unívoca. Cada vez que se crea una clase, automáticamente se crea su metaclase. Si A es una clase, su metaclase es A class. Además si A deriva de B, automáticamente A class deriva de B class. Con esto se soluciona el problema de la compatibilidad de metaclases descrito al final de 1.2.3.

```
Object subclass: #Persona
instanceVariableNames:'nombre edad'
classVariableNames: ''
poolDictionaries: ''
category: 'Ejemplo'

p <- Persona new
p class instVarAt: 1 put: 'Pepe'
```

Figura 8. Ejemplo de acceso por índice a un atributo en Smalltalk

En la figura 8 se muestra un ejemplo de declaración de una clase Persona con atributos nombre y edad[2]. A continuación se crea un objeto p perteneciente a esta clase. Por último se accede a su clase con p class y se llama al método instVarAt:put: que toma dos parámetros, el número de atributo que se desea cambiar y el nuevo valor. Este ejemplo muestra la naturaleza completamente dinámica del lenguaje. No hay ninguna comprobación estática de tipos. Clases y métodos se pueden crear y modificar dinámicamente e incluso se puede cambiar (con ciertas restricciones) la clase a la que pertenece un objeto.

Todas las metaclases son instancias de una única metaclase llamada Metaclass. La jerarquía de clases se complica con la introducción de la metaclase Class de la que derivan todas las clases normales. Los aspectos comunes de Class y Metaclass se incluyen en la clase padre de ambas ClassDescription. Ésta hereda de Behaviour. Todo el sistema de clases hereda de una clase común: Object (figura 9).

[2] En los ejemplos de código se utiliza un tipo de letra de ancho proporcional para evitar la partición artificial de líneas que se derivaría de utilizar un tipo de letra con ancho constante. Asimismo, se destaca en subrayado la declaración de identificadores para facilitar la lectura del código.

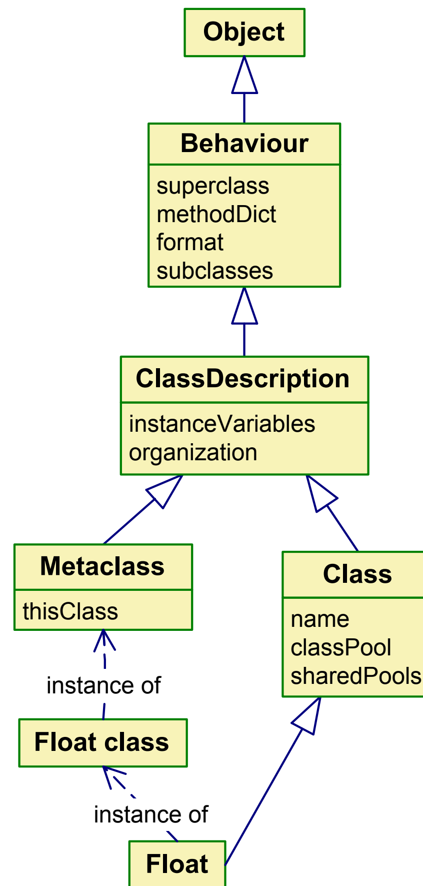


Figura 9. Jerarquía de metaclases en Smalltalk

Smalltalk proporciona facilidades de introspección e intervención. Enviando mensajes al sistema de metaobjetos se puede acceder y modificar la lista de atributos de una clase (instanceVariables en ClassDescription), la jerarquía de clases (superclass y subClasses en Behaviour), cambiar la clase de una instancia, etc. Además, el propio proceso de generación de código está incluido en el propio lenguaje gracias a clases como Parser, Compiler, ProgramNode, etc. Por tanto, es posible extender el lenguaje o cambiar su semántica por medio de estos metaobjetos. [Rivard 96] personaliza las clases Metaclass, Parser y Compiler (mediante herencia) para modificar la generación de código incluyendo pre y postcondiciones.

El modelo de metaclases en Smalltalk queda un tanto complejo y confuso [Borning 87]. Además, la generación automática de la metaclase de cada clase

y el hecho de que si una clase deriva de otra, sus metaclases respectivas también heredan la una de la otra, produce una jerarquía rígida que da lugar a limitaciones de expresividad como se puede ver en [Briot 89] y [Cointe 93]. En estos trabajos se muestra un ejemplo de una arquitectura que mejora las capacidades reflexivas del lenguaje permitiendo la creación explícita de metaclases. Estas metaclases permiten agrupar propiedades comunes de metaclases evitando duplicación de código.

1.3.2. CLOS

CLOS [Bobrow 88] es una extensión orientada a objetos de Common Lisp basada en versiones anteriores de Lisp con objetos: LOOPS [Bobrow 83] y Flavors [Moon 86]. Como extensión de Lisp, CLOS es un lenguaje flexible, dinámico y de gran potencia representativa. Admite herencia múltiple y el polimorfismo se implementa por medio de funciones genéricas que permiten seleccionar el método que se ha de ejecutar teniendo en cuenta todos los parámetros de la llamada (*multimétodos*) y no sólo el primero como suele ser habitual.

CLOS, como Lisp, tiene capacidades de metaprogramación debido a que los programas se representan como datos y a que el proceso de compilación y ejecución puede controlarse por código escrito en el propio lenguaje. Las macros permiten además construir nuevos elementos sintácticos a partir de otros.

Ya sólo con estas características se puede hablar de capacidades reflexivas, pero además, CLOS introduce una capa nueva en la que las estructuras que definen e implementan el lenguaje como los métodos, las clases, las funciones genéricas y los slots (atributos) son asimismo clases que pueden personalizarse. Gracias a esto se puede variar el comportamiento del lenguaje en tiempo de ejecución. El modelo interno de la implementación del lenguaje es así modificable por el programador y recibe el nombre de CLOS Metaobject Protocol [Kizcales 91]. En este sentido, CLOS y Smalltalk tienen puntos en común aunque el sistema de metaclases es completamente diferente.

En CLOS cada objeto pertenece a una clase y cada clase es a su vez un objeto y por tanto es instancia de otra clase llamada su metaclase. A diferencia de Smalltalk no hay ligadura fija entre la clase y su metaclase de modo que al declarar una clase se puede especificar libremente cuál es su metaclase. Además, varias clases pueden ser instancias de la misma metaclase. CLOS adopta por tanto un modelo uniforme de objetos y clases. La única diferencia concep-

tual entre un objeto que no es clase y una clase es que una clase puede crear objetos. CLOS no exige que la derivación de clases se propague a sus metaclases con lo que surge el problema de la compatibilidad mencionado anteriormente.

En la figura 10 se puede ver la estructura básica de la jerarquía de clases e instancias en CLOS.

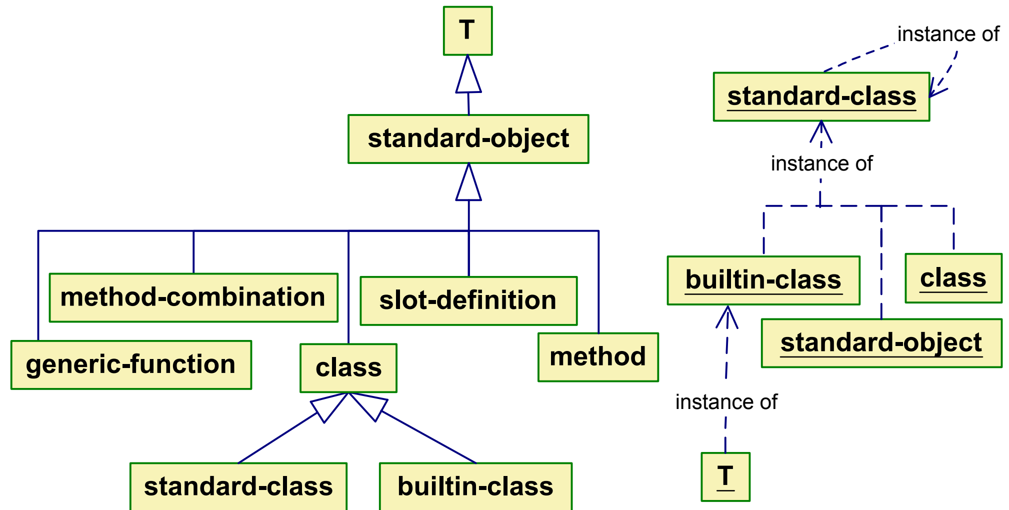
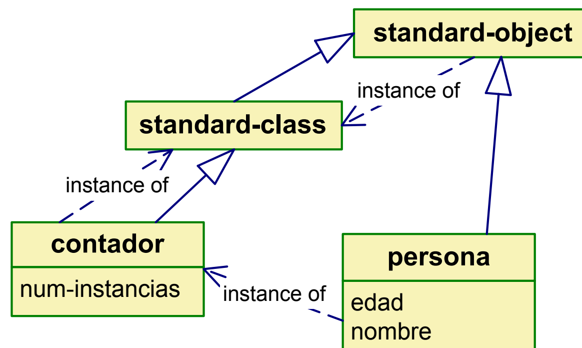


Figura 10. Jerarquía de clases e instancias en CLOS

T es la clase raíz de la jerarquía de clases. class representa el comportamiento básico de las clases. builtin-class es la raíz de las clases predefinidas.

El comportamiento estándar de CLOS viene determinado por la jerarquía de clases que heredan de dos clases raíz: standard-class (raíz de las metaclases estándar; todas las clases son instancias directa o indirectamente de ella) y standard-object (raíz de las clases estándar; todas las clases heredan directa o indirectamente de ella).

Las estructuras básicas de CLOS (clases, atributos, métodos, funciones genéricas y combinación de métodos) se representan por las clases class, slot-definition, generic-function, method y method-combination. Todas ellas derivan de standard-object.




```
(defclass contador (standard-class)
     ((num-instancias :initform 0))
     (:metaclass standard-class))
)
(defclass persona (standard-object)
     (edad nombre)
     (:metaclass contador))
)
(defmethod make-instance :before ((clase contador) &key)
     (incf (slot-value clase 'num-instancias)
)
```

Figura 11. Asociación de una metaclase contador a una clase

En la figura 11 se puede ver un ejemplo de creación de una clase persona en el que su metaclase no es standard-class sino una clase contador que lleva cuenta del número de personas que se vayan creando (en el atributo num-instancias). El método genérico make-instance toma un argumento, que es una instancia de una metaclase. En el ejemplo se ha definido una versión para el caso de que la metaclase sea de tipo contador. Lo que hace es obtener el valor de su atributo num-instancias con el método genérico slot-value incrementándolo en una unidad con setf.

Este ejemplo es sólo una pequeña muestra de la flexibilidad de CLOS para trabajar con metaclases. La capacidad del sistema va mucho más allá. Se puede

utilizar el MOP para crear un sistema de objetos paralelo al determinado por standard-class y standard-object derivando de un par de clases análogo, una será la raíz de las clases y otra la raíz de las metaclases de ese nuevo sistema. Normalmente, estas clases personalizarán el comportamiento estándar y por tanto, lo lógico es que deriven de las correspondientes clases estándar para no tener que duplicar las características que no se modifiquen.

[Attardi 93] utiliza las clases atomic-class y atomic-object para crear un sistema de concurrencia basado en objetos atómicos (objetos que admiten sincronización mediante exclusión mutua de métodos que actualizan su estado). Un ejemplo de clases análogas para implementar clases cuyos objetos admitan persistencia en bases de datos puede verse en [Paepcke 93].

1.3.3. Java

Java [Horstmann 99] es un lenguaje que ha alcanzado una popularidad notable gracias a su aplicabilidad en Internet y a su independencia de la arquitectura (que se consigue compilando para una máquina virtual). Java es un lenguaje orientado a objetos que admite herencia simple y una forma limitada de herencia múltiple: se puede heredar de más de una clase si estas clases extra son *interfaces*. Son éstas clases degeneradas en las que todos los métodos son abstractos.

Java es fuertemente tipificado y más rígido que los lenguajes que se acaban de citar. Las clases no son elementos de primer orden: no se pueden crear en tiempo de ejecución ni modificar su estructura ni usarse como variables. Estas limitaciones se reflejan en su soporte para la reflexión: el lenguaje no tiene plenas capacidades reflexivas ya que su semántica es inalterable. No obstante, incluye un sistema sencillo de metaobjetos [Sun 97b] [McCluskey 98] que permite realizar tareas introspectivas sobre la estructura de las clases.

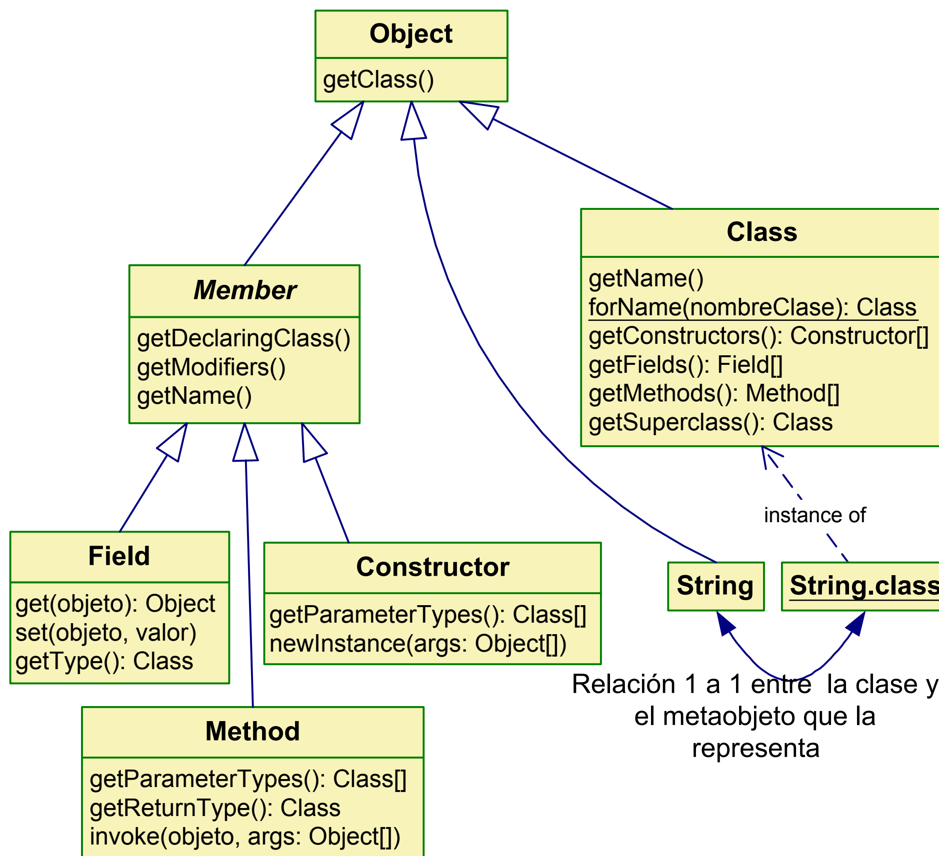


Figura 12. Jerarquía de metainformación en Java

La figura 12 muestra un esquema de las clases existentes para dar soporte a la reflexión en Java.

Object es la clase raíz de la jerarquía de clases. Todos los tipos de datos salvo los primitivos son clases y por tanto son descendientes de Object. Para cada tipo básico hay una clase asociada que actúa como *wrapper*. De este modo se consigue que cualquier conjunto de variables se pueda representar como un vector de Objects.

Las estructuras del lenguaje, es decir, las clases, los métodos, los atributos, los paquetes de clases, etc. no son elementos básicos del lenguaje y no se pueden usar como variables ni realizar operaciones con ellos. No obstante, para algunos de estos elementos, existen metaobjetos que permiten describirlos y realizar operaciones de lectura en ellos.

Cada clase tiene un metaobjeto asociado que pertenece a la metaclase Class, única metaclase del sistema[3]. Todas las clases (identificando la clase con su metaobjeto asociado) son instancias de la metaclase Class. El sufijo .class se usa para obtener en tiempo de compilación el metaobjeto asociado a una clase. Así, el metaobjeto asociado a una clase Persona sería Persona.class. Los métodos de Class permiten acceder al nombre de la clase y a su estructura interna: por ejemplo pueden acceder a la lista de métodos y atributos de la clase así como a su clase padre. Los atributos y métodos tienen metaobjetos asociados que pertenecen a las clases Field y Method. Así, el método que devuelve los métodos de una clase lo hace en forma de vector de Method. Los constructores (funciones especiales para iniciar objetos) pertenecen a la clase Constructor. Estas tres clases implementan la interfaz Member. Aunque los objetos de estas clases no se pueden modificar sí se puede acceder a su función. Así, para el caso de los atributos se puede obtener su valor para un objeto concreto. En el caso de las instancias de Method, dados el objeto y los argumentos se puede realizar una llamada al método asociado. La figura 13 muestra un ejemplo elemental de código que accede a la metainformación de una clase en tiempo de ejecución.

El soporte para la reflexión de Java presenta ciertas limitaciones debido a su naturaleza cerrada que impide personalizar o añadir metainformación. En la elaboración del trabajo presentado en [Fernández 00] se intentó aprovechar las posibilidades de introspección de Java ya que el servidor debía proporcionar información acerca de los servicios con los que contaba. La rigidez de las posibilidades reflexivas de Java obligó a realizar una implementación manual de estos servicios.

---

[3] Única metaclase si se considera que las metaclases son aquellas clases cuyas instancias son clases. Si se consideran como metaclases las clases cuyos objetos describen estructuras del lenguaje también lo serían por ejemplo Method y Field.

```
class Persona {
    string nombre;
    int edad;
};
...
Object o = new Persona();
Class c = o.getClass(); //Obtención del metaobjeto de o (Persona.class)
c = Class.forName("Persona"); //Metaobjeto a partir de su cadena
for(int i = 0; i < c.getFields(); ++i) { //Recorro los campos
    Field f = c.getFields()[i]; //Campo n-ésimo de la clase
    System.out.print(f.getName()); //Nombre del campo
    System.out.print(f.get(o)); //Valor del campo f para el objeto o
}
```

Figura 13. Ejemplo de acceso a metainformación en Java

En cualquier caso, hay varios campos donde ha resultado muy útil el sistema de reflexión de Java:

- Programación por componentes (*Java Beans* y *Enterprise Java Beans* [Roth 98]) tal y como se ha explicado en el apartado 1.1.3.
- Programación distribuida mediante la creación y ejecución de objetos ubicados en plataformas remotas [McCluskey 87] [Sun 98].
- Persistencia mediante la transformación de objetos en un flujo de datos [Coker 97] [Sun 97]. El acceso a la metainformación permite definir e implementar una representación de los objetos en forma de *stream* que puede guardarse en soporte permanente o enviarse por un canal de comunicación.

Se han creado varios sistemas para dotar a Java de mayores capacidades de reflexión en la línea de los lenguajes anteriormente citados. OpenJava [Chiba 98] [Tatsubori 99] usa la misma filosofía que OpenC++ [Chiba 95], sistema que se describe más abajo. [Tatsubori 98] presenta un ejemplo de uso de OpenJava para utilizar como entidades básicas del sistema patrones de objetos como *Adapter* y *Visitor* [Gamma 96]. Por otro lado, *metaXa* [Golm 98] utiliza un sistema de metaobjetos que controlan el comportamiento de los objetos básicos

mediante eventos que se disparan al acceder a un objeto básico y que son capturados por el metaobjeto para realizar acciones a metanivel. Para implementar el sistema de metaobjetos se hace uso de una versión extendida de la máquina virtual Java. Otra aproximación puede verse en [Braux 99]. En ella, se intentan computar en tiempo de compilación las llamadas al metanivel para ganar en eficiencia.

1.4. Metainformación en C++

C++ [Stroustrup 97] es un lenguaje de programación que, tomando como base el lenguaje C [Kernighan 98] [C 89], añade un conjunto notable de características como son la orientación a objetos, el manejo de excepciones y las plantillas de clases. Su (casi) compatibilidad ascendente con C, su potencia y su eficiencia han hecho de él el lenguaje orientado a objetos más empleado en las aplicaciones profesionales. Sin embargo, C++ presenta una serie de problemas para la programación de los aspectos necesarios para dar soporte a la metainformación que no surgen cuando se utilizan otros lenguajes de programación más clásicos en el área de IA como CLOS. La realización de la implementación de una arquitectura de metainformación en C++ hace añorar algunas utilidades de programación en el desarrollo de sistemas basados en frames en CLOS entre las que cabe destacar las siguientes:

- Posibilidad de manejar los programas (tipos, clases, código) como datos.
- Posibilidad de crear nuevas clases y modificar las existentes en tiempo de ejecución.
- Acceso en tiempo de ejecución a la estructura de las clases (nombres, atributos, métodos) pudiendo invocarse un método pasando simplemente una cadena con su nombre.
- Posibilidad de trabajo sin restricciones de tipos, frente a la programación fuertemente tipificada de C++ que proporciona a éste último lenguaje una mayor eficiencia y ventajas en la de detección de un mayor número de errores en tiempo de compilación.

C++ proporciona soporte mínimo para trabajar con metainformación. Esto no debe extrañar ya que es una consecuencia lógica de algunos objetivos básicos que guiaron el diseño del lenguaje [Stroustrup 94]:

- Compatibilidad (en lo posible) con C.
- Eficiencia en tiempo de ejecución.
- Comprobación de tipos en tiempo de compilación.
- No incluir una característica al lenguaje que produzca una disminución de la eficiencia de un programa que no use dicha característica. Esta propiedad se conoce popularmente como *Don't pay for that you don't use* o *The Zero Overhead Rule*.

Estos objetivos se consiguen en buena medida trasladando a tiempo de compilación muchos de los cálculos que en otros lenguajes orientados a objetos deben efectuarse en tiempo de ejecución. En gran parte de los casos, el método que ha de llamarse como consecuencia del envío de un mensaje a un objeto es conocido durante la compilación. Gracias a ello, el envío de dicho mensaje se implementa por medio de una llamada simple a una función siguiendo el modelo de bajo nivel de C, con lo cual la eficiencia está garantizada. Solamente en el caso de que no se conozca el tipo exacto del objeto y que el método llamado se haya declarado como *virtual* debe hacerse un pequeño cálculo en tiempo de ejecución para determinar el método real que debe llamarse. Esto se consigue guardando para cada objeto un puntero a un objeto especial con información de la clase a la que pertenece realmente dicho objeto. Este objeto informativo se denota por vtbl y contiene una tabla con las direcciones de los métodos de la clase a los que se aplica polimorfismo (métodos virtuales).

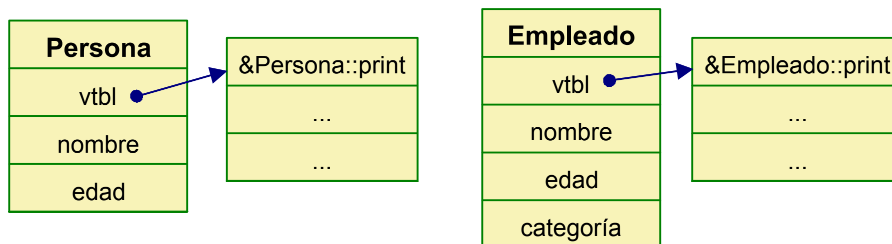


Figura 14. Esquema de memoria de las clases de C++

La figura 14 presenta un esquema simplificado de la representación en memoria de los objetos de una clase y de otra que deriva de ella. Cuando se hace una llamada a un método virtual, se consulta la tabla de métodos virtuales del objeto para obtener la dirección del método que hay que llamar y se hace la

consiguiente llamada. En el caso de haber herencia múltiple el esquema es más elaborado pero en cualquier caso unas pocas indirecciones son suficientes para determinar el método de disparo. En [Ellis 90] se describen con detalle las técnicas de implementación en C++ del polimorfismo.

Como se ve, únicamente se necesita guardar un objeto para cada clase con la lista de métodos virtuales. Este objeto, que sería el metaobjeto de la clase que representa, no es visible para el programador en tiempo de ejecución, de modo que ni siquiera se puede obtener una lista de los métodos de una clase.

La falta de información de tipos en tiempo de ejecución dificulta la cooperación entre programas y el almacenamiento y envío de objetos. También obliga a realizar a mano tareas de trazado y depuración. Sin información de tipos, las soluciones a estos problemas son rudimentarias, poco potentes, poco flexibles y de difícil reutilización. Por ejemplo, si un módulo C++ quiere colaborar con otro es necesario compilarlo incluyendo el fichero cabecera de ese otro módulo. Las limitaciones son obvias: no se puede realizar la colaboración dinámicamente, el cambio de uno de los módulos obliga a recompilar, etc. Si se desea colaborar con otra herramienta se puede recurrir a ficheros compartidos para intercambiar información, teniendo que realizarse esta tarea de forma manual para cada aplicación. Una mayor información de las clases en tiempo de ejecución proporcionaría el conocimiento necesario para publicar los objetos y servicios de un módulo y hacerlos accesibles al resto del mundo de manera uniforme.

1.4.1. Posibilidades de metainformación en C++

A la hora de construir un sistema de metainformación hay que diseñar cuidadosamente el alcance de dicho sistema [Strickland 93]. Un sistema sofisticado y completo permitirá encontrar soluciones adecuadas a un rango importante de problemas pero a cambio el sistema será más complejo y pueden surgir problemas de mantenimiento y de eficiencia. Se citan a continuación las diferentes piezas de metainformación que pueden adjuntarse a los programas y sus posibles usos:

- **Información de las clases**. A cada clase se le asigna un objeto con información de la clase (por ejemplo el nombre de la clase). Un método virtual permitirá preguntar a cualquier objeto cuál es el objeto represen-

tativo de la clase a la que pertenece. Esto puede ser útil para tareas de depuración.

- **Información de la jerarquía de clases**. Si se guarda información de la herencia de clases se pueden hacer conversiones dinámicas de tipos, es decir, dado un objeto que pertenezca a una clase B se puede saber si dicho objeto se puede asignar a un objeto de clase A. Para ello, el tipo dinámico del objeto debe ser A o descendiente de A. Esta característica tiene varias posibilidades de aplicación [Lea 92]. Por ejemplo, es común tener una lista heterogénea de objetos de la cual se desea extraer aquellos que pertenezcan a (o deriven de) una clase concreta (figura 15).

```
vector<Persona>* v = ...
for(int i = 0; i < v.size(); ++i) {
 if (Empleado* e = (Empleado*)v[i]) //si v[i] es un Empleado
      ...
}
```

Figura 15. Comprobación dinámica de conversión de tipos

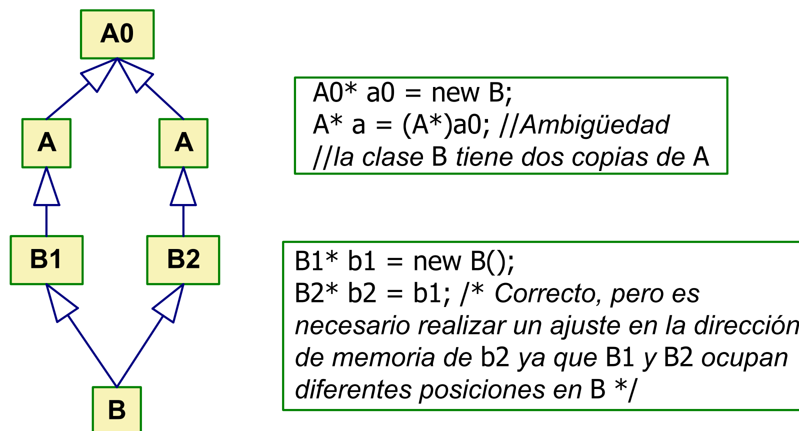


Figura 16. Conversiones de clase derivada a clase base

Otra aplicación consiste en la simulación de multimétodos (mensajes polimórficos que se resuelven según la clase a la que pertenezcan varios parámetros) [Stroustrup 94].

Hay que hacer notar que para comprobar si un B se puede asignar a un A no basta con comprobar que B hereda de A pues pueden surgir problemas relacionados con la herencia múltiple y virtual [Reeves 97] (figura 16).

- **Información de los atributos de una clase**[4]. Esta información consiste en la identificación de la clase en la que se ha declarado el atributo y el desplazamiento (puntero a miembro) del atributo dentro de dicha clase. Normalmente interesa también guardar también el nombre del atributo, para depurar o para comunicarse con bases de datos en las que se accede a los atributos por medio del nombre. El acceso a la estructura interna de las clases permite implementar automáticamente servicios para exportar el estado (valor) de un objeto, bien para almacenarlo en memoria permanente bien para enviarlo por un canal de comunicación a otro programa. Esto se puede hacer siguiendo el patrón *Serializer* de [Riehle 95]. Implementaciones de este patrón en C++ pueden verse en [Grossman 93] y [Cohen 96]. Para guardar un objeto basta obtener de su clase la lista de atributos y, para cada uno de estos atributos hay que obtener el valor de dicho atributo para el objeto en cuestión, para a continuación llamar recursivamente al método de guardar. También se debe guardar el nombre de la clase para su posterior recuperación (figura 17).

  La tarea de recuperación de un objeto es más complicada. En primer lugar hay que leer el nombre de la clase y a partir de él construir un objeto nuevo de ese tipo. Esto se puede conseguir manteniendo una lista de las clases del programa (en realidad una lista de los objetos que representan a las clases del programa). Recorriendo esa lista se puede localizar la clase cuyo nombre coincida con el leído. Una vez en posesión del objeto-clase en cuestión se necesita construir un objeto nuevo de esa clase. Para ello basta con añadir a los objetos-clase un atributo con la dirección de un método constructor. Una vez creado el objeto se rellenan sus atri-

[4] Los términos "atributo", "campo" y "objeto miembro" son sinónimos en C++. La palabra "atributo" se emplea en el contexto de orientación a objetos. "Objeto miembro" es el término utilizado por el autor del lenguaje y se usa especialmente en C++. La palabra "campo" se utiliza heredada del lenguaje C.

butos en una operación de recorrido en lectura similar a la utilizada para guardarlos. En la práctica, el sistema debe ser más sofisticado debido a la presencia de campos privados y de punteros o referencias a otros objetos que obligaría a guardar para cada objeto un código de identificación.

```
void guardar() {
 guardar(nombreClase());
 for(int i = 0; i < listaAtributos().length(); ++i) {
      guardar(listaAtributos[i]);
 }
}
```

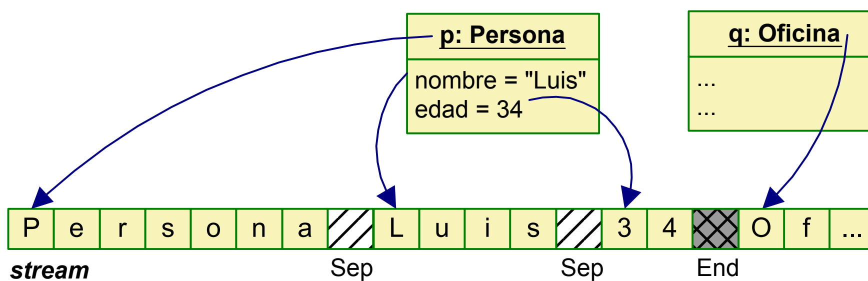


Figura 17. Volcado de información en un stream

En cualquier caso se observa que la solución se puede aplicar de forma genérica a todas las clases. Sin estas capacidades reflexivas hay que especializar el mecanismo de guardado y recuperación para cada clase y cualquier cambio en la lista de atributos obliga a modificar el código de guardado y recuperación.

- **Información de métodos**. La información de atributos permite exportar el estado de un objeto hacia otros programas (o hacia el propio programa, en una ejecución posterior del mismo). Este sistema puede resultar ineficiente en casos donde se desee una interacción dinámica constante entre dos programas, ya que obligaría a guardar el estado del objeto cada vez que el otro programa necesitara leerlo. La información de métodos proporciona una interfaz a otros programas para enviar mensajes remo-

tos a los objetos sin necesidad de que éstos guarden o envíen su estado. El mensaje normalmente incluirá el nombre del objeto, el nombre del método que se debe ejecutar y los parámetros. El programa servidor interpretará esta información gracias a la metainformación de métodos, que le permitirá deducir el método que se ha de llamar a partir de su nombre. La metainformación de métodos debe incluir para cada clase su lista de métodos y para cada método de esta lista deberá guardarse su nombre, una lista de los tipos de sus parámetros y del tipo devuelto y un puntero a la dirección del método para poder ejecutarlo. Si se desea permitir sobrecarga en los nombres de los métodos se buscará en la lista de métodos aquél cuya signatura coincida con la de la llamada.

1.4.2. Soporte para la metainformación en el estándar del lenguaje

Durante el proceso de estandarización de C++ se observó la necesidad de incluir algún tipo de herramientas para el manejo de tipos en la línea anteriormente expuesta. En [Stroustrup 92] se hace una propuesta para añadir al lenguaje información de tipos en tiempo de ejecución (RTTI) que en esencia es aceptada por el estándar ISO/ANSI C++ [C++ 98]. Para cada clase se genera un objeto perteneciente a la clase `type_info`. Se puede obtener el `type_info` de un objeto o una clase con el operador `typeid`. Desafortunadamente, la interfaz de `type_info` apenas permite algo más que preguntar por el nombre de la clase o comparar dos `type_info`s tal y como se ve en la figura 18. De los cuatro tipos de información antes citados el estándar sólo cubre, por tanto, los dos primeros. Este soporte mínimo sigue la filosofía antes citada de no sobrecargar los programas que no hagan uso de ella.

```
Persona* p = new Empleado();
cout << (typeid(*p) == typeid(Empleado)); //true
cout << typeid(*p).name(); //Empleado
//Comprobación de que p es un Empleado o derivado de Empleado:
if(dynamic_cast<Empleado*>(p)) ...
```

Figura 18. Metaclase estándar de C++ type_info

Por tanto, en tiempo de ejecución el programador tiene acceso a un conjunto mínimo de metainformación. Las clases y los atributos (los atributos en sí, no los valores que toman para un objeto concreto) no son entidades con las que se puedan hacer operaciones, no son ciudadanos de primera clase. No se pueden hacer preguntas del tipo, "¿qué atributos tiene esta clase?" o "¿cuál es el nombre de este atributo y a qué clase pertenece?". Esta información se pierde en su mayor parte en tiempo de compilación. Así, cuando se accede a un atributo de una instancia con la notación p.nombre se está accediendo simplemente a un string. En el código ejecutable no se guarda ninguna información de que p.nombre es en realidad el atributo nombre del objeto p de la clase Persona.

Por supuesto, si ni siquiera se pueden leer las propiedades de una clase mucho menos se pueden modificar: no se pueden añadir nuevos atributos en tiempo de ejecución ni modificar los existentes.

```
Persona p = ...
//Defino un tipo para guardar atributos de Persona de tipo entero:
typedef int Persona::* AtributoIntPersona;
//Creo una variable de dicho tipo y le asigno el atributo edad:
AtributoIntPersona atributo = &Persona::edad;
//Accedo al valor de ese atributo para una persona concreta
p.*offset = 20;
/* Defino un tipo de datos para representar métodos de Persona sin parámetros y sin valor de devolución: */
typedef void (Persona::* MetodoVoidPersona)();
//Creo una variable de ese tipo y le asigno el método imprime:
MetodoVoidPersona metodo = &Persona::imprime;
//Llamo al método para una persona concreta:
(p.*metodo)();
```

Figura 19. Punteros a miembro en C++

En lenguajes como Smalltalk se puede enviar cualquier mensaje a cualquier objeto. Esta flexibilidad implica la necesidad de incluir en el entorno de ejecu-

ción información sobre la estructura de los objetos y las clases ya que sólo en tiempo de ejecución se detecta si la llamada es incorrecta.

Ya se ha visto que las clases no son elementos de primer orden en el lenguaje. Otros elementos como los métodos y los atributos admiten algunas posibilidades de manejo por medio de los *punteros a miembro*. En el caso de los atributos son variables que representan el desplazamiento de la posición de un atributo dentro de una clase. En el caso de los métodos guardan simplemente la dirección de la función asociada, permitiendo realizar una llamada a dicha función por medio de la variable en cuestión. La figura 19 muestra un ejemplo de su uso.

Hay que hacer notar que la única capacidad de estas variables es la de asignación y la de acceder a su valor para un objeto dado o la de ejecutar el método asociado dados los parámetros.

El soporte directo de C++ para metainformación se puede resumir en los siguientes puntos:

- Un metaobjeto para cada clase que permite acceder a su nombre a efectos de diagnóstico y depuración.
- Un sistema (RTTI) que permite identificar en tiempo de ejecución el tipo real de un objeto con el fin de poder realizar conversiones de tipo.
- Una tabla de métodos virtuales para cada clase que permite seleccionar en tiempo de ejecución el método que se ha de ejecutar en el caso de no quedar determinada en tiempo de compilación la clase a la que pertenece el objeto. El único uso de la tabla es el citado. Solamente el compilador puede hacer uso de ella ya que queda oculta al programador al no haber una definición estándar de su implementación.
- Punteros a miembro que permiten crear variables que representan atributos y métodos. La única posibilidad de estos elementos es la de realizar asignaciones y acceder al valor del atributo para un objeto dado o llamar al método asociado.
- Posibilidad de personalizar la gestión de la memoria dinámica tanto de forma global (redefiniendo los métodos `::new` y `::delete`) como en ámbito de clase (redefiniendo los métodos estáticos `new` y `delete` de la clase).

- Uso del preprocesador para metaprogramación y para implementar características especiales del lenguaje. Es bien conocido el empleo del preprocesador en C para representar constantes, estructuras de iteración o simulación de estructuras parametrizadas por tipos de datos. No obstante, el desconocimiento por parte del preprocesador de reglas de ámbito, tipos y visibilidad da lugar a grandes problemas de mantenimiento. Las características que ha añadido C++ han permitido disminuir el número de tareas en las que era necesario utilizar el preprocesador.

- Algunas directivas del lenguaje son ejemplos de reflexión pues permiten controlar la implementación del código. Por ejemplo el modificador `inline` sirve para indicar que una función no debe compilarse sino que debe insertarse su código en aquellos lugares donde se haga la llamada (*funciones de expansión*).

- Uso de plantillas. Las plantillas permiten parametrizar clases y funciones por tipos. La importancia de las plantillas en C++ es grande permitiendo aplicar técnicas novedosas de programación con un espectro amplio de aplicación. Más adelante se analiza el uso de las plantillas para aplicar técnicas de metaprogramación.

1.4.3. Aproximaciones para añadir metainformación a C++

El reducido soporte que proporciona directamente C++ en cuanto a metainformación obliga a desarrollar *frameworks* o librerías para añadir características más potentes. Hay dos técnicas básicas: realización de MOPs con la posibilidad de modificar el comportamiento del modelo de objetos y creación de estructuras que permitan introspección para acceder en tiempo de ejecución a las clases, los atributos, los métodos, etc.

Se presenta a continuación una clasificación básica de las diferentes técnicas de implementación de estos sistemas.

- *Precompiladores dinámicos*. Mediante una ampliación de la sintaxis de C++ permiten declarar metaobjetos y metaclases para personalizar algunas de las características del modelo de objetos: creación y borrado, envío de mensajes, herencia, invocación de métodos y otras. También proporcionan acceso a todas las estructuras del programa por medio de los metaobjetos: clases, objetos, métodos, código fuente, objetos miembro, jerarquía de clases. Estos sistemas se implementan mediante un precom-

pilador que sustituye las acciones normales de C++ por llamadas a metaobjetos en los que se delegan las tareas. Así, la llamada p.f() es sustituida por p.meta().process(p, f) donde meta() proporciona acceso al metaobjeto asociado a p y process es un método del metaobjeto que procesa la llamada a f en el objeto p. Dentro de este método se hará una llamada al método original (figura 20).

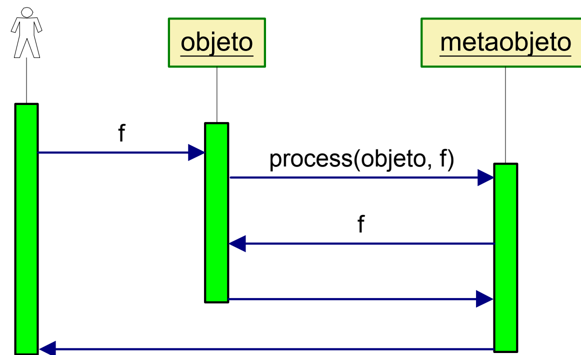


Figura 20. Delegación de la llamada al metaobjeto

Estos sistemas son muy flexibles pero acarrean una importante disminución de prestaciones. Ocupan mucha memoria al tener que incluir un sistema amplio de metainformación en tiempo de ejecución. La eficiencia se resiente también ya que la llamada a un método implica realizar algunas indirecciones adicionales.

Iguana [Gowing 96] es un MOP para C++ inspirado en el MOP de CLOS que permite incluir en un programa diferentes modelos de objetos El entorno proporciona metaclases para dar soporte a algunos modelos típicos de datos. Dinámicamente puede cambiarse el modelo de datos asociado a un objeto.

- *Precompiladores estáticos*. En ellos, la sustitución de código actúa en tiempo de compilación, es decir en el código generado no aparecen los accesos al metaobjeto sino que se incrusta directamente la funcionalidad del metaobjeto en cuestión. De este modo se consigue incrementar la eficiencia aunque se pierde cierta flexibilidad durante la ejecución al no poder modificarse dinámicamente la política del modelo de objetos. Un ejemplo es OpenC++ Versión 2 [Chiba 95]. MPC++ [Ishikawa 94] utili-

za la misma aproximación estática de OpenC++. Su MOP permite extender la sintaxis del lenguaje para añadir nuevas características. Esta implementación abierta lo convierte en un verdadero metacompilador o generador de compiladores. Fog [Willink 99] dispone de un precompilador de un metalenguaje que actúa sobre el código fuente. El programador de este metalenguaje tiene a su disposición objetos con toda la estructura de las clases y métodos y con estos datos puede generar o modificar código escribiendo un metaprograma que maneje esos datos.

- *Creación de un lenguaje y compilador nuevo*. Muchos lenguajes orientados a objetos han nacido como extensión de un lenguaje tradicional al que se han añadido características para dar soporte al nuevo paradigma. Se pueden citar C++ como extensión de C, Ada95 como extensión de Ada83, Object Pascal como extensión de Pascal, etc. Siguiendo esta filosofía se puede crear un nuevo lenguaje como extensión de C++ que incorpore posibilidades de reflexión. Naturalmente esto implica la realización de un proyecto nada trivial pues se trata de diseñar un nuevo lenguaje e implementar su compilador. En algunos casos las adiciones que se realizan son tan pequeñas que al lenguaje generado se le sigue llamando C++. C++Builder [Inprise 00] enriquece la potencia y semántica de los atributos añadiendo métodos de lectura y escritura:

```
class Persona {
 private:
  String getNombre() {... return nombre;}
  String setNombre(String nom) { ...; nombre = nom; }
 public:
  __property String
       nombre = {read = getNombre, write = setNombre};
  ...
};
```

Figura 21. Deamons *de lectura escritura en C++Builder*

También se puede caracterizar a los atributos con la propiedad `__published` lo que hace que en el código generado se incluya metainformación útil para el diseño de componentes.

- *Generadores de metainformación a partir del código fuente*. Estos sistemas utilizan un *parser* que recorre los ficheros cabecera de las clases generando un fichero C++ con esta metainformación. Este fichero se compila y se enlaza con el resto de ficheros para generar la aplicación.

  Este modelo respeta el código fuente original y con ello permite la integración de librerías creadas por terceras partes. La desventaja de este sistema radica en la dificultad de la realización del *parser* (un verdadero compilador) y que sólo permiten generar metainformación para clases concretas. La generación de metainformación para plantillas de clases obligaría a realizar un *parser* muy sofisticado y la arquitectura de la solución se complicaría extraordinariamente.

  Root [Brun 96] utiliza esta aproximación. Su *parser* genera la siguiente información para clase: nombre, tamaño de las instancias, lista de clases padres, nombres, tipos y descripciones de las instancias, nombres y signaturas de los métodos, referencia al fichero fuente y dirección de un método estático factoría para crear instancias. El sistema se aprovecha de esta información para implementar persistencia utilizando una base de datos propia.

  [Chuang 98] también realiza una exploración de las declaraciones de las clases y genera un metaobjeto por clase. Este metaobjeto guarda el nombre de la clase, referencias a los metaobjetos padre, los nombres de atributos más su posición relativa con respecto al comienzo del objetos y los nombres y direcciones de los métodos. El carácter virtual o abstracto de las clases también se guarda. Tiene la desventaja de que el manejo de la metainformación posterga a tiempo de ejecución todas las comprobaciones de tipos.

  MIP [Buschmann 92] es un proyecto más ambicioso pues asocia un objeto con información del tipo no sólo a las clases sino también al resto de tipos, como los enumerados, las uniones, los *arrays* primitivos e incluso los typedefs. En el caso de las clases se puede obtener información de sus clases padre, sus clases amigas, sus atributos y sus métodos. En cuanto a los métodos se puede obtener la lista de sus parámetros y el tipo devuelto. Un parser llamado *MIP-generator* genera toda la información necesaria. Actúa justo después del preprocesador de C++. Las ideas sobre el patrón de reflexión explicadas en [Buschmann 96b] se apoyan en esta librería.

- *Generadores de metainformación a partir de la información de depurado*. Estos entornos compilan los ficheros con opciones de depurado activadas y a partir de este código objeto generan, mediante un programa especial, un fichero fuente C++ con la metainformación. Los problemas de esta solución son la limitada cantidad de información que se genera al compilar con la opción de depurado y la pérdida de capacidades de transporte que surgen al depender la implementación del compilador.

  [Kakkad 98] usa esta técnica para implementar un sistema de generación de información de la representación en memoria de todas las variables de un programa. El operador `new` se redefine para que al asignar memoria a un objeto guarde un enlace al descriptor de su tipo. La distribución completa de la ocupación de memoria dinámica queda por tanto al alcance del programador haciendo posible la creación de entornos para el manejo de persistencia [Singhal 92] y recolectores de basura en C++ [Wilson 93]. La implementación del entorno se ha hecho para el depurador *gdb* de *gnu* [Stallman 00].

- *Generadores de metainformación por medio de macros en el código fuente*. Siguiendo la filosofía de los precompiladores, estos sistemas se aprovechan del preprocesador de macros de C++ para realizar una transformación del código. La expansión de las macros genera el código de la metainformación. Son sistemas menos flexibles y potentes que los basados en precompilador ya que sólo se puede realizar un tratamiento local de las zonas de código que representan datos de metainformación. No obstante, se pueden conseguir facilidades avanzadas de introspección. La ventaja fundamental es la simplificación del entorno de trabajo al ahorrarse un paso en el sistema de compilación. Además, en los entornos de programación en C++, el preprocesador está integrado en el sistema con lo cual se facilitan las tareas de edición, compilación, detección de errores y depuración. [CIDLib 00] utiliza macros para implementar un mecanismo de RTTI similar al estándar con la importante adición de permitir crear objetos a partir del nombre de la clase. En esta tesis se presenta otra aproximación basada en las ideas que aparecen en [Valiño 97] [Zarazaga 98b] y [Zarazaga 98c].

- *Uso de plantillas*. Por su importancia y por ser objeto de especial atención en esta tesis se dedica el apartado siguiente al estudio de las posibilidades de las plantillas.

1.5. Las plantillas en C++ como herramienta de metaprogramación

La idea inicial de incorporar las plantillas a C++ fue capturar el mecanismo de abstracción que surge cuando se desea trabajar con contenedores de datos en los que su funcionamiento es independiente del tipo de datos de los elementos. Por ejemplo, una lista doblemente enlazada de enteros funciona de la misma manera que una lista doblemente enlazada de cadenas. Las soluciones tradicionales a este problema si no se dispone de plantillas plantean graves inconvenientes (emplear el preprocesador o definir un contenedor único en el que el tipo de los elementos pertenezca a una cierta clase raíz).

La solución de C++ consiste en permitir declarar una plantilla de clases parametrizada por un tipo de datos. Esta plantilla se instancia posteriormente para generar una clase concreta. El mecanismo actúa en tiempo de compilación pues es éste el único momento en que se pueden crear clases. También se pueden usar constantes en la lista de parámetros de la plantilla. La figura 22 muestra un ejemplo de una plantilla para trabajar con vectores que consta dos parámetros: el tipo de los elementos y la dimensión del vector. Vector<T, dim> se instancia para unos valores concretos de los parámetros produciendo una clase normal, por ejemplo Vector<string, 4>. Cuando se declara la variable v como un vector de 4 cadenas se crea automáticamente en tiempo de compilación la clase Vector<string, 4> a partir del molde o plantilla anterior.

Aunque Ada y Eiffel permiten tipos parametrizados su flexibilidad es mucho menor que la de C++. En Ada, por ejemplo, las plantillas no se instancian automáticamente: es necesario declarar la instanciación expresamente dándole un nombre al tipo o función instanciada. Java ni siquiera admite clases parametrizadas aunque se han hecho propuestas en este sentido [Myers 97] [Bracha 98]. Hay una palabra reservada (generic) para una probable implementación futura.

```
template<class T, int dim>
class Vector {
 private:
     T datos[dim]; //Vector interno de C para almacenar los datos
 public:
     //Acceso por índice una componente de un objeto de Vector<T, dim>
     T& operator[](int pos) { return datos[pos]; }
     ...
};
...
Vector<string, 4> v;
v[1] = "superjuan";
```

Figura 22. Ejemplo de plantilla de C++

Las plantillas de C++ proporcionan propiedades reflexivas al lenguaje ya que actúan como metaclases cuyas instancias son clases. La utilidad de las plantillas de C++ en este sentido va mucho más allá gracias al impulso que se ha dado para su soporte por parte del lenguaje durante el proceso de estandarización [OCS 99] y se analiza en los puntos siguientes.

1.5.1. Plantillas como metafunciones

Las plantillas pueden considerarse como funciones que admiten tipos como parámetros. Estas funciones actúan sobre el código y las clases creando tipos en tiempo de compilación. Por tanto, las plantillas son un ejemplo de metafunciones que trabajan con las propias estructuras del programa.

En el ejemplo de la figura 23 se define una plantilla `Rango<int izda, int dcha>` cuyas clases representan números enteros con la restricción de pertenecer a ese rango.

La plantilla `Rango` se puede considerar como una función que toma como entrada dos enteros `izda` y `dcha` devolviendo el tipo `Rango<izda, dcha>`.

```
template<int izda, int dcha>
class Rango {
 public:
     static const int minimo = izda;
     static const int maximo = dcha;
     static const int cardinal = dcha - izda + 1;
     int valor;
     Rango(int v) {
          if(v < izda || v > dcha) throw "Error"; else valor = v;
     }
};
```

Figura 23. Plantillas como funciones

Asimismo, cardinal se puede considerar como una función que toma como entrada una clase de rango R y devuelve un entero R::cardinal. Por ejemplo, Rango<2, 5>::cardinal vale 4. Esta función cardinal se puede extender para que admita otros tipos como parámetro. Un ejemplo didáctico se puede ver en la figura 24.

```
class ColorSemaforo {
     enum {rojo, ambar, verde} color;
 public:
     static const int cardinal = 3;
};

template<class T>
bool esCodificableEnChar() { return T::cardinal < 255; }

template<class T>
struct EsCodificableEnChar {
     static const bool valor = T::cardinal < 255;
};
```

Figura 24. Extensión de la función cardinal

Una vez declarado el tipo, se puede usar el método cardinal en funciones más generales sin preocuparnos del tipo real por el que se está parametrizando. La función esCodificableEnChar aplicada a un tipo T devuelve true si el tipo T es codificable como un char (es decir, no tiene más de 256 posibles valores). La misma figura 24 muestra una versión dinámica y otra estática de dicha función.

