Universidad Carlos III de Madrid
Ingeniería Superior en Informática
Proyecto Fin de Carrera
(junio de 2002)

IMPLEMENTACIÓN EN *JAVA* DE ASOCIACIONES BINARIAS *UML*

Carlos Ruiz del Castillo y Ubach
Tutor: **Gonzalo Génova Fuster**

A Teresa con todo mi cariño.

Agradecimientos a Gonzalo por su rigurosidad, interés
y paciencia, a mis padres por su ayuda y apoyo, y
a Carlos por el icono de JUMLA.

ÍNDICE

1 INTRODUCCIÓN 8

2 OBJETIVOS 10

PARTE I: ESTUDIO TEÓRICO 12

3 TIPOS DE ASOCIACIONES 13

4 EL PROBLEMA DEL CONTROL DE LAS ASOCIACIONES Y SU SOLUCIÓN 16

4.1 IMPLEMENTACIÓN DE LAS ASOCIACIONES 16
4.2 LOS MÉTODOS NECESARIOS 17
4.3 EL PROBLEMA DEL CONTROL DE LA MULTIPLICIDAD 17
4.3.1 MULTIPLICIDAD MÁXIMA 17
4.3.2 MULTIPLICIDAD MÍNIMA 18
4.3.3 CONCLUSIÓN AL PROBLEMA DE LA MULTIPLICIDAD 20
4.4 LA SOLUCIÓN EMPLEADA 20

5 UN INTERFAZ UNIFORME 22

5.1 RESUMEN DEL CÓDIGO PARA LA GESTIÓN DE UNA ASOCIACIÓN 22
5.2 CONDICIONES DE USO 23
5.2.1 CLASE '*SOURCE*' Y ATRIBUTO '*TARGETROLE*' 23
5.2.2 CONSTANTES DE LA CLASE '*ASSOCIATION*' 23
5.2.3 MÉTODOS DE DEFINICIÓN DE LA ASOCIACIÓN 25
5.2.4 MÉTODOS PARA EL CONOCIMIENTO DEL ESTADO DE LOS ENLACES 25
5.2.5 MÉTODOS PARA LA COMPROBACIÓN, ACCESO, ESTABLECIMIENTO Y ELIMINACIÓN DE LOS ENLACES 26

6 IDEAS ALTERNATIVAS PARA IMPLEMENTAR ASOCIACIONES 30

6.1 ATRIBUTOS Y MÉTODOS DE CONTROL DE LA ASOCIACIÓN EN LA PROPIA CLASE 30
6.2 CLASES ASOCIATIVAS 30

6.3 CLASES ASOCIATIVAS ESTÁTICAS 31
6.4 CLASES VALIDADAS 31

PARTE II: IMPLEMENTACIÓN DE ASOCIACIONES BINARIAS 34

7 GENERACIÓN AUTOMÁTICA DEL CÓDIGO DE CONTROL DE LAS ASOCIACIONES 35

7.1 FUNCIONAMIENTO BÁSICO DE LA HERRAMIENTA 35
7.1.1 NOMBRES CLAVES EN LAS PLANTILLAS 35
7.1.2 TIPOS DE PLANTILLAS 37
7.2 CONVENCIÓN DE NOMBRES 37
7.2.1 CLASE INTERNA DE GESTIÓN DE LA ASOCIACIÓN 37
7.2.2 ATRIBUTO DE ALMACÉN DE ENLACES 39
7.2.3 NOMBRADO DE LOS MÉTODOS 40
7.2.4 EJEMPLO ACLARATIVO 40

8 EXCEPCIÓN DE ASOCIACIÓN NO ESTABLECIDA 42

9 IMPLEMENTACIÓN DE ASOCIACIONES UNIDIRECCIONALES 43

9.1 ASOCIACIONES UNIDIRECCIONALES SENCILLAS 43
9.1.1 LA CLASE '*SOURCE*' 47
9.1.2 LA CLASE '*ASSOCIATION*' 47
9.1.3 VISIBILIDAD DE LA ASOCIACIÓN 49
9.2 ASOCIACIONES UNIDIRECCIONALES MÚLTIPLES 50
9.2.1 CLASES IMPORTADAS 55
9.2.2 LA CLASE '*ASSOCIATION*' 56
9.2.3 VISIBILIDAD DE LA ASOCIACIÓN 58

10 IMPLEMENTACIÓN DE ASOCIACIONES BIDIRECCIONALES 59

10.1 ASOCIACIONES SENCILLAS – SENCILLAS 60
10.1.1 LA CLASE '*SOURCE*' 64
10.1.2 LA CLASE '*ASSOCIATION*' 64
10.1.3 VISIBILIDAD 68
10.2 ASOCIACIONES MÚLTIPLES – MÚLTIPLES 68
10.2.1 LA CLASE '*ASSOCIATION*' 75
10.3 ASOCIACIONES SENCILLAS – MÚLTIPLES Y MÚLTIPLES – SENCILLAS 77

11 ASOCIACIONES REFLEXIVAS 80

11.1 ASOCIACIONES UNIDIRECCIONALES REFLEXIVAS 80

11.2 ASOCIACIONES BIDIRECCIONALES REFLEXIVAS 80

12 ASOCIACIONES *TODO / PARTE* 82

12.1 AGREGACIÓN 82
12.2 COMPOSICIÓN 82

13 VISIBILIDAD EN LAS ASOCIACIONES 84

13.1 VISIBILIDAD EN LAS ASOCIACIONES UNIDIRECCIONALES 84
13.2 EL PROBLEMA DE LA VISIBILIDAD EN LAS ASOCIACIONES BIDIRECCIONALES 85
13.2.1 UNA SOLUCIÓN FUNCIONAL 87
13.3 RESUMEN DE LA IMPLEMENTACIÓN DE LA VISIBILIDAD EN LAS ASOCIACIONES 87

PARTE III: APLICACIÓN DE DEMOSTRACIÓN - *UML_ASSOCIATIONS* 89

14 INTRODUCCIÓN A *UML_ASSOCIATIONS* 90

14.1 FUNCIONAMIENTO BÁSICO DE LA APLICACIÓN 90
14.2 INSTRUCCIONES DE USO 92
14.2.1 INSTALACIÓN Y EJECUCIÓN 92
14.2.2 USO 92

15 DISEÑO DE LA APLICACIÓN 95

15.1.1 '*MAIN*' 96
15.1.2 '*UNSETASSOCIATIONEXCEPTION*' 96
15.1.3 RESTO DE CLASES 96

PARTE IV: HERRAMIENTA PARA LA GENERACIÓN DE CÓDIGO - *JUMLA* 98

16 INTRODUCCIÓN A *JUMLA* 99

16.1 CONDICIONES DE VALIDEZ DEL MODELO 99
16.2 CONDICIONES DE VALIDEZ DEL CÓDIGO GENERADO 100
16.3 INGENIERÍA DIRECTA E INVERSA 101

17 MANUAL DE USUARIO 102

17.1 REQUERIMIENTOS DEL SISTEMA 102
17.2 INSTALACIÓN Y CONFIGURACIÓN 102

17.2.1 DISTRIBUCIÓN DE LOS FICHEROS DE LA HERRAMIENTA 102
17.2.2 EJECUCIÓN DE LA APLICACIÓN 104
17.2.3 CONFIGURACIÓN DE *JUMLA* 105
17.3 USO DE LA HERRAMIENTA 108
17.3.1 CARGA DE UN FICHERO *XMI* 109
17.3.2 INTERPRETACIÓN DE LAS ASOCIACIONES, SELECCIÓN Y GENERACIÓN DE CÓDIGO 110

18 DISEÑO DE LA APLICACIÓN 113

18.1 'APPLICATION' 113
18.2 'MAINFRAME' 113
18.3 'UMLASSOCIATION' 113
18.4 'XMIFILEPARSER' 115
18.5 'JUMLACLASSGENERATOR' 115
18.6 'JUMLACONFIG' 116
18.7 'JUMLALOG' 116
18.8 'CONFIGURATIONFRAME' 116
18.9 'HELPFRAME' 116
18.10 'ABOUTFRAME' 116

CONCLUSIONES 117

19 CONCLUSIONES 118

19.1 ACERCA DE UML Y JAVA 118
19.2 ACERCA DEL MÉTODO DE IMPLEMENTACIÓN DE ASOCIACIONES 119
19.3 ASPECTOS QUE QUEDAN POR VER 120

ANEXOS 122

ANEXO A: CONVENCIÓN DE NOMBRES EMPLEADA EN EL CÓDIGO GENERADO 123

NOMBRES CLAVES EN LAS PLANTILLAS 123
NOMBRES CLAVES EN EL CÓDIGO GENERADO 124
NOMBRE DE LA CLASE INTERNA GESTORA DE UNA ASOCIACIÓN 124
ATRIBUTO DE GESTIÓN DE LA ASOCIACIÓN 124
NOMBRE DE LA CLASE INTERNA EN ASOCIACIONES REFLEXIVAS 124
CLASE INTERNA DE ASOCIACIONES *TODO / PARTE* 125
CLASE INTERNA DE ASOCIACIONES REFLEXIVAS *TODO / PARTE* 125

ANEXO B: CONDICIONES DE VALIDEZ DE *JUMLA* 126

CONDICIONES DE VALIDEZ DEL MODELO DE ENTRADA PARA SER ACEPTADO 126
CONDICIONES DE VALIDEZ DEL CÓDIGO GENERADO..... 127

ANEXO C: CONDICIONES DE USO DEL INTERFAZ..... 128

INTERFAZ DEL CÓDIGO PARA LA GESTIÓN DE UNA ASOCIACIÓN..... 128
CONDICIONES DE USO 128
CLASE '*SOURCE*' Y ATRIBUTO '*TARGETROLE*'..... 128
CONSTANTES DE LA CLASE '*ASSOCIATION*' 129
MÉTODOS DE DEFINICIÓN DE LA ASOCIACIÓN 130
MÉTODOS PARA EL CONOCIMIENTO DEL ESTADO DE LOS ENLACES..... 130
MÉTODOS PARA LA COMPROBACIÓN, ACCESO, ESTABLECIMIENTO Y ELIMINACIÓN DE LOS ENLACES 131

ANEXO D: PLANTILLAS EMPLEADAS EN LA GENERACIÓN DE CÓDIGO 135

'SINGLEUNIDIRECTIONAL.JAVA~'..... 135
'MULTIPLEUNIDIRECTIONAL.JAVA~'..... 138
'SINGLEBIDIRECTIONAL.JAVA~' 142
'MULTIPLEBIDIRECTIONAL.JAVA~' 145
'EMPTYCLASS.JAVA~' 150

ANEXO E: PLANTILLAS DE CÓDIGO ALTERNATIVAS..... 151

'SINGLEBIDIRECTIONALALTERNATIVE.JAVA~'..... 151
'MULTIPLEBIDIRECTIONALALTERNATIVE.JAVA~'..... 155

ANEXO F: CONTENIDO DEL CD 163

BIBLIOGRAFÍA..... 164

1 INTRODUCCIÓN

El lenguaje unificado de modelado *UML* (*Unified Modeling Language*) es un lenguaje gráfico que se inventó para diseñar sistemas informáticos. Con él se pretende proporcionar en la fase de diseño las especificaciones que va a tener un sistema así como sus restricciones, y todo, de una forma estándar, para que cualquier persona que tenga unos conocimientos esenciales de *UML* pueda entender rápida y fácilmente lo que se está construyendo.

Ahora bien, por un lado está *UML* y, por otro, los lenguajes de programación (concretamente los orientados a objetos) con los que realmente se construyen los sistemas software. No cabe duda de que *UML* ha aportado mucho a la comunidad informática, pero desde su aparición surgió la idea de intentar automatizar el proceso de paso de especificación de diagrama de clases *UML* a codificación en un lenguaje de programación concreto. Esto se capta rápidamente si se observa la evolución que han sufrido las herramientas *UML*. Las primeras herramientas apenas servían más que para realizar los diagramas *UML*, pero las más actuales son capaces de generar cierto esqueleto de código en más de un lenguaje de programación.

El nivel alcanzado de paso automático de *UML* a código no es suficiente y son varios los intentos y propuestas realizados. Los principales escollos que encontramos muchas veces se deben a una falta de integración y acoplamiento entre *UML* y los lenguajes de programación OO. No obstante, el problema fundamental es que *UML* sirve para hacer un diseño en abstracto, puesto que se trata de modelar y adelantar detalles de implementación, pero no de implementar en sí. El paso de un diseño abstracto, en el que se ha suprimido información para simplificar y facilitar su entendimiento, a un código con todos los detalles, parece una tarea difícilmente automatizable para que la pueda realizar enteramente una máquina sin intervención humana.

En este trabajo nos vamos a centrar en la implementación de las asociaciones binarias de *UML*. Pero antes de seguir, veamos que se entiende por una asociación: *"Una asociación no es más que una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro. Dada una asociación entre dos clases, se puede navegar desde un objeto de una clase hasta un objeto de otra clase, y viceversa"* [2]. Entenderemos por "asociación binaria" la relación que se acaba de definir entre dos clases, y por "enlace", la asociación o conexión específica que hay entre dos instancias de las clases participantes en una asociación.

Comprendido lo que es una asociación, ya podemos señalar que en este trabajo se pretende aportar un método que permita, de una forma genérica y automática, escribir el código necesario para la gestión de una asociación entre dos clases. Veremos que no hay una relación directa entre la asociación existente entre dos clases *UML* y cualquiera de las utilidades que aportan los lenguajes OO debido a que éstos no soportan directamente el concepto de asociación. No obstante, con el método que se va a presentar, se quiere conseguir que un programador tenga una forma fácil de gestionar las asociaciones en su código.

El lenguaje OO sobre el cual se va a adoptar este método va a ser *Java*. El motivo por el que se ha seleccionado *Java* es porque este lenguaje goza de mucha popularidad, aparte de ser OO en un alto grado de pureza y estar diseñado para poder ejecutarse en cualquier plataforma. Si realmente se consigue aportar un buen método que haga fácil la gestión de asociaciones entre dos clases *Java*, creemos que podremos ayudar a los programadores de dicho lenguaje que previamente, utilizan *UML* en sus diseños. No obstante, el método que vamos a presentar es aplicable a cualquier

lenguaje OO, por lo que basta con coger las ideas que aquí se presentan para adaptarlo a otros lenguajes como *C++*, *SmallTalk*, etc.

El proyecto está dividido en cuatro partes. Tras esta breve introducción y marcar los objetivos que se pretenden lograr, en la primera parte se hace un estudio teórico del problema. En dicho estudio se profundiza en la semántica de las asociaciones y en los tipos de asociaciones que nos podemos encontrar para, así, agruparlas de alguna manera. Después, se estudia el problema del control de las asociaciones, para ver qué características de las asociaciones son importantes y cuáles son las que pueden presentar dificultades a la hora de implementarlas en un lenguaje OO. Tras evaluar más a fondo el problema, se comenta la solución que proponemos y que intentará conseguir en la medida de lo posible los objetivos marcados. Una vez presentada la solución, se muestran de una forma más específica las condiciones de uso y el interfaz que ofrecerá el código generado al programador. Para finalizar la primera parte, se muestran otras posibles soluciones que, si bien han sido descartadas en este proyecto, podrían valer como alternativas.

En la parte II, se pasa a describir la implementación práctica y concreta de las asociaciones *UML* en *Java*. Se establecen los criterios de nombres a utilizar en el código generado, las excepciones y el código necesarios para la gestión de todos los tipos de asociaciones, así como de algunos casos especiales: asociaciones reflexivas, agregaciones y composiciones.

En la tercera parte, se presenta una aplicación (*UML_Associations*) de demostración para comprobar que la implementación de las asociaciones, tal y como se ha presentado en la segunda parte, funciona y es válida. Para ello, la aplicación es capaz de trabajar con cualquier asociación que solicite el usuario, crear instancias y enlazarlas, borrarlas y eliminar los enlaces.

Por último, en la parte IV, se muestra la herramienta final (*JUMLA*[1]) desarrollada en este proyecto (fruto de las otras partes del documento) capaz de generar el código necesario para la gestión de las asociaciones que se encuentren especificadas en un fichero *XMI*[2] generado por una herramienta *CASE* comercial. Primero se dan las condiciones de validez del modelo y, después, se muestra el manual de usuario, que contiene las instrucciones de instalación, configuración y de uso. También se explica brevemente el diseño de la aplicación.

Al final de este documento, se han puesto varios anexos. Con ellos, se intenta recoger un resumen (a modo de referencia) de los asuntos más importantes desde el punto de vista práctico: convención de nombres, plantillas de código, condiciones de uso, etc. También, en el último de ellos, se muestra el contenido del CD que se presenta conjuntamente con este proyecto.

---

[1] *"Java code generator for UML Associations"*

[2] *"XML Metadata Interchange"*. Posteriormente veremos que se trata de un modelo de intercambio de información abierto idóneo para almacenar modelos *UML*.

2 OBJETIVOS

Vamos a desarrollar un método, como se ha dicho, para la gestión de asociaciones binarias *UML* en *Java*. Dicho método se va a implementar en una herramienta software capaz de generar el código necesario para la gestión de las asociaciones. De esta forma, el usuario de la herramienta deberá ser capaz de introducir asociaciones y obtener la estructura en *Java* de las clases participantes en ellas, así como el código necesario para el control de los enlaces.

Nos ceñiremos a las asociaciones binarias y, dentro de este grupo, abordaremos todos los tipos de asociaciones excepto las clases-asociación y las asociaciones cualificadas, es decir, abordaremos las asociaciones unidireccionales y bidireccionales, las composiciones y agregaciones, así como las asociaciones reflexivas.

En cuanto a las características de las asociaciones que vamos a tener en cuenta, nos centraremos principalmente en la navegabilidad y multiplicidad y, aunque de un modo menos exhaustivo, también en la visibilidad. En cuanto a la multiplicidad sólo se contemplan las multiplicidades que se especifican dentro de un único rango y no en varios. Así, se admiten como válidos, por ejemplo, (0..1) y (m..n), donde *m* y *n* son valores concretos, pero no se admiten rangos compuestos como (1..2, 4..5) o multiplicidades que no son rangos como (números pares). Otras restricciones como *{ordered}*, *{subset}* o *{xor}* también quedan fuera del ámbito de este proyecto. Señalaremos también los casos en los que se haga obligatoria la especificación de nombres en las asociaciones y en los roles.

Del código *Java* generado automáticamente por la herramienta, se intentará que tenga las siguientes características:

- **Simplicidad**: El código ha de ser lo más simple posible.
- Facilidad de **localización** e **identificación**: Tiene que ser fácil identificar qué atributos y métodos se encargan del control de cada asociación y deben encontrarse agrupados de alguna manera.
- **Atomicidad**: La creación o cambio dinámico de los enlaces debe realizarse en un proceso indivisible. Por tanto, no se debe dejar, a no ser que sea inevitable, que la asociación se quede en un estado no permitido por las restricciones impuestas.
- **Bajo acoplamiento**: El control de la validez de la asociación debe restringirse en la medida de lo posible a las clases que poseen la asociación y éstas deben interferir entre sí lo mínimo posible.
- **Generalidad**: Se buscará que el código sea válido para todo tipo de asociaciones y por tanto, si varía la definición de una asociación, los cambios a realizar, dentro del código que de alguna manera hace uso de la asociación, han de ser mínimos. Además, se intentará que sólo afecten, en la medida de lo posible, a las clases implicadas, y no a cualquier pieza de código que haga uso de dichas clases.
- **Interfaz uniforme**: Asimismo, se intenta que absolutamente todas las asociaciones se utilicen de la misma forma y por tanto, haya un único interfaz que utilizar y que aprender en la gestión de las asociaciones.

- **Completitud**: Se deben soportar todos los métodos necesarios para el control y variación dinámica de los enlaces. El usuario final deberá decidir qué métodos va a utilizar o no en función de sus necesidades.
- **Elección de visibilidad**: El usuario podrá elegir la visibilidad de las asociaciones pudiendo hacer que sólo una clase participante de la asociación sea capaz de controlarla, sólo esa clase y sus herederas, las clases que se encuentran en el mismo paquete que la asociación o que no haya ninguna restricción específica de acceso.
- **Robustez**: El código deberá poder hacer frente a todo tipo de anomalías: parámetros de entrada nulos cuando se espere algún valor de referencia válido, especificación incorrecta de enlaces, colecciones de objetos *Java* con elementos de tipo inesperado, etc.

Veremos que no siempre es posible alcanzar estos objetivos pero intentaremos aproximarnos lo máximo posible o dar una solución suficientemente satisfactoria. En cualquier caso, cuando no se puedan acometer algunos de estos objetivos, daremos una explicación de por qué esa inviabilidad. Normalmente, esa inviabilidad será por falta de integración directa entre *UML* y *Java* por lo que incluso se llegará a hacer propuestas de cambios en *Java* que faciliten la implementación de las asociaciones.

PARTE I: ESTUDIO TEÓRICO

3 TIPOS DE ASOCIACIONES

Para la implementación de las asociaciones binarias *UML*, primero tenemos que ver qué tipos tenemos para luego agruparlos y estudiar como se va a realizar la implementación en esos grupos. Según el estándar de *UML* si nos ceñimos a la navegabilidad, nos encontramos con dos grandes grupos: las **unidireccionales** y las **bidireccionales** y dentro de cada grupo, según la multiplicidad que se tenga en cada lado de la asociación, hallaremos distintas combinaciones.

Por lo pronto, dejaremos de lado las asociaciones reflexivas así como las relaciones *todo / parte* (agregación y composición) que merecen ser estudiadas aparte, por lo que en este apartado, nos ceñiremos a los dos grandes grupos de asociaciones típicas: unidireccionales y bidireccionales.

Existen cinco tipos distintos de multiplicidades que aparecen en función de que la asociación sea obligatoria (*mandatory*) u opcional y de que sea restringida en el número máximo permitido de posibles enlaces o no tenga límites. He aquí las cinco posibles combinaciones:

- (0..1) – La asociación con esta multiplicidad puede tener ninguno o un único enlace por lo que no es obligatoria (no *mandatory*).
- (1..1) – Obligatoriamente habrá siempre un enlace y sólo uno (*mandatory*).
- (0..*) – Esta es la menos restrictiva de todas y dice que se puede tener cualquier número de enlaces incluyéndose la posibilidad de no tener ninguno (no *mandatory*).
- (1..*) – Aunque parecida a la anterior, se diferencia en que se obliga a que por lo menos exista un único enlace (*mandatory*).
- (*m*..*n*) – Especifica un número mínimo y máximo de enlaces por lo que la multiplicidad en la asociación se va a mover entre *m* y *n* enlaces (obligatoria si *m* mayor que cero)[3].

Viendo estas cinco posibles multiplicidades, nos damos cuenta que hay dos grupos: las sencillas y las múltiples. Las **sencillas** son las dos primeras y son aquellas en las que a lo sumo, una instancia tiene un enlace, y las **múltiples** las otras tres, es decir, en las que una instancia puede tener más de un enlace.

En las asociaciones unidireccionales sólo se puede restringir la multiplicidad en el destino puesto que si se supone que la asociación sólo es navegable desde la clase origen a la de destino, entonces no se debe poder controlar la multiplicidad de destino a origen. Si se hiciese, nos encontraríamos ante una asociación bidireccional porque tendríamos que obligar al destino a tener conocimiento de la clase origen. La restricción de multiplicidad en la clase fuente de una asociación unidireccional, por tanto, sólo se puede garantizar si el programador se asegura de su cumplimiento y no hay forma posible de que un método genérico la garantice.

Es decir, en el método que se va a implementar, la multiplicidad en el extremo no navegable de una asociación deber ser siempre 0..* (no restringida) [4]. Es por esto que vamos a tener sólo cinco tipos de asociaciones unidireccionales. Las tenemos en Tabla 3.1.

[3] Nótese que en el fondo todas las multiplicidades son del tipo *m*..*n* pero las otras, por ser bastante típicas, se estudian individualmente.

A → (0..1) B A → (1..1) B	Asociaciones unidireccionales sencillas
A → (0..*) B A → (1..*) B A → (m..n) B	Asociaciones unidireccionales múltiples

Tabla 3.1: Asociaciones unidireccionales.

Las asociaciones bidireccionales, por contra, sí exigen que se especifique multiplicidad en ambos lados por lo que nos encontramos con 25 posibles combinaciones (Tabla 3.2).

No obstante, puede observarse que de estos 25 casos realmente sólo 15 son distintos y los otros 10 son producto de invertir las multiplicidades en '*A*' y en '*B*'. Así, por ejemplo, las asociaciones '*A (0..1)* ↔ *(1) B*' y '*A (1)* ↔ *(0..1) B*' son las mismas desde el punto de vista práctico de implementación (que no semántico). Cuando se especifica una asociación, es normal interpretarla de izquierda a derecha pues es la forma natural en la que leemos y escribimos. Ahora bien, *UML* no dice que esto sea así necesariamente y de hecho, cuando se pone el nombre de la asociación, se suele acompañar de una flecha que indica en qué sentido se debe interpretar, pero desde el punto de vista práctico de utilización, en las asociaciones bidireccionales, ambas clases participantes pueden acceder por igual a los enlaces.

Debido a que estamos buscando un bajo acoplamiento entre las clases conectadas mediante asociación, vamos a intentar separar la multiplicidad de cada lado y hacer independiente el hecho de que un lado '*A*' tenga multiplicidad sencilla o no respecto al otro extremo y lo mismo para '*B*' respecto a '*A*'. De este modo, el estudio se nos queda reducido a cuatro grupos:

- Asociaciones unidireccionales sencillas.
- Asociaciones unidireccionales múltiples.
- Extremo de asociación bidireccional sencillo.
- Extremo de asociación bidireccional múltiple.

Los dos primeros son los que teníamos en las asociaciones unidireccionales y los otros dos se refieren al estudio por separado de cada clase participante en una asociación según que su multiplicidad respecto a la otra clase sea sencilla o múltiple. Haciendo todas las combinaciones de los dos casos que tenemos de lado sencillo y los tres de lado múltiple para las dos clases que participan en una asociación bidireccional, obtendremos los 25 posibles casos de asociaciones que se ha presentado anteriormente en la Tabla 3.2.

A (0..1) ↔ (0..1) B A (0..1) ↔ (1) B A (1) ↔ (0..1) B A (1) ↔ (1) B	Asociaciones bidirecionales sencillas – sencillas
A (0..1) ↔ (0..*) B A (0..1) ↔ (1..*) B A (0..1) ↔ (m..n) B A (1) ↔ (0..*) B A (1) ↔ (1..*) B A (1) ↔ (m..n) B	Asociaciones bidireccionales sencillas – múltiples
A (0..*) ↔ (0..1) B A (0..*) ↔ (1) B A (1..*) ↔ (0..1) B A (1..*) ↔ (1) B A (m..n) ↔ (0..1) B A (m..n) ↔ (1) B	Asociaciones bidireccionales múltiples – sencillas
A (0..*) ↔ (0..*) B A (0..*) ↔ (1..*) B A (0..*) ↔ (m..n) B A (1..*) ↔ (0..*) B A (1..*) ↔ (1..*) B A (1..*) ↔ (m..n) B A (m..n) ↔ (0..*) B A (m..n) ↔ (1..*) B A (m..n) ↔ (m..n) B	Asociaciones bidireccionales múltiples – múltiples

Tabla 3.2: Asociaciones bidireccionales.

4 EL PROBLEMA DEL CONTROL DE LAS ASOCIACIONES Y SU SOLUCIÓN

Autores como *James Noble* y *Scott Ambler* han estudiado la implementación de asociaciones en lenguajes orientados a objetos. Aunque sus soluciones sean distintas y las aproximaciones realizadas están hechas desde distintos puntos de vista se coincide en que los lenguajes OO de hoy en día, no soportan de una forma directa la gestión de las relaciones entre clases.

Noble realiza un estudio teórico general presentando varias aproximaciones según los casos pero por no introducirse en la implementación práctica, no aporta ninguna sistematización del problema [5]. *Ambler*, por el contrario, hace una disertación práctica y concreta (asociaciones *UML* en *Java*) pero deja al usuario de las clases el control real de las restricciones de una asociación [8] [9] [10] [11].

Antes de meternos de lleno en la implementación de las asociaciones *UML* en *Java*, vamos a hacer un estudio más profundo del problema en general, sobre todo, para poder encontrar una solución uniforme a todos los casos que se han presentado.

4.1 Implementación de las asociaciones

Primero, habrá que empezar por ver cómo se van a implementar e identificar las asociaciones de una clase. Es fácil intuir que una asociación va a implicar un atributo en el que se almacenan los enlaces y unos métodos que controlan el establecimiento, acceso y eliminación de los enlaces en función de las restricciones impuestas por el diseño de la asociación. Sería conveniente agrupar estos métodos y atributos de alguna manera para, así, identificar más fácilmente, dentro de una clase, qué pertenece al control de la asociación y qué a los atributos y métodos propios de la clase. De este modo, la asociación se implementará en una clase interna.

Una clase interna no es más que una clase que se define y se encapsula dentro de otra clase. Son varios los lenguajes de programación OO que permiten trabajar con clases internas, y entre ellos, *Java*. La clase interna nos proporciona la ventaja de que la asociación queda encapsulada y perfectamente identificada dentro de las clases que participan en la asociación. Además, con una clase interna se hace sencillo el conocimiento de sus instancias puesto que se pueden definir como atributos de la clase que la embebe y en cualquier caso, éstas no se pueden crear fuera de la clase agrupadora.

Se puede barajar la posibilidad de representar una asociación en una clase externa. En este caso, nos tenemos que asegurar bien de que los extremos de la asociación son capaces de conocer las instancias gestoras de la asociación. Para ello, lo ideal es que las instancias de las clases gestoras de la asociación se representen en atributos internos. Sin embargo, pensamos que una clase cuyas instancias van a ser utilizadas por una única clase, no tiene mucho sentido ponerla fuera de ésta pues da lugar a confusión y no aprovecha las ventajas que aporta la encapsulación. Un último motivo para justificar la ventaja de representar las asociaciones en clases internas frente a clases externas es la visibilidad. La visibilidad de una asociación puede ser privada y de una forma directa, podemos hacer que una clase interna sea privada y por tanto, que sólo acceda a ella la clase

agrupadora. Con clases externas, no sería tan directo el establecimiento de la visibilidad de la asociación[4].

Se podría haber optado también por ponerlo todo dentro de la misma clase y mezclarlo con el código propio de la clase. Esto traería la ventaja de que el código sería más simple de cara a la compilación, pero desde luego, no sería tan legible para los analistas como lo es separar la asociación en una clase interna. Además, complicaría el proceso de ingeniería inversa de pasar de código a diagrama de clases en *UML*. Ocurriría además que no podríamos usar un nombrado de métodos uniforme debido a que una clase puede tener más de una asociación con varias clases. Si esto fuese así, habría que distinguir en el nombre de cada método, sobre qué asociación concreta actúa, lo cual resultaría engorroso; aparte de no permitir un interfaz uniforme para el tratamiento de todas las asociaciones. Un último motivo, y quizás el más importante, es que hay que distinguir bien qué métodos pertenecen al funcionamiento propio de una clase y son representativos de ésta, puesto que son los que la definen, y cuáles son accesorios en el sentido de que sirven para controlar las especificaciones de diseño como en este caso, las asociaciones.

Parece pues, que lo más práctico a todos los efectos es implementar el control de las asociaciones en clases internas; frente a la implementación de atributos y métodos dentro de las clases extremas de una asociación o a la implementación del control de la asociación en clases separadas.

4.2 Los métodos necesarios

Después, habrá que ver qué métodos son necesarios, y si realmente podemos usar los mismos nombres de métodos (aunque la implementación sea distinta) en los cuatro grupos que vamos a estudiar.

Por lo pronto, hay tres tipos de métodos básicos que servirán para **establecer** enlaces, **borrar**los y **acceder** a ellos. No obstante, luego veremos que también son necesarios otros métodos adicionales para controlar bien las restricciones de las asociaciones, pero de momento nos centraremos en estos tres por ser los fundamentales.

Ahora bien, para implementar dichos métodos hay que hacerse una pregunta básica: ¿cómo se va a hacer el control de las restricciones?

4.3 El problema del control de la multiplicidad

Para ver el control de las restricciones fundamentalmente nos vamos a centrar en la multiplicidad.

4.3.1 Multiplicidad máxima

La multiplicidad máxima no presenta mucha problemática puesto que simplemente hay que asegurarse de que nunca existan más enlaces de los permitidos. Por tanto, los métodos que se encarguen de establecer enlaces entre dos clases tendrán que mirar que no se exceda de la multiplicidad máxima. Cuando esto sea así, entonces no habrá ningún problema de asignar los

[4] El tema de la visibilidad de las asociaciones es complejo y ambiguo por lo que va a merecer un capítulo aparte "*Visibilidad en las asociaciones*" (capítulo 13).

nuevos enlaces y si no es así, no se hará nada y se indicará de alguna manera mediante un valor de retorno.

4.3.2 Multiplicidad mínima

En cambio, la multiplicidad mínima no es tan fácil de controlar y son varios los problemas que nos podemos encontrar.

Imaginemos por lo pronto que tenemos una asociación bidireccional que es obligatoria en ambos lados, por ejemplo: '*A (1)* ↔ *(1) B*'. Dicha asociación especifica que los objetos '*A*' están siempre obligatoriamente enlazados con un objeto '*B*' y viceversa. Ahora bien, cuando el programa final que utiliza estas dos clases empiece a funcionar, no podrá crear a la vez objetos '*A*' y '*B*' y enlazarlos en una única operación. Tendrá que ir por partes: primero crear una instancia, luego la instancia del otro tipo y por último enlazarlos. Esto quiere decir que, aunque sea por un instante de tiempo mínimo, no siempre podremos asegurar que un objeto cumpla con las restricciones de multiplicidad mínima.

Este problema podría tener una solución que aunque compleja podría valer. Consistiría en crear los objetos '*A*' y '*B*' y aunque sería imposible a la vez, sí sería posible hacerlo de una forma atómica en la que se instanciasen ambos objetos y quedasen enlazados. Esto se haría pasando los parámetros necesarios para crear un objeto '*B*' al constructor de la clase '*A*' (o al revés). De esta forma, cuando se crease un objeto '*A*', éste inmediatamente podría mandar crear un objeto '*B*' y establecer el correspondiente enlace.

La solución es compleja y presenta varios problemas. Uno es que el constructor de un objeto puede recibir varios parámetros e incluso puede existir más de una definición de él en cuyo caso, si además tenemos que especificar los parámetros necesarios para crear el objeto del otro lado de la asociación, las combinaciones y definiciones de constructores se multiplican y se hacen extraordinariamente complejos. Recordemos que estamos buscando automatización y sería difícil conjugar la creación de objetos (aún suponiendo que se nos dan los constructores hechos) con el código adicional para el establecimiento del enlace. La cosa se podría agravar aún más si la multiplicidad mínima es de más de uno y estos enlaces adicionales, aunque fuesen de la misma clase, requiriesen de distinto constructor.

Pero además tenemos otro problema. Suponiendo que se pudiese resolver el establecimiento de los enlaces necesarios nada más crear las instancias, no podemos asegurar que siempre vayamos a poder tener al mismo tiempo los parámetros de creación de los objetos '*A*' y '*B*' de una asociación. Puede ocurrir que primero se nos pida crear una instancia '*A*' y hacer varias operaciones con ella y, en función de los resultados o de los parámetros introducidos por un usuario final, que se establezca el enlace con unas instancias de '*B*' u otras.

Ahora supongamos que tenemos una asociación con multiplicidades obligatorias en ambos lados y se ha conseguido establecer correctamente todos los enlaces en las instancias. Podría ocurrir que se necesitase cambiar dinámicamente los enlaces. Sería fácil hacer el reemplazo de un enlace en un lado de la asociación, pero en el otro, tendríamos que romper con las restricciones impuestas en el diseño o asignarle nuevos enlaces inmediatamente. En la Figura 4.1 se ve gráficamente la situación de partida de dos instancias 'a_1' y 'b_1' enlazadas en una asociación bidireccional uno a uno.

Figura 4.1: Una instancia 'a_1' de '*A*' está enlazada con una instancia 'b_1' de '*B*'.

En la Figura 4.2, se produce un cambio dinámico de enlace. En ella, la instancia 'a_1', pasa a enlazarse con la instancia 'a_2' y deja a 'b_1' suelta, sin enlazar, cuando la definición de la asociación especifica que esto no puede ocurrir.

Figura 4.2: La instancia 'a_1' de '*A*' se enlaza con una instancia 'b_2' de '*B*' y deja a la instancia 'b_1' de '*B*' sin cumplir con las restricciones de multiplicidad.

Aquí, se ve claramente que el cambio dinámico de enlaces no es trivial y máxime si queremos que en todo momento se cumpla con las restricciones de la asociación en todos los objetos de '*A*' y de '*B*'.

Suponiendo que pudiésemos asignar a 'b_1' un nuevo enlace, el problema anterior se podría solucionar implementando la posibilidad de intercambiar enlaces de una forma atómica pero la dificultad es poner un límite al número máximo de intercambios. Es fácil hacer un método de intercambio ('*switchLinks*') que cambie los enlaces '$a_1 \leftrightarrow b_1$', '$a_2 \leftrightarrow b_2$' por los enlaces '$a_1 \leftrightarrow b_2$', '$a_2 \leftrightarrow b_1$' pero esto puede no ser suficiente para un usuario que necesite hacer intercambio entre tres, cuatro o más grupos de enlaces. Sería difícil establecer un método que nos permita de una forma genérica hacer todas las combinaciones de intercambios posibles o implementar absolutamente todas las necesidades que se nos ocurran, puesto que hipotéticamente, pueden llegar a ser infinitas.

Otro problema adicional ocurre cuando ya no necesitamos las instancias y haya que eliminarlas. Nos podemos encontrar con dos situaciones pero que en el fondo nos van a llevar al mismo escollo: dejaremos por unos instantes (aunque sean mínimos) que dos objetos de tipo '*A*' y '*B*' puedan no cumplir con las restricciones de multiplicidad mínima.

El primer caso ocurre cuando haya que eliminar ambos objetos de cada extremo de un enlace. En este caso, al tratarse de dos objetos independientes, si se elimina uno primero, el otro quedará en estado no válido pues tendrá enlazado un objeto que realmente no existe o se romperá el enlace primero pudiendo dejar al objeto del otro extremo aún sin eliminar y sin cumplir con las restricciones de multiplicidad mínima.

El segundo caso ocurre cuando haya que eliminar sólo uno de las dos instancias de un enlace. En esta situación, al objeto que permanece habrá que asignarle un nuevo enlace si la asociación así lo exige. Será complejo conseguir de una forma atómica la eliminación del enlace y

del objeto que ya no se necesita, y el establecimiento de un nuevo enlace para la instancia que permanece.

Centrándonos en *Java*, ocurre que en dicho lenguaje no existen los destructores y es el *garbage collector* el encargado de eliminar las instancias que ya no se usan. En este caso, el problema de la destrucción de objetos enlazados se reduce a hacer de una forma atómica la eliminación del enlace y la asignación de un nuevo enlace para la instancia que fuese a permanecer si la hubiese. Se necesitaría así, también métodos para poder realizar nuevos enlaces y dejar instancias sueltas que el *garbage collector* pudiese eliminar. La principal dificultad que nos encontramos aquí es el poder asegurarnos que las instancias que se quedasen sin enlazar y por tanto, sin cumplir con las restricciones de multiplicidad, quedasen sin que ninguna otra instancia las referenciase para que pudiesen ser realmente eliminadas por el recolector de basura.

4.3.3 Conclusión al problema de la multiplicidad

La conclusión de todos los problemas que se han presentado es que es impredecible el comportamiento dinámico de los enlaces, por lo que no basta con establecer métodos que establezcan y eliminen enlaces de una forma controlada. Se requieren también métodos de reemplazo de enlaces y eliminación controlada de objetos enlazados cuya complejidad es prácticamente inabordable puesto que las necesidades de un usuario de una asociación pueden ser infinitas e incluso imposible de conocer por él mismo de antemano.

Aún así, supongamos que sí se conociese de antemano todas las necesidades del usuario de las clases '*A*' y '*B*' respecto a los cambios dinámicos de enlaces posibles, y que fuéramos capaces de proporcionar la implementación de la asociación con todos los métodos necesarios para satisfacer dichas necesidades. Tal implementación, por muy bien que funcionase y abarcase todas las situaciones, sería compleja de utilizar, y un usuario podría verse abrumado por la cantidad de métodos con pequeñas diferencias entre sí que tendría que conocer. Es decir, una clase que tenga implementada una asociación de esta forma, no sólo sería compleja de desarrollar, sino también de utilizar.

4.4 La solución empleada

La solución que vamos a presentar pasa por no ser tan restrictivo pero no por ello, como veremos, vamos a dejar de cumplir con el significado pleno de lo que es una asociación y sus restricciones.

Para facilitar las cosas, vamos a ver más a fondo cual es el significado de la multiplicidad mínima en una asociación. El que una asociación tenga una multiplicidad mínima obligatoria de *m* (de '*A*' a '*B*' p.e.), ¿significa realmente que '*A*', en todo momento, va a tener como mínimo *m* enlaces con '*B*', es decir, desde que '*A*' se crea hasta que se destruye? La respuesta es no, puesto que lo que realmente significa es que '*A*' en algún momento dado va a acceder a sus enlaces y, es en ese momento, cuando tiene que tener como mínimo *m* enlaces. Cuando no se accede a ellos, no es necesario que los enlaces estén establecidos correctamente.

"No es necesario que una instancia cumpla con las restricciones de multiplicidad mínima en sus enlaces durante todo su tiempo de vida. Sólo se requiere que las condiciones de multiplicidad mínima se satisfagan en los momentos en los que se intenta acceder a los enlaces."

Teniendo en cuenta esta interpretación de las asociaciones obligatorias, el control de las restricciones se hace mucho más sencillo.

En la solución que vamos a presentar, el establecimiento o la adición de enlaces se permite siempre que no se exceda de la multiplicidad máxima. Por el contrario, se permiten eliminar todos los enlaces que se quiera, sin tener en cuenta la multiplicidad mínima. Sólo cuando se intente acceder a los enlaces se comprobará si realmente se está cumpliendo con las restricciones impuestas por la especificación de la asociación, en concreto, con las restricciones de multiplicidad. En caso de no cumplirse, no se permitirá acceder a los enlaces produciéndose una excepción. De esta forma, el cambio dinámico de los enlaces es bastante sencillo puesto que no hay problema de eliminar enlaces y asignar nuevos en la medida que se necesite.

Aquí, hay que advertir que para el cambio dinámico de enlaces, primero habrá que eliminar los viejos y luego establecer los nuevos puesto que si se hace al revés, nos podemos encontrar con que no se nos permite establecer los nuevos, al excederse la multiplicidad máxima (la cuál sí va a estar controlada).

Con este planteamiento, cuando se acceda a los enlaces, habrá que asegurarse que existe el número de enlaces adecuado por lo que habrá que proporcionar métodos para controlar si se está cumpliendo o no con las restricciones impuestas en la asociación. Al fin y al cabo será siempre responsabilidad directa del usuario de la clase de la asociación su establecimiento correcto.

