UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Departamento de Informática

Tesis Doctoral
Programa de Doctorado en Ingeniería Informática

Entrelazamiento de los aspectos estático y dinámico en las asociaciones UML

AUTOR: Gonzalo Génova Fuster
DIRECTOR: Dr. Juan Llorens Morillo

Leganés, Mayo de 2003

A Charo, mi amor,
y a nuestros hijos (0..).*

Agradecimientos

A Juan Llorens, director de esta Tesis Doctoral
y amable consejero en muchos otros asuntos.

A todos los componentes del grupo de investigación
Information Engineering de la Universidad Carlos III de Madrid,
especialmente a José Miguel "JM", Víctor, Jorge, Vicente y Carlos.

A Perdita Stevens, que me acogió como investigador visitante en el
Laboratory for Foundations of Computer Science (LFCS),
parte de la *Division of Informatics* de la Universidad de Edimburgo,
en los meses de febrero-abril de 2002,
donde maduraron buena parte de los resultados de este trabajo.

A Guy Genilloud y Joaquin Miller, que junto con Perdita Stevens
han sido importantes interlocutores de estas ideas.

A mis padres, no es necesario decir por qué.

Y a esas tres personas -ellas saben quiénes son-
que han inspirado este trabajo desde su principio hasta su fin
y conocen mejor que nadie a su autor.

El mundo es la totalidad de los hechos.
Un hecho atómico es una combinación de objetos (entidades, cosas).

Die Welt ist die Gesamtheit der Tatsachen.
Der Sachverhalt ist eine Verbindung von Gegenständen. (Sachen, Dingen.)

Ludwig Wittgenstein
Tractatus Logico-Philosophicus, *1.1 y 2.01*

Índice

PRIMERA PARTE: PLANTEAMIENTO 4
1. Introducción 5
1.1. Presentación del Lenguaje Unificado de Modelado 5
1.2. La falta de precisión de la semántica de UML 5
1.3. La definición de relación y asociación en UML 6
1.4. Aspectos que deben ser clarificados 6
1.4.1. El nombrado de asociaciones 7
1.4.2. Concepción estática y dinámica de las asociaciones 8
1.4.3. Instanciación de asociaciones 10
1.4.4. La multiplicidad 12
1.4.5. Direccionalidad y navegabilidad 13
1.4.6. Dependencias y asociaciones 13
1.4.7. Visibilidad e interfaces 14
1.4.8. Implementación de asociaciones 15
1.5. Objetivos y método 16
1.6. Resultados y aplicaciones previstos 17
1.7. Experimentación y validación 18
1.8. Estructura de esta Tesis Doctoral 19
2. Estado de la cuestión 20
2.1. Breve historia de UML 20
2.1.1. Precursores de UML 20
2.1.2. Booch, Rumbaugh y Jacobson unen sus fuerzas 21
2.1.3. Los colaboradores de UML 22
2.1.4. La estandarización de UML 22
2.2. El estándar de UML: notación y semántica 23
2.3. La notación de las asociaciones 27
2.3.1. Clases y objetos 27
2.3.2. Asociación binaria 29
2.3.3. Asociación reflexiva 34
2.3.4. Agregación y composición 35
2.3.5. Asociación cualificada 37
2.3.6. Clase-asociación 37
2.3.7. Asociación n-aria 38
2.4. La semántica de las asociaciones 39
2.4.1. Clases, atributos y operaciones 39
2.4.2. Tipos de relaciones 41
2.4.3. La metaclase `Association` 42
2.4.4. La metaclase `AssociationEnd` 43
2.4.5. La metaclase `AssociationClass` 47

SEGUNDA PARTE: DESARROLLO TEÓRICO 48
3. La multiplicidad de las asociaciones 49
3.1. Introducción 49