EsCodificableEnChar supone que, para cualquier tipo T al que se aplique, se puede obtener su cardinal mediante T::cardinal. cardinal puede considerarse como un método estático de la clase T. Puede ocurrir que para una clase ya definida tenga sentido hablar de su cardinal pero, al estar ya construida y tal vez compilada, dicha clase no pueda utilizarse como parámetro de EsCodificable. Lo mismo ocurre con los tipos predefinidos, los cuales no se pueden modificar. En estos casos, la solución es análoga a la que se emplea con los métodos y funciones normales. Se puede derivar la clase para que incluya cardinal como valor estático y también se puede crear una función global que tome como entrada el tipo y devuelva su cardinal siguiendo el patrón *Traits* [Myers 95]. La figura 25 muestra cómo se puede implementar cardinal como función sin necesidad de modificar la clase.

```
template<class T>
struct Cardinal {
    static const int val = T::cardinal; //Caso normal
};
template<>
struct Cardinal<bool> {
    static const int val = 2;
};
...
cout << EsCodificableEnChar<bool>::valor;
//Escribe false, valor conocido en compilación
cout << EsCodificableEnChar<Range<2, 90> >::valor;
//Escribe true
```

Figura 25. cardinal como función de tipos

En este ejemplo, la función cardinal aplicada a un tipo cualquiera T se usa así: Cardinal<T>::val. Esta es la expresión que se debe usar en cualquier lugar donde se quiera usar el cardinal de forma genérica. La propia figura muestra como quedaría la función EsCodificableEnChar. Esta última función ya es aplicable tanto a clases que implementen cardinal por medio de T::cardinal como a clases o tipos que lo implementen de otra forma, por ejemplo bool. En el código de esta figura se muestra cómo se puede hacer una llamada a la función EsCodificableEnChar. Nótese que EsCodificableEnChar también es una función que toma como entrada un tipo, y su resultado en vez de ser un int es un bool.

En la figura 26 se define la plantilla Interseccion<T1, T2>. Dentro de ella se define un tipo de datos que va a ser el rango determinado por la intersección de los rangos T1 y T2. De esta forma, Interseccion se puede considerar como una función que toma como entrada dos clases T1 y T2 devolviendo una tercera:

```
(T1, T2) -> Interseccion<T1,T2>::Tipo
     = Rango<max(T1::izda, T2::izda), min(T1::dcha, T2::dcha)>
```

```
#define max(x, y) (x)>(y)? (x) : (y)
#define min(x, y) (x)<(y)? (x) : (y)

template<class T1, class T2>
class Interseccion {
 public:
     typedef Rango<max(T1::izda, T2::izda),
                   min<T1::dcha, T2::dcha> Tipo;
};

typedef Rango<2, 7> R1;
typedef Rango<3, 9> R2;
Interseccion<R1, R2>::Tipo x; //=Rango<3, 7>
```

Figura 26. Plantillas como funciones de tipos

1.5.2. Metaprogramación lógica

En principio puede parecer que las plantillas consideradas como funciones ofrecen una flexibilidad muy reducida. Sin embargo, gracias a la potencia de las plantillas se puede realizar una verdadera metaprogramación con estas funciones. Por ejemplo se pueden definir funciones de modo análogo al que se hace en programación lógica gracias a la *especialización de plantillas*, que permite personalizar la implementación de una plantilla para valores particulares de los parámetros de la misma. En la figura 27 se muestra un ejemplo de una clase Factorial parametrizada por un entero n en la que el campo valor se calcula recursivamente como n multiplicado por el valor que toma n en la clase Factorial<n - 1>. De este modo, Factorial<5>::valor se calcula en tiempo de compilación.

```
template<int n>
class Factorial {
 public: static const int valor= n * Factorial<n-1>::valor;
};
template<>
class Factorial<0> {
 public: static const int valor = 1;
};
...
int x = Factorial<3>::valor; // x = 3 * 2 * 1  calculado al compilar
```

Figura 27. Ejemplo de función implementada por reglas

Para que la recurrencia sea finita es preciso definir Factorial<0> aparte. Esto se puede hacer gracias a la especialización de plantillas. Haciendo abstracción de la sintaxis poco amigable lo que en realidad se está haciendo es implementar el factorial por medio de reglas:

```
Factorial(n) -> n * Factorial(n-1)
Factorial(0) -> 1
```

La verdadera fuerza de la definición de las funciones mediante reglas se muestra al aplicar *especialización parcial* de plantillas. Se puede especializar una función no sólo para un valor fijo sino también para un conjunto de valo-

res. En el ejemplo de la figura 28 se utiliza esta técnica para implementar una metafunción de potenciación. Las reglas que se implementan son:

```
Elevar(m, n) -> Elevar(m, n-1)
Elevar(m, 0) -> 1
```

La segunda regla es la especialización parcial para el conjunto de pares `(m,n)` tales que `n == 0`.

```
template<int m, int n>
struct Elevar {
    static const int valor = m * Elevar<m, n – 1>::valor;
};
template<int m>
struct Elevar<m, 0> {
    static const int valor = 1;
};
const int x = Elevar<2, 5>::valor; // x = 32;
```

Figura 28. Especialización parcial de plantillas

1.5.3. Generación del árbol de una expresión

Las plantillas pueden servir para construir una estructura que represente el árbol de una expresión (definida informalmente como una combinación de llamadas a funciones, operadores, variables y constantes). Un objeto contendrá la información de esta expresión y cuando le sea requerido la evaluará. El hecho de que el objeto contenga toda la información de la expresión asociada permite que se pueda generar código más eficiente. La figura 29 muestra una plantilla `BVector` que representa una operación binaria entre dos vectores; los parámetros de la plantilla son los tipos de esos dos vectores y el de la operación que se ha de realizar con ellos.

Una expresión del tipo `A – (B + C)` siendo `A`, `B` y `C` vectores genera una expresión en tiempo de compilación que puede representarse por el árbol que se ve en la figura 30. Mediante *evaluación perezosa* (*lazy evaluation*) se consiguen eliminar los vectores temporales que se generarían implementando `A –`

(B + C) mediante una implementación tradicional de las operaciones – y + en vectores. El uso de las funciones de expansión (que expanden su código en la llamada) permite que el código generado sea tan eficiente como el obtenido si se hubiera hecho a mano. Estas técnicas se han aplicado en varias librerías numéricas como Blitz++ [Veldhuizen 98].

```
template<class F, class V, class W>
class BVector {
    const F& f; const V& v; const W& w;
 public:
    BVector (const F& f_, const V& v_, const W& w_) :
        f(f_), v(v_), w(w_) {}
    float operator[](int i) { return f(v[i], w[i]); }
};
template<class V, class W>
BVector <Plus, V, W> operator+(const V& v, const W& w) {
    return BVector<Plus, V, W>(plus, v, w);
}
```

Figura 29. Clase para almacenar expresiones binarias de vectores

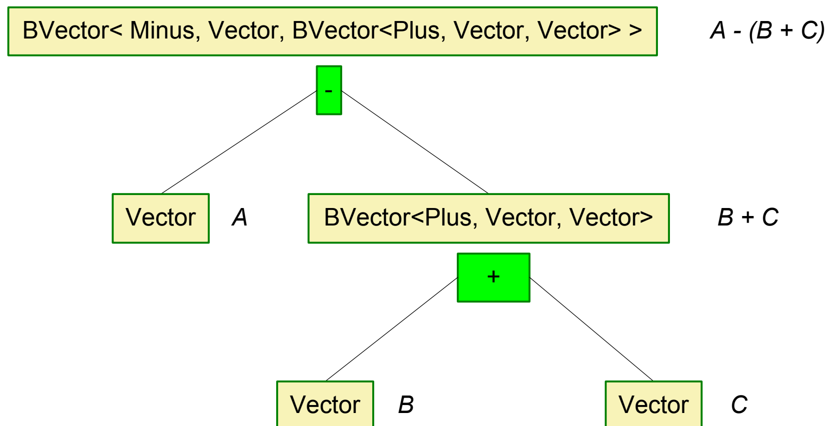


Figura 30. Plantillas representando expresiones

1.5.4. Programación funcional

La técnica de la evaluación perezosa permite crear árboles de expresiones de funciones a partir de la función identidad x. Así 2 * x + 3 es la función que multiplica por dos su parámetro y le suma tres unidades. Como se ve, se les está dando a las funciones propiedades de objetos de primera clase ya que se pueden hacer operaciones con ellas.

La librería estándar de C++ incluye muchos algoritmos en los que uno de los parámetros es una función (encapsulada en un objeto [Rasala 97]). Gracias a la posibilidad de realizar operaciones con funciones se puede incluir la expresión de la función como parámetro de la llamada sin necesidad de definirla trabajosamente en un sitio aparte (figura 31).

```
//Búsqueda "tradicional" en un vector del primer elemento sea mayor que 2
class X {
	static bool mayor2(int a) {return a > 2;}
	void f() {
		...
		vector<int> v;
		...
		i = find(v.begin(), v.end(), ptr_fun(mayor2));
	}
};

//Búsqueda usando expresiones de funciones, IdInt es la función identidad
class X {
	void f() {
		...
		vector<int> v;
		...
		i = find(v.begin(), v.end(), idInt > 2);
	}
};
```

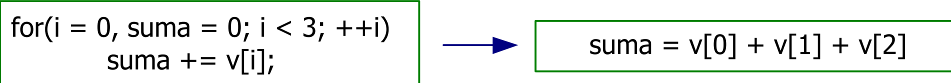
Figura 31. Expresiones de funciones

Esta característica se encuentra en Lisp con las expresiones lambda. En el caso de C++ la ventaja se hace más patente todavía por el hecho de que no se permite declarar funciones dentro de funciones. Si no se pudieran usar expresiones de funciones habría que declarar la función en un lugar "lejano" al lugar donde se usa (posiblemente como función estática con ámbito de clase) lo que produciría interferencias con el resto de elementos de la clase los cuales no necesitan ver dicha función.

[Veldhuizen 95b] presenta una arquitectura de expresiones funcionales en una variable. [Musser 98] utiliza la misma técnica para implementar funciones en dos variables.

1.5.5. Control de generación de código

La metaprogramación con plantillas [Veldhuizen 95] permite controlar la forma en que se genera código. Con ello los programas pasan a ser elementos con los que se puede operar. La figura 32 muestra como las plantillas permiten "desenrollar" un bucle de manera que el cálculo de la suma de un vector mediante un bucle se convierte en una expresión donde aparecen todas las sumas explícitamente




```
class Vector {
    float v[];

    template<int i> float suma() {
        return v[i] + suma<i – 1>();
    }

    template<> float suma<0>() {
        return v[0];
    }
};
```

Figura 32. Generación de código con plantillas

Esta característica de las plantillas de permitir personalizar el código generado muestra que el código fuente es tratado por dos programas, el metaprograma de las plantillas que se ejecuta en tiempo de compilación y el programa normal de C++ que se ejecuta en tiempo de ejecución (!), permitiendo introducir a C++ en el campo de los evaluadores parciales [Jones 96], que son programas que ejecutan computaciones estáticas en tiempo de compilación y dinámicas en tiempo de ejecución (figura 33).

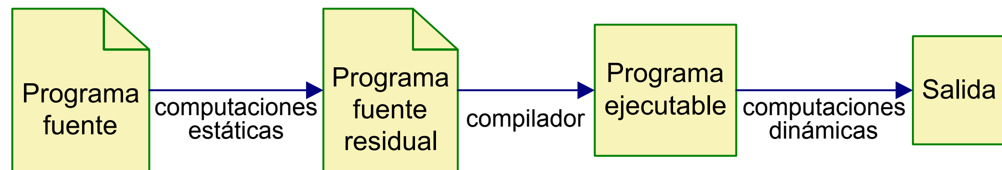


Figura 33. Evaluador parcial

```
template<bool condicion, class A, class B>
struct IF {};

template<class A, class B>
struct IF<true, A, B> { typedef A VAL; };

template<class A, class B>
struct IF<false, A, B> { typedef B VAL; };

struct Si { static void escribe() { cout << "Sí"; } };
struct No { static void escribe() { cout << "No"; } };

//Se llama al método Si::escribe() si 5! > 32
//y al método No::escribe() en caso contrario
IF< (Factorial<5>::valor > 32), Si, No >::VAL::escribe();
```

Figura 34. Composición condicional en el metalenguaje

El lenguaje objeto de compilaciones estáticas en C++ es bastante limitado ya que sólo puede trabajar con constantes de valor numérico entero y con tipos. No obstante, es posible implementar las estructuras básicas de los lenguajes estructurados como son la composición condicional (figura 34), la condicional múltiple y la iterativa [Czarnecki 98]. La figura 34 es un ejemplo de cómo la

metaprogramación con plantillas puede permitir que se genere un código u otro dependiendo de una condición.

1.5.6. Polimorfismo en tiempo de compilación

Un concepto básico en los lenguajes orientados a objetos es el polimorfismo o habilidad de seleccionar en tiempo de ejecución el método que se ha disparar dependiendo de la clase a la que pertenezca el objeto destinatario del mensaje. El modelo de objetos se implementa de tal modo que en tiempo de ejecución se selecciona un método entre varios con el mismo nombre y signatura. El metalenguaje de las plantillas también posee una característica análoga permitiendo seleccionar, esta vez en tiempo de compilación, el método que se ha de disparar dependiendo del tipo estático del objeto. (Si a se declara como una referencia a la clase A, el tipo estático de a es siempre A, el tipo dinámico puede ser A o alguna clase derivada de A).

Una ventaja del polimorfismo en tiempo de compilación es que hay mucha más flexibilidad en cuanto al conjunto de tipos involucrados en la selección del método. Pueden intervenir tipos de datos predefinidos y no sólo clases y además no hay restricciones en cuanto a la herencia. En la figura 35 se define una función abs (valor absoluto) que es polimórfica respecto de cualquier tipo de datos para el que tenga sentido realizar comparaciones, copia y cambio de signo. Además, se pueden añadir tipos adicionales que no cumplan los requisitos de la plantilla (el equivalente a derivar nuevas clases). En el ejemplo se añade una versión de la función abs para números complejos, en los cuales no hay una relación de orden y por tanto no está definido el operador menor.

```
template<typename T>
T abs(const T& t)
    if (x < 0) return –x; else return x;
}
float abs(const Complex& c) {
    //Devolver el módulo del complejo c
}
```

Figura 35. Polimorfismo estático por medio de plantillas

1.5.7. Limitaciones de las plantillas para metaprogramación

Las plantillas no se desarrollaron con la idea de permitir metaprogramación en el sentido que se acaba de ver, por ello presentan unas deficiencias estructurales que hay que tener en cuenta:

- Sintaxis poco elegante y pesada. La figura 34 constituye un ejemplo gráfico de esta afirmación.
- El lenguaje estático es muy limitado: no permite trabajar con números en coma flotante, no se puede metaprogramar de manera que se muestren en la consola los mensajes durante la compilación, etc.
- La cantidad de código generado aumenta considerablemente si se hace un uso intensivo de plantillas. (Este fenómeno se conoce con el nombre de *code bloat*).
- El nivel base y el meta no interactúan adecuadamente. Por ejemplo, las plantillas no son clases ni objetos. La implementación y sintaxis es completamente diferente y no hay integración ni posibilidades adecuadas de comunicación.
- La especialización parcial sólo cubre unos pocos casos de subconjuntos de reglas. Por ejemplo, la implementación de una especialización para enteros menores que 100 no es directa.
- Los compiladores actuales generan mensajes extremadamente largos y difíciles de interpretar cuando se trabaja con expresiones de plantillas. Una propuesta de simplificación de mensajes de error al compilar plantillas puede verse en [Alexandrescu 99].
- Los compiladores actuales aún no implementan todas las características indicadas por el estándar debido a su complejidad. Además, no hay una manera establecida de compilar las plantillas, la mayoría de las implementaciones incluye el código fuente de las plantillas en ficheros cabecera con lo cual los tiempos de compilación pueden dispararse.

1.6. Conclusiones

En este capítulo se han visto las ventajas que se obtienen gracias a la posibilidad de acceder y gestionar la metainformación de los programas y cómo la adición de propiedades reflexivas a los lenguajes de programación facilita esta tarea.

Se ha analizado el concepto de reflexión en los lenguajes de programación como herramienta para realizar un modelo de abstracción basado en separar la lógica del dominio de la aplicación de la del dominio del comportamiento del sistema que la gobierna. Con esto se consigue establecer un alto grado de independencia entre módulos, facilitar las tareas de encapsulado y fomentar la reutilización del código fuente escrito. Se ha mostrado que los lenguajes orientados a objetos son muy apropiados para incorporar esta funcionalidad de reflexión. Se han analizado las capacidades reflexivas de varios lenguajes de programación viéndose que algunos de ellos, como Smalltalk y Lisp, son muy potentes en este aspecto. Por otro lado, se ha visto que el modelo de reflexión de Java es un ejemplo de una estructura sencilla pero suficientemente potente como para tener muchas posibilidades de aplicación.

Una parte del trabajo realizado ha consistido en analizar más extensamente el modelo de C++. Sus capacidades reflexivas resultan más pobres que las de Java. Para dotar al lenguaje de capacidades reflexivas útiles se hace necesario crear una infraestructura específica para tal fin. Se han estudiado diferentes aproximaciones en este sentido. Algunas de ellas no se integran bien en la filosofía estáticamente tipificada de C++ y en algunos casos la complejidad de la infraestructura creada obliga a plantearse la cuestión de su viabilidad industrial. Muchas soluciones propuestas han quedado parcialmente obsoletas debido a la gran evolución del lenguaje en los años 90 como consecuencia de su proceso de estandarización. Por tanto, es necesario realizar nuevas aproximaciones que proporcionen soluciones más realistas que sean lo suficientemente potentes para ser útiles, pero lo suficientemente sencillas como para no sobrecargar peligrosamente el sistema de desarrollo. Una de las más interesantes aproximaciones la constituyen las plantillas. Aunque originariamente se diseñaron para poder definir contenedores parametrizados por el tipo, pronto se vio que sus posibilidades son mucho mayores y permiten realizar programación en tiempo de compilación y aumentar la eficiencia gracias a que permiten controlar la generación de código. El descubrimiento, casi por accidente, de estas características ha tenido la negativa consecuencia de que la sintaxis es poco

elegante y las soluciones mucho más limitadas que si se hubiera hecho el diseño teniendo en cuenta estos objetivos. Aún con estas limitaciones, se ha demostrado que las plantillas son un instrumento de gran utilidad para generar código eficiente y para representar en el lenguaje nuevos paradigmas de programación.

Capítulo 2

Infraestructura para dar soporte a metainformación en C++

En este capítulo se presenta una infraestructura para dotar a C++ de capacidades reflexivas que permitan desarrollar aplicaciones utilizando un esquema de representación a partir de metainformación. Ésta podrá ser utilizada para realizar la construcción tanto del núcleo de una aplicación, como de los componentes especializados de la misma (interfaz gráfica, persistencia en bases de datos, comunicaciones).

El objetivo básico consiste en proporcionar al lenguaje capacidades de auto-inspección. Para ello, se desarrollan un conjunto de estructuras que van a permitir ubicar la metainformación de los programas en unas localizaciones "fijas". La metainformación contendrá datos sobre las clases, los atributos y los métodos. Para aprovecharse de las ventajas de la orientación a objetos, las estructuras de metainformación deberán poderse especializar de acuerdo a una serie de mecanismos de herencia que permitan personalizar la metainformación que se asocie a las diferentes estructuras.

No se pretende dotar al lenguaje de capacidades dinámicas como la modificación del comportamiento de las diferentes estructuras sintácticas o la creación de clases en tiempo de ejecución. Para ello sería necesario desarrollar o reutilizar herramientas más complejas y de más difícil mantenimiento. La aproximación que se presenta en este capítulo utiliza los propios recursos del lenguaje como las plantillas de clases. Sí se persigue que la solución sea abierta y que se pueda personalizar para poder añadir cómodamente diferentes tipos de metainformación a las clases y sus atributos. Se incide en respetar conceptos claves y básicos tanto en orientación a objetos (reutilización, encapsulado, evitación de redundancias), como en el propio C++ (eficiencia, detección de errores en tiempo de compilación).

En el primer apartado se formulan las capacidades que se desean incorporar al lenguaje con arreglo a los criterios que se acaban de citar. Estas capacidades deben permitir trabajar con metainformación relacionada con las estructuras fundamentales del lenguaje como atributos, métodos y clases. A la vez que se enumeran estos objetivos se estudian las aproximaciones directas que se suelen emplear para conseguirlos, analizándose los problemas que acarrean.

A continuación se introduce un patrón de C++ llamado *atributo enriquecido* y se presenta la arquitectura básica de la aproximación adoptada, que se apoya en dicho patrón y que se materializa en una librería llamada *librería de facets*. Se estudian los componentes de esta librería, su filosofía y las facilidades que ofrece para añadir metainformación a los atributos, los métodos y las clases. Otras clases permiten explorar la estructura global del sistema de modo que se pueda inspeccionar la lista de clases o de objetos creados en el programa. Se muestra cómo una serie de macros incluidas en la librería permite simplificar la sintaxis para hacer uso de ella. También se demuestra su naturaleza extensible que permite la adición de componentes para ampliar sus posibilidades de aplicación.

Un último apartado se dedica a analizar las posibilidades de herencia de la metainformación estudiándose toda la problemática relacionada con los distintos tipos de herencia admitidos por C++ y viendo cómo la solución propuesta se puede adaptar para permitir todos estos tipos de herencia.

2.1. Objetivos del esquema de metainformación

Los principios básicos que se persiguen consisten en dotar de información adicional a las estructuras del lenguaje para realizar tareas de introspección. Esta información adicional deberá poder especificarse por el usuario del sistema para adaptarlo a sus necesidades concretas pues son éstas muy variadas y una solución cerrada no podría dar satisfacción a un amplio conjunto de problemas. A continuación se enumeran los diferentes tipos de metainformación que deberá proporcionar la solución y cuál es el sistema habitual en C++ de proporcionar dicha metainformación. Posteriormente se indican qué problemas surgen con estas soluciones habituales.

2.1.1. Información de atributos

En el código compilado de C++ se pierde cierta información asociada a los atributos como puede ser por ejemplo su nombre o la identificación de la clase en la que están declarados. Resulta interesante poder recuperar dicha información y añadir información nueva para poder realizar tareas como las siguientes:

- *Acceso encapsulado a los atributos*. Los usuarios de una clase no deben tener acceso directo a un atributo de un objeto. Impidiendo el acceso exterior a su implementación se gana independencia entre las distintas partes del programa. Esto permite modificar dicha implementación sin que esto afecte a los usuarios de la clase.

  La solución habitual en C++ consiste en declarar privados los atributos y definir métodos públicos que accedan a ellos. La figura 36 muestra un ejemplo de una clase Tiempo que almacena la hora y el minuto de un día. El acceso a estos dos atributos se encapsula mediante métodos set (para cambiar su valor) y get (para leerlo):

```
class Tiempo {
 private:
    int _hora;
    int _minuto;
 public:
    int getHora() const { return _hora; }
    void setHora(int nuevaHora) {
          _hora = nuevaHora;
    }
    ... //Ídem para acceder a _minuto
};
```

Figura 36. Acceso encapsulado a un atributo

  El acceso encapsulado a los atributos permite realizar operaciones adicionales como comprobar la corrección de un valor, calcularlo dinámicamente sin necesidad de que haya un atributo real, realizar cómputos estadísticos, etc. La figura 37 muestra una modificación de la implementación de la clase Tiempo en la que se almacena únicamente un atributo que representa los

minutos totales que han pasado desde las 0 horas (para ahorrar espacio). El usuario de la clase no tiene que modificar sus programas ya que los métodos de acceso (getHora, setHora) se han adaptado a la nueva implementación proporcionando al usuario la ilusión de la existencia de atributos que almacenan la hora y el minuto por separado.

```
class Tiempo {
 private:
    int _minutosTotales;
 public:
    int getHora() const { return _minutosTotales / 60; }
    void setHora(int nuevaHora) {
        _minutosTotales = 60 * nuevaHora + getMinuto();
    }
    ... //ídem con getMinuto y setMinuto
};
```

Figura 37. Acceso encapsulado a un atributo no real

- *Valores por defecto en los atributos*. En ocasiones interesa que el atributo de una instancia tome un valor por defecto en caso de no darle ninguno explícitamente. Esto se puede hacer en C++ utilizando un valor opcional como parámetro del constructor de objetos de la clase (figura 38).

```
class Persona {
    string _nombre;
 public:
    Persona(string nombre = "Anónimo"): _nombre(nombre) {}
 ...
};
```

Figura 38. Indicación de valores por defecto en el constructor

- *Restricción del rango de valores correctos de los atributos*. No todos los posibles valores de la clase a la que pertenece un atributo son válidos para

un atributo concreto. Así, por ejemplo, en el caso del nombre de una persona podemos considerar inválidos los valores de más de 20 caracteres, simplemente porque se van a guardar los datos de las personas en una tabla de base de datos con esa longitud máxima para la columna del nombre. Se necesita, por tanto, un mecanismo que permita detectar una asignación de un valor incorrecto a un atributo.

Tradicionalmente, en C++ se realiza este test en cualquier lugar donde se pueda cambiar el valor del atributo (figura 39).

```
void Persona::cambiaNombre(string nuevoNombre) {
    assert(nuevoNombre.size() <= 20);
    //Si no se cumple la condición del assert se produce un error
    _nombre = nuevoNombre;
}
```

Figura 39. Comprobación de la validez del valor de un atributo

Para que la restricción sea variable será necesario declarar una variable estática a la clase (figura 40).

```
class Persona {...
    static int longitudMaximaNombre;
    void cambiaNombre(string nuevoNombre) {
        assert(nuevoNombre.size() <= longitudMaximaNombre);
        ...
    }
};
```

Figura 40. Restricción variable al acceder al valor de un atributo

- *Adición de otros tipos de información a los atributos*. Resulta interesante dotar de cierta información a los atributos de modo que a través de ellos se pueda acceder por ejemplo a una cadena con el nombre del atributo. Esto facilitaría la realización de tareas de entrada/salida (mostrando ese nombre al usuario le identificamos el atributo) o el trabajo con tablas de bases de

datos donde tiene persistencia el objeto. Así, por ejemplo, dado el atributo se podría identificar el campo que le corresponde en la tabla. En el caso de las interfaces gráficas de usuario, el atributo podría guardar el aspecto o apariencia con que se muestra, etc. De esta forma el propio atributo "sabe" cómo almacenarse de forma persistente o cómo representarse gráficamente.

Debido a la gran variedad de información adicional que puede interesar guardar en un atributo cualquier solución debe permitir al usuario establecer cuál es esa información adicional.

La solución en C++ sería similar al caso anterior, declarando variables estáticas que guarden estos valores.

Las soluciones anteriores plantean una serie de problemas que hacen inviable su aplicación en proyectos serios de software:

- *Redundancia de código*. Una de las ventajas de la orientación a objetos consiste en poder sacar "factor común" a dos clases que tienen cierto parecido de modo que esa parte común figure sólo una vez en el código. Esto se traduce en grandes ventajas a la hora de mantener el programa. Así, por ejemplo, si se quiere hacer un cambio ya no hay que recorrer todo el código buscando los lugares donde llevar a cabo el mismo. Basta ir al lugar concreto y hacer el cambio allí. Esta filosofía se está violando con las soluciones indicadas. Por ejemplo, si se quiere que el método de acceso a un atributo no se llame `get` sino `read` dicho cambio habrá que hacerlo en muchos lugares del código.

- *La información propia de un atributo se encuentra esparcida por la clase*. Para cada atributo hay una serie de variables y métodos que están definidos en el ámbito de la clase y que, en realidad, son propios de cada uno de ellos (los métodos para leer y escribir, las variables estáticas que guardan el nombre del atributo y los valores de restricción, el valor por defecto en el constructor). La falta de información de los atributos obliga a definir estos elementos en ámbito de clase en vez de en ámbito de atributo, que es el que les corresponde de modo natural. Si, por ejemplo, se quiere borrar un atributo hay que inspeccionar el resto de la clase para localizar todos los elementos asociados a dicho atributo. El problema sería mucho menor si esta información estuviese localizada en el propio atributo (bastaría borrar su declaración).

- *Ausencia de soporte directo a elementos del diseño*. Al no existir en C++ el concepto de atributo enriquecido se abre un hueco entre el diseño y la implementación. En el código no quedan reflejados de modo directo los elementos del diseño. Debido a eso, una simple adición de un atributo obliga a escribir a mano todos los objetos relacionados con dicho atributo, forzando al programador a preocuparse de detalles de bajo nivel que nada tienen que ver con el diseño de la aplicación.

2.1.2. Acceso a información de las clases

En el capítulo anterior se vieron las ventajas que proporciona el hecho de tratar a las clases como elementos de primer orden del lenguaje, tanto para extraer y modificar la información de la estructura de dicha clase y sus objetos como para establecer el comportamiento de estos objetos.

La adición de propiedades dinámicas a las clases choca fuertemente con la naturaleza estática del lenguaje; por eso, el objetivo en este aspecto se centra en conseguir un sistema que permita obtener fácilmente cierta información relacionada con las clases, información que se encuentra implícitamente en el código y que por tanto es inaccesible.

- La lista de atributos de la clase es útil para automatizar operaciones de entrada-salida, tales como guardar los datos de un objeto en un fichero o en una tabla de base de datos. Esto se puede hacer utilizando las técnicas vistas en 1.4.1.
- El nombre de la clase y la lista de clases de las que hereda pueden permitir la creación de instancias de la clase a partir del nombre de la misma y ayudar en la construcción de la lista de atributos mediante navegación por el árbol de clases padre.
- La lista de métodos de una clase que incluya para cada método cierta información como el nombre y sus parámetros permitiría llamar a un método a partir de su representación textual, es decir, se podría ejecutar código de forma interpretada y se facilitaría la comunicación del programa con otros elementos que integren el sistema al proporcionar una interfaz de acceso.

Se ha visto en el capítulo anterior que C++ proporciona un soporte primitivo para metainformación en el ámbito de las clases por medio del RTTI. Para cada

clase del programa que tenga un método virtual se genera un objeto de tipo type_info con información sobre la clase en cuestión. type_info es muy pobre por cuanto no proporciona información básica sobre el tipo como puede ser su lista de atributos o las clases de las que hereda. Para poder añadir información a un tipo [Stroustrup 97] indica un método que consiste básicamente en asociar a cada objeto de type_info (es decir a cada clase) un objeto de una clase ExtendedTypeInfo que tenga la información adicional (figura 41).

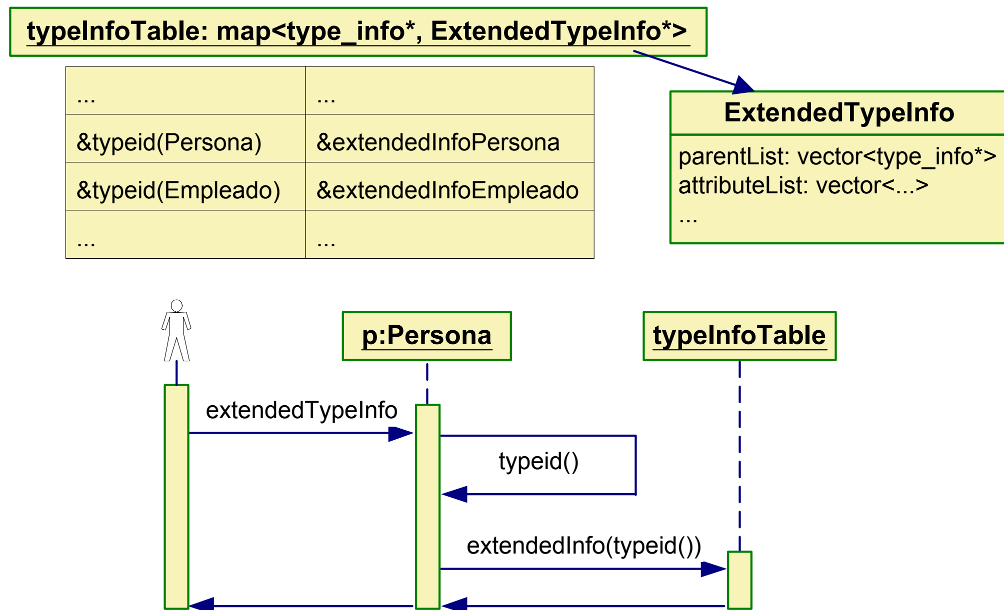


Figura 41. Acceso a información extendida de tipos

Esta asociación se puede implementar mediante una tabla de pares type_info - ExtendedTypeInfo. Dado un objeto, se puede acceder al objeto que representa a su clase accediendo en primer lugar a su type_info (con el operador typeid(objeto)) para a continuación obtener su ExtendedTypeInfo consultando la tabla. Esta tabla podría implementarse utilizando estructuras que permitan un acceso rápido a un ExtendedTypeInfo dado su type_info, por ejemplo con el contenedor map de la librería estándar o mediante tablas de dispersión (*hash*).

En esta clase con información ampliada se pueden indicar las clases padre mediante una lista con los type_infos correspondientes a dichas clases padre.

Más complicado resulta incluir otra información de la clase como los atributos y los métodos ya que la única estructura de C++ que los representa son los punteros a miembro y estos sólo permiten acceder al valor del atributo o llamar al método para un objeto concreto. Además, el acceso a la información extendida se realiza en tiempo no constante al tener que consultar la tabla de equivalencias.

2.1.3. Acceso a información global del sistema

Una aproximación a la representación del estado global de la aplicación en un momento dado puede conseguirse a partir de una lista de las clases junto con los objetos existentes de cada clase. Con esta información puede guardarse el estado completo del sistema para restablecerlo posteriormente, bien para realizar depuraciones y volver a un estado anterior del programa, bien para almacenar el estado de forma persistente para recuperarlo después. Al tener información completa de todos los objetos y clases del sistema se pueden realizar tareas de depuración usando el propio lenguaje, implementar recolectores de basura etc. El uso de información extendida de RTTI puede ayudar en este sentido. La tabla de la figura 41 serviría para obtener la lista completa de clases. Un campo de la información extendida de cada clase podría ser una lista con los objetos de dicha clase.

2.1.4. La librería de facets

Para añadir metainformación a C++ se ha desarrollado una librería llamada librería de facets. Está formada por un conjunto de elementos que constituyen un *framework* reflexivo orientado a objetos, es decir, un conjunto de clases y componentes que proporcionan una solución genérica a un amplio espectro de requerimientos de aplicación (en este caso, aplicaciones que requieren hacer uso de metainformación). Los componentes de la librería pueden personalizarse y ampliarse por los clientes para satisfacer los requerimientos de aplicaciones específicas [Foote 92]. En posteriores apartados se verá como se consiguen estos objetivos.

La librería de facets hace uso de un conjunto de clases de diversa utilidad. En la figura 42 se muestra la jerarquía de dicho conjunto de clases básicas de soporte para la librería de facets.

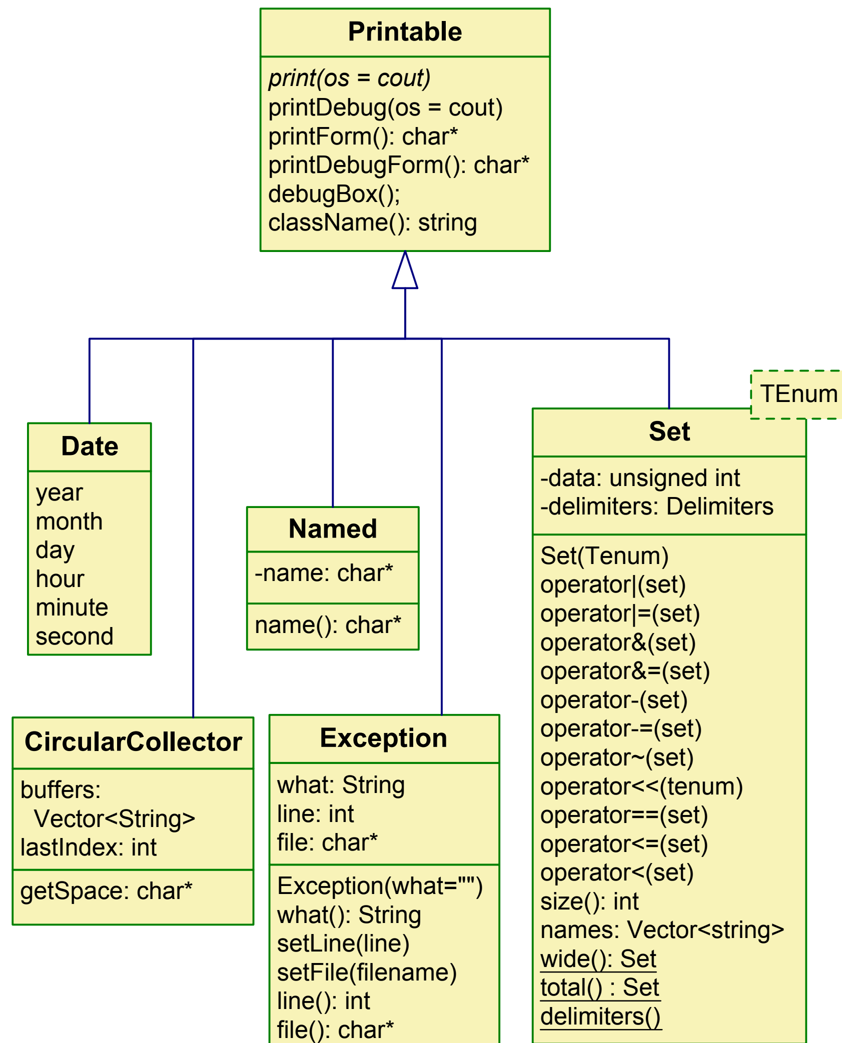


Figura 42. Jerarquía de las clases básicas de soporte

La clase Printable es la clase base de cualquier clase cuyos elementos sean imprimibles en un *stream* [Langer 00]. Una clase que derive de Printable deberá implementar el método virtual print(stream). A partir de print la clase Printable implementa métodos útiles de depurado como printForm(). Éste método captura los caracteres que imprime en el *stream* el método print proporcionando una representación del dato en forma de string predefinido de C que puede utilizarse con los depuradores de C++ para mostrar información compacta del valor del dato.

CircularCollector es un recolector de basura elemental que sirve para recuperar la memoria utilizada para almacenar el char* que devuelve printForm. Se implementa mediante una lista circular de 40 char*. Cuando la lista se llena se libera la memoria de la cadena más antigua.

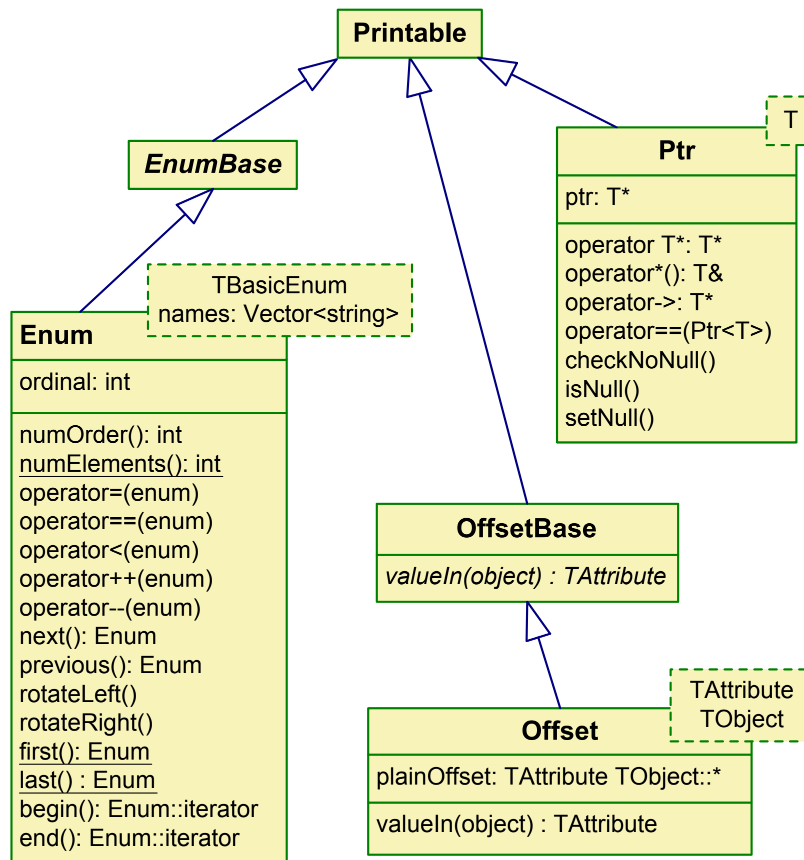


Figura 43. Encapsulado de tipos básicos

Otras clases básicas son Named, que es el padre de todas aquellas cuyos elementos tengan un nombre, Date para trabajar con fechas y Exception que es la clase base de todas las excepciones de la librería de facets.

Algunos tipos básicos se han encapsulado dentro de clases (figura 43). La clase genérica Enum< TBasicEnum, Vector<string> > sirve para encapsular tipos enumerados. Tiene varias ventajas con respecto a estos: añade seguridad y gestión de errores al tipo primitivo, incluye operaciones adicionales úti-

les como la obtención de la cadena que representa el valor de un enumerado o del siguiente elemento y permite crear una clase ancestro común (EnumBase).

La misma filosofía se ha seguido para encapsular punteros a miembro: la plantilla Offset<TAttribute, TObject> encapsula un puntero a miembro de tipo TAttribute dentro de la clase TObject. Así por ejemplo si nombre es un atributo de Persona, la clase que lo encapsula es Offset<string, Persona>. Como en el caso de los enumerados se ha definido una clase ancestro común OffsetAttribute para representar desplazamientos genéricos.

Por último Ptr<T> encapsula un puntero normal de tipo T. Tiene operadores automáticos de conversión, de impresión y comprueba siempre su validez si se quiere desreferenciar el puntero.

2.2. Metainformación de atributos.

En este apartado se va a ver una forma de asociar metainformación a los atributos de una clase mediante es uso de un nuevo patrón del lenguaje llamado *atributo enriquecido*.

C++ proporciona para sus atributos u objetos miembro dos tipos de ámbito (figura 44):

- Ámbito de objeto. Cada objeto de la clase contiene al objeto miembro. Por defecto, los campos de C++ tienen ámbito de objeto.
- Ámbito de clase. El objeto miembro se asocia a la clase de modo que sólo existe una copia del mismo para cada clase. La palabra reservada static se utiliza para declarar un objeto miembro con ámbito de clase.

```
class Persona {
	string nombre;
	int edad; //Ámbito de objeto. Cada persona tiene una edad
	static string nombrePorDefecto;
	/* Ámbito de clase. Sólo hay un nombre por defecto
	compartido por todas las personas */
	...
};
```

Figura 44. Ámbitos de los objetos miembro en C++

2.2.1. Problemas para incluir información en los atributos

Para incluir información adicional a los atributos de una clase no resulta apropiado ninguno de los dos niveles anteriores. Lo vamos a ver con unos sencillos ejemplos.

```
class AttributeString {
	string value; //Valor del atributo
	int maxLen; //Longitud máxima
};
class Persona {
	AttributeString apellido;
	...
};
```

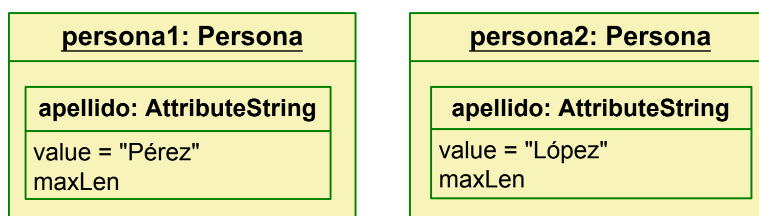


Figura 45. persona1.apellido.maxLen ≠ persona2.apellido.maxLen

En la figura 45 se muestra un ejemplo de una clase AttributeString que sirve para representar atributos de cadenas con la información adicional de la longitud máxima del atributo. Con esta aproximación surge el problema de que todas las instancias de Persona guardarán en su campo apellido la longitud máxima de la cadena. Esta redundancia provoca un aumento innecesario de la memoria que ocupa cada objeto de la clase y abre la puerta a la aparición de incosistencias.

El segundo tipo de ámbito tampoco resuelve el problema, como se puede ver en la figura 46.

```
class AttributeString {
	string value;
	static int maxLen; //Longitud máxima
	//compartida por todos los objetos de AttributeString
};
class Persona {
	AttributeString apellido;
	AttributeString nombre;
};
```

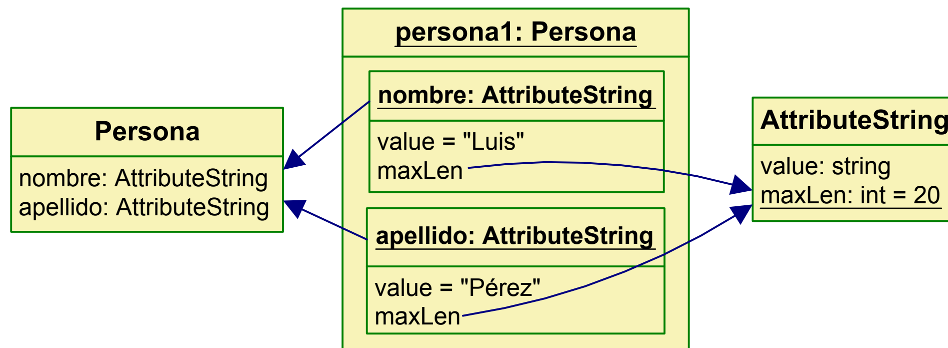


Figura 46: Los atributos nombre y apellido comparten la longitud máxima

Al ser maxLen común a todos los objetos de clase AttributeString resultará que nombre y apellido compartirán el valor de maxLen, sin embargo debería permitirse a diferentes atributos tener diferente longitud máxima.

2.2.2. Atributos enriquecidos

Se necesita un tercer tipo de ámbito para los objetos miembros, de manera que se pueda asociar información a cada atributo de una clase en vez de a cada objeto de la misma. A los objetos con este nuevo tipo de ámbito se les llamará *facets* utilizando la misma terminología que en los lenguajes basados en frames en cuya estructuración nos hemos inspirado.

Como C++ no proporciona soporte directo a este tipo de objetos es preciso construirlos mediante el uso de otras características del lenguaje. Las plantillas de C++ resultan útiles para este propósito. La idea consiste en parametrizar la clase del atributo por una instancia que estará ligada a dicho atributo. Cada atributo tendrá por tanto una instancia asociada que será el miembro estático en ámbito de atributo. En las figuras 47 y 48 se muestra un ejemplo en el que se crea dicha plantilla y dos de tales instancias para los atributos `nombre` y `apellido`. A estos atributos que contienen información estática adicional por medio de un parámetro de una plantilla se les llamará *atributos enriquecidos*.

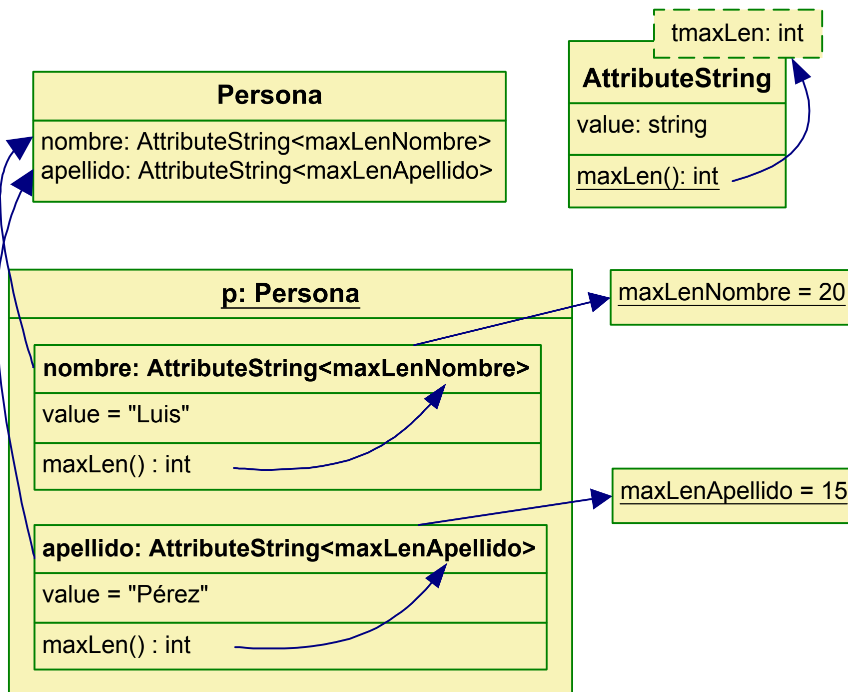


Figura 47. `nombre` y `apellido` tienen su propia parte estática.

```
template<int& tmaxLen>
class AttributeString {
 public:
     string value;
     inline static int& maxLen() { return tmaxLen; }
};
class Persona {
     ...
     static int maxLenNombre;
     AttributeString<maxLenNombre> nombre;
     static int maxLenApellido;
     AttributeString<maxLenApellido> apellido;
};
//Valores iniciales de los miembros en ámbito de atributo:
int Persona::maxLenNombre = 20;
int Persona::maxLenApellido = 15;
...
Persona p;
...
p.nombre.value = "Luis"; //Valor diferente para cada objeto
p.nombre.maxLen() = 25; //Valor común a todos los objetos
p.apellido.maxLen() = 35; //pero diferente para cada atributo
```

Figura 48. Atributos enriquecidos con información estática.

```
class Persona : public Frame {
     public: int leerEdad() const;
     slot:
          default_value("Anónimo") String nombre;
          if_needed(leerEdad) range(0-99) int edad;
};
```

Figura 49. Implementación de facets con OpenC++

[Cavarroc 98] hace uso de OpenC++ para implementar facets de atributos de forma análoga a los lenguajes basados en frames (figura 49).

2.2.3. Clases raíz de facets y atributos enriquecidos

Las clases raíz de la librería se pueden ver en la figura 50.

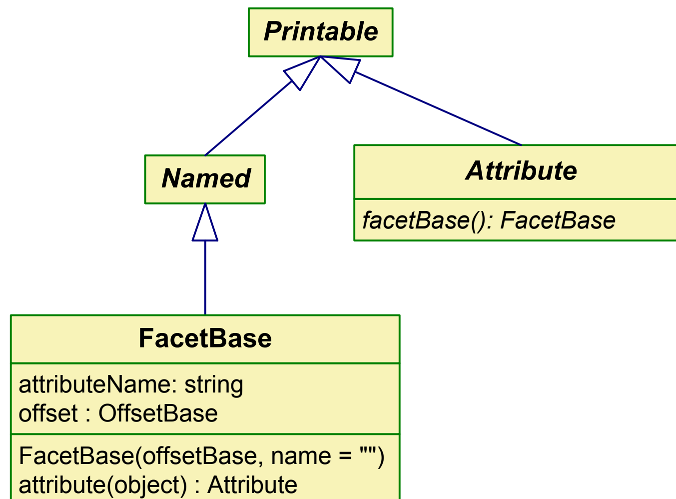


Figura 50. Raíz de la jerarquía de facets

Todas las clases de atributos heredan de una clase padre común Attribute. Asimismo las clases de facets heredan de FacetBase. Gracias a que todos los facets derivan de una clase común se pueden crear listas de facets heterogéneas implementadas como una lista de punteros a FacetBase. Por ejemplo, se puede tener una lista con los facets definidos para una clase.