Podría decirse que, por motivos de simetría, sería conveniente que los métodos de establecimiento de enlaces no tuviesen en cuenta la multiplicidad máxima y por tanto, que fuera válido crear más enlaces de los especificados. De este modo, el exceso de enlaces sólo habría que tenerlo en cuento a la hora de acceder a ellos, en cuyo caso, se impediría su acceso. Se buscaría, por tanto, simetría con los métodos de eliminación de enlaces, puesto que permiten romper enlaces incluso cuando se deja de cumplir con las restricciones de multiplicidad mínima. No somos partidarios de hacerlo así porque pensamos que la multiplicidad máxima es fácil de controlar, y que se debe poner a disposición del usuario el máximo control posible de las restricciones, siempre y cuando no se ahogue y se dificulte la gestión de los enlaces. Ya hemos visto como hay que ser laxo con la multiplicidad mínima pero no hay motivo para serlo con la máxima. La única ventaja que vemos que aportaría no controlar la multiplicidad máxima en el establecimiento de los enlaces es que para su cambio dinámico, daría igual el orden de establecer los nuevos y eliminar los viejos.

5 UN INTERFAZ UNIFORME

Ahora que sabemos cómo vamos a crear, eliminar y cambiar enlaces, tenemos que definir los métodos necesarios, así como sus entradas y salidas. Se busca un interfaz uniforme para todos los tipos de asociaciones por aquello de que resultará muy cómodo (de cara a su utilización) no tener que estar andando con distintos juegos de métodos, según el tipo de asociación con el que estemos tratando en cada ocasión. No obstante, esto no se puede conseguir al cien por cien, puesto que en algunos casos se trata con multiplicidades sencillas y en otros, con múltiples. Por tanto, a veces se trabaja con un único enlace, y otras veces existe la posibilidad de trabajar con un conjunto de ellos. Pero esta pequeña diferencia, como veremos, sólo afecta a los parámetros de entrada, y no al nombre de los métodos.

5.1 Resumen del código para la gestión de una asociación

En toda clase '*Source*' en la que se especifique una asociación binaria con otra clase '*Target*' se genera una clase interna que controla la asociación y un atributo cuyo tipo es de dicha clase. A través de este atributo se establecen y se eliminan los enlaces y se accede a ellos. En la Tabla 5.1 se muestra un resumen del código generado automáticamente con la definición de las clases, atributos y cabeceras de los métodos. Nos centraremos especialmente en la clase interna que gestiona la asociación que de forma genérica llamaremos '*Association*'.

Por clase '*Source*' se entiende la clase de la cual parte una asociación, mientras que por clase '*Target*' se entiende la clase a la que llega una asociación. De este modo, en las asociaciones unidireccionales la clase '*Source*' será la clase que está asociada con una clase '*Target*' pero al ser la navegabilidad en ese único sentido, ésta última no tendrá conocimiento de la otra (salvo que tenga definida otra asociación distinta o lo haga de algún otro modo ajeno a la asociación). Debido a que en las asociaciones bidireccionales la navegabilidad es en doble sentido, según el sentido en que nos estemos refiriendo en cada momento, la clase '*Source*' y '*Target*' cambiarán de papel.

Por el momento, vamos a presentar las cabeceras de los métodos y atributos que se necesitarán en la clase '*Association*' así como las declaraciones necesarias en la clase '*Source*'. Los nombres empleados son genéricos. Posteriormente, cuando presentemos el código final, lo haremos a modo de plantillas. En dichas plantillas, será fácil sustituir e identificar los nombres '*Source*', '*Target*', '*targetRole*', etc. por los correspondientes, tal y como hayan sido introducidos en el diseño de la asociación.

No se definen siempre todos los métodos y constantes que mostramos en la Tabla 5.1, puesto que algunos, sólo existirán en las clases '*Source*' cuya multiplicidad en '*Target*' es sencilla y otros, sólo existirán en clases '*Source*' con multiplicidad mayor que uno en '*Target*'. Dichos métodos pueden identificarse fácilmente en la tabla.

Los métodos de la clase '*Association*' se pueden agrupar en tres categorías:

1. **Definición de la asociación**: Métodos para conocer la especificación de diseño de la asociación ('*isBidirectional*', '*isMandatory*', '*isMultiple*', '*getMIN*' y '*getMAX'*).

2. **Estado de los enlaces**: Métodos para conocer el estado de la asociación ('*isValid*', '*numberOfLinks*').

3. **Manipulación de los enlaces**: Métodos de comprobación, acceso, establecimiento y eliminación de enlaces ('*test*', '*get*' , '*add*' y '*remove*').

```
class Source {
  final Association targetRole = new Association();
  class Association {
    private final static boolean mandatory; // Sólo para multiplicidades sencillas
    private final static long MIN; // Sólo para multiplicidades mayor que uno
    static final int MANY = -1;
    private final static long MAX; // Sólo para multiplicidades mayor que uno
    boolean isBidirectional();
    boolean isMandatory();
    boolean isMultiple();
    long getMIN();
    long getMAX();
    boolean isValid();
    long numberOfLinks();
    boolean test(Target query_link);
    boolean test(Collection query_links); // Sólo para multiplicidades mayor que uno
    Target get() throws Exception; // Sólo para multiplicidades sencillas
    Collection get() throws Exception; // Sólo para multiplicidades mayor que uno
    boolean add(Target new_link);
    boolean add(Collection new_links); // Sólo para multiplicidades mayor que uno
    int remove();
    int remove(Target old_link);
    int remove(Collection old_links); // Sólo para multiplicidades mayor que uno
  }
}
```

Tabla 5.1: Interfaz del extremo de una asociación.

5.2 Condiciones de uso

5.2.1 Clase '*Source*' y atributo '*targetRole*'

En cada clase '*Source*', habrá declarada una clase interna del mismo estilo que '*Association*' por cada asociación que se especifique en el diseño en la que '*Source*' se vea involucrada. Asimismo, habrá un único atributo '*targetRole*' (de tipo '*Association*') por cada clase interna '*Association*' y para evitar manipulaciones no permitidas de éste, se declara como *final*.

Hay que advertir que desde dentro de la clase '*Source*' no se puede evitar la creación de nuevas instancias de '*Association*' pero esto no debe hacerse como condición para que se pueda garantizar un correcto control de las asociaciones.

5.2.2 Constantes de la clase '*Association*'

Todas las constantes de '*Association*' son privadas y tienen que estar especificadas en tiempo de compilación. Sirven para especificar la multiplicidad de '*Target*' respecto a '*Source*'. El usuario de la clase '*Source*', en general, no debe tener acceso directo a ellas.

5.2.2.1 private boolean mandatory

Esta constante se utiliza para asociaciones con multiplicidad sencilla en el extremo '*Target*'. Si se le asigna el valor *true*, se está especificando que todas las instancias de '*Source*' a '*Target*' tienen que tener siempre un enlace (multiplicidad 1..1) y con el valor *false*, se está especificando que los enlaces son opcionales (multiplicidad 0..1).

- Sólo se declara en la clases '*Source*' cuya multiplicidad en el extremo es sencilla.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*'. Se recomienda que para conocer si la multiplicidad es *mandatory* o no, se utilice el método '*isMandatory*' puesto que la constante '*mandatory*' es para uso interno en la clase '*Association*'.

5.2.2.2 private long MIN

Especifica la multiplicidad mínima de '*Target*' respecto a '*Source*'.

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*' pero se recomienda que para conocer la multiplicidad mínima se utilice el método '*getMIN*' puesto que es una constante de uso interno en la clase '*Association*'.
- Su valor tiene que ser mayor o igual que cero.

5.2.2.3 int MANY

La constante '*MANY*' sirve para representar la multiplicidad muchos de *UML* (*). Con el método '*getMAX*' veremos para qué puede ser útil el conocimiento de esta constante.

- La constante '*MANY*' se declarará siempre, incluso cuando se trate con multiplicidades sencillas.

5.2.2.4 private long MAX

Especifica la multiplicidad máxima de '*Target*' respecto a '*Source*'. Junto con '*MIN*', sirve para establecer la multiplicidad en las asociaciones múltiples. En las asociaciones sencillas, no se han utilizado estas dos constantes (aunque podría ser deseable a efectos de uniformidad de interfaz) para evitar que por accidente se especifiquen valores que se salgan de los rangos (0..1) y (1..1), puesto que el código está preparado para tratar un único enlace y no muchos.

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*' pero se recomienda que para conocer la multiplicidad máxima se utilice el método '*getMAX*' puesto que es una constante de uso interno en la clase '*Association*'.
- Su valor tiene que ser mayor o igual que '*MIN*'.
- Cuando la multiplicidad máxima sea de muchos, a '*MAX*' se le asignará la constante '*MANY*'.

5.2.3 Métodos de definición de la asociación

Los métodos que aquí se detallan sirven para poder conocer la especificación de las asociaciones.

5.2.3.1 boolean isBidirectional()

Dice si la asociación es bidireccional (verdadero) o navegable sólo desde la clase fuente (falso).

5.2.3.2 boolean isMandatory()

El método devuelve verdadero cuando la asociación es obligatoria (multiplicidad mínima mayor que 0) para la clase '*Source*'.

5.2.3.3 boolean isMultiple()

Devuelve verdadero cuando la multiplicidad en el destino con respecto a la fuente es múltiple (multiplicidad máxima mayor que 1) y falso cuando es sencilla (multiplicidad máxima igual a 1).

5.2.3.4 long getMIN()

Devuelve la multiplicidad mínima del extremo '*Target*' respecto a '*Source*'.

5.2.3.5 long getMAX()

Devuelve la multiplicidad máxima del extremo '*Target*' respecto a '*Source*'.

- La multiplicidad muchos (*) se especifica con la constante '*MANY*', de forma que si la multiplicidad máxima es muchos, en la invocación a este método se obtendrá el mismo valor que el especificado en '*MANY*'.

5.2.4 Métodos para el conocimiento del estado de los enlaces

En este apartado, definimos los métodos de la segunda categoría, que sirven para conocer en qué estado se encuentran los enlaces.

5.2.4.1 boolean isValid()

Devuelve verdadero cuando existe el número correcto de enlaces de acuerdo con la multiplicidad de la asociación en el sentido de '*Source*' a '*Target*' y, por tanto, se considera que se está cumpliendo con la especificación de diseño de la asociación en el extremo '*Source*'.

Esta función está pensada para obtener la validez de todas las restricciones de diseño de la asociación. No obstante, como los problemas a los que nos enfrentamos según los objetivos marcados en este trabajo se centran en la multiplicidad, de momento, '*isValid*' sólo comprueba esta restricción. Futuras versiones de este método, podrían realizar otras comprobaciones adicionales.

- En las asociaciones bidireccionales, que se obtenga el valor verdadero no implica necesariamente que los enlaces estén bien establecidos, puesto que habría que comprobar también el sentido contrario, desde las instancias de '*Target*' a '*Source*'.

5.2.4.2 long numberOfLinks()

Devuelve el número de enlaces que la instancia '*Source*' tiene establecidos con instancias '*Target*'.

5.2.5 Métodos para la comprobación, acceso, establecimiento y eliminación de los enlaces

En la última categoría se especifican los métodos de gestión de la asociación.

5.2.5.1 boolean test(Target query_link)

Dice si existe enlace con la instancia de '*Target*' que se pasa como parámetro de entrada.

- Si de entrada se introduce una variable con valor *null*, se devuelve verdadero sólo si no hay establecido ningún enlace.

5.2.5.2 boolean test(Collection query_links)

Dice si existe enlace con todas las instancias '*Target*' que se pasan como parámetro de entrada contenidas en una colección de objetos[5].

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Si algún elemento de la colección de entrada no es de tipo '*Target*' se devuelve falso.
- Si la colección de entrada está vacía, se devuelve verdadero.
- Si de entrada se introduce una variable con valor *null*, se devuelve verdadero si no hay establecido ningún enlace.

5.2.5.3 Target get() throws UnsetAssociationException

Obtiene la única instancia '*Target*' con la que se tiene establecido un enlace.

- Sólo se declara en la clases '*Source*' cuya multiplicidad en el extremo es sencilla.
- Lanza la excepción '*UnsetAssociationException*'[6] si no se cumple con las restricciones de diseño de la asociación en '*Source*', es decir, si la llamada a '*isValid*' devuelve falso.
- Si no existe ningún enlace pero se cumplen con las restricciones de multiplicidad mínima, se devuelve el valor *null*.

5.2.5.4 Collection get() throws UnsetAssociationException

Obtiene todas las instancias '*Target*' con las que se tiene establecido un enlace.

- Sólo se declara en clases '*Source*' cuya multiplicidad en el extremo es mayor que uno.
- Lanza la excepción '*UnsetAssociationException*' si no se cumple con las restricciones de diseño de la asociación en '*Source*', es decir, si la llamada a '*isValid*' devuelve falso.

---

[5] Siempre que tratemos con un conjunto de enlaces funcionaremos con el interfaz *Collection* de *Java* para que así, el usuario tenga la máxima flexibilidad posible cuando tenga que tratar con varios enlaces simultáneamente y pueda funcionar con el tipo de colección que mejor se adapte a su situación: *Vector*, *ArrayList*, *HashSet*, etc.

[6] La veremos más adelante en el capítulo 8 "*Excepción de asociación no establecida*".

- Si no existe ningún enlace pero se cumplen con las restricciones de multiplicidad mínima, se devuelve una colección vacía, sin elementos.

5.2.5.5 boolean add(Target new_link)

Añade un enlace con la instancia de tipo '*Target*' que se introduce como parámetro de entrada.

- Si '*Source*' ya tiene establecido el enlace con la instancia de entrada, el método devuelve verdadero pero sin realizar ningún cambio.
- Si '*Source*' ya tiene establecidos el número máximo de enlaces permitidos, no se hace nada y se devuelve falso. En este caso, la única forma de establecer enlace con la instancia '*new_link*' es borrando otro enlace (mediante el método '*remove*') e invocar a '*add(new_link)*' (en ese orden). Esto último hay que tenerlo especialmente en cuenta en las asociaciones sencillas puesto que en ellas, si ya hay establecido un enlace, '*add*' no funciona hasta que se elimine.
- Si el parámetro de entrada es *null* no se hace nada y se devuelve falso.
- En las asociaciones bidireccionales, debido a que hay que establecer un doble enlace (uno para cada uno de los dos sentidos de la asociación), primero se comprueba que se dan las condiciones necesarias para establecer el enlace de '*Source*' a '*Target*'. Si se puede, se establece y se comprueba que se puede establecer enlace en el otro sentido (de '*Target*' a '*Source*'). Para ello, se hace internamente una llamada al método 'add' del otro extremo, es decir, del atributo '*sourceRole*'[7] de '*new_link*' para que establezca enlace con '*Source*'. Si '*new_link.sourceRole.add*' responde diciendo que no ha establecido el enlace, entonces se elimina el enlace de '*Source*' a '*Target*' que se acababa de establecer y se devuelve falso. [8]
- Si el enlace queda correctamente establecido, se devuelve verdadero.

5.2.5.6 boolean add(Collection new_links)

Establece un enlace con cada una de las instancias de tipo '*Target*' contenidas en la colección de entrada.

- Sólo se declara en las clases '*Source*' cuya multiplicidad con el extremo '*Target*' es mayor que uno.
- Si alguna instancia de la colección de entrada no es de tipo '*Target*' se devuelve falso y no se produce ningún cambio.
- Si el parámetro de entrada es *null* no se hace nada y se devuelve falso.
- El número de enlaces final no puede ser mayor que los permitidos por la multiplicidad máxima, en cuyo caso se devuelve falso y se dejan los enlaces de '*Source*' tal y como estaban antes de la llamada.

---

[7] '*Target*' sigue la misma estructura que '*Source*', por lo que el atributo que gestiona los enlaces en '*Target*' es '*sourceRole*' en vez de '*targetRole*'.

[8] Este párrafo, en definitiva está diciendo que en las asociaciones bidireccionales, mediante una única llamada a '*add*' de '*Source*', se consigue enlace en los dos sentidos, es decir, se consigue enlace bidireccional.

- En las asociaciones bidireccionales, se tiene que lograr establecer enlace en el sentido inverso con todos los extremos opuestos, desde las instancias '*Target*' a la de '*Source*'. Por tanto, a igual que ocurría en la definición del método '*add*' anterior, por cada instancia '*Target*' contenida en '*new_links*' se hace una llamada al método '*add*' definido en '*new_link[x].sourceRole*' que establece enlace en el sentido opuesto. Si una sola de estas llamadas falla, el método devuelve falso y deja los enlaces tal y como estaban antes de la llamada.

- Sólo si todos los enlaces quedan correctamente establecidos, se devuelve verdadero.

5.2.5.7 int remove()

Elimina todos los enlaces que contiene '*Source*'.

- Devuelve "cero" si al borrar los enlaces se siguen cumpliendo las restricciones de multiplicidad mínima en '*Source*'.

- Devuelve "uno" si al borrar los enlaces se deja de cumplir con las restricciones de multiplicidad en '*Source*'.

- En las asociaciones bidireccionales, devuelve "dos", y no "cero", si en alguno de los enlaces destino se han dejado de cumplir con las restricciones de multiplicidad mínima respecto al extremo '*Source*', es decir, en el sentido de '*Target*' a '*Source*'[9].

- En las asociaciones bidireccionales, devuelve "tres", y no "uno", si tanto en el extremo '*Source*' como en alguno de los antiguos enlaces '*Target*' se deja de cumplir con las restricciones de multiplicidad mínima.

5.2.5.8 int remove(Target old_link)

Elimina el enlace establecido con la instancia '*Target*' de entrada.

- En las asociaciones de multiplicidad sencilla en el extremo '*Target*' respecto a '*Source*', éste método se declara por mantener un interfaz uniforme pero prácticamente equivale a 'remove*()*' puesto que se trata de borrar el único enlace que contiene '*Source*'.

- Los valores de retorno del método '*remove*' anterior se aplican aquí también.

- Además, devuelve "–1" si no existe enlace con las instancia '*Target*' de entrada y por tanto, no se puede borrar el enlace.

- Devuelve también "–1 " si de entrada se introduce el valor *null*.

5.2.5.9 int remove(Collection old_links)

Borra los enlaces establecidos con todas las instancias '*Target*' contenidas en la colección de entrada.

---

[9] La eliminación de enlaces en las asociaciones bidireccionales se consigue con la misma técnica que se ha explicado en '*add*'. Nuevamente, se trata de borrar el enlace en el sentido de '*Source*' a '*Target*' y de invocar al extremo '*Target*' pero esta vez, para que borre el enlace en el sentido de '*Target*' a '*Source*'.

- Sólo se declara en las clases '*Source*' cuyo extremo tiene multiplicidad mayor que uno.
- Los valores de retorno del primer método '*remove*' se aplican aquí también.
- Se ignoran aquellos elementos de la colección que no sean de tipo '*Target*' o con los que no se tenga ningún enlace. Es decir, en la colección de entrada sólo se tienen en cuenta las instancias '*Target*' con las que '*Source*' tiene establecido enlace.
- Si algún elemento de la colección de entrada no es de tipo '*Target*', se obtiene como valor de retorno "–1".
- Si no se llega a borrar ningún enlace ('*old_links*' es *null* o no contiene ningún enlace existente) se devuelve "–1".

6 IDEAS ALTERNATIVAS PARA IMPLEMENTAR ASOCIACIONES

Para llegar a la solución que hemos presentado se han pensado previamente otras formas que vamos a exponer a continuación. Dichos métodos se han descartado por diversos motivos, pero no por eso pensamos que sean malas soluciones o que puedan resultar ventajosas en ciertos aspectos. Además, si bien no se han utilizado tal y como las vamos a presentar, hay que tener en cuenta que han servido de base y de apoyo para llegar a la solución final planteada.

6.1 Atributos y métodos de control de la asociación en la propia clase

Lógicamente, la primera solución que se nos puede ocurrir es establecer un atributo por cada asociación donde almacenar los enlaces. Tiene como ventaja que es la más fácil de construir, pero exige que el control de las restricciones sea hecho directamente por el programador, con lo que la asociación queda distribuida a lo largo de todo el código de la clase (e incluso puede que fuera). De este modo, aparte de que se hace fácil que por error se incumplan con las restricciones impuestas, la asociación no resulta legible en absoluto, y es muy difícil determinar a partir del código de la clase cuales son sus relaciones y su definición o especificación.

Para hacerlo todo más fácil, se pueden implementar métodos de control y acceso a los atributos que almacenan los enlaces, y hacer obligatorio el uso de esos métodos en lugar de controlar directamente dichos atributos. Si se hace así, se gana en legibilidad y en facilidad de reconocimiento automático de las asociaciones, pero aunque esto resulta bastante parecido a la solución que se ha presentado, lógicamente, no es tan limpio y claro como representar cada asociación en una clase interna, donde todo queda mejor encapsulado.

Autores como James Noble [5] y James Rumbaugh [6] vienen apuntando desde hace tiempo que los lenguajes de programación orientada a objetos no soportan nada bien la implementación de las relaciones. En las soluciones que estos autores aportan para relaciones sencillas, todos invitan al uso de atributos básicos para almacenar relaciones uno a uno o colecciones de objetos para relaciones uno a muchos.

Por consiguiente, nuestra solución ha tomado de estos autores la idea del almacenamiento de los enlaces en atributos, y el control y acceso a estos mediante métodos de lectura y escritura.

6.2 Clases asociativas

Esta solución es parecida a la idea que plantea *UML* con la clase asociación y ha sido señalada, aunque con distinto nombre ("*relationship object*"), por el propio James Noble [5]. En principio, en *UML* se utiliza la clase asociación para almacenar atributos propios de ésta y no de las clases participantes en la relación.

La idea de la clase asociativa es que para representar en un lenguaje de programación una asociación, se ha de escribir una clase por cada asociación especificada y usar sus instancias para contener y controlar los enlaces.

Esta solución tiene una ventaja inmediata en un tema que no se ha resuelto en este trabajo y es que se hace sencillo la implementación de las clases asociación de *UML*. Debido a que los

enlaces se gestionan en clases separadas, éstas servirán también para almacenar atributos propios de la asociación.

Consideramos que esta solución resultaría buena para asociaciones unidireccionales, debido a que sólo la clase origen puede conocer sus enlaces y por tanto, sería ella la encargada de crear las instancias de la clase de asociación[10]. Ahora bien, en las asociaciones bidireccionales las dos clases participantes deben poder acceder a sus enlaces y con esta solución se haría complejo instanciar una clase asociativa y conseguir que ambas clases la conocieran y la utilizasen. Además, pensemos en una asociación de tipo muchos a muchos. ¿Qué instancias de la clase de asociación tendríamos que usar para controlar los enlaces? Es evidente que en las relaciones uno a muchos o uno a lo que sea, una única instancia de la clase asociativa nos puede servir para guardar todos esos enlaces en un único objeto, pero en las asociaciones muchos a muchos el control de las restricciones se hace distribuido, lo cual es prácticamente imposible de gestionar de este modo. Además, se complica también el conocimiento de todos los enlaces. Este problema nos lleva a la siguiente posible solución.

6.3 Clases asociativas estáticas

Usando el patrón de diseño *Singleton*, se puede crear una clase asociativa en la que tan sólo haya una instancia de dicha clase. A través de dicha instancia, se deben gestionar absolutamente todos los enlaces existentes entre las clases '*A*' y '*B*'. Una forma de implementar dicho patrón en *Java* es mediante la utilización de una clase estática [19]. Una clase estática no tiene instancias y por tanto, las clases participantes de cada asociación deben acceder directamente a la propia clase asociativa estática para establecer y conocer sus enlaces.

Esta solución es sencilla de implementar y consideramos que funciona suficientemente bien. Tiene la ventaja de que serviría también para implementar clases asociación si no sólo se usase para guardar enlaces, sino también atributos propios de cada enlace. Además, la asociación queda claramente establecida e identificada en una única clase, por lo que se hace sencillo también conocer en un momento dado todos los enlaces existentes entre todos los objetos '*A*' y '*B*'.

Uno de los motivos por los que se ha descartado es porque forma un cuello de botella al ser una solución centralizada. Si todos los enlaces entre todas las instancias de una asociación tienen que pasar a través de una única clase estática, será habitual en los programas distribuidos, o con múltiples hilos de ejecución funcionando al mismo tiempo, que un objeto tenga que esperar a que otros objetos terminen de realizar la gestión de sus enlaces antes de poder realizar la gestión de los propios.

6.4 Clases validadas

Las clases validadas es una solución que se ha meditado considerablemente pero que al final ha habido que descartar por su inviabilidad debido a las limitaciones de los lenguajes orientados a objetos. No obstante, en cierto modo sí que se ha utilizado su idea en la solución final que se ha presentado.

[10] Si esta clase asociativa se hiciese interna en la clase origen, obtendríamos precisamente la solución que hemos presentado.

La idea básica de una clase validada es una clase cuyos métodos y atributos dejan de ser accesibles cuando las instancias no cumplen unas condiciones determinadas o no se encuentran en un estado que se considere válido. Se pretendía que las clases tuvieran atributos y métodos para el control de los enlaces y que el resto de atributos y métodos (los propios de la clase) dejasen de funcionar cuando el estado de sus enlaces no cumpliesen con las restricciones de diseño de las asociaciones. De este modo, se puede permitir máxima flexibilidad a la hora de crear instancias y establecer enlaces pero se puede bloquear su uso cuando los enlaces no son correctos.

Una clase validada requeriría de una extensión a la sintaxis de *Java*. Veamos una propuesta en el Cuadro 6.1.

```
validated class A {

  rule {}

  A {}

  void method1() {}

  unvalidated method2() {}

  ...

}
```

Cuadro 6.1: Propuesta de clase validada en *Java*.

Las clases se definirían con la palabra reservada '*validated*' para indicar de que se trata de una clase validada. '*rule*' sería también una palabra reservada y contendría en un bloque la regla de validación, que sería una operación que devolviese un valor booleano. Todos los métodos y atributos (como '*method1*') serían por defecto validables, es decir, antes de ejecutarse, mirarían el valor devuelto por la regla de validación y si fuese falso, no se ejecutarían. Dichos métodos no necesitarían ser declarados de alguna forma particular y, salvo que se especificase lo contrario, serían de este tipo. Si en cambio, un método se declarase como '*unvalidated*', funcionaría siempre, independientemente del valor que se obtuviese en la regla de validación. Estos métodos no restringidos deberían ser precisamente aquellos que sirven para cambiar el estado de la clase y, por tanto, aquellos en los que a priori, no se espera que el estado de la clase sea válido.

Queda por determinar qué se haría en el caso de que un método no pudiese ejecutarse al no cumplirse la regla de validación. Lo normal, sería lanzar una excepción o un error.

Como *Java* no tiene esta particularidad, se han buscado formas de implementarla. Un primer intento de solución ha sido declarar como regla de validación un método tipo '*isValid*' que es lo primero que se ejecuta en el resto de los métodos. Si esta función devuelve falso, entonces o bien no se hace nada en el método o se lanza una excepción. El problema está en que se obliga a poner la regla de validación en cada método, lo cual, es bastante engorroso. Además, suponiendo que se pudiese conseguir que todos los métodos se implementasen con esa regla de validación, un programador debería estar muy atento cuando utilizase herencia en una clase de este tipo puesto que en los métodos nuevos que declarase o redefiniese, tendría que colocar también la regla de validación.

En resumen, la clase validada implementada con la sintaxis actual de *Java* resulta engorrosa y poco clara, pero creemos que añadir la sintaxis que se ha mostrado anteriormente o una similar sería útil no sólo para el control de restricciones de las asociaciones (cuya regla de validación sería comprobar si el número de enlaces es el correcto de acuerdo con la multiplicidad) sino también para clases especiales que sólo deban responder ante un estado concreto.

No obstante, en la solución presentada sí que se ha hecho uso de esta idea puesto que '*Association*' es una clase validada en cierto modo. Si una clase no tiene bien establecidos sus enlaces, no puede acceder a ellos a través del método '*get*' de '*Association*' porque se lanza la excepción '*unsetAssociationException*'. De esta forma, en nuestra solución, se permite establecer y eliminar enlaces en cualquier momento y de cualquier modo con la ventaja inmediata de haberlos dotado de máxima flexibilidad en su definición, pero el método '*get*' actúa como un método validado y así, se consigue además un uso perfectamente controlado de los enlaces.

PARTE II: IMPLEMENTACIÓN DE ASOCIACIONES BINARIAS

7 GENERACIÓN AUTOMÁTICA DEL CÓDIGO DE CONTROL DE LAS ASOCIACIONES

Conocido el interfaz para la gestión de las asociaciones, en este capítulo se va a explicar cómo se va a obtener, en la herramienta que se va a desarrollar, el código de control de una asociación una vez sea especificada. A modo de ejemplo aclarativo, en este apartado emplearemos la asociación de la Figura 7.1.

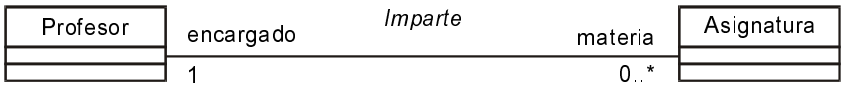

Figura 7.1: Un profesor puede impartir asignaturas.

7.1 Funcionamiento básico de la herramienta

La herramienta que se va a implementar debe desarrollar el código básico en *Java* para poder trabajar con las asociaciones y clases introducidas por el usuario. De este modo, cuando el usuario diseñe una asociación especificará los nombres de clase y de la asociación y los roles que juega cada clase, así como el tipo de asociación y demás restricciones.

La herramienta funcionará de la siguiente forma:

1. Leerá las especificaciones de diseño de una asociación.
2. Interpretará el tipo de asociación según las restricciones introducidas de navegabilidad y multiplicidad.
3. En función del tipo de asociación escogerá una plantilla de código.
4. En dicha plantilla, cambiará los nombres de los atributos y clases para que se adapten a las especificaciones introducidas por el usuario, y cambiará las constantes necesarias para que el código final sea coherente con las especificaciones.

7.1.1 Nombres claves en las plantillas

Por tanto, la clave está en las plantillas de código y en los valores introducidos por el usuario para poder generar los nombres y valores correctos (en lugar de los genéricos que tengan las plantillas), y obtener así un código legible. En las plantillas de la herramienta y en el código genérico que vamos a presentar en los siguientes apartados, nos vamos a encontrar unos nombres clave que tendrán que ser sustituidos en el momento de la generación del código concreto. Dichos nombres se identifican en las plantillas porque empiezan con el símbolo '%' y son:

- ***%Package***: Nombre completo del paquete de la clase *Java*.

- ***%Source***: Se referirá a la clase '*Source*', es decir, se sustituirá por la primera clase que especifique el usuario. La primera letra se pondrá en mayúscula, incluso aunque en el diseño el nombre de la clase estuviera con minúscula. Por ejemplo, '*Profesor*'.

- ***%Target***: Se referirá a la clase '*Target*', es decir, se sustituirá por la segunda clase que será la asociada con '*Source*'. También se pondrá la primera letra en mayúscula. Por ejemplo, '*Asignatura*'. Aquí, hay que tener en cuenta que en las asociaciones bidireccionales se necesita código en la clase '*Source*' que enlace con '*Target*' y al revés, por lo que dentro de las plantillas, *%Source* se referirá a la clase que implementa la plantilla y *%Target* a la clase asociada. Es decir, según qué clase estemos tratando, *%Source* y *%Target* pueden ser una clase o la otra.

- ***%targetRole***: Es el nombre de rol que juega la clase '*Target*' con respecto a '*Source*'. Por ejemplo, '*materia*'. En *UML*, tanto el nombre de rol como el nombre de asociación son opcionales, de modo que, en función de si están o no definidos estos nombres, el valor que usaremos para sustituir *%targetRole* en la generación del código será, en este orden de prioridad:

    1. nombre de rol: '*materia*'
    2. nombre de asociación y nombre de clase en minúscula: '*imparte_asignatura*'
    3. nombre de clase en minúscula: '*asignatura*'

- ***%sourceRole***: Es el nombre de rol que juega la clase '*Source*' con respecto a '*Target*'. Por ejemplo, '*encargado*'. A igual que '*targetRole*', no se requiere que sea introducido forzosamente y en caso de no ser especificado, se aplicarán los mismos criterios que con '*%targetRole*':

    1. nombre de rol: '*encargado*'
    2. nombre de la asociación y nombre de clase en minúscula: '*imparte_profesor*'
    3. nombre de clase en minúscula: '*profesor*'

- ***%mandatory***: En las plantillas de multiplicidad sencilla, se sustituirá este valor por '*true*' o '*false*' en función de si la asociación hace obligatorio o no la existencia de enlaces.

- ***%min_multiplicity*** y ***%max_multiplicity***: Estas palabras aparecen en las plantillas de multiplicidad mayor que uno y se deben sustituir por el valor asignado en la multiplicidad mínima y máxima. En caso de querer asignar multiplicidad *muchos* en *%max_multiplicity*, se debe sustituir por '*MANY*' (en mayúsculas).

- ***%AssociationKind***: Especifica el tipo de asociación. Una asociación convencional será reconocida con el valor '*Association*' pero como en seguida se va a ver, existe otro tipo de asociaciones a las que se le da un nombrado especial en este campo. Las adelantamos:

    - '*Association*': para asociaciones normales.
    - '*RefelexiveAssociation*': asociaciones reflexivas.
    - '*AggregationWhole*': parte contenedora de las agregaciones.

- '*AggregationPart*': parte contenida de las agregaciones.
- '*CompositionWhole*': parte contenedora de las composiciones.
- '*CompositionPart*': parte contenida de las composiciones..
- '*ReflexiveAggregationWhole*': parte contenedora en agregaciones reflexivas.
- '*ReflexiveAggregationPart*': parte contenedora en agregaciones reflexivas.
- '*ReflexiveCompositionWhole*': parte contenedora en composiciones reflexivas.
- '*ReflexiveCompositionPart*': parte contenida en composiciones reflexivas.

- ***%visibility***: En este campo se quiere especificar la visibilidad de los enlaces que puede ser, a igual que la notación *Java*, '*public*', '*private*', '*protected*' o nada para indicar que es *friendly* (o de paquete (*package*) según *UML 1.4* [3]). Cuando no se especifique la visibilidad, se entenderá que es pública (como en el ejemplo de la Figura 7.1).

7.1.2 Tipos de plantillas

Existen cuatro plantillas, que se corresponden con los cuatro grupos vistos en el capítulo 3 "*Tipos de asociaciones*":

1. '*SingleUnidirectional.java~*': para asociaciones unidireccionales de multiplicidad sencilla.
2. '*MultipleUnidirectional.java~*': para asociaciones unidireccionales de multiplicidad mayor que uno.
3. '*SingleBidirectional.java~*': para las clases de asociaciones bidireccionales cuyo extremo tiene multiplicidad sencilla.
4. '*MultipleBidirectional.java~*': para las clases de asociaciones bidireccionales cuyo extremo es mayor que uno en la multiplicidad.

Según el ejemplo de la Figura 7.1, para la clase '*Profesor*' tendríamos que coger la plantilla '*MultipleBidirectional.java~*' puesto que la asociación es bidireccional y un profesor puede impartir más de una asignatura. La constante '*MIN*' se sustituiría por "cero" y '*MAX*' por el valor '*MANY*'. Para la clase '*Asignatura*' cogeríamos '*SingleBidirectional*' puesto que una asignatura es impartida por un único profesor y la constante '*mandatory*' se le asignaría el valor "*true*". Si la asociación fuese unidireccional de '*Profesor*' a '*Asignatura*', sencillamente cogeríamos la plantilla '*MultipleUnidirectional.java~*' y la aplicaríamos a la clase '*Profesor*'.

7.2 Convención de nombres

7.2.1 Clase interna de gestión de la asociación

En las plantillas que utiliza la herramienta simplemente va a haber que cambiar los valores claves por los introducidos en el diseño de las asociaciones, según se ha especificado en el apartado anterior. Ahora bien, ya se ha visto cómo la implementación de asociaciones *UML* en *Java* se va a traducir en una serie de atributos, métodos y clases internas en las clases participantes. Hemos de

tener en cuenta que una clase puede tener más de una asociación con otras clases; pudiendo incluso llegar a tener más de una con otra misma clase. Por tanto, hemos de buscar un nombrado unívoco para cada elemento de control de una asociación y como vamos a ver ahora, estos elementos de control tendrán como nombre combinaciones de los valores introducidos en el diseño de la asociación.

7.2.1.1 Nombre de la clase interna

Las dos clases que participan en una asociación van a tener una clase interna donde se gestionarán los enlaces. En adelante, para facilitar las cosas, a esta clase la notaremos (dentro de este documento) de una forma genérica como '*Association*', aunque en realidad, en el código generado tendrá el siguiente nombre que resulta más largo:

```
%Source%Target_MultiplicityNavigability_%AssociationKind

Por ejemplo:
ProfesorAsignatura_MultipleBidirectional_Association
AsignaturaProfesor_SingleBidirectional_Association
```

'*%AssociationKind*' se sustituirá por el tipo de asociación. En el ejemplo que estamos viendo se trata de una asociación normal por lo que, en este caso, se sustituye por '*Association*'.

'*Multiplicity*' puede ser '*Single*' o '*Multiple*' según sea la multiplicidad del otro extremo y '*Navigability*' será '*Unidirectional*' o '*Bidirectional*'. Estos valores no son propiamente sustituidos por la herramienta, sino que están especificados en las plantillas y, de hecho, son los que diferencian el tipo de plantilla. Nos encontramos, por tanto, con cuatro modalidades de plantilla que son las cuatro combinaciones que se pueden obtener de '*Multiplicity*' y '*Navigability*'.

El nombre que se le ha dado a la clase interna de gestión de la asociación es suficientemente representativo del tipo de asociación y da una idea bastante clara de su especificación.

7.2.1.2 El atributo de gestión de la asociación

El control de la asociación se hará sobre una única instancia de la clase interna '*Association*' por lo que habrá que definir un atributo de dicho tipo con el nombre *%targetRole*, cuyos posibles valores ya se han explicado más arriba. El nombre de ese atributo tiene que ser conocido por el otro lado de la asociación en el caso de asociaciones bidireccionales, es decir, por el código de control de la asociación que se encuentra en la otra clase.

Como ya se ha dicho, ni el nombre de rol ni el nombre de asociación son obligatorios en *UML*, pero, para evitar un conflicto de nombres en el código generado, la herramienta rechazará un diseño en el que haya dos asociaciones anónimas que no especifiquen nombres de rol entre las mismas dos clases.

7.2.1.3 Nombre de la clase interna en asociaciones reflexivas

En las asociaciones reflexivas también haremos obligatorio el nombre de rol porque, si no, es fácil que se produzcan conflictos de nombre. Además, para hacerlo más explícito, incluiremos la palabra '*reflexive*' en el nombre de la clase interna, es decir, '*%AssociationKind*' se sustituye por '*ReflexiveAssociation*'. De este modo el nombre de la clase queda de la siguiente manera:

```
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveAssociation

Por ejemplo:
PadreHijo_MultipleBidirectional_ReflexiveAssociation
HijoPadre_MultipleBidirectional_ReflexiveAssociation
```

y el del atributo queda igual que antes.

7.2.1.4 Asociaciones *todo / parte*

Las agregaciones y composiciones son un caso especial de las asociaciones que merecen ser estudiadas aparte. De momento, sólo adelantamos que es conveniente, a efectos de ingeniería inversa, que se pueda diferenciar cuándo estamos tratando con asociaciones normales y cuándo con asociaciones *todo / parte*. Por esto, las agregaciones y composiciones se van a diferenciar de la siguiente manera:

```
%Source%Target_MultiplicityNavegability_AggregationWhole
%Source%Target_MultiplicityNavegability_AggregationPart
%Source%Target_MultiplicityNavegability_CompositionWhole
%Source%Target_MultiplicityNavegability_CompositionPart

Por ejemplo:
DepartamentoMiembro_MultipleUnidirectional_AggregationPart
PalabraParrafo_SingleBidirectional_CompositionWhole
ParrafoPalabra_MultipleBidirectional_CompositionPart
```

Es decir, el campo '*%AssociationKind*' se sustituye por un nombre más representativo que nos dice el tipo de asociación y si nos estamos refiriendo al todo o a la parte.

7.2.1.5 Asociaciones reflexivas *todo / parte*

Por último, puede darse una combinación de los dos últimos casos, es decir, asociaciones reflexivas que sean a su vez del tipo *todo / parte*. Las clase internas que definen estas asociaciones siguen la siguiente convención de nombre:

```
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveAggregationWhole
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveAggregationPart
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveCompositionWhole
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveCompositionPart
```

7.2.2 Atributo de almacén de enlaces

Dentro de la clase interna '*Association*' existirá un atributo privado donde se almacenen los enlaces. En caso de que la clase '*Source*' se trate de una clase cuyo extremo en la asociación tiene multiplicidad sencilla, el atributo será de tipo '*Target*'. Por el contrario, si el extremo es múltiple, entonces, los enlaces se almacenarán en un *HashSet* cuyos elementos han de ser todos de tipo '*Target*'.

Este atributo tendrá por nombre genérico '*link*' para multiplicidades sencillas y '*links_hs*' (*HashSet* de enlaces) para multiplicidades complejas.

```
private %Target link
private HashSet links_hs

Por ejemplo:
private HashSet links_hs // en 'Profesor' para almacenar las asignaturas
private Profesor link // en 'Asignatura' para almacenar los profesores
```

Hay que comentar también que los métodos recibirán de entrada enlaces de tipo '*Target*' y que harán uso de variables de ese tipo. Dichas variables siempre se diferenciarán con la palabra clave '*link*' cuando se refieran a un único enlace y '*links*' cuando se refieran a un conjunto de ellos. Por ejemplo: '*new_link*', '*old_link*', '*new_links*', etc.

7.2.3 Nombrado de los métodos

En cuanto a los métodos habrá que identificar cuál es su función o cometido. Para ello, utilizaremos el nombre que más fácil nos hace identificar dicha funcionalidad y buscaremos que sea una notación internacional por lo que se utilizará el vocabulario en inglés que comúnmente se encuentra en los nombres de los métodos.

Así, por ejemplo, un método que añade un enlace, se llamará '*add*', y actuará sobre el parámetro de entrada con nombre '*x*' que designará una instancia de la clase al otro lado de la asociación. Otros nombres de métodos típicos que nos vamos a encontrar van a ser '*remove*', '*get*', '*test*', etc. y ya han sido presentados en el apartado *5*: "*Un interfaz uniforme*".

7.2.4 Ejemplo aclarativo

Lo que se ha dicho hasta ahora puede resultar un poco confuso de entender en abstracto y por tanto, vamos a explicarlo con un ejemplo concreto. Supongamos que tenemos una asociación como la de la Figura 7.1. En ella se ha especificado la asociación entre la clase '*Profesor*' y la clase '*Asignatura*'.

Para implementar la asociación con la clase '*Asignatura*' dentro de la clase '*Profesor*', necesitaríamos definir una clase interna '*ProfesorAsignatura_MultipleBidirectional_Association*' en '*Profesor*' y otra '*AsignaturaProfesor_SingleBidirectional_Association*' en '*Asignatura*'. Dentro de '*ProfesorAsignatura_MultipleBidirectional_Association*' sería necesario un atributo de tipo colección (*HashSet*) donde se almacenarán todas las asignaturas o materias, que se denominará '*links_hs*', es decir, *HashSet* de enlaces. Supongamos que de momento sólo se necesitan métodos para añadir y borrar asignaturas. Los nombres escogidos para estos métodos serían '*add*' y '*remove*'. En la Figura 7.2 se encuentra la clase '*ProfesorAsignatura_MultipleBidirectional_Association*' con los atributos y métodos necesarios para el control de la asociación en '*Profesor*'.

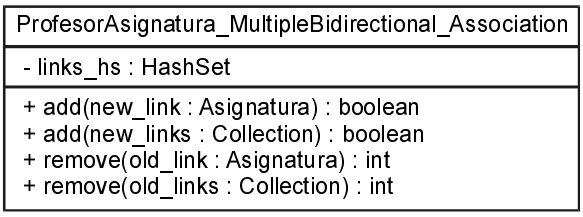

Figura 7.2: Clase interna '*ProfesorAsignatura_Multiple-Bidirectional_Association*' de '*Profesor*'.