3.2. Definiciones de multiplicidad ... 51
3.2.1. Definición de multiplicidad en UML ... 51
3.2.2. Definición de multiplicidad en otras técnicas de modelado de datos ... 53
3.3. La multiplicidad de las asociaciones ternarias ... 55
3.3.1. Paradojas y ambigüedades de las multiplicidades ternarias ... 55
3.3.2. La necesidad de expresar la restricción de participación ... 62
3.3.3. Propuesta sobre asociaciones n-arias para el Estándar de UML ... 68
3.4. La multiplicidad de las asociaciones binarias especiales ... 69
3.4.1. La multiplicidad de la asociación cualificada ... 69
3.4.2. La multiplicidad de la clase-asociación ... 72
3.5. Las instancias de la asociación: ¿tuplas no repetidas? ... 74
3.5.1. La asociación, ¿conjunto o saco de tuplas? ... 75
3.5.2. En enlace, ¿tupla o elemento? ... 79
4. La navegabilidad de las asociaciones ... 84
4.1. Introducción ... 84
4.2. Definición de navegabilidad ... 86
4.2.1. Traversabilidad, accesibilidad ... 86
4.2.2. Expresiones de navegación ... 87
4.3. Algunos conceptos relacionados ... 88
4.3.1. Dirección del nombre ... 88
4.3.2. Visibilidad ... 91
4.4. El envío de mensajes ... 94
4.4.1. Ruta navegable y operación visible ... 94
4.4.2. Representación de los mensajes en los diagramas de UML ... 96
4.4.3. ¿Es todo enlace de comunicación una instancia de una asociación? ... 99
4.5. Asociaciones estructurales y contextuales ... 104
4.5.1. Asociaciones estáticas y dinámicas ... 104
4.5.2. El contexto de las asociaciones ... 107
4.5.3. Propuesta sobre estereotipos de asociaciones para el Estándar de UML ... 111
4.5.4. Representación de asociaciones contextuales ... 113
4.6. Propiedades de la navegabilidad ... 119
4.6.1. Dependencia ... 119
4.6.2. Invertibilidad y bidireccionalidad ... 121
4.6.3. Eficiencia ... 125
4.7. Notación de la navegabilidad ... 127
4.8. La navegabilidad en las asociaciones más complejas ... 129
4.8.1. Expresiones de navegación ... 129
4.8.2. Envío de mensajes ... 134
5. La visibilidad de las asociaciones ... 137
5.1. Introducción ... 137
5.2. Definición de visibilidad ... 139
5.2.1. La visibilidad de atributos y operaciones ... 139
5.2.2. La visibilidad de los extremos de asociación ... 146
5.2.3. Asociaciones bidireccionales entre clases de distintos paquetes ... 148
5.3. Otras formas de especificar la interfaz de una asociación ... 152
5.3.1. Especificadores de interfaz ... 153
5.3.2. Interfaces ... 155
5.3.3. Comparación ... 158
5.4. Solución propuesta: asociaciones entre interfaces ... 161
5.4.1. Notación ... 162
5.4.2. Metamodelo ... 165
5.4.3. Simplificación de la visibilidad de atributos y operaciones ... 165
5.4.4. Interfaces con múltiples asociaciones ... 167

TERCERA PARTE: EXPERIMENTACIÓN 169
6. La implementación de las asociaciones 170
6.1. Introducción 170
6.2. Experimentación con la multiplicidad 171
6.2.1. Asociaciones opcionales y obligatorias 172
6.2.2. Asociaciones sencillas y múltiples 174
6.3. Experimentación con la navegabilidad 176
6.3.1. Asociaciones unidireccionales 178
6.3.2. Asociaciones bidireccionales 180
6.4. Experimentación con la visibilidad 184
6.5. Descripción del código que implementa la solución 188
6.5.1. Métodos lectores 189
6.5.2. Métodos mutadores 189
6.5.3. Métodos auxiliares para averiguar el estado de la asociación 191
6.5.4. Métodos auxiliares para averiguar la definición de la asociación 191
6.6. La herramienta de generación de código 192
6.7. Comparación de resultados 194
6.7.1. Código generado por *Rational Rose* 195
6.7.2. Código generado por *MEGA* 196
6.7.3. Código generado por *JUMLA* 198
6.7.4. Valoración final 206

CUARTA PARTE: CONCLUSIONES Y REFERENCIAS 207
7. Conclusiones 208
7.1. Aportaciones originales 208
7.1.1. ¿Qué es una asociación? 208
7.1.2. La multiplicidad de las asociaciones 209
7.1.3. La navegabilidad de las asociaciones 212
7.1.4. La visibilidad de las asociaciones 215
7.1.5. La implementación de las asociaciones 217
7.2. Propuestas de modificación del Estándar de UML 219
7.2.1. Sobre la multiplicidad de las asociaciones 219
7.2.2. Sobre la navegabilidad de las asociaciones 219
7.2.3. Sobre la visibilidad de las asociaciones 219
7.3. Otros resultados obtenidos 220
7.4. Trabajos futuros 221
8. Referencias 224

PRIMERA PARTE:

PLANTEAMIENTO

1. Introducción

1.1. Presentación del Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado (*Unified Modeling Language - UML*) [UML] es un lenguaje visual de modelado de propósito general utilizado para especificar, visualizar, construir y documentar los artefactos (piezas de información) de un sistema informático. UML es el lenguaje estándar de modelado del OMG (*Object Management Group*) [OMG] para la especificación y diseño de sistemas informáticos complejos. El OMG es una iniciativa privada surgida en 1989, cuyo fin es la estandarización del software orientado a objetos.