La estructura es extensible pues permite añadir clases tanto en la jerarquía de atributos como en la jerarquía de facets. La estructura es asimismo flexible gracias a que la jerarquía de atributos y facets no se realiza de forma paralela (a cada clase de atributo le correspondería una clase de facet) sino ortogonal: una clase de atributo puede parametrizarse por un facet cualquiera. Cuando se añade una clase de facet a la jerarquía de facets este facet puede utilizarse en cualquier clase de atributo a no ser que se establezca una restricción específica.

En FacetBase se ha incluido la información mínima útil que van a contener todos los facets. Esta información consiste en el nombre del atributo que describe y la posición del atributo dentro de la clase, es decir, la distancia entre el principio de un objeto de la clase y el lugar donde se guarda el valor del atributo. Gracias a ello, dado un objeto de la clase y un facet se puede obtener el valor para ese objeto del atributo que describe el facet (figura 51). La implementación de este objeto se hace mediante un puntero a miembro de C++. En C++ dos punteros a miembro que no sean de la misma clase o que se diferencien en el tipo del atributo al que apuntan pertenecen a clases diferentes. Para guardar en FacetBase un puntero a miembro que pueda pertenecer a cualquier tipo de atributo de cualquier clase del modelo es necesario utilizar una clase ancestro común a todos los punteros a miembro. OffsetBase (figura 43) sirve para este propósito.

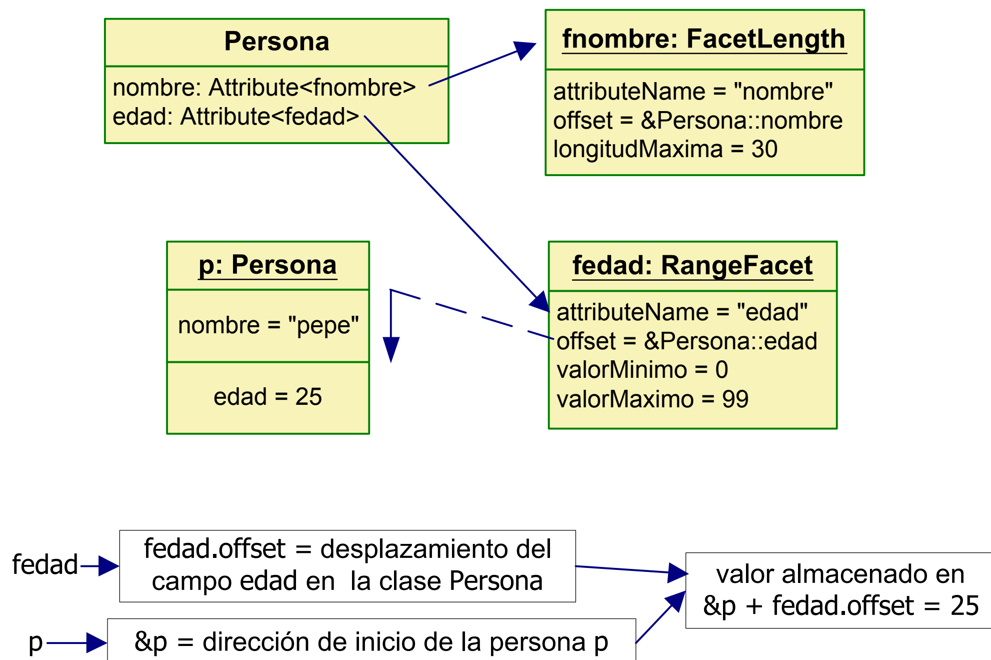


Figura 51. Acceso al atributo de un objeto a partir del facet

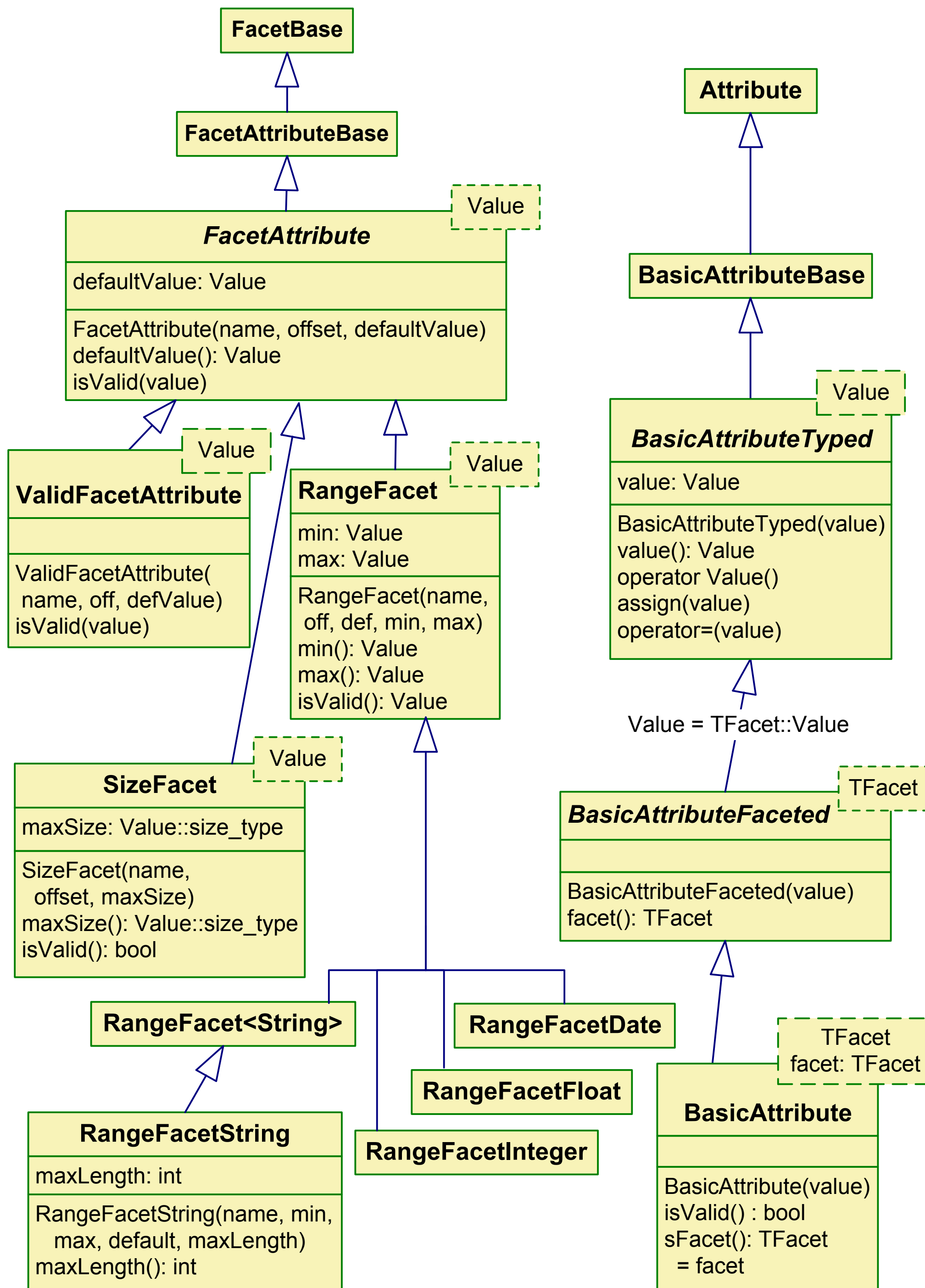


Figura 52. Jerarquía de facets y atributos básicos

2.2.4. Facets y atributos básicos

La librería incluye una familia de clases para representar atributos típicos de uso común. Estos atributos simplemente almacenan localmente un valor y disponen de dos servicios típicos que delegan en su facet: un método de validación y el valor por defecto de iniciación en caso de que no se dé ninguno. La estructura de clases de atributos básicos puede verse en la figura 52.

La clase genérica para crear atributos básicos es BasicAttribute<TFacet, facet> donde TFacet es la clase de facet asociado al atributo y facet es el objeto concreto de clase Tfacet.

Un atributo básico guarda únicamente un valor y dispone de métodos para leerlo (value()) y modificarlo (assign(newValue)). El tipo de datos que almacena un atributo básico viene determinado por el tipo del facet mediante un typedef (TFacet::Value). Value puede ser cualquier tipo con semántica de copia por valor y constructor sin parámetros.

Los atributos básicos implementan el método isValid() (para verificar la validez de un valor) delegando tal responsabilidad en el facet asociado (figura 53). Por eso, el tipo del facet que se asocie a un atributo básico debe disponer a su vez de un método isValid(value).

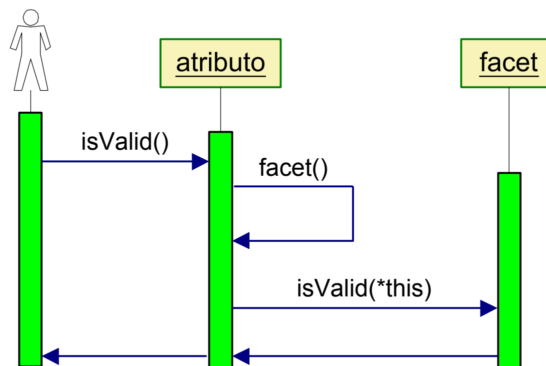


Figura 53. Delegación en el facet de un servicio del atributo

También es responsabilidad del facet el asignar un valor por defecto al atributo en caso de no darle ninguno. Por ello, la clase del facet debe disponer de un método defaultValue() que devuelva ese valor por defecto.

Por último, la clase del facet debe tener un typedef que permita obtener el tipo del valor que se guarda en el atributo. Este es un ejemplo de delegación de información estática de tipos. Sigue el mismo esquema visto en la figura 53 sólo que en este caso se trabaja con tipos en vez de con valores. El tipo de valor que guarda el atributo se define en la case del facet como Value. El atributo obtiene su clase de facet mediante TFacet; por tanto, dentro de la clase del atributo se obtiene el tipo de valor guardado con la expresión de tipos TFacet::Value.

En la jerarquía básica de facets existe una clase de facet simple que puede servir de padre para implementar estos requisitos: FacetAtribute<Value>. Esta clase tiene un campo para almacenar el valor por defecto y una función defaultValue() que lo devuelve. También tiene un typedef para proporcionar al atributo el tipo del valor que almacena y, por último, dispone de un método virtual puro isValid(Value) para comprobar la validez de un valor. Éste método es el único que hay que reescribir para crear una clase de Facet que se pueda asociar a un atributo.

Se han construido unas clases básicas de facets que heredan de FacetAttribute<Value> con implementaciones elementales del método isValid:

- ValidFacetAttribute<Value> da una implementación trivial de isValid() devolviendo true, es decir, todos los valores de un atributo se consideran válidos.

- RangeFacet<Value> permite indicar un rango o intervalo de valores válidos. Almacena internamente el valor mínimo y máximo permitido. Su método isValid devuelve verdad si el parámetro se encuentra dentro de ese intervalo de valores.

- SizeFacet<Value> se usa en clases para las que tenga sentido hablar del tamaño de sus objetos. Por ejemplo, en las listas y los vectores el tamaño es el número de elementos, en cadenas el tamaño es el número de caracteres, etc. El facet comprueba la corrección de un valor mirando si su tamaño no excede del tamaño máximo que se guarda en el facet. El tamaño de un objeto se obtiene mediante el método size() de la clase Value. Todos los contenedores y las cadenas de la librería estándar usan este método para devolver su tamaño por lo que puede usarse este facet

para restringir el tamaño máximo de atributos de estos tipos de datos estándar.

La plantilla RangeFacet<Value> se especializa para clases de cadenas en las que además de un valor mínimo y máximo interesa restringir la longitud máxima de la cadena de caracteres.

Todas las clases de atributos básicos cuyo facet es del mismo tipo tienen una serie de métodos comunes. Por eso la plantilla BasicAttribute<TFacet, facet> hereda de BasicAttributeFaceted<TFacet>. De este modo se consigue dar una implementación única de todos métodos de clases de atributos cuyo tipo de facet sea el mismo aunque el facet concreto sea diferente.

Del mismo modo se hace abstracción de todas las clases que comparten el tipo básico de los valores que guardan aunque el tipo de facet sea diferente. Esta abstracción se materializa en la plantilla BasicAttributeTyped<Value>. Así se consigue, por ejemplo, que aparezcan en un único lugar ciertos métodos de las clases BasicAttributeFaceted< ValidFacet<int> > y BasicAttributeFaceted< RangeFacet<int> >.

Para crear un padre único común a todos los atributos básicos, BasicAttributeTyped<Value> hereda de BasicAttributeBase la cual hereda directamente de la clase Attribute.

```
class Persona {
    static SizeFacet<String> facetNombre;
 public:
    BasicAttribute<SizeFacet<String>, facetNombre> nombre;
    ...
};
Persona p;
p.nombre.facet().setMaxLen(5); //Acceso al facet
p.nombre.assign("pepe"); /* Acceso al atributo (escritura). Se hace la
comprobación de que no se sobrepasa la longitud máxima */
cout << p.nombre.value(); //Acceso al atributo (lectura)
```

Figura 54. Acceso al facet y al valor de un atributo

En la figura 54 se muestra un sencillo ejemplo de uso de atributos y facets.

El operador de asignación se redefine para los atributos redirigiendo la llamada a assign. Además se crea un operador de conversión automática de un atributo al tipo de valor que almacena. De este modo, si se pone el valor de un atributo cuando se espera el tipo subyacente se hará una conversión implícita de uno al otro (figura 55). Gracias a estos métodos, la notación para acceder a los atributos es idéntica a la que se emplearía si en vez de atributos enriquecidos se usaran directamente los tipos que encapsulan. De forma totalmente transparente se hacen las llamadas a assign y value, dándole al usuario la ilusión de que está operando directamente con el valor del atributo.

```
p.nombre = "Paz"; //Aquí se llama implícitamente a p.assign("Paz");
String s = p.nombre; //Llamada implícita a s = p.nombre.value();
```

Figura 55. Acceso transparente al valor del atributo

La eficiencia del uso de atributos en vez de tipos normales se puede comprobar comparando el coste de las llamadas a los métodos de acceso assign y value con el coste de acceder directamente al valor. ValidFacetAttribute tiene la misma semántica que la de acceder directamente al valor pues no hace ninguna comprobación de validez. La llamada a value() para acceder al valor hace lo siguiente:

```
if (facet().isValid()) return _value; else throw Error(...);
```

facet() es una función de expansión que devuelve directamente el parámetro de la plantilla. isValid() es una función de expansión que devuelve directamente true y obsérvese que el compilador conoce el tipo exacto del facet; por ello en tiempo de compilación se conoce la función exacta que se ha de llamar y por tanto se puede expandir, quedando el código:

```
if (true) return _value; else Error(...);
```

Es trivial para el compilador optimizar esta llamada quitando la comprobación de la condicional.

Se ve por tanto que en la zona crítica del manejo de atributos (el acceso a sus valores) no se paga ningún precio en caso de no necesitar comprobación de

valores con lo que se respeta la filosofía de C++ de no pagar por aquello que no se usa.

En vez de (o además de) los nombres assign y value se puede utilizar operator() para acceder al valor del atributo (figura 56). Con esto se consigue poder leer y cambiar el valor de un atributo usando el mismo tipo de notación que en las llamadas a los métodos.

Gracias a esto, los atributos básicos representan una solución directa al patrón *property* [Hastie 95] dentro del lenguaje.

```
Value operator() const { return value(); }
void operator()(const Value& newValue) { assign(newValue); }
...
Persona p;
p.nombre("Pepe"); //Llama a operator()(string&);
cout << p.nombre(); //Llama a operator()
```

Figura 56. Acceso al valor del atributo con notación funcional

```
class Persona {
 private:
     static SizeFacet<String> fnombre;
 public:
     BasicAttribute<SizeFacet<String>, fnombre> nombre;
 private:
     static RangeFacet<int> fedad;
 public:
     BasicAttribute<RangeFacet<int>, fedad> edad;
};
//Persona.cpp
SizeFacet<String> Persona::fnombre(
     "nombre", Persona::nombre, "Anónimo", 10);
RangeFacet<int> Persona::fedad("edad", Persona::edad, 0, 0, 99);
```

Figura 57. Declaración y definición de los atributos de una clase

2.2.5. Macros para declarar atributos y facets básicos

La notación para crear atributos enriquecidos en las clases resulta algo farragosa y obliga al usuario a conocer el sistema de implementación de los atributos enriquecidos con facets. Esto se puede ver en el ejemplo de la figura 57. En él se declara una clase Persona con dos atributos y sus facets asociados.

El hecho de que la información correspondiente a cada atributo se encuentre unida en el código permite utilizar el preprocesador del C++ para simplificar la notación y hacerla transparente a los clientes del sistema de facets:

```
#define DEC_ATRI(TFacet, name) \
 private: \
     static const TFacet* f##name; \
 public: \
     BasicAtribute<TFacet, f##name> name;

#define INIT_DEF_ATRI(TObject, TFacet, name) \
     const TFacet* TObject##::f##name = TFacet( \
          #name, &TObject::name,

#define END_DEF_ATRI );

//Persona.h
class Persona {
 DEC_ATTRI(SizeFacet<String>, nombre);
 DEC_ATTRI(RangeFacet<int>, edad);
};

//Persona.cpp
INIT_DEF_ATTRI(Persona, SizeFacet<String>, nombre)
     "Anónimo", 10
END_DEF_ATTRI

INIT_DEF_ATTRI(Persona, RangeFacet<int>, edad)
     0, 0, 99
END_DEF_ATTRI
```

Figura 58. Declaración de atributos mediante el uso de macros

La macro para declarar los atributos es elemental ya que únicamente depende del nombre del atributo si se utiliza la convención de que el nombre del facet simplemente añade el prefijo f al nombre del atributo. La macro para la definición presenta el problema de que, dependiendo del facet habrá más o menos argumentos en el constructor del facet. Para no tener que definir una macro para cada número de argumentos se ha optado por crear una macro INIT_DEF_ATRI que sustituya el texto hasta la inclusión de argumentos y END_DEF_ATRI para cerrar la definición. En medio de ambas macros el usuario incluirá los parámetros del constructor del facet correspondiente (figura 58). En la figura se ve cómo con estas macros la declaración de la clase Persona se simplifica notablemente, quedando los detalles ocultos.

2.3. Metainformación de métodos

Asociar información a los métodos tiene varias utilidades. Si se incluye el nombre en esta metainformación y se guarda una lista de todos los métodos de una clase se va a poder interpretar y ejecutar un mensaje textual ya que a partir del nombre se consultará en la tabla de métodos aquél cuyo nombre coincida con el dado. También van a beneficiarse de esta información programas de depuración o de creación de documentación automática.

Si se desea incluir un método en la arquitectura de facets hay que declarar un objeto estático en la propia clase que contenga la información de dicho método.

El facet de un método va a ser una instancia de una clase MethodFacet parametrizada por la signatura (tipos de los parámetros y tipo devuelto) del método en cuestión (figura 59).

La plantilla MethodFacet tiene tres parámetros:

- La clase donde se declara el método.
- El tipo de resultado devuelto.
- Una plantilla Params parametrizada por los tipos de los argumentos. Como no se permite sobrecarga de plantillas hay que construir una clase Params0 para 0 argumentos y una plantilla para cada número de argu-

mentos mayor que 0: Params1<T> para un argumento, Params2<T1, T2> para dos argumentos, etc.

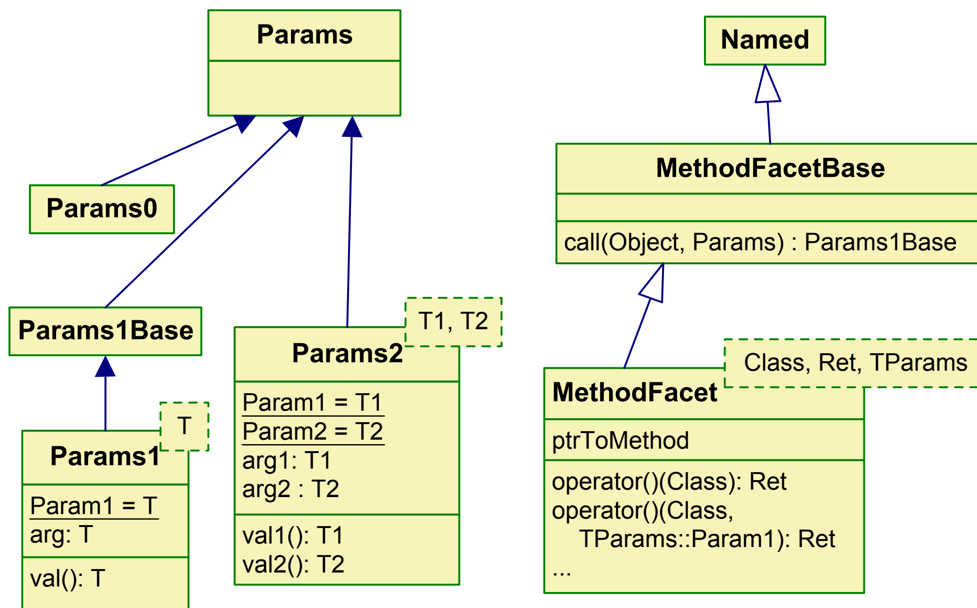


Figura 59. Plantilla para facets de métodos

El constructor toma como entrada la dirección del método asociado y el nombre del método.

MethodFacet incluye un método operator(...) especializado según el número de parámetros del método que encapsula. Este operador tiene la misma signatura que el método subyacente y su ejecución provoca la ejecución de dicho método. Al ser este operador una función de expansión un compilador suficientemente sofisticado puede conseguir que el código generado para la llamada a MethodFacet::operator(...) coincida con el de llamar directamente al método.[5]

La figura 60 muestra cómo se declara un facet de método para un método print de la clase Persona que tome como parámetros una cadena con el forma-

[5] Esto se conseguiría de forma segura si la dirección del método fuera un parámetro de la plantilla en vez de un objeto miembro. Se ha optado por dejarla así para simplificar la notación.

to y un *stream* de salida. Devuelve true si la operación tiene éxito. Para hacer la llamada se utiliza fprint como un funtor que toma como primer argumento el objeto de la clase persona. El resto de argumentos son los parámetros del método print.

Se pueden derivar clases de MethodFacet y reimplementar el operador funcional para personalizar la ejecución del método asociado. Por ejemplo, se puede llamar a un método before antes de la ejecución del método normal y a un método after después. Estos dos métodos serían métodos estáticos de clase. Hay que tener en cuenta, sin embargo, la disminución de eficiencia asociada al manejo de métodos por medio de estos metamétodos.

```
class Persona {
    ...
    bool print(String formato, ostream& lugar) const;
    static
    MethodFacet< Persona, bool, Params2<String, int> > fprint;
};
...
// Persona.cpp:
MethodFacet< Persona, bool, Param2<String, int> >
Persona::fprint(Persona::print, "print");
...
// Ejemplo de uso:
Persona p; ...
Persona::fprint(p, "", cout); //Llama a p.print("", cout)
cout << Persona::fprint.name(); //Escribe print
```

Figura 60. Facets de métodos

La clase MethodFacetBase va a ser la clase raíz de todas las clases cuyos objetos representen métodos. Con esto se puede conseguir guardar en el metaobjeto de una clase una lista de tipo Vector<MethodFacetBase*> con los métodos propios de dicha clase. Como anteriormente, dicha lista se va incrementando dinámicamente conforme se van creando las instancias de los facets de los métodos de la clase. MethodFacetBase incluye un método virtual genérico para realizar una llamada a su método subyacente. En la clase

MethodFacetBase no se conoce ni la clase ni los tipos exactos de los argumentos; por ello, sus argumentos deben ser generales: un argumento que representa un objeto de una clase cualquiera y otro que representa una lista de parámetros y que por tanto pertenecerá a la clase Params.

Por cuestiones técnicas este método se llama apply en vez de usar el operador funcional[6]. No obstante este método sólo van a utilizarlo programas especializados que necesiten trabajar con métodos de forma genérica, por ejemplo un intérprete de llamadas a métodos. Supongamos que f es un MethodFacetBase al que se le ha asignado el método Persona::fprint. Para llamar al método subyacente con dos parámetros concretos hay que hacer:

```
f.apply(p, Params1<String, ostream&>("", cout));
```

2.3.1. Macros de facets de métodos

```
#define DEC_METHOD2(Class, Ret, name, Type1, Type2) \
    static MethodFacet< \
        Class, Ret, Params2<Type1, Type2> > f##name;
    Ret name(Type1, Type2) \
#define DEF_METHOD2(Class, Ret, name, T1, p1, T2, p2) \
    MethodFacet<Class, Ret, Params2<T1, T2> > \
        Class::f##name(#name, Class::name); \
    Ret Class::name(T1 p1, T2 p2)
class Persona {
    DEC_METHOD2(Persona, bool, print, String, int) const;
};
...
DEF_METHOD2(Persona, bool, print, String, format, int, x) const
{...}
```

Figura 61. Macros de facets de métodos

[6] Al redefinir un método en la clase derivada su signatura debe coincidir con la empleada en la clase base.

La figura 61 muestra unas macros para facilitar el trabajo con facets de métodos y las declaraciones resultantes al emplear estas macros.

2.4. Metainformación de clases

En apartados anteriores se han visto las ventajas de asociar información a las clases. Se vio cómo en el caso de los lenguajes que no permiten trabajar con las clases como con cualquier otro dato, es necesario crear un objeto para cada clase que sea su "representante", tanto para almacenar información de la clase como para simular operaciones con ella como si fuera un objeto más.

2.4.1. Metaobjetos estáticos como descriptores de clases

```
class Empleado : public Persona {
 private:
     //Un objeto para cada clase con la información de dicha clase:
     static ClassInfo _classInfo;
 public:
     //Acceso a dicho objeto en ámbito de clase:
     static ClassInfo& sclassInfo() { return _classInfo; }

     //Acceso a dicho objeto en ámbito de objeto:
     virtual ClassInfo& classInfo() const { return _classInfo; }
     ...
};
//Acceso estático a la información de la clase
string s = Empleado::sclassInfo().name(); //"Empleado"

//Acceso a la información de la clase por medio de una instancia
Empleado e;
Persona& p = e;
s = p.classInfo().name(); //"Empleado"
```

Figura 62. Un objeto estático guarda la información de cada clase

La arquitectura que aquí se presenta incluye directamente el objeto asociado a una clase como miembro estático de esa clase. Se añaden también a la clase dos métodos, uno para acceder a ese objeto de forma estática a partir del nombre de la clase y otro virtual para acceder dinámicamente a partir de un objeto de la clase (figura 62). En ambos casos el acceso es directo con coste constante.

La clase ClassInfo a la que pertenece el objeto asociado a una clase se puede considerar como una metaclase ya que sus instancias son clases (en realidad objetos isomórficos a clases). En una primera versión de la librería de facets todos estos objetos pertenecían a una única metaclase de nombre ClassInfo. El problema con esta estructura es que una vez que se obtiene un objeto de ClassInfo ya no es posible llamar a los servicios estáticos de la clase que representan pues la información estática del tipo se ha perdido (figura 63).

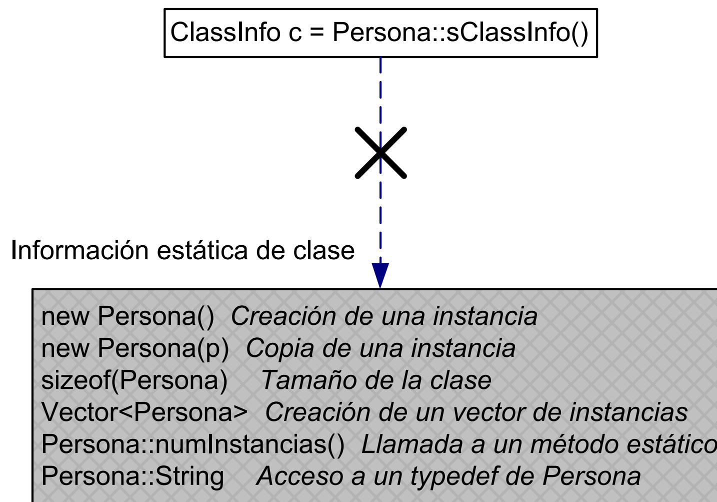


Figura 63. El metaobjeto no puede acceder a los servicios de la clase

Hay que incluir en el ClassInfo métodos que permitan acceder a los servicios estáticos de la clase. Por ejemplo, para acceder al método Persona::numInstancias habrá que implementar en ClassInfo un método (no estático) que de alguna forma llame al método estático numInstancias de la clase Persona. Lo mismo habría que hacer para crear una instancia y para cualquier otro servicio estático. La figura 64 muestra cómo se haría uso de estos métodos.

```
ClassInfo c = Persona::classInfo();
Object o = c.creaInstancia();
//Crea una nueva persona llamando a new Persona()
int x = c.numInstancias();
//Número de instancias de la clase llamando a Persona::numInstancias()
```

Figura 64. Uso del metaobjeto para acceder a servicios estáticos

Una solución consiste en guardar en el metaobjeto las direcciones de los métodos estáticos (si un servicio no es un método estático, por ejemplo new Persona(), se crea un método estático en la clase que lo encapsule). Al constructor de ClassInfo se le pasan estas direcciones (figura 65).

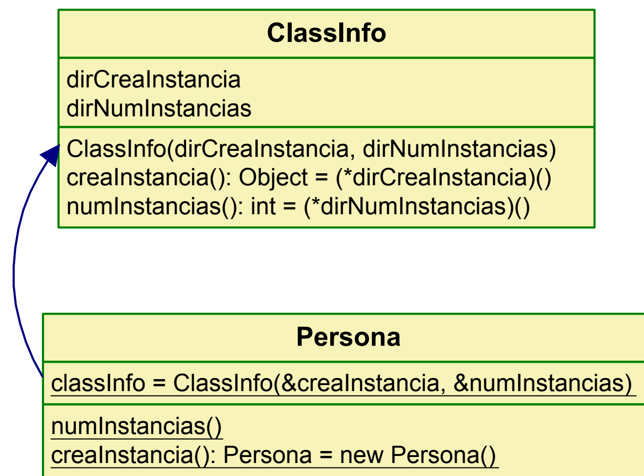


Figura 65. Acceso a servicios estáticos guardando sus direcciones

Una segunda posibilidad, si no se quiere pasar tanta información al constructor de ClassInfo consiste en pasar únicamente un objeto de la propia clase. En cada clase se declarará un método virtual para cada servicio estático y por último, en el ClassInfo, se accederá a cada uno de los servicios estáticos mediante una llamada al método virtual asociado utilizando el ejemplar como objeto destinatario del método (figura 66).

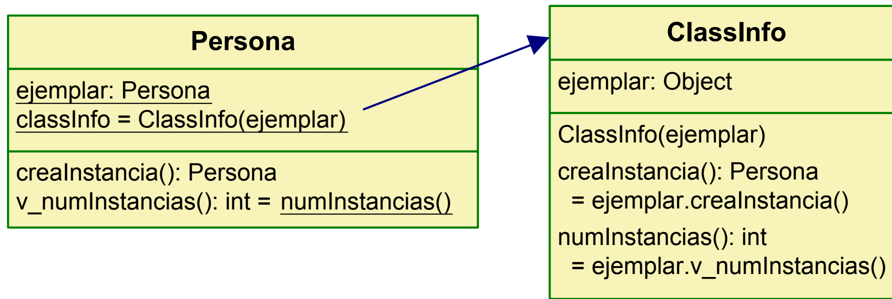


Figura 66. Llamada a métodos estáticos por medio de un ejemplar

Ambas soluciones obligan a replicar código para recuperar la información estática de la clase. En la librería de facets se consigue que el metaobjeto conserve información estática de la clase haciendo que dicha clase sea parámetro de la metaclase ClassInfo. El metaobjeto será la única instancia de dicha clase. Todas las clases parametrizadas heredarán de una única clase ClassInfoBase para abstraer la información común y para poder referirse a un ClassInfo genérico si no se conoce su tipo exacto, por ejemplo, para construir una lista con las clases padre de una dada (figura 67).

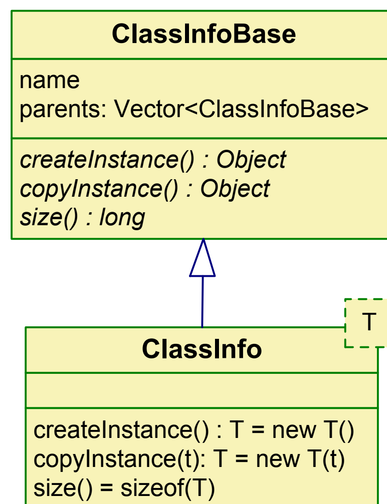


Figura 67. Metaclase parametrizada

Gracias a que el metaobjeto tiene acceso a toda la información estática de su clase T, lo único que hay que incluir en cada clase del modelo es la declaración de dicho metaobjeto como perteneciente a ClassInfo<T> y la de los dos métodos (no estático y estático) que acceden a dicho metaobjeto (figura 68).

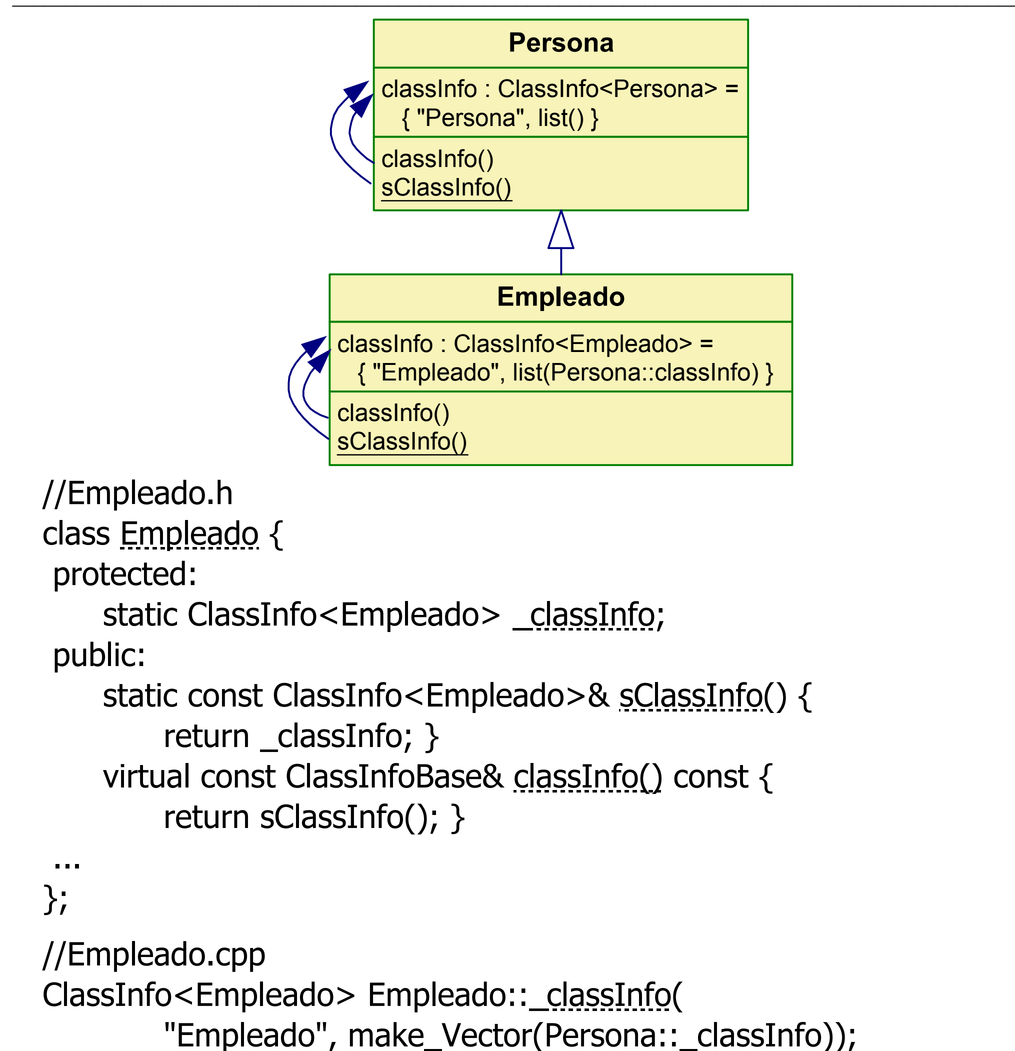


Figura 68. Declaraciones y definiciones para asociar información a las clases

2.4.2. Macros para la inclusión del metaobjeto en las clases

```
#define DEC_METACLASS(ClassInfo, Clase) \
 protected: static ClassInfo<Clase> _classInfo; \
 public: \
     static const ClassInfo<Clase>& sClassInfo() { \
          return _classInfo; } \
     virtual const ClassInfo##Base& classInfo() const { \
          return sClassInfo(); }

#define DEF_METACLASS1(ClassInfo, Clase, Padre1) \
 ClassInfo<Clase> Clase::_classInfo( \
          #Clase, makeVector(&Padre1::_classInfo));

#define DEF_METACLASS2(ClassInfo, Clase, Padre1, Padre2) \
 ClassInfo<Clase>* const Clase::_classInfo( \
     #Clase, makeVector(&Padre1::_classInfo, &Padre2::_classInfo));
//Análogamente hasta DEF_METACLASS4

#define DEC_CLASS_INFO(Clase) \
 DEC_METACLASS(ClassInfo, Clase)
//Análogamente hasta DEF_CLASS_INFO4

//Empleado.h
class Empleado {
 DEC_CLASS_INFO(Empleado)
 ...
};
//Empleado.cpp
DEF_CLASS_INFO1(Empleado, Persona)
...
```

Figura 69. Definición y uso de macros para declarar y definir el metaobjeto

En la figura 68 se ve que, aunque el código que hay que incluir en cada clase parece grande, en realidad se reduce a la declaración y definición de los tres

objetos. Además, dicho código está localizado en cada clase y la única variación de una clase a otra consiste en el nombre de la clase y en indicar la lista de clases padre. Gracias a esto, se puede utilizar el preprocesador de C++ para simplificar la notación mediante la inclusión de dos macros para cada clase, una para la declaración y otra para la definición.

Debido a las limitaciones del preprocesador es necesario crear una macro para cada número de padres de la clase. Al ser este número virtualmente muy pequeño este hecho no presenta mayor problema. La figura 69 muestra estas declaraciones. La macro DEC_CLASS_INFO(Clase) deberá incluirse al principio de la clase Clase. Si por ejemplo Clase tiene 2 padres entonces en el fichero fuente se incluirá la definición del _classInfo de Clase mediante la macro DEF_CLASS_INFO2(Clase, Padre1, Padre2). Estas dos macros se han definido en función de otras más generales en las que se pasa como parámetro adicional el nombre ClassInfo. Se hace así por si se desean introducir otros descriptores de clases (Véase 2.5.4).

La inclusión de la lista de facets propios en el metaobjeto de la clase no se hace en el constructor de dicho metaobjeto pues esto implicaría una redundancia de código con los problemas de mantenimiento consecuentes (la eliminación de un atributo obligaría a eliminar su facet de esta lista). La solución consiste en que el propio facet se añada a la lista de facets en su constructor. Aunque el metaobjeto y el facet que se añade son objetos globales no hay problema con el orden de iniciación ya que están ambos elementos definidos en el mismo fichero (unidad de traducción, según la terminología de [Stroustrup 97]) y la definición del metaobjeto aparece antes que la de los facets de modo que su lista de facets se inicia a la lista vacía y posteriormente cada facet se va añadiendo a dicha lista.

2.4.3. Ampliación de la información de las clases

Siguiendo la terminología del capítulo 1, ClassInfo<T> es una metaclase pues sus instancias son objetos que representan clases. ClassInfo<T> genera únicamente una instancia: T::_classInfo. Por tanto, estamos en presencia de una estructura de metaclase única por cada clase al modo de Smalltalk. Esto es mucho más flexible que tener una metaclase única para todo el modelo (como ocurre por ejemplo en Java) ya que se puede personalizar la metainformación clase por clase. Ciertamente ClassInfo<T> es en realidad una única clase pa-

rametrizada por el tipo T pero gracias a la especialización de plantillas se pueden dar versiones concretas de ClassInfo<T> para los tipos que se desee.

En la figura 70 se muestra una especialización de ClassInfo para la clase Persona. La metaclase de Persona tiene redefinido el método makeInstance para que escriba información de depuración cada vez que se cree una instancia. Para facilitar la tarea de creación de especializaciones se ha introducido en la librería de facets una clase intermedia ClassInfo0 entre ClassInfo<T> y ClassInfoBase. ClassInfo0 tiene todos los campos y métodos que en principio deberían estar en ClassInfo<T>. Ésta última no añade campos ni métodos. Para crear ahora la especialización ClassInfo<Persona> basta con derivar de ClassInfo0<Persona> con lo que se conseguirá heredar el comportamiento por defecto e incluir las modificaciones oportunas. Si ClassInfo<Persona> derivara directamente de ClassInfoBase habría que crear trabajosamente todos los atributos y métodos del mismo modo que se hace en ClassInfo<T>. Sin la clase intermedia no se puede hacer esto ya que no se permite derivar ClassInfo<Persona> de ClassInfo<T> con T = Persona.

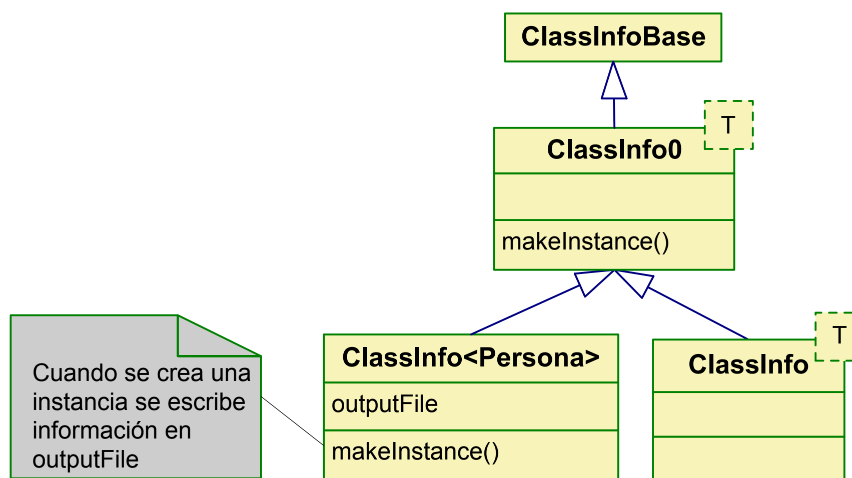


Figura 70. Especialización de la metaclase que gobierna Persona

La especialización de ClassInfo para Persona es un ejemplo de polimorfismo en tiempo de compilación. En el caso del polimorfismo tradicional se selecciona un método en tiempo de ejecución en función del tipo del objeto. En el caso actual, cada clase tiene un metaobjeto que pertenece a un cierto tipo

(metaclase). Este tipo se selecciona en tiempo de compilación en función de la clase: para Persona se selecciona la especialización ClassInfo<Persona>, para otros se usa el molde general. No hay necesidad de modificar las declaraciones en la clase Persona: las macros en la clase Persona siguen funcionando correctamente pues hacen referencia a ClassInfo<Persona>.

El método anterior tiene la desventaja de que hay que ir declarando las metaclases una por una. Se describe a continuación una forma de crear familias de metaclases de tal modo que se pueda precisar el comportamiento común de todas ellas mediante derivación de una metaclase padre común. La figura 71 ilustra el modo de hacerlo. En ella se observa una familia de metaclases ClassInfoDebug<T> que tiene modificado el método makeInstance() para imprimir información de depuración. ClassInfoDebug<T> deriva de ClassInfo<T>, para heredar el comportamiento estándar y modificarlo. Para almacenar la información común a todas estas metaclases de modo que se pueda trabajar de forma genérica con ellas se hace derivar ClassInfoDebug<T> de ClassInfoDebugBase.

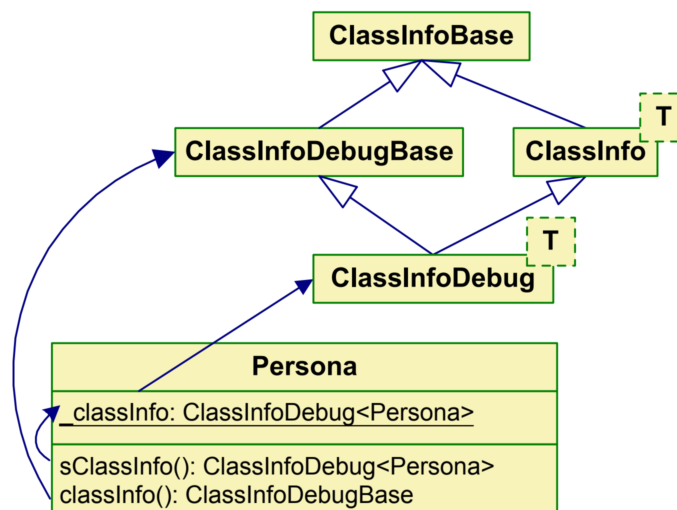


Figura 71. Creación de una nueva metaclase

En esta situación, cuando se cree una clase del modelo habrá que precisar si su metaclase es la estándar ClassInfo o una distinta como ClassInfoDebug. Para ello en las macros habrá que sustituir ClassInfo<T> por ClassInfoDebug<T> y en el método virtual genérico que antes devolvía

ClassInfoBase ahora hay que indicar que devuelve ClassInfoDebugBase. Como ClassInfoDebugBase deriva de ClassInfoBase el mecanismo del polimorfismo funcionará correctamente .

```
class Persona {
 DEC_METACLASS(ClassInfoDebug, Persona)
 ...
};
//Persona.cpp
DEF_METACLASS0(ClassInfoDebug, Persona)
```

Figura 72. Indicación de la metaclase a la que pertenece una clase

Las macros genéricas de la figura 69 permiten declarar fácilmente la metaclase asociada a una clase (figura 72).

Si se deriva de la clase Persona hay que tener en cuenta el problema de la compatibilidad de metaclases descrito en el capítulo anterior. Por ejemplo, si Empleado deriva de Persona y se le asocia como metaclase el ClassInfo estándar el compilador dará un mensaje de error ya que Empleado::classInfo() devuelve un ClassInfoBase que es una clase que no deriva de la clase que devuelve Persona::classInfo() y que es ClassInfoDebugBase. No podía ser de otro modo si C++ pretende seguir siendo un lenguaje seguro de tipos. Si outputFile es un método de ClassInfoDebugBase que devuelve el fichero donde se imprime la información de depurado entonces la ejecución del código de la figura 73 produciría la llamada a un método inexistente.

```
Persona& p = *new Empleado;
p.classInfo().outputFile();
```

Figura 73. Compatibilidad de metaclases

Las macros de creación de metaclases de la figura 69 no están diseñadas para admitir argumentos adicionales al constructor de los metaobjetos (por ejemplo para indicar el fichero de salida). No obstante, resulta muy sencillo modifi-

car o ampliar estas macros para permitir argumentos utilizando la misma técnica que la empleada para declarar argumentos de los facets vista en la figura 58.

[Zarazaga 00 cap. 3] presenta una implementación de persistencia usando la tecnología de facets que demuestra las posibilidades de ampliación de la librería de facets. La figura 74 muestra el esquema de esta implementación.

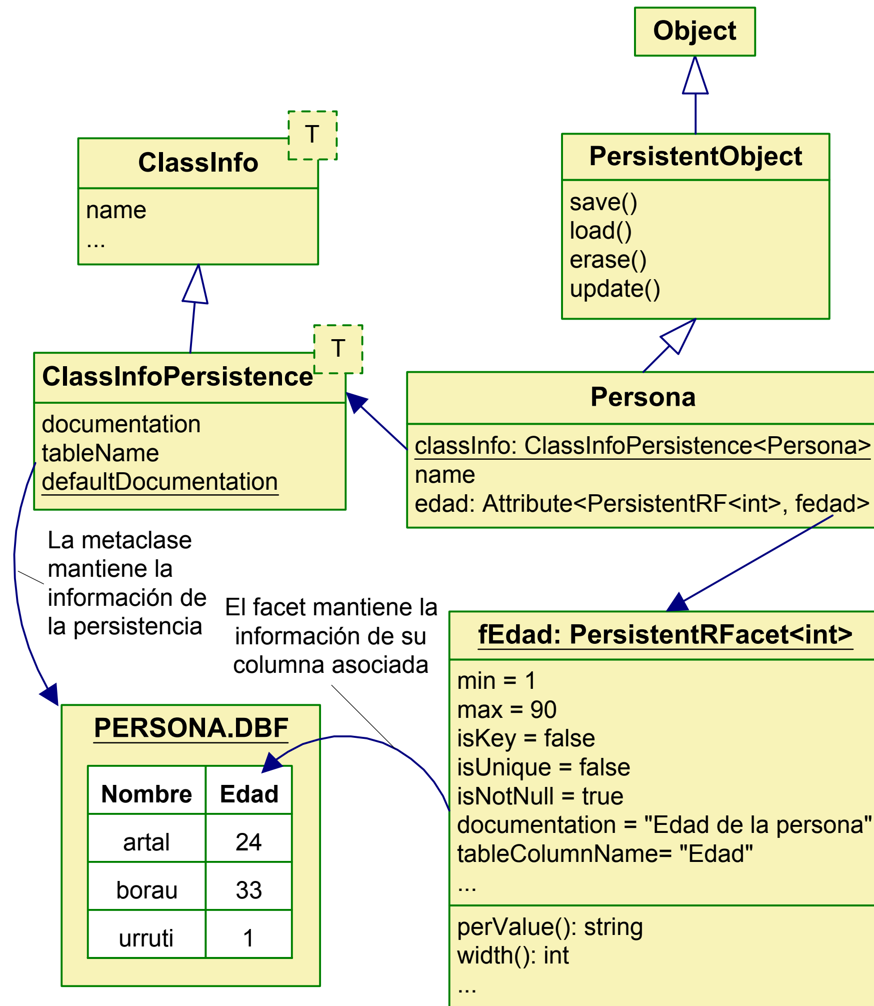


Figura 74. Esquema de la arquitectura de persistencia

La metaclase ClassInfoPersistence permite dotar de facilidades de persistencia a aquellas clases que la declaren como su metaclase. La persistencia se

realiza en tablas de bases de datos relacionales. La infraestructura define un conjunto de facets especializados en persistencia que guardan la información adicional típica de las columnas de tablas relacionales (si es campo clave, si se permiten valores nulos, etc.). El recorrido de lista de facets permite construir automáticamente las tablas y guardar y recuperar instancias de la clase. Las clases persistentes deben derivar de PersistentObject que implementa servicios para que los objetos puedan guardarse y recuperarse de las tablas.