En la Figura 7.3 puede observarse la clase interna '*ProfesorAsignatura_Single-Bidirectional_Association*' de '*Asignatura*'.

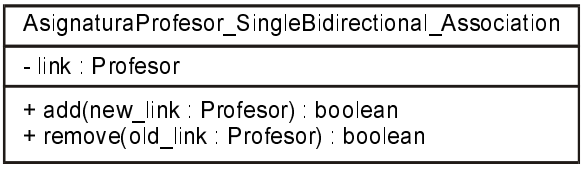

Figura 7.3: Clase interna '*AsignaturaProfesor_Single-Bidirectional_Association*' de '*Asignatura*'.

Como única diferencia a destacar, obsérvese que esta vez la multiplicidad para '*Asignatura*' es de uno por lo que el atributo que almacena el enlace es directamente de tipo '*Profesor*', al tener que almacenarse una única referencia y no varias. Asimismo, no existe la declaración de los métodos '*add*' y '*remove*' que actúan sobre una colección de profesores.

8 EXCEPCIÓN DE ASOCIACIÓN NO ESTABLECIDA

No siempre vamos a poder garantizar que las asociaciones se encuentren en un estado válido, es decir, que cumplan con las especificaciones de diseño y en concreto, con las restricciones de multiplicidad y esto se debe a que será el usuario de las clases que controlan las asociaciones el responsable en última instancia de que así sea. Por esto, nos hemos visto obligados a implementar una excepción que será lanzada cuando el usuario intente acceder a los enlaces de una clase y éstos no estén establecidos correctamente por no cumplir con los requisitos de multiplicidad mínima.

Previamente hemos mencionado que vamos a permitir que se eliminen los enlaces sin exigir que se cumpla severamente con las restricciones de la asociación y que sólo exigiremos corrección en los enlaces a la hora de utilizarlos o acceder a ellos. Cuando se intenten utilizar los enlaces y éstos no cumplan con las restricciones impuestas por la asociación por estar indebidamente establecidos, se lanzará la excepción '*UnsetAssociationException*' cuyo código mínimo presentamos a continuación:

```
/**
 * Título: Excepción de asociación no establecida
 * Descripción: Excepción que se ha de lanzar cuando se intenta realizar una
 * operación sobre los enlaces de una asociación en la que no se está cumpliendo
 * con las restricciones impuestas
 * @author Carlos Ruiz del Castillo
 *         <a href="mailto:crcu@inicia.es">&lt;crcu@inicia.es&gt;</a>
 */

public class UnsetAssociationException extends Exception {

  public UnsetAssociationException() {
  }

  public UnsetAssociationException(String msg) {
    super(msg);
  }

}
```

Como esta excepción se lanza cuando no se cumplen los requisitos de multiplicidad mínima, podría pensarse que un nombre más correcto es '*InvalidMultiplicityException*'. En nuestra decisión final, se ha optado por el nombre de asociación no establecida ('*UnsetAssociation-Exception*') porque pensamos que si bien, ahora mismo, el control sólo se hace sobre la multiplicidad, variaciones de código en el futuro o asociaciones más complejas, pueden requerir un control más férreo sobre otro tipo de restricciones de la asociación y se debe usar esta misma excepción.

9 IMPLEMENTACIÓN DE ASOCIACIONES UNIDIRECCIONALES

Las asociaciones unidireccionales sólo tienen significado en un sentido, es decir, sólo una de las instancias participantes tiene acceso a sus enlaces (clase '*Source*' en adelante), mientras que la otra instancia ignora el hecho de que la instancia origen está enlazada con ella (clase '*Target'*). De este modo, el código que haya que añadir, irá siempre en la clase origen o 'Source' y hará referencia a la clase destino o '*Target*'. El introducir código en una sola clase facilita la implementación y además, como veremos, las funciones que se van a necesitar siguen la misma idea que en las bidireccionales pero serán más sencillas. Entender bien cómo funciona la implementación de asociaciones unidireccionales de *UML* facilitará la comprensión de la implementación de las asociaciones bidireccionales que añadirán un poco más de complejidad.

Recordemos que teníamos cinco posibles asociaciones de este tipo pero las dividíamos en dos grupos: las sencillas y las múltiples.

9.1 Asociaciones unidireccionales sencillas

En este grupo se encuentran las asociaciones unidireccionales de multiplicidad en el destino de 0..1 o 1..1, es decir, el enlace entre una instancia de la clase '*Source*' y instancia de la clase '*Target'* sólo puede ser uno, y puede ser opcional u obligatorio. Por tanto, en la clase interna que gestiona la asociación, y que se añade a la clase origen, vamos a necesitar un atributo que almacena el único enlace posible con la clase destino y una constante booleana que indica si el enlace es obligatorio (1..1) u opcional (0..1). Gracias a esta constante booleana se puede controlar cuando los enlaces están cumpliendo las restricciones de multiplicidad. Si es opcional, no hay ningún control que hacer puesto que se pueden añadir o eliminar los enlaces cuándo se desee o necesite. Además, cuando se quiera acceder a los enlaces, no existe la restricción de que forzosamente tenga que haber uno. Por el contrario, cuando la asociación es obligatoria, se permite añadir y eliminar enlaces, pero cuando se quiera acceder a los enlaces, se requiere que haya necesariamente un enlace, lanzándose la excepción '*UnsetAssociationException*' si no lo hubiese.

Veamos con detenimiento la plantilla del código de una clase '*Source*' que se enlaza con una clase '*Target*'. En la Figura 9.1 se hallan representadas los dos tipos de asociaciones que se resuelven en este apartado.

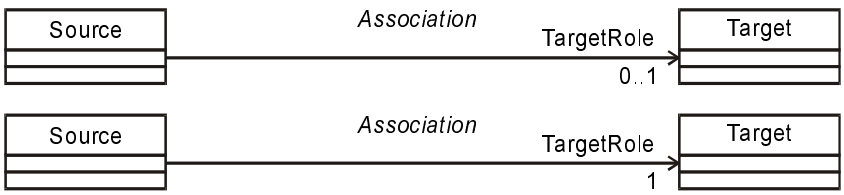

Figura 9.1: Asociaciones unidireccionales sencillas.

A continuación, mostramos el código:

```
package %package;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_SingleUnidirectional_%AssociationKind
        %targetRole = new %Source%Target_SingleUnidirectional_%AssociationKind();

  %visibility class %Source%Target_SingleUnidirectional_%AssociationKind {

    /** Para representar multiplicidad muchos (* en UML)
     *  No se utiliza aquí pero se pone por uniformidad de interfaz con la
     *  implementación de otro tipo de asociaciones
     */
    %visibility final static int MANY = -1;

    /** Enlace de %Source a %Target:
     *       obligatorio (multiplicidad 1..1): mandatory = true
     *       opcional (multiplicidad 0..1): mandatory = false
     */
    private final static boolean mandatory = %mandatory;

    /** Almacena el enlace con %Target */
    private %Target link = null;

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return false;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return mandatory;
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return false;
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      if (isMandatory()) {
        return 1;
      }
      return 0;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
     */
```

```
%visibility long getMAX() {
  return 1;
}

/** Dice si %Source cumple con las restricciones de la asociación
 *  @return true si sí las cumple, false en caso contrario
 */
%visibility boolean isValid() {
  return ((! mandatory) || (link != null));
}

/** Devuelve el número de enlaces activos que tiene el objeto
 *  @return Número de enlaces
 */
%visibility long numberOfLinks() {
  if (link == null) {
    return 0;
  }
  return 1;
}

/** Comprueba si %Source está asociado con un objeto %Target concreto
 *  @param query_link Supuesto enlace %Target
 *  @return true si existe dicho enlace, false en caso contrario
 */
%visibility boolean test(%Target query_link) {
  return link == query_link;
}

/** Devuelve el enlace %Target contenido en link
 *  Lanza la excepción UnsetAssociationException si la asociación no
 *  cumple con los requisitos de la asociación
 *  @return link (null si no existe ningún enlace)
 */
%visibility %Target get() throws UnsetAssociationException {
  if (! isValid()) {
    throw new UnsetAssociationException ("El objeto de clase %Source"
    + " no cumple con los requisitos de la asociación");
  }
  return link;
}

/** Añade un enlace con una instancia de %Target
 *  @param new_link Objeto con el que se intenta enlazar
 *  @return true si se añade el enlace o ya existía, false si no se puede
 *  debido a que ya tiene otro enlace distinto o new_link es null
 */
%visibility boolean add(%Target new_link) {
  synchronized (this) {
    // Si no se puede establecer el enlace, se termina
    if ((new_link == null) || ((link != null) && (link != new_link))) {
      return false;
    }
    // Se establece el enlace en %Source
    link = new_link;
    return true;
  }
}

/** Elimina el enlace establecido en link
 *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
 *  restricciones de multiplicidad, 1 si se deja de cumplir
```

```
     */
    %visibility int remove() {
      synchronized (this) {
        // Elimina el enlace
        link = null;
        // Devuelve el valor correcto
         if (mandatory) {
           return 1;
         }
         return 0;
        }
     }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si deja de cumplir con ellas y -1
     *  si no existe enlace con old_link o old_link es null
     */
    %visibility int remove(%Target old_link) {
      if ((old_link != null) && test(old_link)) {
        return remove();
      }
      return -1;
    }

  }

}
```

En la Figura 9.2 puede verse el diagrama de clases en *UML* de la implementación en *Java* que se acaba de mostrar. Por comodidad de lectura, a la clase '*%Source%Target_Single-Unidirectional_%AssociationKind*', en adelante, la denominaremos '*Association*'.

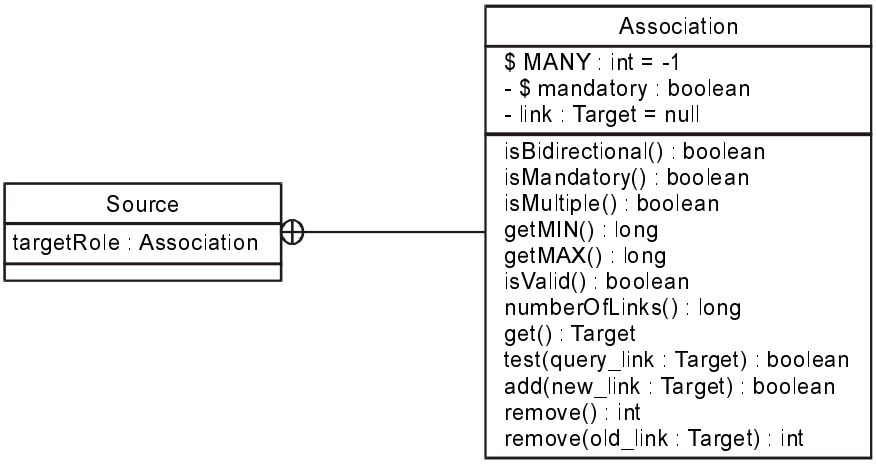

Figura 9.2: Asociación unidireccional sencilla.

9.1.1 La clase '*Source*'

Para empezar a interpretar el código, vamos a centrarnos primero en la clase '*Source*'. Dicha clase tiene declarado el atributo '*targetRole*' que almacenará una instancia de la clase '*Association*'. Es a través de este atributo como se hará toda la gestión de la asociación. El atributo está declarado como *final* para que no se pueda cambiar su referencia bien intencionadamente o bien sin querer. Así, toda la gestión de los enlaces de cada instancia de '*Source*' se hará siempre a través de dicho atributo y no habrá otra forma de crear otra instancia de '*Association*' más que dentro de la propia clase '*Source*', lo cual no está en absoluto recomendado.

No se ha implementado ningún mecanismo para evitar la creación deliberada de otra instancia de '*Association*' dentro de '*Source*'. Esto se debe a que resultaría un mecanismo complejo y es preferible que el usuario esté advertido de que bajo ningún concepto puede crear instancias '*Association*'. El patrón de diseño *Singleton* es difícil de aplicar en esta situación porque no queremos crear una única instancia de una clase sino una única instancia de '*Association*' por cada instancia '*Source*'.

9.1.2 La clase '*Association*'

En la clase '*Association*', primero nos encontramos con la constante '*MANY*' que sirve para especificar la multiplicidad *UML* de muchos (representado por el símbolo '*'). Esta constante no se utiliza en la implementación de este tipo de asociaciones puesto que la multiplicidad máxima es uno pero se ha puesto por uniformidad de interfaz con el resto de asociaciones.

El siguiente atributo definido es la constante '*mandatory*' que sirve para especificar si el enlace es opcional (multiplicidad 0..1) con el valor *false* u obligatorio (multiplicidad 1..1) en cuyo caso, se debe especificar con el valor *true*. Para especificar la obligatoriedad o no de los enlaces se tiene que sustituir el campo '*%mandatory*' por el valor correspondiente.

El último atributo es '*link*' que es de tipo '*Target*'. En este atributo se almacena el enlace de cada instancia de '*Source*' y cuando la asociación no contenga ningún enlace, este campo valdrá *null*.

Obsérvese que los atributos '*mandatory*' y '*link*' son privados puesto que son de uso interno a la clase '*Association*'.

9.1.2.1 isBidirectional

El primer método definido es '*isBidirectional*' que simplemente devuelve falso puesto que se trata de una asociación unidireccional.

9.1.2.2 isMandatory

Devuelve el valor de la constante '*mandatory*' para indicar si es obligatorio para todas las instancias de '*Source*' tener un enlace o no.

9.1.2.3 isMultiple

Devuelve falso puesto que se trata de una asociación sencilla.

9.1.2.4 getMIN

Si la asociación es obligatoria la multiplicidad mínima será una y si no, cero.

9.1.2.5 getMAX

Este método, en este caso, directamente devuelve uno como multiplicidad máxima.

9.1.2.6 isValid

'*isValid*' nos dice si el enlace está cumpliendo con las restricciones de la asociación. Devuelve *true* siempre que la asociación sea opcional o que sea obligatoria pero tenga un enlace asignado.

9.1.2.7 numberOfLinks

Este método devuelve el número de enlaces que tiene la instancia '*Source*' que por tratarse, en este apartado, de una asociación sencilla, sólo puede ser cero o uno. Si *'link*' es nulo, la llamada a esta función responderá con "cero" y si no, con "uno".

9.1.2.8 test

El método '*test*' sirve para comprobar si existe un enlace de '*Source*' a '*Target*' concreto por lo que comprueba si el parámetro de entrada es el enlace precisamente contenido en el atributo *'link*'.

Hemos dicho que no íbamos a permitir el acceso a los enlaces cuando no se cumpliesen con los requisitos de la asociación lazando la excepción '*UnsetAssociationException*'. Podría pensarse que en cierto modo, '*test*' es un acceso a los enlaces pero no lo consideramos así. Además, este método puede servir para que cuando no se cumplan con los requisitos de la asociación en los enlaces, se puedan hacer ciertas comprobaciones y poder conocer qué enlaces se pueden añadir y cuáles no.

9.1.2.9 get

'*get*' sirve para acceder y conocer los enlaces de las asociaciones por lo que devolverá la instancia contenida en '*link*' de tipo '*Target*' cuando exista enlace, o el valor *null* cuando no haya ninguno establecido.

Puede ocurrir que en una asociación obligatoria (en la que se requiera forzosamente de la existencia de un enlace) cuando se intente acceder al enlace, éste no exista. Esto se considera un error grave que se debería haber previsto en el código que implemente el usuario de la clase '*Association*', por lo que se lanzará la excepción de asociación no establecida: '*UnsetAssociation-Exception*'. En este caso, sí que se considera un acceso a los enlaces por lo que se usa dicha excepción.

9.1.2.10 add

Este método sirve para establecer un enlace, pero lo hace si y sólo si no contiene ningún otro enlace previamente, es decir, si '*link*' es nulo. Si se quisiese reemplazar un enlace, habría que llamar primero a '*remove*' para eliminar el antiguo y luego a '*add*' para añadir el nuevo. Quizás, fuese más obvio permitir que en una única llamada a '*add*' se pudiese sustituir un enlace por otro. No obstante, esto se ha hecho así por coherencia con los métodos tipo '*add*' del resto de casos que nos queda por estudiar. En dichos casos, no siempre se debe permitir así como así la sustitución (pero eso lo veremos más adelante). El método devuelve *true* cuando se puede establecer el enlace y *false* en caso contrario, cuando se contiene otro enlace en '*link*' o se intente establecer un enlace nulo, lo cual no tiene ningún sentido.

Si se llama a '*add*' pasando como parámetro de entrada una instancia '*Target*' que coincide con el enlace actual de la asociación, el método devuelve *true* y hace como si sí se hubiese

establecido el enlace. Cuando se intenta asignar un enlace que ya existe, se puede tomar dos posturas: hacer como que se establece un nuevo enlace aunque el estado final del enlace sea el mismo que había antes de la llamada al método (eso es lo que se ha hecho) o devolver *false* para indicar que realmente no se ha hecho nada. En las asociaciones múltiples veremos que cabe la posibilidad de asignar un conjunto de nuevos enlaces en los que se ponga algunos enlaces que previamente estén asignados. En esta situación, parece lógico que lo normal será ignorarlos y sólo añadir los nuevos enlaces. De este modo, y siguiendo con el principio de aplicar siempre los mismos criterios, establecemos que cuando se intente asignar un enlace como nuevo, aunque realmente ya esté asignado, se interpreta como que sí se hace la asignación y el enlace queda tal y como se ha pedido en la invocación del método. En definitiva, cuando el usuario del método quiere establecer un enlace, le interesa saber si el enlace se ha quedado como ha solicitado (valor de retorno *true*) o no (valor *false*).

Obsérvese que '*add*' está gobernado por un bloque que empieza por "*synchronized (this)*". Esta región crítica evita que varios hilos de ejecución intenten cambiar el estado del enlace a la vez y se produzcan irregularidades o estados no deseados. Todos los métodos 'add' y '*remove*' que vamos a ver de ahora en adelante, es decir, los que cambian el estado de las asociaciones, van a llevar el control de acceso a dicha región crítica para así evitar situaciones imprevistas.

9.1.2.11 remove

Existen dos definiciones de este método. La primera definición elimina un enlace sin más; poniendo el atributo '*link*' con el valor nulo. Devuelve "cero" si al eliminar la asociación no se rompe con las restricciones de la asociación, es decir, la asociación es opcional, y "1" si sí se rompe por ser obligatoria. Con esto, se puede avisar al usuario del método que tenga cuidado con el estado del enlace debido a que puede no ser válido y por tanto, puede encontrarse con problemas si no asigna un nuevo enlace.

La segunda definición funciona exactamente igual, a excepción de que recibe como parámetro una instancia de tipo '*Target*' con la que se quiere romper el enlace. Primero se comprueba si realmente se está enlazado con dicho parámetro, en cuyo caso se rompe el enlace y si no, se devuelve el valor "–1". En este caso, esta variante del método '*remove*' es redundante puesto que tenemos la primera para eliminar los enlaces. No obstante, se ha dejado por coherencia con las otras situaciones en las que, como se verá, sí es necesario. La solución que se está presentando intenta que modificaciones de diseño futuras afecten lo mínimo posible al código ya implementado y por tanto, se tenga que hacer el menor número posible de cambios. Se ha pensado que podría haber código que gestionase asociaciones de cualquier otro tipo y por cambios de diseño, dicha asociación pase a ser del tipo que se presenta en este apartado. En tal situación, podría haber código que hiciese llamada a la segunda definición de '*remove*' y a pesar de haber cambiado de tipo de asociación, seguiría funcionando al estar también definida en el nuevo código de la clase '*Association*'.

9.1.3 Visibilidad de la asociación

En el código, obsérvese cómo todos los atributos (a excepción de *'link*' y '*mandatory*') y los métodos de la clase '*Association*' van precedidos por el campo *%visibility*. En las gráficas no se ha especificado la visibilidad para esos atributos y métodos porque es algo que viene impuesto por el diseño de la asociación.

El campo '*%visibility*' del código mostrado ha de ser sustituido por el valor adecuado según se especifique la visibilidad de la asociación. Si la visibilidad se hace pública, cualquier instancia externa a la clase '*Source*' podrá acceder a la clase '*Association*' y por tanto, al atributo que

gestiona los enlaces. Si por el contrario es privada, sólo desde dentro de '*Source*' se podrá controlar la asociación. También se podrá especificar visibilidad de paquete o protegida obteniéndose los resultados deseados para estos tipos de visibilidades.

Para establecer la visibilidad de la asociación acorde a la especificación, lo fundamental es poder delimitar el acceso al atributo de gestión de enlaces ('*targetRole*') a las clases que vayan a tener permiso. No obstante, se ha pensado que es mucho más coherente delimitar la visibilidad también en la clase, en los atributos y en los métodos por especificarlo todo de una forma más coherente y precisa.

Los atributos '*link*' y '*mandatory*' son privados porque no deben ser manipulados y consultados desde fuera de '*Association*'. No obstante, debido a que están introducidos en la clase '*Association*', y ésta es interna a '*Source*', desde '*Source*', sí se puede acceder a ellos. Esto no lo podemos evitar con el lenguaje *Java* pero recomendamos no utilizarlos nunca directamente para garantizar una gestión perfecta de las asociaciones.

La visibilidad de las asociaciones es un tema complejo sobre todo porque la definición que da *UML* resulta un tanto ambiguo y paradójico. En las asociaciones unidireccionales se resuelve fácilmente pero no ocurre lo mismo en las bidireccionales. El tema de la visibilidad, por tanto, va a merecer un capítulo aparte y lo veremos en el 13: "*Visibilidad en las asociaciones*". Por ahora, nos contentaremos con señalar que simplemente hay que sustituir '*%visibility*' en las plantillas por la visibilidad correspondiente: '*public*', '*private*', '*protected*' o nada si es visibilidad de paquete (*friendly* en *Java*).

9.2 Asociaciones unidireccionales múltiples

Los otros tres casos de asociaciones unidireccionales que nos quedan por ver se encuentran en este grupo. En ellos, la multiplicidad de 'Source' a '*Target*' puede no imponer ninguna restricción:

- que se tengan tantos enlaces como se quiera: 0..*,

o poner alguna:

- que al menos se tenga uno (1..*) o
- tener entre *m* y *n* enlaces.

Esta vez no nos valdrá con tener una variable booleana que indique si la asociación es opcional u obligatoria, de modo que a cambio se definirán dos constantes numéricas que indicarán la multiplicidad mínima y la máxima, existiendo un valor especial para indicar "muchos" (*). Además, como se tratará con más de un enlace, se necesitará algún tipo de dato que permita almacenar varios enlaces de una forma flexible, es decir, alguna de las colecciones que existe en *Java*: *ArrayList*, *HashSet*, *Vector*, etc.

Por lo demás, el funcionamiento de la clase '*Association*' va a ser más o menos similar al del apartado anterior. Se implementarán los mismos métodos pero esta vez, en lugar de actuar sobre un atributo que representa un único enlace, actuarán sobre un atributo que almacena varios. Empecemos por ver en la Figura 9.3 los tres tipos de asociaciones que se tratan en este apartado.

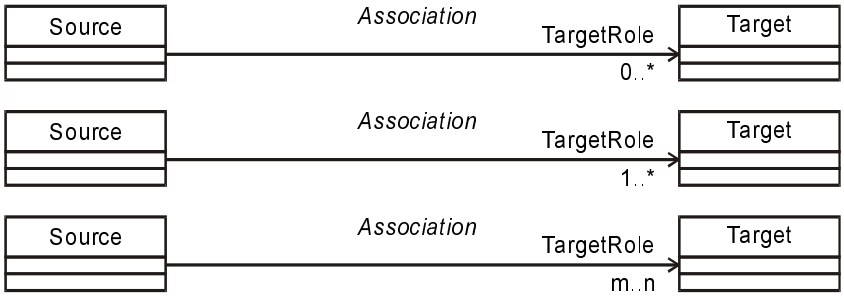

Figura 9.3: Asociaciones unidireccionales múltiples.

A continuación presentamos el código implementado y en la Figura 9.4 se halla el diagrama de clases de '*Source*' y de '*Association*'. Resaltamos con un color de fondo aquellas partes del código que presenten una diferencia sustancial respecto a lo mostrado en la plantilla de código del apartado anterior. Además, en la explicación del código, sólo se van a comentar las diferencias para no repetirnos.

```
package %package;

import java.util.Collection
import java.util.HashSet;
import java.util.Iterator;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_MultipleUnidirectional_%AssociationKind
      %targetRole = new %Source%Target_MultipleUnidirectional_%AssociationKind();

  %visibility class %Source%Target_MultipleUnidirectional_%AssociationKind {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = %min_multiplicity;

    /** Para representar multiplicidad muchos (* en UML) */
    %visibility final static int MANY = -1;
    /** Multiplicidad máxima del enlace */
    private final static long MAX = %max_multiplicity;

    /** HashSet donde se almacenan los enlaces con instancias %Target */
    private HashSet links_hs = new HashSet();

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return false;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return (MIN > 0);
```

```
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return ((MAX > 1) || (MAX == MANY));
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      return MIN;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
     */
    %visibility long getMAX() {
      return MAX;
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((links_hs.size() >= MIN) &&
              ((links_hs.size() <= MAX) || (MAX == MANY)));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      return links_hs.size();
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     */
    %visibility boolean test(%Target query_link) {
      if (query_link == null) {
        return links_hs.isEmpty();
      }
      return links_hs.contains(query_link);
    }

    /** Comprueba si %Source está asociado con una colección de objetos %Target
     *  @param query_links Colección de supuestos enlaces
     *  @return true si todos los elementos de links_hs están enlazados
     *  con %Source o si links_hs está vacío y query_links también o es null
     */
    %visibility boolean test(Collection query_links) {
      if (query_links == null) {
        return links_hs.isEmpty();
      }
      return links_hs.containsAll(query_links);
    }
```

```
    /** Devuelve una colección con la copia de los enlaces con %Target
     *  contenidos en links_hs. Lanza la excepción UnsetAssociationException si
     *  la asociación no cumple con los requisitos de la asociación
     *  @return copia de links_hs
     */
    %visibility Collection get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
      return (Collection) links_hs.clone();
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se realiza el enlace o ya existía, false si no se puede
     *  debido a que se excede la multiplicidad máxima o new_link es null
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        if (new_link != null) {
          // Si se intenta enlazar con un objeto que ya está enlazado, no se
          // hace nada pero se devuelve como si se hubiese establecido el enlace
          if (test(new_link)) {
            return true;
          }
          // Si añadir new_link no excede la multiplicidad máxima
          if ((MAX == MANY) || (links_hs.size() < MAX)) {
            // se establece el enlace
            links_hs.add(new_link);
            return true;
          }
        }
        return false;
      }
    }

    /** Añade un enlace con cada uno de los elementos de new_links
     *  @param new_links Colección de elementos %Target con los que se
     *  intenta enlazar
     *  @return true si se establecen TODOS los enlaces, false si se excede de
     *  la multiplicidad máxima o alguno de los nuevos enlaces no se puede
     *  establecer o new_links es null
     */
    %visibility boolean add(Collection new_links) {
      if (new_links == null) {
        return false;
      }
      synchronized (this) {
        if (new_links != null) {
          // Sobre una copia de new_links se eliminan aquellos objetos
          // %Target que tengan establecido el enlace previamente
          HashSet copied_new_links = new HashSet(new_links);
          copied_new_links.removeAll(links_hs);
          // Si añadir los nuevos elementos no excede de la multiplicidad máxima
          if ((MAX == MANY) ||
              ((copied_new_links.size() + links_hs.size()) <= MAX)) {
            // se añaden de uno en uno para poder asegurarse que todos los
            // elementos son de tipo %Target
            Iterator i = copied_new_links.iterator();
            while (i.hasNext()) {
              try {
```

```
                links_hs.add((%Target) i.next());
              } catch (ClassCastException e) {
                // Si algún elemento no es tipo %Target, se eliminan los
                // añadidos hasta ahora y se devuelve false
                links_hs.removeAll(new_links);
                return false;
              }
            }
            return true;
          }
        }
        return false;
      }
    }

    /** Elimina todos los enlaces establecidos en links_hs
     *  @return 0 si se eliminan todos los enlaces y se siguen cumpliendo con
     *  las restricciones de la asociación y 1 si se deja de cumplir con ellas
     */
    %visibility int remove() {
      synchronized (this) {
        // Elimina los enlaces
        links_hs.clear();
        // Devuelve el valor correcto
        if (MIN > 0) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si deja de cumplir con ellas y -1
     *  si no existe enlace con old_link o old_link es null
     */
    %visibility int remove(%Target old_link) {
      synchronized (this) {
        if ((old_link != null) && test(old_link)) {
          // Elimina el enlace de %Source a %Target
          links_hs.remove(old_link);
          // Devuelve el valor correcto
          if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }

    /** Borra los enlaces establecidos con un conjunto de objetos %Target
     *  @param old_links Colección de objetos %Target cuyo enlace con %Source
     *  se quiere borrar
     *  @return 0 si al eliminar los enlaces de %Source a old_links se sigue
     *  cumpliendo con las restricciones de la asociación y 1 si se deja de
     *  cumplir con las restricciones. -1 si algún elemento de old_links no es
     *  de tipo %Target u old_links es null y por tanto, no se hace nada
     */
    %visibility int remove(Collection old_links) {
      synchronized (this) {
```

```
        if (old_links != null) {
          HashSet copied_links_hs = (HashSet) links_hs.clone();
          // Los enlaces se eliminan de uno en uno
          Iterator i = old_links.iterator();
          while (i.hasNext()) {
            try {
              links_hs.remove((%Target) i.next());
            } catch (ClassCastException e) {
              // Si algún elemento no es tipo %Target, se deja todo como estaba
              links_hs = copied_links_hs;
              return -1;
            }
          }
          // Devuelve el valor correcto
          if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }

  }

}
```

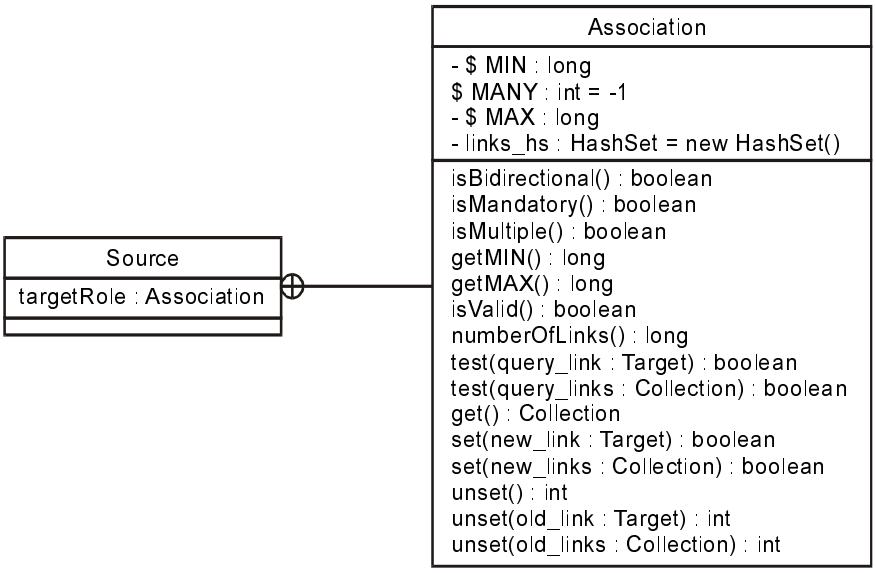

Figura 9.4: Asociación unidireccional múltiple.

9.2.1 Clases importadas

Obsérvese en el código que se importan tres clases de la librería *java.util*: *Collection*, *HashSet* e *Iterator*. Ocurre, que como en muchos de los métodos se puede actuar sobre más de un enlace al mismo tiempo, se utiliza el interfaz *Collection* en los parámetros de entrada y salida de estos. Las

clases que implementan dicha interfaz son conocidas como colecciones y permiten almacenar varios objetos. La utilización de las colecciones de *Java* va a dotar al usuario de grandes ventajas debido a la flexibilidad que estas aportan al utilizar todas el mismo interfaz. El usuario no se va a ver supeditado a un tipo concreto de colección y por tanto, podrá emplear la estructura de datos que más le convenga.

No obstante, internamente a la clase '*Association*', sí que habrá que almacenar los enlaces en una colección concreta. Hemos escogido la clase *HashSet* porque no permite la existencia de elementos repetidos o que sean nulos. Por tanto, *HashSet* es idónea para tratar con conjuntos de enlaces. También se utiliza el interfaz *Iterator* que nos servirá para poder recorrer los elementos de las colecciones.

9.2.2 La clase '*Association*'

Para la gestión de la asociación, primero hay que definir dos constantes en las que se especifique la multiplicidad mínima y máxima de la asociación en el lado del destino. De ahí el nombre de las constantes '*MIN*' y '*MAX*'. Adicionalmente se define la constante '*MANY*' cuyo valor se debe asignar a '*MAX*' cuando la multiplicidad máxima sea de "muchos" (*).

En el atributo privado '*links_hs*' se van a almacenar todos los enlaces. Es una colección de tipo *HashSet* que se inicializa desde el principio para así poder trabajar con ella desde que se crea la instancia '*targetRole*'. Con esta inicialización se consigue que, si un usuario intenta acceder a los enlaces cuando todavía no ha añadido ninguno, obtenga un conjunto vacío en lugar de un valor *null*., aparte de que los métodos de '*Association*' ya no tendrán que encargarse de inicializarla. Si se añaden varios enlaces y luego se borran todos, la colección de enlaces se queda vacía, y por tanto, el conjunto vacío servirá para indicar siempre la no existencia de enlaces. Si no se hiciese la inicialización, el usuario tendría que hacer primero la comprobación de no obtener un valor nulo no fuera a ser que intentase acceder a los elementos del conjunto sobre una variable *null* obteniendo un error de ejecución.

Los métodos declarados llevan los mismos nombres que en las asociaciones unidireccionales sencillas. Ahora bien, esta vez estamos trabajando con la posibilidad de usar más de un enlace al mismo tiempo, por lo que habrá más posibilidades de métodos y éstos, según los casos, permitirán trabajar con todos los enlaces, con uno concreto o con unos específicos. Las variantes de los métodos utilizan el mismo nombre y se distinguen mediante los parámetros de entrada gracias a la sobrecarga de métodos.

9.2.2.1 isMandatory

Devuelve *true* siempre y cuando la multiplicidad mínima sea mayor que uno.

9.2.2.2 isBidirectional

Como seguimos con las asociaciones unidireccionales, el método devuelve falso directamente.

9.2.2.3 isMultiple

Devuelve *true* si la multiplicidad máxima es decir, '*MAX*', es mayor que uno y *false* en caso contrario.

9.2.2.4 getMIN

Devuelve directamente el valor contenido en '*MIN*' para dar a conocer la multiplicidad mínima.

9.2.2.5 getMAX

Este método es igual que el anterior salvo que devuelve el valor de '*MAX*'. El valor que se obtiene conviene compararlo con '*MANY*' para comprobar si se trata de una asociación sin restricciones de multiplicidad máxima.

9.2.2.6 isValid

'*isValid*' indica si las instancias de '*Source*' cumplen con las restricciones de multiplicidad máxima y mínima. Para ello, se comprueba que el número de enlaces contenidos en '*links_hs*' se encuentra dentro del rango que hay entre '*MIN*' y '*MAX*'. No obstante, bastaría con que comprobase que hubiese mayor o igual número de enlaces que el valor de la constante '*MIN*' puesto que va a ser imposible establecer más enlaces que los especificados por el tope '*MAX*' debido a que la única forma de añadir enlaces es mediante el método '*add*' que no funciona cuando se intenta sobrepasar ese límite. En principio, '*isValid*' está pensado para hacer todas las comprobaciones de las restricciones (y no sólo las de multiplicidad) y por eso se ha codificado de una forma tan exhaustiva.

9.2.2.7 numberOfLinks

Para conocer el número de enlaces, basta simplemente con devolver el número de elementos contenidos en '*links_hs*, es decir, el tamaño de dicho *HashSet*. En realidad, este método está pensado por si en algún momento se desconoce el número de enlaces que se tiene y debido a que no se está cumpliendo con las restricciones de la asociación, éstos no se pueden conocer. Gracias a este método, se podrá determinar el número de enlaces que se tienen que añadir para que la asociación sea correcta.

9.2.2.8 test

En esta ocasión, existen dos variantes del método '*test*'. La primera sirve para comprobar la existencia de un único enlace y la segunda para comprobar la existencia de un conjunto de enlaces. En ambas definiciones, simplemente se comprueba si la instancia de '*Target*' que se introduce como parámetro de entrada (o el conjunto de ellas) están contenidas dentro de '*links_hs*'. Si en cualquiera de las dos definiciones de '*test*', el parámetro de entrada son *null*, se hace la comprobación de si el atributo-colección que almacena los enlaces está vacío.

9.2.2.9 get

'*get*' sirve para conocer los enlaces de un objeto '*Source*'. Como se trata de una asociación múltiple y lo normal será que devuelva más de un enlace, se devuelve una colección. Si se observa el código se verá que no devuelve el contenido de '*links_hs*' directamente sino una copia. Esto se ha hecho así para evitar la manipulación directa del conjunto de enlaces. *Java* trabaja siempre con valores por referencia por lo que si se devuelve directamente la referencia a '*links_hs*', se puede actuar impunemente sobre dicha referencia añadiendo y eliminado elementos. Devolviéndose una copia, se evita esta manipulación no deseada.

9.2.2.10 add

Para el método '*add*' existen dos variantes. Una de ellas añade un nuevo enlace siempre que no se exceda de la multiplicidad máxima. Recibe como parámetro una instancia destino tipo '*Target*' y, tras comprobarse que no es nulo y que su adición no va a romper con las restricciones de la asociación, se añade a la colección de enlaces contenida en el *HashSet* '*links_hs*' devolviendo el valor *true*. Si las comprobaciones son negativas y por tanto, no se puede añadir el nuevo enlace, se devuelve *false*.

La otra definición añade más de un enlace a la vez. Recibe como parámetro de entrada una colección de elementos tipo '*Target*' que se supone son los nuevos enlaces. Sobre una copia de dicho parámetro se eliminan aquellos elementos que se quieran añadir como enlaces pero que ya estén establecidos de antemano y, así, poder ignorarlos porque no aportan nada nuevo. Se utiliza una copia de la colección debido a que *Java* funciona con parámetros de entrada en los métodos por referencia y así no modificarlo por si posteriormente se necesitase íntegro. Una vez eliminados los enlaces repetidos, se mira si añadir los nuevos enlaces va a provocar el excederse de la multiplicidad máxima y en caso afirmativo, no se hace nada. Si resulta que sí se pueden añadir los nuevos enlaces, entonces mediante un iterador sobre la copia de la colección de entrada se van añadiendo de uno en uno para terminar saliendo del método con el valor *true* y así, indicar que todo ha ido bien. El motivo por el que se añaden de uno en uno y no de golpe (usando la función tipo '*addAll*' que ofrece el interfaz '*Collection*') es que hay que comprobar que todos los elementos de la colección son de tipo '*Target*' y no se ha introducido por error algún elemento que sea instancia de cualquier otra clase en cuyo caso, se lanza la excepción '*ClassCastException*'. Cuando esta excepción salta, el método '*add*', al tratarla, borra los enlaces nuevos que hasta ese momento se habían añadido y devuelve el valor falso para indicar que no se ha hecho ningún cambio.

9.2.2.11 remove

En cuanto a las definiciones de '*remove*', esta vez se requiere de tres variantes. La primera de ellas sirve para eliminar todos los enlaces por lo que no recibe ningún parámetro de entrada, la segunda elimina un único enlace y la tercera un conjunto de enlaces, pudiendo incluso ser uno o todos. Devuelven "cero" cuando pueden eliminar sus enlaces sin problemas, y "uno" cuando al eliminarlos, se rompe con las restricciones de la multiplicidad avisando así al usuario del método. Las dos últimas definiciones de '*remove*' que reciben como parámetro de entrada un enlace o un conjunto de enlaces concretos a eliminar, pueden devolver además el valor "–1" cuando se pide eliminar un enlace que no existe, en cuyo caso, no se hace nada.

El primer método '*remove*' simplemente limpia todos los elementos contenidos en el atributo que almacena los enlaces ('*links_hs*'). Después, en función del valor de la constante '*MIN*', determinará si se ha roto o no con las restricciones de multiplicidad mínima de la asociación. El segundo (el que elimina un único enlace) primero comprueba que realmente existe enlace con el objeto '*Target*' que se recibe como parámetro de entrada. Si existe, lo elimina y después hace la comprobación de si hay menos enlaces de los requeridos. Por último, en la tercera definición se trabaja sobre un colección de supuestos enlaces '*Target*' de entrada. En esta ocasión, si se especifica algún elemento de tipo '*Target*' que en realidad no está enlazado se ignora y sólo se trata como un error el hecho de que alguna instancia de la colección de entrada no sea de tipo '*Target*', en cuyo caso, se restaura los enlaces borrados y se devuelve "–1". Esto se consigue gracias al *casting* que en caso de fallar, lanza la excepción '*ClassCastException*'.

9.2.3 Visibilidad de la asociación

Nuevamente nos encontramos con el campo '*%visibility*' precediendo a la declaración de la clase '*Association*', al atributo '*targetRole*' y a los métodos de '*Association*' y ha de ser sustituido según sea la visibilidad de '*Target*' respecto a '*Source*'.

10 IMPLEMENTACIÓN DE ASOCIACIONES BIDIRECCIONALES

Una vez comprendido cómo se han de implementar las asociaciones unidireccionales será más fácil de entender la implementación de las bidireccionales; puesto que se basa en los mismos fundamentos salvo que esta vez hay que añadir un poco más de complejidad.

Las asociaciones bidireccionales, tal y como su nombre indica, ofrecen la posibilidad de navegar en ambos sentidos. Por tanto, en la práctica se trata de establecer una asociación unidireccional desde la clase '*A*' hacia '*B*' y otra de '*B*' hacia '*A*' y coordinarlas correctamente. Es decir, la única dificultad que presentan es que hay que tener en cuenta que cuando se establece un enlace en un sentido, también debe establecerse en el otro, y lo mismo cuando se elimina.

"En las asociaciones bidireccionales, se trata de establecer dos asociaciones unidireccionales en cada sentido pero éstas, han de ir a la par: si se establece una, se establece la otra y lo mismo cuando se eliminan."

Vamos a seguir diferenciando entre multiplicidad sencilla y compleja pero además hay que distinguir el lado del que estamos hablando, puesto que las asociaciones bidireccionales tienen multiplicidad en ambos lados. En el apartado 3: "*Tipos de asociaciones*" ya hemos visto que existen cuatro tipo de asociaciones bidireccionales. Las recordamos:

- Sencilla – sencilla
- Sencilla – múltiple
- Múltiple – sencilla
- Múltiple – múltiple

Cada tipo de asociación bidireccional lleva el nombre según como sea la multiplicidad en el lado '*A*' respecto al '*B*' primero, y como sea después en '*B*' respecto a '*A*'. Por ejemplo, una asociación del tipo '*A (0..1)* ↔ *(1..*) B*' tiene multiplicidad sencilla en el lado '*A*' y múltiple en '*B*' por lo que se considera una asociación sencilla – múltiple. En la implementación nos vamos a centrar en cada lado, por lo que si es sencillo, utilizaremos un código de multiplicidad sencilla y si es múltiple usaremos código de múltiple. Ambos tipos de código son capaces de comunicarse con el otro lado y establecer enlaces independientemente del tipo que sea.