En los últimos años UML ha tenido una difusión espectacular en el mundo de la industria y, consecuentemente, también en el de la docencia. La gran variedad de lenguajes de modelado existentes con anterioridad exigían desde hace tiempo un proceso de unificación, una *lingua franca* o lenguaje común que facilitase el entendimiento entre los distintos profesionales de la informática, desde los analistas de sistemas hasta los diseñadores y programadores.

1.2. La falta de precisión de la semántica de UML

Existen ya abundantes experiencias de calidad que avalan el uso de UML en la industria del software. No obstante, se han detectado numerosos problemas relativos a la falta de una semántica precisa, problemas que limitan su eficacia e implantación. Pese a dichos problemas, la difusión de UML ha sido en general muy beneficiosa, gracias al intento de una definición formal del lenguaje que, si bien incompleta, al menos proporciona un marco adecuado para la discusión.

El modelado de sistemas complejos requiere el uso de construcciones primitivas de modelado que proporcionen mecanismos de gestión de la complejidad y permitan la temprana detección de errores en los requerimientos y los diseños. El principio de la *separación de vistas* ha demostrado ser un medio eficaz para controlar la complejidad, y está bien soportado por UML. No obstante, no ocurre lo mismo con el principio de *formalidad y rigor* que facilita la detección de errores, como se ha demostrado en los congresos internacionales dedicados al lenguaje [UMLConf] y en la serie de *Workshops* promovidos por el *Precise UML Group* [pUML]. Esta falta de formalidad y rigor hace que, con frecuencia, la

relación entre los distintos elementos del lenguaje, así como su implementación en distintos lenguajes de programación, no sean bien comprendidos. Y, como es bien sabido, no es posible manejar bien aquello que no se comprende bien.

Así pues, el desarrollo de una semántica de UML clara, comprensible y concisa (*clear, clean, concise*) [cUML], que permita un análisis riguroso, efectivo y soportado por herramientas CASE, facilitará su aplicación al modelado de sistemas complejos.

1.3. La definición de relación y asociación en UML

En UML se define la "relación" (*relationship*) como una "conexión semántica entre elementos del modelo" [UML, p. B-16]. Este concepto engloba diversos tipos de relación, tales como asociación, generalización, flujo, y diversos tipos de dependencia. La expresión "conexión semántica" suele ser particularmente oscura para quien se adentra en el mundo del modelado. "Semántica" es la parte de la lingüística que estudia la significación de las palabras, de modo que "conexión semántica" podría ser traducida por "conexión significativa", aunque esto no acaba de aclarar la vaguedad de la expresión. El concepto de relación no se entiende hasta que no se profundiza en el significado de cada uno de los distintos tipos de relación, es decir, qué propiedades tienen y cómo se usan para construir los modelos.

A su vez, la "asociación" (*association*) se define como una "relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases" [UML, p. B-3]. Brevemente, una *instancia*, a menudo usado como sinónimo de *objeto*, es una "entidad que tiene identidad única, un conjunto de operaciones que se le pueden aplicar, y un estado que almacena los efectos de las operaciones" [UML, p. B-11], y una *clase* es la "descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica" [UML, p. B-5]. En un sistema informático orientado a objetos, *los objetos existentes en un momento dado están conectados entre sí, y estas conexiones son descritas en el nivel abstracto de las clases mediante asociaciones*.

1.4. Aspectos que deben ser clarificados

Prácticamente todos y cada uno de los aspectos de la asociación en UML han sido cuestionados por uno u otro investigador a lo largo de los últimos años. Algunos de estos aspectos han sido aclarados en sucesivas versiones del estándar [UML11, UML13, UML], pero otros, entre ellos los más básicos, aún deben ser clarificados. Podemos señalar, por ejemplo:

1. La concepción estática de la asociación frente a la concepción dinámica, es decir, si la asociación expresa el estado de la

información del sistema (estructura de datos), si expresa el dinamismo de las interacciones dentro del sistema (intercambio de mensajes), o bien si, tal vez, representa ambos aspectos a la vez.

2. La "cosificación" de la asociación, que permite que cada instancia de una asociación (denominada "enlace") tenga identidad, propiedades y comportamiento propio.
3. La multiplicidad de la asociación, especialmente en el caso de asociaciones ternarias y de grado superior.
4. La navegabilidad de la asociación, sus implicaciones sobre la comunicación mediante mensajes.
5. La visibilidad de las asociaciones en relación con el concepto de "interfaz".
6. La implementación de las asociaciones en lenguajes de programación orientados a objetos.