La figura 75 muestra un ejemplo de lo sencillo que resulta trabajar con objetos persistentes. Un objeto se guarda a sí mismo con el método save. Este método construye la cadena SQL que al ejecutarse lo añade a la tabla.

```
Persona p;
p.nombre = "Pepe"; p.edad = 25;
p.save();
```

Figura 75. Guardado de un objeto en su tabla

En este ejemplo, la cadena SQL generada por el método save es

```
insert into Persona (nombre, edad) values ("Pepe", 25)
```

La cadena se puede construir sin problemas gracias a que se puede recorrer la lista de facets para obtener sus nombres y sus valores.

2.5. Información global del modelo

2.5.1. Clase raíz del modelo

Todas las clases del modelo derivan en última instancia de una clase raíz llamada Object (figura 76) que permite trabajar de forma genérica con cualquier objeto de cualquier clase del modelo.

Object

classInfo = {"Object", nullVector}

getAttribute(facet): Attribute
attributes(): Vector<Attribute>
print(where)

Figura 76. Clase raíz del modelo

En Object se implementan los servicios comunes a todas las clases. Por ejemplo attributes() devuelve la lista de atributos de un objeto a partir de la lista de facets de su metaobjeto. Para cada facet se obtiene el valor del atributo para el objeto en cuestión. Es muy habitual obtener únicamente una lista con los nombres de los atributos (para hacer tablas de interfaz gráfica o fabricar consultas SQL). El servicio attributesPrintForm() devuelve dicha lista. attributes() y attributesPrintForm() son dos ejemplos de contenedores que no guardan datos; realmente no se crea una lista con todos los atributos, en su lugar se usa la lista de facets y dinámicamente se va obteniendo cada atributo asociado. Estos contenedores, que realmente no contienen físicamente los datos que van proporcionando, los denominamos *contenedores virtuales* y se explican en el capítulo 4 de esta tesis.

2.5.2. Lista de clases del sistema

La librería de facets mantiene una lista con todos los metaobjetos que se creen, o, lo que es lo mismo, con todas las clases del modelo. Esto va a permitir realizar tareas de depuración y va a aumentar las posibilidades interpretativas del lenguaje. Por ejemplo, dada una cadena con el nombre de una clase se va a poder acceder al metaobjeto de dicha clase y por tanto a sus servicios, principalmente al de creación de instancias (simplemente habrá que buscar en la lista de metaobjetos aquél que tenga como nombre el de la cadena). Esta característica va a facilitar notablemente la implementación de esquemas de persistencia. Por ejemplo para guardar un objeto en un fichero bastará con escribir el nombre de su clase seguido del contenido de sus atributos. A la hora de recuperar el objeto se leerá el nombre de la clase, se creará una instancia vacía y se leerán del fichero los datos para rellenar sus campos.

Para almacenar esta información global de las clases se necesita una lista classes de tipo Vector<ClassInfo*> con todos los metaobjetos del modelo. Esta lista es una variable global. Cada vez que se define un metaobjeto y se llama a su constructor, dicho metaobjeto se autoañade a la lista de metaobjetos. Hay que tener en cuenta que tanto classes como los metaobjetos son variables globales y el estándar no garantiza ningún orden de iniciación de los objetos globales. Sin embargo es necesario que classes se inicie antes que los metaobjetos. [Ellis 90] proporciona una solución al problema de establecer el orden de inicialización de objetos globales que no es del todo satisfactorio ya que depende de la implementación. Una solución mejor es declarar classes como puntero a un vector de ClassInfos. El estándar asegura que este puntero se iniciará con valor nulo (inicialización estática) antes de que los metaobjetos se inicien (inicialización dinámica). En el constructor de ClassInfo<T> se comprueba si el puntero es nulo, en cuyo caso se creará dinámicamente el vector classes reservando memoria para el puntero. Posteriores definiciones de metaobjetos ya no reservarán memoria puesto que el puntero ya no será nulo.

2.5.3. Lista de objetos del modelo

Se muestra a continuación un sistema sencillo que se puede emplear para almacenar información de los objetos del modelo que permitirá guardar y recuperar el estado del sistema.

En el apartado anterior se ha visto la forma de acceder a la lista de metaobjetos del modelo. Para guardar la lista de los objetos del modelo bastará por tanto con almacenar en cada metaobjeto una lista con los objetos de la clase que representa. En esa lista se almacenará, junto con cada objeto, un nombre que servirá como descriptor para realizar tareas de persistencia además de ayudar en tareas de depuración e interpretación.

Cada metaobjeto tendrá un campo que será un vector de punteros a objetos de la clase a la que representa. Cada vez que se construya un objeto habrá que añadirlo al vector (figura 77).

Nada más construir la persona se la añade a la lista de objetos de su clase. Lo que se añade no es exactamente una persona, sino un par String-Persona para incluir información textual del objeto. ClassInfo<T> tendrá por tanto un campo de tipo Vector< Pair<String, T*> >. No parece conveniente incluir el nombre del objeto en la propia clase ya que en muchas ocasiones sólo se desea-

rá incluir un reducido conjunto de objetos en la lista de objetos del sistema y en otras ocasiones no se deseará ni tan siquiera utilizar esta característica. Con la solución propuesta no se sobrecarga el sistema innecesariamente.

```
Persona pepe(...);
Persona::classInfo().addObject(pepe, "pepe"));
```

Figura 77. Adición de un objeto a la lista de objetos de su clase

Muchos objetos se van a crear en ámbito global; en este ámbito no puede haber sentencias de código como la segunda de la figura 77. Se podría modificar el constructor de las clases del modelo de manera que se hiciera esta operación adicional pero esto obligaría a implementar código manualmente clase por clase y constructor por constructor. La solución consiste en utilizar un objeto perteneciente a una clase especial cuyo constructor admita un objeto como parámetro y lo añada a la lista de objetos de su clase. El constructor de este objeto instalador junto con la manera de declarar un objeto se muestra en la figura 78.

```
Persona pepe("Paz Sánchez", 35);
Installer __pepe(pepe, "pepe");
...
struct Installer {
    Installer(Object& o, String s) {
        o.classInfo().addObject(o, s));
    }
};
```

Figura 78. Adición de un objeto a la lista de objetos de su clase

Unas macros sencillas simplifican la declaración de variables. La figura 79 nuestra un ejemplo.

```
#define DEC2(Class, name, a1, a2) \
    Class name(a1, a2); \
    Installer __##name(name, #name);
...
DEC2(pepe, "Paz Sánchez", 35)
```

Figura 79. Macro para crear objetos

2.6. Herencia de facets

2.6.1. Herencia con modificación del valor del facet

Un facet permite asignar propiedades a los atributos de una clase. Cuando se hereda de una clase, cada atributo heredado tiene asociado el mismo facet de la clase padre. Desde un punto de vista de la orientación a objetos es deseable que al heredar un facet se puedan especializar las características del mismo.

La figura 80 muestra un ejemplo de una clase FicheroDOS que deriva de Fichero. FicheroDOS hereda el atributo nombre de la clase Fichero.

```
class Fichero : public Object {
    static SizeFacetString fNombre;
    Attribute<SizeFacetString, fNombre> nombre;
    //nombre del fichero
    ...
};
class FicheroDOS : public Fichero {
    //atributo nombre heredado de Fichero
};
```

Figura 80. Ejemplo de necesidad de la herencia de facets

El atributo nombre de las clases Fichero y FicheroDOS tiene el mismo facet asociado: fNombre. fNombre no puede cambiarse en la clase derivada ya que en C++ no es posible personalizar el tipo de los atributos, y en el tipo del atributo nombre figura el facet fnombre. Debido a ello no se puede conseguir que el facet del atributo nombre para la clase FicheroDOS tenga la restricción de que el nombre no pueda tener más de ocho caracteres o de que ciertos caracteres no puedan formar parte de él.

En este apartado se presenta una modificación de la arquitectura del sistema de facets que permite la reescritura de los mismos. La solución adoptada se basa en el lema "*Todo problema de programación se resuelve aplicando un acceso indirecto adicional*". El acceso al facet se encapsula dentro de un método no estático y el atributo se parametriza por medio de ese método en vez de por el facet mismo (figura 81).

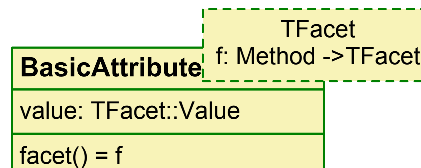


Figura 81. Atributo parametrizado por un método

```
class Fichero : public Object {
 protected:
     static SizeFacetString fnombre; // Longitud máxima 255
     virtual SizeFacetString& vfnombre() { return fnombre; }
     BasicAttribute<SizeFacetString, vfnombre> _nombre;
 public:
     BasicAttribute<SizeFacetString, vfnombre>& nombre() {
          return _nombre; }
};

//Fichero.cpp
SizeFacetString Fichero::fnombre(
     "nombre", Fichero::_nombre, 255, "Anónimo");
```

Figura 82. Declaración de atributos con facets virtuales

```
class FicheroDOS : public Fichero {
 protected:
      static SizeFacetString fnombre;// Longitud máxima 8
      virtual SizeFacetString& vfnombre() { return fnombre; }
      //El atributo y el método que accede a él se heredan
      //automáticamente; no hay que ponerlos
 ...
};

//FicheroDOS.cpp
SizeFacetString Fichero::fnombre(
      "nombre", Fichero::nombre, 8, "Anónimo");
```

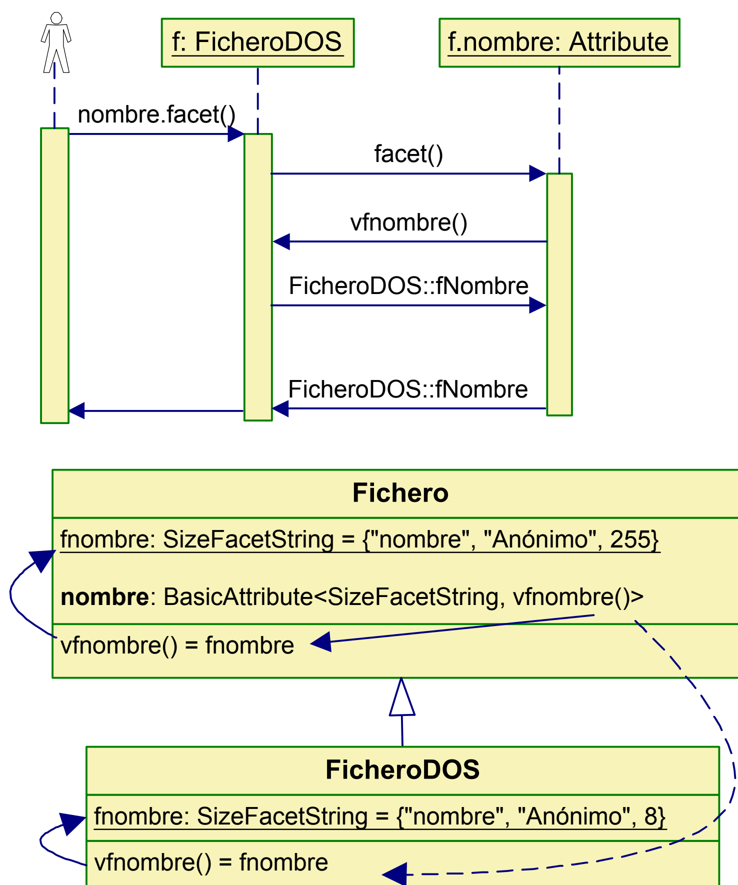


Figura 83. Mecanismo de la herencia de facets

La figura 82 muestra cómo se incluye un atributo de este tipo y su facet en una clase del modelo. Por motivos que se justificarán luego el atributo se hace privado y se proporciona un método público que lo devuelve.

Para redefinir el facet en una clase derivada basta con declarar el nuevo facet y redefinir el método virtual para que lo devuelva (figura 83). Al parametrizar el atributo por el método en vez de por el facet directamente, la llamada p.nombre().facet() se transforma en p.vfnombre() siendo vfnombre() un método virtual que devuelve el facet nombre de la clase de p.

```
class Fichero : public Object {
 DEC_ATTRI(SizeFacetString, nombre);
 ...
};
class FicheroDOS {
 REDEC_ATTRI(SizeFacetString, nombre);
 ...
};
INIT_DEF_ATTRI(Fichero, SizeFacetString, nombre)
     "Anónimo", 255
END_DEF_ATTRI
INIT_REDEF_ATTRI(FicheroDOS, SizeFacetString, nombre)
     "Anónimo", 8
END_DEF_ATTRI
```

Figura 84. Declaración de atributos con herencia mediante macros

Como en el caso de atributos sin herencia todas las declaraciones que tienen que ver con un atributo se encuentran agrupadas en el código y dependen únicamente del nombre del atributo y de los parámetros del facet. Se pueden construir por tanto unas macros sencillas para simplificar la notación. DEC_ATTRI, INIT_DEF_ATTRI y END_DEF_ATTRI se crean de modo totalmente análogo a sus correspondientes de atributos sin herencia. En el caso de redefinir un facet la única diferencia es que no se incluye la declaración del atributo, solamente el facet nuevo y el método que accede a él. La macro REDEC_ATTRI se encarga de esto. INIT_REDEF_ATTRI y END_REDEF_ATTRI coinciden con

sus correspondientes INIT_DEF_ATTRI y END_DEF_ATTRI y se incluyen sólo por simetría. En la figura 84 se muestra el uso de estas macros.

2.6.2. Herencia con modificación del tipo del facet

En el ejemplo anterior, el tipo del facet heredado era el mismo que en la clase base. Puede ser interesante que el tipo del facet heredado herede a su vez del facet de la clase padre. Por ejemplo, el facet fnombre de FicheroDOS podría ser de una clase RestrictedFacet hija de SizeFacet que además del límite de la longitud pusiera la restricción adicional de que ciertos caracteres no puedan formar parte del nombre del fichero.

En la clase derivada (FicheroDOS) se declara el facet fnombre como perteneciente al nuevo tipo (RestrictedFacet en este caso). El método virtual vfnombre() devuelve un SizeFacet. Como el facet que tiene que devolver es el fnombre que pertenece a RestrictedFacet no hay ningún problema en especificar que debe devolver un SizeFacet ya que aquélla deriva de ésta y se puede asignar un RestrictedFacet a un SizeFacet (figura 85).

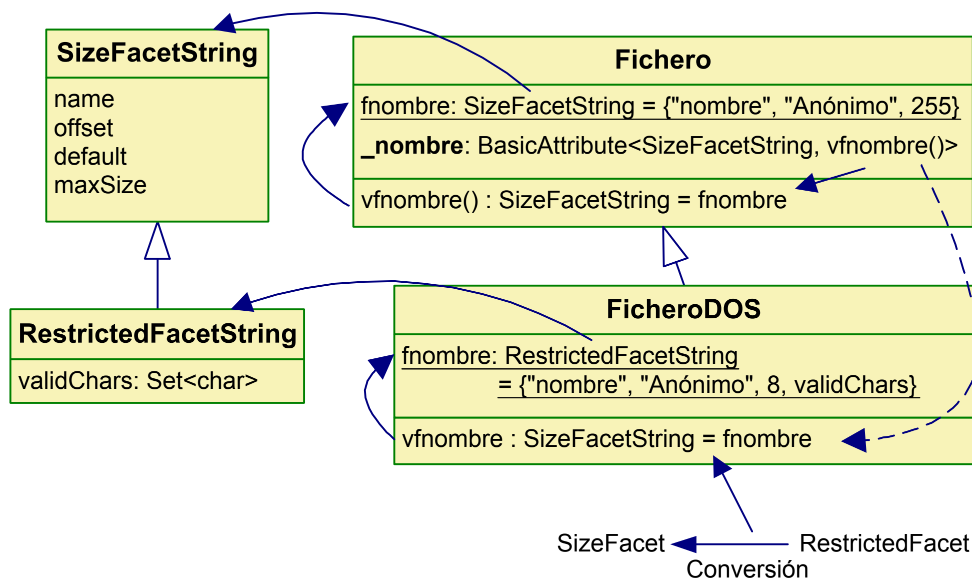


Figura 85. Modificación en herencia del tipo del facet

Por tanto, es posible redefinir el facet en la clase derivada haciendo que sea de un tipo diferente siempre que dicho tipo herede del tipo de facet de la clase base.

Obsérvese que hay una pérdida de información al acceder al facet derivado debido a la relajación en el tipo devuelto por vfnombre para FicheroDOS, por eso, si f es un objeto FicheroDOS es ilegal hacer

```
f.nombre().facet().validChars
```

aunque el facet del nombre de FicheroDOS sea un RestrictedFacetString. No queda otra solución que hacer una antiestética conversión:

```
static_cast<RestrictedFacetString&>(f.nombre().facet()).validChars
```

```
class FicheroDOS : public Fichero {
 protected:
     static RestrictedFacetString fnombre;
     virtual RestrictedFacetString& vfnombre() { return fnombre; }
 public:
     BasicAttribute<RestrictedFacetString, vfnombre>&
     nombre() {
          return reinterpret_cast<BasicAttribute<
           RestrictedFacetString, vfnombre>&>(_nombre);
     }
};
...
FicheroDOS f(...);
f.nombre().facet().validChars = ...;
```

Figura 86. Covarianza del tipo devuelto

En C++ para poder redefinir en una clase derivada un método virtual se debe cumplir que el método de la clase derivada tenga el mismo nombre que el de la clase base con los mismos parámetros de llamada y tipo de devolución. Afortunadamente, ha habido una modificación en este sentido en la versión estándar del lenguaje y ahora se permite que el tipo devuelto en la clase derivada herede del tipo devuelto en la clase base y que además dichos tipos sean referencias o punteros [Saks 92]. Esta regla se denomina covarianza del tipo

devuelto [Meyer 99]. Gracias a esto es correcto hacer que vfnombre() devuelva un RestrictedFacet& (figura 86). Sin embargo, como al facet se accede por medio del atributo y el tipo del atributo no se puede modificar resulta que f.nombre.facet() sigue devolviendo un SizeFacet. La solución es simple, y por eso se ha introducido un método de acceso al atributo: se redefine el método de acceso de modo que devuelva un atributo parametrizado por un RestrictedFacet.

2.6.3. Herencia de facets con modificación del tipo del facet y reaprovechamiento de valores

Para aprovechar el mecanismo de la herencia de facets de forma plena debería permitirse en el caso anterior que el facet derivado tomara por omisión los valores del facet base pudiendo reescribirlos en el caso de que se desee. La figura 87 muestra un ejemplo. En él se observa que en el facet derivado hay que escribir de nuevo los valores del facet padre, lo cual es contrario a la filosofía de orientación a objetos.

```
class Fichero : public Object {
    static SizeFacetString fnombre;
    ...
};
SizeFacetString Fichero::fnombre(
    "nombre", "Anónimo", &Fichero::_nombre, 255);
class FicheroDOS : public Fichero {
    static RestrictedFacetString fnombre;
    ...
};
Set<char> validChars(...);
RestrictedFacetString Fichero::fnombre(
    "nombre", "Anónimo", &FicheroDOS::_nombre, 8, validos);
```

Figura 87. Redundancia de valores en herencia de facets

El uso de un constructor que tome como parámetro un objeto de la clase base permite eliminar esta redundancia. El resto de los parámetros sirven para iniciar los atributos propios de la clase y para modificar los valores heredados (figura 88).

```
class SizeFacetString {
      String _name;
      String _default;
      Offset _off;
      long _maxSize;
 public:
      SizeFacetString(String name, String default,
                       Offset off, long maxSize):
            _name(name), _default(default),
            _off(off), _maxSize(maxSize) {};
};
class RestrictedFacetString: public SizeFacetString {
 private:
      typedef SizeFacetString Parent;
      Set<char> _validChars;
 public:
      //Constructor normal:
      RestrictedFacetString(String name, ...

      //Constructor para reaprovechar valores
      RestrictedFacetString(const Parent& parentObject,
            Set<char> validChars, const Action& action)
                  :SizeFacetString(parentObject),
                  _validChars(validChars) { action(*this); }
      ...
}
```

Figura 88. Constructor para reaprovechar valores

El constructor para aprovechar valores de RestrictedFacetString admite tres argumentos:

- parentObject. Es el facet del que hereda. Gracias a este dato se crea la parte SizeFacet que hay en el objeto que se está construyendo con una copia del valor del facet del que hereda. Los cuatro campos que RestrictedFacetString hereda de SizeFacetString se rellenan con los valores del facet heredado sin necesidad de hacerlos explícitos.

- validChars. Es un parámetro para asignar un valor inicial a un campo propio del facet heredado. Si hubiera más campos propios lo normal es que hubiera igualmente más parámetros en el constructor para indicar sus valores.

- action. Éste es un objeto funcional que pertenece a la clase Action<A> y representa una acción (todavía sin ejecutar) sobre un objeto de A. Para ejecutar la acción que guarda un Action<A> hay que llamar a su método operator() con un parámetro que es precisamente un objeto de A. En el cuerpo del constructor de RestrictedFacetString se ejecuta la acción mediante la llamada al operador funcional operator() pasándole como parámetro el objeto que se está construyendo.

Como muestra la figura 89 el facet derivado hereda por omisión los valores del facet base, establece el valor del campo propio validChars y por último especifica un parámetro con la acción que se debe realizar para modificar valores del facet heredado. En este caso la acción toma dos parámetros que son el método a llamar (setMaxSize) y el argumento del método a llamar (8). Por eso, cuando se ejecuta el constructor y se ejecuta la orden action(*this) se produce la llamada (*this).setMaxSize(8). Los objetos función encapsulan una función o acción. Son objeto de atención en el capítulo 4 de este trabajo.

```
SizeFacetString Fichero::fnombre(
    "nombre", "Anónimo", &Fichero::_nombre, 255);

RestrictedFacetString FicheroDOS::fnombre(
    Fichero::_fnombre,
    Set<Char>(...), action(SizeFacetString::setMaxSize, 8))
);
```

Figura 89. Reaprovechamiento de valores del facet base

2.6.4. Herencia y construcción de la lista completa de facets

Más arriba se ha comentado cómo la lista propia de facets de una clase junto con la lista de clases de las que deriva permite obtener dinámicamente la lista completa de facets. Si se permite la redefinición de facets puede que haya un facet en la clase padre que no deba aparecer en la clase hija debido a que se haya sustituido por otro. Por tanto, si se permite la redefinición de facets hay que incluir la lista completa de facets en cada clase (figura 90).

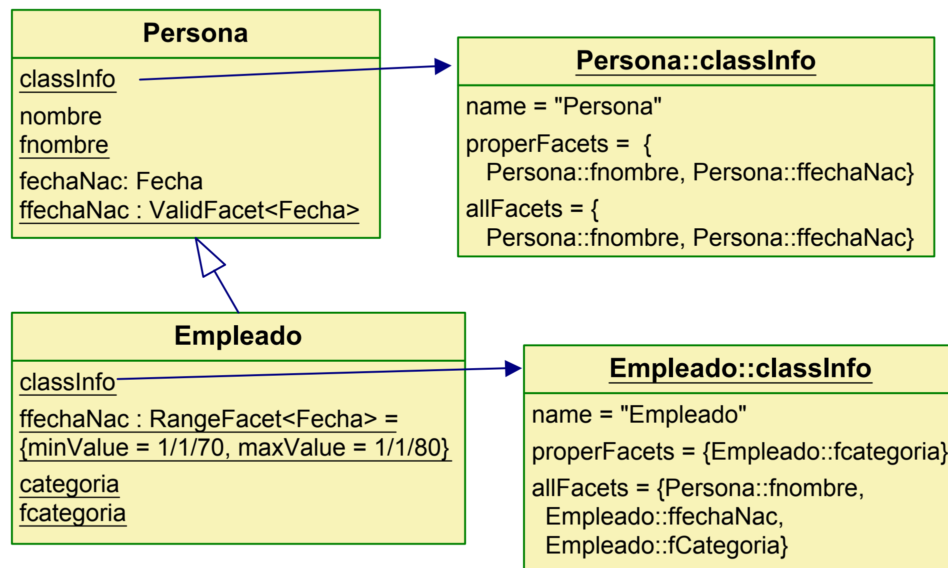


Figura 90. Lista completa de facets explícita

Durante el proceso de iniciación de variables globales se van generando las listas de facets propios ya que el constructor de cada facet lo añade a su lista correspondiente. La generación de la lista completa de facets debe esperar a la ejecución de la función `main()` ya que es en ese momento cuando se puede garantizar que todas las listas están completas. Se precisa por tanto que en la función `main` se llame a un método iniciador de la librería, `initFacets()`, que construirá la lista completa de facets para cada clase. En esencia, `initFacets()` lo que hace es realizar un test para cada facet por si debe ser añadido o no. Para hacer este test se construye un objeto de la clase. Con ese objeto y ese facet se accede al atributo y si el facet de ese atributo coincide con el facet original hay que añadirlo a la lista. Por ejemplo, al construir la lista de todos los facets de

Empleado se comprueba cada uno de los facets de Persona por si debe ser incluido. Al llegar a Persona::ffechaNac se accede a su atributo para un empleado. A este atributo se le pregunta por su facet para un empleado, el resultado es Empleado::ffechaNac por tanto no se añade a la lista.

---

1. *Para cada* classInfo *de la lista de* classInfo*s:*
2. *Añadir a* allFacets *la lista de facets propios*
3. *Generar la lista de* allFacets *de las clases padre si no se ha hecho*
4. *Para cada facet de la lista de* allFacets *de las clases padre*
5. *Obtener el facet asociado al atributo de dicho facet para un objeto de la clase.*
6. *Si coincide con el propio facet añadirlo a la lista, si no, no se añade*

Figura 91. Algoritmo de construcción de la lista de facets

2.6.5. Herencia de facets en casos de herencia múltiple

Si hay herencia múltiple en la jerarquía de clases del modelo puede ocurrir que una clase herede de dos clases padre sendos atributos y por tanto sendos facets con el mismo nombre. En la figura 92 la clase Cliente tiene un atributo cuenta que representa la cuenta de crédito del cliente con su empresa. El atributo cuenta de Empleado representa la cuenta donde se ingresa el salario del empleado. La clase ClienteEmpleado sirve para representar clientes que son empleados a la vez.

Un ClienteEmpleado tendrá dos cuentas, una como cliente y otra como empleado. Se heredarán por tanto dos métodos facet fcuenta(), uno por cada clase. El primero de ellos devolverá Cliente::_fcuenta y el segundo Empleado::_fcuenta.

La situación anterior también se produce si las dos clases padre heredan de una misma clase: ambas clases heredarán los atributos de esta clase padre y estos atributos aparecerán repetidos en el hijo común de dichas clases. Es el caso del atributo formato de la figura 92.

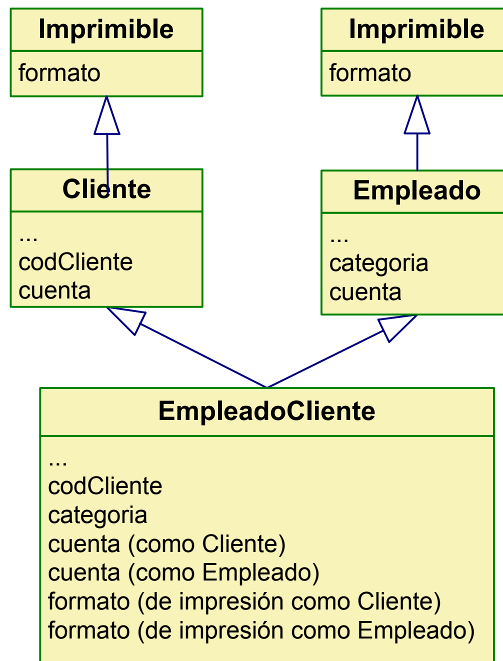


Figura 92. Colisión de nombres de facets en herencia múltiple

Al contrario que Eiffel, C++ permite esta colisión de nombres. La ambigüedad se resuelve obligando a indicar la clase de procedencia en la llamada (figura 93).

```
EmpeleadoCliente e;
//Acceso al facet de la cuenta como empleado por medio del atributo
cout << e.Empleado::fechaNac().facet();
//Acceso ambiguo, error
cout << e.fechaNac().facet();
```

Figura 93. Supresión de la ambigüedad al acceder a facets repetidos

La lista de facets de EmpleadoCliente tiene por tanto facets distintos, con el mismo nombre y distintos atributos asociados. El hecho de que dos facets tengan el mismo nombre impide acceder a un facet a partir de su nombre y dificulta otras tareas como la de construir una tabla relacional con los nombres

guardados en los facets. Si el usuario de la clase quiere realizar este tipo de operaciones deberá redefinir el facet para modificar el nombre. Por ejemplo, en el caso de los dos facets cuenta que hereda EmpleadoCliente habría que redefinir el facet Cliente::fcuenta de modo que su campo nombre pasara a valer "cuenta_cliente".

Si se heredan dos métodos con el mismo nombre y signatura de dos clases diferentes y se redefine el método, sucede que dicha redefinición afecta a los dos métodos heredados. C++ no permite redefinir uno de ellos. Así, si se redefine fcuenta de modo que su nombre pase a ser cuenta_cliente este cambio afectará a los dos facets. Desde luego esto no es lo que se desea.

Según [Stroustrup 97] son pocos los casos en que esto sucede. Para estos casos existe una solución que consiste en declarar una clase intermedia Cliente2 en la que se redefina el campo nombre del facet cuenta y a continuación se deriva EmpleadoCliente de Cuenta2 y de Empleado (figura 94). Como los nombres de los facets de cuenta son distintos no hay necesidad de redefinir dichos facets y todo irá bien.

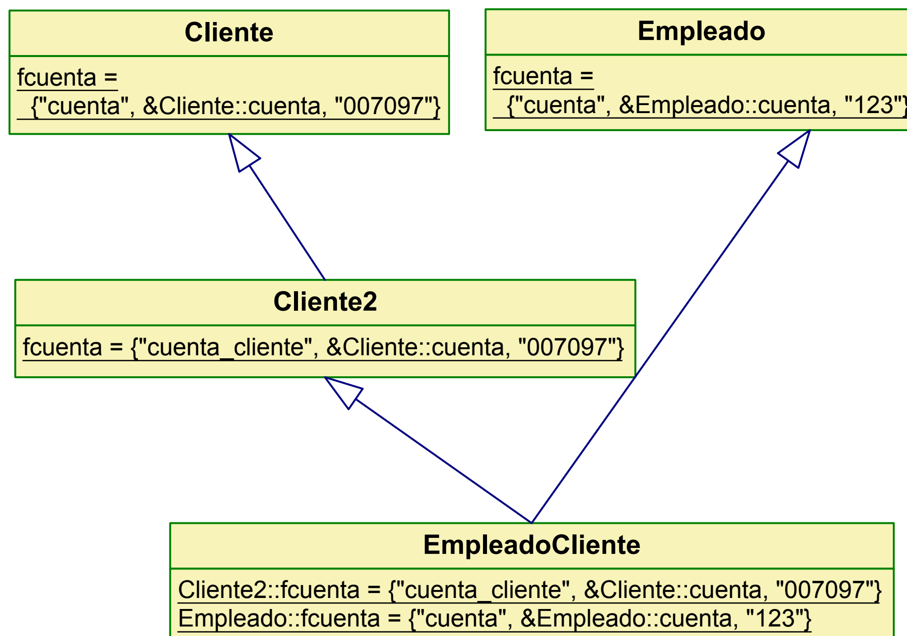


Figura 94. Renombrado de facets

2.6.6. Herencia de facets y herencia virtual

En la figura 94 se ha visto un ejemplo de herencia repetida. Un objeto EmpleadoCliente tiene dos copias de Imprimible, una heredada de Empleado y otra de Cliente. El modelo de herencia de C++ permite realizar herencia múltiple de modo que sólo se tenga una copia de la clase que aparece repetida (figura 95).

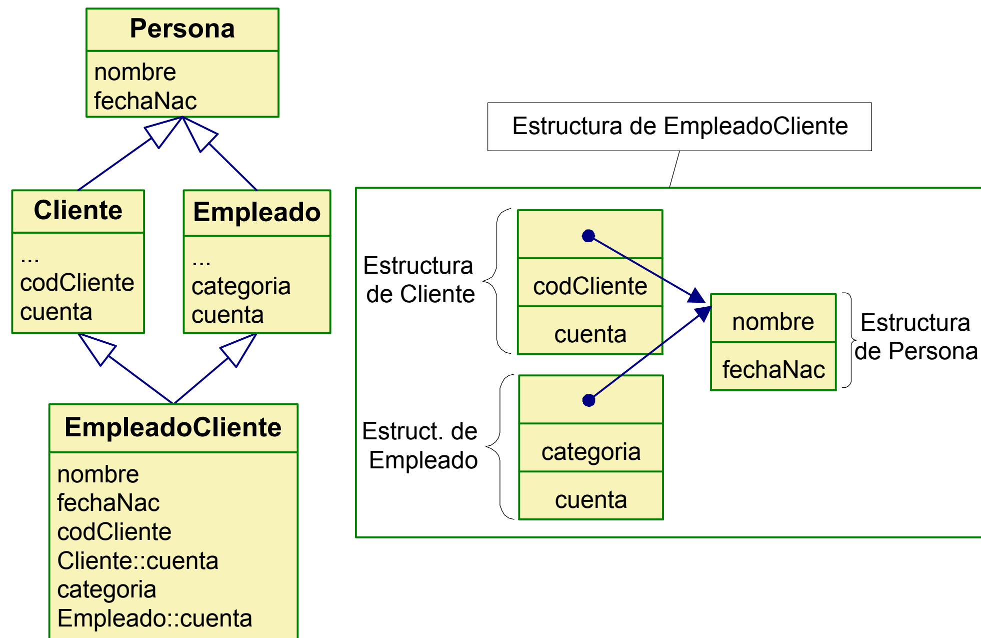


Figura 95. Herencia múltiple de clases del modelo

En este ejemplo, la clase EmpleadoCliente contiene únicamente una Persona, compartida por Cliente y Empleado. EmpleadoCliente tiene por tanto un sólo atributo nombre y un único facet fnombre. El algoritmo de la figura 91 recorre todas las clases padre y añade cada facet no redefinido de estas clases padre en la lista de facets de la clase derivada. Como fnombre no está redefinido en EmpleadoCliente resulta que en la lista de facets figurarían dos copias de fnombre. La solución es sencilla y práctica. Consiste en añadir un paso más al algoritmo de la figura 91 (paso 7) que recorrerá la lista (provisional) de facets y dejará sólo una copia de los facets que aparezcan multiplicados. El algoritmo funciona entonces bien en todos los casos:

- Si fnombre se redefine en EmpleadoCliente entonces al intentar añadir el fnombre de Cliente ocurre que el test del paso 6 es falso y no se añade el facet. (Esto ocurre se haya redefinido o no en Cliente el facet fnombre). Lo mismo pasa con el fnombre de Empleado. Correctamente sólo se añade el facet redefinido en EmpleadoCliente.
- Si fnombre no se redefine en ningún lugar entonces se añaden en principio los dos facets heredados pero el paso 7 eliminará uno de ellos pues apuntan al mismo atributo. Da lo mismo cuál de los dos se elimine pues coinciden.
- Si fnombre se redefine únicamente en Empleado entonces por la llamada *regla de la dominancia*, una llamada a vfnombre() en EmpleadoCliente devolverá el facet redefinido en Empleado. El facet de Cliente quedará oculto y por el paso 6 será eliminado. En la lista de facets sólo se incluirá el facet que se ha redefinido en Empleado.
- Si fnombre se redefine en Cliente y en Empleado se produce un error de compilación ya que no es posible construir la tabla de métodos virtuales pues si a una referencia p a Persona se le asigna un EmpleadoCliente entonces la llamada p.vfnombre() es ambigua.

Respecto del tema de la herencia virtual hay que hacer notar además que, en realidad, toda herencia múltiple que se haga producirá herencia virtual ya que las clases de las que se heredan heredarán a su vez de Object. Cuando aquí se ha hablado de herencia virtual se entiende que se refiere a herencia virtual en la que no aparece implicada la clase Object. Esta clase, por otro lado, no requiere ningún tipo de precauciones en el sentido que se acaba de exponer pues su lista de facets es vacía.

2.7. Conclusiones

En este capítulo se han establecido una serie de propiedades que debe cumplir un sistema de autorrepresentación en C++ de cara a su utilización en un sistema de información. Dentro del propósito general de este trabajo no se han planteado exigencias que fueran más allá de la filosofía general del lenguaje y que obligaran a utilizar herramientas sofisticadas como las vistas en el capítulo 1. Aún así, los objetivos enumerados al principio del capítulo son relativamen-

te ambiciosos, comprobándose las ventajas que se conseguirían de cara al desarrollo de ciertos tipos de proyectos de software. También se han analizado las dificultades que surgen al enfrentarse a estos objetivos utilizando únicamente los recursos básicos del lenguaje viéndose la necesidad de crear una infraestructura de apoyo.

El aumento de la sofisticación del lenguaje en temas como la *genericidad* mediante el uso de plantillas ha sido determinante a la hora de encontrar una aproximación. Durante la exposición de las características de la solución se ha ido comprobando cómo dicha solución satisface de forma completa los objetivos marcados al principio del capítulo. La arquitectura de la aproximación propuesta utiliza de forma generalizada las plantillas estándar de C++ además de emplear el sistema de macros del preprocesador para simplificar la sintaxis. De este modo se ha podido encontrar una solución dentro del lenguaje sin necesidad de utilizar un precompilador específico.

Además, esta librería permite realizar tareas de introspección sin salirse de la filosofía del lenguaje, es decir, llevando a tiempo de compilación muchas tareas que normalmente se realizan en tiempo de ejecución y preservando en gran medida la comprobación estática de tipos y la eficiencia.

Además de incluir de base un conjunto amplio de metainformación con muchos campos de aplicación, la librería es extensible y abierta permitiendo precisar qué metainformación se añade a los atributos mediante la adición de clases a la jerarquía inicial y también qué metainformación se añade a las clases mediante la creación de metaclases y familias de metaclases. Esto resulta muy útil en sistemas reflexivos ya que hay mucha variedad en cuanto al tipo de metainformación que se necesita en cada caso.

Las capacidades de ampliación de la librería se han diseñado de acuerdo a los principios de orientación a objetos. El usuario extiende las posibilidades del sistema mediante la introducción de clases y jerarquías de clases que se integran en el sistema sin afectar al resto de elementos. Además, pueden reutilizarse tanto para la realización de aplicaciones finales cuanto para añadir nuevos componentes a la librería.

La solución propuesta presenta el problema de que hace uso del sistema de macros del preprocesador de C++. Cierto es que desde un punto de vista técnico no se recurre a elementos ajenos a C++, y de este modo se evita una sobrecarga en la complejidad del sistema de desarrollo, pero también es verdad que las macros actúan en una fase previa al proceso de compilación. Por tanto, son

totalmente ajenas al lenguaje en sí sin poder interactuar con elementos del lenguaje como espacios de nombres, tipos de datos, etc. La escasa potencia del sistema de macros (por ejemplo no admitir un número variable de parámetros) ha conducido a una proliferación innecesaria del conjunto de macros del sistema. De hecho se han construido pequeños programas encargados de generar ficheros de macros para el caso de macros con distintos números de parámetros,

En cualquier caso, el uso de macros permite al usuario utilizar una notación clara y concisa siendo muy cómodo el uso de la arquitectura tanto para desarrollar aplicaciones como para implementar nuevos componentes. El sistema de macros no se usa de modo sofisticado y sutil, esto daría lugar con toda probabilidad a problemas de mantenimiento. Se trata únicamente de evitar la escritura de información redundante mediante un conjunto moderadamente pequeño de macros sencillas.

Capítulo aparte merecen las consecuencias derivadas del uso intensivo de plantillas. Hay que tener en cuenta que cada instanciación de una plantilla para unos parámetros concretos implica la creación por parte del compilador de una nueva clase. Para cada clase del modelo se crea siempre una metaclase asociada (`ClassInfo<T>`). Por cada tipo de facet y cada tipo de atributo se genera también una clase. Aunque varios facets pueden compartir clase no ocurre lo mismo en el caso de los atributos generándose una clase distinta para cada atributo.

La proliferación de plantillas y la explosión de clases consiguiente supone una carga importante para el compilador haciendo que los tiempos de compilación crezcan notablemente. Este problema se agudiza en las implementaciones actuales de los compiladores ya que en la mayoría de los casos no es posible realizar una verdadera precompilación de las plantillas. Cada vez que se llama al compilador debe generarse de nuevo el código de las plantillas que se instancien aunque no haya habido ninguna modificación en las mismas. El estándar del lenguaje incluye una característica para permitir la precompilación separada de plantillas mediante la palabra reservada `export` pero los compiladores actuales no incorporan todavía esa característica.

El usuario de la librería debe contar además con el hecho de que las implementaciones actuales plantean problemas en cuanto al uso de plantillas. No es inhabitual que los compiladores cometan errores en el tratamiento de las plantillas y además los mensajes generados suelen ser confusos. Es de esperar una

mejora en la realización de las implementaciones en este sentido teniendo en cuenta que el sistema de *genericidad* de C++ es bastante novedoso y hay que esperar algún tiempo hasta que la experiencia en la implementación de estas técnicas produzca mejoras sustanciales para el usuario de los compiladores.

El aumento de código generado suele ser otra consecuencia negativa del uso profuso de las plantillas. En la arquitectura de facets las plantillas que afectan de manera más importante a la eficiencia del sistema son las plantillas de atributos pues su uso es de un orden de magnitud superior al de las plantillas de facets y de clases. Afortunadamente, estas plantillas son sencillas constando de unos pocos métodos sencillos cuya misión suele ser proporcionar acceso encapsulado a otro elemento. Gracias a ello todos estos métodos se declaran como funciones de expansión sin tener que generarse código para ellos, salvo en el caso de los métodos virtuales. En definitiva, el crecimiento de código por el uso de la librería es moderadamente pequeño.

Las desventajas anteriores se ven compensadas por la comprobación estática de tipos que permite detectar muchos más errores en tiempo de compilación con la consiguiente disminución de los tiempos de desarrollo. Otra ventaja importante que se ha conseguido con el uso de las plantillas es la notable eficiencia del código generado que suele ser muy parecida a la que se obtendría generando el código a mano y muy superior a la de otros sistemas que retrasan las comprobaciones a tiempo de ejecución.

Capítulo 3

Extensión de la jerarquía de facets para trabajar con asociaciones

En Ingeniería del Software se establecen varias fases en el proceso de desarrollo de software. En las fases de análisis y diseño se define el alcance del problema y la estructura general de la solución. En la fase de implementación se escribe y comprueba el código. El diseño sirve de guía o patrón que hay que seguir para realizar la implementación. Tradicionalmente, este paso de la fase de diseño a la de implementación se ha realizado artesanalmente debido, entre otras razones, a que muchos conceptos que aparecen en la fase del diseño no tienen correspondencia directa en elementos del lenguaje empleado para realizar la implementación. Quizás el caso más claro sea el concepto de asociación entre dos clases, concepto que aparece como elemento de primer orden en los sistemas de desarrollo orientado a objeto como [Booch 99]. Pues bien, la mayoría de los lenguajes orientados a objetos no ofrecen soporte directo al concepto de asociación haciéndose necesario emplear algún patrón de implementación para representarlo. Normalmente, el desarrollo de estos patrones hace que el texto relativo a la implementación de una asociación se encuentre diseminado por el código del programa. Un cambio en una asociación obliga a retocar manualmente las zonas del código afectadas por dicho cambio. Si las asociaciones tuvieran soporte directo en el lenguaje se disminuiría notablemente tanto el esfuerzo para pasar del diseño original a la implementación, como para actualizar la implementación ante cambios en el diseño.

Este capítulo muestra cómo la tecnología de facets puede aplicarse para crear componentes que sirvan para plasmar de forma directa el concepto de asociación en el código fuente. En primer lugar se explica cómo los atributos enriquecidos se pueden utilizar para representar el acceso de un objeto a los objetos de la otra clase que están asociados con él. Para ello, en el facet del atributo se incluirá la información del rol o vista de la asociación. Una vez

presentado el mecanismo general de implementación de asociaciones se explica su desarrollo en el marco de la librería de facets. Ésta incluye clases y plantillas de clases para representar atributos de acceso a asociaciones, roles y asociaciones. En la librería se da soporte a varios tipos de asociaciones binarias según la cardinalidad y el tipo de almacenamiento de los objetos y si disponen o no de atributos, mostrándose el proceso de implementación en cada caso. Finalmente, y en consonancia con la filosofía abierta de la librería de facets, se dan las pautas para desarrollar componentes que permitan trabajar con otros tipos de asociaciones.

3.1. Aproximaciones para implementar asociaciones

Una asociación [Rumbaugh 99] establece una relación entre objetos de dos o más clases. El caso más habitual ocurre cuando son dos las clases participantes. Estas asociaciones reciben el nombre de binarias y en lo sucesivo el término asociación hará referencia a este tipo de asociaciones (al final de la memoria se habla de posibles implementaciones de asociaciones de orden superior). La información de los elementos de una asociación es un subconjunto del producto cartesiano de las clases participantes. La figura 96 muestra un ejemplo de una asociación entre dos clases Oficina y Empleado llamada destino en la que cada par o enlace está formado por un empleado y la oficina en la que está destinado.

Oficinas	Empleados
zaragoza1	artal
zaragoza1	borau
norte	urruti

Figura 96. Ejemplo de asociación entre clases

Una asociación introduce indirectamente un seudoatributo nuevo en las clases participantes. Por ejemplo, para la clase Oficina se puede considerar que existe un atributo plantilla que, aplicado a una oficina concreta, devuelve la lista de empleados con destino en esa oficina. Se usará el término rol para referirse a esta vista de la asociación desde una de las clases participantes. Un rol tiene una serie de propiedades [Rumbaugh 99, pp. 160-61] como el nombre, la visibilidad y la multiplicidad. El rol es visible si se considera al seudoatributo de la clase como público, es decir, si dado un objeto se puede preguntar a la clase por sus objetos relacionados. La multiplicidad es la restricción en cuanto al número de objetos con los que puede relacionarse un objeto dado. Por ejemplo, la multiplicidad de trabajaEn es 1 pues un empleado trabaja en una única oficina. La multiplicidad de plantilla es 1..* (uno como mínimo y sin límite máximo). Si en uno de los roles hay multiplicidad máxima 1 y en el otro no hay restricción hablaremos de un asociación 1 a n o 1 a varios utilizando la misma terminología que en el modelo relacional de bases de datos [Date 95]. Del mismo modo se definen las asociaciones 1 a 1 y n a n.

A continuación se estudian algunas formas típicas de implementación.

3.1.1. Utilización de punteros en las clases participantes

Esta es la aproximación más sencilla y es la utilizada como ejemplo en varios libros sobre modelado como [Lee 97]. Cada objeto de la clase tiene un campo en el que se guardan los objetos relacionados con él (figura 97).

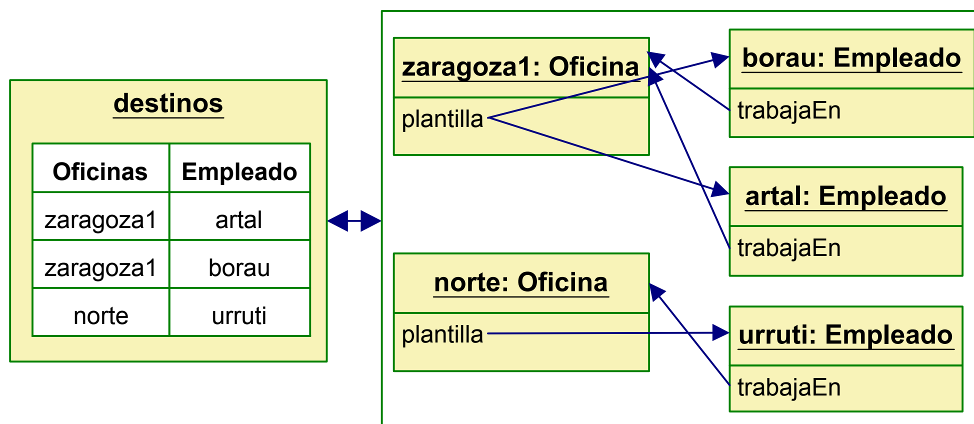


Figura 97. Almacenamiento de la información en las clases

Si la multiplicidad no es mayor que 1 basta con incluir en la clase un miembro que sea un puntero a un objeto de la otra clase. Si es n habrá que guardar un vector de punteros:

```
class Oficina {
    ...
    Vector<Empleado*> plantilla;
};
class Empleado {
    ...
    Oficina* trabajaEn;
};
```

Figura 98. Declaración de la información en las clases participantes

Esta representación tiene la ventaja de que proporciona acceso directo a los objetos asociados con uno dado. Si no se desea proporcionar acceso a los datos desde uno de los lados basta con omitir la declaración del miembro correspondiente. En el caso de permitir visibilidad desde ambos roles hay que tener en cuenta que, cuando se modifiquen los datos desde uno de los roles, habrá que actualizar convenientemente los datos de la otra vista tal y como se puede ver en el ejemplo de la figura 99.

```
void Oficina::insertaEmpleado(const Empleado& nuevoEmpleado) {
    plantilla.push_back(&nuevoEmpleado);
    nuevoEmplado.trabajaEn = this; //Ajuste del rol del empleado
}
```

Figura 99. Ajuste de la información de un rol al variar el otro

El mismo sistema habría que emplear para implementar el resto de métodos típicos como realizar una baja o un cambio.

El sistema que se acaba de ver plantea serios problemas como consecuencia de la discordancia entre los conceptos lógicos que desea expresar el programador (roles, asociaciones) y los elementos empleados en la implementación

(campos en las clases). En la implementación no figura la correspondencia lógica entre los atributos plantilla y trabajaEn, y es que ambos son las vías de acceso a la información de una asociación que no aparece plasmada en el código, sólo en la mente del programador. Por este motivo aparecen problemas como los siguientes:

- *Duplicación de código*. El algoritmo de métodos como insertaEmpleado es el mismo en todas las asociaciones 1 a n en las que se quiera permitir la adición de enlaces. Lo mismo ocurre con otros métodos que se deseen facilitar, como bajas o búsquedas y con otras multiplicidades de asociaciones. La adición de asociaciones implica la reescritura de estos códigos y la modificación de un método obliga a realizar el cambio en todas las asociaciones.
- *No separación de conceptos*. Los elementos relativos a una asociación se encuentran dispersos en el código. Si, por ejemplo, se desea eliminar una asociación hay que eliminar manualmente los campos respectivos en las dos clases participantes así como todos los métodos relacionados.
- *Falta de parametrización de las propiedades de una asociación*. Una modificación en el diseño tan elemental como cambiar la multiplicidad de una asociación provoca importantes cambios en el código. Del mismo modo, si se desea eliminar la visibilidad a un rol entonces es necesario modificar los métodos de acceso del otro rol para que no hagan cambios en el rol eliminado. Lo mismo ocurre si se desea un cambio de representación en memoria de los enlaces.