Para entenderlo bien vamos a estudiar primero las sencillas – sencillas utilizando sólo código de multiplicidad sencilla en ambos extremos, luego veremos las múltiples – múltiples utilizando código de multiplicidad compleja para al final combinar las sencillas – múltiples y las múltiples – sencillas y ver cómo la comunicación entre ambos códigos no plantea ningún problema.

Esta solución tiene la ventaja inmediata de que es muy fácil cambiar de diseño en una asociación. Si estamos tratando con una asociación y cambiamos la multiplicidad pero permanece aún el mismo tipo de asociación bidireccional, bastará con cambiar las constantes de multiplicidad mínima, máxima o de obligatoriedad; según los casos. Sin embargo, si al cambiar la especificación de la asociación, algún lado cambia de tipo de multiplicidad, habrá que cambiar el código de la

clase interna '*Association*' así como su nombre. El atributo que gestiona la asociación seguirá teniendo el mismo nombre así como los métodos de control.

Entenderemos por clase '*Source*' a la clase del lado de la asociación que estemos tratando en un instante y por '*Target*' a la clase del extremo opuesto en la asociación.

10.1 Asociaciones sencillas – sencillas

Son cuatro posibles asociaciones las que nos podemos encontrar en este caso y las tenemos en la Figura 10.1.

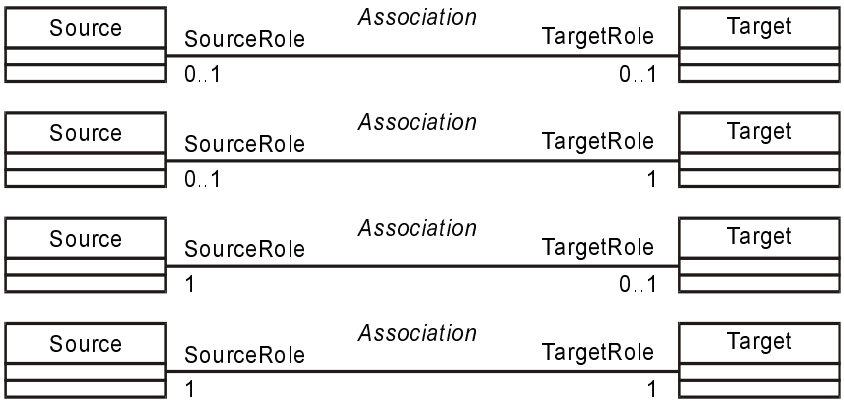

Figura 10.1: Asociaciones bidireccionales sencillas – sencillas.

Utilizaremos en cada lado de la asociación el código que presentamos a continuación. Dicho código sirve para almacenar un único enlace en cada instancia. Sólo presentamos el código en uno de los lados ('*Source*') puesto que en el otro '*Target*' hay que utilizar el mismo, salvo que habrá que intercambiar los roles y por tanto donde ponga '*Source*' habría que poner '*Target*', y viceversa. De esta forma, sólo hablaremos de la clase '*Source*' y se da por hecho que lo que se diga para esta clase o lado de la asociación se dice también para la clase '*Target*' pero invirtiendo los papeles de origen y de destino y cambiando la constante que define la multiplicidad si es que es distinta en cada extremo.

Veamos el código que implementa este tipo de asociaciones. Nuevamente, resaltamos en el código aquellas partes que son nuevas y presentan una diferencia sustancial con respecto a la plantilla de código vista en el apartado 9.1 "*Asociaciones unidireccionales sencillas*"

```
package %package;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_SingleBidirectional_%AssociationKind
         %targetRole = new %Source%Target_SingleBidirectional_%AssociationKind();

  %visibility class %Source%Target_SingleBidirectional_%AssociationKind {
```

```
/** Para representar multiplicidad muchos (* en UML)
 *  No se utiliza aquí pero se pone por uniformidad de interfaz con la
 *  implementación de otro tipo de asociaciones
 */
%visibility final static int MANY = -1;

/** Enlace de %Source a %Target obligatorio (true) u opcional (false) */
private final static boolean mandatory = %mandatory;

/** Almacena el enlace con %Target */
private %Target link = null;

/** Dice si la asociación es bidireccional o no
 *  @return true si la asociación es bidireccional, false si unidireccional
 */
%visibility boolean isBidirectional() {
  return true;
}

/** Dice si la asociación es obligatoria o no
 *  @return true si la asociación es obligatoria
 */
%visibility boolean isMandatory() {
  return mandatory;
}

/** Dice si la asociación es múltiple o no
 *  @return true si es múltiple, false si sencilla
 */
%visibility boolean isMultiple() {
  return false;
}

/** Devuelve la multiplicidad mínima del final de la asociación
 *  @return multiplicidad mínima
 */
%visibility long getMIN() {
  if (isMandatory()) {
    return 1;
  }
  return 0;
}

/** Devuelve la multiplicidad máxima del final de la asociación
 *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
 *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
 */
%visibility long getMAX() {
  return 1;
}

/** Dice si %Source cumple con las restricciones de la asociación
 *  @return true si sí las cumple, false en caso contrario
 */
%visibility boolean isValid() {
  return ((! mandatory) || (link != null));
}

/** Devuelve el número de enlaces activos que tiene el objeto
 *  @return Número de enlaces
 */
```

```
%visibility long numberOfLinks() {
  if (link == null) {
    return 0;
  }
  return 1;
}

/** Comprueba si %Source está asociado con un objeto %Target concreto
 *  @param query_link Supuesto enlace %Target
 *  @return true si existe dicho enlace, false en caso contrario
 */
%visibility boolean test(%Target query_link) {
  return link == query_link;
}

/** Devuelve el enlace %Target contenido en link
 *  Lanza la excepción UnsetAssociationException si la asociación no
 *  cumple con los requisitos de la asociación
 *  @return link (null si no existe ningún enlace)
 */
%visibility %Target get() throws UnsetAssociationException {
  if (! isValid()) {
    throw new UnsetAssociationException ("El objeto de clase %Source"
    + " no cumple con los requisitos de la asociación");
  }
  return link;
}

/** Añade un enlace con una instancia de %Target
 *  @param new_link Objeto con el que se intenta enlazar
 *  @return true si se establece el enlace, false si no se puede debido a
 *  que ya tiene otro enlace distinto, new_link no lo permite o
 *  new_link es nulo
 */
%visibility boolean add(%Target new_link) {
  synchronized (this) {
    // Si no se puede establecer el enlace, se termina
    if ((new_link == null) || ((link != null) && (link != new_link))) {
      return false;
    }
    // Si se intenta enlazar con un objeto con el que ya está enlazado, no
    // se hace nada pero se devuelve como si se hubiese establecido el enlace
    if (test(new_link)) {
      return true;
    }
    // Se establece enlace de %Source a %Target
    link = new_link;
    // Si en el otro extremo no está establecido el enlace y
    if ((! new_link.%sourceRole.test(%Source.this)) &&
        // no se puede establecer de %Target a %Source
        (! new_link.%sourceRole.add(%Source.this))) {
       // se elimina
       link = null;
       return false;
     }
     return true;
  }
}

/** Elimina el enlace establecido en link
 *  @return 0 si se elimina el enlace y los objetos %Source y link
 *  siguen cumpliendo con las restricciones de multiplicidad, 1 si %Source
```

```
     *  deja de cumplir con las restricciones, 2 si link deja de cumplir
     *  y 3 si ambos
     */
    %visibility int remove() {
      synchronized (this) {
        boolean invalid_link = false;
        %Target old_link = link;
        if (link != null) {
          // Se elimina el enlace
          link = null;
          // Si la solicitud de eliminar el enlace viene de otro objeto que no
          // sea el viejo enlace
          if  ((old_link.%sourceRole.test(%Source.this)) &&
               // se llama a %targetRole para que elimine su enlace con %Source
               (old_link.%sourceRole.remove(%Source.this) == 1)) {
            invalid_link = true;
          }
        }
        // Devuelve el valor correcto
        if (mandatory && invalid_link) {
          return 3;
        } else if (invalid_link) {
          return 2;
        } else if (mandatory) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra el enlace supuestamente establecido con un objeto %Target
     *  @param old_link Supuesto enlace de tipo %Target
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de multiplicidad, 1 si se deja de cumplir con las
     *  restricciones y -1 si no existe el enlace con link u old_link es null
     *  y por tanto, no se hace nada
     */
    %visibility int remove(%Target old_link) {
      if ((old_link != null) && test(old_link)) {
        return remove();
      }
      return -1;
    }

  }

}
```

En la Figura 10.2 tenemos el diagrama de clases para implementar un lado sencillo en una asociación bidireccional, y éste es idéntico al presentado en la Figura 9.2 debido a que estamos siguiendo las mismas estructuras y se ha logrado un interfaz uniforme.

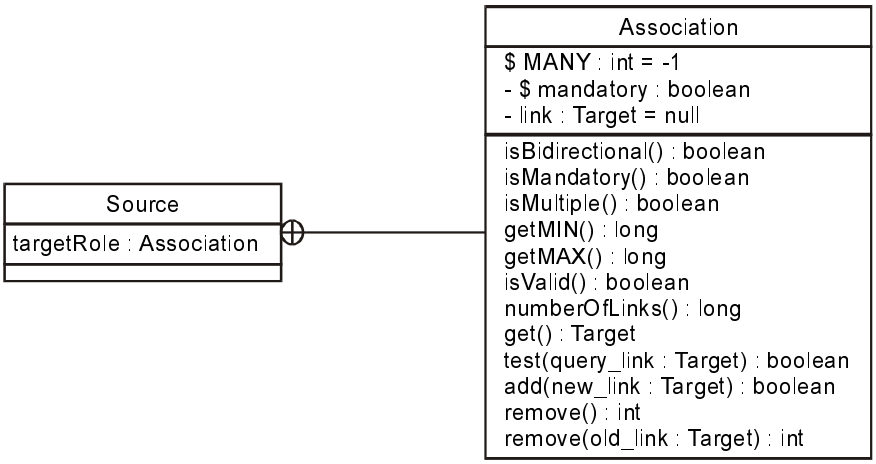

Figura 10.2: Asociación bidireccional, lado sencillo.

10.1.1 La clase '*Source*'

El esqueleto básico para la clase '*Source*' es idéntico al que se ha presentado en el apartado 9.1 "*Asociaciones unidireccionales sencillas*" y por tanto, no hay mucho que comentar. Nuevamente tenemos una única instancia de la clase '*Association*' mediante la cual se va a gestionar la asociación.

10.1.2 La clase '*Association*'

Aunque pocas diferencias se pueden encontrar en la implementación de esta clase a la implementación de la misma clase en las asociaciones unidireccionales sencillas, éstas son sustanciales y vitales.

Para empezar, nos encontramos nuevamente con la definición de la constante '*MANY*' pero por tratarse aquí de un extremo sencillo, realmente nunca va a ser utilizada. A continuación, se declara un atributo constante *booleano* que especifica si son obligatorios o no los enlaces de '*Source*' a '*Target*'. Dicho atributo es '*mandatory*'. También existe un atributo privado de nombre '*targetRole*' y de tipo '*Target*' donde se almacenarán los enlaces.

10.1.2.1 isBidirectional

Como estamos tratando con una asociación bidireccional, este método devuelve *true* directamente.

10.1.2.2 isMandatory

Esta función devolverá el valor contenido en el atributo '*mandatory*' para así, dar a conocer si la asociación es obligatoria o no.

10.1.2.3 isMultiple

Devuelve falso puesto que sólo se va a almacenar a lo sumo un único enlace con el extremo opuesto.

10.1.2.4 getMIN

Para conocer la multiplicidad mínima del extremo opuesto, se observa el valor que devuelve '*isMandatory*', si es verdadero, se devuelve uno y si no cero.

10.1.2.5 getMAX

Como estamos tratando con un extremo de asociación sencillo, la multiplicidad máxima es uno y es ese el valor que directamente se devuelve como valor de retorno.

10.1.2.6 isValid, numberOfLinks, test y get

Estos métodos declarados para el conocimiento del estado de los enlaces están implementados de igual forma que los presentados en el apartado 9.1 "*Asociaciones unidireccionales sencillas*" por lo que no los volvemos a explicar.

10.1.2.7 add

El método '*add*' que sirve para el establecimiento de enlaces funciona de una forma similar que la presentada en el apartado 9.1 "*Asociaciones unidireccionales sencillas*". Ahora bien, hay una diferencia fundamental y es que cuando se establece un enlace de una clase '*Source*' a una '*Target*', hay que asegurarse que también se establece de '*Target*' a '*Source*'. Para ello, cuando se llama al método '*add*' de una instancia de '*Source*' para que establezca enlace con una instancia '*Target*', el método primero comprueba que se dan las condiciones para establecer enlace y si se dan, lo establece. Después, comprueba si el otro extremo '*Target*' tiene establecido enlace con '*Source*' invocando a '*test*' de la instancia '*Target*'. Si lo tiene, termina y si no, invoca al método '*add*' de '*Target*' para que establezca enlace con '*Source*'. Si esta última llamada se ejecuta correctamente, se termina y si no, se rompe el enlace que se había establecido con '*Target*' y se indica (devolviendo *false*) que no se ha podido establecer enlace. Cuando la instancia de tipo '*Target*' recibe la llamada a '*add*', tras comprobar que puede establecer enlace con la instancia '*Target*', llamará primero al método '*test*' de la clase '*Source*' que devolverá *true* puesto que '*Source*' ya ha establecido el enlace por lo que sólo se encargará del establecimiento del enlace de '*Target*' a '*Source*' y terminará sin necesidad de llamar a '*add*' de '*Source*'.

De esta forma, la definición de '*add*' en cada una de las clases de los extremos de la asociación sirve para establecer enlace en ambos sentidos requiriendo una única invocación al método, siendo igual a qué extremo de la asociación se llama primero (si a '*Source*' o '*Target*'). Gracias a la llamada al método '*test*' del lado opuesto, el propio método '*add*' puede distinguir cuando tiene que establecer enlace también en sentido contrario (y por tanto, invocar a '*add*' del otro extremo) puesto que ha sido invocado desde fuera. También, puede distinguir cuando sólo tiene que establecer enlace en un único sentido debido a que ha sido invocado desde el método '*add*' del otro extremo.

A continuación, en la Figura 10.3 se muestra el diagrama de secuencias en el que un objeto cualquiera pide establecimiento de enlace a una instancia de '*Source*' con otra de '*Target*'. En el diagrama, el establecimiento se realiza con éxito. Recordamos que el código del método 'add' es prácticamente el mismo tanto en '*Source*' como en '*Target*' y sólo se diferencian en que las referencias se intercambian.

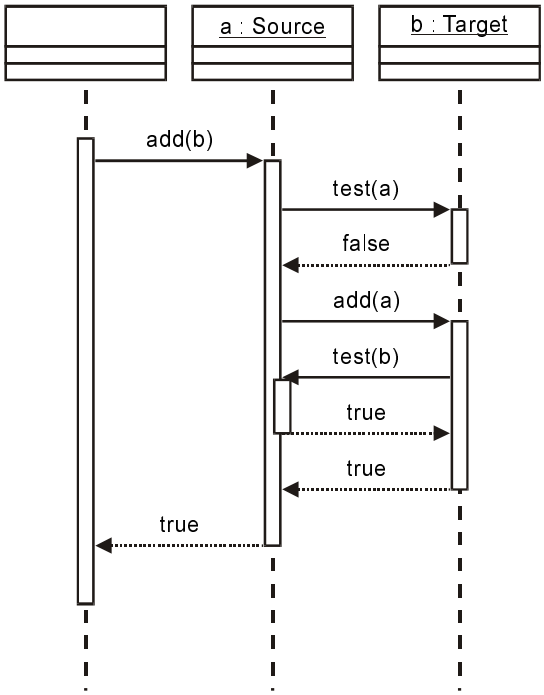

Figura 10.3: Diagrama de secuencia para el establecimiento de enlace entre una instancia '*a*' de '*Source*' y otra '*b*' de '*Target*'.

10.1.2.8 remove

'*remove*' funciona de la misma forma que '*add*' pero esta vez, su función es la eliminación del enlace. Comprendiendo como funciona la declaración del método '*remove*' del apartado 9.1 "*Asociaciones unidireccionales sencillas*" y como se logra el establecimiento de doble enlace en el método '*add*' del apartado anterior, es fácil entender la definición del método '*remove*' de la plantilla.

Por tanto, tan sólo destacaremos que cuando se elimina un enlace, y hay que hacerlo en ambos sentidos, la instancia que inicialmente ha sido invocada, cuando tiene que eliminar el enlace en el sentido contrario, invoca a la segunda definición de '*remove*', es decir, la que recibe como parámetro de entrada el objeto con el que se quiere borrar enlace. En esta ocasión, podría llamarse a la primera pero más adelante comprenderemos por qué se hace mediante la segunda.

Además, en comparación con el método '*add*' del apartado anterior, hay que destacar también que esta vez, para eliminar un enlace en ambos sentidos hay que comprobar que existe enlace en el sentido contrario, y por tanto, el valor de retorno de '*test*' ha de ser verdadero y no falso.

En la Figura 10.4 puede verse un diagrama de secuencias en el que se rompe un enlace entre una instancia '*a*' y otra '*b*' correctamente. La invocación inicial de '*remove*' se hace sobre la instancia '*a*' y obsérvese que se hace sin ningún parámetro de entrada, mientras que la invocación posterior de '*remove*' de '*b*' a '*a*' sí.

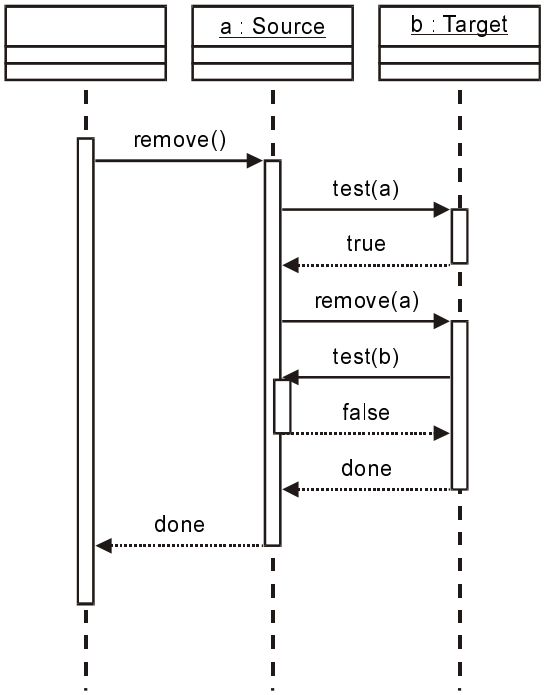

Figura 10.4: Diagrama de secuencia para la eliminación de enlace entre una instancia '*a*' de '*Source*' y otra '*b*' de '*Target*'.

Siguiendo el diagrama y el código de '*remove*()', primero se invoca a '*remove*' de '*a*' para que elimine su enlace. '*a*' elimina su enlace que es '*b*' y comprueba mediante '*test*' si '*b*' todavía tiene enlace con '*a*'. Como resulta que sí por lo que '*remove*' de '*a*' invoca a '*remove(a)*' en '*b*' que borra su enlace y tiene que comprobar si '*a*' tiene todavía enlace con '*b*'. Debido a que '*a*' ya ha eliminado el enlace ('*test(b)*' en '*a*' devolverá *false*) '*b*' terminará la ejecución de '*remove(a)*' pasando el control a '*remove()*' de '*a*' que a su vez finalizará devolviendo el control a la instancia que le pidió borrar el enlace de '*a*' a '*b*'.

Ya hemos visto que existen dos definiciones de '*remove*' que hacen exactamente lo mismo y sólo se diferencian en que la segunda comprueba si se puede eliminar enlace con la instancia recibida como parámetro de entrada y en caso de no existir dicho enlace, se devuelve el valor "–1", mientras que la primera rompe el enlace, independientemente de cual sea.

Del método '*remove*' nos falta por explicar los valores numéricos que pueden devolver y su significado. Devolverán "cero" cuando la eliminación de un enlace sea correcta y no presente ningún problema y "uno" cuando se deje el objeto tipo '*Source*' en un estado no válido por no cumplir con las especificaciones de la asociación. Además, puede devolver el valor "dos" para indicar que se ha borrado el enlace sin problemas pero esta vez, es el objeto del otro extremo de la asociación el que se ha quedado en estado no válido. Por último, si se obtiene un "tres" tras la llamada a '*remove*' significará que ambos lados del antiguo enlace se han quedado en un estado que incumple con las restricciones impuestas por la asociación. Estos valores pueden servir de confirmación a los usuarios de la clase de que la eliminación de enlaces se está haciendo tal y como lo tienen previsto.

10.1.3 Visibilidad

La visibilidad nuevamente se ha de sustituir en el campo '*%visibility*' pero como veremos en el capítulo 13 "*Visibilidad en las asociaciones*", no vamos a obtener el comportamiento que deseamos en su totalidad. De momento, dejamos aparcado este tema puesto que merece un capítulo aparte y nos quedamos con la idea de que sólo hay que sustituir el valor correspondiente a la visibilidad en el campo '*%visibility*'.

10.2 Asociaciones múltiples – múltiples

Presentamos ahora el caso de las asociaciones múltiples – múltiples y la razón de ponerlas por delante de las múltiples – sencillas y de las sencillas – múltiples es que nuevamente el código que hagamos para la clase '*Source*' sirve también para la clase '*Target*' siempre y cuando se cambien los nombres de las clases y roles correctamente. Por tanto, nuevamente presentaremos sólo el código del lado '*Source*' y, cuando presentemos las asociaciones sencillas – múltiples o múltiples – sencillas veremos que basta con poner el código de enlace sencillo en la clase que sólo puede tener a lo sumo un único enlace, y el de múltiple en el lado que puede tener más de uno, para que la asociación funcione correctamente.

En la Figura 10.5 están representados los nueve casos posibles de asociaciones bidireccionales *UML* que nos podemos encontrar atendiendo a la multiplicidad compleja en ambos lados.

A continuación, mostramos el código necesario en '*Source*' y volvemos a resaltar los cambios sustanciales respecto a lo que se ha visto hasta ahora con un color de fondo. El código es una combinación entre el código de las asociaciones unidireccionales múltiples y el código de los extremos sencillos de las asociaciones bidireccionales.

```
package %package;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_MultipleBidirectional_%AssociationKind
       %targetRole = new %Source%Target_MultipleBidirectional_%AssociationKind();

  %visibility class %Source%Target_MultipleBidirectional_%AssociationKind {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = %min_multiplicity;

    /** Para representar multiplicidad muchos (* en UML) */
    %visibility final static int MANY = -1;
    /** Multiplicidad máxima del enlace */
    private final static long MAX = %max_multiplicity;

    /** HashSet donde se almacenan los enlaces con instancias %Target */
    private HashSet links_hs = new HashSet();

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
```

```
     */
    %visibility boolean isBidirectional() {
      return true;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return (MIN > 0);
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return ((MAX > 1) || (MAX == MANY));
    }
```

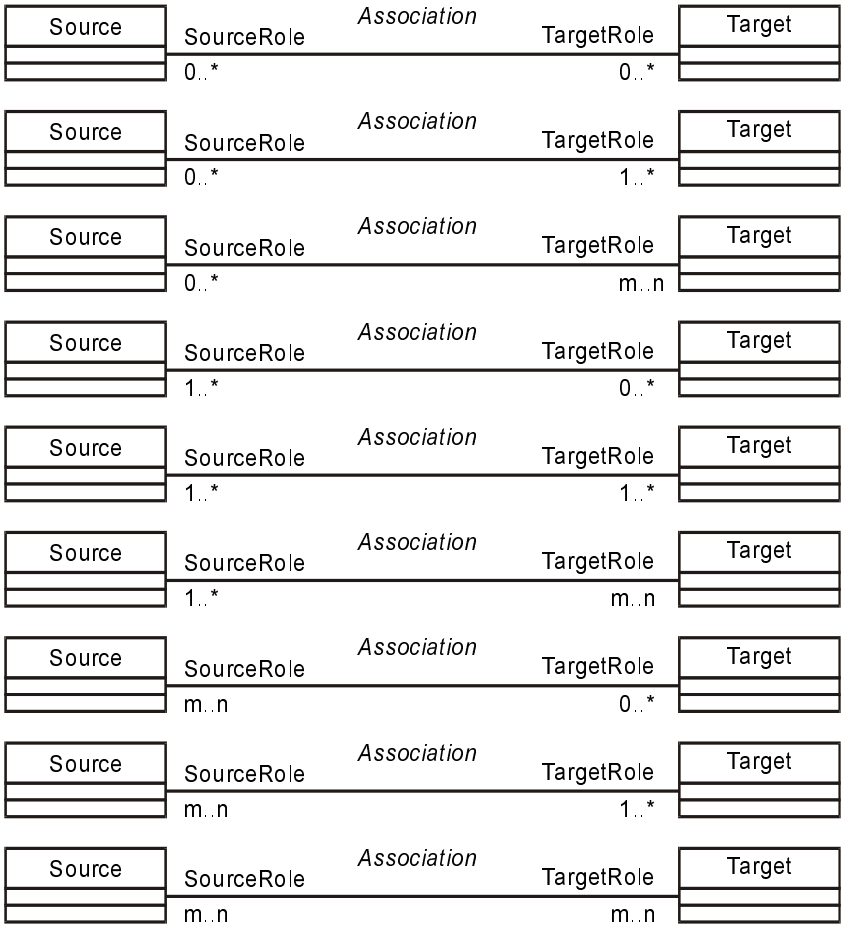

Figura 10.5: Asociaciones bidireccionales múltiple – múltiple.

```
/** Devuelve la multiplicidad mínima del final de la asociación
 *  @return multiplicidad mínima
 */
%visibility long getMIN() {
  return MIN;
}

/** Devuelve la multiplicidad máxima del final de la asociación
 *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
 *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
 */
%visibility long getMAX() {
  return MAX;
}

/** Dice si %Source cumple con las restricciones de la asociación
 *  @return true si sí las cumple, false en caso contrario
 */
%visibility boolean isValid() {
  return ((links_hs.size() >= MIN) &&
          ((links_hs.size() <= MAX) || (MAX == MANY)));
}

/** Devuelve el número de enlaces activos que tiene el objeto
 *  @return Número de enlaces
 */
%visibility long numberOfLinks() {
  return links_hs.size();
}

/** Comprueba si %Source está asociado con un objeto %Target concreto
 *  @param query_link Supuesto enlace %Target
 *  @return true si existe dicho enlace, false en caso contrario
 *  query_link == null implica comprobar si no existen enlaces
 */
%visibility boolean test(%Target query_link) {
  if (query_link == null) {
    return links_hs.isEmpty();
  }
  return links_hs.contains(query_link);
}

/** Comprueba si %Source está asociado con una colección de objetos %Target
 *  @param query_links Colección de supuestos enlaces
 *  @return true si todos los elementos de links_hs están enlazados
 *  con %Source o si links_hs está vacío y query_links también o es null
 */
%visibility boolean test(Collection query_links) {
  if (query_links == null) {
    return links_hs.isEmpty();
  }
  return links_hs.containsAll(query_links);
}

/** Devuelve una colección con la copia de los enlaces con %Target
 *  contenidos en links_hs. Lanza la excepción UnsetAssociationException si
 *  la asociación no cumple con los requisitos de la asociación
 *  @return Copia de links_hs
 */
%visibility Collection get() throws UnsetAssociationException {
  if (! isValid()) {
    throw new UnsetAssociationException ("El objeto de clase %Source"
```

```
        + " no cumple con los requisitos de la asociación");
      }
      return (Collection) links_hs.clone();
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se realiza el enlace o ya existía, false si no se puede
     *  debido a que se excede la multiplicidad máxima o new_link es null
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        if (new_link != null) {
          // Si se intenta enlazar con un objeto que ya está enlazado, no se
          // hace nada pero se devuelve como si se hubiese establecido el enlace
          if (test(new_link)) {
            return true;
          }
          // Si añadir new_link no excede la multiplicidad máxima
          if ((MAX == MANY) || (links_hs.size() < MAX)) {
            // se establece el enlace de %Source a %Target
            links_hs.add(new_link);
            // Si en el otro extremo no está establecido el enlace y
            if ((! new_link.%sourceRole.test(%Source.this)) &&
                // no se puede establecer de %Target a %Source
                (! new_link.%sourceRole.add(%Source.this))) {
              // se elimina
              links_hs.remove(new_link);
            } else {
              return true;
            }
          }
        }
        return false;
      }
    }

    /** Añade un enlace con cada uno de los elementos de new_links_hs
     *  @param new_links Colección de elementos %Target con los que se
     *  intenta enlazar
     *  @return true si se establecen TODOS los enlaces, false si se excede de
     *  la multiplicidad máxima o alguno de los nuevos enlaces no se puede
     *  establecer
     */
    %visibility boolean add(Collection new_links) {
      boolean stop = false; // Parada forzosa
      %Target last_link_i = null;
      %Target last_link_ii = null;
      Iterator i;

      synchronized (this) {
        if (new_links != null) {
          // Sobre una copia de new_links se eliminan aquellos objetos
          // %Target que tengan establecido el enlace previamente
          HashSet copied_new_links = new HashSet (new_links);
          i = new_links.iterator();
          while (i.hasNext()) {
            try {
              last_link_i = (%Target) i.next();
            } catch (ClassCastException e) {
              // Si algún elemento no es de tipo %Target no se hace nada
              return false;
```

```
            }
            if (links_hs.contains(last_link_i)) {
              copied_new_links.remove(last_link_i);
            }
          }

          // Si añadir los nuevos elementos no excede de la multiplicidad máxima
          if ((MAX == MANY) ||
              ((copied_new_links.size() + links_hs.size()) <= MAX)) {
            // por cada elemento de copied_new_links se establecerá enlace
            i = copied_new_links.iterator();
            while (i.hasNext() && (! stop)) {
              last_link_i = (%Target) i.next();
              // de %Source a %Target y de
              links_hs.add(last_link_i);
              // de %Target a %Source
              if (! last_link_i.%sourceRole.add(%Source.this)) {
                stop = true;
              }
            }

            // Si la parada es forzosa <=> algún enlace no se establece
            if (stop) {
              // se borran los nuevos enlaces hasta ahora establecidos
              i = copied_new_links.iterator();
              last_link_ii = (%Target) i.next();
              while (last_link_i != last_link_ii) {
                // de %Source a %Target
                links_hs.remove(last_link_ii);
                // y de %Target a %Source
                last_link_ii.%sourceRole.remove(%Source.this);
                last_link_ii = (%Target) i.next();
              }
            // Si no es forzosa, se termina con éxito
            } else {
              return true;
            }
          }
        }
        return false;
      }
    }

    /** Elimina todos los enlaces establecidos en links_hs
     *  @return 0 si se eliminan todos los enlaces y %Source y las instancias
     *  de links_hs siguen cumpliendo con las restricciones de la asociación,
     *  1 si %Source deja de cumplir con las restricciones, 2 si al menos un
     *  elemento de links_hs deja de cumplirlas y 3 si ambos (%Source y al
     *  menos un elemento de links_hs)
     */
    %visibility int remove() {
      boolean invalid_old_link = false;
      synchronized (this) {
        // Por cada elemento (last_link) de links_hs
        Iterator i = ((HashSet) links_hs.clone()).iterator();
        while (i.hasNext()) {
          %Target last_link = (%Target) i.next();
          // se elimina el enlace de %Source a last_link
          links_hs.remove(last_link);
          // y de last_link a %Source
          if (last_link.%sourceRole.remove(%Source.this) == 1) {
            invalid_old_link = true;
```

```
        }
      }
      // Devuelve el valor correcto
      if ((MIN > 0) && invalid_old_link) {
        return 3;
      } else if (invalid_old_link) {
        return 2;
      } else if (MIN > 0) {
        return 1;
      }
      return 0;
    }
  }

  /** Borra un enlace establecido con un objeto %Target
   *  @param old_link Enlace de tipo %Target a borrar
   *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
   *  restricciones de la asociación, 1 si se deja de cumplir con ellas en
   *  %Source, 2 si en %targetRole, 3 si en ambos y -1 si no existe el enlace
   *  con %targetRole u old_link es null
   */
  %visibility int remove(%Target old_link) {
    synchronized (this) {
      boolean invalid_old_link = false;
      if ((old_link != null) && test(old_link)) {
        // Elimina el enlace de %Source a %Target
        links_hs.remove(old_link);
        // Si la solicitud de eliminar el enlace viene de otro objeto que no
        // sea el viejo enlace
        if  ((old_link.%sourceRole.test(%Source.this)) &&
             // se llama a old_link para que elimine su enlace con %Source
             (old_link.%sourceRole.remove(%Source.this) == 1)) {
          invalid_link = true;
        }
        // Devuelve el valor correcto
        if ((links_hs.size() < MIN) && (invalid_old_link)) {
          return 3;
        } else if (invalid_old_link) {
          return 2;
        } else if (links_hs.size() < MIN) {
          return 1;
        }
        return 0;
      }
      return -1;
    }
  }

  /** Borra los enlaces establecidos con un conjunto de objetos %Target
   *  @param old_links Colección de objetos %Target cuyo enlace con
   *  %Source se quiere borrar
   *  @return 0 si se eliminan los enlaces solicitados y se sigue cumpliendo
   *  con las restricciones de la asociación, 1 si %Source deja de cumplir con
   *  las restricciones, 2 si al menos un elemento de old_links deja de cumplir
   *  y 3 si ambos (%Source y al menos un elemento de old_links_hs). -1 si
   *  algún elemento de old_links no es de tipo %Target u old_links es null y
   *  por tanto, no se hace nada
   */
  %visibility int remove(Collection old_links) {
    boolean invalid_old_link = false;
    %Target last_link = null;
    Iterator i;
```

```
      if (old_links == null) {
        return -1;
      }
      // Se comprueba que todos los elementos de old_links son de tipo %Target
      i = old_links.iterator();
      while (i.hasNext()) {
        try {
          last_link = (%Target) i.next();
        } catch (ClassCastException e) {
          return -1;
        }
      }

      synchronized (this) {
        // Se borran los enlaces
        i = old_links.iterator();
        while (i.hasNext()) {
          last_link = (%Target) i.next();
          // de %Source a last_link
          links_hs.remove(last_link);
          // y de last_link a %Source
          if (last_link.%sourceRole.remove(%Source.this) == 1) {
            invalid_old_link = true;
          }
        }

        // Devuelve el valor correcto
        if ((links_hs.size() < MIN) && (invalid_old_link)) {
          return 3;
        } else if (invalid_old_link) {
          return 2;
        } else if (links_hs.size() < MIN) {
          return 1;
        }
        return 0;
      }
    }

  }

}
```

En la Figura 10.6 se encuentra el diagrama de clases del código que acabamos de representar. En esta ocasión, coincide con el presentado en la Figura 9.4 es decir, con el diagrama de clases de las asociaciones unidireccionales múltiples.

Nada nuevo hay que reseñar de la clase '*Source*' por lo que directamente pasamos a ver la clase '*Association*'.

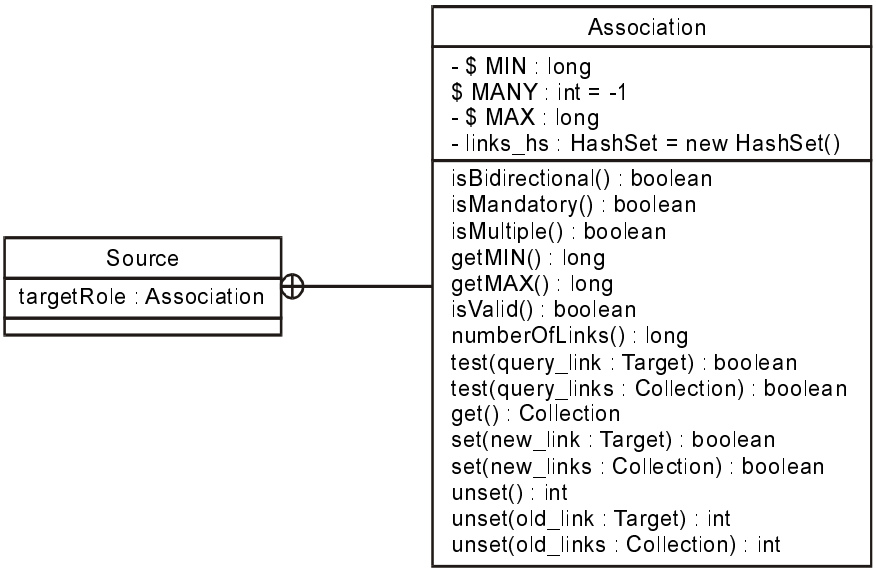

Figura 10.6: Asociación bidireccional, lado múltiple.

10.2.1 La clase '*Association*'

Prácticamente todo lo que se ha dicho y presentado en el apartado 9.2 de "*Asociaciones unidireccionales múltiples*" es válido para este tipo de asociaciones con lo que nos centraremos en lo que verdaderamente es novedoso. Además, como esta vez estamos hablando de enlaces bidireccionales, también se aplican las ideas que se han visto en el apartado 10.1 "*Asociaciones sencillas – sencillas*" en lo referente a la recursividad recíproca para el establecimiento y borrado de enlaces en ambos sentidos.

Como existe la posibilidad de enlazar con más de dos objetos habrá que especificar en las constantes '*MIN*' y '*MAX*' la multiplicidad mínima y máxima y el *HashSet* '*links_hs*' se utilizará para almacenar los enlaces.

10.2.1.1 isBidirectional

Devuelve verdadero puesto que se trata de código para la gestión de asociaciones bidireccionales.

10.2.1.2 isMandatory

Este método devuelve *true* en función de que la multiplicidad mínima especificada en la constante "*MIN*" sea mayor que cero.

10.2.1.3 isMultiple

Generalmente devolverá verdadero puesto que el código de la plantilla que se ha mostrado está pensado para trabajar con múltiples enlaces. No obstante, puede ocurrir que la multiplicidad mínima y máxima estén definidas como 0 y 1 o 1 y 1, en cuyo caso, la asociación es sencilla. La codificación de este método comprueba que esto no sea así para poder devolver verdadero y si no, como valor de retorno dará falso.

10.2.1.4 getMIN, getMAX, isValid, numberOfLinks, test y get

Estos métodos son una copia exacta de los vistos en el apartado 9.2 con las "*Asociaciones unidireccionales múltiples*".

10.2.1.5 add

Existen dos definiciones del método '*add*'. La primera añade un único enlace y la segunda un conjunto de enlaces. Debido a que se está trabajando siempre sobre un conjunto de enlaces y que cuando se establece uno, también hay que establecerlo en el otro sentido, el código de ambas definiciones mezcla la complejidad de los métodos '*add*' vistos en los apartados anteriores. Por un lado, hay que actuar sobre una colección de objetos y no se deben almacenar más enlaces de los permitidos por la constante de multiplicidad máxima, y por otro lado, si se puede establecer un enlace pedido, hay que llamar al lado opuesto del enlace para que también lo establezca y por tanto, utilizar recursividad recíproca.

El método '*add*' que sirve para establecer enlaces con más de un objeto tiene varias particularidades. Lo primero que hace es eliminar de la colección de entrada aquellos elementos con los que ya hay establecido un enlace puesto que no hace falta trabajar con ellos. La eliminación de elementos repetidos se ha hecho mediante el recorrido de un iterador de la colección de entrada. Se podría haber hecho de golpe, mediante una única sentencia, pero se ha optado por hacerlo así porque de ambas formas, al final, sea directa o indirectamente siempre se hace un recorrido por dentro de la colección pero haciéndolo en nuestro código directamente, podemos aprovechar para hacer un *casting* de los elementos. Gracias a este *casting*, antes de hacer ningún cambio podemos detectar si algún elemento no es de tipo '*Target*' en cuyo caso, se lanza la excepción '*ClassCastException*' y '*add*' puede finalizar señalando que no se ha realizado ningún cambio.

Otra particularidad es que '*add*' toma de uno en uno los posibles nuevos enlaces y llama al otro lado de cada nuevo enlace para establecer la asociación (en el sentido de '*T*arget' a '*Source*') existiendo la posibilidad de que en un momento dado, uno de los extremos conteste diciendo que no puede establecerlo. En tal situación, '*add*' deja de pedir el establecimiento de enlaces e inicia un bucle en el que pide a los extremos que hasta ese momento sí han establecido enlace que lo eliminen. Una vez finalizado el proceso de anulación se devuelve falso para indicar que no se ha hecho ningún cambio.

Esta segunda definición de '*add*' que se acaba de exponer sólo puede ser utilizada de forma directa por el usuario de la asociación y nunca será llamada de forma interna por el otro extremo de la asociación como puede ocurrir con la primera definición.

10.2.1.6 remove

Existen tres definiciones de '*remove*' para borrar uno, varios o todos los enlaces. Devuelven "cero" cuando se eliminan los enlaces solicitados, "uno" cuando se eliminan pero se deja el lado inicialmente invocado en estado no válido, es decir, sin cumplir con las restricciones de multiplicidad. El valor "dos" indica que es el otro lado de la asociación el que se ha quedado en estado no válido y "tres" si ambos extremos juntos dejan de cumplir las restricciones. Además, se puede obtener el valor "menos uno" en la llamada a los métodos que eliminan uno o un conjunto de enlaces especificados (segunda y tercera definición) en el caso de que los enlaces se hayan especificado de forma errónea porque en realidad no existen.

Comprendiendo los métodos '*remove*' que hemos visto hasta ahora sólo hay un aspecto que destacar en la tercera declaración (la que elimina un conjunto de enlaces). Puede ocurrir que se introduzca una colección que contenga un elemento que no es de tipo '*Target*' en cuyo caso, el

método ha de parar la ejecución y no realizar ningún cambio. Por tanto, lo primero que se hace tras comprobar que el parámetro de entrada no es *null*, es un recorrido por la colección en el que se hace un *casting* capaz de parar la ejecución en el caso de que una instancia no sea de tipo '*Target*'. Dicho *casting* de todos los elementos de la colección se hace lo primero de todo y no al mismo tiempo que se van borrando los enlaces para evitar un posible problema. Este método '*remove*' podría haberse implementado de forma que se recorriese la colección y al mismo tiempo, se fuesen eliminando los enlaces. En el momento que se encontrase un elemento que no fuese de tipo '*Target*', se pasaría a restaurar los enlaces para dejarlo todo tal y como estaba antes de la llamada a '*remove*'. Ahora bien, mientras se van eliminando los enlaces, puede ocurrir que un hilo de ejecución distinto establezca enlaces con alguno de las instancias recién eliminadas de la asociación y éstas, dejen de permitir nuevos enlaces. Al fallar el *casting* de algún elemento posterior y restaurar los enlaces, este método '*remove*' se puede encontrar con que alguno de los antiguos enlaces no le permite reestablecerlo dando lugar a una situación de error grave e irreparable.

10.3 Asociaciones sencillas – múltiples y múltiples – sencillas

En este tipo de asociaciones nos encontramos los seis ejemplos que pueden verse en la Figura 10.7 y los seis inversos de la Figura 10.8.