A continuación vamos a exponer con más detalle algunas de las ambigüedades y problemas detectados en relación con el concepto de asociación en UML. El estudio pormenorizado de estos problemas es el objeto de la Segunda Parte de la Tesis.

1.4.1. El nombrado de asociaciones

Las asociaciones tienen dos tipos de nombres: nombre de la asociación en cuanto tal, situado en el centro de la misma, y nombres de rol, situados en cada uno de los extremos. El Estándar deja completa libertad al modelador para que escoja los nombres según su conveniencia. Elegir bien los nombres de los elementos de un modelo no es una tarea fácil, pero el esfuerzo se ve recompensado por un fuerte incremento en la legibilidad del modelo. Los métodos orientados a objetos suelen recomendar el empleo de nombres tomados del dominio para denominar los elementos del modelo [Rumbaugh 91]. En particular, los sustantivos del dominio, que designan entidades, pasarían a denominar clases, y los verbos del dominio, que designan interacciones, pasarían a denominar asociaciones; las formas verbales en participio serían adecuadas para denominar nombres de rol. En este sentido debe entenderse el pequeño triángulo negro junto al nombre de la asociación, que indica la "dirección en la que debe leerse el nombre" [UML, p. 3-69], es decir, quién es el sujeto y quién es el objeto de una determinada interacción representada mediante una asociación.

Esta técnica, no obstante, tiene graves limitaciones en la práctica:

- A menudo es preferible un sustantivo en lugar de un verbo para denominar una asociación, por ejemplo, la asociación `matrimonio` entre las clases `Hombre` y `Mujer`; en estos casos es muy difícil decir en qué consiste la "dirección del nombre". Esto es particularmente claro en el caso de una clase-asociación, que, de acuerdo con esta técnica, debería ser nombrada con sustantivo en cuanto que es una clase, y a la vez con un nombre verbal en cuanto

asociación. Por otra parte, cuando se trata de asociaciones n-arias, un triángulo de dirección resulta muy inadecuado para representar la ordenación de los extremos.

- Los verbos empleados en el dominio designan interacciones, pero las interacciones se modelan no sólo como asociaciones entre clases, sino también como operaciones dentro de una clase, que pueden ser invocadas por los objetos de las clases asociadas. Así, el mismo verbo denominaría tanto la asociación como la operación. Por ejemplo, el nombre de la asociación `contrata` entre las clases `Empresa` y `Persona` interfiere con el nombre de la operación `contratar`, definida en esta última clase (ver **Figura 1.1**). En realidad, aquí se está desvelando un problema más grave, y es que, a pesar de lo que propone la técnica, no está claro si las interacciones del dominio deben modelarse como asociaciones o como operaciones.

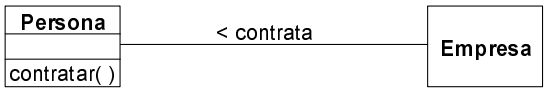

Figura 1.1. Conflicto de nombres entre asociación y operación

Los problemas detectados para nombrar asociaciones y nombres de rol conectan con el siguiente Apartado, en el que se plantean dos concepciones del significado de "asociación" que no son fáciles de coordinar bien.

1.4.2. Concepción estática y dinámica de las asociaciones

A lo largo de la última década se ha producido una intensa controversia desde que James Rumbaugh intentó reforzar el concepto de asociación en los modelos orientados a objetos tomando ideas del modelo Entidad/Relación [Chen 76]. En opinión de Rumbaugh [Rumbaugh 87], las asociaciones deben considerarse construcciones semánticas de primera clase, con la misma importancia que se da a las clases y generalizaciones, porque mediante las clases y asociaciones conjuntamente se logra abstraer, de modo natural, no sólo la *estructura estática de alto nivel* del sistema, sino también la *estructura global de las interacciones* entre objetos.

Rumbaugh sugirió que también los lenguajes de programación orientados a objetos tenían que evolucionar para implementar mejor las asociaciones. El proyecto original, el *Object-Relation Model* [Rumbaugh 87] derivó en el *Object Modeling Technique* [Rumbaugh 91], que finalmente ha sido una de las principales fuentes conceptuales y notacionales de UML. Este punto de vista tiene fuertes oponentes en investigadores como Brian Henderson-Sellers y otros promotores de la metodología OPEN (*Object-Oriented Process, Environment and Notation*) [Graham 97b], que critican a UML por estar excesivamente basado en el

modelado de datos, con un concepto bidireccional de asociación que viola el principio de encapsulamiento, y así compromete la reutilización [Henderson-Sellers 99a].