3.1.2. Implementación mediante herencia de una clase asociación

En [Papurt 95] una clase se asocia con otra mediante herencia de una clase asociación. La figura 100[7] muestra el código de una plantilla asociación 1 a 1 entre dos clases A y B. Las clases participantes deben heredar ambas de esta plantilla asociación. El método linkTo sirve para realizar un enlace entre las dos clases. Este método establece el enlace directo con setPtr(b) y además envía un mensaje a la otra clase participante para que establezca el enlace inverso.

[7] El código que aquí se muestra es una simplificación del original conseguida gracias al hecho de que una clase puede derivar de una plantilla en la que aparece la propia clase.

```
template<class A, class B>
class Association {
 B* ptr;
 public:
     void linkTo(B* b) { setPtr(b); b->setPtr((A*)this); }
     void setPtr(B* b) { ptr = b; }
     B* getAssociatedObject() { return ptr; }
};
class Oficina: public Association<Oficina, Empleado> {
 ...
};
class Empleado: public Association<Empleado, Oficina> {
 ...
};
int main() {
     Oficina* of; Empleado* director;
     of->linkTo(director);
     //Automáticamente se hace el enlace inverso director->setPtr(of)
}
```

Figura 100. Implementación de asociaciones mediante herencia

El inconveniente principal de esta aproximación es que una clase no puede participar en más de una asociación. Si se planteara esta situación se podría pensar en utilizar herencia múltiple. En ese caso la clase tendría varios métodos linkTo y se deberían usar nombres cualificados de métodos para romper la ambigüedad (figura 101).

En cualquier caso no se podrían tener dos asociaciones diferentes con una misma clase ya que no se puede derivar dos veces de una misma clase. [Papurt 95b, cap. 15.5] proporciona una solución al problema de asociaciones múltiples que tiene el inconveniente de tener que generar una clase nueva por cada asociación que se añada. Así, en el caso de Oficina habría que declarar tres clases: OficinaBase (sin asociaciones), OficinaConDirector (hereda de OficinaBase e

implementa la asociación con Empleado) y Oficina (hereda de OficinaConDirector e implementa la asociación restante).

```
class Oficina: public Association<Oficina, Empleado>,
     public Association<Oficina, Ciudad> {
 public:
     typedef Association<Oficina, Empleado> Director;
     typedef Association<Oficina, Ciudad> Ubicacion;
 ...
};
int main() {
     Oficina* of; Empleado* director; Ciudad* ciudad;
     of->Director::linkTo(director);
     of->Ubicacion::linkTo(ciudad);
}
```

Figura 101. Clase con varias asociaciones

3.1.3. Implementación por medio de campos asociación

[Linenbach 96] utiliza también clases asociación, pero en lugar de heredar de ellas las emplea para declarar campos en las clase participantes. El esquema de su implementación puede verse en la figura 102.

El siguiente código muestra la forma de crear un enlace y de consultarlo.

```
Oficina zaragoza;
Director peláez;
zaragoza.director.attach(peláez.oficina);
Director d = zaragoza.director.getRhsObject();
```

Figura 102. Uso de clases asociación como atributos

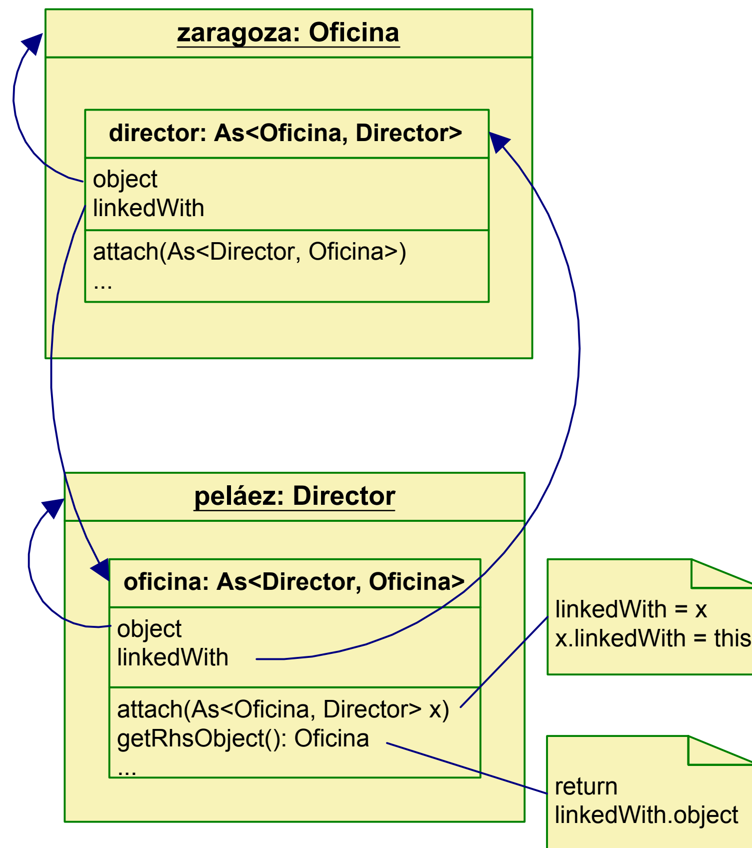


Figura 103. Clases asociación utilizadas como atributos

Con este tipo de implementación no aparece el problema de utilizar varias asociaciones dentro de una misma clase. Hay, sin embargo, algunos inconvenientes:

- Uso de una notación inadecuada. Al crear un enlace se indica el subobjeto asociación con el que se enlaza escribiéndose

  zaragoza.director.attach(peláez.oficina)

  en vez de

  zaragoza.director.attach(peláez)

  Éste es un detalle de implementación que debería quedar oculto. Además, deja abierta la posibilidad de cometer errores si se realiza un enlace con un subobjeto asociación diferente.

- Cada subobjeto asociación debe mantener una referencia a su objeto (el campo object de la figura 103). De nuevo, es una redundancia exigida por la implementación. El coste puede ser significativo puesto que ni siquiera se trata de un campo estático: cada objeto tendrá este campo redundante. La iniciación de esta referencia debe hacerse en el constructor de la clase principal lo que contraviene el principio de independencia de las asociaciones (un cambio de implementación obligará a retocar todas las clases).
- Las asociaciones no son elementos de primer orden. No existen clases que representen asociaciones. Para cada asociación hay dos objetos As que son meros punteros dobles para actualizar el otro lado de la asociación.

3.1.4. Implementación interna del lenguaje

Existen propuestas de diseño de lenguajes que admiten directamente las asociaciones como elementos de primer orden. En DSM [Rumbaugh 87] [Shah 89] existen primitivas para declarar asociaciones en las que se indican el nombre de la asociación y de sus dos roles, las clases participantes y la multiplicidad (figura 104).

```
DEFINE destino RELATION
    trabajaEn: Oficina 1-* plantilla: Empleado
```

Figura 104. Primitivas en DSM para declarar asociaciones

Una vez definida una asociación el compilador automáticamente genera métodos en las clases participantes para acceder a ella. Por ejemplo, en la clase Oficina se crea el método get_plantilla para obtener la lista de empleados de la oficina a la que se envíe el mensaje. Naturalmente, la implementación interna de las asociaciones las convierte en una herramienta poco flexible que impide realizar adaptaciones o ampliaciones. Las tuplas de la asociación se guardan en el propio objeto asociación y no en los objetos de las clases participantes. Esto disminuye la eficiencia ya que para obtener los elementos asociados a uno dado hay que consultar una tabla. Para disminuir la pérdida de eficiencia, los datos se almacenan mediante dos tablas de dispersión, una para cada clase par-

ticipante. Uno de los coautores del lenguaje estima en un factor de 10 la disminución de la eficiencia ocasionada por la implementación centralizada en vez de usar punteros en las clases [Rumbaugh 96].

[Peralta 98] presenta otro ejemplo de implementación directa por parte del lenguaje mediante la creación de un lenguaje que extiende a Eiffel.

Hay que citar también las herramientas de transformación automática del diseño a la implementación como *Rational Rose*. Esta herramienta genera punteros de acceso a los elementos asociados e incluye automáticamente métodos para añadir y borrar instancias que se ocupan de actualizar convenientemente el otro lado.

3.2. Implementación de asociaciones mediante la librería de facets

Para desarrollar una infraestructura que consiga que las asociaciones y los atributos de acceso a ella sean elementos de primer orden vamos a seguir la fructífera idea "*si en tu diseño hay un concepto haz de él una clase*". Por ello, la librería de facets dispone de clases para representar asociaciones que centralizan todas las operaciones relacionadas con ellas. Por ejemplo, la asociación será la que se encargue de realizar las altas y bajas de enlaces y también se ocupará de obtener los objetos asociados a uno dado. También tiene clases para representar los roles donde se almacene por ejemplo la multiplicidad, el nombre y el acceso a la asociación correspondiente. Por último se dispone de clases especiales que representan los atributos que permiten acceder a la asociación desde las clases participantes. Se estudiará a continuación el diseño de todas estas clases.

3.2.1. Uso de atributos enriquecidos para trabajar con asociaciones

Se ha visto el problema que surge al utilizar punteros y contenedores de punteros para implementar atributos de acceso a una asociación. El problema surge de la falta de información adicional de estos atributos; por ejemplo en el atributo no hay información de la multiplicidad o de cuál es el atributo correspondiente en la clase compañera. Es decir, en el atributo no se guardan los

datos de la asociación concreta que representa ni los datos del lado de la asociación al que accede. A esta información formada por la asociación más el lado desde el que se la ve se la va a llamar *rol* del atributo.

El rol es común a todos los objetos de la clase. Por eso, la tecnología de atributos enriquecidos resulta muy apropiada para implementar esta clase de atributos. Un atributo de este tipo almacenará una información local que posiblemente incluirá los objetos relacionados con él y una información estática por medio de un rol que será el facet con los datos comunes a todas las instancias (figura 105).

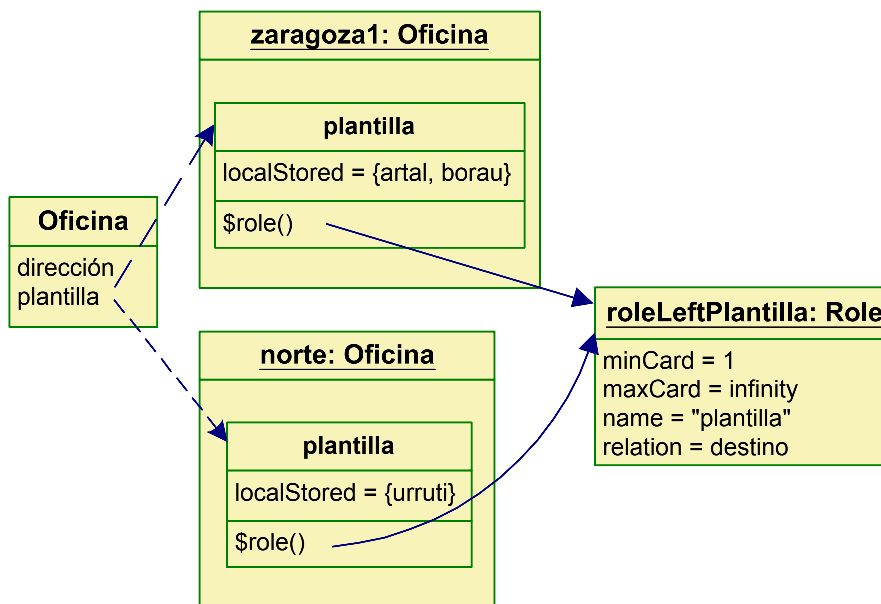


Figura 105. Los roles como información estática de los atributos de acceso

Después se verá cómo se organiza la información de los roles y las asociaciones. A los atributos que acceden a las asociaciones se les llamará *atributos de acceso* (el término atributos de asociación podría llevar a confusión con los atributos que pueden tener las asociaciones).

Los atributos de acceso a asociaciones se pueden integrar por tanto en la jerarquía de clases de atributos de la librería de facets asignando al rol el papel del facet del atributo. La figura 106 muestra la jerarquía de las clases de este tipo de atributos.

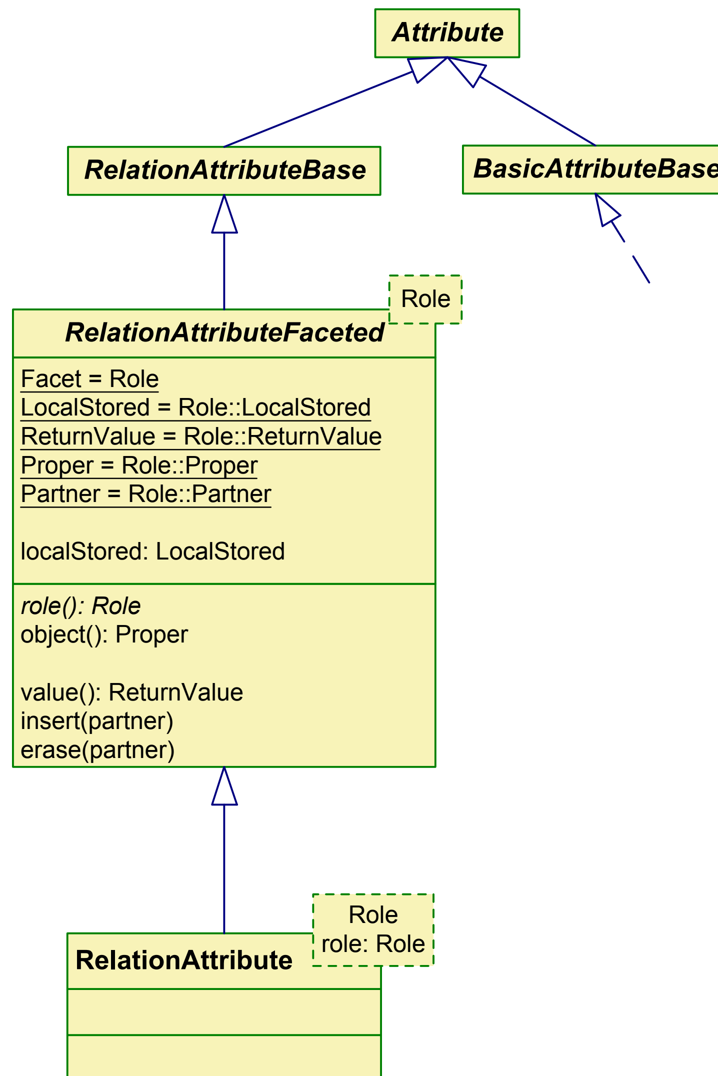


Figura 106. Jerarquía de atributos de acceso a asociaciones

La plantilla RelationAttribute es la que se usa a la hora de definir atributos de acceso. Está parametrizada por el rol al que accede el atributo. RelationAttributeFaceted se introduce para abstraer la información común de todas las clases de RelationAttribute que comparten el tipo del rol. En ella se implementan todos los métodos que no necesitan hacer uso del rol concreto. Por último, todas las clases de atributos de acceso tienen un ancestro común en RelationAttributeBase que está en el mismo nivel que BasicAttributeBase heredando de Attribute, la clase raíz de todos los atributos de la librería.

Los atributos de acceso, como su propio nombre indica, sirven para acceder a los elementos asociados a un objeto dado. Por sí mismos ellos no implementan ninguna función de la asociación y por eso delegan todas las operaciones en el objeto rol que hace de facet. La información se centraliza por tanto en los roles, y los atributos de acceso no necesitan tener ningún conocimiento de la implementación ni de las características de los roles y de las asociaciones. Esta independencia va a permitir modificar las características de las asociaciones sin que esto afecte a los atributos de acceso. Al estar toda la información centralizada en los roles y las asociaciones en vez de estar dispersa por las clases participantes en la asociación, si se desean modificar las características de éstos, el cambio habrá que realizarlo en un único lugar.

Para obtener los objetos asociados a uno dado se dispone del método value(). Es un método equivalente al que figura para los atributos básicos, sólo que en este caso el valor devuelto puede ser una lista de objetos. Para modificar esta lista se dispone de los métodos insert(partner) y erase(partner). Como se ha dicho, todos estos métodos delegan su acción al rol. Por ejemplo insert(partner) se implementa como role().insert(object(), partner). Nótese que al método insert del rol se le deben pasar los dos objetos ya que el rol es un objeto estático. El objeto partner está disponible porque se lo han pasado como parámetro al insert del atributo, sin embargo el objeto propio no está en principio disponible ya que el método pertenece a la clase del atributo. A partir del atributo this se puede conseguir el objeto que lo contiene mediante el método object(). ¿Cómo puede conocer un atributo la dirección en la que comienza el objeto que lo contiene?. La solución consiste en delegar esta tarea al rol con la llamada role().object(this). Ahora el rol tiene en su poder el atributo y además tiene guardado el *offset* del atributo. Basta aplicar el *offset* a la inversa para obtener la dirección de comienzo del objeto (figura 107).

Las capacidades de las plantillas como funciones que trabajan con tipos como parámetros permiten que las clases involucradas en los atributos de acceso se obtengan igualmente por delegación en el rol, permitiendo reducir al mínimo la información que se incluye en la propia clase. Así, en la clase del atributo de acceso se necesita conocer cuál es la clase compañera, por ejemplo para incluir el método insert(partner). El “valor” de esta clase Partner se obtiene a partir de la clase del rol mediante Role::Partner. Lo mismo ocurre con la clase ReturnValue que representa el tipo de datos al que pertenece el objeto o lista de objetos asociados a uno dado (el devuelto por value()).

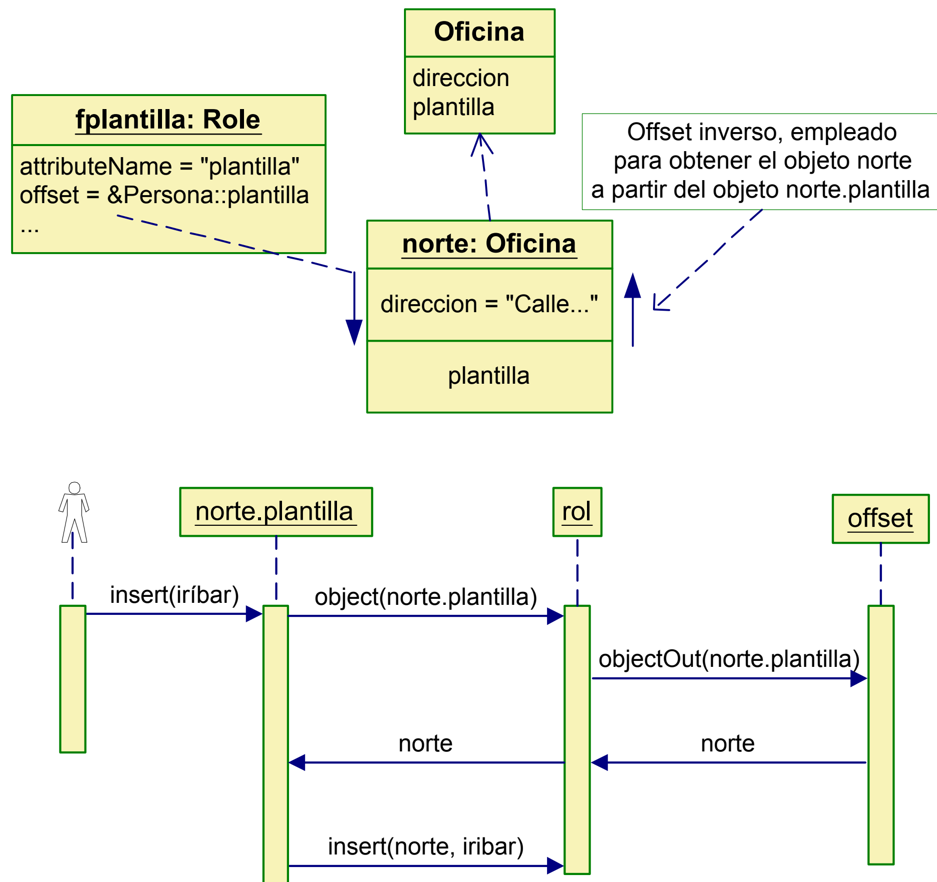


Figura 107. Con el offset *inverso se accede al objeto que contiene al atributo*

Siguiendo esta misma línea se consigue delegar en el rol el tipo de datos empleado para guardar localmente los objetos asociados a uno dado. Esta clase se llama LocalStored. Su valor es igual a Role::LocalStored. El rol es el que determina por tanto el tipo de datos local y esto permite llevar a cabo diferentes políticas de almacenamiento de la información de los enlaces de la asociación. En una implementación típica, Role::LocalStored podría ser Partner* o Vector<Partner*>, sin embargo un tipo de asociación puede almacenar la información de los enlaces en otro lugar, por ejemplo en una tabla de base de datos. En este caso no se necesita guardar nada en el LocalStored y se definirá como Void, una clase vacía. LocalStored es un campo que se incluye en los atributos de acceso para brindar la posibilidad a las asociaciones de guardar la información localmente en los objetos.

3.2.2. Roles

En el apartado anterior se ha visto que los atributos de acceso tienen un facet asociado llamado rol que permite obtener la información de la asociación y del lado desde el que se accede. Por cada asociación habrá dos objetos rol, izquierdo y derecho, que tendrán una información propia más un puntero o referencia a la asociación común a la que pertenecen (figura 108).

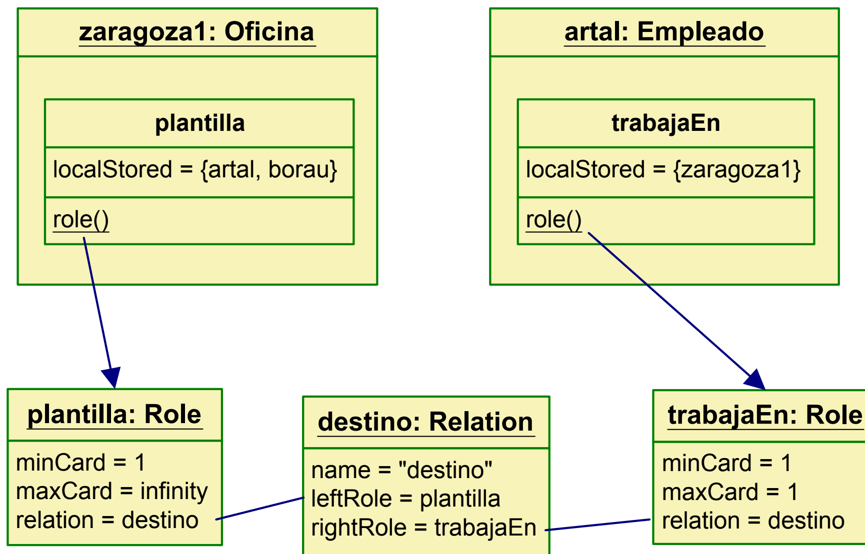


Figura 108. Los dos roles comparten la información de la asociación[8]

Todo el código de gestión de las asociaciones debe incluirse lógicamente en el objeto asociación siendo los roles únicamente vías de acceso a esta asociación. La implementación de todos los servicios del rol consistirá en delegar ese servicio a su asociación. De este modo, cuando se añada una nueva clase de asociación no habrá que crear nuevas clases de rol. Bastará por tanto con tener dos clases de role (LeftRole y RightRole) que servirán para todas las asociaciones que se creen.

Un rol oferta esencialmente tres tipos de servicios. Veamos cómo se puede conseguir implementar estos servicios que valga para cualquier tipo de asociación:

[8] En la librería de facets se usa el término Relation para identificar las asociaciones.

- Servicios para dar información particular del rol desde el que se accede a la asociación. Por ejemplo la cardinalidad mínima. Estos datos son propios del rol y se guardan localmente en él de modo que no necesita consultar a la asociación para su implementación.

- Servicios de manejo de los enlaces de un objeto (inserción, borrado). Cuando un atributo de acceso requiera un servicio del rol éste lo delegará a su vez en la asociación correspondiente. Así, si el atributo requiere hacer una inserción de un nuevo enlace se lo pedirá al rol el cual inmediatamente se lo requerirá a la asociación (figura 109).

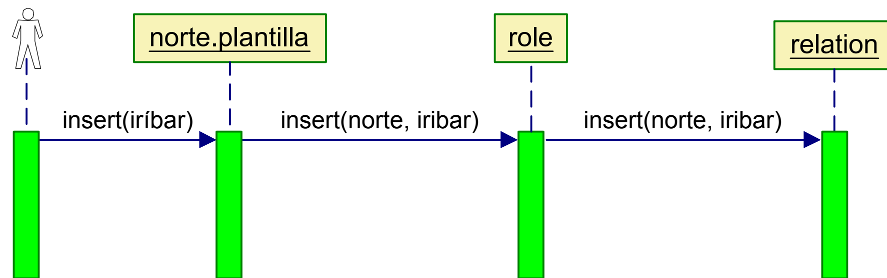


Figura 109. Delegación sucesiva de petición de servicios

- Valor de ciertos tipos de datos. Los roles deben guardar valores de ciertos tipos para poder ofertárselos al atributo de acceso ya que éste los necesita, por ejemplo, para saber el tipo de dato que va a guardar localmente (LocalStored). Estos valores dependen de la asociación y el rol deberá solicitárselos. Como se trata de realizar operaciones con tipos la solución consiste en parametrizar las clases de rol por el tipo de la asociación. La figura 110 muestra un ejemplo de una clase asociación que oferta las clases LeftLocalStored y RightLocalStored como los tipos que deben usar las clases Empleado y Cliente para almacenamiento local.

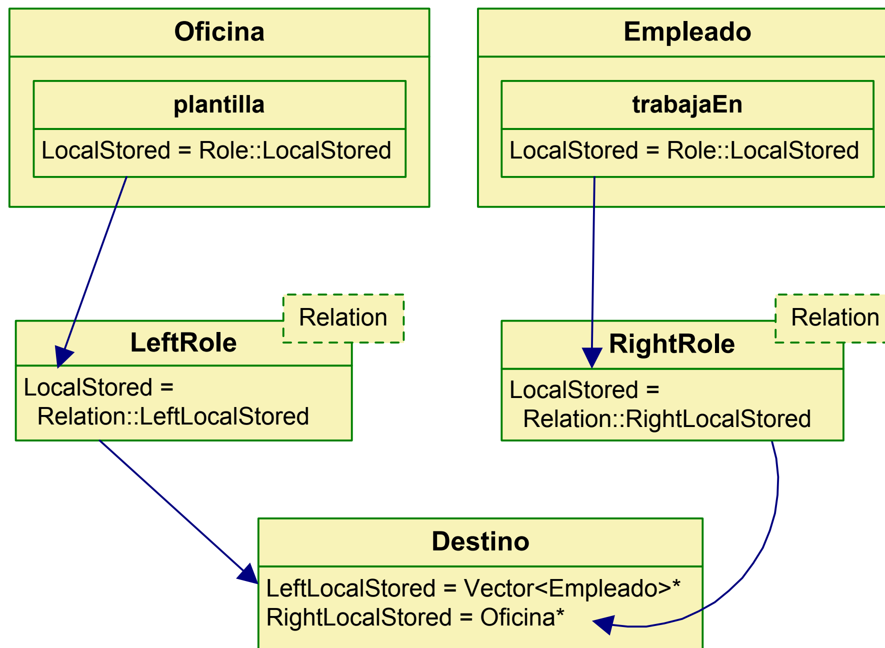


Figura 110. La asociación proporciona el valor LocalStored

Al no tener el rol información interna acerca del comportamiento de la asociación se consigue hacerlo independiente de ésta y por tanto no se necesitarán crear clases de rol nuevas cuando se añada un nuevo tipo de asociación al sistema. Únicamente se necesitarán dos clases de rol LeftRole y RightRole parametrizadas por el tipo de la asociación.

LeftRole y RightRole son plantillas con una gran cantidad de elementos comunes. Podría pensarse en colocar esta información común en una clase padre. Sin embargo, las diferencias entre LeftRole y RightRole residen principalmente en los tipos empleados. Por ejemplo, LocalStored se define en LeftRole como Relation::LeftLocalStored y en RightRole como Relation::RightLocalStored. Lo mismo ocurre con el resto de tipos, y la mayoría de los servicios de estas clases de rol toman parámetros o devuelven valores de estos tipos. Los servicios tienen signaturas diferentes y por tanto no se puede aplicar el mecanismo de la herencia. Como se está trabajando con tipos, las plantillas van a permitir abstraer estas propiedades comunes. La clase padre común de LeftRole y RightRole va a estar parametrizada (además de por el tipo de la asociación) por el tipo de acceso izquierdo o derecho.

Dentro de Role hay que definir LocalStored como Relation::LeftLocalStored si el lado es el izquierdo y como RightLocalStored si el lado es el derecho. La solución es muy sencilla recurriendo a la programación en tiempo de compilación y se muestra en la figura 111. Se trata de una condicional simple y se puede emplear la metainstrucción IF vista en la figura 34.

```
enum Sides {leftSide, rightSide};

template<class Relation, Side side>
class Role {
    typedef IF<side == leftSide,
        Relation::LeftLocalStored,
        Relation::RightLocalStored>::VAL LocalStored;
    ...
};

template<class Relation>
class LeftRole : public Role<Relation, leftSide> {
    LeftRole(...) : ... {}
}
```

Figura 111. Fusión de los dos roles en uno mediante metaprogramación

La figura 112 muestra la estructura de las clases de rol. Todas las clases de rol que se generan con la plantilla Role heredan de RoleBase, clase en la que se guarda la información que no depende de la asociación ni del lado desde el que se ve. RoleBase hereda de FacetBase para poder incluirlo en la lista de facets de la clase.

Las clases de Role tienen un puntero de acceso a la asociación y un *offset* o puntero a miembro para acceder al atributo de acceso con el que está relacionado (este último lo hereda de FacetBase).

Los typedefs Proper y Partner se refieren a la clases participantes en la asociación. Proper es la clase del lado de la vista, es decir la clase izquierda en el caso de leftSide y la derecha en el caso de rightSide. Partner es la clase del otro lado de la vista.

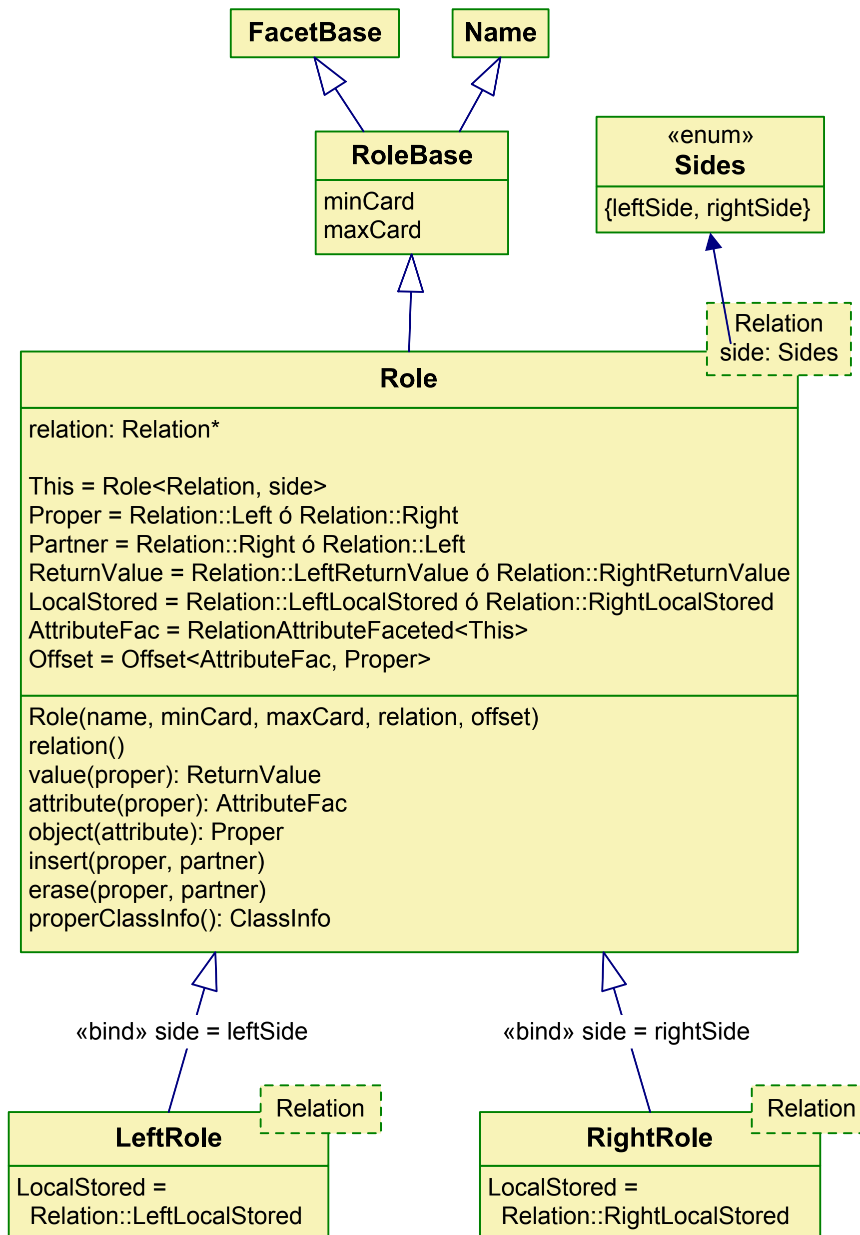


Figura 112. Jerarquía de las clases de rol

ReturnValue es el tipo de valor devuelto cuando se accede al rol para pedir los elementos asociados con un objeto.

LocalStored es el tipo que se almacena en el atributo de la clase Proper para guardar el valor de los elementos relacionados con uno dado.

Para obtener los objetos asociados a uno dado se usa value(proper).

Para insertar o borrar una pareja de datos en la asociación se usan insert(proper, partner) y erase(proper, partner).

La implementación de estos métodos es bastante sencilla, sólo hay que tener en cuenta el lado desde el que se accede a la asociación. Por ejemplo, en el caso del método insert hay que hacer una llamada a relation().insert(proper, partner) en el caso de LeftRole y a relation().insert(partner, proper) en el caso de un rightRole ya que el método insert de la asociación espera que el primer parámetro sea de la clase izquierda y el segundo de la clase derecha (figura 113).

```
template<class Relation>
void Role<Relation, leftSide>::insert(
          const Relation::Left& proper,
          const Relation::Right& partner) {
     relation().insert(proper, partner);
};
template<class Relation>
void Role<Relation, rightSide>::insert(
          const Relation::Right& proper,
          const Relation::Left& partner) {
     relation().insert(partner, proper);
};
```

Figura 113. Especialización del método insert para los dos tipos de rol

3.2.3. Asociaciones

En los apartados anteriores se ha visto cómo toda la gestión de la información de las asociaciones se centraliza en las asociaciones mismas siendo las clases de atributos de acceso y los roles izquierdo y derecho meras vías de acceso a esta información. Se puede pensar en muchas políticas diferentes de

implementación de asociaciones según la multiplicidad, el modo de guardar los datos (en los objetos o en la propia asociación) o según la ubicación de estos en memoria o en un soporte persistente. Gracias a la centralización de información en las asociaciones, para integrar un nuevo tipo de asociación en la librería de facets bastará con crear la clase asociación correspondiente. Los atributos de acceso y los roles se parametrizarán por esta nueva clase de asociación delegando los servicios en ella. Una clase asociación deberá cumplir una serie de requisitos para poder integrarse en la arquitectura de la librería de facets. Estos requisitos son de dos tipos, los referentes a la existencia de ciertos métodos con una signatura dada y los que se refieren a la definición de ciertos tipos (typedefs) dentro de la clase de asociación.

Sea R la clase de asociación que se desea integrar en la librería.

- *Requisitos de definición de tipos internos*:
  - R::LeftReturnValue. Es el tipo del dato que se devuelve cuando se pide la instancia o instancias asociadas a un objeto del lado izquierdo de la asociación
  - R::RightReturnValue. Tipo de dato devuelto para un objeto del lado derecho de la asociación
  - R::Left. El tipo de la clase izquierda de la asociación.
  - R::Right. El tipo de la clase derecha de la asociación.
  - R::LeftLocalStored. Como ya se ha visto es el dato que se usa para informar a los atributos de acceso de cuál es la clase del campo donde almacenar los objetos relacionados con uno dado.
  - R::RightLocalStored. Lo mismo para el caso derecho.
- *Requisitos de signaturas de métodos*:
  - insert(left, right). Inserta la pareja correspondiente en la asociación. left y right deben ser de tipo Left y Right o de tipos con conversión implícita a Left y Right.
  - erase(left, right). Elimina la pareja correspondiente de la asociación.
  - getLeft(left). Devuelve los objetos asociados a left.

- getRight(right). Devuelve los objetos asociados a right.
- printLeft(left, os). Vuelca en el ostream os una representación textual de los objetos asociados a left.
- printRight(right, os). Ídem para un objeto right.

Cualquier asociación que cumpla los requisitos indicados en el apartado anterior puede usarse para instanciar LeftRole y RightRole. Por tanto, dada una asociación, las clases que representan sus vistas y sus atributos de acceso quedan automáticamente establecidas por la arquitectura.

En casi todos los casos, las clases de asociación que se creen irán parametrizadas por las clases izquierda y derecha para permitir que dicha asociación pueda establecerse entre dos clases cualesquiera.

La existencia de requisitos de tipos internos hace que no sea posible simplificar la caracterización de la validez de una clase asociación mediante herencia de una clase con ciertos métodos virtuales. Además, en los requisitos de métodos hay una relajación en cuanto a la exigencia de tipos, que proporciona una libertad mayor a la hora de diseñar una clase asociación (no se exige que left sea de tipo Left sino simplemente convertible a Left). Debido a ello no es posible determinar la validez de una clase asociación simplemente por la exigencia de que herede de una cierta clase con esos métodos virtuales. No obstante, la construcción de la librería por medio de plantillas hace que se detecte cualquier omisión o error en los requisitos antes de la ejecución del programa (en tiempo de compilación o de enlace).

La librería de facets incluye varias plantillas útiles de clases asociación que cubren un amplio rango de necesidades. La naturaleza extensible de la librería permite al usuario añadir nuevas clases asociación según sus necesidades. A continuación se describen algunas de estas clases, correspondientes a asociaciones en memoria 1 a 1, 1 a n y n a n con y sin atributos.

3.3. Asociaciones en memoria

En la librería hay definidas 3 plantillas de clases para trabajar con asociaciones que almacenan los datos en memoria. Se han creado 3 plantillas diferentes para dar soporte a diferentes tipos de cardinalidad. Se parametrizan por las

clases izquierda y derecha de la asociación. En el caso de asociaciones que permiten cardinalidades mayores que uno la plantilla asociación tiene un tercer parámetro que es la plantilla que va a hacer las veces de contenedor. El valor por defecto de este contenedor es la plantilla Vector, un contenedor propio de la librería que se explica en el capítulo 4.

- Plantilla para asociaciones 1 a 1:
  ```
  Relation11np<Left, Right>
  ```
- Plantilla para asociaciones 1 a n:
  ```
  Relation1Nnp<Left, Right, template<class> class V = Vector>
  ```
- Plantilla para asociaciones n a n:
  ```
  RelationNNnp<Left, Right, template<class> class V = Vector>
  ```

Por ejemplo, para crear una clase asociación entre Oficina y Empleado utilizando el vector de la librería de facets la plantilla se instancia así:

```
Relation1Nnp<Oficina, Empleado>
```

Si lo que se desea es usar el vector de la librería estándar de C++ la instanciación sería:

```
Relation1Nnp<Oficina, Empleado, std::vector>
```

Obsérvese cómo el estándar C++ permite que el parámetro de una plantilla sea a su vez una plantilla, en vez de una clase.

La jerarquía de las clases de asociación en memoria se puede ver en la figura 114. Estas clases distribuyen las instancias de la asociación en las clases participantes. Por ello definen LeftLocalStored y RightLocalStored adecuadamente como un puntero o un contenedor según la cardinalidad del rol.

Las asociaciones 1 a 1 cubren todos los casos en los que la cardinalidad máxima de cada rol sea a lo sumo 1 (por ejemplo una asociación 0..1 1..1). La implementación de este tipo de asociaciones es sencilla: LeftLocalStored y RightLocalStored (es decir, el tipo de datos que se guarda en el atributo) son simplemente un puntero a un objeto de la otra clase. El puntero será nulo si no hay objeto asociado. Hay que hacer notar que en el caso de que la cardinalidad mínima sea 1 esta restricción se aplicará una vez que se asigne un compañero a la instancia pues en el momento de crearse la asociación ésta no tendrá instancias, la cardinalidad de cada objeto será 0 y se estaría violando la restricción de

cardinalidad. En resumen, cardinalidad mínima 1 significa que la cardinalidad mínima es 0 hasta el momento en que pase a ser 1.

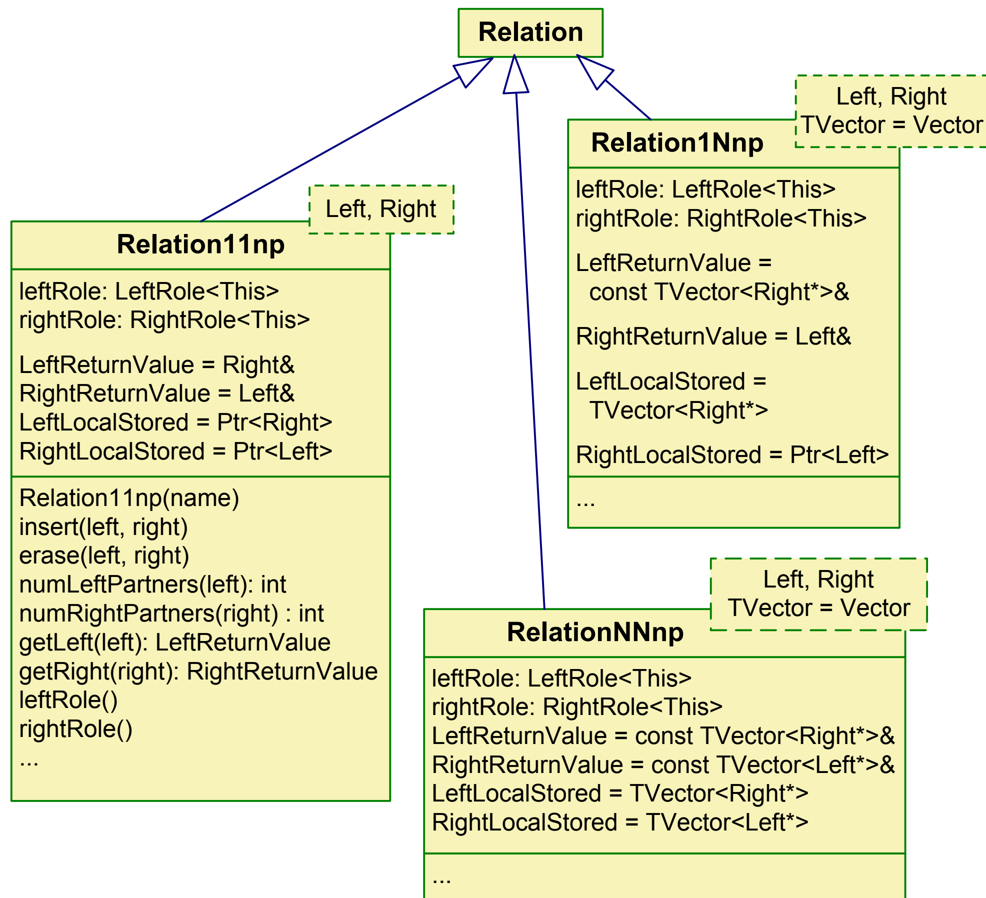


Figura 114. Jerarquía de asociaciones en memoria

En las asociaciones 1 a n, RightLocalStored sigue siendo un puntero a la clase izquierda siguiéndose el mismo mecanismo que en el caso anterior. En cambio, LeftLocalStored es un vector de punteros a elementos del lado izquierdo. El tipo devuelto LeftReturnValue se establece como una referencia constante a ese tipo de vector. De este modo, el valor de la lista de instancias relacionadas con una dada se devuelve por referencia ganándose en eficiencia. Asimismo, la referencia se devuelve como constante para que el usuario no pueda modificar el vector manualmente, cosa que produciría riesgos de inconsistencia entre los dos lados de la asociación. En el caso n a n, tanto

LeftLocalStored como RightLocalStored son vectores de punteros a elementos de la otra clase.

La implementación de los servicios de estas asociaciones sigue siempre el mismo patrón. La asociación le pide al rol que le proporcione el atributo de acceso y a partir de ahí consigue el objeto localStored correspondiente, para consultar su contenido (por ejemplo en getLeft()) y modificarlo (por ejemplo en erase()). Véase la figura 115.

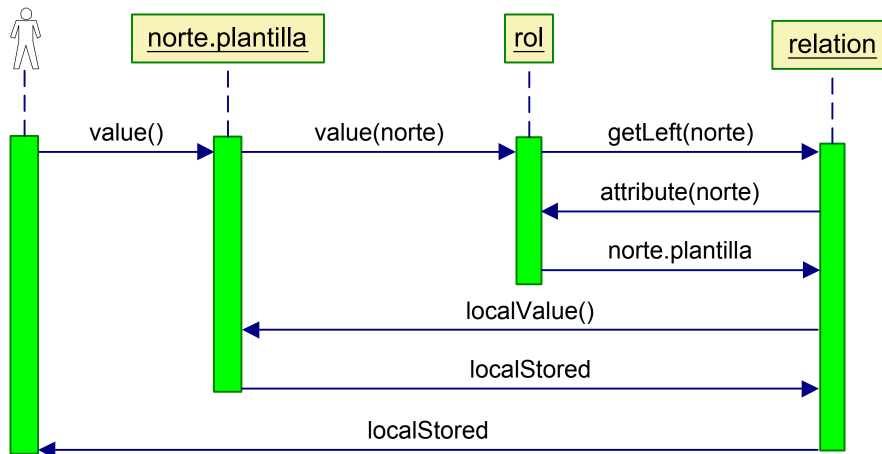


Figura 115. Cálculo de los objetos asociados a uno dado

En el caso de operaciones de escritura, la asociación debe modificar el contenido almacenado en los dos atributos de acceso. La figura 116 muestra el proceso seguido cuando a un atributo se le solicita la inserción de un compañero. La asociación se encarga del proceso modificando el LocalStored de dicho atributo añadiéndole la nueva instancia y actualizando el LocalStored de esa nueva instancia añadiéndole el objeto. De este modo, al asignar un compañero a un objeto, automáticamente se asigna dicho objeto como compañero del otro siendo imposible que aparezcan inconsistencias. El coste de una operación de inserción es constante pues todos los accesos son directos. En el caso de una operación de borrado el coste coincide con el de eliminar una componente en un vector.

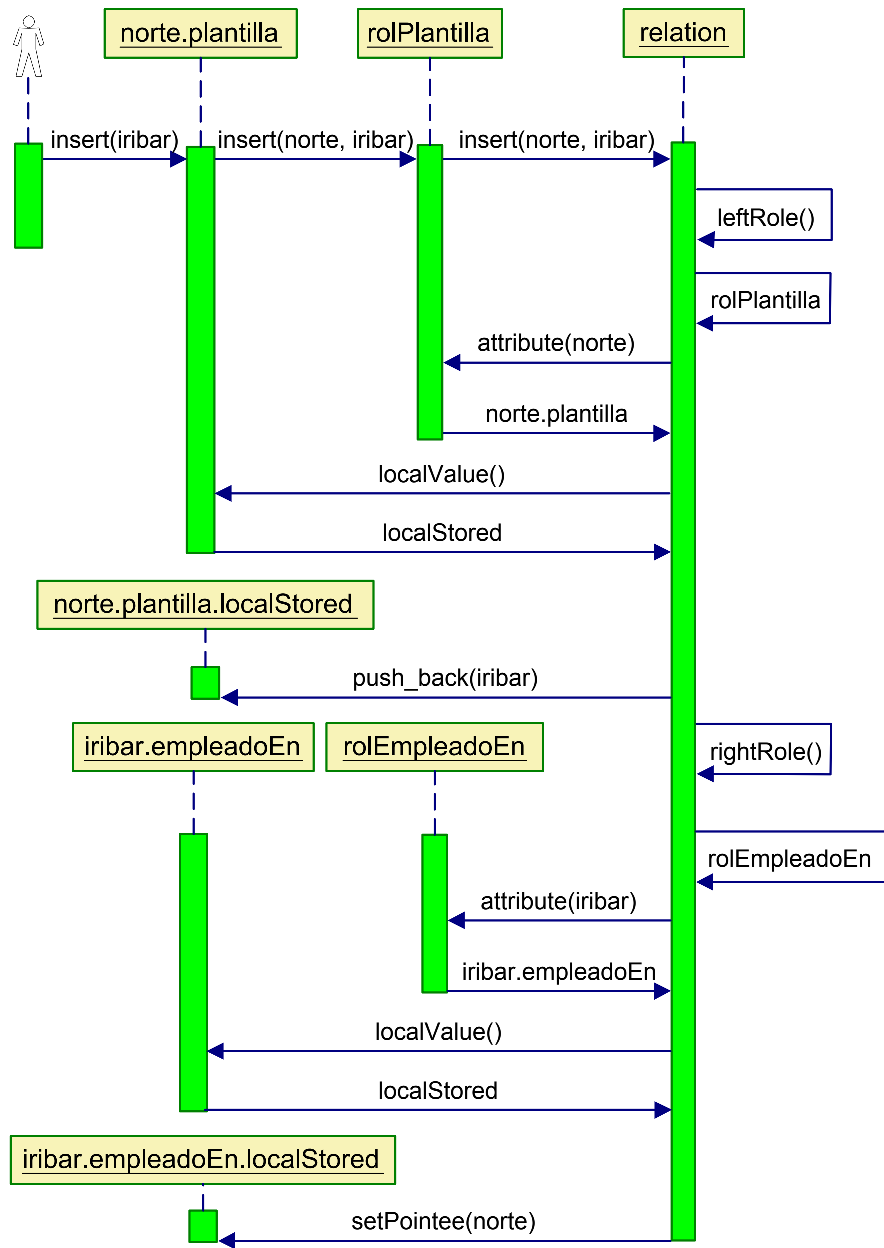


Figura 116. Algoritmo de inserción de instancias de asociación

3.3.1. Desarrollo de código con asociaciones en memoria

En este apartado se muestra cómo se debe usar la librería de facets para trabajar con asociaciones. En primer lugar se va a ver cómo se declara y define el objeto asociación. Como este objeto no pertenece a ninguna de las clases participantes sino que está relacionado con las dos, lo más lógico es declararlo en un fichero aparte.

En el fichero fuente se incluirá la definición de la asociación y también la definición de los dos roles. Aunque los roles podrían colocarse en sus clases respectivas la inclusión se hace en este fichero para evitar problemas de iniciaciones y para simplificar la creación de macros. Si hubiera más asociaciones entre las dos clases se colocarían también en este mismo fichero.