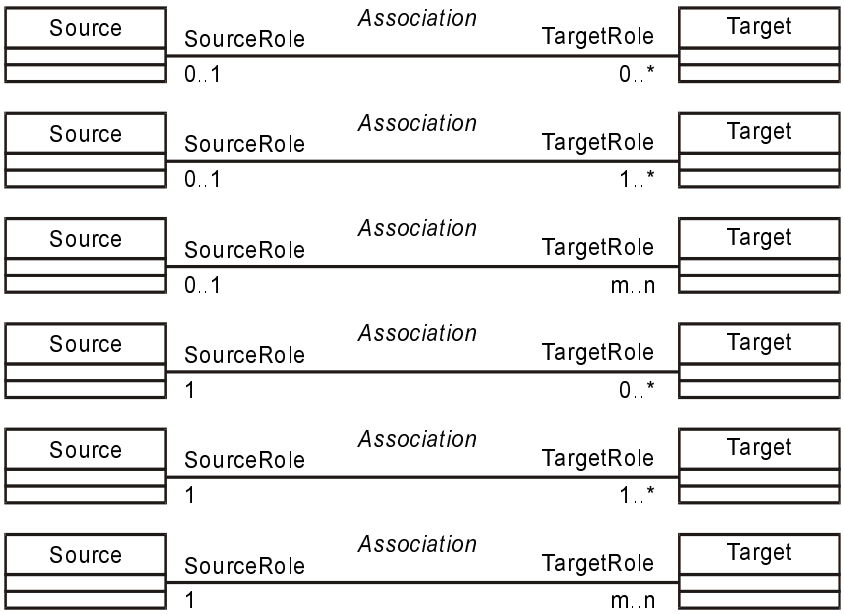

Figura 10.7: Asociaciones bidireccionales sencillas – múltiples.

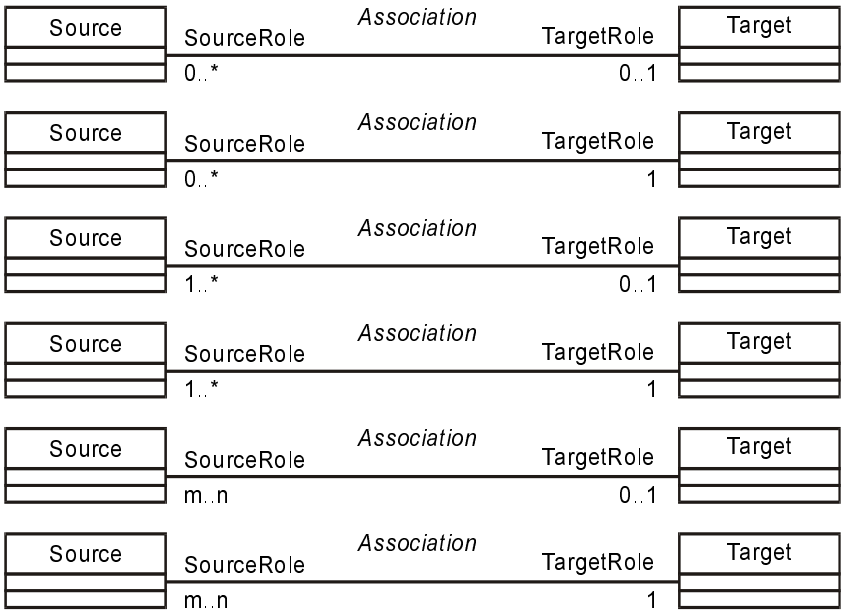

Figura 10.8: Asociaciones bidireccionales múltiples – sencillas.

Para su implementación basta con utilizar el código para asociación bidireccional sencilla en la clase cuya multiplicidad es sencilla en el otro extremo y el código para asociación bidireccional múltiple en la clase cuya multiplicidad en el otro lado es compleja. Una vez definida la clase '*Association*' en ambos extremos, hay que establecer las constantes de multiplicidad para indicar si es obligatoria o no en la clase con enlaces sencillos y la multiplicidad mínima y máxima en la clase compleja cuyos enlaces son varios.

Es en este apartado donde cobra pleno sentido la definición del método '*remove*' para multiplicidades sencillas que recibe como parámetro el enlace que hay que borrar. Técnicamente no tiene sentido que se llame a la clase '*Association*' para que borre su único enlace y se le tenga que especificar cuál es puesto que sólo hay uno. Ahora bien, cuando se pide borrar un enlace, se hace a través de una de las instancias '*A*' o '*B*' y es ésta la encargada de borrar el enlace en un sentido y de pedir a la instancia de la clase opuesta que lo elimine en el otro sentido. El problema está en que una clase '*A*' o '*B*' no sabe qué tipo de multiplicidad tiene respecto el lado opuesto por lo que resulta más cómodo programar una llamada '*remove*' especificando el enlace concreto que hay que eliminar (pasándose como parámetro de entrada la propia instancia que lo invoca). Si se hace sobre una clase '*Association*' sencilla, esta comprobará que existe dicho enlace y lo eliminará, mientras que si se hace sobre una múltiple, sabrá qué enlace concreto borrar. Es así como se consigue la independencia de multiplicidad en ambos lados de la asociación y el método '*remove*' sin parámetros semánticamente significa borrar todos los enlaces ya sean uno (multiplicidad sencilla) o varios (múltiple).

Para finalizar este apartado queremos observar que el código presentado para las asociaciones unidireccionales múltiple y el presentado para el lado múltiple en una asociación bidireccional son suficientes para representar el conjunto de todas las asociaciones unidireccionales y bidireccionales. De este modo, es perfectamente válido usar las clases '*Association*' inicialmente

pensadas para soportar multiplicidades complejas y aplicarlas en multiplicidades sencillas. Debido a que ambas piezas de código tienen dos constantes para expresar la multiplicidad mínima y máxima, bastaría con asignar a dichas constantes los valores 0 y 1 para asociaciones sencillas opcionales (0..1) y 1 y 1 para las sencillas obligatorias (1..1).

No obstante, aunque esto funcionaría, el hecho de usar la clase '*HashSet*' para almacenar a lo sumo un único enlace no sólo complica el código interno de la clase '*Association*' sino que también requiere disponer de más recursos de los realmente necesarios, es decir, no es eficiente. Además, dificulta su utilización al usuario al tener que trabajar con colecciones de objetos que sólo almacenan un único elemento aunque también la facilita, al uniformar con las múltiples. Las asociaciones sencillas son muy comunes, por lo que vale la pena implementar para ellas un código más eficiente.

Podría ser aconsejable recurrir al código para una asociación múltiple siendo ésta en realidad sencilla cuando se prevean futuros cambios de diseño en la que la asociación pueda pasar a ser compleja. Ocurriría lo mismo si esto fuese al revés, es decir, si se tuviese una asociación múltiple y, más adelante, ésta cambia a sencilla. Esto tendría la ventaja de que sólo habría que modificar las constantes de multiplicidad en la clase '*Association*', pero nada más en el código de las clases participantes y en las zonas donde se realice cualquier operación con los enlaces (en principio). Ahora bien, nótese que se ha conseguido un interfaz casi uniforme en ambas especificaciones y que si en lugar de usar el código complejo siempre, si se cambia de código complejo a simple, o al revés, según las necesidades de cada momento, se tendrá siempre el código más eficiente para cada caso. Si se hace así, se va a notar diferencia de utilización es el método '*get*' pues en un caso devuelve un conjunto de enlaces y en otro un único enlace. También se notaría diferencia en las antiguas llamadas a '*add*' y '*remove*' que actúan sobre un conjunto de elementos. En el resto de los métodos no se debe notar mucha diferencia puesto que el código complejo soporta el trato de los enlaces de uno en uno además de por conjuntos o agrupaciones.

11 ASOCIACIONES REFLEXIVAS

Las asociaciones reflexivas son aquellas que parten de la misma clase a la que llegan, es decir, ocurren cuando una clase se asocia consigo misma. Hasta ahora hemos hablado en las asociaciones, para referirnos a las clases participantes en ellas, de una clase '*A*' y otra '*B*' o una clase origen ('*Source*') y otra destino ('*Target*'). Ahora, hablaremos de una única clase '*A*' o de una clase origen y destino al mismo tiempo. Vamos a ver en este apartado hasta que punto nos sirve lo que se ha visto hasta ahora para poder aplicarlo en este tipo particular de asociaciones.

11.1 Asociaciones unidireccionales reflexivas

En la Figura 11.1 se halla representada una clase genérica con una asociación unidireccional reflexiva en la que no se ha hecho ninguna mención sobre su multiplicidad.

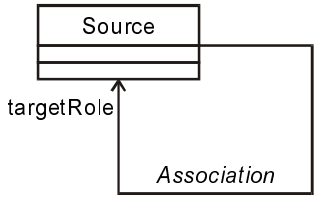

Figura 11.1: Asociación unidireccional reflexiva.

Este tipo de asociaciones reflexivas no presentan ningún problema respecto a lo que se ha visto en el apartado 9: "*Implementación de asociaciones unidireccionales*". Dado que cada clase '*Source*' tiene implementada internamente otra clase para el control de la asociación, habrá que utilizar el código de dicha clase interna y cambiarlo para que la clase '*Target*' a la que apunte la asociación sea la propia clase '*Source*'; sin más complicaciones.

Sólo tendremos en cuenta que a efectos de ingeniería inversa, a la clase interna que gestiona la asociación le daremos un nombre especial, según se ha especificado en el apartado 7.2: "*Convención de nombres*".

11.2 Asociaciones bidireccionales reflexivas

Las asociaciones bidireccionales presentan una pequeña complicación, y es que esta vez el hecho de que una asociación bidireccional implique un doble enlace, forzará a que dentro de la misma clase necesitemos de dos clases internas de asociación (una para cada lado de la asociación o por cada rol que puede jugar la clase) y por consiguiente, dos atributos para su gestión. En la Figura 11.2 se encuentra la representación genérica en *UML* de la asociación bidireccional reflexiva. En la ilustración no se han representado las multiplicidades.

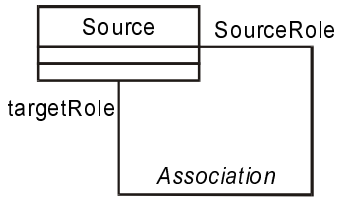

Figura 11.2: Asociación bidireccional reflexiva.

Para entender bien el problema, podemos ver este tipo de asociaciones como un problema de "doble personalidad" en las que unas veces una clase actúa con un rol particular y otras con el segundo rol.

En esta situación, no nos queda más remedio que definir una clase '*Association*' por cada rol necesario y dos atributos que gestionen cada enlace en un sentido. Siguiendo la propuesta de nombres del apartado 7.2 "*Convención de nombres*" se puede distinguir perfectamente qué clase interna de '*Association*' y su atributo correspondiente hay que usar en cada momento. Además, hay que tener cuidado de poner las constantes en las que se define la multiplicidad de cada clase correctamente, es decir, en el rol correspondiente.

Esta solución puede resultar bastante compleja para el usuario de la clase '*Source*', que tendrá que tener bien claro en cada momento qué papel juega la clase y en función de ello, establecer y acceder a sus enlaces a través de la clase '*Association*' adecuada.

12 ASOCIACIONES *TODO / PARTE*

Normalmente las asociaciones representan relaciones estructurales entre clases iguales. En *UML* se distingue además como un tipo especial de asociación la relación *todo / parte*. Dicha relación se considera agregación cuando la parte puede existir independientemente del todo y como composición cuando la parte está fuertemente ligada al todo y, por tanto, el todo es el único con capacidad de instanciar a la parte y también de destruirla [2] [3]. Aunque esto sea un caso específico de asociación, sí se puede distinguir cierto significado distinto, por lo que *UML* especifica las agregaciones y composiciones con una notación especial.

Vamos a ver ahora cómo implementar, si es que es posible, el control de este tipo especial de asociaciones. No obstante, no haremos referencia a la convención de nombres que se va a emplear puesto que ésta, se puede encontrar en el apartado 7.2: "*Convención de nombres*".

12.1 Agregación

En la Figura 12.1 se puede ver la representación en *UML* de una agregación.

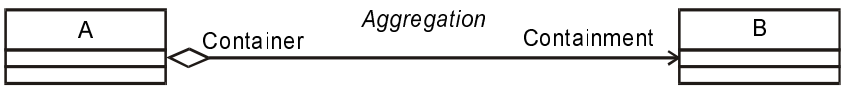

Figura 12.1: Agregación.

En el fondo, la agregación es sólo una diferenciación semántica más específica de la asociación en sí, en el que el su nombre suele ser algún sinónimo de "tiene", "es miembro de", etc. Por tanto, la agregación se va a implementar del mismo modo en el que venimos implementando las asociaciones hasta ahora. El único asunto en la agregación que no se puede solventar en *Java* es que se supone que el contenedor es "más importante" que el contenido, y esto es algo que va a tener que recordar el usuario de las clases que participan en la agregación.

12.2 Composición

La composición exige una ligadura entre la parte contenedora y la contenida tan fuerte que hace totalmente dependiente el contenido respecto al contenedor. De este modo, la composición tendrá siempre una multiplicidad mínima de uno en el lado del contenedor si es navegable en ambos sentidos. En la Figura 12.2 puede verse la representación en *UML* de este tipo de asociación.

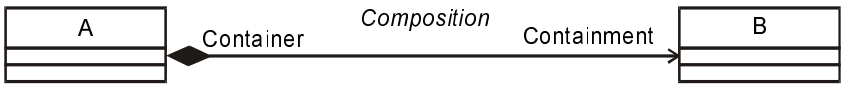

Figura 12.2: Composición.

Un análisis a fondo de la composición nos dice que lo que expresa es que las partes están contenidas físicamente en el todo. *Java* siempre funciona con valores por referencia por lo que parece que la composición no se puede representar adecuadamente [1]. No obstante, eso no quita para que no se pueda representar como asociación y se le pida al usuario del código de las clases que participan en una composición que tenga en cuenta que se está trabajando con esta semántica especial. Así, este usuario tendrá que ser consciente de que sólo puede crear instancias '*B*' desde '*A*' y que cuando se destruya un objeto '*A*', habrá que eliminar también todas sus partes '*B*'. Además, sería conveniente que sólo a través de '*A*' se pueda tener acceso a las instancias '*B*'.

13 VISIBILIDAD EN LAS ASOCIACIONES

Lamentablemente, no parece que se pueda resolver bien el tema de la visibilidad en las asociaciones, a no ser que se realice con relativa complejidad. No obstante, veremos que sí se puede llegar a una solución suficientemente satisfactoria. El problema radica en que, como veremos, no hay una relación directa (aunque sí parecida) entre la visibilidad definida en los lenguajes orientados a objetos para los métodos, atributos y clases, y la semántica de la visibilidad en una asociación *UML*. Concretamente, el problema lo vamos a tener en las asociaciones bidireccionales.

Primero, vamos a intentar entender qué significa cuando decimos que un extremo de una asociación tiene visibilidad privada, protegida, pública o de paquete. De "*Unified Modeling Language Specification*" en su versión 1.4 [3], la visibilidad en los roles especifica quiénes pueden tener acceso a la asociación en el sentido que va desde la clase '*Source*' a la clase con el nombre de rol dado del extremo opuesto. "*A menos que se diga lo contrario, la visibilidad de un rol es pública. La visibilidad privada indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación; la visibilidad protegida indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación, excepto los hijos del otro extremo*" ([2], página 127) .

De este modo, cuando un extremo '*Target*' de una asociación tenga visibilidad pública, entendemos que cualquier instancia, sea del tipo que sea, si puede acceder a la clase '*Source*', podrá también acceder, establecer y eliminar los enlaces de la asociación con la clase '*Target*'. Por el contrario, si la visibilidad es privada, sólo la instancia de la clase '*Source*' podrá manipular y acceder a los enlaces de tipo '*Target*'. La visibilidad protegida es igual que la privada, salvo que a las clases herederas de '*Source*' también se les permite acceder a los enlaces. Por último la visibilidad de paquete es como la pública, pero restringida a las clases definidas dentro del mismo paquete que la clase '*Source*'. De ahora en adelante, nos centraremos principalmente en la visibilidad pública y privada puesto que las otras dos, tal y como se ha visto, son casos similares con más o menos restricciones. En líneas generales, todo lo que se diga para la visibilidad privada podrá aplicarse a la protegida y lo que se diga para la pública, se podrá aplicar a la de paquete.

13.1 Visibilidad en las asociaciones unidireccionales

Con la solución que se ha planteado para la implementación de asociaciones en este proyecto, no existe ninguna dificultad para establecer la visibilidad de una asociación unidireccional según las especificaciones de diseño. Debido a que en las asociaciones unidireccionales sólo hay que tener en cuenta la dirección '*Source*' a '*Target*' (y no al revés), la visibilidad se ha de establecer en la clase '*Source*', y concretamente sobre el atributo de tipo '*Association*' definido para contener los enlaces ('*targetRole*'). Basta con establecer dicho atributo según la visibilidad del extremo; pues en este tipo de asociaciones, sí hay relación directa entre la visibilidad de la asociación y la visibilidad de los atributos que, en nuestro caso, tiene el lenguaje *Java*.

Así, si el extremo de la asociación es público, privado o protegido, haremos el atributo público (*public*), privado (*private*) o protegido (*protected*), respectivamente. Si, por el contrario, la visibilidad está definida como de paquete, el atributo habrá que declararlo *friendly*, es decir, no hará falta especificar ningún valor de visibilidad puesto que, en *Java* esa es la visibilidad por defecto. En resumen, para establecer la visibilidad en las plantillas, basta con poner el valor adecuado en el campo '*%visibility*'.

Observando más detenidamente las plantillas de código presentados en el capítulo 9, "*Implementación de asociaciones unidireccionales*", se puede apreciar que en realidad el campo '*%visibility*' no sólo se ha establecido en la clase '*Association*' y en el atributo '*targetRole*' sino también en los métodos de dicha clase. Esto se ha hecho así para conseguir completa correctitud en el código y marcar de algún modo qué es accesible y qué no por los objetos que intenten acceder a la clase '*Association*'.

Vemos de este modo que se puede establecer una relación directa entre la visibilidad de una asociación navegable en un solo sentido y la solución presentada. Por tanto, el paso de la visibilidad de una asociación unidireccional en el diseño a la visibilidad en la implementación es inmediato y sencillo.

13.2 El problema de la visibilidad en las asociaciones bidireccionales

En las asociaciones bidireccionales, si hacemos lo mismo, nos encontramos con una paradoja. El problema estriba en que hay que establecer los enlaces siempre en ambos sentidos. Vamos a suponer que realizamos lo mismo que en el apartado anterior y establecemos la visibilidad en la clase '*Association*', en sus métodos y en el atributo '*targetRole*' según sea la visibilidad en el otro extremo de la asociación.

Supongamos que tenemos una asociación de visibilidad pública – privada como el de la Figura 13.1. En ella se modela la relación que tienen los bancos con sus clientes y se supone que un banco nunca va a revelar la identidad de sus clientes por lo que la visibilidad en el extremo de la asociación ligado a la clase '*Cliente*' es privada.

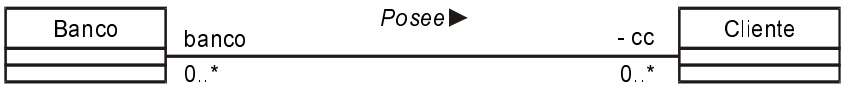

Figura 13.1: Los bancos poseen clientes.

En esta situación y aplicando la visibilidad al código, ocurrirá que el acceso a la clase '*Association*' de '*Banco*' y sus métodos será privado así como el acceso al atributo de tipo '*Association*' que gestiona los enlaces de '*Banco*' a '*Cliente*' que será llamado '*cc*' (cuenta corriente) por ser éste el '*targetRole*' correspondiente. De esta forma, nunca se podrán establecer enlaces desde la clase '*Cliente*' debido a que ésta no podrá establecer el doble enlace porque no puede acceder al atributo '*cc*' de '*Banco*'.

De esta forma, con la solución que se ha aportado, nos encontramos que en las asociaciones de visibilidad pública en un extremo y privada en el otro, sólo vamos a poder añadir y eliminar enlaces desde el lado de visibilidad pública.

Supongamos ahora un modelado del mismo caso pero con una restricción un tanto mayor. En él, tampoco los clientes van a revelar la identidad de sus bancos y por tanto, el extremo '*Banco*' va a tener también visibilidad privada respecto a los clientes. Se trata por consiguiente, de un "secreto" entre ambas clases. En la Figura 13.2 está representada esta asociación.

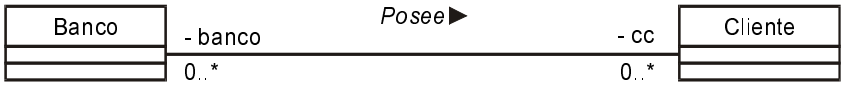

Figura 13.2: Los bancos poseen clientes.

En la implementación de esta asociación los dos atributos que gestionan '*Association*' en cada extremo van a ser privados y nos encontramos con que ya no se puede establecer y eliminar enlaces desde ninguno de los dos extremos. Por tanto, en nuestra solución resulta inviable la gestión de asociaciones con visibilidad privada – privada.

Parece indiscutible que, cuando se pasa del diagrama de clases a la implementación, las clases del modelo de diseño se implementan directamente como una clase OO. Sí se podría discutir, por el contrario, si la implementación de una asociación se debe hacer como se propone en este trabajo, o de otra forma[11]. Ahora bien, los lenguajes OO, por lo general, sólo ofrecen establecimiento de visibilidad en las clases, métodos y atributos, y prácticamente sólo en las cuatro niveles típicos: público, privado, protegido o de paquete. No parece viable, en estas condiciones, poder restringir la visibilidad de determinados elementos a las clases que participan en la asociación y máxime, cuando una clase puede estar asociada con más de una, exigiéndose así, visibilidades exclusivas según con qué asociación estemos tratando en cada momento.

Una posible solución sería meter las dos clases de una asociación en un mismo paquete, pero esto conlleva grandes complicaciones, puesto que una clase, como se ha dicho, puede tener asociaciones de distinto tipo con más de una clase, pero no puede pertenecer a más de un paquete. Además, no es una idea brillante limitar un paquete a las clases que participan en una asociación. Otra solución pasaría por especializar las clases que participan en una asociación de una clase raíz en la que se definiesen los atributos de enlace y trabajar con visibilidad protegida. El problema de esto es que por lo general los lenguajes OO (por ejemplo *Java*) no permiten trabajar con herencia múltiple, y si se hace que una clase herede de una clase concreta para que se pueda trabajar con una asociación concreta, se impedirá, al mismo tiempo, heredar de una clase padre auténtica o de alguna otra asociación adicional. No obstante, en lenguajes como *C++* (en los que sí se puede hacer uso de la herencia múltiple) podría ser una solución digna de estudio. Las interfaces de *Java* tampoco sirven, puesto que no permiten declarar atributos que no sean estáticos y finales debido a que sólo definen interfaz y no implementación.

Vista la imposibilidad de restringir la visibilidad de ciertos atributos y métodos a un conjunto cerrado de clases, queda demostrada la incapacidad de traducir de forma directa la visibilidad de las asociaciones a la visibilidad que ofrece el lenguaje *Java* en sus elementos. Por tanto, si quisiéramos lograr el funcionamiento de la visibilidad de las asociaciones, necesitaríamos que los métodos de control de éstas fuesen bastante más complejos, puesto que deberían ser de acceso público (o al menos de paquete) para que los extremos de la asociación pudiesen acceder a ellos; pero, internamente tendrían que comprobar, según la visibilidad del final de la asociación, si la instancia que accediese debería permitirse o no. Dicha comprobación no es trivial, puesto que los métodos no tienen forma de conocer las instancias invocadoras y el pasarlas como parámetro de entrada se presta a engaños y suplantaciones.

[11] Véase el capítulo 6: "*Ideas alternativas para implementar asociaciones*".

13.2.1 Una solución funcional

El problema de la visibilidad que estamos teniendo está en la necesidad de establecer un doble enlace cuando no se puede acceder por lo menos a uno de los extremos de la asociación. Por tanto, podría hacerse que las clases gestoras de las asociaciones fuesen siempre públicas así como sus instancias. No obstante, los métodos se declararían según fuesen los requisitos de visibilidad. En el establecimiento de doble enlace o borrado, en lugar de invocar a los métodos homólogos en el otro extremo, se invocarían a unos métodos públicos que sólo establecerían (o eliminarían) enlaces en un sentido. Dichos métodos, primero comprobarían si realmente habrían sido invocados desde el otro extremo de la asociación (en lugar de '*test*' preguntarían si han sido invocados por el extremo opuesto: '*isAddRequired*' o '*isRemoveRequired*') y si esto fuese así, entonces establecerían el enlace en un sentido. De esta forma, estos métodos públicos, en realidad, sólo funcionarían cuando fuesen invocados desde los extremos opuestos y nunca desde cualquier otra instancia.

Esta solución se ha llegado a probar y funciona perfectamente. Da igual desde que lado se intenten establecer y eliminar enlaces y cual sea la visibilidad de los extremos de la asociación. Ahora bien, en realidad se ha hecho una pequeña trampa y es declarar la clase '*Association*' y el atributo '*targetRole*' como públicos. Desde el punto de vista funcional esto no presenta ningún problema pues se trata de que, con visibilidades privadas, sólo las clases participantes en la asociación sean capaces de establecer y acceder a los enlaces. La trampa está en que el resto de instancias que tienen acceso sobre las clases que participan en la asociación, son capaces de ver esta asociación, si bien, no van a poder hacer nada con ella.

En los anexos, en el apartado "*ANEXO E: Plantillas de* código alternativas" se encuentra las plantillas de código para implementar asociaciones bidireccionales capaces de funcionar con cualquiera de los niveles de visibilidad que define *UML* en las asociaciones.

13.3 Resumen de la implementación de la visibilidad en las asociaciones

La herramienta que se va a implementar va a trasladar de una forma directa la visibilidad de los extremos de la asociación al atributo que contiene los enlaces en las instancias de cada extremo, sustituyendo para ello el campo '*%visibility*' de las plantillas. De este modo, el atributo se declarará como público, privado, protegido o *friendly* según la visibilidad de la clase en el extremo de la asociación sea pública, privada, protegida o de paquete (respectivamente). Esto va a dar lugar a ciertos casos que no se van a poder tratar, o que funcionarán de una forma restringida. En la Tabla 13.1 se resume la solución, entendiéndose esta vez, por clase '*Source*' la de más a la izquierda o más arriba y '*Target*' la de más a la derecha o más abajo en el diseño.

NAVEGABILIDAD	VISIBILIDAD DE LA CLASE '*Target*' RESPECTO A LA CLASE '*Source*'	VISIBILIDAD DE LA CLASE '*Source*' RESPECTO A LA CLASE '*Target*'	FUNCIONAMIENTO
Unidireccional	Pública, privada, protegida o de paquete		Según lo deseado
Bidireccional	Pública o de paquete	Pública o de paquete	Según lo deseado
Bidireccional	Pública o de paquete	Privada o protegida	Según lo esperado salvo que los enlaces sólo se podrán establecer y eliminar desde la clase '*Target*'
Bidireccional	Privada o protegida	Pública o de paquete	Según lo esperado salvo que los enlaces sólo se podrán establecer y eliminar desde la clase '*Source*'
Bidireccional	Privada o protegida	Privada o protegida	No se trata este caso y la visibilidad en la clase '*Source*' se redefine como pública

Tabla 13.1: Cuadro resumen de la implementación de la visibilidad de las asociaciones.

PARTE III: APLICACIÓN DE DEMOSTRACIÓN - *UML_ASSOCIATIONS*

14 INTRODUCCIÓN A *UML_ASSOCIATIONS*

La aplicación *UML_Associations* sirve para demostrar que el código de las cuatro plantillas que hemos presentado en la parte II de este trabajo es correcto y robusto. Para ello, el programa es capaz de trabajar con cualquier tipo de asociación y da al usuario la posibilidad de crear instancias y enlazarlas dinámicamente a su voluntad. En la aplicación se deberá comprobar como se está efectivamente limitado por las restricciones de especificación de las asociaciones.

14.1 Funcionamiento básico de la aplicación

Debido a que el programa en sí no tiene mayor utilidad que la de servir de demostración, no se ha puesto mucho esfuerzo en la interfaz, que es modo texto. Cuando se ejecuta, se muestra un menú en el que se puede seleccionar entre treinta tipos distintos de asociación (los vemos en la Tabla 14.1).

Caso	**Grupo y navegabilidad**	**Multiplicidades**	
		A	**B**
1	Unidireccionales sencillas		0..1
2			1..1
3	Unidireccionales múltiples		0..*
4			1..*
5			2..5
6	Bidireccionales sencillas - sencillas	0..1	0..1
7		0..1	1..1
8		1..1	0..1
9		1..1	1..1
10	Bidireccionales sencillas - múltiples	0..1	0..*
11		0..1	1..*
12		0..1	2..2
13		1..1	0..*
14		1..1	1..*
15		1..1	3..4

16	Bidireccionales múltiples - sencillas	0..*	0..1
17		0..*	1..1
18		1..*	0..1
19		1..*	1..1
20		2..2	0..1
21		3..6	1..1
22	Bidireccionales múltiples - múltiples	0..*	0..*
23		0..*	1..*
24		0..*	0..4
25		1..*	0..*
26		1..*	1..*
27		1..*	3..5
28		2..3	0..*
29		3..3	1..*
30		2..4	1..2

Tabla 14.1: Las treinta asociaciones con las que se puede trabajar en *UML_Associations*.

Cuando en el menú principal se selecciona una asociación específica aparece un menú que permite crear instancias en ambos extremos de ésta y enlazarlas entre sí. También existen opciones adicionales para la eliminación de enlaces, comprobación de estados, conocimiento del número de enlaces, etc. Las opciones existentes cubren todos los métodos que hemos indicado en las variantes de la clase '*Association*' de las cuatro plantillas presentadas. De este modo, el usuario puede realizar en cualquier orden operaciones con los enlaces y comprobar que el código gestor de asociaciones se comporta siempre como se espera.

Las opciones que presentan todas los casos de asociación son similares salvo en pequeños detalles que tienen que ver con cada tipo de asociación concreta. Así, por ejemplo, en una asociación unidireccional nunca se dará la opción de solicitar la creación de un enlace desde la clase '*Target*' a '*Source*'. También, si se observa los treinta casos de asociaciones con los que se pueden funcionar, se puede apreciar que en las asociaciones bidireccionales no se habla de clases fuente y destino sino de clases '*A*' y '*B*'. Esto se debe a que ambos extremos de una asociación bidireccional pueden realizar las funciones de '*Source*' y '*Target*' según las situaciones lo requieran.

Cuando un usuario solicita ver las instancias que tiene creadas y los enlaces asignados, puede encontrarse con que algunos enlaces no son mostrados. Esto se debe a que cuando se solicita conocer los enlaces (mediante el método '*get*') y no se tiene asignado un número suficiente de ellos, se lanza la excepción '*UnsetAssociationException*' y no se permite conocerlos hasta que se solvente el problema.

14.2 Instrucciones de uso

14.2.1 Instalación y ejecución

Para instalar *UML_Associations* basta con copiarlo desde el CD que se adjunta con el proyecto a un directorio del disco duro. Después, para plataformas *Windows* hay que utilizar el *script* '*run.bat*' y para *Linux/Unix* '*run.sh*'. Se deben editar los *scripts* para establecer correctamente las variables de entorno que son:

- '*UML_Associations_HOME*': directorio donde se encuentra instalada la aplicación y
- '*JAVA_HOME*': directorio donde se encuentra el ejecutable de aplicaciones.

Para otras plataformas, el comando que hay que ejecutar es:

```
%JAVA_HOME%/java -jar "%UML_Associations_HOME%/UML_Associations.jar"
```

14.2.2 Uso

Una vez ejecutado *UML_Associations* aparece el menú que vemos en el Cuadro 14.1. Existe la opción de trabajar con cualquier asociación concreta y para ello, basta con introducir por teclado el número de caso.

Pulsando la letra 'R' y después 'intro' se borran todos los datos hasta ese momento introducidos para así, si el usuario lo desea poder volver a empezar de cero. Las opciones 'C' y 'G' permiten cargar un fichero y grabarlo para trabajar en múltiples sesiones. Por último, introduciendo 'S' se sale del programa.

Suponiendo que el usuario selecciona la opción 18 aparece el menú del Cuadro 14.2. Todas las asociaciones presentan el mismo menú por lo que sólo procederemos a explicar el uso de este caso. No obstante, hay que decir que cada caso de asociación tiene sus particularidades según la navegabilidad y multiplicidad que presenta pero en líneas generales, no hay muchas diferencias. En todos los casos, existe la posibilidad de usar todos los métodos que implementan las asociaciones ('*add*', '*remove*', '*get*', etc.) con sus variaciones con distinta sobrecarga.

```
1.- Caso 1: UNIDIRECCIONAL, Source ---> (0..1) Target
2.- Caso 2: UNIDIRECCIONAL, Source ---> (1..1) Target
3.- Caso 3: UNIDIRECCIONAL, Source ---> (0..*) Target
4.- Caso 4: UNIDIRECCIONAL, Source ---> (1..*) Target
5.- Caso 5: UNIDIRECCIONAL, Source ---> (2..5) Target

6.- Caso 6: BIDIRECCIONAL, A (0..1) --- (0..1) B
7.- Caso 7: BIDIRECCIONAL, A (0..1) --- (1..1) B
8.- Caso 8: BIDIRECCIONAL, A (1..1) --- (0..1) B
9.- Caso 9: BIDIRECCIONAL, A (1..1) --- (1..1) B

10.- Caso 10: BIDIRECCIONAL, A (0..1) --- (0..*) B
11.- Caso 11: BIDIRECCIONAL, A (0..1) --- (1..*) B
12.- Caso 12: BIDIRECCIONAL, A (0..1) --- (2..2) B
13.- Caso 13: BIDIRECCIONAL, A (1..1) --- (0..*) B
14.- Caso 14: BIDIRECCIONAL, A (1..1) --- (1..*) B
15.- Caso 15: BIDIRECCIONAL, A (1..1) --- (3..4) B

16.- Caso 16: BIDIRECCIONAL, A (0..*) --- (0..1) B
17.- Caso 17: BIDIRECCIONAL, A (0..*) --- (1..1) B
18.- Caso 18: BIDIRECCIONAL, A (1..*) --- (0..1) B
19.- Caso 19: BIDIRECCIONAL, A (1..*) --- (1..1) B
20.- Caso 20: BIDIRECCIONAL, A (2..2) --- (0..1) B
21.- Caso 21: BIDIRECCIONAL, A (3..6) --- (1..1) B

22.- Caso 22: BIDIRECCIONAL, A (0..*) --- (0..*) B
23.- Caso 23: BIDIRECCIONAL, A (0..*) --- (1..*) B
24.- Caso 24: BIDIRECCIONAL, A (0..*) --- (0..4) B
25.- Caso 25: BIDIRECCIONAL, A (1..*) --- (0..*) B
26.- Caso 26: BIDIRECCIONAL, A (1..*) --- (1..*) B
27.- Caso 27: BIDIRECCIONAL, A (1..*) --- (3..5) B
28.- Caso 28: BIDIRECCIONAL, A (2..3) --- (0..*) B
29.- Caso 29: BIDIRECCIONAL, A (3..3) --- (1..*) B
30.- Caso 30: BIDIRECCIONAL, A (2..4) --- (1..2) B

R.- Reiniciar todos los casos
C.- Cargar fichero
G.- Grabar

S.- Salir
```

Cuadro 14.1: Menú principal de *UML_Associations*.

```
Caso 18: BIDIRECCIONAL, A (1..*) --- (0..1) B

1.- Crear instancia A
2.- Crear instancia B
3.- Establecer enlace/s (set)
4.- Borrar instancia A
5.- Borrar instancia B
6.- Borrar enlace/s (unset)
7.- Comprobar enlace/s (test)
8.- Comprobar si un objeto cumple con las restricciones de la asociación (isValid)
9.- Conocer el número de enlaces de un objeto (numberOfLinks)

V.- Ver enlaces (get)

A.- Atrás
```

Cuadro 14.2: Menú del caso de asociación 18.

Con las opciones '1' y '2' se permite crear instancias de tipo '*A*' y '*B*'. Cuando se crean instancias, es necesario asignarles un nombre que servirá de identificativo para futuras referencias que tenga que hacer el usuario.

La opción '3' establece enlaces. Si se selecciona, se pide el nombre de las instancias '*A*' y '*B*' que se quieren enlazar. El caso 18 que estamos mostrando de ejemplo tiene multiplicidad mayor que uno en la clase '*A*' por lo que se puede establecer más de un enlace desde una instancia '*B*' a una instancia '*A*'. Para ello, en este caso, se permite introducir más de una instancia tipo '*A*'. Si sólo se pide establecer un único enlace bidireccional entre '*A*' y '*B*', se da la opción a invocar al método '*add*' de '*A*' o al de '*B*'.

Seleccionando en el menú '5' o '6' se pueden borrar instancias. Para ello basta con introducir el nombre de las instancias que se quieren borrar. Al borrar instancias, se invoca al método '*unset()*' para así, eliminar todos los enlaces que previamente se tenían establecidos.

La opción '6' es la opuesta a la '3' y elimina enlaces. Las '7', '8' y '9' sirven para comprobar el estado de los enlaces.

Cuando se elige ver los enlaces (opción 'V') se muestran tres apartados :

1. enlaces existentes,
2. instancias tipo '*A*' que no contienen ningún enlace e
3. instancias tipo '*B*' que no contienen ningún enlace.

Puede ocurrir que una instancia no cumpla con las restricciones de multiplicidad y pese a que tenga enlaces, éstos no se muestren al no existir el número suficiente de ellos. Esto ocurre porque el primer apartado (enlaces existentes) se obtiene mediante la invocación del método '*get*' y como no existe un número suficiente de enlaces, se lanza la excepción '*UnsetAssociation-Exception*'.

Por último, con la opción 'A' (atrás) se puede regresar al menú principal y seleccionar otra asociación con la que trabajar. Si se vuelve a una asociación con la que se ha hecho cosas anteriormente, se podrá comprobar que aún existen las instancias creadas así como los enlaces establecidos.

15 DISEÑO DE LA APLICACIÓN

En la Figura 15.1 se muestra el diagrama de clases de la aplicación demo *UML_Associations*.

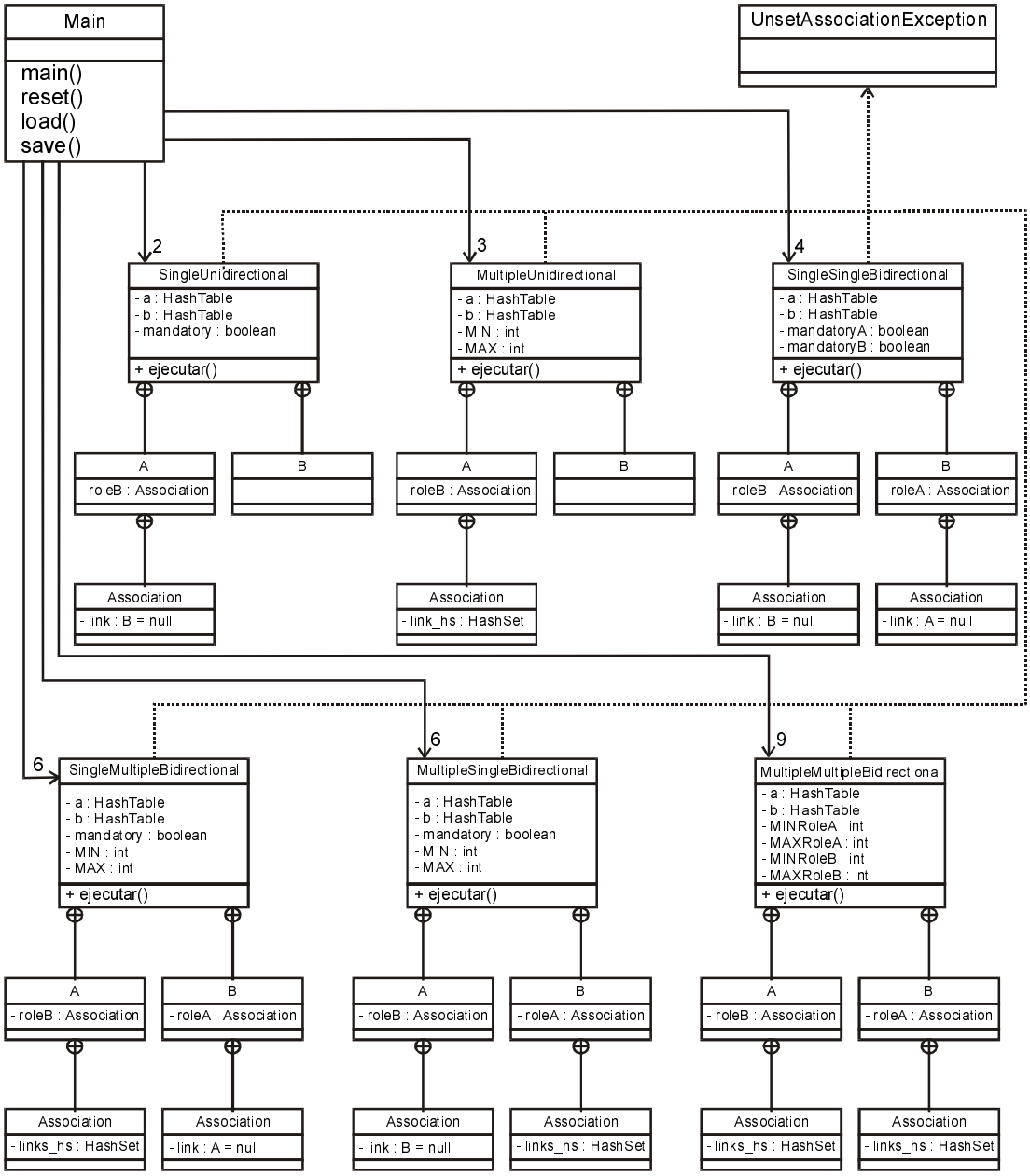

Figura 15.1: Diagrama de clases de la aplicación *UML_Associations*.

15.1.1 '*Main*'

La clase '*Main*' contiene el programa principal. En ella se almacena una instancia de cada uno de los treinta tipos de asociación y se puede trabajar con los treinta casos que hemos visto.

Cuando se ejecuta la clase '*Main*' se presenta un menú que da la opción de elegir un caso de asociación con el que trabajar. También se da la posibilidad de reiniciar todos los casos (borrar las asociaciones que contengan) y de cargar o grabar un fichero para, así, poder trabajar en múltiples sesiones.

15.1.2 '*UnsetAssociationException*'

Esta clase se utiliza cuando es necesario lanzar la excepción de asociación no establecida. Su código es exactamente el mismo que el presentado en el capítulo 8 "*Excepción de asociación no establecida*".

15.1.3 Resto de clases

Las seis clases que quedan por explicar son las que gestionan los distintos casos de asociación. Según el tipo de caso, se utilizará una clase u otra:

- '*SingleUnidirectional*': para contener los casos de asociaciones unidireccionales sencillas, es decir, los dos primeros de la Tabla 14.1.
- '*MultipleUnidirectional*': asociaciones unidireccionales múltiples; casos del tres al cinco del mismo cuadro.
- '*SingleSingleBidirectional*': asociaciones bidireccionales sencillas – sencillas; casos del seis al nueve.
- '*SingleMultipleBidirectional*': asociaciones bidireccionales sencillas – múltiples; casos del 10 al 15.
- '*MultipleSingleBidirectional*': asociaciones bidireccionales múltiples – sencillas; casos del 16 al 21.
- '*MultipleMultipleBidirectional*' : asociaciones bidireccionales múltiples – múltiples; casos del 22 al 30.

El manejo de estas clases es similar en todas las situaciones aunque cambian ciertos aspectos según la navegabilidad y multiplicidad que trate cada una.

Contienen dos clases internas que implementan cada extremo de una asociación binaria. En las asociaciones unidireccionales dichas clases son '*Source*' y '*Target*' mientras que en las bidireccionales son '*A*' y '*B*' puesto que en ese caso, el papel de fuente o destino cambia según se haga la interpretación de los enlaces. El contenido de estas clases se corresponde con el mostrado en las plantillas de implementación de las asociaciones vistas a lo largo de los capítulos 9 y 10 y que se pueden encontrar todas juntas en el "*ANEXO D: Plantillas empleadas en la generación de código*".