En el constructor de la asociación se indican simplemente su nombre y los roles participantes. En el constructor del rol se indica la cardinalidad mínima y máxima y la asociación correspondiente. Además, como en el caso de los facets básicos, los dos primeros parámetros serán el nombre del rol y el atributo al que está asociado.

```
//RelOficinaEmpleado.h
class Oficina; class Empleado;
extern Relation1Nnp<Oficina, Empleado> destino;

//RelOficinaEmpleado.cpp
#include "RelOficinaEmpleado.h"
Relation1Nnp<Oficina, Empleado> destino("destino");
LeftRole< Relation1Nnp<Oficina, Empleado> >
Oficina::fplantilla("plantilla", Oficina::plantilla, destino, 1, 100000);
RightRole<Relation1Nnp<Oficina, Empleado> >
Empleado::ftrabajaEn(
    "trabajaEn", Empleado::trabajaEn, destino, 1, 1);
```

Figura 117. Declaración de la asociación

Para evitar redundancias innecesarias y, sobre todo, para preservar el carácter privado de los roles, en el constructor de la asociación no se indican los

roles. Como en el constructor de los roles sí se indica la asociación, es la propia clase rol la que se encarga de establecer el enlace inverso diciéndole a la asociación cuáles son sus roles. La figura 117 muestra un ejemplo de declaración y definición de una asociación junto con sus roles.

El proceso de declaración y definición de los atributos de acceso y de los roles es muy similar al empleado para trabajar con atributos básicos. En las declaraciones de las clases participantes se declara el rol como elemento estático y el atributo se parametriza por el rol. En este caso no hay que definir el facet en el fichero cpp pues ya se ha hecho en el fichero de la asociación (figura 118).

```
//Oficina.h
class Empleado;
class Oficina : public Object {
 protected:
     static LeftRole< Relation1Nnp<Oficina, Empleado> > fplantilla;
 public:
     RelationAtribute<LeftRole<Relation1Nnp<Oficina, Empleado>
>,
                         fplantilla > plantilla;
 ...
};

//Empleado.h
class Empleado : public Object {
 protected:
     static RightRole< Relation1Nnp<Oficina, Empleado> >
          ftrabajaEn;
 public:
     RelationAtribute< RightRole<Relation1Nnp<Oficina, Empleado>
                                    >, ftrabajaEn > trabajaEn;
 ...
};
```

Figura 118. Declaración de clases con atributos de acceso

3.3.2. Macros para manejar asociaciones en memoria

En el código de la figura 118 se puede observar que, aunque no se necesitan muchas líneas de código para crear una asociación y sus atributos, sí es cierto que las declaraciones de tipos resultan un poco farragosas. Como en el caso de los atributos básicos, unas macros sencillas van a permitir disminuir notablemente la longitud de estas declaraciones quedando el código más claro y compacto. La declaración de estas macros se puede ver en la figura 119.

```
#define DEC_LEFT_ATTRI(Rel, name) \
 protected: static const LeftRole<Rel> f##name; \
 public: RelationAtribute<LeftRole<Rel>, f##name> name;

#define DEC_RIGHT_ATTRI(Rel, name) \
 protected: static const RightRole<Rel> f##name>; \
 public: RelationAtribute<RightRole<Rel>, f##name> name;

#define DEC_RELATION(Rel, rel) extern Rel rel;

#define DEF_RELATION( \
  Rel, rel, leftRole, rightRole, m1, m2, m3, m4) \
      Rel rel(#rel); \
      LeftRole<Rel> Rel::Left::f##leftRole( \
       #leftRole, Rel::Left::leftRole, rel, m1, m2);
      RightRole<Rel> Rel::Right::f##rightRole( \
       #rightRole, Rel::Right::rightRole, rel, m3, m4);
```

Figura 119. Macros para trabajar con asociaciones

DEC_LEFT_ATTRI y DEC_RIGHT_ATTRI sirven para declarar los atributos de acceso y sus roles. DEC_RELATION se usa para declarar la asociación y DEF_RELATION para definir la asociación y los roles.

Las declaraciones de las asociaciones y los atributos haciendo uso de estas macros se pueden ver en la figura 120. Se observa que el código resulta mucho más legible y comprensible que antes.

```
//RelOficinaEmpleado.h
class Oficina;
class Empleado;
DEC_RELATION(Relation1Nnp<Oficina, Empleado>, destino);

//RelOficinaEmpleado.cpp
#include "RelOficinaEmpleado.h"
DEF_RELATION(Relation1Nnp<Oficina, Empleado>,
    destino, plantilla, trabajaEn, 1, 1000, 1, 1);

//Oficina.h
class Empleado;
class Oficina : public Object {
    DEC_LEFT_ATTRI(Relation1Nnp<Oficina, Empleado>, plantilla);
    ...
};

//Empleado.h
class Empleado : public Object {
    DEC_RIGHT_ATTRI(Relation1Nnp<Oficina, Empleado>,
                    trabajaEn);
    ...
};
```

Figura 120. Declaración de atributos y asociaciones con macros

3.3.3. Visibilidad y control de acceso

Un factor muy importante en la calidad del software es la independencia entre los distintos módulos de la aplicación. Si dos módulos son independientes se puede modificar uno sin que esto afecte al otro. Los lenguajes de programación dan soporte a esta característica mediante construcciones sintácticas que permiten especificar que un elemento tiene prohibido el acceso a otro. Si el programador intenta violar esta prohibición se produce un error en tiempo de compilación o ejecución según los casos. La visibilidad es una restricción menor, se permite a un módulo acceder a otro pero debe indicarlo expresamente, es decir, por defecto no se le permite el acceso pero puede conseguir ese acce-

so mediante una petición expresa. Con esto se consigue reducir el espacio de nombres evitando que se produzcan colisiones y además se evita la aparición de errores accidentales.

En el caso de las asociaciones que se están analizando, resulta importante poder establecer los permisos de acceso a los diferentes elementos. Se indica a continuación cómo se puede establecer o restringir el acceso a los objetos involucrados en una asociación.

Acceso a la asociación. Tal y como se han introducido, las asociaciones son visibles desde las clases participantes. El resto de objetos no tiene visibilidad de la asociación debido a que está declarada en un fichero de cabecera aparte. Incluyendo este fichero cabecera se consigue la visibilidad. Sin embargo, el control de acceso, una vez obtenida la visibilidad, es público al tratarse de variables globales. Puede interesar que se pueda acceder a las asociaciones solamente desde los atributos de acceso y tal vez desde algún elemento más. Para ello basta incluir el objeto asociación como campo estático privado de una clase e indicar que se puede acceder a ella desde las clases participantes:

```
class ControlDestino {
 private:
     static Relation1Nnp<Oficina, Empleado> destino;

     //Permitimos que los roles tengan acceso a la asociación
     //poniendo como amigas las clases donde se declaran:
     friend class Oficina;
     friend class Empleado;
};
```

Figura 121. Control de acceso a la asociación

Al hacer esto ya sólo es posible acceder a la asociación por medio de los atributos de acceso.

Acceso a los roles. Por defecto, se puede acceder a los roles desde la asociación (ya que en el constructor del rol se le proporciona su dirección a la asociación) y desde las clases participantes y las heredadas de esta (ya que está declarado como miembro protegido de la clase). Para permitir el acceso público al rol basta con hacer público el acceso a la asociación ya que desde ésta sí se

puede acceder al rol. Más interés tiene restringir el acceso al rol desde su propio atributo de acceso. Gracias a esto, desde un objeto no se va a poder acceder a sus elementos asociados ya que para ello el atributo debería tener acceso al rol y eso es lo que se va a impedir. De este modo, una expresión del tipo `norte.plantilla.insert(e)` daría error. Esta característica recibe el nombre de *navegabilidad* de la asociación. No se puede aplicar la misma técnica que con las asociaciones ya que rol y atributo están declarados en la misma clase. La solución consiste simplemente en impedir acceder al atributo declarándolo como privado y haciendo al rol clase amiga del atributo. Otra posible solución sería suprimir sin más la declaración del atributo pero esto complicaría la implementación de las asociaciones y además impediría realizar comprobaciones de cardinalidad.

3.3.4. Uso de las asociaciones

Una vez declaradas las clases y las asociaciones su uso es muy sencillo y se puede ver en el ejemplo de la figura 123. En él se ve cómo los atributos de acceso se tratan de forma uniforme y cómoda y el usuario tiene la impresión de que está trabajando con atributos simples.

Por otro lado, el operador de asignación se puede redefinir para el caso de los atributos de acceso para que tenga la semántica intuitiva de sustituir los posibles elementos asociados por los que se están asignando (figura 122).

```
iribar.trabajaEn = zaragoza;
//Elimina el posible enlace que tenga iribar sustituyéndolo por zaragoza

Vector<Empleado*> v = makeVector(&urruti, &iribar);
norte.plantilla = v;
/* Elimina uno a uno todos los enlaces de plantilla y luego añade uno a uno
todos los elementos del vector de modo que al final los elementos asociados
a norte sean exactamente los indicados en el vector */
```

Figura 122. El operador de asignación en atributos de acceso

```
Oficina norte;
Oficina zaragoza;
Empleado urruti;
Empleado iribar;

//Adición de un elemento a la asociación:
norte.plantilla.insert(iribar);
//Equivale a iribar.trabajaEn.insert(norte)

iribar.trabajaEn.insert(zaragoza);
//Excepción: iribar ya está relacionado con un objeto

urruti.trabajaEn.insert(norte);
//norte cuenta con dos empleados

//Acceso a las instancias relacionadas con una dada:
cout << norte.plantilla.value(); //Escribe [urruti, iribar]
const Oficina& of = urruti.trabajaEn.value(); // of = norte

//El conversor automático permite omitir value() para acceder al valor:
const Vector<Empleado*>& v = norte.plantilla;
//Equivale a norte.plantilla.value()

//Acceso a las instancias de la asociación desde la propia asociación:
destino.erase(norte, urruti);
//Equivale a norte.erase(urruti) y a urruti.erase(norte)

//Acceso a datos de la asociación y del rol:
cout << norte.plantilla.role().minCard(); //Escribe 1
cout << norte.plantilla.role().relation().name();
//Escribe destino
```

Figura 123. Trabajo con asociaciones

3.4. Herencia

En el capítulo anterior se vio cómo la librería de facets admite varios mecanismos de herencia permitiendo en una clase derivada personalizar un facet de

la clase base. Además, la solución funciona en casos de herencia múltiple o virtual. El mecanismo de herencia se puede trasladar al caso de las asociaciones ya que los roles no son más que un caso especial de facets. Por ello, es posible sustituir en la clase derivada, una asociación por otra que tenga un comportamiento particularizado para las clases derivadas. Para conseguirlo bastará con redefinir los roles izquierdo y derecho y crear una nueva asociación que utilice estos nuevos roles.

Como ejemplo considérese que en la empresa hay empleados autónomos que no están asignados a ninguna oficina. Se considera que su oficina es una oficina móvil representada por ejemplo por el coche de la empresa que utilizan (figura 124). Si se sigue usando en estas clases la asociación destino, surgirá una incongruencia entre la multiplicidad de la asociación (1 oficina y n empleados) y el hecho de que en las clases derivadas sólo se permite 1 empleado por oficina. La asociación permitiría incorrectamente que una oficina móvil tuviera más de un empleado.

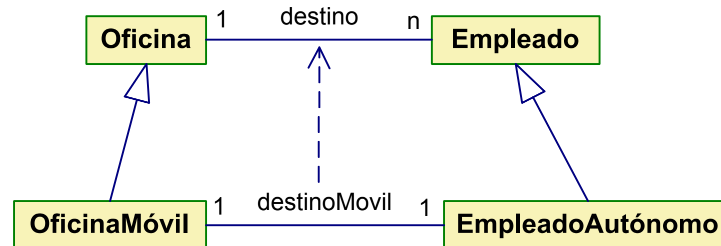


Figura 124. Especialización de asociaciones en herencia

En las clases derivadas sólo hay que declarar el role ya que el atributo se hereda automáticamente. Utilizando la misma definición de RelationAttribute vista en el caso de facets básicos la declaración de las clases para el caso izquierdo podría ser la mostrada en la figura 125.

```
//Oficina.h
class EmpleadoAut;
class Oficina : public Object {
    ...
    typedef LeftRole< Relation1Nnp<Oficina, Empleado> > LR;
```

```
 protected:
     static LR fplantilla;
     virtual LR& vfplantilla() {return fplantilla;}
     RelationAttribute<LR, vfplantilla> _plantilla;
 public:
     RelationAttribute<LR, vfplantilla> plantilla() { return _plantilla; }
};
class OficinaMov : public Oficina {
     ...
     typedef LeftRole<
          Relation1Nnp<OficinaMov, EmpleadoAut> > LR;
 protected:
     static LR fplantilla;
     virtual Oficina::LR& vfplantilla() {
          return reinterpret_cast<OficinaMov::LR&>(fplantilla);
     }
     // RelationAttribute<Oficina::LR, vfplantilla>
     // _plantilla; Omitido, el atributo se hereda del padre
 public:
     RelationAttribute<LR, vfplantilla> plantilla() {
          return reinterpret_cast<...>(_plantilla); }
};
//Ídem para Empleado y EmpleadoMov
...
//En RelOficinaEmpleado.cpp:
Relation1Nnp<Oficina, Empleado> destino("destino");
Oficina::LR Oficina::fplantilla(
      "plantilla", Oficina::_plantilla, destino, 1, 100000);
//En RelOficinaMovEmpleadoAut.cpp:
Relation1Nnp<OficinaMov, EmpleadoAut>
 destinoMovil("destinoMovil");
OficinaMov::LR OficinaMov::fplantilla(
 "plantilla", OficinaMov::_plantilla, destinoMovil, 1, 1);
```

Figura 125. Código para especialización de asociaciones

Naturalmente el uso de macros permite simplificar la notación. Por ejemplo para OficinaMov basta poner:

```
class OficinaMov {
    REDEC_LEFT_ATRI(Relation1Nnp<OficinaMov, EmpleadoAut>,
        plantilla);
    ...
};
```

Figura 126. Uso de asociaciones especializadas

Como en el caso de la herencia de facets básicos, cuando el atributo necesita el rol, llama a su parámetro de plantilla (el método virtual vfplantilla) que devuelve el rol de la clase padre o de la clase hija según el tipo dinámico del objeto. El código de la figura 127 muestra cómo se usan estas asociaciones.

```
OficinaMov coche1;
Oficina& of = coche1;
EmpleadoAut knight, lauda;
of.plantilla().insert(knight);

of.plantilla().insert(lauda);
/*Equivale a of.vfplantilla().insert(of, lauda)
Como el método vfplantilla es virtual se devuelve el rol
OficinaMovil::fplantilla que es el que se encarga de la inserción. Como
este rol admite cardinalidad máxima 1 se produce un error.*/

//Si la inserción se hace desde el otro lado:
lauda.trabajaEn().insert(of)
/*Equivale a lauda.vfplantilla().insert(lauda, of)
lauda no es una referencia, no se aplica virtualidad, directamente se usa el
rol y la asociación de la clase derivada, su rol compañero admite
multiplicidad máxima 1 y de nuevo se detecta el error */
```

Figura 127. Uso de asociaciones especializadas

3.4.1. Inconsistencias en asociaciones especializadas

Una diferencia importante de la herencia de roles con respecto a la de facets básicos es que la clase de rol derivado (LeftRole<OficinaMov, EmpleadoAut>) no hereda de la clase de rol padre (LeftRole<Oficina, Empleado>). Por eso, en la clase derivada el método redefinido vfplantilla() debe devolver un valor de la clase del rol padre obligando a usar una conversión de tipos con reinterpret_cast. Como siempre, el uso de reinterpret_cast es síntoma de un posible problema. En efecto, el problema surge cuando se hace una inserción mixta, por ejemplo de una oficina móvil con un empleado normal:

```
coche1.plantilla.insert(iribar)
```

Con el sistema que se acaba de ver esta orden es correcta. Se usa la asociación hija destino_movil la cual sólo permite parejas OficinaMov-EmpleadoAut produciéndose una inconsistencia en la asociación. La solución consiste en incluir en el método insert de una asociación "virtual" un test que compruebe que el rol de coche1 y el rol de iribar pertenecen a dicha asociación. Realmente lo que se está haciendo es realizar una implementación manual de multimétodos, que, como ya se ha dicho, son métodos en los que la selección del método de disparo depende del tipo real de más de un objeto.

3.4.2. Asociaciones y herencia múltiple o virtual

Cuando una clase del modelo hereda de dos puede ocurrir que la distancia del comienzo del objeto a un atributo no sea fija. Los punteros a miembro de C++ son lo suficientemente inteligentes como para que of.*offset sea siempre el atributo correcto independientemente del tipo de of. Por eso, dado un facet (y por tanto el offset) y un objeto se puede acceder al atributo correspondiente.

Para realizar el camino inverso, los atributos tienen el método object() que en teoría devuelve el objeto que lo contiene. Cierto es que a partir del atributo se puede obtener el offset pero el offset es la función opuesta. Se puede conseguir hallar el offset inverso en caso de herencia simple ya que la posición relativa de objeto y atributo no varía. En caso de herencia múltiple no es posible:

Sin embargo, el acceso al objeto es absolutamente necesario ya que cuando el atributo dice al rol que inserte un par de datos uno de ellos es el objeto que contiene al atributo y que se ha perdido al poner el sufijo del atributo. Para que

la llamada of.plantilla.insert(emp) se pueda traducir en of.plantilla.role(of, emp) debe haber alguna forma de recuperar el objeto.

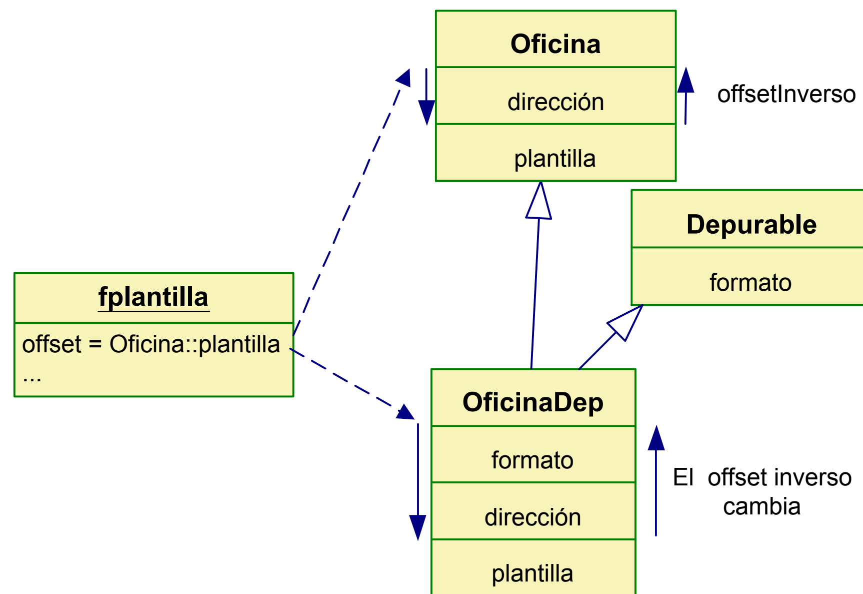


Figura 128. La distancia objeto-atributo puede no ser constante

La solución propuesta consiste en crear una clase AttributeAndObject que guarde una referencia a un RelationAttribute y además una referencia a un objeto. De este modo, un AttributeAndObject guarda internamente el objeto y accede directamente a él. Para usar un AttributeAndObject se crea un método de acceso al atributo como en el caso de herencia de facets. Este método no devolverá el atributo, sino el atributo más el objeto (figura 129).

Si ahora se hace una operación como esta:

```
oficina.plantilla().insert(emp);
```

la expresión se transforma en

```
oficina.plantilla().role().insert(object(), emp);
```

```
class Oficina : public Object {
     ...
 protected:
     RelationAttribute<Oficina, Empleado> _plantilla;
 public:
     AttributeAndObject<Oficina, Empleado> plantilla() {
          return makeAttributeAndObject(*this, _plantilla);
     }
     ...
};
```

Figura 129. Al acceder al atributo se conserva la información del objeto

AttributeAndObject tiene definido el método object() de forma trivial. El resto de operaciones, como por ejemplo la de acceso al role, las delega al otro componente que guarda, es decir, al atributo, con lo que el problema queda finalmente resuelto.

3.5. Otras clases de asociaciones

Las clases que se acaban de ver cubren un buen número de necesidades en cuanto a asociaciones. Además, es posible añadir nuevas clases de asociación, desde cero o aprovechando las tres plantillas ya creadas.

Por ejemplo, se pueden crear clases de asociación para otros tipos de multiplicidad o bien modificar la política de almacenamiento de instancias haciéndolo local a la asociación. Esta centralización de los datos facilitaría tareas en las que se requiera un manejo global de la información de la asociación, por ejemplo para guardar los datos en disco. Además, la adición de atributos de asociación es mucho más sencilla. La desventaja consiste en que tareas como la inserción y la del cálculo de los objetos asociados a uno dado ya no se pueden ejecutar en tiempo constante.

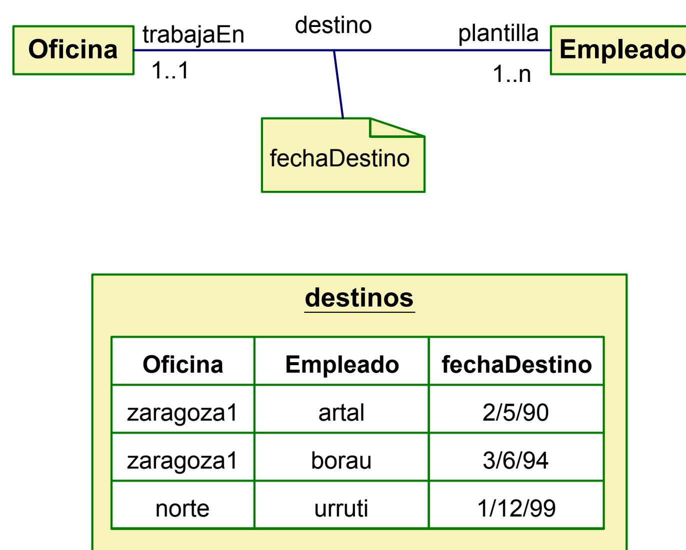
LeftLocalStored y RightLocalStored deberán definirse como Void ya que no se va a almacenar ningún dato en los objetos de las clases participantes. En la clase asociación habrá un campo que contendrá el vector de parejas de la

asociación. Una operación de inserción simplemente añadirá una pareja nueva al vector. La obtención de los elementos asociados a uno dado aconseja utilizar una estructura sofisticada de búsqueda. Por eso, el contenedor de parejas deberá implementarse utilizando un diccionario o una tabla de dispersión.

También se pueden crear asociaciones cuyos valores se almacenen permanentemente en bases de datos. Esto resulta muy interesante cuando los atributos de las clases también tienen propiedades de persistencia ya que entonces toda la información del modelo podrá guardarse y reconstruirse posteriormente. [Zarazaga 00] utiliza la librería de facets para construir un entorno de persistencia en bases de datos relacionales. La información de las tuplas se guarda en una tabla formada por los campos claves de las dos clases participantes.

Especialmente interesante resulta el poder proporcionar soporte a asociaciones que estén cualificadas con atributos propios. Por ejemplo, en la asociación destino cada enlace puede incluir la fecha en que se destinó al empleado a la oficina (figura 130).

Para modelar estas asociaciones con atributos hay que crear nuevas clases que permitan establecer y recuperar esta información adicional.



Oficina	Empleado	fechaDestino
zaragoza1	artal	2/5/90
zaragoza1	borau	3/6/94
norte	urruti	1/12/99

Figura 130. Asociación con atributos

En primer lugar hay que determinar el lugar físico donde se guardan los atributos de asociación. Si la implementación es local al objeto asociación la solución consiste en añadir columnas adicionales a la tabla de pares. En el caso de que la información esté distribuida por las instancias ocurre que, como el enlace es doble, el atributo debe guardarse únicamente en uno de los dos lados para no duplicar información. En la librería de facets la información de los atributos se guarda en las instancias de la clase derecha. Así, en el caso de una asociación 1 a 1, si Attributes es la clase que guarda la información de los atributos entonces LeftLocalStored seguiría siendo un Ptr<Right> pero RightLocalStored sería un par (Ptr<Left>, Attributes) con objeto de guardar la oficina donde se trabaja y la fecha de alta (figura 131).

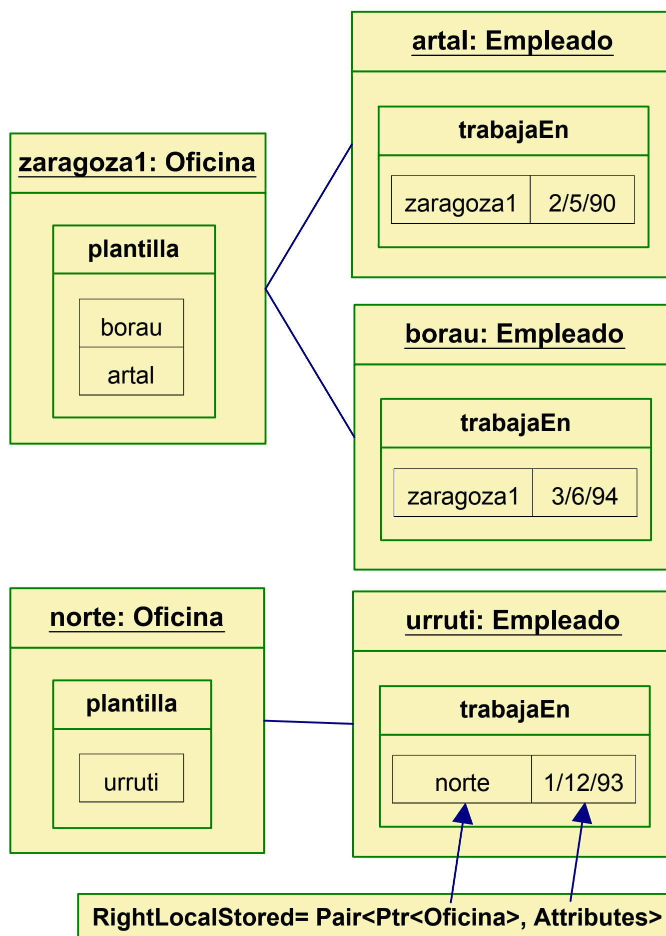


Figura 131. Los atributos de asociación se guardan en la derecha

Para trabajar con pares clase-atributos se ha creado la plantilla Pair (se puede ver en la figura 132).

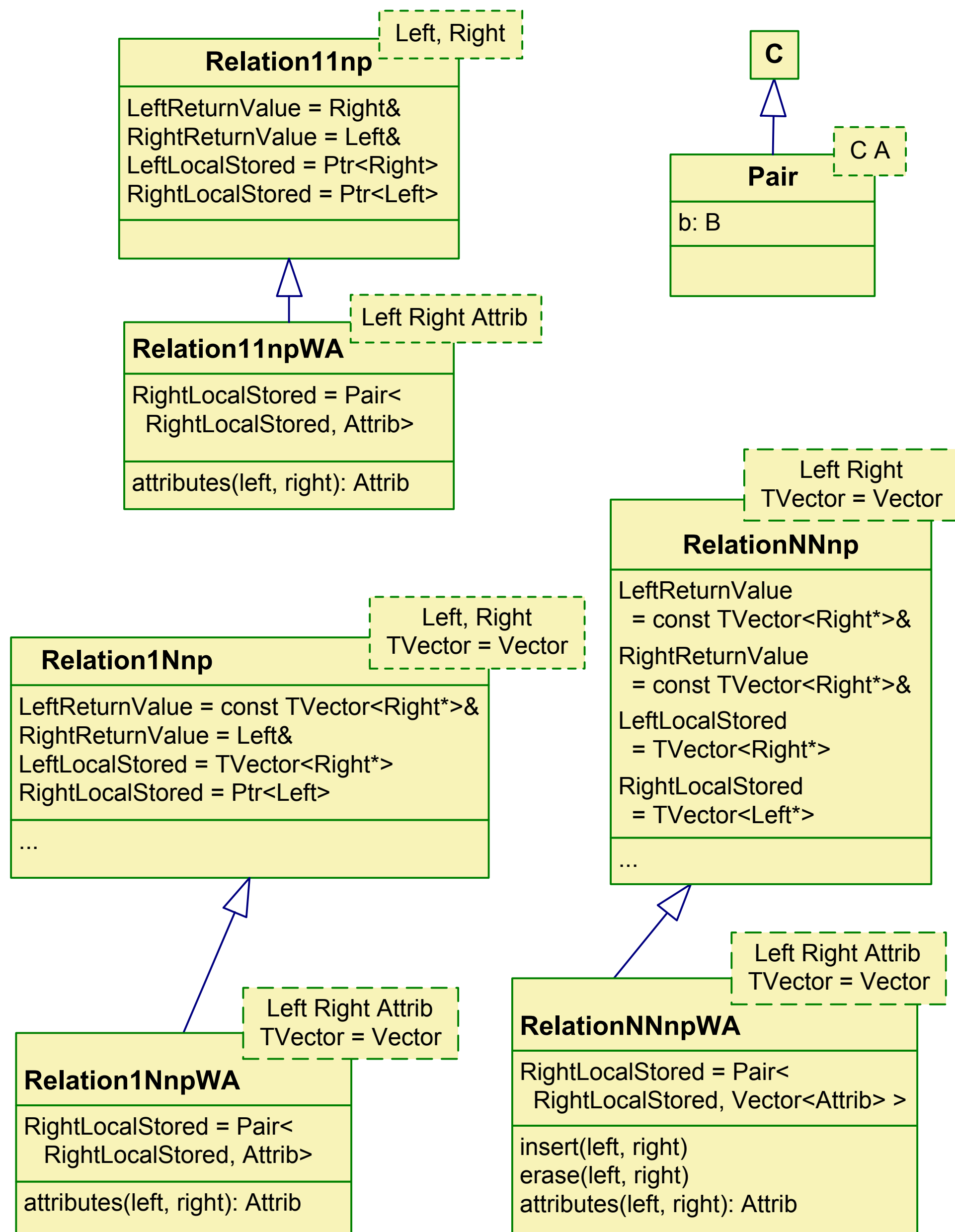


Figura 132. Asociaciones en memoria con atributos

Pair<C, A> hereda de C y añade como atributo un elemento de A. Gracias a esta propiedad, el RightLocalStored de las asociaciones con atributos (por ejemplo Pair<Ptr<Oficina>, Attributes>) hereda del RightLocalStored de la correspondiente asociación sin atributos (en este caso Ptr<Oficina>). Si ahora se define la clase asociación con atributos RelA como hija de su correspondiente sin atributos Rel resultará que la mayoría de los servicios serán válidos. Por ejemplo, el método erase en principio lo que hace es llamar a setNull para poner a 0 el RightLocalStored. Ahora se le pasará a setNull un Pair<Ptr<Oficina>, Attributes>. La llamada es correcta ya que esta clase deriva de Ptr<Oficina> y lo que hará setNull será poner un valor nulo en la parte Ptr<Oficina> que hay en Pair<Ptr<Oficina>, Attributes>, es decir, pondrá un valor nulo en el puntero dejando inalterado el valor del atributo. Todo es correcto ya que cuando se haga una comparación de igualdad para saber si el puntero es nulo la comparación también la hará con la parte izquierda, es decir, con la que ha puesto a 0.

La librería incluye 3 plantillas de asociaciones con atributos que, por los motivos expuestos, se declaran como heredadas de sus correspondientes plantillas de asociaciones sin atributos. Estas plantillas se parametrizan por las dos clases participantes más la clase de los atributos (y en su caso más la plantilla vector). Su estructura se puede ver en la figura 132.

Los únicos métodos heredados que hay que reimplementar son el erase y el insert en el caso de que la información guardada localmente sea un vector. La razón de esta reimplementación está motivada por la implementación de la librería estándar: al llamar a push_back para añadir una instancia se realizará esta llamada en un Pair<Vector<Empleado*>, Vector<Attrib> > y sólo crecerá el primer vector. Debería crecer también el segundo vector añadiendo un valor nulo o un valor por defecto para reservar sitio para el atributo. Naturalmente podría redefinirse push_back en Pair<...> pero lamentablemente dicho método no es virtual en el vector estándar de C++ y se seguiría aplicando el push_back de Vector.

El método attributes(of, emp) sirve para acceder a los atributos del enlace entre of y emp (dando un mensaje de error si no están asociados). Naturalmente se debe permitir acceder a los atributos de asociación a través de las instancias con una llamada del tipo of.attributes(emp). Para no complicar el diseño con una nueva clase de atributos de acceso lo que se ha hecho es incluir sim-

plemente este método en la clase RelationAttribute (y el correspondiente en la clase del rol) (figura 133).

```
template<class Role, Role& role>
Role::Relation::Attributes
RelationAttribute<Role, role>::attributes(Role::Partner x) {
    return role().attributes(*this, x);
}
```

Figura 133. Método genérico para solicitar atributos desde una de las clases

```
//RelOficinaEmpleado.h
//DEC_RELATION_1NnpWA(Oficina, Empleado, destino)
class Oficina;
class Empleado;
struct Attributes_destino;
Relation1NnpWA<Oficina, Empleado, Attributes_destino> destino;
struct Attributes_destino {
    //ADD_ATTRI(Date, fechaDestino)
    Date fechaDestino;
//END_DE_RELATION_1NnpWA
};

//RelOficinaEmpleado.cpp
//DEF_RELATION_1NnpWA(Oficina, Empleado, destino),
Relation1Nnp<Oficina, Empleado> destino("destino");
LeftRole< Relation1Nnp<Oficina, Empleado> > Oficina::fplantilla(
     "plantilla", Oficina::plantilla, destino, 1, 100000);
RightRole<
    Relation1Nnp<Oficina, Empleado> > Empleado::ftrabajaEn(
        "trabajaEn", Empleado::trabajaEn, destino, 1, 1);
```

Figura 134. Declaración y definición de asociaciones con atributos

Al incluir este método en RelationAttribute puede ocurrir que se instancie esta clase por un rol cuya asociación no tenga atributos de asociación. Esta clase asociación no tendrá hecho el typedef Relation::Attributes. Esto, que en principio daría lugar a un error de compilación, en la práctica no produce ningún problema ya que un método de una plantilla sólo se instancia para una clase concreta si dicho método es invocado. Por eso, si estamos en presencia de una asociación normal y no se llama a attributes(x) no ocurrirá nada, y si se llama, entonces dará un error de compilación.

El código de la figura 134 muestra cómo se declaran los objetos asociación.

Se ponen en comentarios las macros que sirven para generar dicho código. Si se usan macros, la clase de los atributos se generará automáticamente con nombre Attributes_*nombreDeLaAsociación*. A esta clase se le pueden ir añadiendo campos con ADD_ATTRI(Tipo, nombre).

Para declarar el atributo de acceso hay tres macros según la multiplicidad. El código de la figura 135 muestra su uso y cómo se expansiona. También se puede usar la macro DEC_LEFT_ATTRI empleada para asociaciones normales, pero en este caso habrá que hacer explícita la clase de los atributos de asociación.

```
class Oficina : public Object {
    //DEC_LEFT_ATRI_1NnpWA(Oficina, Empleado,
    //                              plantilla, destino);
 protected:
    static LeftRole<Relation1Nnp<
     Oficina, Empleado, Attributes_destino> > fplantilla;
 public:
    RelationAtribute<LeftRole<Relation1Nnp<Oficina, Empleado,
         Attributes_destino> >, fplantilla > plantilla; ...
};
```

Figura 135. Atributos de acceso a asociación con atributos

Un ejemplo del uso de estas asociacione se muestra en la figura 136.

```
Oficina norte; Empleado urruti, iribar;
//Adición de dos elementos a la asociación:
norte.plantilla.insert(iribar);
urruti.trabajaEn.insert(norte);
//Acceso a las instancias relacionadas con una dada:
cout << norte.plantilla //Escribe [urruti, iribar]
//Establecimiento de los atributos de un enlace
norte.plantilla.attributes(urruti).fechaDestino = Date(2, 5, 1995);
/* norte.plantilla.attributes(urruti) es una referencia a un objeto de la
clase de los atributos. Puede llamarse a cualquier método de dicha clase
con él o acceder a campos públicos como en el ejemplo */
//Lectura de los atributos de un enlace
cout << norte.plantilla.attributes(urruti).fechaDestino;
Attributes_destino atri =
    destino.attributes(norte, urruti).fechaDestino;
//Borrado de un enlace
norte.plantilla.erase(urruti);
//Error al acceder a los atributos de un enlace inexistente:
cout << norte.plantilla.attributes(norte, urruti);
```

Figura 136. Ejemplo de uso de una asociación con atributos

3.6. Conclusiones

En este capítulo se ha mostrado cómo la librería de facets puede extenderse para implementar asociaciones. Se ha aprovechado la propiedad de los atributos enriquecidos de disponer de un lugar donde colocar metainformación para ubicar ahí los datos de la asociación y el lado desde el que el atributo va a acceder a la asociación. Se ha llamado rol a esta información y se han creado unas clases de facets para representar roles consiguiendo un modo de trabajo homogéneo con el de los atributos básicos.

Los atributos de acceso a las asociaciones y los roles son clases ligeras que delegan todos los servicios en la asociación. Gracias a esto, la política de gestión de los datos de la asociación no está dispersa por las clases sino que se

centraliza en la asociación misma. En una implementación no sofisticada debe replicarse en las clases participantes cuando se crea una segunda asociación el código. En nuestro caso el problema se resuelve simplemente creando una nueva instancia de la clase de la asociación ya que inmediatamente se pueden declarar los atributos de acceso a dicha asociación. Éstos dispondrán de todos los servicios de manejo de la asociación al delegar en ella todo el trabajo. Cada vez que se quiera introducir una nueva asociación sólo hay que implementar la clase asociación correspondiente. Las clases de los atributos y de los roles que permitirán trabajar con esa asociación se generarán automáticamente.

La infraestructura desarrollada permite a las asociaciones guardar información en los objetos de las clases participantes. Esto ha permitido construir asociaciones que trabajan en memoria con complejidad computacional constante para consultas e inserciones ofreciendo un rendimiento similar al que se obtendría realizando la codificación a mano mediante algún patrón. Estas asociaciones son muy versátiles ya que pueden contar o no con atributos y admiten las multiplicidades más usuales (1 a 1, 1 a varios y varios a varios). Además, están parametrizadas por las clases participantes de modo que se pueden emplear con dos clases cualesquiera. El contenedor empleado para guardar enlaces en el caso 1 a varios o varios a varios también es un parámetro de la plantilla consiguiéndose un máximo de flexibilidad.

Finalmente, se ha mostrado cómo la capacidad de herencia de los facets permite implementar especialización de asociaciones de manera que en sendas clases derivadas de las clases participantes se puedan establecer restricciones o personalizar el comportamiento de dicha asociación.

El tema de las asociaciones ha vuelto a confirmar la naturaleza extensible de la librería: no sólo se ha extendido la librería de facets añadiendo unos facets especiales para trabajar con asociaciones, sino que además cada nueva clase de asociación que se cree quedará automáticamente integrada en la arquitectura permitiendo crear atributos de acceso a ella de forma instantánea. Una prueba de esta capacidad extensiva ha sido el desarrollo de unas clases de asociación que almacenan los datos en una tabla base de datos relacional (estas clases también forman parte de la librería y se explican en [Zarazaga 00]).

En definitiva, la implementación de asociaciones con facets permite tratarlas como elementos de primer nivel dentro del lenguaje, evitando redundancias y facilitando el mantenimiento y la reutilización de código. Gracias a ello, el esfuerzo de paso de la fase de diseño a la fase de implementación se reduce

notablemente. C++ ofrece soporte directo a muchos de los elementos que aparecen en el modelado orientado a objetos pero entre ellos no se encuentran las asociaciones. La implementación de las asociaciones cubre un hueco importante en este sentido y el uso de asociaciones junto con atributos básicos y los mecanismos de inferencia de la herencia permiten que la traducción del esquema del modelo a código C++ sea una tarea casi automática.

Capítulo 4

Contenedores de objetos

En el desarrollo de la librería de facets se hace un amplio uso de estructuras que manejan colecciones de objetos. Las herramientas que proporciona el estándar del lenguaje para este fin presentan ciertas limitaciones que hacen conveniente su ampliación. En este capítulo se estudian estas limitaciones junto con las ampliaciones propuestas.

Los contenedores de datos son una de las estructuras más empleadas y útiles en un lenguaje de programación. Cualquier programa serio hace uso profuso de estas estructuras. Desafortunadamente, en los lenguajes de programación clásicos (Pascal, C, Fortran) el soporte a estas estructuras es escaso limitándose normalmente a proporcionar un vector predefinido poco flexible de tamaño fijo y sin posibilidades de personalización o ampliación.

El advenimiento de los lenguajes de programación orientados a objeto ha permitido introducir clases contenedoras que pueden aparecer en la librería estándar del lenguaje y que el usuario puede personalizar y aumentar [Wright 00] [Meyer 94] [Patapis 95] [Rogue 00] [McCluskey 98b]. Desgraciadamente, en muchos casos, estas clases no pueden manejarse de la misma forma ni tienen la eficiencia de los arrays primitivos. C++ ha dado un paso adelante en este sentido gracias a las plantillas, a la sobrecarga de operadores y al control explícito del manejo de memoria por parte del programador. Con este conjunto de herramientas se pueden crear familias de contenedores similares en prestaciones a los vectores primitivos y que solucionan los problemas inherentes a usar una estructura de bajo nivel como los vectores de C. Éstas clases permiten construir programas más robustos y con menos coste de desarrollo. Muchas de estas librerías de contenedores han ido apareciendo antes del establecimiento del estándar del lenguaje.

La aparición del estándar de C++ ha significado un nuevo paso adelante en este sentido mediante el uso de un paradigma de programación que ha permitido la construcción de una librería potente, eficiente y con amplias posibilidades de ampliación. Esta librería de contenedores se llama STL [Stepanov 94]. STL se basa en el paradigma de la programación genérica [Musser 89]. La idea fundamental de este estilo de programación consiste en desarrollar los algoritmos haciendo abstracción de las estructuras de datos con las que trabajan. De este modo, los algoritmos son independientes de estas estructuras. Analizando la implementación clásica de un algoritmo como el de búsqueda secuencial en un vector se observa que en dicho algoritmo aparecen mezclados dos conceptos: el algoritmo en sí que consiste en recorrer el vector hasta encontrar el dato o llegar al final, y el vector al que se aplica. La separación de estos dos conceptos, de modo que se dé entidad propia al algoritmo, va a permitir su reutilización aplicándolo a otras estructuras como listas o ficheros secuenciales. Por eso, el conjunto de algoritmos de STL es relativamente pequeño comparado con otras librerías existentes, pero con mayores posibilidades de aplicación.

En primer lugar, este capítulo muestra cómo es posible realizar programación genérica en C++ mediante la capacidad de las plantillas para implementar funciones de tipos y polimorfismo estático. Se explica a continuación cómo estos conceptos se han aplicado al diseño de la librería estándar de contenedores STL analizándose los problemas que plantea esta librería para el desarrollo de programas.

Seguidamente se propone una ampliación de dicha librería que, utilizando la misma filosofía de programación genérica, permite trabajar con contenedores más potentes. Se ve cómo problemas, que en otro caso requerirían varias decenas de líneas, se van a poder resolver en unas pocas gracias a la potencia expresiva de estas clases contenedoras.

A continuación se muestra cómo la librería de facets hace uso de estos contenedores avanzados para trabajar con asociaciones múltiples en memoria.

Por último se muestran las capacidades de ampliación de la librería mediante la presentación de unas clases contenedoras especiales que almacenan su información de forma persistente. Estas clases contenedoras se han usado para implementar asociaciones que almacenan sus instancias en una base de datos relacional.

4.1. Programación genérica y contenedores estándar

4.1.1. Programación genérica

Si la programación orientada a objetos hace hincapié en las propiedades comunes de clases, la programación genérica se centra en las propiedades comunes de algoritmos. En esencia se persigue realizar algoritmos lo más generales posibles de modo que puedan aplicarse a la mayor cantidad de tipos de datos posible. Por ejemplo, el algoritmo de búsqueda secuencial escrito de la manera más general posible debería poder aplicarse a cualquier estructura que permita acceso secuencial, por ejemplo vectores, listas enlazadas o ficheros. Un segundo objetivo fundamental de una implementación genérica es que esta generalidad no se consiga a expensas del rendimiento. Un algoritmo genérico aplicado a una cierta estructura contenedora debe ser tan eficiente como uno generado a mano para dicha estructura. Una consecuencia de este requerimiento es que las comprobaciones de tipos deben hacerse en tiempo de compilación. Si se realizaran en tiempo de ejecución habría una perdida de eficiencia respecto del mismo algoritmo creado a mano para una estructura concreta ya que en este caso no es necesario comprobar tipos debido a que el tipo es único.

Considérese el algoritmo de ordenación rápida de los datos de un contenedor [Hoare 62]. La librería de C proporciona una implementación de este algoritmo cuya interfaz es:

```
typedef int (*ComparatorFunction) (void*, void*);
void qsort(void* vectorInit, size_t vectorLength,
     size_t componentSize, ComparatorFunction f);
```

El algoritmo tiene una capacidad genérica parcial: se puede aplicar a cualquier estructura cuyos datos sean contiguos en memoria, sea cual sea el número de datos (vectorLength) y el tipo de sus componentes siempre que estas componentes tengan el mismo tamaño (componentSize). El criterio de ordenación también es libre y se le pasa a qsort por medio de un puntero a una función de comparación que devuelve un valor negativo, nulo o positivo según el primer parámetro de esa función sea menor, igual o mayor que el segundo. Dentro del algoritmo se llama a esta función frecuentemente para comparar dos elementos, perdiéndose eficiencia con respecto a una implementación manual que incluyera directamente en el algoritmo la expresión de comparación. (También se po-

dría hablar de los problemas ocasionados por la nula comprobación de tipos, que puede provocar la caída del programa si se utilizan valores inapropiados).

En el caso de Java su librería estándar proporciona el método:

```
public static void sort(List list, Comparator c)
```

list es un contenedor que debe implementar la interfaz List, por ejemplo LinkedList (lista enlazada) o ArrayList (vector con componentes contiguas en memoria). La implementación por medio de una interfaz o clase abstracta padre obliga a realizar llamadas virtuales para manejar la lista con la consiguiente pérdida de eficiencia. Como en el caso de qsort el uso de un objeto comparador obliga a realizar una llamada a una función para comparar elementos. Este comparador debe, además, comprobar la corrección de los tipos de los elementos. Además, debido a la naturaleza de List las componentes de la lista no pueden ser de tipo predefinido. Tampoco se puede aplicar el algoritmo a vectores predefinidos. Para éste último caso se dispone de una decena de funciones sort que trabajan con vectores predefinidos y con subvectores de éstos y cuyos escalares pertenecen a diferentes tipos básicos o al general Object.

Se han realizado intentos de construcción de librerías de contenedores más generales [ObjectSpace 00] pero el problema de la eficiencia no se ha conseguido resolver.

A continuación se va a ver cómo las plantillas de C++ permiten subsanar estos problemas mediante la construcción de algoritmos que se aplican a cualquier tipo de datos que cumpla los requisitos intrínsecos de dichos algoritmos y preservando la eficiencia.

```
int sum(vector<int> v) {
    int suma = v[0];
    for (int i = 1; i < v.size(); ++i)
        suma = suma + v[i];
    return suma;
}
```

Figura 137. Recorrido secuencial de un vector

Considérese un algoritmo elemental que calcule la suma de un vector de enteros (figura 137). Utilizando plantillas se va a proceder a ampliar el universo de aplicación de este algoritmo. Se van a realizar cuatro tipos de generalizaciones: abstracción del tipo escalar, del contenedor, de las operaciones y del tipo de acceso al contenedor.

Abstracción del tipo escalar

Evidentemente se debe permitir que el algoritmo funcione con otros tipos de datos como reales y complejos y no sólo con enteros. Ésta es la primera generalización y corresponde al propósito inicial que guio la introducción de plantillas en C++ (figura 138).

```
template<class T>
T sum(const vector<T>& v) {
    T suma = v[0];
    for(int i = 1; i < v.size(); ++i)
        suma = suma + v[i];
    return suma;
}
```

Figura 138. Abstracción del tipo escalar

Como ya se vio en el capítulo 1, el polimorfismo estático (el método real se selecciona en función de un tipo conocido en tiempo de compilación) no exige que el tipo de datos pertenezca a una jerarquía de herencia sino que debe cumplir una serie de requisitos impuestos únicamente por las operaciones en las que aparece involucrado dicho tipo dentro del algoritmo. En el caso del algoritmo del ejemplo es necesario que el tipo T tenga definido el operador de asignación, el constructor de copia y la operación de suma.

Abstracción del contenedor

El algoritmo del ejemplo es esencialmente el mismo si en vez de sumar componentes de un vector se suman componentes de un fichero de acceso directo o de cualquier otra secuencia con acceso directo a sus componentes. De nuevo, las plantillas permiten construir una única función que sirve para todos los casos. Lo que se hace es poner como parámetro de la plantilla otra plantilla (figura 139). Como ocurre en el caso del tipo escalar el contenedor debe cum-

plir una serie de condiciones para poder se parámetro de sum: disponer de un operador de acceso directo a una componente (v[i]) y un método para calcular su tamaño.

```
template<class T, template<class> class V>
T sum(const V<T>& v) {
	T suma = v[0];
	for(int i = 1; i < v.size(); ++i)
		suma = suma + v[i];
	return suma;
}
```

Figura 139. Abstracción del contenedor

Abstracción de las operaciones

Se va a generalizar aún más el algoritmo permitiendo emplear cualquier operador y no sólo el de la suma. Por ejemplo, el algoritmo que calcula el producto de los componentes de un vector es esencialmente igual al visto. Esto se puede conseguir parametrizando la plantilla por un objeto que represente la función a emplear (figura 140).

```
template<class F, class T, template<class> class V>
T apply(const V<T>& v) {
	F f;
	T acumulador = v[0];
	for(int i = 1; i < v.size(); ++i)
		acumulador = f(acumulador, v[i]);
	return acumulador;
}
```

Figura 140. Abstracción del tipo escalar

La figura 141 muestra un ejemplo de cómo se puede utilizar apply para sumar.

```
template<class T>
struct Suma {
    T operator()(const T& t1, const T& t2) {
        return t1 + t2;
    }
};
Vector<int> v ...
int s = apply<Suma>(v);
```

Figura 141. Instanciación de apply para sumar[9]

Abstracción del tipo de acceso al contenedor.

Obsérvese que el algoritmo apply recorre el vector secuencialmente y por tanto debería poder aplicarse a otros contenedores como listas y ficheros secuenciales, en los que el acceso a las componentes no es directo. Como estos contenedores no tienen operador de acceso directo es necesario realizar una modificación previa para que el recorrido sea secuencial. Para ello basta utilizar un puntero para recorrer el vector (figura 142).

```
template<class F, class T, template<class> class V>
T apply(const V<T>& v) {
    F f;
    V* p = &v[0];
    T suma = *p; ++p;
    while(p != &v[v.size()])
        suma = f(suma, *p);
        ++p;
    return suma;
}
```

Figura 142. Acceso secuencial

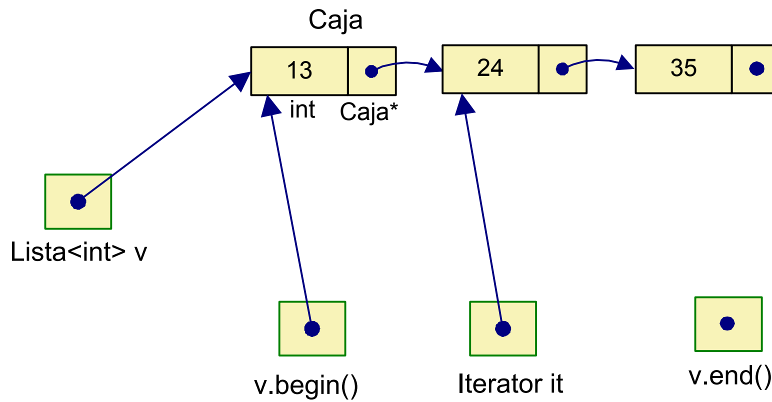
[9] En vez de la estructura Sumar se podría usar el funtor estándar plus<int>.

Para recorrer el vector V<T> se usa el tipo de datos T* que representa un objeto que puede "caminar" por el vector, apuntando en cada momento a una componente y que recibe el nombre de iterador (a veces también se le llama cursor). En el caso de otros contenedores, un puntero a T no es un iterador válido. Por ejemplo, en el caso de una lista enlazada el avance a la siguiente componente deberá ser una operación del tipo p = p->next y en el caso de un fichero secuencial equivaldrá a avanzar una posición en el fichero. Por ello, para poder aplicar el algoritmo de forma genérica a cualquier tipo de contenedor es necesario que el contenedor oferte un tipo de datos iterador que pueda usar el algoritmo para declarar una variable de ese tipo y recorrer con ella el contenedor. En el siguiente ejemplo de apply, el algoritmo presupone que la clase V del contenedor tiene hecho un typedef V::Iterator y que el iterador cuenta con las operaciones == para comparar dos iteradores, * para obtener el elemento apuntado y ++ para avanzar a la siguiente componente. Asimismo se supone que se puede obtener un iterador al primer elemento con v.begin() y un iterador a "fin de contenedor" con v.end(). Para simplificar los parámetros de la plantilla se supone que V es ahora el contenedor concreto el cual proporciona el tipo de sus componentes con V::Escalar (figura 143).

```
template<class F, class V>
T apply(const V& v) {
    F f;
    V::Iterator p = v.begin();
    V::Escalar suma = *p;
    ++p;
    while(p != v.end())
        suma = f(suma, *p);
        ++p;
    return suma;
}
```

Figura 143. Abstracción del tipo de acceso

El algoritmo se podrá aplicar con cualquier contenedor que cumpla los requisitos antes indicados. La figura 144 muestra cómo hacer que una lista enlazada los cumpla.




```
template<class T>
class Lista {
	struct Caja {
		Caja* siguiente;
		T valor;
	};
	Caja* inicioLista; //Puntero al inicio de la lista

 public:
	struct Iterator {
		Caja* posicion;
		Iterator(Caja* pos = 0) : posicion(pos) {}
		void operator++() {
			posicion = posicion->siguiente;
		}
		T operator*() { return posicion->valor; }
		bool operator==(Iterator it) {
			return posicion == it.posicion;
		}
	Iterator begin() { return Iterator(inicioLista); }
	Iterator end() { return NULL; }
};
```

Figura 144. Lista con iterador

4.1.2. La librería de contenedores estándar STL

STL es el primer ejemplo de una librería realmente genérica de contenedores. Sus algoritmos compiten en eficiencia con los de otras librerías y son aplicables a cualquier estructura de datos que cumpla una serie de requisitos impuestos por la lógica interna del algoritmo. Una versión primitiva de la STL se implementó en Ada [Musser 87] pero la mayor rigidez de la genericidad en Ada [Stepanov 95] llevó a sus autores a utilizar un lenguaje más flexible para este propósito. El resultado, STL, se incorporó a la librería estándar de C++.

```
template<class It, class T>
It find(It first, It last, const T& searched) {
/* Busca el valor searched en la secuencia delimitada por los iteradores
first y last (este último exclusive). Devuelve un iterador que apunta al
elemento encontrado o last si no lo encuentra */
  for (It cursor = first; cursor != last; ++cursor)
        if (*cursor == searched) return cursor;
  return last;
}
...
//Sustitución en un vector del elemento "Pepe" por "José"
vector<string> v;
v.push_back("Luis"); v.push_back("Pepe"); ...
vector<string>::iterator p =
    find(v.begin(), v.end(), "Pepe");
if (p == v.end())
    cout << "No encontrado";
else
    *p = "José";
}
```

Figura 145. Ejemplo de uso de un algoritmo estándar

STL consta de una serie de contenedores típicos como vectores, listas doblemente enlazadas, colas y vectores asociativos. Todos estos contenedores están parametrizados por el tipo de sus elementos. Tienen asociado un tipo iterador para caminar por los elementos de la secuencia y cuentan únicamente

con métodos básicos para devolver su tamaño, añadir o eliminar elementos. Los contenedores STL son por tanto estructuras ligeras en las que no se implementan la mayoría de los algoritmos de uso general como búsquedas y ordenaciones. Estos algoritmos se definen como funciones globales que van a poder aplicarse a todos los contenedores. Siendo más precisos, estos métodos se aplican en realidad a un par de iteradores que representan el principio y el final de la secuencia. Todos los contenedores implementan los métodos begin y end que devuelven un iterador al primer elemento y a una posición más allá del último elemento. La figura 145 muestra una posible implementación de la función estándar find para realizar búsqueda secuencial y un ejemplo de su uso.

A cada clase de iterador se le asigna, mediante una función de tipos, una clase que únicamente sirve para representar la categoría del iterador. Estas son las cinco categorías de iteradores:

Nombre de la clase	Contenedores	Ejemplo
output_iterator_tag	Contenedores de acceso secuencial para escribir en ellos	Un stream de salida
input_iterator_tag	Contenedores de acceso secuencial para leer de ellos	Un stream de entrada
forward_iterator_tag	Contenedores de acceso secuencial que permiten leer y escribir	Una lista de enlaces simples
bidirectional_iterator_tag	Contenedores de acceso secuencial tanto para avanzar como para retroceder	Una lista doblemente enlazada
random_iterator_tag	Contenedores que permiten acceso directo en tiempo constante	Un vector

La categoría del iterador puede usarse para seleccionar en tiempo de compilación la implementación más eficiente de un cierto algoritmo. Por ejemplo, la función que calcula el número de elementos que hay entre dos iteradores re-

quiere recorrido secuencial si el iterador es un input iterator pero si se trata de un random iterator el cálculo es directo.

Además de contenedores e iteradores la librería incluye objetos función también conocidos como funtores. Un funtor es un objeto que tiene las propiedades de una función gracias a que su clase implementa el operador funcional operator(). Muchos algoritmos admiten un parámetro adicional que sirve para parametrizar dicho algoritmo por una función que permite hacerlo más general. Por ejemplo, la función find_if busca en una secuencia el primer elemento que satisfaga la condición indicada por el funtor que se le pasa como parámetro. Por satisfacer la condición se entiende que dicho funtor devuelva true para ese elemento. La figura 146 muestra un funtor que se utiliza para buscar en un vector la primera cadena que empiece por la letra P o posterior.

```
struct MayorQueP {
    bool operator()(const string& s) {
        return s >= "P";
    }
};

Vector<string> v ...
Vector<string>::iterator p =
    find_if(v.begin(), v.end(), MayorQueP());
```

Figura 146. Uso de funtores en los algoritmos

Para no tener que declarar explícitamente una clase para cada funtor la librería incluye una serie de clases "fabricantes" de funtores haciendo uso de la técnica de evaluación perezosa vista en el apartado 1.6.3. El funtor MayorQueP() se puede construir dinámicamente utilizando la expresión

```
bind2nd(greater_equal<string>(), "P")
```

greater_equal es un funtor con dos argumentos x, y que devuelve verdad si x > y. Por otra parte, bind2nd es un funtor que toma como entrada otro funtor y un valor. Este funtor fija la y del primer funtor con el valor indicado de modo que se convierte en un funtor de un argumento: bind2nd(f, y)(x) = f(x, y).

De este modo el funtor de la expresión anterior devolverá verdad para un parámetro x si dicho parámetro es mayor que "P". La llamada sería:

```
findif(v.begin(), v.end(), bind2nd(greater_equal<string>(), "P");
```

4.1.3. Problemas en el uso de la STL

Aunque hay un consenso en el sentido de que la STL es una librería potente y que facilita el desarrollo de programas también es cierto que su uso plantea algunos inconvenientes.

1. **Uso de una sintaxis farragosa para llamar a los algoritmos**.

   Por ejemplo, para buscar una cadena v dentro de otra w sin tener en cuenta mayúsculas y minúsculas hay que escribir:

   ```
   search(v.begin(), v.end(), w.begin(), w.end(),
      compose2(equal_to<char>, ptr_fun(toupper),
          ptr_fun(toupper))
   ```

   Aquí, la repetición de la notación begin-end resulta tediosa, y aún lo sería más si los contenedores son a su vez el resultado de una expresión. Por otro lado, la sintaxis resultante de utilizar funtores compuestos es excesivamente complicada haciendo el código más oscuro.

   El problema de la repetición begin-end podría haberse evitado si no se hubiera marcado el objetivo de que los algoritmos de STL funcionasen con vectores planos de C. La notación begin-end permite utilizar un vector dando la dirección de inicio y la dirección de fin del vector:

   ```
   int v[3] = {1, 2, 4};
   find(&v, &v[3], 2);
   ```

   No obstante se podría haber creado un contenedor simple que encapsulara vectores básicos. Podría usarse así:

   ```
   int v[3] = {1, 2, 4};
   find(plain_vector(v, 3), 2);
   ```

   En [Stroustrup 97, pp. 513] se propone una solución mediante la creación de una estructura secuencia que agrupe los dos iteradores.

2. **Riesgo de errores graves por el uso de iteradores**.

Los algoritmos globales de la librería, al trabajar con iteradores no pueden realizar operaciones que impliquen un crecimiento o decrecimiento del vector. ¿Cómo hacer una inserción en un vector teniendo únicamente un puntero a una de sus componentes?. No es posible porque en una componente no hay información de cuál es el vector del que forma parte. Esto hace que ciertas operaciones que proporcionan una secuencia resultado asuman que el contenedor destino tiene suficiente espacio para almacenarla. Por ejemplo, la función transform aplica un funtor a cada elemento de una secuencia colocando los resultados en otra secuencia:

```
vector<int> v; v.push_back(2); v.push_back(5); // v = {2, 5}
vector<int> w; // w = {}
transform(v.begin(), v.end(), w, negate<int>())
// Error, intenta copiar -2 en w[0] y -5 en w[1]
```

Como ejemplo más sencillo de errores en el uso de iteradores considérese la función count que sirve para contar el número de apariciones de un valor en una secuencia. Una llamada del tipo

```
x = count(v.begin(), w.end());
```

producirá un bucle infinito y probablemente una caída del programa ya que el iterador nunca tomará el valor de w.end(). El mismo error se produciría si usamos dos iteradores que recorran un mismo vector siendo el primer iterador mayor que el segundo.

3. **Uso de funciones globales.**

   El hecho de que los algoritmos sean funciones globales produce una polución del espacio estándar de nombres. STL consta de decenas de algoritmos para trabajar con contenedores. Están implementados como funciones globales. Si dichos algoritmos se implementaran como métodos de los contenedores ocuparían el espacio de nombres de las clases contenedoras disminuyendo la posibilidad de usarlos incorrectamente o de que se produzcan colisiones de nombres. Además, el hecho de que algunas operaciones se implementen como métodos obliga a utilizar una notación no uniforme:

```
v.insert(p, w.begin(), w.end());
find(v.begin(), v.end(), 3);
```

4. **Carácter de bajo nivel de la librería.**

Muchas operaciones sencillas que deberían poder realizarse en un único paso necesitan implementarse mediante código confuso que obliga al programador a pensar en detalles de bajo nivel que quedan fuera de la lógica del programa.

Por ejemplo si se desea eliminar todos los componentes de un vector con valor 1 hay que hacer

```
vector<int> v; ...
vector<int>::iterator corte = remove(v.begin(), v.end(), 1);
v.erase(corte, v.end());
```

El problema surge porque `remove` no elimina realmente elementos sino que simplemente los coloca al final devolviendo un iterador al primero de esos elementos "eliminados". A continuación se debe proceder a borrar dichos elementos.

4.2. Ampliación de la librería de contenedores

4.2.1. Extensión de los contenedores estándar

Teniendo en cuenta la importancia de los contenedores en cualquier aplicación, se ha desarrollado una ampliación de la librería que pretende eliminar los problemas anteriores además de proporcionar nuevos tipos de contenedores y otras características que hagan más cómodo y eficaz el trabajo con este tipo de estructuras.

Un objetivo inicial es uniformar las operaciones relacionadas con contenedores de modo que todas ellas sean métodos de la propia clase contenedora. Con ello se va a conseguir una notación uniforme, se van a evitar los problemas de utilizar dos iteradores para indicar una secuencia en cualquier algoritmo y se va a poder usar el mecanismo de la herencia para personalizar la estructura y funcionamiento de los contenedores.

Esta librería de contenedores se puede considerar una ampliación de la STL y hace uso profuso de ella. Se ha visto que los contenedores de STL son clases ligeras que contienen únicamente algunos métodos básicos para trabajar con el contenedor. Por medio de una función de tipos se va a asociar a cada contene-

dor STL un contenedor "pesado" que incluirá como métodos todos los algoritmos globales de STL. Este contenedor se implementa por medio de la plantilla Sequence<V> donde V es un contenedor STL. Sequence<V> hereda de V para que el funcionamiento básico de V esté disponible en su contenedor asociado (figura 147).

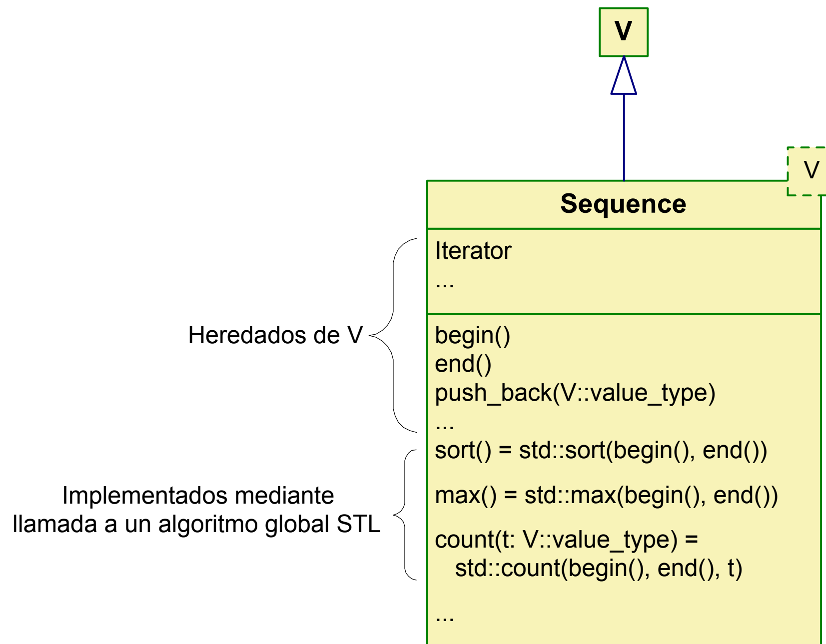


Figura 147. Plantilla para crear contenedores pesados a partir de los de STL

La figura 148 muestra la estructura básica de la librería ampliada de contenedores. SequenceBase mantiene el código común a todas las secuencias. Al igual que los facets y atributos enriquecidos hereda de Printable ya que se puede escribir el contenido de un contenedor. Esto consiste en escribir el valor de cada elemento utilizando unos delimitadores de apertura, separación y cierre que se encuentran almacenados en el campo estático delimiters.

Find es una estructura que se utiliza para devolver información en el caso de búsqueda binaria. Find agrupa dos valores, un booleano para saber si se ha encontrado el dato y un iterador que apunta al elemento encontrado. Si no se ha encontrado apunta al elemento inmediatamente siguiente al que se está buscan-

do. A efectos de inserción esta información resulta útil para seguir mantenien-do la lista ordenada.

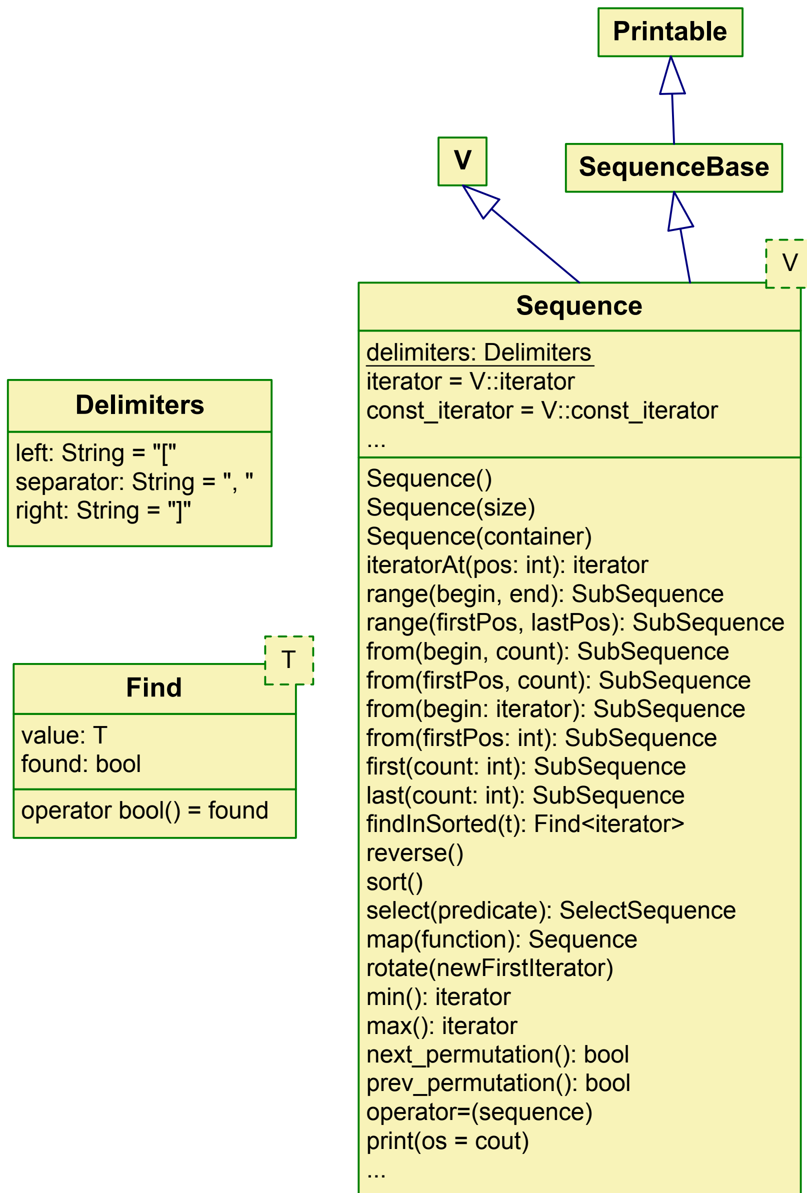


Figura 148. La plantilla Sequence

Sequence se ha instanciado para cada uno de los contenedores básicos de STL dándole al contenedor resultante el mismo nombre que el básico sólo que en mayúsculas. Así, Vector<T> es el contenedor de la librería ampliación del vector STL (figura 149).

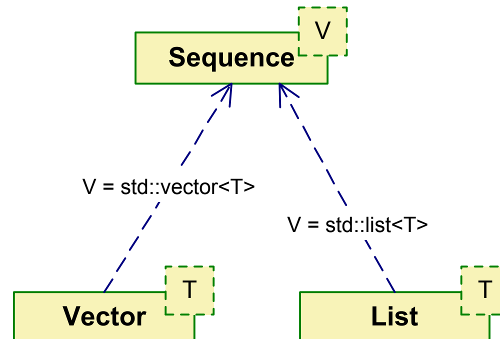


Figura 149. Instanciación de contenedores STL

4.2.2. Vistas

En la librería figuran una serie de contenedores que actúan como vistas de otros contenedores siguiendo la misma idea de las vistas en el modelo relacional de bases de datos. [Powell 00] presenta una librería que utiliza los mismos principios. Un contenedor vista almacena una referencia al contenedor que ve y define su propio iterador para recorrerlo. Por ejemplo, SelectSequence es un contenedor formado por aquellos elementos de otro contenedor que cumplan una cierta condición (figura 150). Cuando se crea un SelectSequence realmente no se hace ninguna operación aunque el usuario puede operar con este contenedor como con cualquier otro, teniendo en cuenta que los cambios en este contenedor afectarán al contenedor visto. Al no tener que crear un nuevo vector con esos elementos se consigue un aumento importante de eficiencia. El iterador de un SelectSequence se implementa de manera sencilla, el operador de avance va avanzando en el contenedor visto, saltándose las componentes que no cumplan la condición.

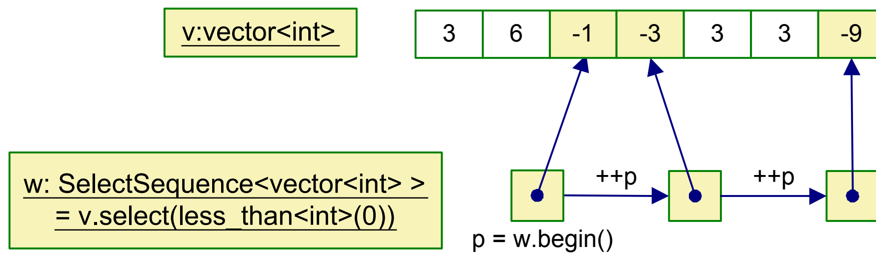


Figura 150. SelectSequence

Los contenedores vista proporcionan muchas posibilidades y permiten realizar de forma trivial acciones que en otro caso no lo serían tanto. De hecho, algunos de los algoritmos globales de STL se pueden implementar con vistas en una sola orden y por eso no se han incluido en Sequence. Así, el algoritmo STL count_if que cuenta las apariciones dentro de un contenedor de los valores que cumplan una cierta condición se puede imitar con la instrucción v.select(condición).size().

SubSequence es un tipo de vista cuyo concepto es bien conocido, representa una subsecuencia de valores contiguos de un contenedor (figura 151). El constructor necesita el elemento inicial y final de la subsecuencia. El iterador de SubSequence es de acceso directo si el contenedor padre también lo es.

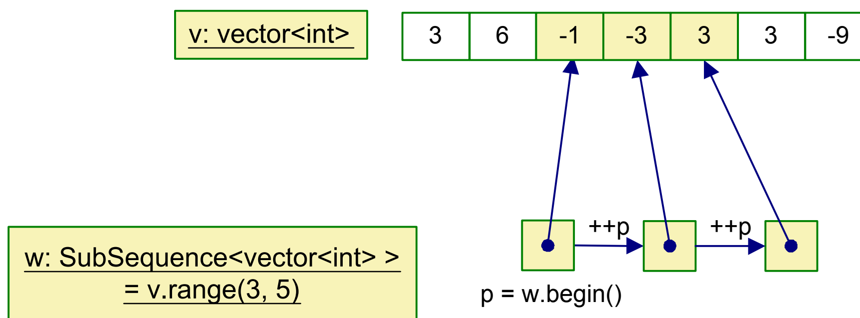


Figura 151. SubSequence

MapSequence permite ver los elementos de un contenedor aplicándoles una función a cada uno de ellos (figura 152). El iterador de este contenedor se implementa como un iterador del contenedor madre al que se le ha modificado el operador * para obtener un elemento aplicándole la función que se le haya

asociado al contenedor. Map es una vista que tiene también muchas aplicaciones. Por ejemplo, dado un vector de personas, map va a permitir obtener una vista con los nombres de dichas personas. Así, para convertir en mayúsculas los nombres de las personas se haría lo siguiente:

```
void toUpper(string& s) {
 //Pone s en mayúsculas
}
...
v.map(Persona::nombre()).for_each(ptr_fun(toUpper)));
```

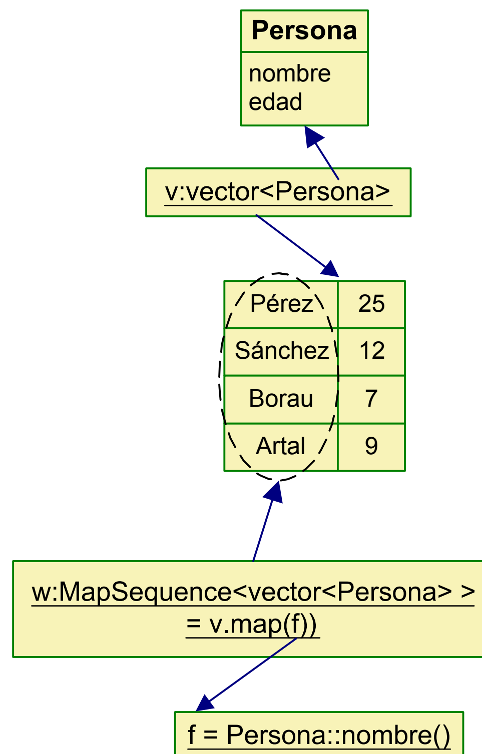


Figura 152. MapSequence

IndirectSequence permite ver el contenedor haciendo una permutación de sus elementos. Internamente guarda un vector de iteradores (figura 153). IndirectSequence es de acceso directo si su contenedor también lo es. El método sortIndirect de Sequence devuelve un IndirectSequence que es una vista ordenada de los elementos del contenedor.

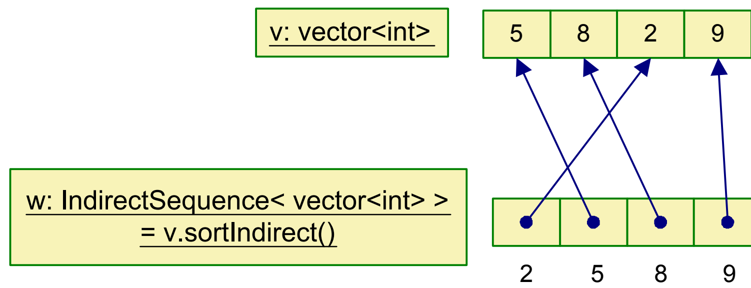


Figura 153. IndirectSequence

FlatSequence opera sobre una secuencia cuyos elementos son a su vez secuencias. Obtiene una secuencia única formada por la unión de todas las secuencias (figura 154).

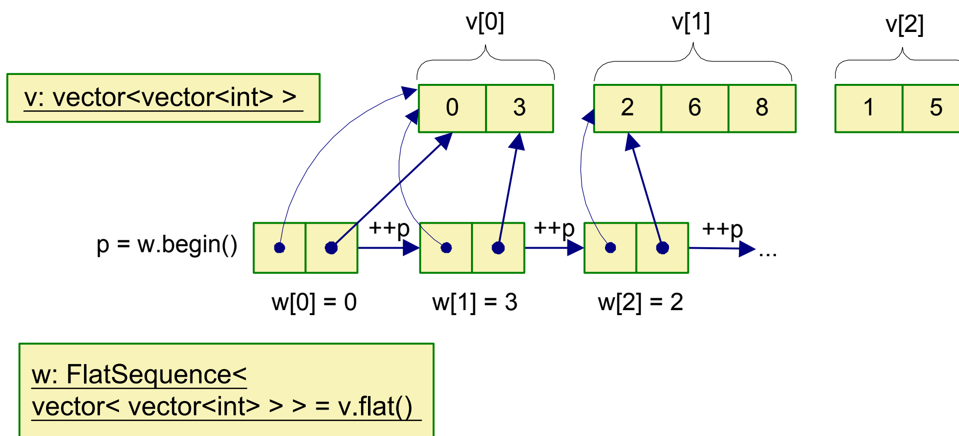


Figura 154. FlatSequence

El iterador de FlatSequence almacena un iterador “grande” a la componente del vector que mira y otro iterador “pequeño” a una subcomponente de esta componente. Cuando el iterador pequeño llega al final se avanza el iterador grande una posición y se pone el iterador pequeño al inicio de esta nueva componente.

Por último ReverseSequence ve los elementos de un contenedor en orden inverso (figura 155). Su implementación es sencilla: el operador de avance se

implementa mediante el operador de retroceso del iterador del contenedor visto.

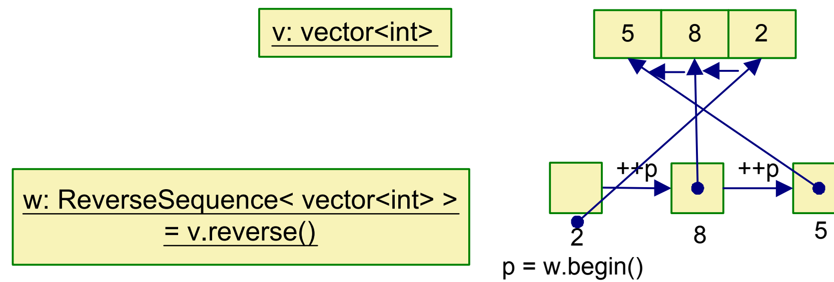


Figura 155. ReverseSequence

Todos los contenedores vista se implementan mediante un contenedor básico (del tipo de los de STL) que se utiliza luego como instancia de Sequence para dotarle de todas las operaciones restantes. Por ejemplo, para implementar SelectSequence se crea una clase ligera SelectSequenceLight con los métodos básicos y el iterador. Sequence se instancia con esta clase para obtener SelectSequence (figura 156).

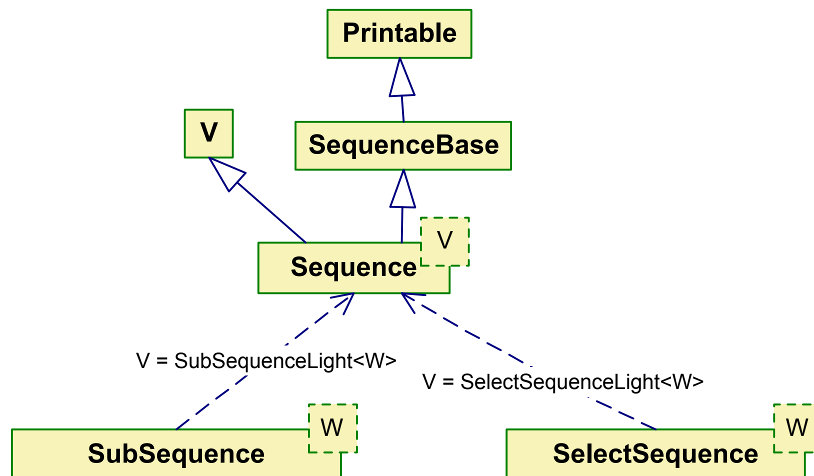


Figura 156. Vistas como instancias de Sequence

Como una vista es un contenedor normal puede ser a su vez el contenedor madre de otra vista. Por ejemplo, si se quiere acceder a los dos primeros valores de un contenedor que son menores que 0 se puede hacer:

```
v.select(less_than(0)).range(1, 2)
```

4.2.3. Contenedores virtuales

Las vistas permiten crear contenedores cuyos datos figuran en otro contenedor. Se puede ir un paso más allá y crear contenedores cuyos datos no existan realmente en ningún lugar sino que se vayan generando dinámicamente conforme se necesiten (de modo análogo a los generadores de CLOS). En la librería se incluyen una serie de estas clases contenedoras. [Koenig 96] utiliza esta idea para construir iteradores aritméticos que van proporcionando valores según una progresión aritmética. Como en el caso de las vistas, los contenedores virtuales (ConstantSequence, etc.) se implementan mediante la creación de clases ligeras (ConstantSequenceLight, etc.) que se pasan como parámetro a Sequence (figura 157).

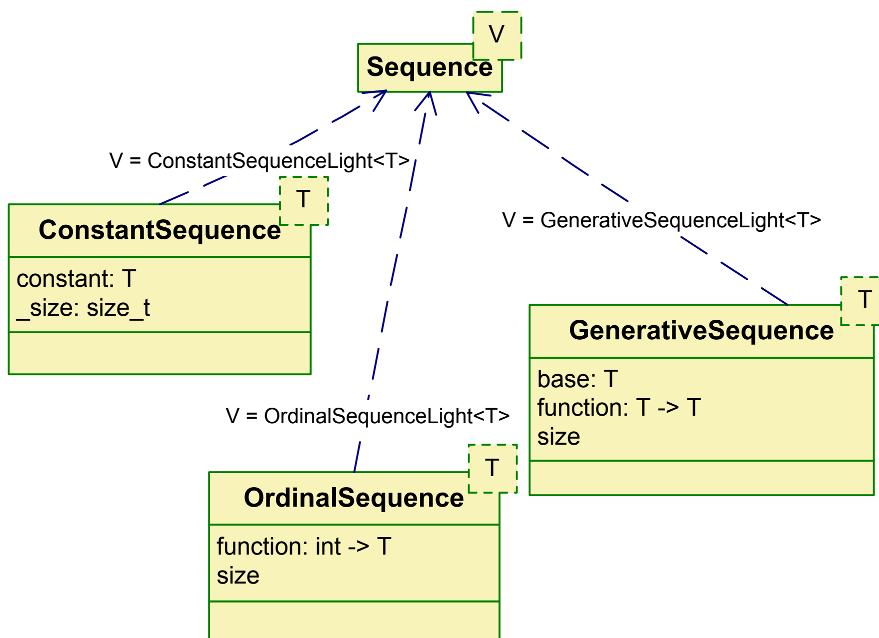


Figura 157. Contenedores virtuales

El contenedor ConstantSequence toma un valor constante y un número de elementos en su constructor. El valor de cada componente es precisamente ese valor constante. Naturalmente no se crea un vector para guardar estos valores repetidos. Simplemente se construye un iterador que al desreferenciarlo devuelve siempre el mismo valor. El iterador guarda un valor con el número de componente a la que apunta para saber cuando se llega al final. La figura 158 muestra un ejemplo de uso en el que se asignan cinco unos al vector v.

```
Vector<string> v;
v = constantSequence(1, 5);
cout << v; //Escribe 1 1 1 1 1
```

Figura 158. Uso del contenedor constante

OrdinalSequence tiene dos parámetros en su constructor: una función f y el tamaño n. El contenedor virtual que representa viene dado por:

{f(1), f(2), ... , f(n)}

Un vector que contiene 10 puntos de la gráfica de x^2 (correspondientes a los valores 1, 2, ..., 10) puede construirse así:

```
Vector<double> v = ordinalSequence(square(), 10);
```

Del mismo estilo es GenerativeSequence. Admite los mismos parámetros más un valor semilla s y el contenedor que representa viene dado por:

{s, f(s), f(f(s)), ... , f^n(s)}

Como ejemplo de uso de GenerativeSequence la siguiente instrucción construye un vector de 20 números aleatorios mediante la función rnd:

```
Vector<double> v = generativeSequence(0.123, rnd(), 20);
```

La secuencia -6, -4, -2, 0, 2, 4, 6 se puede obtener así:

```
generativeSequence(-6, bind2nd(plus<int>(), 2), 7)
```

y también así:

```
int f(int x) {return 2 * x - 8;}
ordinalSequence(f, 7)
```

Si ahora se desean los puntos de la gráfica de x^2 para los siete valores anteriores se puede hacer:

```
ordinalSequence(f, 7).map(square);
```

es decir, a cada elemento de la secuencia se le aplica la función cuadrática.

Cada contenedor virtual tiene una versión en la que el número de componentes es ilimitado. Al constructor no se le pasa el tamaño, y los métodos en los que se necesita dicho tamaño o los que recorren el contenedor quedan sin definir. Se usan como parámetro derecho de una instrucción de asignación. Se asignan al vector tantos valores del contenedor virtual ilimitado como componentes tenga dicho vector. Por ejemplo, si se quiere cambiar en un vector todos los unos que haya por ceros se puede hacer:

```
v.select(equal_to<int>(1)) = constantSequence(0);
```

La función constantSequence está sobrecargada. Cuando se omite el número de elementos devuelve un contenedor virtual ilimitado.

El nombre de las clases virtuales ilimitadas se forma añadiendo el sufijo Unlimited. Por ejemplo ConstantSequenceUnlimited.

4.2.4. Posibilidades de implementación de expresiones funcionales

Se ha visto en apartados anteriores la complejidad de la notación usada por STL para indicar dinámicamente una función. En el capítulo 1 se vio la técnica usada por [Veldhuizen 95b] para utilizar expresiones que trabajen con funciones usando la misma notación que la empleada en expresiones aritméticas. En la librería se intentó usar esta técnica para implementar expresiones en dos variables aunque no se llegó a hacerla operativa por los problemas de los compiladores a la hora de compilar plantillas de expresiones. Para seleccionar en un vector los valores comprendidos en el rango 1..10 se haría:

```
v.select(xInt < 1 || xInt > 10)
```

donde xInt representa la indeterminada en x de tipo entero.

Toda expresión pertenece a una instancia de la plantilla Expr<Func, Inds> donde Func es una clase de funtor e Inds una clase que representa las indeterminadas que aparecen en la expresión. Así, xInt < 1 pertenece a Expr<...,

TagX> y xInt < yInt pertenece a Expr<..., TagXY>. Cuando se combinan dos expresiones es fácil calcular en tiempo de compilación las indeterminadas de la composición como unión de las indeterminadas de los operandos.

En [Jaakko 99] y [Powell 00] se muestran dos implementaciones de expresiones funcionales. [Jaakko 99] es una implementación muy completa que desgraciadamente se encuentra todavía en fase alfa. Si se convierte en una librería operativa podría incorporarse a la librería de facets.

4.3. Aplicación a la librería de facets.

En este apartado se va a ver cómo se han usado los contenedores que se han acaban de ver en la construcción de la librería de facets.

4.3.1. Contenedores en memoria

Los contenedores que se acaban de ver son útiles para la implementación de la librería de facets. Considérese el servicio instances() que devuelve todas las tuplas de una asociación en forma de vector de pares. Si el almacenamiento de las tuplas es local a la asociación la implementación de este servicio es trivial. El problema ocurre cuando esta información está distribuida por los objetos de las clases (figura 159).

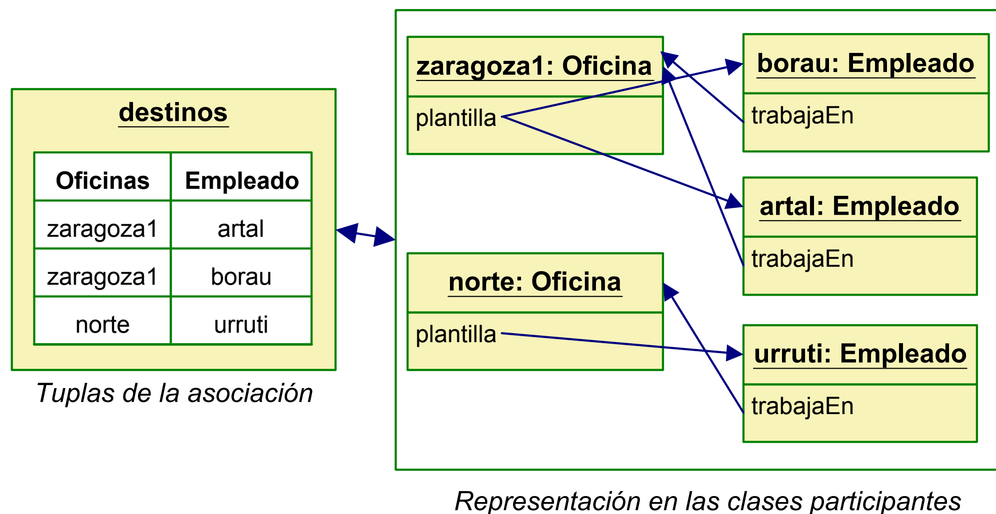


Figura 159. Almacenamiento en las clases

En el caso de asociaciones 1 a 1 o 1 a n vamos a ver cómo se puede crear un vector vista de todas las tuplas.

El siguiente método privado de las asociaciones 1 a n toma como entrada un valor de la clase derecha y devuelve un par (izda, dcha) donde izda es el objeto asociado a dcha (figura 160).

```
pair<const Left&, const Right&> proyRight(const Right& e) {
    return pair<const Left&, const Right&>(getRight(e), e); }
```

Figura 160. Método que adjunta a un empleado su oficina

Utilizando la metainformación de las clases se puede obtener la lista de objetos de la clase derecha así:

```
v = Right::sClassInfo().objects();
```

Si a cada componente de este vector la aplicamos el método proyRight se obtendrá la lista de pares deseada. Por tanto hay que hacer un map (figura 161).

```
MapSequence<...> instances() {
    return
        Right::sClassInfo().objects().map(bind1st(proyRight, this));
}
```

Figura 161. Método que devuelve la lista de pares

Nótese que proyRight, al ser un método, se puede considerar como una función de dos parámetros, la asociación y el objeto right. Con bind1st se fija la asociación quedando como una función de un parámetro. La secuencia devuelta es un contenedor de acceso directo.

En el caso de asociaciones n a n el proceso es algo más complicado debido a que cada elemento tiene asociado un vector de elementos. En primer lugar se

crea un método privado de la asociación que toma un elemento x de la clase derecha y devuelve un vector de tuplas [(a1, x), (a2, x) ... (an, x)] donde [a1, a2, ... , an] son los objetos relacionados con x. Para ello se construye este vector con getRight(x) y a continuación se le aplica un map a cada elemento y mediante una función que devuelve el par (y, x) con x fijo (figura 162).

```
MapSequence<...> proyRight(const Right& x) {
    getRight(x).map(bind2nd(make_pair, x));
}
```

Figura 162. Método que adjunta a un empleado todas sus oficinas

Para cada objeto derecho se obtiene la lista de pares en los que aparece dicho objeto. Si hacemos un map para todos los objetos de la derecha:

```
Right::sClassInfo().objects().map(bind1st(proyRight, this))
```

se obtiene una lista en la que cada elemento es una lista de pares. Fusionando todos estos pares se obtiene la lista deseada (figura 163).

```
FlatSequence<...> instances() {
    return Right::sClassInfo().objects().map(
        bind1st(proyRight, this)).flat();
}
```

Figura 163. Lista virtual de instancias en asociaciones n a n.

Estos ejemplos demuestran el resultado de aplicar en C++ técnicas habituales de Lisp de programación funcional con listas. Aunque la filosofía es muy parecida la notación empleada es mucho más farragosa y la detección de errores es más difícil.

4.3.2. Contenedores en base de datos

En [Zarazaga 00, cap. 3] se presenta el desarrollo de un mecanismo para proporcionar persistencia a objetos sobre una base de datos relacional mediante la utilización de los recursos de metainformación que se han presentado en capítulos anteriores de esta tesis. Este mecanismo permite dotar automáticamente a los objetos de servicios para guardar un objeto en la base de datos (*save*), recuperarlo de la base de datos a memoria (*load*), modificarlo (*update*) o borrarlo de la base de datos (*erase*). Para poder realizar todas estas operaciones es necesario conocer la identidad del objeto (en la aproximación que se referencia esta identidad se consigue mediante la determinación de uno o más atributos del objeto como atributos clave). Sin embargo, surgen situaciones donde no conocemos la identidad exacta del objeto persistente que se desea recuperar o ni siquiera nos interesa conocerla. En ocasiones, atendiendo a motivaciones dinámicas que se presentan durante la ejecución del software de la aplicación, se requiere mostrar al usuario un conjunto de objetos que cumplen ciertas características o realizar cierto procesamiento sobre ellos (por ejemplo cuando se procede a la visualización de tablas con informaciones de numerosas instancias de una clase).

En este apartado se va a proceder a presentar un sistema de gestión de colecciones de objetos de una misma clase que se encuentran almacenados en una tabla de base de datos de acuerdo a la arquitectura que se acaba de citar. Estos contenedores van a ser compatibles con los de STL y por tanto van a poder ser parámetros de Sequence. Su implementación se va a realizar descargando en los servicios de persistencia explicados en [Zarazaga 00] la mayor parte de la funcionalidad de acceso a la base de datos. Estos servicios de persistencia obtienen de la metainformación del sistema los datos de las tablas, nombres de las columnas, tipos de estas, etc. necesarios para construir las sentencias de recuperación de información de la base de datos (sentencias SELECT de SQL 92). Una vez construida la sentencia SQL es necesario comunicarse con la base de datos. La figura 164 muestra las clases que encapsulan este acceso haciendo abstracción del servidor de base de datos que se utilice. Basic_SQL_Expert contiene los servicios comunes a todas las clases, por ejemplo execute(sql_string) para enviar una instrucción SQL a la base de datos. De esta deriva la plantilla SQL_Expert<TObject>, que está parametrizada por una clase del modelo y permite implementar servicios dependientes de la clase

con cuya tabla se va a trabajar. El método fundamental que va a usar el sistema de contenedores es

select(String sql_string, vector<T>& v, int firstRow, int numRows)

Este método envía al motor de la base de datos la instrucción SELECT contenida en sql_string. De la vista obtenida se queda con numRows filas partiendo de la fila número firstRow.

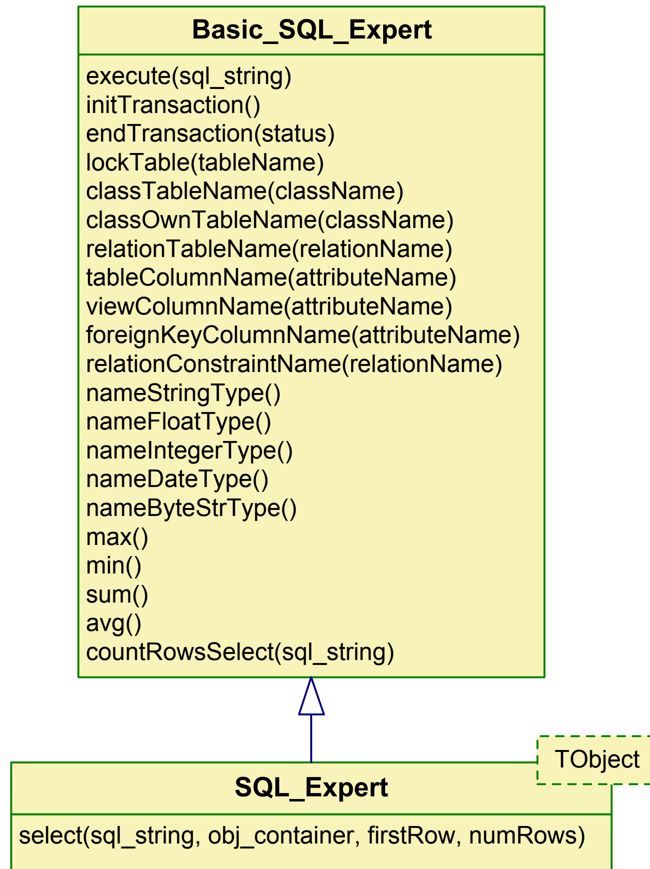


Figura 164. Clases que encapsulan el acceso a la base de datos

La jerarquía de clases construida que permite el manejo conjunto de grupos de objetos del modelo es la que aparece en la figura 165.

La plantilla DBContainer es el contenedor básico para trabajar con objetos persistentes. Está parametrizada por el tipo de datos que contiene y que deberá ser una clase persistente (si no lo es se detecta en tiempo de compilación). Dispone de los servicios de los contenedores básicos de STL y además incluye unos servicios adicionales que pueden implementarse fácilmente interrogando a la base de datos.

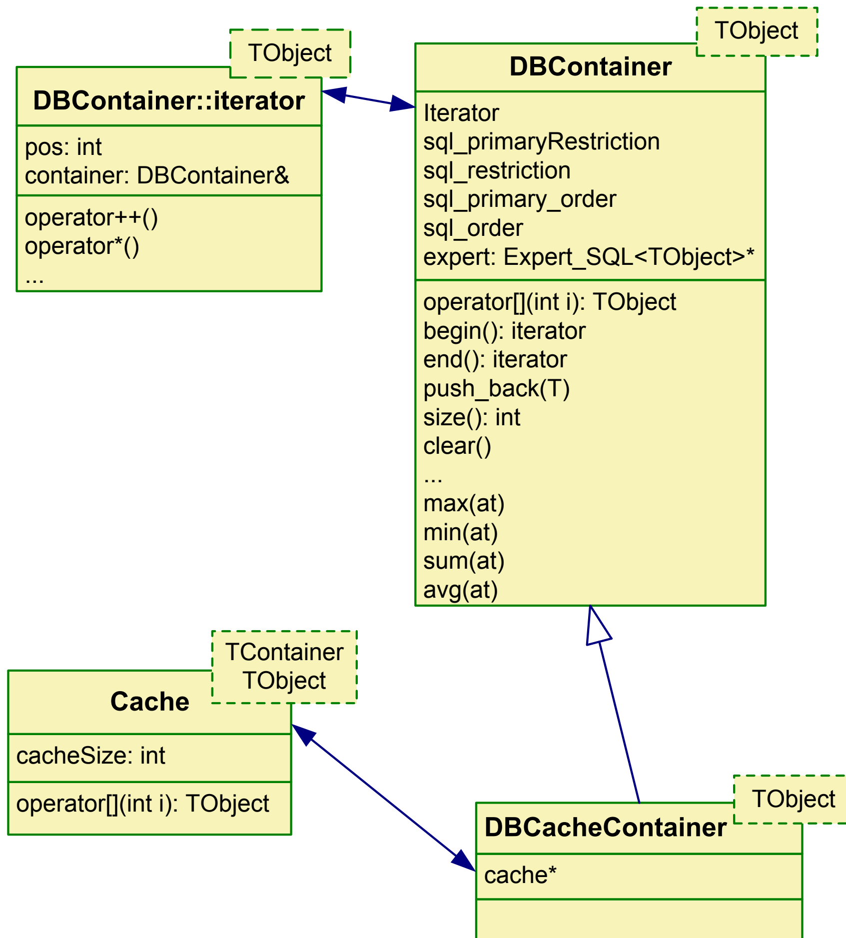


Figura 165. Jerarquía parcial de contenedores

En primer lugar se puede restringir el conjunto de objetos que va a contener mediante dos cadenas que van a expresar la condición de filtro:

- String _sql_primaryRestriction

  Restricción primaria sobre los objetos del contenedor y no modificable fuera del constructor de la clase.

- String _sql_restriction

  Restricciones secundarias añadidas a la restricción primaria. Se puede reemplazar o incrementar esta restricción una vez construido el contenedor. Posibilita que el contenido del contenedor sea acotado en sucesivos pasos.

Por otro lado, se puede acceder a los datos del contenedor según el orden que se desee y que va a estar determinado por los siguientes campos:

- String _sql_primaryOrder

  Permite establecer un orden primario sobre los objetos seleccionados a traer desde la base de datos y no es modificable fuera del constructor de la clase.

- String _sql_order

  Permite establecer un orden secundario dentro de los objetos seleccionados a traer desde la base de datos.

Por último, dentro de la clase DBContainer se añaden métodos para realizar operaciones matemáticas sobre el conjunto de los valores de un atributo de los objetos contenidos. Estos servicios son max(at), min(at), avg(at) y sum(at) que devuelven respectivamente el máximo, el mínimo, la media y la suma de los valores del atributo indicado por at. Todos estos servicios están parametrizados por el tipo del atributo.