Estas clases que utiliza '*Main*' para la comprobación del código de asociaciones reciben en el constructor los parámetros necesarios para la especificación de las restricciones de multiplicidad. Por tanto, las clases internas '*A*' y '*B*' (o '*Source*' y '*Target*') no funcionan con las constantes '*mandatory*', '*MIN*' y '*MAX*' sino que toman dichos valores de la clase que las contiene. De esta

forma, se consigue que dentro del mismo esquema de plantilla, se pueda trabajar con distintas multiplicidades y, por consiguiente, que cada una de las seis clases que aquí estamos explicando sirvan para trabajar con más de una asociación. Con sólo estas seis clases se pueden gestionar los treinta casos de asociación que tenemos.

Todas estas clases tienen definido el método '*ejecutar*'. Dicho método presenta un menú que da la opción de crear instancias '*A*' o '*B*' y enlazarlas o eliminar instancias y enlaces. Al crear enlaces, se debe poder comprobar que cuando se intenta exceder la multiplicidad máxima no se permite el establecimiento de ellos. También se da la opción de comprobar la existencia de enlaces ('*test*') y el cumplimiento con las restricciones de multiplicidad mínima ('*isValid*') y de conocer el número de enlaces que una instancia posee ('*numberOfLinks*').

Por último, falta por decir que las instancias de tipo '*A*' y '*B*' que crea el usuario dinámicamente se almacenan en tablas *hash* (mediante la clase '*HashTable*' de *Java*). Existen declaradas dos atributos que son tablas *hash* en cada una de estas seis clases: una para almacenar las instancias tipo '*A*' y otra las instancias tipo '*B*'. En dichas tablas, se utilizan como palabras claves (para poder identificar las instancias) los nombres que el usuario asigna a los objetos que crea.

PARTE IV: HERRAMIENTA PARA LA GENERACIÓN DE CÓDIGO - *JUMLA*

16 INTRODUCCIÓN A *JUMLA*

JUMLA son las siglas de "***J**ava code generator for **U**nified **M**odeling **L**anguage **A**ssociations*". Se trata de una aplicación que genera el código en *Java* capaz de gestionar las asociaciones *UML* especificadas en un fichero *XMI* ("*XML Metadata Interchange Specification*") de entrada, basándose en el método de implementación de asociaciones que se ha explicado en la parte II de este proyecto. Para ello, extrae de un fichero *XMI* introducido por el usuario las clases y asociaciones especificadas. Una vez reconocidas las asociaciones, el usuario puede seleccionar las asociaciones de las cuales quiere obtener el código en *Java*. El código se obtiene en un directorio que se selecciona a través de un cuadro de diálogo.

"XMI es un modelo de intercambio de información con el que se pretende dar a los desarrolladores que trabajan con tecnología de objetos la posibilidad de intercambiar datos a través de Internet de una forma estandarizada, aportando consistencia y compatibilidad entre las aplicaciones creadas en entornos cooperativos." [14]. La propuesta de *XMI* cubre la transferencia de modelos *UML* y meta modelos *MOF*, y está basada en el estándar *XML*.

Cuando se pensó el formato a partir del cual *JUMLA* podría extraer las asociaciones de un diagrama de clases pronto surgió la idea de hacerlo a partir de *XMI*. El motivo es evidente: *XMI* es abierto y son muchas las aplicaciones de diseño *UML* capaces de interpretarlo. Ahora bien, *XMI* es muy complejo y presenta distintas variantes para almacenar un mismo artefacto *UML* (como p.e. las asociaciones). Nos hemos encontrado que es muy frecuente que las aplicaciones capaces de interpretar y generar *XMI* muchas veces no pueden leer ficheros *XMI* escritos por otras herramientas. Incluso hay herramientas (como *Together* 5.5) que no interpretan bien el código *XMI* generado por ellas mismas. Esto nos ha llevado a tener que limitar en gran medida las posibilidades de interpretación de *JUMLA* de los modelos *XMI* de entrada.

16.1 Condiciones de validez del modelo

Para que *JUMLA* sea capaz de interpretar el modelo de entrada se han de cumplir las siguientes condiciones:

- El fichero *XMI* tiene que haber sido generado por la herramienta *MagicDraw UML*[12] en su versión 5.0 o por una herramienta que escriba código *XMI* similar. Por tanto, la versión de *XMI* utilizada es la 1.0, y de *UML* la 1.3 (por ser las versiones que utiliza *MagicDraw* 5.0).
- Los ficheros *XMI* de entrada pueden estar comprimidos en formato *ZIP*.
- Las extensiones de ficheros que se reconocen son '*.xml*', '*.xmi*' y '*.zip*'.
- En las asociaciones unidireccionales la clase '*Source*' no puede tener ninguna restricción de multiplicidad y debe ser, por tanto, del tipo *many* no restringida (0..*). Si

[12] *MagicDraw UML* es una herramienta escrita en *Java* para el diseño *UML*. Es una herramienta bastante popular capaz de funcionar en cualquier plataforma que tenga instalado *Java*™ *2 Runtime Environment*. Se puede obtener una versión demo con un tiempo limitado de uso en la *URL* http://www.magicdraw.com.

se especifica restricción de multiplicidad en '*Source*' en una asociación unidireccional, *JUMLA* da la opción de descartar la asociación entera o de ignorar la restricción y tratarla como si fuese (0..*).

- Si existen dos o más asociaciones entre las mismas dos clases, se requiere que las asociaciones no sean anónimas.
- Si una asociación es reflexiva, se requiere que se especifique nombre de rol en ambos extremos de ella.
- La generación de las clases en *Java* se hace por paquetes, siendo el paquete raíz el nombre del modelo de entrada. El nombre del modelo de entrada, de los paquetes en él especificados, así como de las clases, han de cumplir las reglas de nombrado de *Java*. Es decir, tienen que ser caracteres alfanuméricos (no se contemplan los caracteres ortográficos particulares de una lengua: acentos, letra "eñe", etc.) y sólo se permite como símbolo adicional el carácter de subrayado: '_'.
- *JUMLA* no interpreta del modelo de entrada los artefactos *UML* que no se refieren directamente a una asociación. Así, pese a que una clase tenga atributos y métodos definidos o sea heredera de otra, estas características no se tienen en cuenta y son ignoradas por la herramienta en la generación de código.

16.2 Condiciones de validez del código generado

Una vez reconocido el modelo de entrada y las asociaciones que éste contiene, *JUMLA* genera el código de las asociaciones seleccionadas por el usuario. El código generado automáticamente requiere que las siguientes condiciones sean tenidas en cuenta:

- Si al generar el código de una clase *JUMLA* se encuentra con que ya existe dicha clase, se intenta añadir sólo el código referente a la asociación. En esta situación, se garantiza la perfecta compilación de la clase si y sólo si ésta compilaba perfectamente antes de la adición del código gestor de la asociación. En resumen, se garantiza que el código que inserta *JUMLA* es correcto pero no el que existía previamente.
- El código generado para las asociaciones bidireccionales con visibilidad pública o de paquete en un extremo y privada o protegida en otro tienen la particularidad de que sólo permite establecimiento y eliminación de enlaces desde el extremo privado o protegido. Se da la opción de no generar el código correspondiente a las asociaciones que así estén definidas.
- No se implementan las asociaciones bidireccionales con visibilidad privada (o protegida) – privada (o protegida).
- El código de una asociación (clase interna '*Association*' y su atributo) se inserta siempre al final de la clase y se puede reconocer dónde comienza porque lleva la siguiente etiqueta:

  ```
  /** !Código generado por JUMLA */
  ```

- Asimismo, el final del código generado por la herramienta correspondiente a cada asociación se puede reconocer por la etiqueta:

  ```
  /** !Fin de código generado por JUMLA */
  ```

- Al insertar una asociación en una clase que existía previamente, *JUMLA* intenta reconocer si se trata de una nueva asociación que hay que añadir al código de la clase o si se trata de un cambio de diseño. Para ello, si se encuentra con el código de una asociación con el mismo nombre de '*targetRole*', lo que se hará es sustituir esta asociación por la nueva.

16.3 Ingeniería directa e inversa

JUMLA, en esta primera versión, sólo funciona en una dirección, es decir, es capaz de escribir código *Java* para la implementación de asociaciones *UML*, pero no es capaz de obtener a partir de código *Java* las asociaciones *UML* que existen entre las posibles clases de entrada.

No obstante, el código que se genera sí está pensado para podérsele aplicar ingeniería inversa de una forma sencilla y cómoda. Cada asociación implementada dentro de una clase está perfectamente delimitada por las etiquetas de "*código generado por JUMLA*" y "*fin de código generado por JUMLA*". Asimismo, tal y como se ha explicado en el capítulo 7.2 "*Convención de nombres*", los nombres de las clases que implementan las asociaciones y sus atributos siguen unas reglas que hacen fácilmente identificable el tipo de asociación y las clases que involucra, así como las restricciones de diseño.

Por tanto, si bien no se ha desarrollado ninguna opción de ingeniería inversa, el análisis del código por un analista programador es sencillo y rápido, y futuras versiones de *JUMLA* sí podrían desarrollar esta funcionalidad fácilmente.

17 MANUAL DE USUARIO

17.1 Requerimientos del sistema

Para poder utilizar la herramienta *JUMLA* no se necesita disponer de excesivos recursos. Estos son los requisitos mínimos:

- Cualquier plataforma capaz de ejecutar (y que previamente tenga instalado) *Java™ 2 Runtime Environment (Standard Edition Versión 1.3.0_02)* o superior y que disponga de un sistema gráfico de ventanas.
- Al menos 2,5 MB de espacio libre en el disco duro.
- Al menos 50 KB de memoria *RAM* libre.

17.2 Instalación y configuración

JUMLA se puede copiar directamente del CD que se adjunta con este proyecto. Para ello se debe acceder al directorio '*JUMLA*' y copiarlo en el disco duro. También se puede distribuir en formato *ZIP* en cuyo caso habrá que descomprimirlo en una carpeta.

17.2.1 Distribución de los ficheros de la herramienta

Una vez instalado *JUMLA* en el disco duro, debe existir la jerarquía de directorios y ficheros que se muestra en el Cuadro 17.1.

En el directorio '*help*' se encuentra la ayuda de *JUMLA* en formato *HTML*. El fichero '*index.html*' es la página principal. En '*lib*' están los archivos *jar*[13] de la aplicación. La aplicación propiamente dicha reside en el archivo *jar* '*JUMLA.jar*' y la clase principal que hay que ejecutar es '*Application*'. '*XMIFramework.jar*' es una *API* de *IBM* para crear e interpretar ficheros *XMI*[14]. La versión de este *jar* es la 1.0 y es capaz de interpretar *XMI* 1.0 o *XMI* 1.1. Otro archivo *jar* es '*xerces.jar*' que contiene un analizador *XML*. El paquete *Xerces* que se distribuye con *JUMLA* contiene la versión 2.0.1 y es utilizado por *XMIFramework* y no directamente por *JUMLA*.

En el directorio '*templates*' están las plantillas de código que se utilizan para generar el código de las asociaciones. '*EmptyClass.java~*' se utiliza cuando hay que generar una clase vacía que simplemente contenga el nombre del paquete y declare la clase. Esta plantilla se utiliza para las clases '*Target*' de las asociaciones unidireccionales puesto que no deben llevar ningún código relacionado con la asociación. Los otros cuatro archivos del directorio '*templates*' son las plantillas

---

[13] *jar = "Java archive"*. La herramienta *jar* de *Java* permite introducir en un único archivo varias clases precompiladas y archivos adicionales y accesorios. *Java* es capaz de interpretar dichos ficheros consiguiéndose así el empaquetamiento de clases y archivos auxiliares de las aplicaciones así como compresión de datos.

[14] Se puede obtener la librería *XMIFramework* completa con su documentación en el CD que se adjunta con este documento y en http://www.alphaworks.ibm.com/tech/xmiframework.

de código con las que venimos trabajando a lo largo de este trabajo y su código se puede ver en el "*ANEXO D: Plantillas empleadas en la generación de código*". El directorio '*alternative templates*' contiene plantillas de código para poder implementar asociaciones bidireccionales en las que no haya ninguna restricción respecto a la visibilidad. Su código puede verse en el "*ANEXO E: Plantillas de código alternativas*". *JUMLA* no trabaja con estas plantillas pero se ponen a disposición del usuario por si las necesita.

El archivo '*JUMLA.config*' guarda la configuración del usuario y, en caso de no existir, *JUMLA* se encarga de crear uno siempre y cuando así lo desee el usuario. En el fichero '*jumla.log*' se almacenan las incidencias que ocurren (fechas de inicio y fin de sesiones de *JUMLA*, asociaciones cargadas y generadas, errores producidos, etc.). En un principio este fichero no existe pero si el usuario así lo especifica en las opciones de configuración, se genera automáticamente. Existe también la posibilidad de especificar otro fichero de seguimiento de incidencias. '*Readme.html*' es un enlace a la ayuda de *JUMLA* para poder tener una breve introducción a la herramienta, instrucciones de uso e instalación, etc.

Por último, para ejecutar la herramienta hay que utilizar el *script* '*run.bat*' en plataformas *Windows* y '*run.sh*' en plataformas *Unix/Linux*.

Cuadro 17.1: Distribución de los ficheros que componen la aplicación *JUMLA*.

17.2.2 Ejecución de la aplicación

17.2.2.1 Plataformas *Windows*: '*run.bat*'

En plataformas *Windows*, *JUMLA* se tiene que ejecutar mediante el *script* '*run.bat*' que necesita ser modificado previamente para que se adapte a la configuración del ordenador en el que se encuentra instalado. Veamos el contenido de este *script* en el Cuadro 17.2.

```
@echo off

REM Modifique las variables JUMLA_HOME y JAVA_HOME de acuerdo a su configuración

REM Directorio donde se aloja JUMLA
SET JUMLA_HOME=C:\JUMLA

REM Directorio dónde se encuentra java
SET JAVA_HOME=C:\jdk1.3\bin

%JAVA_HOME%\java.exe -classpath
"%JUMLA_HOME%/lib/JUMLA.jar;%JUMLA_HOME%/lib/XMIFramework.jar;%JUMLA_HOME%/lib/xe
rces.jar" jumla.Application %1
```

Cuadro 17.2: Script '*run.bat*'.

Sólo hay que modificar las dos variables que se resaltan en negrita. En '*JUMLA_HOME*' se ha de especificar el directorio donde se encuentra instalado *JUMLA* y en '*JAVA_HOME*' el directorio completo en el que se encuentra el ejecutable de aplicaciones *Java.*

17.2.2.2 Plataformas *Unix*/*Linux*: '*run.sh*'

En este tipo de plataformas se requiere del *script* '*run.sh*'. El contenido debe ser similar al que mostramos en el Cuadro 17.3 y sólo hace falta entender lo que se resalta con letra en negrita.

Nuevamente hay que editar el *script* y modificarlo para que la variable '*JUMLA_HOME*' apunte al directorio donde está instalado la aplicación y '*JAVA_HOME*' indique el directorio donde se encuentra el ejecutable '*java*'.

```
#!/bin/sh

# Especifica el directorio de JUMLA (donde está instalado)
if [ -z "$JUMLA_HOME" ]; then
JUMLA_HOME=/usr/local/JUMLA
fi


# Directorio donde está instalado el ejecutable java
if [ -z "$JAVA_HOME" ]; then
JAVA_HOME=/usr/bin
fi

if [ -x $JAVA_HOME/java ]; then
$JAVA_HOME/java -classpath
"$JUMLA_HOME/lib/JUMLA.jar;$JUMLA_HOME/lib/XMIFramework.jar;$JUMLA_HOME/lib/xerce
s.jar" jumla.Application $1
else
echo "ERROR"
echo "No se encuentra el ejecutable de java " $JAVA_HOME/java
echo "Establezca la variable de entorno JAVA_HOME o modifíquela en este script"
$0
fi
```

Cuadro 17.3: Script '*run.sh*'.

17.2.2.3 Otras plataformas

Para ejecutar *JUMLA* en cualquier otra plataforma y suponiendo que la aplicación está instalada tal y como se ha indicado en el apartado 17.2.1 "*Distribución de los ficheros de la herramienta*" hay que escribir lo siguiente en la línea de comandos:

```
%JAVA% -classpath
"%JUMLA_HOME%/lib/JUMLA.jar;%JUMLA_HOME%/lib/XMIFramework.jar;%JUMLA_HOME%/lib/xe
rces.jar" jumla.Application
```

Donde '*%JAVA%*' ha de ser sustituido por el *path* completo del ejecutable de aplicaciones *Java* y '*%JUMLA_HOME%*' por el directorio donde está instalada la herramienta.

17.2.3 Configuración de *JUMLA*

La primera versión de *JUMLA* tiene siete parámetros de configuración que, si bien tienen sus valores por defecto, conviene que el usuario modifique según sus especificaciones. La configuración de la aplicación se almacena en el fichero '*JUMLA.config*'. En caso de que no exista el fichero o no pueda ser leído al ejecutarse la herramienta, se da al usuario la opción de crearlo. Si existe y se quiere modificar, se puede hacer mediante las opciones del menú: "*Archivo*" y "*Opciones*". La configuración se hace a través del cuadro de diálogo que se muestra en la Figura 17.1.

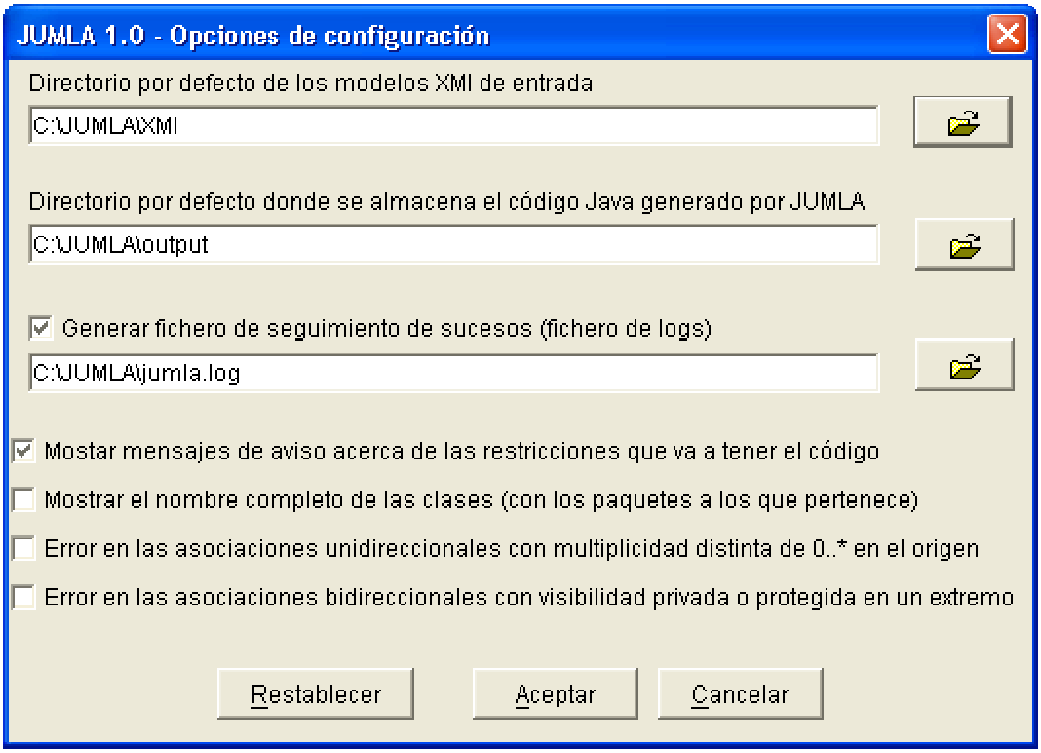

Figura 17.1: Configuración de opciones.

En el primer campo de texto se nos pide introducir el directorio de entrada de los ficheros *XMI* por defecto. Cuando se solicita abrir un fichero *XMI*, la herramienta muestra los modelos que se encuentren en dicho directorio puesto que lo lógico es que el usuario tenga un directorio propio donde los almacena y así, se le evita tener que navegar por el árbol de directorios y puede encontrar sus modelos de una forma rápida.

El siguiente campo de texto solicita la entrada del directorio por defecto en el que se almacenan las clases generadas automáticamente. En el tercer y último campo de texto se puede especificar un fichero de seguimiento de incidencias. Este campo va precedido de una casilla de verificación para indicar si se desea o no generar el fichero de *logs*. Los tres campos de texto no se pueden editar directamente y sólo pueden ser modificados a través de un cuadro de diálogo que permite seleccionar directorios o archivos y que se accede a él con los botones que se encuentran a la derecha de estos campos de texto.

Después de los dos campos de directorio de entrada y de salida existen cuatro casillas de verificación. La primera da la opción de mostrar mensajes de aviso indicando las restricciones de uso que puede tener el código generado.

La segunda permite o no mostrar el nombre completo de las clases cuando se muestran las asociaciones. Si se activa esta casilla, no sólo se nos mostrará el nombre de las clases sino también los paquetes y subpaquetes en los que se encuentra contenida.

La tercera casilla de verificación quizás sea un poco más compleja de entender. Ya hemos dicho que *JUMLA* ignora la multiplicidad establecida en las clases '*Source*' de las asociaciones

unidireccionales, puesto que ésta debe ser siempre (0..*). Si se activa esta casilla de verificación y en un momento dado en la generación de código *JUMLA* se encuentra una asociación unidireccional con multiplicidad distinta de (0..*) en '*Source*', no se genera nada de código relacionado con la asociación y se muestra por pantalla un mensaje de error. Si no se activa, se genera el código sin tener en cuenta la multiplicidad distinta de (0..*).

La última casilla de verificación sirve para rechazar las asociaciones bidireccionales con visibilidad privada o protegida en un extremo y pública o de paquete en el otro. Si no se marca esta casilla, las asociaciones bidireccionales tipo *private* (o *protected*) – *public* (o *package*), o al revés, se generarán con la particularidad de que sólo permitirán el establecimiento y borrado de enlaces desde el extremo con visibilidad *private* (o *protected*).

Una vez configuradas estas siete opciones hay que pulsar el botón "*Aceptar*" para que se almacenen y sean utilizadas. Si se pulsa "*Cancelar*" no se produce ninguna modificación. Por último, "*Restablecer*" sirve para volver a poner los campos tal y como estaban al mostrarse inicialmente el cuadro de diálogo de opciones.

El archivo '*JUMLA.config*' en el que se almacena la configuración tiene el formato que se muestra en el Cuadro 17.4 y las variables pueden ser modificadas directamente a través de un editor de texto siempre y cuando se respete la sintaxis adecuada:

%<nombre_variable>=<valor_variable>[15]

```
Directorio de entrada de ficheros XMI por defecto
%inputDir=C:\JUMLA\XMI

Directorio de generación de código por defecto
%outputDir=C:\JUMLA\XMI

Usar fichero log (true o false)
%useLogFile=true
Fichero de log
%logFile=jumla.log

Mostar mensajes de aviso acerca de las restricciones que va a tener el código
(true o false)
%showWarningMessages=true

Mostrar el nombre completo de las clases (paquete incluido) (true o false)
%showWholeClassName=false

Error cuando las asociaciones unidireccionales tengan multiplicidad distinta de
0..* en Source (true o false)
(error on Source Unidirectional Association Multiplicity Restriction)
%errorOnSUAMR=false

Error cuando las asociaciones bidireccionales tengan visibilidad (privada o
protegida) - (pública o de paquete) o al revés (true o false)
%errorOnPrivatePublicBA=false
```

Cuadro 17.4: Formato del fichero de configuración[16].

[15] Sin dejar ningún espacio en blanco y las variables empezando siempre con el carácter '%'.

Para concluir este apartado, falta por decir que la aplicación viene preparada para recibir un parámetro de entrada en el que se especifica un fichero de configuración alternativo a '*JUMLA.config*'. Para ello basta con ejecutarla (bien mediante los *scripts* de ejecución o bien directamente) indicando la ubicación del fichero de configuración que se quiere utilizar. Si se ejecuta *JUMLA* de este modo, al configurar las opciones las modificaciones se harán sobre este fichero.

17.3 Uso de la herramienta

Una vez configurada y ejecutada la herramienta aparece la ventana principal de la aplicación que se muestra en la Figura 17.2. La ventana contiene un área de menú y un área con una barra de botones en la parte de arriba. El área principal de trabajo está dividido en dos zonas (las veremos más adelante). En la parte de abajo hay una barra de progreso y una barra de estado que inicialmente informa de que no hay ningún fichero abierto.

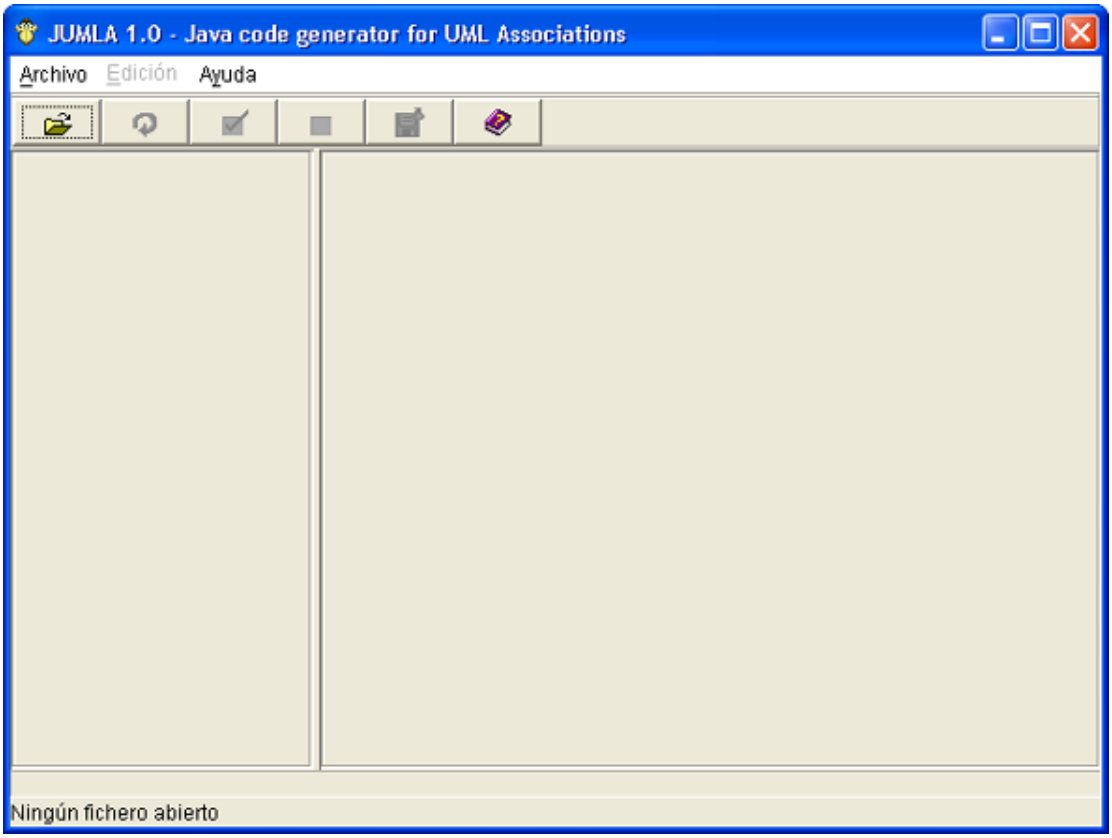

Figura 17.2: Pantalla principal de *JUMLA*.

[16] En plataformas que no sean *Windows* se debe utilizar la notación de *path* de directorio propia de la plataforma (p.e. /home/user/XMI_models).

17.3.1 Carga de un fichero *XMI*

Recién ejecutada la aplicación, prácticamente la única opción que tiene el usuario es la de abrir un fichero (también puede acceder a la configuración de *JUMLA* y a la ayuda). Para ello debe pulsar el botón de "*Abrir Fichero*" o seleccionar en el menú "*Archivo*" y después "*Abrir Fichero*". Aparece un cuadro de diálogo para abrir un fichero *XMI* y por tanto, salvo que se indique lo contrario en el campo "*tipo de fichero*", en principio sólo se muestran los directorios y ficheros con extensión '*.xml*', '*xmi*' y '*.zip*'. Tras seleccionar un fichero, si este no se puede interpretar por *JUMLA* o no se encuentra ninguna asociación *UML* definida se muestra un mensaje de error, y en caso contrario se muestra en la parte izquierda del área de trabajo el árbol de clases y en la parte derecha las asociaciones encontradas.

En la Figura 17.3 puede verse cómo se muestra en la pantalla el árbol de clases y las asociaciones tras abrir el fichero '*ejemplo-asociaciones.xml.zip*'. En el árbol de clases puede verse que todas las clases cuelgan del paquete raíz '*ejemplo-asociaciones*' que se corresponde con el nombre del modelo. Si en el modelo hubiese definidos subpaquetes estos aparecerían también. En la Figura 17.4 se muestra una captura de pantalla de *MagicDraw* del diagrama de clases que se ha guardado en '*ejemplo-asociaciones.xml.zip*'.

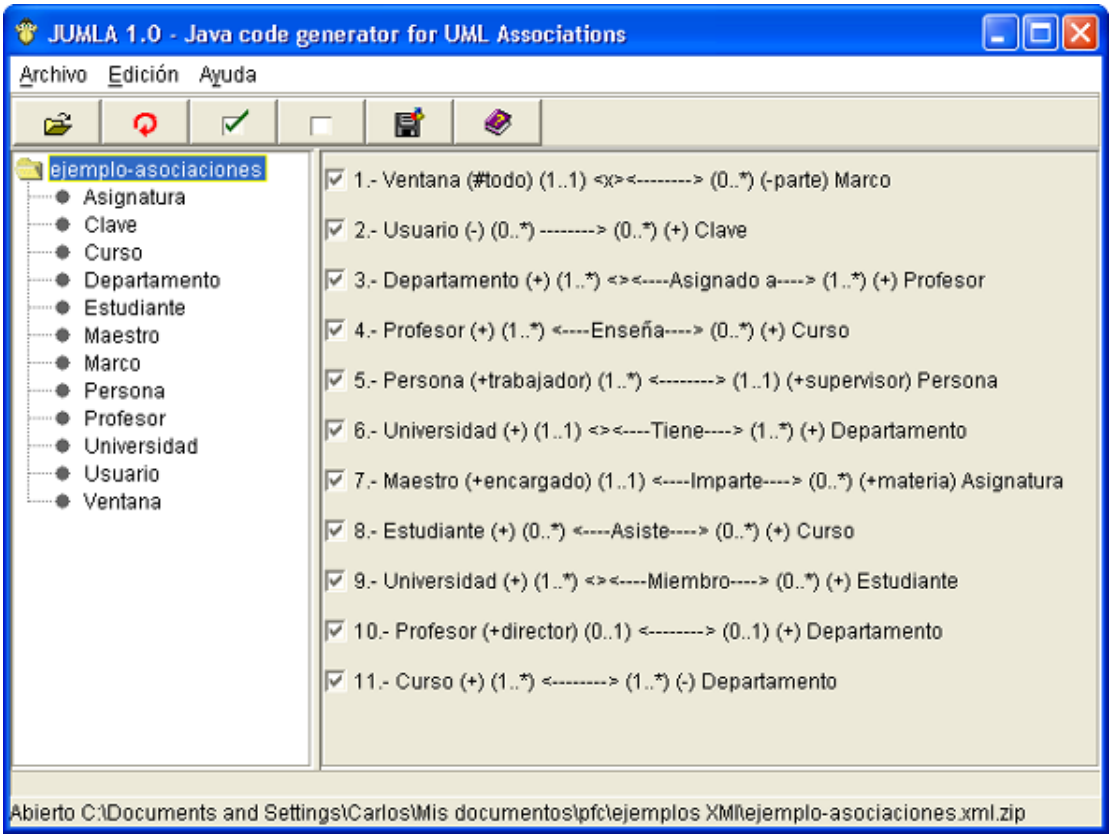

Figura 17.3: *JUMLA* tras la apertura del fichero '*ejemplo-asociaciones.xml.zip*'.

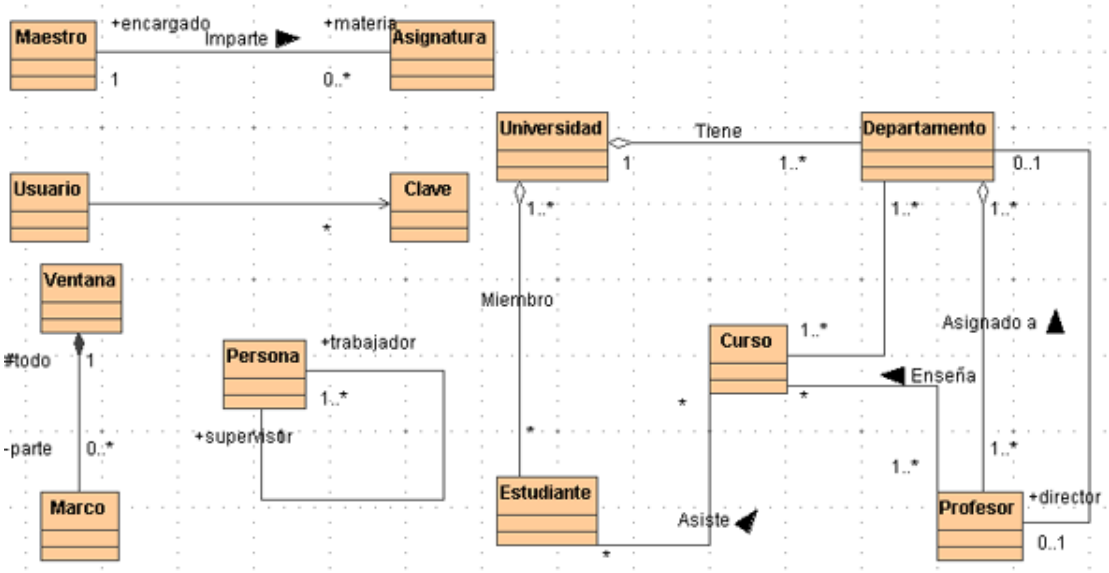

Figura 17.4: Diagrama de clases contenido en el fichero '*ejemplo-asociaciones.xml.zip*'.

17.3.2 Interpretación de las asociaciones, selección y generación de código

Para entender las asociaciones mostradas en la Figura 17.3 y comprobar que son las mismas especificadas en el modelo de entrada mostrado en la Figura 17.4, hay que saber cómo se representan en el área de trabajo de *JUMLA*. Todas las asociaciones se representan según la estructura que se muestra a continuación y en negrita se resaltan los campos que aparecen siempre:

```
<Id> <AClassName> (<visibility><ARoleName>) (<multiplicity>)
                  ----<AssociationName>---
(<multiplicity>) (<visibility><BRoleName>) <BClassName>
```

- **<Id>**: Identificativo de asociación. Es un número que se asigna por orden de aparición de las asociaciones en el fichero *XMI* de entrada.
- **<AClassName>** y **<BClassName>**: nombre de las clases '*A*' y '*B*' [17].
- **<visibility>**: visibilidad. Se representa según los siguientes caracteres:
  - '+': *public*
  - '~': *package*[18]

[17] No hablaremos de clase '*Source*' y '*Target*' sino de clase en el lado izquierdo de la asociación ('*A*') o en el lado derecho ('*B*').

    - '#': *protected*
    - '-': *private*

- <ARoleName> y <BRoleName>: nombre de rol de las clases '*A*' y '*B*' respecto al otro extremo de la asociación.
- **<multiplicity>**: especifica la multiplicidad y existen las siguientes posibilidades:
    - (0..1) Sencilla
    - (1..1) Sencilla *mandatory*
    - (0..*) Múltiple
    - (*m*..*) Múltiple *mandatory* con $m > 0$
    - (*m*..*n*) Múltiple restringida ($m \geq 0$)
- <AssociationName>: nombre de la asociación.

Los diferentes casos de asociación se representan de la siguiente manera:

- --------> Asociación unidireccional
- <-------> Asociación bidireccional
- ------><> agregación unidireccional
- <----><> Agregación bidireccional
- ------><x> Composición unidireccional
- <----><x> Composición bidireccional

Puede que se esté diseñando y modificando el modelo a la vez que se utiliza *JUMLA*. Para eso, si se tiene abierto un fichero *XMI* y posteriormente se producen cambios en él, no hay más que pulsar el botón de "*Volver a abrir fichero*" para volver a cargar el nuevo contenido. Si se prefiere realizar esta opción a través del menú, la opción se encuentra dentro del submenú de "*Archivo*".

Una vez contrastado que las asociaciones mostradas por la aplicación *JUMLA* se corresponden con las especificadas en el modelo de entrada se procede a seleccionar las asociaciones cuyo código *Java* queremos obtener. Todas las asociaciones tienen una casilla de verificación que por defecto está marcada. Si se desea se pueden marcar (o desmarcar) las asociaciones de una en una, pero a modo de ayuda se dispone de dos botones que permiten marcar

[18] Hay que señalar que *MagicDraw* no contempla la posibilidad de trabajar con visibilidad de paquete (incorporada en la versión 1.4 de *UML*) pero *JUMLA* se ha implementado para ser capaz de interpretarla.

todas las asociaciones o ninguna. También se puede marcar (o desmarcar) todas las asociaciones a través de la opción del menú "*Edición*".

El árbol de clases también sirve para la selección de asociaciones. Si se selecciona una clase en el árbol, automáticamente se activan sólo las asociaciones que contienen dicha clase y el resto se desactiva. Asimismo, si se selecciona un paquete o subpaquete, se activan sólo las asociaciones que implican una clase perteneciente a dicho paquete o subpaquete. De este modo, cuando no están activas todas las asociaciones, los botones de "Marcar todos" o "Marcar ninguno" actúan sólo sobre las asociaciones que sí lo están.

Cuando se termine de seleccionar aquellas asociaciones de las cuales se quiere obtener el código *Java* se puede proceder a pulsar el botón de "*Generar código*" o, en el menú, en "*Archivo*" y después en "*Generar código*". Debe aparecer un cuadro de diálogo en el que hay que seleccionar el directorio a partir del cual se desea que se guarden los ficheros de las clases *Java*. En la barra de progreso que hay en la parte inferior de la ventana principal se podrá observar el curso del proceso y el final de él se informará al usuario mediante un cuadro de diálogo. Si se produce cualquier error, el proceso de generación de código se detiene y, según la gravedad del asunto, se podrá continuar si el usuario así lo desea, o se detendrá hasta que se solventen los problemas que provocan el fallo.

18 DISEÑO DE LA APLICACIÓN

En la Figura 18.1 se muestra el paquete '*jumla*' con las clases que se han implementado y por simplicidad y por no repetirse con la siguiente figura no se muestran las relaciones existentes entre ellas. Para proceder a la explicación del diseño interno de *JUMLA* necesitamos ver la Figura 18.2 que contiene el diagrama de clases completo de la aplicación.

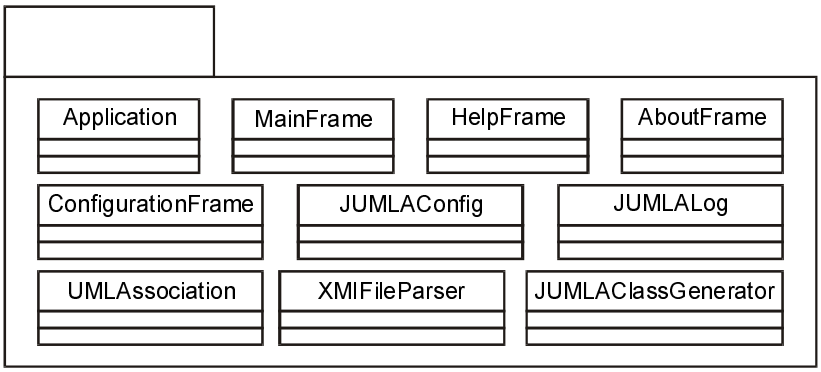

Figura 18.1: Paquete '*jumla*'.

18.1 'Application'

La clase '*Application*' es la clase principal de *JUMLA* por lo que contiene el método '*main*'. Esta clase prácticamente lo único que hace es crear una instancia de tipo '*MainFrame*' y la ejecuta.

18.2 'MainFrame'

La ventana principal del programa y las acciones que en ella se pueden realizar se encuentran implementadas en esta clase. En '*MainFrame*' se especifican el menú, la barra de botones, la barra de estado y el cuadro principal que se divide en un panel con un componente tipo árbol y otro panel donde se visualizan las asociaciones.

Por ser la ventana principal, estamos hablando de la clase central de toda la herramienta. Tal y como se puede ver en el diagrama de clases de la Figura 18.1, a partir de esta clase se crean el resto de cuadros de diálogo y se realizan todas las acciones: apertura de un ficheros *XMI* y su interpretación, generación de código *Java*, etc.

18.3 'UMLAssociation'

Con '*UMLAssociation*' se representan las asociaciones *UML*. Por razones de simplicidad, en el diagrama de clases de *JUMLA* sólo se muestra los atributos pero, obviamente, dispone de métodos para definir las asociaciones y sus restricciones de diseño (métodos tipo '*set*') y para obtener su definición (tipo '*get*' o '*is*').

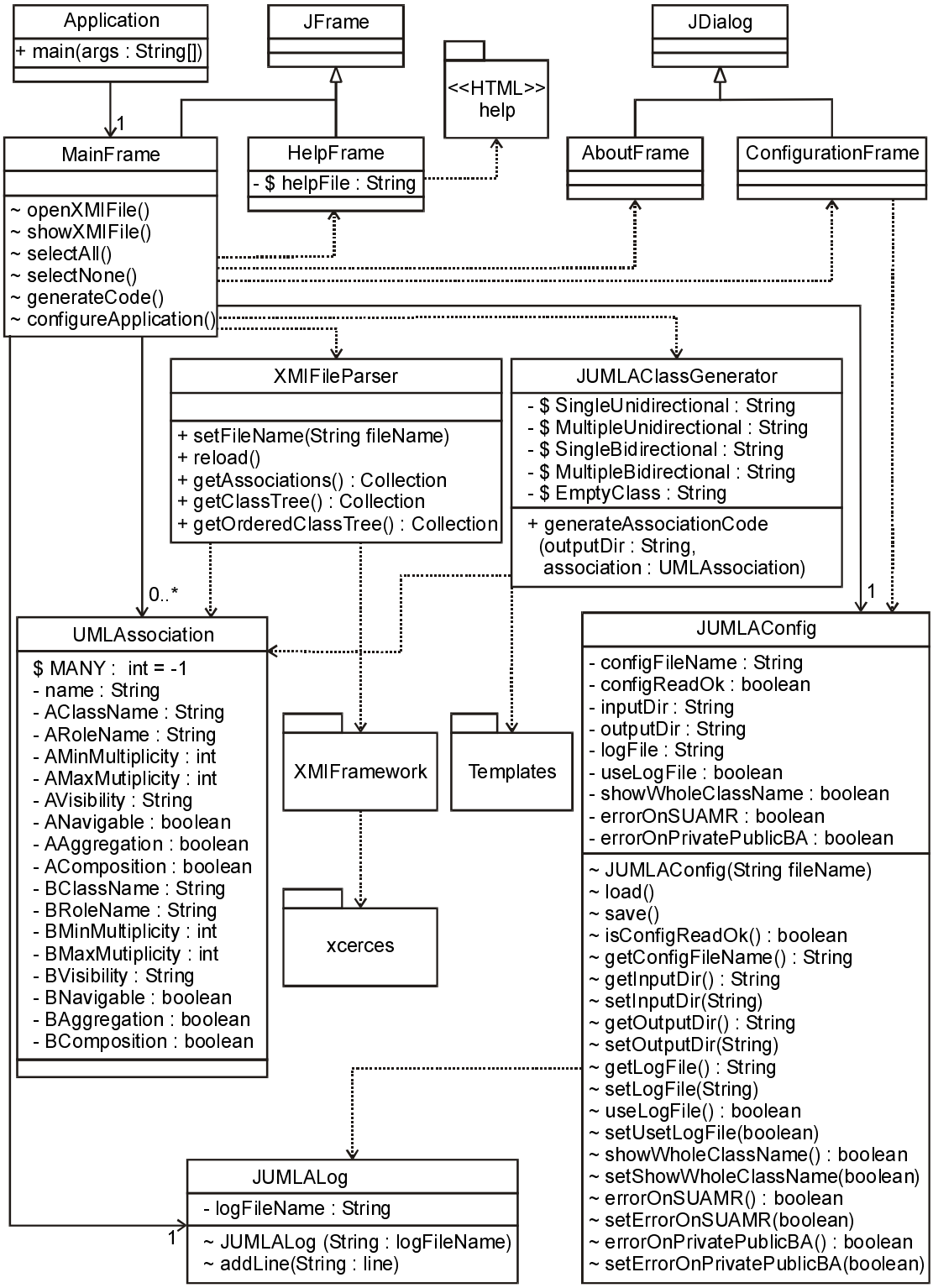

Figura 18.2: Diagrama de clases de *JUMLA*.

18.4 'XMIFileParser'

En esta clase se realiza el análisis de un fichero *XMI*. Se apoya en el paquete *XMIFrameWork* 1.0 de *IBM* [17] que a su vez se apoya en el paquete *Xerces* 2.0.1. El nombre que se le ha asignado a esta clase está pensado para posteriores revisiones pero, de momento, sólo sirve para la obtención del árbol de clases contenido en el diagrama de clases de un fichero *XMI* y de las asociaciones existentes, por lo que los métodos principales son '*getAssociations*' y '*getClassTree*'.

'*getAssociations*' devuelve en una colección instancias de '*UMLAssociation*', es decir, devuelve todas las asociaciones obtenidas en el análisis del fichero *XMI* que se ha introducido mediante la invocación a '*setFileName*' o en la construcción de la instancia tipo '*XMIFileParser*'. '*getClassTree*' devuelve el árbol de clases que se representa mediante una colección de *strings*. Cada *string* contiene una clase que se especifica mediante una secuencia de nombres separados por un punto para representar la jerarquía de paquetes a la que pertenece la clase. El nombre propiamente dicho de la clase se sitúa tras el último punto. Es decir, si una clase '*A*' pertenece al subpaquete '*subX*' del paquete '*X*', se representará como '*X.subX.A*'. Existe también el método '*getOrderedClassTree*' que en lugar de devolver el árbol de clases según el orden en el que se han ido encontrando en el modelo de entrada, lo devuelve con las clases ordenadas alfabéticamente. Este método se ha implementado debido a que parece más lógico representar el árbol de clases ordenado alfabéticamente, y de hecho, *JUMLA* muestra el árbol de clases con ese orden.

18.5 'JUMLAClassGenerator'

Es con '*JUMLAClassGenerator*' como se genera el código de las clases *Java*. A igual que ocurría en la clase que hemos explicado antes, su nombre está pensado para tratar con todos los artefactos de las clases en general, pero hasta ahora sólo se ha implementado la posibilidad de trabajar con las asociaciones.

Tiene definidas cinco constantes privadas que son *strings* y sirven para indicar la ubicación y el nombre de las plantillas de código. Las plantillas de código se encuentran en el paquete '*Templates*'.

Los métodos que generan el código de las asociaciones necesitan de varios parámetros de entrada para indicar la definición de la asociación. Existen cinco métodos públicos para generar el código de la asociación en una clase *Java* y según el tipo de asociación detectado en el modelo se utilizará uno u otro:

- '*generateClassSUA*': genera una clase con una asociación unidireccional sencilla.
- '*generateClassMUA*': clase con asociación múltiple unidireccional.
- '*generateClassSBA*': clase con asociación sencilla bidireccional.
- '*generateClassMBA*': clase con asociación múltiple bidireccional.
- '*generateEmptyClass*': genera una clase vacía que sólo contiene la declaración de ésta.

Debido a que estos métodos son complejos de utilizar (excepto '*generateEmptyClass*') y *JUMLA* requiere generar asociaciones y no clases sueltas, se ha implementado el método '*generateAssociationCode*' que es capaz de utilizar los métodos anteriormente citados en función de la instancia '*UMLAssociation*' que recibe de entrada. Por tanto, de cara al usuario de esta clase, este método es el único que realmente le interesa. Además, este método realiza las comprobaciones

necesarias para conocer si el modelo de entrada cumple con las condiciones impuestas en el apartado 16.1 "*Condiciones de validez del modelo*" y asegura el total cumplimiento con la convención de nombres explicada en el apartado 7.2 "*Convención de nombres*".

18.6 'JUMLAConfig'

En '*JUMLAConfig*' se almacenan los parámetros de configuración del usuario de la herramienta. Se proporcionan métodos para la lectura y escritura de ficheros de configuración así como para leer el estado de la configuración o cambiarlo.

Cuando se ejecuta la aplicación esta es una de las primeras clases en entrar en acción puesto que una de las primeras cosas que hay que hacer es leer la configuración del usuario.

18.7 'JUMLALog'

Para el fichero de seguimiento de sucesos (fichero de *logs*) se emplea esta clase. En el constructor de '*JUMLALog*' se indica como parámetro de entrada el nombre completo del fichero donde se almacenan las incidencias, y mediante el método '*addLine*' se introducen los sucesos que se van produciendo y que se desean que sean añadidos. '*addLine*' se encarga por sí mismo de introducir la fecha y hora en la que se añade cada incidencia.

18.8 'ConfigurationFrame'

Esta clase implementa un cuadro de diálogo donde el usuario puede establecer sus parámetros de configuración personal. Funciona por tanto en estrecha colaboración con '*JUMLAConfig*' y cualquier modificación que se haga en las opciones de usuario requerirá invocar a dicha clase para almacenar el nuevo estado de configuración.

18.9 'HelpFrame'

'*HelpFrame*' implementa la ventana de ayuda de *JUMLA*. Hereda de '*JFrame*' y no de '*JDialog*' porque se ha considerado más conveniente poder tener dos ventanas (la aplicación y la ayuda) por separado y que '*MainFrame*' pueda seguir funcionando mientras se tiene la ayuda abierta. Esta clase básicamente contiene un panel de texto de tipo '*JEditorPane*' dónde se muestra la ayuda en formato *HTML* que se encuentra en el directorio o paquete '*help*'. El fichero inicial de ayuda se especifica en la constante '*helpFile*'.

18.10 'AboutFrame'

Esta última clase es el típico cuadro de diálogo "*acerca de*" con información sobre la herramienta *JUMLA*, su versión y el autor.

CONCLUSIONES

19 CONCLUSIONES

A lo largo de este documento se ha hecho mención de aspectos de *UML* y de *Java* que necesitan de una especial atención para entenderlos y unificarlos bien, y que incluso presentan paradojas de difícil solución. En este capítulo final vamos a comentar a modo de resumen esos aspectos y a hacer una valoración del método que se ha empleado para la implementación de las asociaciones binarias en *Java*.

19.1 Acerca de UML y Java

Para empezar, hemos visto que las asociaciones unidireccionales se deben convertir en bidireccionales cuando se necesita tener restricción de multiplicidad en el extremo '*Source*' si queremos que pueda ser controlada de una forma automática y no directamente por el programador. El estándar *UML* permite utilizar esta restricción pero, según hemos visto, no es posible hacer el control de esta restricción sin convertir la asociación en bidireccional. La razón es sencilla, supongamos que tenemos la asociación de la Figura 19.1 que se ha extraído de la modelación de un sistema operativo. En ella se especifica que un usuario tiene varias claves de acceso pero que dado un objeto '*Clave*', éste sólo pertenece a una instancia de '*Usuario*'.

Figura 19.1: Un usuario posee varias claves (ejemplo tomado de [2]).

El diagrama parece claro y parece tener sentido la restricción de multiplicidad 1 en el propietario, y que sólo sea navegable en un sentido, puesto que dadas las claves, no deberíamos poder conocer a los usuarios. Para controlar la restricción de multiplicidad 1 en el lado del usuario, encontramos dos maneras:

1. El control de la restricción se hace a través de todas las instancias de '*Usuario*' para que no haya dos con la misma clave[19].
2. Se hace a través de cada instancia de '*Clave*'.

La primera opción es compleja de implementar siempre que no tengamos a todos los usuarios metidos en un conjunto pero, además, según el ejemplo, implica dar a conocer las claves a todas las instancias '*Usuario*' lo cual no se debe permitir por razones de seguridad según el modelo del dominio del problema.

La segunda, en cambio, implica necesariamente que una clave conozca a su usuario propietario y por tanto, que la asociación se convierta en bidireccional. Por consiguiente, o bien nos

[19] Nos referimos a que no existan dos usuarios con la misma instancia '*Clave*' y no dos usuarios con distintas instancias '*Clave*' pero con los mismos valores en los atributos.

quedamos con esta segunda opción y cambiamos el diseño por una asociación bidireccional, o bien nos tenemos que olvidar de la restricción de multiplicidad en '*Usuario*' para no tener que hacer nada especial en la clase '*Clave*'.

Sólo nos queda la opción de que si realmente se necesita tener restricciones de multiplicidad en la clase fuente de una asociación unidireccional, sea el programador quien tenga que tener en cuenta esta restricción de diseño en todo momento y garantice su cumplimiento.

En cuanto a la restricción de multiplicidad obligatoria en los extremos de las asociaciones, tenemos que decir que no siempre podemos garantizar su cumplimiento a lo largo de todo el tiempo de vida de las instancias que participan en una asociación. Hay momentos críticos (creación de instancias, cambio dinámico de enlaces y destrucción de instancias) que prácticamente imposibilitan satisfacer en todo momento las restricciones impuestas por la multiplicidad mínima. De este modo, sólo se debe forzar que se cumpla con la multiplicidad en los instantes de tiempo en los que se intente acceder a los enlaces. Esta forma de entender la multiplicidad facilita mucho la implementación de las asociaciones y encontramos que es una interpretación más realista.

En el tema de la visibilidad de las asociaciones, hemos visto que no se ha podido implementar de una forma satisfactoria en las bidireccionales. No es suficiente con los mecanismos de visibilidad que proporciona *Java*, sino que se necesita poder establecer visibilidades a nivel de grupos cerrados de clases y no sólo de paquete. El lenguaje *C++* tiene el concepto de clase amiga que significa que una clase amiga de otra puede acceder a sus elementos privados. Por tanto, las clases amigas de *C++* parecen ser la solución que hemos buscado y echado en falta en *Java* en la implementación de la visibilidad. La clase interna '*Association*' (que implementa el control de los enlaces en uno de los extremos en una asociación bidireccional) puede ser declarada como amiga de la clase del extremo opuesto de la asociación. Así, pese a que '*Association*' sea privada o pública (según la visibilidad establecida en el diseño de la asociación), se consigue acceso mutuo y exclusivo entre ambos extremos de una asociación.

Una forma de implementar la visibilidad de las asociaciones bidireccionales con los mecanismos de implementación de visibilidad que ofrece *Java* es mezclando visibilidades públicas y privadas, y estableciendo comunicación bidireccional entre dos instancias extremas de una asociación. Es decir, se declaran las clases como públicas y algunos métodos se definen como públicos para que los dos extremos de una asociación puedan tener acceso mutuo donde sea necesario; otros métodos se definen como privados. Hay que garantizar que este acceso mutuo es exclusivo a sólo las clases extremas de la asociación, pero esto no se puede conseguir directamente mediante la visibilidad, porque dos extremos privados serían mutuamente no visibles, y por tanto no permitirían la bidireccionalidad; por lo tanto, recurrimos a métodos especiales de "autenticación" en la implementación de los métodos. Así, nos aseguramos de que las asociaciones se comportan según las restricciones del diseño y que sólo las instancias permitidas acceden a los enlaces. Si bien la mezcla de visibilidades pública y privada para la implementación de una asociación funciona, la idea va en contra de lo natural y estético pues, lógicamente, queremos establecer la visibilidad en todos los niveles (en clases, métodos y atributos) según sea la visibilidad en cada extremo de una asociación. Convenimos por tanto, en que esto último es una solución funcional y práctica pero no elegante.

19.2 Acerca del método de implementación de asociaciones

Tras la implementación del método que se ha presentado en este documento hay que preguntarse hasta qué punto vale la pena utilizarlo y hasta qué punto sería más conveniente que se

implementasen las asociaciones de una forma más sencilla mediante un único atributo que controlasen directamente los programadores.

El código que se ha desarrollado para la automatización de la implementación de asociaciones no es muy eficiente; aparte de ocupar mucho espacio y aunque esté encapsulado en una clase interna. No es eficiente sobre todo en las asociaciones unidireccionales puesto que requieren de muchos métodos cuya funcionalidad podría ser controlada directamente sobre un único atributo que almacenase los enlaces. Además, es muy corriente que los enlaces sean estáticos, es decir, se establecen al comienzo de la ejecución del programa y no sufren ningún cambio durante la ejecución, por lo que en este caso, gran parte de código sobra y puede ser reducido.

Encontramos que en las asociaciones bidireccionales está más justificado el uso de este método que nos ayuda fácilmente a la obtención del doble enlace necesario, puesto que con una única sentencia se consigue establecimiento o eliminación de enlaces en las dos direcciones.

Creemos además, que se han logrado los siguientes aspectos positivos:

- Se gana en tiempo de desarrollo puesto que una vez diseñado el diagrama de clases, en seguida pueden estar implementadas las asociaciones y el programador sólo se tiene que encargar de escribir el código para el uso de los enlaces.
- Con el método hay una relación directa entre el diseño y la implementación lo cual es idóneo para poder aplicar fácilmente ingeniería inversa.
- Además, se consigue que el código de las aplicaciones se acerque más al dominio del problema, con lo que el análisis de código se facilita enormemente.
- Los cambios de diseño también se ven beneficiados al reducirse el tiempo que se necesita para acometerlos. Un cambio de especificación de una asociación prácticamente va a requerir cambiar el código de la asociación y, según los casos, puede que sólo implique cambiar el valor de un par de constantes. En caso de requerirse un cambio sustancial de especificación de asociación, la herramienta diseñada para implementar asociaciones lo genera automáticamente de una forma rápida.
- El control de las restricciones de las asociaciones queda establecido en una zona de código fácilmente localizable y está centralizado, por lo que no queda distribuido por todo el código de la aplicación final. Además, el acoplamiento es bajo porque el código de una asociación afecta a lo sumo al par de clases que se ven involucradas en las asociaciones binarias.

A parte de las ventajas e inconvenientes de este método, hay que decir que por sí sólo no sirve para el paso de diseño a implementación y por consiguiente, para el desarrollo rápido de software. Este método hay que combinarlo también con la automatización de la implementación de otros artefactos de diseño de *UML* como la herencia, especificación de atributos y métodos en las clases, dependencias, etc.

19.3 Aspectos que quedan por ver

Son muchas las dificultades que se han tenido para el desarrollo de esta metodología y, por desgracia, tenemos que decir que son bastantes las restricciones de diseño de las asociaciones que no se han podido tener en cuenta. Algunos aspectos parecen tener fácil arreglo y adaptación como por ejemplo la restricción *ordered* que "*especifica que el conjunto de objetos en un extremo de una*

asociación sigue un orden explícito" [2]. En este caso bastaría con dotar al método '*add*' para que los enlaces se añadan según el orden necesario: índice alfabético de un atributo, instante de tiempo en el que se añade el enlace, etc. Otros aspectos, en cambio, posiblemente impliquen abordar la implementación de las asociaciones de una forma distinta. Un ejemplo de esto pueden ser las clases asociación que, a primera vista, parece ser que requieren que la asociación se implemente en una clase independiente de los extremos. Encontramos también que hay aspectos de dificultad intermedia como pueden ser la multiplicidad por rangos, es decir, aquella que no especifica sólo un límite mínimo y máximo sino rangos de valores válidos (p.e. 1, 3, 4 y del 7 al 10 pero no 0, 2, 5, 6 y mayor que 10). Aquí, la solución parece estar en especificar la multiplicidad en un *array* de rangos válidos y simplemente modificar el método '*isValid*' para que compruebe la satisfacción de estos rangos.

Tampoco se han abordado las asociaciones ternarias, cuaternarias, etc., ni la cualificación, el especificador de interfaz o restricciones como *xor*, *subset*, *implicit*, *changeable*, *addOnly*, *frozen*. No obstante, consideramos que los aspectos que sí se han estudiado son suficientes para cubrir la mayoría de las asociaciones que comúnmente se emplean.

Por último, un aspecto importante que queda por ver es la adaptación de este método a otros lenguajes de programación como *C++*, *Smalltalk*, etc. En principio, puede parecer que todo es cuestión de desarrollar las plantillas de código correspondiente al lenguaje de programación que se quiera utilizar, pero casi con toda seguridad el cambio de lenguaje no implica sólo realizar una traducción de plantilla, sino que aspectos concretos de cada lenguaje pueden requerir un tratamiento especializado. No obstante, confiamos en que esos tratamientos específicos no requieran por lo general, de muchos cambios y de afrontar la implementación de las asociaciones de un modo radicalmente diferente. Ya hemos mencionado anteriormente cómo con *C++*, por ejemplo, la visibilidad de las asociaciones se puede implementar de una forma más fácil y elegante que con *Java*. Esperamos pues, haber sentado las bases para la implementación de asociaciones *UML* en cualquier lenguaje OO.

ANEXOS

ANEXO A: CONVENCIÓN DE NOMBRES EMPLEADA EN EL CÓDIGO GENERADO

Nombres claves en las plantillas

En las plantillas de código encontramos palabras claves identificadas fácilmente porque comienzan con el carácter '*%*' y que se han de sustituir según las especificaciones de las asociaciones:

- ***%Package***: Nombre completo del paquete de la clase *Java*.
- ***%Source***: Se referirá a la clase '*Source*', es decir, se sustituirá por la primera clase que especifique el usuario. La primera letra se pondrá en mayúscula, incluso aunque en el diseño el nombre de la clase estuviera con minúscula.
- ***%Target***: Se referirá a la clase '*Target*', es decir, se sustituirá por la segunda clase que será la asociada con '*Source*'. También se pondrá la primera letra en mayúscula. Aquí, hay que tener en cuenta que en las asociaciones bidireccionales se necesita código en la clase '*Source*' que enlace con '*Target*' y al revés, por lo que dentro de las plantillas, *%Source* se referirá a la clase que implementa la plantilla y *%Target* a la clase asociada.
- ***%targetRole***: Es el nombre de rol que juega la clase '*Target*' con respecto a '*Source*'. En *UML*, tanto el nombre de rol como el nombre de asociación son opcionales, de modo que, en función de que estén definidos estos nombres, el valor que usaremos para sustituir *%targetRole* en la generación del código será, en este orden de prioridad:
    1. nombre de rol.
    2. nombre de asociación y nombre de clase en minúscula.
    3. nombre de clase en minúscula.
- ***%sourceRole***: Es el nombre de rol que juega la clase '*Source*' con respecto a '*Target*'. A igual que '*targetRole*', no se requiere que sea introducido forzosamente y en caso de no ser especificado, se aplicarán los mismos criterios que con '*%targetRole*':
    1. nombre de rol.
    2. nombre de la asociación y nombre de clase en minúscula.
    3. nombre de clase en minúscula.
- ***%mandatory***: En las plantillas de multiplicidad sencilla, se sustituirá este valor por '*true* o '*false*' en función de si la asociación hace obligatorio o no la existencia de enlaces.
- ***%min_multiplicity*** y ***%max_multiplicity***: Estas palabras aparecen en las plantillas de multiplicidad mayor que uno y se deben sustituir por el valor asignado en la multiplicidad mínima y máxima. En caso de querer asignar multiplicidad *muchos* en *%max_multiplicity*, se debe sustituir por '*MANY*' (en mayúsculas).

- ***%AssociationKind***: Especifica el tipo de asociación:
    - '*Association*': para asociaciones normales.
    - '*RefelexiveAssociation*': asociaciones reflexivas.
    - '*AggregationWhole*': parte contenedora de las agregaciones.
    - '*AggregationPart*': parte contenida de las agregaciones.
    - '*CompositionWhole*': parte contenedora de las composiciones.
    - '*CompositionPart*': parte contenida de las composiciones..
    - *'ReflexiveAggregationWhole*': parte contenedora en agregaciones reflexivas.
    - '*ReflexiveAggregationPart*': parte contenedora en agregaciones reflexivas.
    - '*ReflexiveCompositionWhole*': parte contenedora en composiciones reflexivas.
    - '*ReflexiveCompositionPart*': parte contenida en composiciones reflexivas.
- ***%visibility***: En este campo se quiere especificar la visibilidad de los enlaces que puede ser, a igual que la notación *Java*, '*public*', '*private*', '*protected*' o nada para indicar que es *friendly* (o de paquete (*package*) según *UML 1.4* [3]). Cuando no se especifique la visibilidad, se entenderá que es pública.

Nombres claves en el código generado

Nombre de la clase interna gestora de una asociación

```
%Source%Target_MultiplicityNavigability_%AssociationKind
```

Atributo de gestión de la asociación

```
%targetRole
```

Nombre de la clase interna en asociaciones reflexivas

```
%SourceRole%TargetRole_MultiplicityNavegability_ReflexiveAssociation
```

Clase interna de asociaciones *todo* / *parte*

```
%Source%Target_MultiplicityNavegability_AggregationWhole
%Source%Target_MultiplicityNavegability_AggregationPart
%Source%Target_MultiplicityNavegability_CompositionWhole
%Source%Target_MultiplicityNavegability_CompositionPart
```

Clase interna de asociaciones reflexivas *todo* / *parte*

```
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveAggregationWhole
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveAggregationPart
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveCompositionWhole
%sourceRole%targetRole_MultiplicityNavegability_ReflexiveCompositionPart
```

ANEXO B: CONDICIONES DE VALIDEZ DE *JUMLA*

Condiciones de validez del modelo de entrada para ser aceptado

Para que *JUMLA* sea capaz de interpretar el modelo de entrada se han de cumplir las siguientes condiciones:

- El fichero *XMI* tiene que haber sido generado por la herramienta *MagicDraw UML*[20] en su versión 5.0 o por una herramienta que escriba código *XMI* similar. Por tanto, la versión de *XMI* utilizada es la 1.0, y de *UML* la 1.3 (por ser las versiones que utiliza *MagicDraw* 5.0).
- Los ficheros *XMI* de entrada pueden estar comprimidos en formato *ZIP*.
- Las extensiones de ficheros que se reconocen son '*.xml*', '*.xmi*' y '*.zip*'.
- En las asociaciones unidireccionales la clase '*Source*' no puede tener ninguna restricción de multiplicidad y debe ser, por tanto, del tipo *many* no restringida (0..*). Si se especifica restricción de multiplicidad en '*Source*' en una asociación unidireccional, *JUMLA* da la opción de descartar la asociación entera o de ignorar la restricción y tratarla como si fuese (0..*).
- Si existen dos o más asociaciones entre las mismas dos clases, se requiere que las asociaciones no sean anónimas.
- Si una asociación es reflexiva, se requiere que se especifique nombre de rol en ambos extremos de ella.
- La generación de las clases en *Java* se hace por paquetes, siendo el paquete raíz el nombre del modelo de entrada. El nombre del modelo de entrada, de los paquetes en él especificados, así como de las clases, han de cumplir las reglas de nombrado de *Java*. Es decir, tienen que ser caracteres alfanuméricos (no se contemplan los caracteres ortográficos particulares de una lengua: acentos, letra "eñe", etc.) y sólo se permite como símbolo adicional el carácter de subrayado: '_'.
- *JUMLA* no interpreta del modelo de entrada los artefactos *UML* que no se refieren directamente a una asociación. Así, pese a que una clase tenga atributos y métodos definidos o sea heredera de otra, estas características no se tienen en cuenta y son ignoradas por la herramienta en la generación de código.

[20] *MagicDraw UML* es una herramienta escrita en *Java* para el diseño *UML*. Es una herramienta bastante popular capaz de funcionar en cualquier plataforma que tenga instalado *Java*™ *2 Runtime Environment*. Se puede obtener una versión demo con un tiempo limitado de uso en la *URL* http://www.magicdraw.com.

Condiciones de validez del código generado

Una vez reconocido el modelo de entrada y las asociaciones que este contiene, *JUMLA* genera el código de las asociaciones seleccionadas por el usuario. El código generado automáticamente requiere que las siguientes condiciones sean tenidas en cuenta:

- Si al generar el código de una clase *JUMLA* se encuentra con que ya existe dicha clase, se intenta añadir sólo el código referente a la asociación. En esta situación, se garantiza la perfecta compilación de la clase si y sólo si ésta compilaba perfectamente antes de la adición del código gestor de la asociación. En resumen, se garantiza que el código que inserta *JUMLA* es correcto pero no el que existía previamente.

- El código generado para las asociaciones bidireccionales con visibilidad pública o de paquete en un extremo y privada o protegida en otro tienen la particularidad de que sólo permite establecimiento y eliminación de enlaces desde el extremo privado o protegido. Se da la opción de no generar el código correspondiente a las asociaciones que así estén definidas.

- No se implementan las asociaciones bidireccionales con visibilidad privada (o protegida) – privada (o protegida).

- El código de una asociación (clase interna '*Association*' y su atributo) se inserta siempre al final de la clase y se puede reconocer dónde comienza porque lleva la siguiente etiqueta:

  ```
  /** !Código generado por JUMLA */
  ```

- Asimismo, el final del código generado por la herramienta correspondiente a cada asociación se puede reconocer por la etiqueta:

  ```
  /** !Fin de código generado por JUMLA */
  ```

- Al insertar una asociación en una clase que existía previamente, *JUMLA* intenta reconocer si se trata de una nueva asociación que hay que añadir al código de la clase o si se trata de un cambio de diseño. Para ello, si se encuentra con el código de una asociación con el mismo nombre de '*targetRole*', lo que se hará es sustituir esta asociación por la nueva.

ANEXO C: CONDICIONES DE USO DEL INTERFAZ

En este anexo se hace un resumen del interfaz para el manejo de las asociaciones generadas automáticamente.

Interfaz del código para la gestión de una asociación

```
class Source {
  final Association targetRole = new Association();
  class Association {
    private final static boolean mandatory; // Sólo para multiplicidades sencillas
    private final static long MIN; // Sólo para multiplicidades mayor que uno
    static final int MANY = -1;
    private final static long MAX; // Sólo para multiplicidades mayor que uno
    boolean isBidirectional();
    boolean isMandatory();
    boolean isMultiple();
    long getMIN();
    long getMAX();
    boolean isValid();
    long numberOfLinks();
    boolean test(Target query_link);
    boolean test(Collection query_links); // Sólo para multiplicidades mayor que uno
    Target get() throws Exception; // Sólo para multiplicidades sencillas
    Collection get() throws Exception; // Sólo para multiplicidades mayor que uno
    boolean add(Target new_link);
    boolean add(Collection new_links); // Sólo para multiplicidades mayor que uno
    int remove();
    int remove(Target old_link);
    int remove(Collection old_links); // Sólo para multiplicidades mayor que uno
  }
}
```

Condiciones de uso

Clase '*Source*' y atributo '*targetRole*'

En cada clase '*Source*', habrá declarada una clase interna del mismo estilo que '*Association*' por cada asociación que se especifique en el diseño en la que '*Source*' se vea involucrada. Asimismo, habrá un único atributo '*targetRole*' (de tipo '*Association*') por cada clase interna '*Association*' y para evitar manipulaciones no permitidas de éste, se declara como *final*.

Hay que advertir que desde dentro de la clase '*Source*' no se puede evitar la creación de nuevas instancias de '*Association*' pero esto no debe hacerse como condición para que se pueda garantizar un correcto control de las asociaciones.

Constantes de la clase '*Association*'

Todas las constantes de '*Association*' son privadas y tienen que estar especificadas en tiempo de compilación. Sirven para especificar la multiplicidad de '*Target*' respecto a '*Source*'. El usuario de la clase '*Source*', en general, no va a tener acceso directo a ellas.

private boolean mandatory

Esta constante se utiliza para asociaciones con multiplicidad sencilla en el extremo '*Target*'. Si se le asigna el valor *true*, se está especificando que todas las instancias de '*Source*' a '*Target*' tienen que tener siempre un enlace (multiplicidad 1..1) y con el valor *false*, se está especificando que los enlaces son opcionales (multiplicidad 0..1).

- Sólo se declara en la clases '*Source*' cuya multiplicidad en el extremo es sencilla.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*'. Se recomienda que para conocer si la multiplicidad es *mandatory* o no, se utilice el método '*isMandatory*' puesto que la constante '*mandatory*' es para uso interno en la clase '*Association*'.

private long MIN

Especifica la multiplicidad mínima de '*Target*' respecto a '*Source*'.

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*' pero se recomienda que para conocer la multiplicidad mínima se utilice el método '*getMIN*' puesto que es una constante de uso interno en la clase '*Association*'.
- Su valor tiene que ser mayor o igual que cero.

int MANY

La constante '*MANY*' sirve para representar la multiplicidad muchos de *UML* (*). Con el método '*getMAX*' veremos para qué puede ser útil el conocimiento de esta constante.

- La constante '*MANY*' se declarará siempre, incluso cuando se trate con multiplicidades sencillas.

private long MAX

Especifica la multiplicidad máxima de '*Target*' respecto a '*Source*'. Junto con '*MIN*', sirve para establecer la multiplicidad en las asociaciones múltiples. En las asociaciones sencillas, no se han utilizado estas dos constantes (aunque podría ser deseable a efectos de uniformidad de interfaz) para evitar que por accidente se especifiquen valores que se salgan de los rangos (0..1) y (1..1), puesto que el código está preparado para tratar un único enlace y no muchos.

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Se declara como privada y por tanto, sólo se puede acceder a ella desde la clase '*Source*' pero se recomienda que para conocer la multiplicidad máxima se utilice el método '*getMAX*' puesto que es una constante de uso interno en la clase '*Association*'.
- Su valor tiene que ser mayor o igual que '*MIN*'.

- Cuando la multiplicidad máxima sea de muchos, '*MAX*' será igual a la constante '*MANY*'.

Métodos de definición de la asociación

Los métodos que aquí se detallan sirven de apoyo para el conocimiento de la definición de las especificaciones de la asociación.

boolean isBidirectional()

Dice si la asociación es bidireccional (verdadero) o navegable sólo desde la clase fuente (falso).

boolean isMandatory()

El método devuelve verdadero cuando la asociación es obligatoria (multiplicidad mínima mayor que 0) para la clase '*Source*'.

boolean isMultiple()

Devuelve verdadero cuando la multiplicidad en el destino con respecto a la fuente es múltiple (multiplicidad máxima mayor que 1) y falso cuando es sencilla (multiplicidad máxima igual a 1).

long getMIN()

Devuelve la multiplicidad mínima del extremo '*Target*' respecto a '*Source*'.

long getMAX()

Devuelve la multiplicidad máxima del extremo '*Target*' respecto a '*Source*'.

- La multiplicidad muchos (*) se especifica con la constante '*MANY*', de forma que si la multiplicidad máxima es muchos, en la invocación a este método se obtendrá el mismo valor que el especificado en '*MANY*'.

Métodos para el conocimiento del estado de los enlaces

En este apartado, definimos los métodos de la segunda categoría, que sirven para conocer en qué estado se encuentran los enlaces.

boolean isValid()

Devuelve verdadero cuando existe el número correcto de enlaces de acuerdo con la multiplicidad de la asociación en el sentido de '*Source*' a '*Target*' y, por tanto, se considera que se está cumpliendo con la especificación de diseño de la asociación en el extremo '*Source*'.

Esta función está pensada para obtener la validez de todas las restricciones de diseño de la asociación. No obstante, como los problemas a los que nos enfrentamos según los objetivos marcados en este trabajo se centran en la multiplicidad, de momento, '*isValid*' sólo comprueba esta restricción. Futuras versiones de este método, podrían realizar otras comprobaciones adicionales.

- En las asociaciones bidireccionales, que se obtenga el valor verdadero no implica necesariamente que los enlaces estén bien establecidos, puesto que habría que comprobar también el sentido contrario, desde las instancias de '*Target*' a '*Source*'.

long numberOfLinks()

Devuelve el número de enlaces que la instancia '*Source*' tiene establecidos con instancias '*Target*'.

Métodos para la comprobación, acceso, establecimiento y eliminación de los enlaces

En la última categoría se especifican los métodos de gestión de la asociación.

boolean test(Target query_link)

Dice si existe enlace con la instancia de '*Target*' que se pasa como parámetro de entrada.

- Si de entrada se introduce una variable con valor *null*, se devuelve verdadero si no hay establecido ningún enlace.

boolean test(Collection query_links)

Dice si existe enlace con todas las instancias '*Target*' que se pasan como parámetro de entrada contenidas en una colección de objetos.

- Sólo se declara en clases '*Source*' con multiplicidad en el extremo mayor que uno.
- Si algún elemento de la colección de entrada no es de tipo '*Target*' se devuelve falso.
- Si la colección de entrada está vacía, se devuelve verdadero.
- Si de entrada se introduce una variable con valor *null*, se devuelve verdadero si no hay establecido ningún enlace.

Target get() throws UnsetAssociationException

Obtiene la única instancia '*Target*' con la que se tiene establecido un enlace.

- Sólo se declara en la clases '*Source*' cuya multiplicidad en el extremo es sencilla.
- Lanza la excepción '*UnsetAssociationException*' si no se cumple con las restricciones de diseño de la asociación en '*Source*', es decir, si la llamada a '*isValid*' devuelve falso.
- Si no existe ningún enlace pero se cumplen con las restricciones de multiplicidad mínima, se devuelve el valor *null*.

Collection get() throws UnsetAssociationException

Obtiene todas las instancias '*Target*' con las que se tiene establecido un enlace.

- Sólo se declara en clases '*Source*' cuya multiplicidad en el extremo es mayor que uno.
- Lanza la excepción '*UnsetAssociationException*' si no se cumple con las restricciones de diseño de la asociación en '*Source*', es decir, si la llamada a '*isValid*' devuelve falso.
- Si no existe ningún enlace pero se cumplen con las restricciones de multiplicidad mínima, se devuelve una colección vacía, sin elementos.

boolean add(Target new_link)

Añade un enlace con la instancia de tipo '*Target*' que se introduce como parámetro de entrada.

- Si '*Source*' ya tiene establecido el enlace con la instancia de entrada, el método devuelve verdadero pero sin realizar ningún cambio.
- Si '*Source*' ya tiene establecidos el número máximo de enlaces permitidos, no se hace nada y se devuelve falso. En este caso, la única forma de establecer enlace con la instancia '*new_link*' es borrando otro enlace (mediante el método '*remove*') e invocar a '*add(new_link)*' (en ese orden). Esto último hay que tenerlo especialmente en cuenta en las asociaciones sencillas puesto que en ellas, si ya hay establecido un enlace, '*add*' no funciona hasta que se elimine.
- Si el parámetro de entrada es *null* no se hace nada y se devuelve falso.
- En las asociaciones bidireccionales, debido a que hay que establecer un doble enlace (uno para cada uno de los dos sentidos de la asociación), primero se comprueba que se dan las condiciones necesarias para establecer el enlace de '*Source*' a '*Target*'. Si se puede, se establece y se comprueba que se puede establecer enlace en el otro sentido (de '*Target*' a '*Source*'). Para ello, se hace internamente una llamada al método 'add' del otro extremo, es decir, del atributo '*sourceRole*'[21] de '*new_link*' para que establezca enlace con '*Source*'. Si '*new_link.sourceRole.add*' responde diciendo que no ha establecido el enlace, entonces se elimina el enlace de '*Source*' a '*Target*' que se acababa de establecer y se devuelve falso. [22]
- Si el enlace queda correctamente establecido, se devuelve verdadero.

boolean add(Collection new_links)

Establece un enlace con cada una de las instancias de tipo '*Target*' contenidas en la colección de entrada.

- Sólo se declara en las clases '*Source*' cuya multiplicidad con el extremo '*Target*' es mayor que uno.
- Si alguna instancia de la colección de entrada no es de tipo '*Target*' se devuelve falso y no se produce ningún cambio.
- Si el parámetro de entrada es *null* no se hace nada y se devuelve falso.
- El número de enlaces final no puede ser mayor que los permitidos por la multiplicidad máxima, en cuyo caso se devuelve falso y se dejan los enlaces de '*Source*' tal y como estaban antes de la llamada.
- En las asociaciones bidireccionales, se tiene que lograr establecer enlace en el sentido inverso con todos los extremos opuestos, desde las instancias '*Target*' a la de '*Source*'.

[21] '*Target*' sigue la misma estructura que '*Source*', por lo que el atributo que gestiona los enlaces en '*Target*' es '*sourceRole*'.

[22] Este párrafo, en definitiva está diciendo que en las asociaciones bidireccionales, mediante una única llamada a '*add*' de '*Source*', se consigue enlace en los dos sentidos, es decir, se consigue enlace bidireccional.

Por tanto, a igual que ocurría en la definición del método '*add*' anterior, por cada instancia '*Target*' contenida en '*new_links*' se hace una llamada al método '*add*' definido en '*Target.sourceRole*' que establece enlace en el sentido opuesto. Si una sola de estas llamadas falla, el método devuelve falso y deja los enlaces tal y como estaban antes de la llamada.

- Sólo si todos los enlaces quedan correctamente establecidos, se devuelve verdadero.

int remove()

Elimina todos los enlaces que contiene '*Source*'.

- Devuelve "cero" si al borrar los enlaces se siguen cumpliendo las restricciones de multiplicidad mínima en '*Source*'.
- Devuelve "uno" si al borrar los enlaces se deja de cumplir con las restricciones de multiplicidad en '*Source*'.
- En las asociaciones bidireccionales, devuelve "dos", y no "cero", si en alguno de los enlaces destino se han dejado de cumplir con las restricciones de multiplicidad mínima respecto al extremo '*Source*', es decir, en el sentido de '*Target*' a '*Source*'[23].
- En las asociaciones bidireccionales, devuelve "tres", y no "uno", si tanto en el extremo '*Source*' como en alguno de los antiguos enlaces '*Target*' se deja de cumplir con las restricciones de multiplicidad mínima.

int remove(Target old_link)

Elimina el enlace establecido con la instancia '*Target*' de entrada.

- En las asociaciones de multiplicidad sencilla en el extremo '*Target*' respecto a '*Source*', éste método se declara por mantener un interfaz uniforme pero prácticamente equivale a 'remove*()*' puesto que se trata de borrar el único enlace que contiene '*Source*'.
- Los valores de retorno del método '*remove*' anterior se aplican aquí también.
- Además, devuelve "–1" si no existe enlace con las instancia '*Target*' de entrada y por tanto, no se puede borrar el enlace.
- Devuelve también "–1 " si de entrada se introduce el valor *null*.

19.3.1.1 int remove(Collection old_links)

Borra los enlaces establecidos con todas las instancias '*Target*' contenidas en la colección de entrada.

- Sólo se declara en las clases '*Source*' cuyo extremo tiene multiplicidad mayor que uno.

[23] La eliminación de enlaces en las asociaciones bidireccionales se consigue con la misma técnica que se ha explicado en '*add*'. Nuevamente, se trata de borrar el enlace en el sentido de '*Source*' a '*Target*' y de invocar al extremo '*Target*' pero esta vez, para que borre el enlace en el sentido de '*Target*' a '*Source*'.

- Los valores de retorno del primer método '*remove*' se aplican aquí también.
- Se ignoran aquellos elementos de la colección que no sean de tipo '*Target*' o con los que no se tenga ningún enlace. Es decir, en la colección de entrada sólo se tienen en cuenta las instancias '*Target*' con las que '*Source*' tiene establecido enlace.
- Si algún elemento de la colección de entrada no es de tipo 'Target', se obtiene como valor de retorno "–1".
- Si no se llega a borrar ningún enlace ('*old_links*' es *null* o no contiene ningún enlace existente) se devuelve "–1".

ANEXO D: PLANTILLAS EMPLEADAS EN LA GENERACIÓN DE CÓDIGO

A partir de las plantillas de código se obtiene la codificación de las asociaciones. Éstas son:

- '*SingleUnidirectional.java~*': para asociaciones unidireccionales con multiplicidad sencilla en la clase '*Target*'.
- '*MultipleUnidirectional.java~*': para asociaciones unidireccionales con multiplicidad mayor que uno en la clase '*Target*'.
- '*SingleBidirectional.java~*': para clases con una asociación bidireccional cuyo extremo tiene multiplicidad sencilla.
- '*MultipleBidirectional.java~*': para clases con una asociación bidireccional cuyo extremo tiene multiplicidad mayor que uno.
- '*EmptyClass.java~*': para clases vacías que sólo necesitan que sean declaradas. Esto se da típicamente en las clases '*Target*' de las asociaciones unidireccionales.