Para observar el funcionamiento de la clase DBContainer, se detalla a continuación la secuencia de eventos que se producen cuando un objeto de tipo DBContainer intenta recuperar un objeto persistente que estaba guardado en la base de datos relacional. Para ello, se muestra primero un pequeño ejemplo de código de una aplicación cualquiera donde surge esta situación (figura 166).

```
String restrictionPrimaria("edad > 20");
DBContainer<Persona> v(restriccionPrimaria);
for ( int i = 0; i< v.size(); ++i)
     cout << v[i];
```

Figura 166. Ejemplo de uso de un DBContainer

En primer lugar, se ha declarado el objeto v, el cual va a contener los objetos de la clase Persona que cumplan la restricción primaria indicada. A continuación, se accede dentro de un bucle a cada uno de los objetos del contenedor para realizar cierto procesamiento (mostrarlo por pantalla). Al hacer la llamada v[i] es cuando realmente se interacciona con el SGBDR ya que la implementación del operador [] consiste realmente en una llamada a la función select perteneciente a la especialización para un SGBDR concreto de la clase SQL_Expert<Tobject> (figura 167).

```
template<class TObject> inline
TObject DBContainer<TObject>::operator[](int i)
{    ......
     Vector<TObject*> contObj;
     expert().select(_selectStr(), contObj, i, 1);
     return (*contObj[0]);
}
```

Figura 167. Implementación del acceso a un elemento

En cuanto a los iteradores, su construcción y funcionamiento es elemental: guardan la posición pos del elemento al que están apuntado y el contenedor v que recorren. La indirección se implementa de modo trivial como v[i] y el paso al siguiente elemento como ++pos.

La figura 168 muestra la interacción que se produce con la base de datos al realizar la llamada a select.

En esta figura, se puede observar la interacción por medio de SQL de los

dos procesos participantes en el sistema: el del servidor del SGBDR y el de la aplicación. La secuencia de eventos que ocurren al llamar a select es la siguiente:

1. La base de datos relacional determina qué pagina en disco (o dispositivo de almacenamiento estático) contiene la tabla (o tablas) donde se encuentran las filas correspondientes a los objetos del contenedor.

2. La base de datos lee la página de disco y la copia dentro de su espacio de direcciones de la memoria principal trayéndola a su propio buffer de memoria principal.

3. Como la aplicación no puede acceder directamente al *buffer* de la base de datos, la base de datos se encarga de traer el registro (fila) solicitado a la memoria de la aplicación. Esto se realiza gracias a las sentencias SQL fetch incluidas en la implementación de select.

4. Finalmente, se convierten los valores de los campos de las filas en los valores de los atributos del objeto, con la transformación de tipos correspondiente. Con esto ya se ha recuperado el objeto en la memoria de la aplicación.

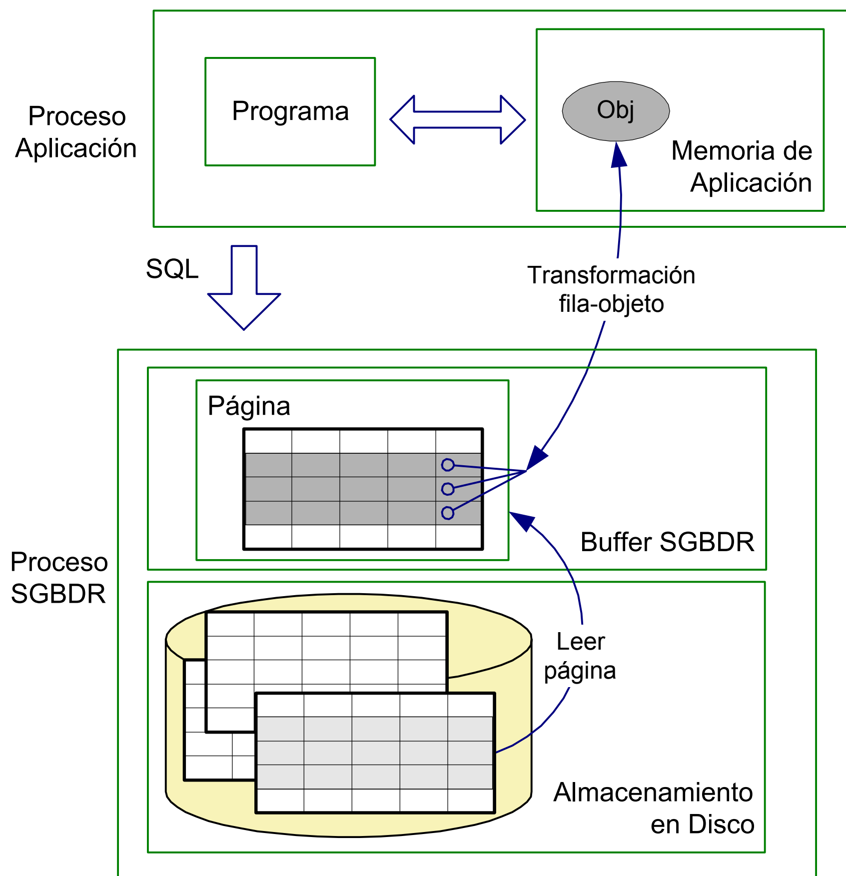


Figura 168. Recuperación usando un DBContainer

La clase DBContainer consigue el objetivo de definir el conjunto de objetos sobre el cual la aplicación va a realizar un procesamiento. Sin embargo, carece de eficiencia si se observa detenidamente la forma de traer los objetos a memoria. Cada vez que se utiliza el operador v[i] nos traemos una fila de la base de datos relacional a la memoria de la aplicación. Este hecho se puede observar en la implementación del operador [] el cual hace una llamada a select poniendo 1 como número de filas a traer. Cada vez que la aplicación acceda a un objeto del contenedor con v[i] o con *p se interacciona con el proceso del SGBDR, produciéndose un acceso a disco si la tabla que contiene la fila requerida ya no se encuentra dentro del buffer en memoria de la base de datos.

Para solucionar este problema de eficiencia se introduce la clase DBCache-Container<TObject>. Este contenedor utiliza la técnica clásica de caché me-

diante la utilización de un buffer implementado por la clase Cache<TContainer, TObject>. El tamaño de esta caché se guarda en el campo local cache_size.

Cuando se llama al operador [] para obtener un objeto del contenedor no sólo se transfiere ese objeto desde la base de datos a la caché (memoria de nuestra aplicación) sino que también se transfiere a la caché el segmento de objetos dentro del cual se incluye el objeto que necesitamos. Se adopta una política de *localidad en referencia,* es decir, el próximo acceso a un objeto del contenedor será probablemente al vecino anterior o posterior.

Cuando se vuelva a llamar al operador [] para acceder a otro objeto se mirará en primer lugar si dicho objeto se encuentra ya en la caché. Si es el caso se devuelve directamente sin necesidad de tener que interactuar con la base de datos. En caso contrario habrá que refrescar la caché mediante una petición a la base de datos.

La figura 169 muestra interacción que se produce entre la base de datos y el proceso de la aplicación utilizando este nuevo tipo de contenedor al utilizar el operador [].

Como se puede observar, ahora se transfiere el máximo numero posible de filas (limitado por el tamaño de la cache) desde la base de datos a la memoria de la aplicación. Sucesivos accesos a objetos próximos al primer objeto al que se ha accedido no requerirán interacción con la base de datos.

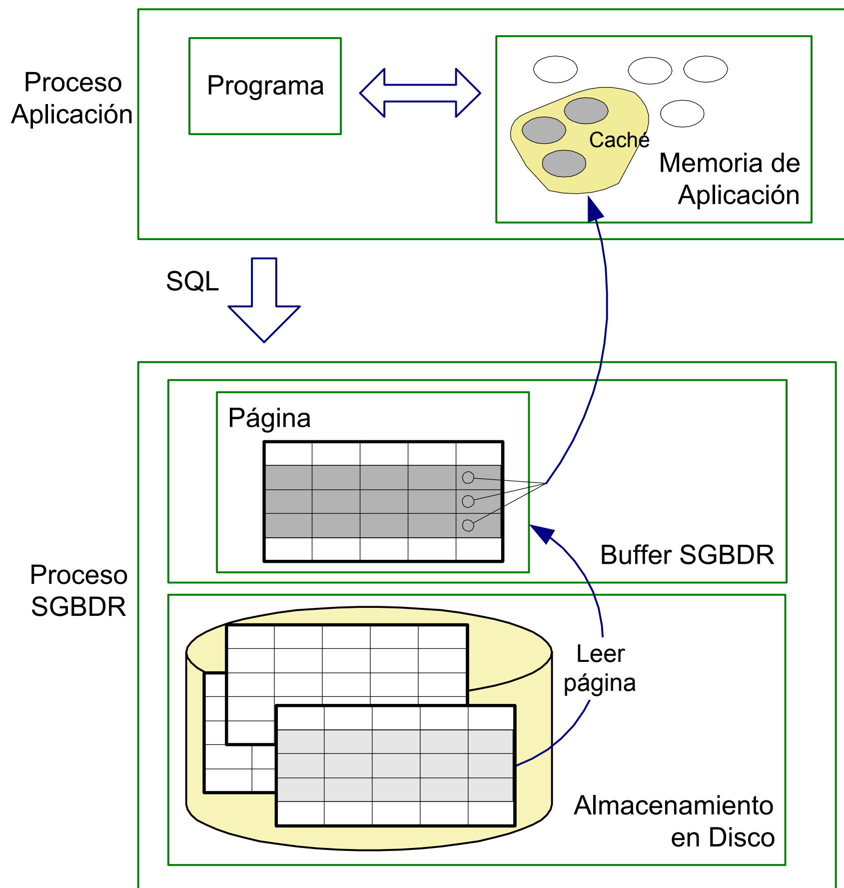


Figura 169. Recuperación usando un DBCacheContainer

4.4. Conclusiones

En este capítulo se ha presentado una ampliación de los contenedores proporcionados por el estándar C++, buscando cubrir aquellas limitaciones que presentan para el desarrollo de la librería de facets.

En primer lugar se ha visto cómo las capacidades de evaluación en tiempo de compilación de las plantillas permiten realizar programación genérica en C++. Este tipo de programación consigue que los algoritmos que se desarrollen se puedan aplicar al conjunto más amplio de estructuras posibles sin pérdida de eficiencia. La utilización más clara de la programación genérica se encuentra a

la hora de desarrollar una librería de contenedores: debido a que la esencia de un algoritmo es independiente del contenedor particular con el que trabaje es posible parametrizar dicho algoritmo por el contenedor. Esta aplicabilidad general de los algoritmos hace que, al introducir un nuevo contenedor, automáticamente queden implementados los algoritmos cuyas exigencias cumpla dicho contenedor. Del mismo modo, los nuevos algoritmos que se introduzcan se podrán aplicar a todos los contenedores existentes que satisfagan sus requisitos. Los costes de desarrollo disminuyen, se evitan redundancias estructurales en el código y la separación de conceptos amplía las posibilidades de reutilización de código. Seguidamente se ha hecho un pequeño análisis de la aplicación de estos principios en el diseño de la librería estándar de contenedores de C++. A continuación se han señalado algunas deficiencias de esta librería y se ha propuesto una ampliación de la misma para hacer cómodo y seguro el trabajo con contenedores a la hora de trabajar con colecciones en la librería de facets. Estas mejoras no quedan únicamente localizadas en esta librería sino que tienen una aplicabilidad más general. Además, la arquitectura propuesta para dar cabida a estas mejoras permite añadir fácilmente nuevos modelos de contenedor: cada vez que se quiere integrar un nuevo contenedor únicamente hay que implementar los servicios básicos de dicho contenedor ya que una función de tipos permite crear automáticamente, a partir de ese contenedor ligero, uno más potente con una gran cantidad de servicios. La nueva librería de contenedores se aprovecha y hace uso de la estándar ya que todos los contenedores estándar pueden ser parámetro de esta función de tipos.

Además, para dotarla de mayores capacidades expresivas se incluyen una serie de contenedores virtuales muy útiles que hacen el código más compacto. En muchos casos, estos contenedores sirven para mejorar la eficiencia de los programas ya que ahorran el coste de construir físicamente el contenedor que imitan porque, sobre una organización determinada de los datos, construyen una vista que accede a ellos bajo un prisma distinto.

A continuación se ha visto cómo estos contenedores se han podido utilizar para implementar ciertos servicios de la librería de facets en relación con el manejo de asociaciones múltiples que, en caso de no disponer de ellos, no habrían podido implementarse sin una disminución muy importante de la eficiencia.

Por último se ha presentado un contenedor para trabajar con datos que se almacenan físicamente en memoria secundaria. El problema de eficiencia que se deriva de realizar múltiples accesos individuales a la base de datos se solu-

ciona mediante la introducción de un nuevo contenedor que utiliza una política de caché en memoria mediante un buffer de almacenamiento local. Debido a que se han implementado en estos contenedores los servicios básicos de STL su uso es similar al de los contenedores de datos en memoria.

En conjunto, la ampliación de la librería de contenedores libera al programador de la mayoría de los detalles técnicos en relación con su manejo, permitiéndole centrarse en el problema real. El resultado final permite trabajar con contenedores casi tan agradable y cómodamente como con las listas en Lisp, aunque la notación en algunos casos no resulta tan elegante. En particular, la ausencia de funciones anidadas en C++ (funciones declaradas dentro de otra función) obliga a desarrollar una tecnología compleja para permitir construir funciones por medio de expresiones. Estas expresiones funcionales son necesarias ya que muchos algoritmos tienen un parámetro que representa una acción o función. La indicación de este parámetro por medio de una expresión funcional evita tener que declarar la función expresamente y además en un lugar lejano ya que las funciones no se pueden anidar.

Capítulo 5

Conclusiones

En este trabajo se ha enriquecido el lenguaje de programación C++ incrementando la capacidad expresiva de sus objetos miembro (atributos) y sus capacidades de manejo de tipos (RTTI). La experiencia en el uso de un lenguaje como Lisp y su potencia en cuanto a sus capacidades de autoconocimiento y de manejo de listas ha dirigido la investigación hacia dos campos de actuación fundamentales: la dotación al lenguaje de estructuras que permitan a un programa obtener información sobre sus propios y la creación de un conjunto de estructuras contenedoras potentes y de fácil uso. Tres principios fundamentales se han intentado respetar a la hora de perseguir estos objetivos: mantenimiento de la eficiencia, comprobación estática de tipos y uso cómodo y amigable de los elementos que se añadan.

En primer lugar se han visto las importantes ventajas que brindan las capacidades reflexivas y de metainformación del lenguaje para el desarrollo de aplicaciones. Se ha repasado la literatura existente y se han analizado las capacidades de los distintos lenguajes en este sentido. Se ha visto que en el caso de C++ el soporte para la metainformación es mínimo y se han estudiado diferentes aproximaciones para solucionar esta carencia. Se ha visto también las posibilidades potenciales que tienen las plantillas para la metainformación.

A continuación se ha presentado el núcleo central de una librería llamada librería de facets para dotar a los programas de metainformación. Esta metainformación se consigue mediante la creación de una serie de objetos a los que se les puede preguntar información sobre los diferentes componentes estructurales del programa como las clases, los atributos y los métodos. En el caso de los atributos se ha introducido un nuevo patrón en el lenguaje llamado atributo enriquecido que permite asociar a los atributos cualquier clase de metainformación. Se han visto las ventajas derivadas de tener acceso a esta información a la hora de integrar los diferentes componentes de un programa (código, inter-

faz gráfica, tablas de bases de datos) ya que todos estos componentes normalmente necesitan información de la estructura de las clases, información que habría que reintroducir manualmente si no se dispusiera de medios para acceder a ella. La librería de facets respeta los principios indicados más arriba: la eficiencia y comprobación estática se ha conseguido gracias a las plantillas. La facilidad de uso se ha logrado mediante un sistema sencillo de macros del preprocesador. En cualquier caso, la solución se ha encontrado dentro del marco del propio lenguaje sin tener que recurrir a herramientas más sofisticadas como precompiladores lo que iría en contra del objetivo de simplicidad y facilidad de uso. También se han comprobado las capacidades de reutilización de la librería gracias a su carácter extensible. El usuario de la librería puede ampliarla mediante la creación de nuevas clases que permiten personalizar la información que se asocia a las clases y mediante la creación de clases personalizadas para enriquecer la información que contienen los atributos. Estas capacidades de ampliación utilizan una aproximación orientada a objetos con las ventajas subsiguientes: el usuario extiende las posibilidades del sistema mediante la introducción de clases y jerarquías de clases que se integran en el producto sin afectar al resto de elementos del sistema. Las clases pueden reutilizarse para crear nuevos componentes y para desarrollar aplicaciones finales.

Seguidamente se ha mostrado el uso de la arquitectura de facets para ampliar la librería con estructuras que van a dar soporte directo a las asociaciones entre clases. Se ha aprovechado la propiedad de los atributos enriquecidos de disponer de un lugar donde colocar metainformación para colocar ahí los datos de la asociación. Se han introducido clases que permiten trabajar con asociaciones en memoria con o sin atributos y con diferentes cardinalidades. También se ha vuelto a poner de manifiesto las capacidades de expansión de la librería: para incorporar nuevas asociaciones al sistema basta con declarar una clase que cumpla una serie de requisitos y automáticamente quedará integrada en el sistema. Además se ha visto cómo el soporte para herencia de la librería de facets permite implementar asociaciones que se especializan en clases hijas de dos que estén asociadas. El hecho de poder trabajar con las asociaciones como elementos directos del lenguaje va a permitir la disminución de redundancias en el código facilitando su mantenimiento y reutilización. Además, puesto que las asociaciones son elementos que forman parte del diseño de la aplicación, el coste del paso de la fase de diseño a la fase de implementación va a disminuir notablemente debido a esta correspondencia directa entre elementos de diseño y los elementos del lenguaje.

El uso de las asociaciones en la librería ha hecho necesaria la implementación de una familia de contenedores de uso sencillo y de gran potencia expresiva cuyo diseño se ha basado en la librería estándar de C++. Esta familia de contenedores corrige algunas deficiencias de la librería estándar mediante la introducción de unas plantillas de clases que permiten usar los contenedores de modo uniforme y minimizando los riesgos de que se produzcan errores en su manejo. Unos contenedores adicionales se han introducido para poder trabajar con colecciones de datos persistentes. El desarrollo de estos componentes, al igual que el de la librería estándar, se ha basado en un paradigma reciente de programación llamado programación genérica que está resultando muy productivo.

Uno de los objetivos que se ha perseguido en la construcción de la librería de facets ha sido la obtención de unos resultados que pudiesen ser aplicables industrialmente. Esto ha quedado contrastado con la utilización de esta librería para el desarrollo de una aplicación industrial. El producto desarrollado es un sistema de gestión para una red de radiotelefonía móvil basada en tecnología *trunking* (asignación dinámica de recursos). Se ha tratado de un proyecto de tamaño medio/alto en cuyo desarrollo han trabajado un total de nueve personas entre analistas, diseñadores y programadores. La aplicación se encuentra plenamente operativa desde 1998. Actualmente se encuentra implantada en tres redes ubicadas en Gran Canarias, Alicante y Viladecans (Barcelona). Esta prevista su instalación a principios de 2000 conjuntamente con una red *trunking* vendida en Brasil. El sistema se ejecuta sobre Microsoft NT versión 4.0. con Oracle como servidor de bases de datos. El desarrollo se ha hecho con Microsoft Visual C++ versión 4.1. Este compilador fue impuesto por la elección del modo de acceso a la base de datos (se ha utilizado SQL incluido en C++ con el precompilador Oracle Pro C/C++ que garantizaba su generación de código únicamente para este compilador y versión). La ejecución puede realizarse de forma distribuida sin tener que realizar ninguna tarea especial, salvo indicar en los correspondientes ficheros de inicialización los IP's de las máquinas en las que se encuentran corriendo cada uno de los componentes. De igual modo, la configuración permite su ejecución centralizada en un único PC. En este caso, éste debe estar equipado con un procesador equivalente a un Intel Pentium 133 MHz o superior, 64 Mb de RAM, un mínimo de 1 GB de disco duro, 1 puerto serie libre (en muchos casos, uno se encuentra utilizado por el ratón), y una tarjeta gráfica capaz de ofrecer 16 colores a un monitor color de 17" con una resolución mínima de 1152 x 864 pixels (esta resolución es necesaria ya que existen ventanas que muestran una cantidad de información que no podría ser

claramente observada en una resolución menor). En [Zarazaga 98a] se presentan los sistemas *trunking* con más profundidad y se puede observar en detalle la funcionalidad que proporciona la aplicación desarrollada. En [Zarazaga 97] se dan detalles sobre la arquitectura de la misma, mientras que en [Zarazaga 99] se sintetizan las experiencias más interesantes derivadas del desarrollo del proyecto. Finalmente, en [Zarazaga 00, cap. 5] se revisan aspectos del desarrollo de la aplicación vinculados con la utilización de la librería de facets.

El desarrollo de la aplicación ha requerido poco esfuerzo por parte de los programadores finales: a partir del diseño, el paso a la codificación ha resultado ser una tarea bastante automática gracias, en parte, al soporte de las asociaciones como elementos de primer orden. Esta circunstancia ha permitido disminuir notablemente los tiempos de desarrollo. El mayor cuello de botella ha sido ocasionado por el uso generalizado de las plantillas de tipos de C++. La propia naturaleza de las plantillas junto con la deficiente calidad de su implementación por parte del compilador utilizado ha creado problemas de tiempos de compilación excesivos, de explosión de código y de dificultades de depuración debidos a mensajes imprecisos o extremadamente largos. Es de esperar que los nuevos compiladores mejoren en este sentido. Un paso fundamental será el soporte para la compilación separada de plantillas, algo que está previsto en el lenguaje mediante la palabra reservada `export` pero que todavía ningún compilador comercial contempla.

Tanto el desarrollo de la librería como el de las aplicación realizada con ella se han visto perjudicados por los errores del compilador. En bastantes ocasiones el compilador ha indicado errores en porciones de código correctas obligando al programador a un esfuerzo adicional destinado a detectar si el error se debía a algún fallo de programación o al propio compilador. Normalmente, en estos casos lo que se hacía era realizar una prueba en otro compilador diferente (Borland o Sillicon). En este sentido es interesante que el propio diseño de la librería estándar STL (que emplea intensamente las plantillas) contiene algunos errores. [Simonis 00] da una pequeña muestra de estos problemas, surgidos en parte por no haber podido probar la librería durante su diseño debido a las limitaciones de los compiladores de entonces. Hoy en día los compiladores han mejorado sensiblemente en este aspecto lo que redundará sin duda en mejoras importantes en los tiempos de desarrollo.

A pesar de estos problemas, las plantillas se han revelado como una herramienta importante para lograr los objetivos indicados al principio de estas conclusiones. Su potencia para realizar cálculos estáticos, su eficiencia y su respe-

to a la comprobación de tipos han sido determinantes para lograr el objetivo marcado. Su uso ha sido profuso e intenso y ha permitido obtener soluciones respetando los principios básicos de eficiencia, facilidad de uso y comprobación estática de tipos

La programación genérica con plantillas es una disciplina relativamente novedosa. A la vista de los resultados obtenidos en éste y otros trabajos sus capacidades son muy prometedoras y es de esperar un aumento importante de recursos destinados a investigar en este terreno. También es probable que aparezcan nuevos lenguajes en los que la genericidad esté diseñada más cuidadosamente y con objetivos más amplios que los que guiaron en principio su implantación en C++, y que no fueron otros que permitir contenedores parametrizados por el tipo. Esto evitará muchos de los problemas que surgen en C++ cuando se hace un uso avanzado de las plantillas y permitirá desarrollar compiladores más robustos con menos esfuerzo. Los tiempos de compilación y el tamaño de código también deberán beneficiarse de un diseño más coherente.

Se indican a continuación posibles líneas de investigación futuras en las que ampliar o desarrollar el trabajo aquí presentado.

- Integración de la librería de facets con sistemas distribuidos como CORBA. Este tipo de sistemas de objetos distribuidos utilizan gran volumen de información sobre comunicaciones tal y como se ha visto en trabajos desarrollados dentro de nuestro grupo de trabajo, como [Comella 98] y [Zarazaga 00b]. Esta información podría ser exportada fuera de los objetos mediante sistemas de facets para su aprovechamiento en otras áreas.

- Creación de un sistema de persistencia por medio de *streams*. La librería proporciona persistencia en bases de datos relacionales. La metainformación de los atributos y las clases podría usarse para implementar un sistema de escritura/lectura en *streams* de modo análogo a como se hace en Java. Yendo un paso más allá, se puede pensar en el diseño de intérpretes reducidos del lenguaje gracias a la metainformación de objetos, clases, atributos y métodos. Gracias a ello se podría procesar un fichero con instrucciones tanto para crear objetos como para realizar acciones con ellos.

- Traslado de la tecnología de facets a otros lenguajes. En particular se está pensando en Java. La naturaleza de esta tecnología, altamente depen-

diente de las capacidades de genericidad del lenguaje, obliga a demorar este proyecto hasta que se introduzca en Java su propio sistema de genericidad.

- Mejoras en la librería de contenedores. Estas mejoras consisten en integrar en la librería otros contenedores que no son estrictamente secuenciales, como árboles, tablas hash y matrices. Queda además, como trabajo pendiente, mejorar la eficiencia de algunos algoritmos e implementar de forma completa algunas estructuras que no compilan adecuadamente debido a las limitaciones del compilador.

- Implementación de asociaciones ternarias. En este trabajo, la implementación de las asociaciones se ha centrado en las binarias. La librería de facets podría extenderse para poder trabajar con asociaciones ternarias o de grado superior. Una posible aproximación consistiría en definir tres clases de rol (digamos izquierdo, central y derecho). Cuando un atributo de acceso solicitase una operación a su rol (por ejemplo `a.insert(b,c)` éste la trasladaría a la asociación poniendo la tupla a insertar en el orden adecuado (por ejemplo `relation.insert(b, a, c)` si el rol anterior fuera el central). La implementación de estas operaciones no conllevaría ninguna dificultad especial ya que son meras extensiones de los conceptos empleados en las asociaciones binarias. Más trabajo requeriría la creación de una interfaz adecuada para obtener objetos asociados ya que se puede pensar en obtener todos los pares `(b, c)` asociados con un elemento `a` así como todos los elementos `c` asociados a un par dado `(a, b)`. En cuanto al almacenamiento de las tuplas, la ubicación más apropiada podría ser dentro de la propia asociación en forma de lista de triples `(a, b, c)`. Para trabajar con asociaciones de cardinalidades superiores se utilizarían criterios similares utilizando objetos contenedores para representar las tuplas que se utilizasen como parámetros de los métodos `insert`, `erase`, etc. No obstante, se trata de ideas preliminares que necesitarían un esfuerzo importante de diseño y programación.

Bibliografía

[Alexandrescu 99] A.Alexandrescu: *Better template error messages.* C++ User Journal, Mar. 1999.

[Attardi 93] G.Attardi: *Metaobject programming in CLOS*. The CLOS Perspective, pp. 119-131, MIT Press, 1993.

[Austern 98] M.H.Austern: *Generic Programming and the STL*. Addison-Wesley, 1998.

[Barnes 95] J.G.P. Barnes: *Programming in Ada 95*. Addison-Wesley, 1995.

[Bobrow 77] D.G.Bobrow, T.Winograd: *An Overview of KRL, a Knowledge Represenation Language*. Cognitive Science, Vol. 1, Num. 1, pp. 3-46, 1977.

[Bobrow 83] D.G.Bobrow, M.Stefik: *The LOOPS Manual*. Intelligent Systems Laboratory, Xerox, 1983.

[Bobrow 88] D.G.Bobrow, I.G. DeMichiel, R.P.Gabriel, S. Keene, G.Kizcales, A.D. Moon: *The Common Lisp Object System Specification*. Tech. Doc. 88-002R of x3J13, Jun. 1988.

[Bobrow 93] D.G.Bobrow, R.P.Gabriel, J.L.White: *CLOS in Context*. The CLOS Perspective, chap. 2. MIT Press, 1993.

[Booch 99] G.Booch, J.Rumbaugh, I.Jacobson: *El Lenguaje Unificado de Modelado*, Addison-Wesley Iberoamericana, 1999.

[Borning 87] A.Borning, T.O'Shea: *Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language*. Proceedings of ECOOP'87, num. 276, pp. 1-10, LNCS, Jun. 1987.

[Bouraqadi 98] Bouraqadi-Saâdani, T.Ledoux, F.Rivard: *Safe Metaclass Programming.* Proceedings OOPSLA'98, pp. 84-96, Oct. 1998.

[Bracha 98] G.Bracha, M.Odersky, D.Stontamire, P.Wadler: *Adding Genericity to the Java Programming Language.* OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.

[Brachman 79] R.J.Brachman: *On the Epistemological Status of Semantic Networks*. Associative Networks: Representation and Use of Knowledge by Computers, pp. 3-50. Ed. N.V. Findler, New York: Academic Press, 1979.

[Brachman 85] R.J. Brachman & H.J. Levesque: *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Inc. 1985.

[Braux 99] M.Braux: *Speeding up the Meta-Level Processing of Java Through Partial Evaluation*. ECOOP'99 Workshop on Object Technology for Product-line Architecture, 1999.

[Briot 89] J.P.Briot, P.Cointe: *Programming with Explicit Metaclasses in Smalltalk*. Proceedings of OOSPLA'89, pp. 419-431, New Orleans, USA, Oct. 1989.

[Brun 96] R.Brun, F.Rademakers: *ROOT – An Object Oriented Data Analysis Framework*. Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996.

[Buschmann 92] F.Buschmann, K.Kiefer, F.Paulisch, M.Stal: *A Runtime Type Information System for C++*. Workshop on Object-Oriented Reflection and Metalevel Architectures, ECOOP'95, Ultrecht, Netherlands, Jun. 1992.

[Buschmann 96] F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, M.Stal: *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley, 1996.

[Buschmann 96b] F.Buschmann, P.Sommerlad, M.Stal: *The pattern Reflection*. En Pattern-Oriented Software Architecture: A System of Patterns. John Wiley, 1996.

[C 89] ANSI X3.159-1989. *American National Standard for Information Systems – Programming Language – C.* American National Standards Institute, 1990.

[C++ 98] ISO/IEC 14882:1998(E). *International Standard – Programming Languages – C++*. American National Standards Institute, Jul. 1998.

[Cavarroc 98] J.Cavarroc, S.Moisan, J.P.Rigault: *Simplifying an Extensible Class Library Interface with OpenC++*. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.

[Chiba 95] S.Chiba: *A Metaobject Protocol for C++*. ACM SIGPLAN Notices vol. 30, num. 10, pp. 285-299, Oct. 1995.

[Chiba 98] S.Chiba, M.Tatsubori: *Yet another* java.lang.Class. ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems, 1998.

[Chuang 98] T.R.Chuang, Y.S.Kuo, C.M.Wang: *Non-intrusive Object Introspection in C++. Architecture and application.* Proceedings of the 1998 International Conference on Software Engineering, pp. 312-321, 1998.

[CIDLib 00] Charmedquark, *The CIDLib library.* www.charmedquark.com.

[Cohen 96] S.Cohen: *Lightweight persistence in C++.* www.stat.cmu.edu/~lamj/sigs/c++-report/cppr9605.f.cohen.html. C++Report, May. 1996.

[Cointe 87] P. Cointe: *Metaclasses are First Class: the ObjVlisp Model.* Proceedings of OOPSLA'87, pp. 156-167, Orlando, USA, Oct. 1987.

[Cointe 93] P.Cointe: *CLOS and Smalltalk. A comparation*. The CLOS Perspective, pp. 215-250, MIT Press, 1993.

[Cointe 96] P.Cointe: *Reflective Languages and MetaLevel Architectures*. ACM Computing Surveys, Dic. 1996.

[Coker 97] J.Coker: *Object Persistence and Distribution*. Feb. 1997. http://developer.java.sun.com/developer/technicalArticles/RMI/ObjectPersist/index.html

[Comella 98] S.Comella, J.Ezpeleta, F.J.Zarazaga, **J.Valiño**, P.R.Muro: *Proporcionando capacidades de trabajo concurrente a aplicaciones GIS sobre un entorno distribuido basado en CORBA*. VI Jornadas de Concurrencia, Pamplona, España. Jun. 1998.

[Czarnecki 98] K.Czarnecki: *Generative programming principles and techniques of software engineering based on automated configuration and fragment-based component models.* PhD thesis, Germany 1998.

[Danforth 94] S.Danforth, I.R.Forman: *Reflections on Metaclass Programming in SOM.* Proceedings of the OOPSLA'94, pp. 440-452, Portland, Oregon, USA, Oct. 1994.

[Date 95] C.J.Date: *An Introduction to Database Systems, Sixth Edition*. Addison Wesley Publishing Company, 1995.

[Demers 95] F.N.Demers, J.Malenfant: *Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study*. Workshop of International Joint Conference on Artificial Intelligence, Montreal, Aug. 1995.

[Ellis 90] M. A. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass, 1990.

[Ferber 89] J. Ferber: *Computational Reflection in Class Based Object Oriented Languages*. OOPSLA'89 Conference Proceedings, ACM, SIGPLAN Notices, 24(10) pp. 317-326, Oct. 1989.

[Fernández 00] P.Fernández, R.Béjar, M.A.Latre, **J.Valiño**, J.A.Bañares, P.R.Muro-Medrano: *Web Mapping Interoperability in Practice, a Java Approach Guided by the OpenGis Web Map Server Interface Specification.* EC-GIS 2000, 6th European Commision GI and GIS Workshop, Lyon, Francia, Jun. 2000.

[Fickes 85] R.Fickes, T.Kehler*: The Role of Frame-Based Representation in Reasoning*. Communications of the ACM. vol. 28, num. 9, pp. 904-920, Sep. 1985.

[Foote 89] B.Foote, R.E.Johnson: *Reflective Facilities in Smalltalk-80*. OOPSLA'89, New Orleans, Oct. 1989.

[Foote 90] B.Foote: *Object-oriented Reflective Metalevel Architectures: Pyrite or Panacea?*. ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures, 1990.

[Foote 92] B.Foote: *Objects, Reflection and Open Languages*. ECOOP 92 Workshop on Object Oriented Reflection and Metalevel Architectures, Utrech, Netherlands, 1992.

[Fox 86] M.S.Fox, J.M.Wright, D.Adam: *Experiences with SRL*. Proceedings of 1st International Workshop on Expert Database Systems, Charleston, South Carolina, pp. 161-172. Benjamin/Cummings Publishing, 1986.

[Gamma 96] E.Gamma, R.Helm, R.Johnson, J.Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley 1996.

[Goldberg 83] A.Goldberg, D.Robson: *Smalltalk-80 – The language and Its Implementation*. Addison-Wesley. Reading, Mass. 1983.

[Golm 98] M. Golm, J. Kleinoder. *metaXa and the Future of Reflection*. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.

[Gowing 96] B.Gowing, V.Cahill: *Meta-Object Protocols for C++: The Iguana Approach*. Proceedings of Reflection'96, San Francisco 1996.

[Grossman 93] M.Grossman: *Object I/O and runtime type information via automatic code generation in C++*. Journal of Object Oriented Programming, Jul.-Aug. 1993.

[Hastie 95] C. Hastie: *A property template for C++*. C++ Report, Nov-Dec 1995.

[Hoare 62] C.A.R.Hoare: *Quicksort*. Computer Journal, n. 5, pp. 10-15, 1962.

[Horstmann 99] C.Horstmann: *Core Java 2*. Prentice Hall, 1999.

[Ingalls 78] D.H.H. Ingalls: *The Smalltalk-76 Programming System: Design and Implementation*. Proceedings of the ACM Principles of Programming Languages Symposium, Jan. 1978.

[Inprise 00] Inprise: *Página principal de C++ Builder*. http://www.inprise.com/c++builder

[Ishikawa 94] Y.Ishikawa: *Metalevel Architecture for Extended C++*. Technical Report TR-94024, Tsukuba Research Center, 1994.

[Jaakko 99] J.Jaakko: *C++ Function Object Binders Made Easy*. Proceedings of The First International Symposium on Generative and Component-Based Software Engineering (GCSE'99), Erfurt, Germany, Sep. 1999.

[Jones 96] N.D.Jones: *An Introduction to Partial Evaluation.* ACM Computing Surveys 28, 3, pp. 480-503, Sep. 1996.

[Kakkad 98] S.Kakkad, M.S.Johnstone, P.R.Wilson: *Portable Run-Time Type Description for Conventional Compilers*. Proceedings of International Symposium on Memory Management, Vancouver, pp. 146-153, Oct. 1998.

[Kasbekar 98] M.Kasbekar, C.Narayanan, C.R.Das: *Using Reflection for Checkpointing Concurrent Object Oriented Programs.* OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.

[Kempf 87] R.Kempf, M.Stelzner: *Teaching Object Oriented Programming with the KEE System*. Proceedings of the OOPSLA'87, Orlando, USA, pp. 11-25, 1987.

[Kernighan 98] B.Kernighan, D.Ritchie: *The C Programming Language.* 2nd ed., Prentice-Hall, 1998.

[Kizcales 91] G.Kiczales, J. des Rivieres, D.G.Bobrow: *The Art of the Metaobject Protocol*. MIT Press, Cambridge 1991.

[Kizcales 96] G.Kizcales: *Beyond the Black Box: Open Implementation*. IEEE Software, Jan. 1996.

[Kizcales 97] G.Kizcales, J.Lamping, C.V. Lopes, C.Maeda, A.Mendhekar, G.Murphy: *Open Implementation Design Guidelines*. Proceedings of the 19th International Conference on Software Engineering, pp. 481-490, Boston, USA, IEEE Press, May. 1997.

[Koenig 96] A.Koenig: *Arithmetic sequence iterators*. Journal of Object-Oriented Programming, vol. 9, num. 6, pp. 38-39, 92, Oct. 1996.

[Langer 00] A.Langer, K.Kreft: *Standard C++ IOStreams and Locales*. Addison Wesley Longman Inc., 2000.

[Lassila 90] O.Lassila: *Frames or Objects or both?*. Workshop notes from the 8th National Conference on Artificial Intelligence, Boston, USA, 1990.

[Lea 92] D.Lea: *Run-time Type Information and Class Design.* Proceedings of the 1994 Usenix C++ Conference, 1992.

[Ledoux 97] T.Ledoux: *Implementing Proxy Objects in a Reflective ORB*. ECCOP'97 Workshop CORBA: Implementation, Use and Evaluation, 1997.

[Lee 97] R.C.Lee, W.M.Tepfenhart: *UML and C++. A practical guide to object-oriented development*, pp. 250-263, Prentice-Hall, 1997.

[Lee 98] A.H.Lee, H.Shin: *Building a Persistent Object Store Using the Java Reflection API.* OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998

[Linenbach 96] T.Linenbach: *Implementing reusable binary associations in C++*. C++ Report, May. 1996.

[Maes 87] P.Maes: *Concepts and Experiments in Computational Reflection*. Proceedings of Object Oriented Programming Systems Languages and Applications Conference, pp. 147-155, Dec. 1987.

[Martin 98] R.Martin, D.Riehle, F.Buschmann: *Pattern Languages of Program Design 3*. Addison Wesley, 1998.

[Masini 91] G.Masini, A.Napoli, D.Colnet, D.Leonard, K.Tombe: *Object Oriented Languages*. The APIC Series, vol. 34, Academic Press, 1991.

[McCluskey 97] G.McCluskey: *Remote Method Invocation: Creating Distributed Java-to-Java Applications*. Oct. 1997. http://developer.java.sun.com/developer/technicalArticles/RMI/CreatingApps/index.html

[McCluskey 98] G.McCluskey: *Using Java Reflection*. Jan. 1998. http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/index.html

[McCluskey 98b] G.McCluskey: *Using Collections with JDKTM 1.2*. Dec. 1998. http://developer.java.sun.com/developer/technicalArticles/Collections/Using/index.html

[Meyer 92] B.Meyer: *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Meyer 94] B.Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall 1994.

[Meyer 99] B.Meyer: *Construcción de Software Orientado a Objetos, segunda edición*. Prentice Hall, 1999.

[Minsky 81] M.Minsky: *Framework for Representing Knowledge*. Mind Design, pp. 95-128, Ed. J. Haugeland, MA: The MIT Press, 1981.

[Moon 86] D.Moon: *Object-Oriented Programming with Flavors*. Proceedings of the ACM OOPSLA Conference, 1986.

[Musser 87] D.R.Musser, A.A. Stepanov: *A Library of Generic Algorithms in Ada*. Proceedings of the 1987 ACM SIGAda International Conference on Using Ada, Boston, pp. 216-225, Dec. 1987.

[Musser 89] D.R.Musser, A.A.Stepanov: *Generic Programming.* Lecture Notes in Computer Science 358, pp 13-25, Springer-Verlag, 1989.

[Musser 98] D.R.Musser: *Notes on Expression Templates.* Generic Programming Seminar, 1998.

[Myers 95] N.C.Myers: *Traits: a new and useful template technique.* C++Report, Jun. 1995.

[Myers 97] A.L.Myers, J.A.Bank, B.Liskov: *Parametrized Types for Java.* ACM Symposium on Principles of Programming Languages, pp. 132-145, Jan. 1997.

[ObjectSpace 00] Object Space: *JGL*. http://www.objectspace.com/products/prodJGL.asp.

[OCS 99] Object Consultance Services: *C++ Beyond the ARM*. http://www.ocsltd.com/c++

[OMG 95] Object Management Group and X/Open : *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* Jul. 1995.

[Paepcke 93] A.Paepcke: *User-Level Language Crafting: Introducing the CLOS Metaobject Protocol*. The CLOS Perspective, pp. 65-99, MIT Press, 1993.

[Papurt 95] D.M.Papurt: *Automatic Association Implementation in C++*. Dr. Dobb's Journal, pp. 18-25, Oct. 1995.

[Papurt 95b] D.M.Papurt: *Inside the Object Model: The Sensible Use of C++*. SIGS books, 1995.

[Patapis 95] G.Patapis: *The OSE C++ Libraries, the Unix/C++ community's best kept secret revealed*. C/C++ Journal, Nov. 1995. Dirección de la librería: http://www.dscpl.com.au/

[Peralta 98] A.J.Peralta, P.Botella, J.Serras: *It's time for full life-cycle languages*. Journal of Object Oriented Programming, pp. 19-28, vol. 10, num. 9, Feb. 1998.

[Powell 00] G.Powell, M.Weiser: *Container Adaptors*. C/C++ Users Journal, 18 2000, No. 4, pp. 40-51, Abr. 2000.

[Powell 00b] G. Powell, P. Higley: *Expression Templates as a Replacement for simple Functors*. C++ Report, May. 2000.

[Rasala 97] R.Rasala: *Function objects, Function Templates and Passage by Behavior in C++.* SIGCSE'97, pp. 35-38, 1997.

[Reeves 97] J.W.Reeves: *Faking (and exploring) runtime type information.* C++Report, pp. 55-63, Oct. 1997.

[Riehle 95] D.Riehle, W.Siberski, D.Bäumer, D.Megert, H.Züllighove: *Serial-izer.* Pattern Languages of Program Design 3, pp. 293-312, Addison Wesley, 1998.

[Rivard 96] borrar? F.Rivard: *A new Smalltalk Kernel allowing both Explicit and Implicit Metaclass Programming*. OOPSLA'96, Oct. 1996.

[Rivard 96] F.Rivard: *Smalltalk: a Reflective Language*. Reflection'96, pp. 21-38, San Francisco, USA, Apr. 1996.

[Roberts 77] R.B.Roberts, I.P.Goldstein: *The FRL Manual*. AI Memo, AI Lab, MIT, Cambridge, Massachusetts, 1977.

[Rogue 00] Rogue Wave Software: *Tools.h++*.
http://www.roguewave.com/products/xplatform/tools/

[Roth 98] B.Roth: *An Introduction to Enterprise JavaBeansTM Technology.* Oct. 1998.
http://developer.java.sun.com/developer/technicalArticles/Beans/IntroEJB/index.html

[Rumbaugh 87] J.E.Rumbaugh: *Relations as Semantic Constructors in an Object-Oriented Language*. Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA'87, 1987.

[Rumbaugh 96] J.E.Rumbaugh: *Models for design: Generating code for associations.* Journal of Object Oriented Programming, Feb. 1996.

[Rumbaugh 99] J.E.Rumbaugh, I.Jacobson, G.Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Saks 92] D.Saks: *The return type of virtual functions*. C++Report, vol. 4, num 7, pp. 61-62, Sep. 1992.

[Scaife 96] N.R. Scaife: *A Survey of Concurrent Object-Oriented Programming Languages.* Technical Report TM/96/4, Edinburg, 1996.

[Shah 89] A.V.Shah, J.E.Rumbaugh, J.H. Hamel, R.A.Borsari: *DSM: An Object-Relationship Modeling Language.* Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA'89, New Orleans, Oct. 1989.

[Simonis 00] V.Simonis: *Adapters and Binders - Overcoming Problems in the Design and Implementation of the C++-STL.* ACM SIGPLAN Notices, vol. 35, num. 2, pp. 46-53, Feb. 2000.

[Singhal 92] V.Singhal, S.V.Kakkad, P.R.Wilson: *Texas: An Efficient, Portable Persistent Store*. Fifth International Workshop on Persistent Object Systems, pp 11-33, San Miniato, Italy, Sep. 1992.

[Smith 90] B.Smith: *Informal Proceedings of the First Workshop on Reflection and Metalevel Architectures in Object Oriented Programming*. OOPSLA-ECOOP'90, Ottawa, 1990.

[Soukup 95] J.Soukup: *Implementing Patterns.* Pattern Languages of Program Design. Addison-Wesley, 1995.

[Stallman 00] R.M.Stallman: *Debugging with GDB The GNU Source-Level Debugger*. Free Software Fundation, 2000.

[Stefik 85] M. Stefik, D. G. Bobrow*: Object Oriented Programming: Themes and Variations.* The AI Magazine, pp. 40-62. 1985.

[Stepanov 94] A.Stepanov, L.Meng: *The Standard Template Library. ANSI X3J16-94-0095/ISO WG21-NO482*, 1994.

[Stepanov 95] A.Stepanov: *The Standard Template Library*. Byte Magazine, Oct. 1995.

[Strickland 93] H.Strickland: *Taking the lid off C++ effectively using runtime type information*. C++Report, pp. 19-22, Feb. 1993.

[Stroustrup 92] B. Stroustrup, D. Lenkov: *Runtime type identification for C++*. C++ Report, Mar.-Apr. 1992.

[Stroustrup 94] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass, 1991.

[Stroustrup 97] B.Stroustrup: *The C++ Language, Third Edition*. Addison-Wesley, 1997.

[Sun 97] Sun Microsystems: *Object Serialization Specification*. 1997. http://java.sun.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html

[Sun 97b] Sun Microsystems: *Java Core Reflection. API and Specification.* Feb. 1997. http://java.sun.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html

[Sun 98] Sun Microsystems: *RMI - Remote Method Invocation.* 1998. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html

[Sun 99] Sun Microsystems: *Write to the JavaBeans Component Architecture*. 1999. http://java.sun.com/beans/

[Taft 97] S.T.Taft, R.A.Duff, T.Taft (Eds.): *Ada 95 Reference Manual: Language and Standard Libraries*. International Standard ISO/IEC 8652:1995(E). Lecture Notes in Computer Science, vol. 1246. Springer-Verlag, 1997.

[Tatsubori 98] M.Tatsubori, S.Chiba: *Programming Support of Design Patterns with Compile-Time Reflection*. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.

[Tatsubori 99] M.Tatsubori: *An Extension Mechanism for the Java Language.* Master of Engineering of the University of Tsukuba, 1999.

[Valiño 97] **J.Valiño**, F.J.Zarazaga, S.Comella, J.Nogueras, P.R.Muro: *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.

[Veldhuizen 95] T.Veldhuizen: *Using C++ template metaprograms.* C++Report, vol. 7, num. 4, pp. 36-43, May. 1995.

[Veldhuizen 95b] T.Veldhuizen: *Expression Templates.* C++Report, vol. 7, num. 5, pp. 26-31, Jun. 1995.

[Veldhuizen 98] T.Veldhuizen: *Arrays in Blitz++*. ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science. 1998.

[Willink 99] E.D.Willink, V.B.Muchnick: *Prepocessing C++: Meta-class Aspects.* Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems, TOOLS EE 99, Blagoevgrad, Bulgaria, Jun. 1999.

[Wilson 93] P.Wilson, M.S.Johnstone: *Truly Real-Time Non-Copying Garbage Collection*. OOPSLA 93 Workshop on Memory Management and Garbage Collection, Dic. 1993.

[Woods 75] W.A.Woods: *What´s in a Link: Foundations for Semantic Networks*. *Representation and Understanding.* Studies in Cognitive Science, pp. 35-82. Ed. D.G. Bobrow & A.M. Collins, New York, Academic Press, 1975.

[Wright 00] S.Wright: *An Overview of the Booch Components for Ada95*. http://www.pogner.demon.co.uk/components/bc/documentation.html

[Zarazaga 97] F.J.Zarazaga, **J.Valiño**, J.A.Bañares, C.Felipe, F.J.Salas, Pedro R.Muro-Medrano: *Aplicación de técnicas de representación basadas en frames para el desarrollo del modelo de objetos de un sistema de gestión de redes de radiotelefonía trunking*. Segundas Jornadas sobre Transferencia Tecnológica de la Inteligencia Artificial, TTIA 97. pp 59-66, Málaga, España. Nov, 1997

[Zarazaga 98a] F.J.Zarazaga, **J.Valiño**, S.Comella, J.Nogueras, A.Romay, C.Felipe, F.J.Salas, P.Muro-Medrano: *TRUNIS: sistema de información orientado a objeto para la gestión de redes de radiotelefonía trunking*. Actas de las IV Jornadas de Informática*.* Las Palmas de Gran Canaria, España, pp. 283-292, Jul. 1998.

[Zarazaga 98b] F.J.Zarazaga, S.Comella, J.Nogueras, **J.Valiño**, P.R.Muro-Medrano: *Una aproximación para la utilización de metainformación en aplicaciones de C++*. IV Jornadas sobre tecnologías de objetos, Bilbao, España, Oct. 1998.

[Zarazaga 98c] F.J.Zarazaga, S.Comella, J.Nogueras, **J.Valiño**: *Automatizando el paso de Diseño Orientado a Objeto a Codificación mediante el uso de Metainformación basada en facets*. III Jornadas de Ingeniería del Software. pp 411-422, Murcia, España. Nov. 1998.

[Zarazaga 99] F.J.Zarazaga, **J.Valiño**, S.Comella, J.Nogueras, P.R.Muro-Medrano: *TRUNIS: an Object Oriented Trunking Radio Telephone Network Information System. An experience report*. Proceedings of the 29 Conference on Technology of Object-Oriented Languages and Systems. IEEE Computer Society Press pp. 251 - 260. Nancy, Francia, Jun. 1999.

[Zarazaga 00] F.J.Zarazaga: *Una aproximación a la mejora de la reusabilidad de código C++ basada en metainformación del modelo de objetos*. Tesis Doctoral, Zaragoza, Ene. 2000.

[Zarazaga 00b] F.J.Zarazaga, P.Álvarez, J.Guillo, R.López, **J.Valiño**, P.R.Muro-Medrano: *Use cases of vehicle location systems based on distributed real-time GPS data*. Second International Symposiun on Telegeoprocessing Telegeo'2000, Niza, Francia, May. 2000. Aceptada publicación en Actas LNCS, Springer Verlag.