'SingleUnidirectional.java~'

```
package %package;

public class %Source {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_SingleUnidirectional_%AssociationKind
        %targetRole = new %Source%Target_SingleUnidirectional_%AssociationKind();

  %visibility class %Source%Target_SingleUnidirectional_%AssociationKind {

    /** Para representar multiplicidad muchos (* en UML)
     *  No se utiliza aquí pero se pone por uniformidad de interfaz con la
     *  implementación de otro tipo de asociaciones
     */
    %visibility final static int MANY = -1;

    /** Enlace de %Source a %Target:
     *       obligatorio (multiplicidad 1..1): mandatory = true
     *       opcional (multiplicidad 0..1): mandatory = false
     */
    private final static boolean mandatory = %mandatory;

    /** Almacena el enlace con %Target */
    private %Target link = null;

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return false;
    }
```

```
/** Dice si la asociación es obligatoria o no
 *  @return true si la asociación es obligatoria
 */
%visibility boolean isMandatory() {
  return mandatory;
}

/** Dice si la asociación es múltiple o no
 *  @return true si es múltiple, false si sencilla
 */
%visibility boolean isMultiple() {
  return false;
}

/** Devuelve la multiplicidad mínima del final de la asociación
 *  @return multiplicidad mínima
 */
%visibility long getMIN() {
  if (isMandatory()) {
    return 1;
  }
  return 0;
}

/** Devuelve la multiplicidad máxima del final de la asociación
 *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
 *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
 */
%visibility long getMAX() {
  return 1;
}

/** Dice si %Source cumple con las restricciones de la asociación
 *  @return true si sí las cumple, false en caso contrario
 */
%visibility boolean isValid() {
  return ((! mandatory) || (link != null));
}

/** Devuelve el número de enlaces activos que tiene el objeto
 *  @return Número de enlaces
 */
%visibility long numberOfLinks() {
  if (link == null) {
    return 0;
  }
  return 1;
}

/** Comprueba si %Source está asociado con un objeto %Target concreto
 *  @param query_link Supuesto enlace %Target
 *  @return true si existe dicho enlace, false en caso contrario
 */
%visibility boolean test(%Target query_link) {
  return link == query_link;
}

/** Devuelve el enlace %Target contenido en link
 *  Lanza la excepción UnsetAssociationException si la asociación no
 *  cumple con los requisitos de la asociación
 *  @return link (null si no existe ningún enlace)
```

```
     */
    %visibility %Target get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
      return link;
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se añade el enlace o ya existía, false si no se puede
     *  debido a que ya tiene otro enlace distinto o new_link es null
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        // Si no se puede establecer el enlace, se termina
        if ((new_link == null) || ((link != null) && (link != new_link))) {
          return false;
        }
        // Se establece el enlace en %Source
        link = new_link;
        return true;
      }
    }

    /** Elimina el enlace establecido en link
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de multiplicidad, 1 si se deja de cumplir
     */
    %visibility int remove() {
      synchronized (this) {
        // Elimina el enlace
        link = null;
        // Devuelve el valor correcto
         if (mandatory) {
           return 1;
         }
         return 0;
        }
     }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si deja de cumplir con ellas y -1
     *  si no existe enlace con old_link o old_link es null
     */
    %visibility int remove(%Target old_link) {
      if ((old_link != null) && test(old_link)) {
        return remove();
      }
      return -1;
    }

  }
  /** !Fin de código generado por JUMLA */

}
```

'MultipleUnidirectional.java~'

```
package %package;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class %Source {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_MultipleUnidirectional_%AssociationKind
      %targetRole = new %Source%Target_MultipleUnidirectional_%AssociationKind();

  %visibility class %Source%Target_MultipleUnidirectional_%AssociationKind {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = %min_multiplicity;

    /** Para representar multiplicidad muchos (* en UML) */
    %visibility final static int MANY = -1;
    /** Multiplicidad máxima del enlace */
    private final static long MAX = %max_multiplicity;

    /** HashSet donde se almacenan los enlaces con instancias %Target */
    private HashSet links_hs = new HashSet();

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return false;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return (MIN > 0);
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return ((MAX > 1) || (MAX == MANY));
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      return MIN;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
     */
    %visibility long getMAX() {
      return MAX;
```

```
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((links_hs.size() >= MIN) &&
              ((links_hs.size() <= MAX) || (MAX == MANY)));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      return links_hs.size();
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     */
    %visibility boolean test(%Target query_link) {
      if (query_link == null) {
        return links_hs.isEmpty();
      }
      return links_hs.contains(query_link);
    }

    /** Comprueba si %Source está asociado con una colección de objetos %Target
     *  @param query_links Colección de supuestos enlaces
     *  @return true si todos los elementos de links_hs están enlazados
     *  con %Source o si links_hs está vacío y query_links también o es null
     */
    %visibility boolean test(Collection query_links) {
      if (query_links == null) {
        return links_hs.isEmpty();
      }
      return links_hs.containsAll(query_links);
    }

    /** Devuelve una colección con la copia de los enlaces con %Target
     *  contenidos en links_hs. Lanza la excepción UnsetAssociationException si
     *  la asociación no cumple con los requisitos de la asociación
     *  @return copia de links_hs
     */
    %visibility Collection get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
      return (Collection) links_hs.clone();
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se añade el enlace o ya existía, false si no se puede
     *  debido a que se excede la multiplicidad máxima o new_link es null
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        if (new_link != null) {
          // Si se intenta enlazar con un objeto que ya está enlazado, no se
```

```
          // hace nada pero se devuelve como si se hubiese establecido el enlace
          if (test(new_link)) {
            return true;
          }
          // Si añadir new_link no excede la multiplicidad máxima
          if ((MAX == MANY) || (links_hs.size() < MAX)) {
            // se establece el enlace
            links_hs.add(new_link);
            return true;
          }
        }
        return false;
      }
    }

    /** Añade un enlace con cada uno de los elementos de new_links
     *  @param new_links Colección de elementos %Target con los que se
     *  intenta enlazar
     *  @return true si se establecen TODOS los enlaces, false si se excede de
     *  la multiplicidad máxima o alguno de los nuevos enlaces no se puede
     *  establecer o new_links es null
     */
    %visibility boolean add(Collection new_links) {
      if (new_links == null) {
        return false;
      }
      synchronized (this) {
        if (new_links != null) {
          // Sobre una copia de new_links se eliminan aquellos objetos
          // %Target que tengan establecido el enlace previamente
          HashSet copied_new_links = new HashSet(new_links);
          copied_new_links.removeAll(links_hs);
          // Si añadir los nuevos elementos no excede de la multiplicidad máxima
          if ((MAX == MANY) ||
              ((copied_new_links.size() + links_hs.size()) <= MAX)) {
            // se añaden de uno en uno para poder asegurarse que todos los
            // elementos son de tipo %Target
            Iterator i = copied_new_links.iterator();
            while (i.hasNext()) {
              try {
                links_hs.add((%Target) i.next());
              } catch (ClassCastException e) {
                // Si algún elemento no es tipo %Target, se eliminan los
                // añadidos hasta ahora y se devuelve false
                links_hs.removeAll(new_links);
                return false;
              }
            }
            return true;
          }
        }
        return false;
      }
    }

    /** Elimina todos los enlaces establecidos en links_hs
     *  @return 0 si se eliminan todos los enlaces y se siguen cumpliendo con
     *  las restricciones de la asociación y 1 si se deja de cumplir con ellas
     */
    %visibility int remove() {
      synchronized (this) {
        // Elimina los enlaces
```

```
        links_hs.clear();
        // Devuelve el valor correcto
        if (MIN > 0) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si deja de cumplir con ellas y -1
     *  si no existe enlace con old_link o old_link es null
     */
    %visibility int remove(%Target old_link) {
      synchronized (this) {
        if ((old_link != null) && test(old_link)) {
          // Elimina el enlace de %Source a %Target
          links_hs.remove(old_link);
          // Devuelve el valor correcto
          if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }

    /** Borra los enlaces establecidos con un conjunto de objetos %Target
     *  @param old_links Colección de objetos %Target cuyo enlace con %Source
     *  se quiere borrar
     *  @return 0 si al eliminar los enlaces de %Source a old_links se sigue
     *  cumpliendo con las restricciones de la asociación y 1 si se deja de
     *  cumplir con las restricciones. -1 si algún elemento de old_links no es
     *  de tipo %Target u old_links es null y por tanto, no se hace nada
     */
    %visibility int remove(Collection old_links) {
      synchronized (this) {
        if (old_links != null) {
          HashSet copied_links_hs = (HashSet) links_hs.clone();
          // Los enlaces se eliminan de uno en uno
          Iterator i = old_links.iterator();
          while (i.hasNext()) {
            try {
              links_hs.remove((%Target) i.next());
            } catch (ClassCastException e) {
              // Si algún elemento no es tipo %Target, se deja todo como estaba
              links_hs = copied_links_hs;
              return -1;
            }
          }
          // Devuelve el valor correcto
          if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }
```

```
  }
  /** !Fin de código generado por JUMLA */

}
```

'SingleBidirectional.java~'

```
package %package;

public class %Source {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_SingleBidirectional_%AssociationKind
         %targetRole = new %Source%Target_SingleBidirectional_%AssociationKind();

  %visibility class %Source%Target_SingleBidirectional_%AssociationKind {

    /** Para representar multiplicidad muchos (* en UML)
     *  No se utiliza aquí pero se pone por uniformidad de interfaz con la
     *  implementación de otro tipo de asociaciones
     */
    %visibility final static int MANY = -1;

    /** Enlace de %Source a %Target obligatorio (true) u opcional (false) */
    private final static boolean mandatory = %mandatory;

    /** Almacena el enlace con %Target */
    private %Target link = null;

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return true;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return mandatory;
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return false;
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      if (isMandatory()) {
        return 1;
      }
```

```
      return 0;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
     */
    %visibility long getMAX() {
      return 1;
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((! mandatory) || (link != null));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      if (link == null) {
        return 0;
      }
      return 1;
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     */
    %visibility boolean test(%Target query_link) {
      return link == query_link;
    }

    /** Devuelve el enlace %Target contenido en link
     *  Lanza la excepción UnsetAssociationException si la asociación no
     *  cumple con los requisitos de la asociación
     *  @return link (null si no existe ningún enlace)
     */
    %visibility %Target get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
      return link;
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se establece el enlace, false si no se puede debido a
     *  que ya tiene otro enlace distinto, new_link no lo permite o
     *  new_link es nulo
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        // Si no se puede establecer el enlace, se termina
        if ((new_link == null) || ((link != null) && (link != new_link))) {
          return false;
        }
        // Si se intenta enlazar con un objeto con el que ya está enlazado, no
```

```
        // se hace nada pero se devuelve como si se hubiese establecido el enlace
        if (test(new_link)) {
          return true;
        }
        // Se establece enlace de %Source a %Target
        link = new_link;
        // Si en el otro extremo no está establecido el enlace y
        if ((! new_link.%sourceRole.test(%Source.this)) &&
            // no se puede establecer de %Target a %Source
            (! new_link.%sourceRole.add(%Source.this))) {
           // se elimina
           link = null;
           return false;
         }
         return true;
      }
    }

    /** Elimina el enlace establecido en link
     *  @return 0 si se elimina el enlace y los objetos %Source y link
     *  siguen cumpliendo con las restricciones de multiplicidad, 1 si %Source
     *  deja de cumplir con las restricciones, 2 si link deja de cumplir
     *  y 3 si ambos
     */
    %visibility int remove() {
      synchronized (this) {
        boolean invalid_link = false;
        %Target old_link = link;
        if (link != null) {
          // Se elimina el enlace
          link = null;
          // Si la solicitud de eliminar el enlace viene de otro objeto que no
          // sea el viejo enlace
          if  ((old_link.%sourceRole.test(%Source.this)) &&
               // se llama a %targetRole para que elimine su enlace con %Source
               (old_link.%sourceRole.remove(%Source.this) == 1)) {
            invalid_link = true;
          }
        }
        // Devuelve el valor correcto
        if (mandatory && invalid_link) {
          return 3;
        } else if (invalid_link) {
          return 2;
        } else if (mandatory) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra el enlace supuestamente establecido con un objeto %Target
     *  @param old_link Supuesto enlace de tipo %Target
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de multiplicidad, 1 si se deja de cumplir con las
     *  restricciones y -1 si no existe el enlace con link u old_link es null
     *  y por tanto, no se hace nada
     */
    %visibility int remove(%Target old_link) {
      if ((old_link != null) && test(old_link)) {
        return remove();
      }
```

```
      return -1;
    }

  }
  /** !Fin de código generado por JUMLA */

}
```

'MultipleBidirectional.java~'

```
package %package;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class %Source {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  %visibility final %Source%Target_MultipleBidirectional_%AssociationKind
       %targetRole = new %Source%Target_MultipleBidirectional_%AssociationKind();

  %visibility class %Source%Target_MultipleBidirectional_%AssociationKind {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = %min_multiplicity;

    /** Para representar multiplicidad muchos (* en UML) */
    %visibility final static int MANY = -1;
    /** Multiplicidad máxima del enlace */
    private final static long MAX = %max_multiplicity;

    /** HashSet donde se almacenan los enlaces con instancias %Target */
    private HashSet links_hs = new HashSet();

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return true;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return (MIN > 0);
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return ((MAX > 1) || (MAX == MANY));
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
```

```
     */
    %visibility long getMIN() {
      return MIN;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY (multiplicidad máxima *)
     */
    %visibility long getMAX() {
      return MAX;
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((links_hs.size() >= MIN) &&
              ((links_hs.size() <= MAX) || (MAX == MANY)));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      return links_hs.size();
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     *  query_link == null implica comprobar si no existen enlaces
     */
    %visibility boolean test(%Target query_link) {
      if (query_link == null) {
        return links_hs.isEmpty();
      }
      return links_hs.contains(query_link);
    }

    /** Comprueba si %Source está asociado con una colección de objetos %Target
     *  @param query_links Colección de supuestos enlaces
     *  @return true si todos los elementos de links_hs están enlazados
     *  con %Source o si links_hs está vacío y query_links también o es null
     */
    %visibility boolean test(Collection query_links) {
      if (query_links == null) {
        return links_hs.isEmpty();
      }
      return links_hs.containsAll(query_links);
    }

    /** Devuelve una colección con la copia de los enlaces con %Target
     *  contenidos en links_hs. Lanza la excepción UnsetAssociationException si
     *  la asociación no cumple con los requisitos de la asociación
     *  @return Copia de links_hs
     */
    %visibility Collection get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
```

```
      return (Collection) links_hs.clone();
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se realiza el enlace o ya existía, false si no se puede
     *  debido a que se excede la multiplicidad máxima o new_link es null
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        if (new_link != null) {
          // Si se intenta enlazar con un objeto que ya está enlazado, no se
          // hace nada pero se devuelve como si se hubiese establecido el enlace
          if (test(new_link)) {
            return true;
          }
          // Si añadir new_link no excede la multiplicidad máxima
          if ((MAX == MANY) || (links_hs.size() < MAX)) {
            // se establece el enlace de %Source a %Target
            links_hs.add(new_link);
            // Si en el otro extremo no está establecido el enlace y
            if ((! new_link.%sourceRole.test(%Source.this)) &&
                // no se puede establecer de %Target a %Source
                (! new_link.%sourceRole.add(%Source.this))) {
              // se elimina
              links_hs.remove(new_link);
            } else {
              return true;
            }
          }
        }
        return false;
      }
    }

    /** Añade un enlace con cada uno de los elementos de new_links_hs
     *  @param new_links Colección de elementos %Target con los que se
     *  intenta enlazar
     *  @return true si se establecen TODOS los enlaces, false si se excede de
     *  la multiplicidad máxima o alguno de los nuevos enlaces no se puede
     *  establecer
     */
    %visibility boolean add(Collection new_links) {
      boolean stop = false; // Parada forzosa
      %Target last_link_i = null;
      %Target last_link_ii = null;
      Iterator i;

      synchronized (this) {
        if (new_links != null) {
          // Sobre una copia de new_links se eliminan aquellos objetos
          // %Target que tengan establecido el enlace previamente
          HashSet copied_new_links = new HashSet (new_links);
          i = new_links.iterator();
          while (i.hasNext()) {
            try {
              last_link_i = (%Target) i.next();
            } catch (ClassCastException e) {
              // Si algún elemento no es de tipo %Target no se hace nada
              return false;
            }
            if (links_hs.contains(last_link_i)) {
```

```
            copied_new_links.remove(last_link_i);
          }
        }

        // Si añadir los nuevos elementos no excede de la multiplicidad máxima
        if ((MAX == MANY) ||
            ((copied_new_links.size() + links_hs.size()) <= MAX)) {
          // por cada elemento de copied_new_links se establecerá enlace
          i = copied_new_links.iterator();
          while (i.hasNext() && (! stop)) {
            last_link_i = (%Target) i.next();
            // de %Source a %Target y de
            links_hs.add(last_link_i);
            // de %Target a %Source
            if (! last_link_i.%sourceRole.add(%Source.this)) {
              stop = true;
            }
          }

          // Si la parada es forzosa <=> algún enlace no se establece
          if (stop) {
            // se borran los nuevos enlaces hasta ahora establecidos
            i = copied_new_links.iterator();
            last_link_ii = (%Target) i.next();
            while (last_link_i != last_link_ii) {
              // de %Source a %Target
              links_hs.remove(last_link_ii);
              // y de %Target a %Source
              last_link_ii.%sourceRole.remove(%Source.this);
              last_link_ii = (%Target) i.next();
            }
          // Si no es forzosa, se termina con éxito
          } else {
            return true;
          }
        }
      }
      return false;
    }
  }

  /** Elimina todos los enlaces establecidos en links_hs
   *  @return 0 si se eliminan todos los enlaces y %Source y las instancias
   *  de links_hs siguen cumpliendo con las restricciones de la asociación,
   *  1 si %Source deja de cumplir con las restricciones, 2 si al menos un
   *  elemento de links_hs deja de cumplirlas y 3 si ambos (%Source y al
   *  menos un elemento de links_hs)
   */
  %visibility int remove() {
    boolean invalid_old_link = false;
    synchronized (this) {
      // Por cada elemento (last_link) de links_hs
      Iterator i = ((HashSet) links_hs.clone()).iterator();
      while (i.hasNext()) {
        %Target last_link = (%Target) i.next();
        // se elimina el enlace de %Source a last_link
        links_hs.remove(last_link);
        // y de last_link a %Source
        if (last_link.%sourceRole.remove(%Source.this) == 1) {
          invalid_old_link = true;
        }
      }
```

```
        // Devuelve el valor correcto
        if ((MIN > 0) && invalid_old_link) {
          return 3;
        } else if (invalid_old_link) {
          return 2;
        } else if (MIN > 0) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si se deja de cumplir con ellas en
     *  %Source, 2 si en %targetRole, 3 si en ambos y -1 si no existe el enlace
     *  con %targetRole u old_link es null
     */
    %visibility int remove(%Target old_link) {
      synchronized (this) {
        boolean invalid_old_link = false;
        if ((old_link != null) && test(old_link)) {
          // Elimina el enlace de %Source a %Target
          links_hs.remove(old_link);
          // Si la solicitud de eliminar el enlace viene de otro objeto que no
          // sea el viejo enlace
          if  ((old_link.%sourceRole.test(%Source.this)) &&
               // se llama a old_link para que elimine su enlace con %Source
               (old_link.%sourceRole.remove(%Source.this) == 1)) {
            invalid_link = true;
          }
          // Devuelve el valor correcto
          if ((links_hs.size() < MIN) && (invalid_old_link)) {
            return 3;
          } else if (invalid_old_link) {
            return 2;
          } else if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }

    /** Borra los enlaces establecidos con un conjunto de objetos %Target
     *  @param old_links Colección de objetos %Target cuyo enlace con
     *  %Source se quiere borrar
     *  @return 0 si se eliminan los enlaces solicitados y se sigue cumpliendo
     *  con las restricciones de la asociación, 1 si %Source deja de cumplir con
     *  las restricciones, 2 si al menos un elemento de old_links deja de cumplir
     *  y 3 si ambos (%Source y al menos un elemento de old_links_hs). -1 si
     *  algún elemento de old_links no es de tipo %Target u old_links es null y
     *  por tanto, no se hace nada
     */
    %visibility int remove(Collection old_links) {
      boolean invalid_old_link = false;
      %Target last_link = null;
      Iterator i;

      if (old_links == null) {
```

```
        return -1;
      }
      // Se comprueba que todos los elementos de old_links son de tipo %Target
      i = old_links.iterator();
      while (i.hasNext()) {
        try {
          last_link = (%Target) i.next();
        } catch (ClassCastException e) {
          return -1;
        }
      }

      synchronized (this) {
        // Se borran los enlaces
        i = old_links.iterator();
        while (i.hasNext()) {
          last_link = (%Target) i.next();
          // de %Source a last_link
          links_hs.remove(last_link);
          // y de last_link a %Source
          if (last_link.%sourceRole.remove(%Source.this) == 1) {
            invalid_old_link = true;
          }
        }

        // Devuelve el valor correcto
        if ((links_hs.size() < MIN) && (invalid_old_link)) {
          return 3;
        } else if (invalid_old_link) {
          return 2;
        } else if (links_hs.size() < MIN) {
          return 1;
        }
        return 0;
      }
    }

  }
  /** !Fin de código generado por JUMLA */

}
```

'EmptyClass.java~'

```
package %package;

public class %Source {

}
```

ANEXO E: PLANTILLAS DE CÓDIGO ALTERNATIVAS

Las plantillas de código que hemos presentado en el anexo anterior no son capaces de funcionar con asociaciones bidireccionales con visibilidad privada (o protegida) – privada (o protegida). Además, con las asociaciones bidireccionales que mezclan visibilidad privada o protegida en un extremo y pública o de paquete en el otro se tiene la restricción de que sólo desde el lado privado se puede establecer y eliminar enlaces.

Las plantillas de código que se muestran a continuación, son alternativas a '*SingleBidirectional.java~*' y a '*MultipleBidirectional.java~*' para que sí se puedan aplicar correctamente todas las combinaciones con visibilidades privadas o protegidas en las asociaciones bidireccionales. Tienen el inconveniente de que obligan a que la clase '*Association*', el atributo correspondiente que la instancia y algunos métodos a sean declarados públicos forzosamente, lo cual no es correcto desde el punto de vista teórico, puesto que deberían ser según se declare la visibilidad del extremo de asociación. No obstante, esta alternativa sí es funcional y pese a que permite que se pueda conocer siempre todas las asociaciones de las clases (independientemente de su visibilidad) no permite su uso más que dentro del conjunto de las clases que pertenecen a la asociación.

'SingleBidirectionalAlternative.java~'

```
package %package;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  public final %Source%Target_SingleBidirectional_%AssociationKind
         %targetRole = new %Source%Target_SingleBidirectional_%AssociationKind();

  public class %Source%Target_SingleBidirectional_%AssociationKind {

    /** Para representar multiplicidad muchos (* en UML)
     *  No se utiliza aquí pero se pone por uniformidad de interfaz con la
     *  implementación de otro tipo de asociaciones
     */
    %visibility final static int MANY = -1;

    /** Enlace de %Source a %Target obligatorio (true) u opcional (false) */
    private final static boolean mandatory = %mandatory;

    /** Enlace con %Target */
    private %Target link = null;

    /** Para poder indicar al otro extremo de la asociación si se le ha
     *  solicitado añadir un enlace invocando a addLink
     */
    private boolean addLink_required = false;

    /** Para poder indicar al otro extremo de la asociación si se le ha
     *  solicitado borrar un enlace invocando a removeLink
     */
    private boolean removeLink_required = false;

    /** Dice si la asociación es bidireccional o no
```

```
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return true;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return mandatory;
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return false;
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      if (isMandatory()) {
        return 1;
      }
      return 0;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY
     */
    %visibility long getMAX() {
      return 1;
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((! mandatory) || (link != null));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      if (link == null) {
        return 0;
      }
      return 1;
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     */
    %visibility boolean test(%Target query_link) {
      return link == query_link;
    }
```

```
/** Devuelve el enlace %Target contenido en link
 *  Lanza la excepción UnsetAssociationException si la asociación no
 *  cumple con los requisitos de la asociación
 *  @return link
 */
%visibility %Target get() throws UnsetAssociationException {
  if (! isValid()) {
    throw new UnsetAssociationException ("El objeto de clase %Source"
    + " no cumple con los requisitos de la asociación");
  }
  return link;
}

/** NO UTILIZAR ESTE MÉTODO NUNCA
 *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
 *  Dice si está solicitando una operación de addLink con alguna
 *  instancia de %Target
 *  @return true si sí
 */
public boolean is_addLink_required() {
  return addLink_required;
}

/** NO UTILIZAR ESTE MÉTODO NUNCA
 *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
 *  Añade un enlace en un único sentido. Sólo funciona cuando es llamado
 *  desde dentro de new_link.%sourceRole.set y sirve para
 *  poder conseguir un doble enlace cuando se llama a dicho método
 *  @param new_link Objeto con el que se intenta enlazar
 *  @return true si se realiza el enlace, false si no se puede debido a que
 *  ya tiene otro enlace distinto, new_link es nulo o no existe el
 *  enlace de new_link a %Source (normalmente pore la llamada se
 *  produce desde un sitio no permitido)
 */
public boolean addLink(%Target new_link) {
  synchronized (this) {
    // Si se puede establecer el enlace de %Source a %Target
    if ((new_link != null) && (link == null) &&
        // y es new_link quién ha invocado este método
        (new_link.%sourceRole.is_addLink_required())) {
      // se establece el enlace
      link = new_link;
      return true;
    }
    return false;
  }
}

/** Añade un enlace con una instancia de %Target
 *  @param new_link Objeto con el que se intenta enlazar
 *  @return true si se establece el enlace, false si no se puede debido a
 *  que ya tiene otro enlace distinto, new_link no lo permite o
 *  new_link es nulo
 */
%visibility boolean add(%Target new_link) {
  synchronized (this) {
    addLink_required = true;
    // Si %Source puede establecer el enlace y
    if ((new_link != null) && (link == null) &&
        // se consigue establecer enlace de %Target a %Source
        (new_link.%sourceRole.addLink(%Source.this))) {
```

```
          // se establece de %Source a %Target
          link = new_link;
        }
        addLink_required = false;
        return (link == new_link);
      }
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Dice si está solicitando una operación de removeLink con alguna
     *  instancia de %Target
     *  @return true si sí
     */
    public boolean is_removeLink_required() {
      return removeLink_required;
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Rompe un enlace en un único sentido. Sólo funciona cuando es llamado
     *  desde dentro de old_link.%association_%Target.unset y sirve para
     *  poder eliminar el doble enlace cuando se llama a dicho método
     *  @param old_link Objeto con el que se intenta romper el enlace
     *  @return 1 si al finalizar el método no se cumplen con las restricciones
     *  de multiplicidad en la instancia %Source y 0 si sí. -1 si no se hace
     *  ningún cambio
     */
    public int removeLink(%Target old_link) {
      synchronized (this) {
        // Si existe enlace con old_link y
        if ((old_link != null) && (test(old_link)) &&
            // es old_link quién ha invocado a este método
            (old_link.%sourceRole.is_removeLink_required())) {
          // se elimina el enlace de %Source a %Target
          link = null;
          if (isValid()) {
            return 0;
          }
          return 1;
        }
        return -1;
      }
    }

    /** Elimina el enlace establecido en link
     *  @return 0 si se elimina el enlace y los objetos %Source y link
     *  siguen cumpliendo con las restricciones de multiplicidad, 1 si %Source
     *  deja de cumplir con las restricciones, 2 si link deja de cumplir
     *  y 3 si ambos
     */
    %visibility int remove() {
      synchronized (this) {
        boolean invalid_link = false;
        if (link != null) {
          // Se elimina el enlace de %Target a %Source
          removeLink_required = true;
          if (link.%sourceRole.removeLink(%Source.this) == 1) {
            invalid_link = true;
          }
          removeLink_required = false;
          // Se elimina el enlace de %Source a %Target
```

```
          link = null;
        }
        // Devuelve el valor correcto
        if (mandatory && invalid_link) {
          return 3;
        } else if (invalid_link) {
          return 2;
        } else if (mandatory) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra el enlace supuestamente establecido con un objeto %Target
     *  @param old_link Supuesto enlace de tipo %Target
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de multiplicidad, 1 si se deja de cumplir con las
     *  restricciones y -1 si no existe el enlace con link y por tanto,
     *  no se hace nada
     */
    %visibility int remove(%Target old_link) {
      if (test(old_link)) {
        return remove();
      }
      return -1;
    }

  }

}
```

'MultipleBidirectionalAlternative.java~'

```
package %package;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class %Source {

  /** Atributo por el cual se gestiona la asociación */
  public final %Source%Target_MultipleBidirectional_%AssociationKind
       %targetRole = new %Source%Target_MultipleBidirectional_%AssociationKind();

  public class %Source%Target_MultipleBidirectional_%AssociationKind {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = %min_multiplicity;

    /** Para representar multiplicidad muchos (* en UML) */
    %visibility final static int MANY = -1;
    /** Multiplicidad máxima del enlace */
    private final static long MAX = %max_multiplicity;

    /** HashSet donde se almacenan los enlaces con instancias %Target */
    private HashSet links_hs = new HashSet();
```

```
    /** Para poder indicar al otro extremo de la asociación si se le ha
     *  solicitado añadir un enlace invocando a addLink
     */
    private boolean addLink_required = false;

    /** Para poder indicar al otro extremo de la asociación si se le ha
     *  solicitado borrar un enlace invocando a removeLink
     */
    private boolean removeLink_required = false;

    /** Dice si la asociación es bidireccional o no
     *  @return true si la asociación es bidireccional, false si unidireccional
     */
    %visibility boolean isBidirectional() {
      return true;
    }

    /** Dice si la asociación es obligatoria o no
     *  @return true si la asociación es obligatoria
     */
    %visibility boolean isMandatory() {
      return (MIN > 0);
    }

    /** Dice si la asociación es múltiple o no
     *  @return true si es múltiple, false si sencilla
     */
    %visibility boolean isMultiple() {
      return ((MAX > 1) || (MAX == MANY));
    }

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    %visibility long getMIN() {
      return MIN;
    }

    /** Devuelve la multiplicidad máxima del final de la asociación
     *  @return multiplicidad máxima. Si se obtiene algún valor extraño, hay que
     *  comprobar si éste se corresponde con MANY
     */
    %visibility long getMAX() {
      return MAX;
    }

    /** Dice si %Source cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
     */
    %visibility boolean isValid() {
      return ((links_hs.size() >= MIN) &&
              ((links_hs.size() <= MAX) || (MAX == MANY)));
    }

    /** Devuelve el número de enlaces activos que tiene el objeto
     *  @return Número de enlaces
     */
    %visibility long numberOfLinks() {
      return links_hs.size();
    }

    /** Comprueba si %Source está asociado con un objeto %Target concreto
```

```
     *  @param query_link Supuesto enlace %Target
     *  @return true si existe dicho enlace, false en caso contrario
     */
    %visibility boolean test(%Target query_link) {
      if (query_link == null) {
        return links_hs.isEmpty();
      }
      return links_hs.contains(query_link);
    }

    /** Comprueba si %Source está asociado con una colección de objetos %Target
     *  @param query_links Colección de supuestos enlaces
     *  @return true si todos los elementos de links_hs están enlazados
     *  con %Source o si links_hs está vacío y query_links también o es null
     */
    %visibility boolean test(Collection query_links) {
      if (query_links == null) {
        return links_hs.isEmpty();
      }
      return links_hs.containsAll(query_links);
    }

    /** Devuelve una colección con la copia de los enlaces con %Target
     *  contenidos en links_hs. Lanza la excepción UnsetAssociationException si
     *  la asociación no cumple con los requisitos de la asociación
     *  @return Copia de links_hs
     */
    %visibility Collection get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase %Source"
        + " no cumple con los requisitos de la asociación");
      }
      return (Collection) links_hs.clone();
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Dice si está solicitando una operación de addLink con alguna
     *  instancia de %Target
     *  @return true si sí
     */
    public boolean is_addLink_required() {
      return addLink_required;
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Añade un enlace en un único sentido. Sólo funciona cuando es llamado
     *  desde dentro de new_link.%sourceRole.set y sirve para
     *  poder conseguir un doble enlace cuando se llama a dicho método
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se realiza el enlace, false si no se puede debido a que
     *  ya tiene otro enlace distinto, new_link es nulo o no existe el
     *  enlace de new_link a %Source (normalmente pore la llamada se
     *  produce desde un sitio no permitido)
     */
    public boolean addLink(%Target new_link) {
      synchronized (this) {
        // Si se puede establecer el enlace de %Source a %Target
        if ((new_link != null) && ((MAX == MANY) || (links_hs.size() < MAX)) &&
            // y es new_link quién ha invocado este método
            (new_link.%sourceRole.is_addLink_required())) {
```

```
          // se establece el enlace
          links_hs.add(new_link);
          return true;
        }
        return false;
      }
    }

    /** Añade un enlace con una instancia de %Target
     *  @param new_link Objeto con el que se intenta enlazar
     *  @return true si se realiza el enlace, false si no se puede debido a
     *  que se excede la multiplicidad máxima
     */
    %visibility boolean add(%Target new_link) {
      synchronized (this) {
        // Si se intenta enlazar con un objeto que ya está enlazado, no se hace
        // nada pero se devuelve como si se hubiese establecido el enlace
        if (test(new_link)) {
          return true;
        }
        addLink_required = true;
        // Si añadir new_link no excede la multiplicidad máxima y
        if (((MAX == MANY) || (links_hs.size() < MAX)) &&
            // se consigue establecer enlace de %Target a %Source
            (new_link.%sourceRole.addLink(%Source.this))) {
          // se establece el enlace de %Source a %Target
          links_hs.add(new_link);
        }
        addLink_required = false;
        return (links_hs.contains(new_link));
      }
    }

    /** Añade un enlace con cada uno de los elementos de new_links_hs
     *  @param new_links Colección de elementos %Target con los que se
     *  intenta enlazar
     *  @return true si se establecen TODOS los enlaces, false si se excede de
     *  la multiplicidad máxima o alguno de los nuevos enlaces no se puede
     *  establecer
     */
    %visibility boolean add(Collection new_links) {
      boolean stop = false; // Parada forzosa
      %Target last_link_i = null;
      %Target last_link_ii = null;
      Iterator i;

      synchronized (this) {
        // Sobre una copia de new_links se eliminan aquellos objetos
        // %Target que tengan establecido el enlace previamente
        HashSet copied_new_links = new HashSet (new_links);
        i = new_links.iterator();
        while (i.hasNext()) {
          try {
            last_link_i = (%Target) i.next();
          } catch (ClassCastException e) {
            // Si algún elemento no es de tipo %Target no se hace nada
            return false;
          }
          if (links_hs.contains(last_link_i)) {
            copied_new_links.remove(last_link_i);
          }
        }
```

```
        // Si añadir los nuevos elementos no excede de la multiplicidad máxima
        if ((MAX == MANY) ||
            ((copied_new_links.size() + links_hs.size()) <= MAX)) {
          addLink_required = true;
          // por cada elemento de copied_new_links se establecerá enlace
          i = copied_new_links.iterator();
          while (i.hasNext() && (! stop)) {
            last_link_i = (%Target) i.next();
            // de %Target a %Source
            if (! last_link_i.%sourceRole.addLink(%Source.this)) {
              stop = true;
            }
          }
          addLink_required = false;

          // Si la parada es forzosa <=> falla el establecimiento de algún enlace
          if (stop) {
            // se borran los nuevos enlaces hasta ahora establecidos
            // de %Target a %Source
            removeLink_required = true;
            i = copied_new_links.iterator();
            last_link_ii = (%Target) i.next();
            while (last_link_i != last_link_ii) {
              last_link_ii.%sourceRole.removeLink(%Source.this);
              last_link_ii = (%Target) i.next();
            }
            removeLink_required = false;
          // Si no es forzosa, se añaden los enlaces de %Source a %Target
          } else {
            links_hs.addAll(copied_new_links);
            return true;
          }
        }
        return false;
      }
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Dice si está solicitando una operación de removeLink con alguna
     *  instancia de %Target
     *  @return true si sí
     */
    public boolean is_removeLink_required() {
      return removeLink_required;
    }

    /** NO UTILIZAR ESTE MÉTODO NUNCA
     *  ES DE CARÁCTER INTERNO A LA GESTIÓN DE LA ASOCIACIÓN
     *  Rompe un enlace en un único sentido. Sólo funciona cuando es llamado
     *  desde dentro de old_link.%association_%Target.unset y sirve para
     *  poder eliminar el doble enlace cuando se llama a dicho método
     *  @param old_link Objeto con el que se intenta romper el enlace
     *  @return 1 si al finalizar el método no se cumplen con las restricciones
     *  de multiplicidad en la instancia %Source y 0 si sí. -1 si no se hace
     *  ningún cambio
     */
    public int removeLink(%Target old_link) {
      synchronized (this) {
        // Si existe enlace con old_link y
        if ((old_link != null) && (test(old_link)) &&
```

```
            // es old_link quién ha invocado a este método
            (old_link.%sourceRole.is_removeLink_required())) {
          // se elimina el enlace de %Source a %Target
          links_hs.remove(old_link);
          if (isValid()) {
            return 0;
          }
          return 1;
        }
        return -1;
      }
    }

    /** Elimina todos los enlaces establecidos en links_hs
     *  @return 0 si se eliminan todos los enlaces y %Source y las instancias
     *  de links_hs siguen cumpliendo con las restricciones de la asociación,
     *  1 si %Source deja de cumplir con las restricciones, 2 si al menos un
     *  elemento de links_hs deja de cumplirlas y 3 si ambos (%Source y al
     *  menos un elemento de links_hs)
     */
    %visibility int remove() {
      boolean invalid_old_link = false;
      synchronized (this) {
        // Por cada elemento de links_hs
        removeLink_required = true;
        Iterator i = ((HashSet) links_hs.clone()).iterator();
        while (i.hasNext()) {
          %Target last_link = (%Target) i.next();
          // se elimina el enlace de last_link a %Source
          if (last_link.%sourceRole.removeLink(%Source.this) == 1) {
            invalid_old_link = true;
          }
        }
        removeLink_required = false;
        // Elimina todos los enlaces en %source
        links_hs.clear();
        // Devuelve el valor correcto
        if ((MIN > 0) && (invalid_old_link)) {
          return 3;
        } else if (invalid_old_link) {
          return 2;
        } else if (MIN > 0) {
          return 1;
        }
        return 0;
      }
    }

    /** Borra un enlace establecido con un objeto %Target
     *  @param old_link Enlace de tipo %Target a borrar
     *  @return 0 si se elimina el enlace y se sigue cumpliendo con las
     *  restricciones de la asociación, 1 si se deja de cumplir con ellas en
     *  %Source, 2 si en %targetRole, 3 si en ambos y -1 si no existe el enlace
     *  con %targetRole
     */
    %visibility int remove(%Target old_link) {
      boolean invalid_old_link = false;
      synchronized (this) {
        if (test(old_link)) {
          // Elimina el enlace de %Target a %Source
          removeLink_required = true;
          if (old_link.%sourceRole.removeLink(%Source.this) == 1) {
```

```
            invalid_old_link = true;
          }
          removeLink_required = false;
          // y de %Source a %Target
          links_hs.remove(old_link);
          // Devuelve el valor correcto
          if ((links_hs.size() < MIN) && (invalid_old_link)) {
            return 3;
          } else if (invalid_old_link) {
            return 2;
          } else if (links_hs.size() < MIN) {
            return 1;
          }
          return 0;
        }
        return -1;
      }
    }

    /** Borra los enlaces establecidos con un conjunto de objetos %Target
     *  @param old_links Colección de objetos %Target cuyo enlace con
     *  %Source se quiere borrar
     *  @return 0 si se eliminan los enlaces solicitados y se sigue cumpliendo
     *  con las restricciones de la asociación, 1 si %Source deja de cumplir con
     *  las restricciones, 2 si al menos un elemento de old_links deja
     *  de cumplir y 3 si ambos (%Source y al menos un elemento de
     *  old_links_hs). -1 si algún elemento de old_links no es de tipo %Target y
     *  por tanto, no se hace nada
     */
    %visibility int remove(Collection old_links) {
      boolean invalid_old_link = false;
      %Target last_link = null;
      Iterator i;

      synchronized (this) {
        // Se comprueba que todos los elementos de old_links son de tipo %Target
        i = old_links.iterator();
        while (i.hasNext()) {
          try {
            last_link = (%Target) i.next();
          } catch (ClassCastException e) {
            return -1;
          }
        }

        // Se borran los enlaces de %Target a %Source
        removeLink_required = true;
        i = old_links.iterator();
        while (i.hasNext()) {
          if (((%Target) i.next()).%sourceRole.removeLink(%Source.this) == 1) {
            invalid_old_link = true;
          }
        }
        removeLink_required = false;

        // Se eliminan los enlaces de %Source a %Target
        links_hs.removeAll(old_links);

        // Devuelve el valor correcto
        if ((links_hs.size() < MIN) && (invalid_old_link)) {
          return 3;
        } else if (invalid_old_link) {
```

```
          return 2;
        } else if (links_hs.size() < MIN) {
          return 1;
        }
      }
      return 0;
    }

  }

}
```

ANEXO F: CONTENIDO DEL CD

Readme	Fichero de información con el contenido del CD.
Manual de usuario.doc	Manual de la herramienta *JUMLA*.
PFC.doc	Documento que se presenta como Proyecto Fin de Carrera.
JUMLA	Aplicación *JUMLA*.
JUMLA source	Código fuente de *JUMLA*.
UML_ASSOCIATIONS	Aplicación *UML_ASSOCIATIONS*.
UML_ASSOCIATIONS source	Código fuente de *UML_ASSOCIATIONS*.
lib	Librerías "XMIFramework" y "Xerces" con toda la documentación que las acompaña.
xmiframework	
xerces-2_0_1	
doc	Documentos adicionales:
classify.ps	• *"Classifying Relationships Between Object-Oriented Design Patterns"* (James Noble).
rels97.ps	• *"Basic Relationship Patterns"* (James Noble).
towards.ps	• *"Towards a Pattern Language for Object Oriented Design"* (James Noble).
SWAmbler	• Contiene cuatro documentos de Scott W. Ambler que marca unas pautas para la implementación de asociaciones *UML* en *Java* de una forma sencilla y rápida [8] [9] [10] [11].
OMG UML Specif....pdf	• *"OMG UML Specification Version 1-4 September 200"*.
OMG XML Metad....pdf	• *"OMG XML Metadata Interchange (XMI) Specification"*.
xmiwhite0399.pdf	• *"XMI Opens Application Interchange"* (S. Brodsky).
j2re-1_4_0	Versión 1.4.0 de *Java*™ *2 Runtime Environment* para poder ejecutar aplicaciones *Java*. Contiene dos subdirectorios: uno para plataformas *Windows* y otro para *Linux*.

BIBLIOGRAFÍA

[1] *"Verificación de la Implementación de Elementos UML en Java"*
Belén Criado Sánchez
Proyecto Fin de Carrera, Ingeniería Superior en Informática
Universidad Carlos III de Madrid, 2001

[2] *"El lenguaje de unificado de modelado"*
Grady Booch, James Rumbaugh e Ivar Jacobson
Addison Wesley Iberoamericana, Madrid, 1999

[3] *"Unified Modeling Language Specification"*, version 1.4
Object Management Group, septiembre de 2001

[4] *"Mapping UML Associations into Java code"*
Gonzalo Génova, Carlos Ruiz del Castillo, Juan Lloréns
Universidad Carlos III de Madrid

[5] *"Basic relationship patterns"*
James Noble (kjx@mri.mq.edu.au)
http://www.mri.mq.edu.au/~kjx/
Macquarie University, Sydney, 1997

[6] *"Relations as Semantic Constructs in an Object-Oriented Language"*
James Rumbaugh
OOPSLA'87, Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications: 466-481, Orlando, Florida, Oct. 4-8 (1987)

[7] *"Models for Design: Generating Code for Associations"*
James Rumbaugh
Journal of Object Oriented Programming, 8(9):13-17, February 1996

[8] *"An overview of object relationships"*
Scott W. Ambler (scott.ambler@ronin-intl.com)
http://www-106.ibm.com/developerworks/library/tip-objrel/
Ronin International, 2001

[9] *"Unidirectional object relationships"*
Scott W. Ambler (scott.ambler@ronin-intl.com)
http://www-106.ibm.com/developerworks/library/tip-implor/
Ronin International, 2001

[10] *"Implementing one-to-many object relationships"*
Scott W. Ambler (scott.ambler@ronin-intl.com)
http://www-106.ibm.com/developersworks/library/tip-objrel3/
Ronin International, 2001

[11] *"Implementing many-to-many object relationships"*
Scott W. Ambler (scott.ambler@ronin-intl.com)
http://www-106.ibm.com/developerworks/library/tip-objrel4/
Ronin International, 2001

[12] *"The Java Programming Language"*
Ken Arnold, James Gosling, David Holmes
Addison-Wesley, 3rd edition, 2000

[13] *"Thinking in Java"*
Bruce Eckel
http://www.bruceeckel.com (2nd edition, revision 10, 2000)

[14] *"XMI Specification Press Kit"*
OMG's XMI Audio Conference
http://www.omg.org/news/pr99/xmi_overview.html

[15] *"XMI Open Application Interchange"*
S. Brodsky (sbrodsky@us.ibm.com)
http://www.omg.org/docs/omg/99-04-04.pdf

[16] *"MagicDraw UML, User's Manual"*
Version 5.0.
http://www.magicdraw.com

[17] Documentación de *XMIFrameWork* 1.0
AlphaWorks, March 2001
http://www.alphaworks.ibm.com/aw.nsf/download/xmiframework

[18] "*Design Patterns. Elements of reusable Object-Oriented software*"
Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides
Addison-Wesley, 1994

[19] "*Patterns in Java, volume 1, a catalogue of reusable design patterns illustrated with UML*"
Mark Grand
Wiley, 1999