En orientación a objetos, el comportamiento de un sistema se define en términos de interacciones entre objetos, es decir, *intercambios de mensajes*. "Enviar un mensaje" habitualmente resulta en la invocación de una operación en el receptor (aunque los mensajes también tienen otros efectos, como crear y destruir objetos, etc.). Las asociaciones son necesarias para la comunicación, ya que *los mensajes son enviados a través de las asociaciones;* sin asociaciones, los objetos quedarían aislados, incapaces de interactuar.

Así pues, tenemos aquí entrelazadas las dos concepciones de la asociación. En la *concepción estática*, la asociación se usa para representar una estructura de datos; en la *concepción dinámica*, la asociación se usa como infraestructura de comunicación entre objetos. Podemos decir que, si en ambas concepciones un objeto (instancia de una clase) representa una "cosa" del mundo real, en la concepción estática un enlace (instancia de una asociación) representa un "hecho" del mundo, mientras que en la concepción dinámica un enlace representa más bien una "posibilidad de comunicación" entre los objetos enlazados. Ambas concepciones están entrelazadas: la asociación expresa una parte del estado del sistema (*aspecto estático*), y en cuanto que especifica el estado, expresa también la capacidad del sistema de comportarse de una determinada manera (*aspecto dinámico*). Como pretendía Rumbaugh, mediante las clases y asociaciones conjuntamente se logra abstraer, de modo natural, no sólo la estructura estática de alto nivel del sistema, sino también la estructura global de las interacciones entre objetos. No obstante, teniendo en cuenta la preeminencia de la concepción estática sobre la concepción dinámica, debe observarse que los nombres de asociaciones no deberían ser nombres de (inter-)acciones sino nombres de predicados (hechos), reservando los nombres de acciones para las operaciones que son invocadas en los objetos a través de las asociaciones. En el ejemplo ya citado de la asociación entre las clases `Empresa` y `Persona` (ver **Figura 1.1**), si esta última clase tiene una operación `contratar`, que expresa una acción, la asociación debería llamarse `está-contratado-por` (o, en sentido inverso, `ha-contratado`), para expresar adecuadamente el hecho mediante un predicado que involucre a ambas clases.

Desafortunadamente, en opinión de algunos autores [Simons 99], el intento de UML de abstraer con la misma construcción semántica (la asociación) tanto la estructura estática del sistema como la estructura de las interacciones entre objetos no está libre de problemas. Se trata de dos nociones de asociación contradictorias que mezclan relaciones entre estructuras de datos con relaciones cliente-servidor, confundiendo así la perspectiva de modelado de datos con la perspectiva del modelado de

servicios. La perspectiva de modelado de datos, heredada del modelado Entidad/Relación (*Entity Relationship Modeling* – ERM), es adecuada para minimizar el acoplamiento entre datos inter-referenciados, y prefiere asociaciones bidireccionales, mientras que la perspectiva del modelado de servicios, heredada del diseño dirigido por responsabilidades (*Responsibility Driven Design* – RDD) es adecuada para minimizar el acoplamiento entre subsistemas que invocan servicios unos de otros, y prefiere asociaciones unidireccionales.

1.4.3. Instanciación de asociaciones

Las instancias de las asociaciones son los enlaces [UML, p. 2-20], como las instancias de las clases son los objetos. El doble aspecto estático y dinámico de las asociaciones debe manifestarse también en los enlaces:

- El aspecto estático aparece en los *diagramas de objetos*, que representan una situación concreta de una parte del sistema en un momento dado, como si fuera una fotografía instantánea. Un diagrama de objetos se usa para expresar el estado del sistema: los objetos representan "cosas", mientras que los enlaces representan "hechos".
- El aspecto dinámico de los enlaces aparece en los *diagramas de colaboración*, que representan una interacción entre objetos que forman parte del sistema, es decir, un intercambio de mensajes. Esta interacción tiene lugar en un contexto, que son los objetos y enlaces que intervienen en ella: los objetos, en cuanto origen, destino o contenido de los mensajes; los enlaces, en cuanto infraestructura de comunicación por la que viajan los mensajes. Un diagrama de colaboración es similar a un diagrama de objetos al que se añadieran los mensajes intercambiados en la interacción.

El Estándar dice que un diagrama de objetos es una instancia de un diagrama de clases [UML, p. 3-35]. En realidad esta expresión es poco afortunada: habría que decir más bien que un diagrama de objetos *se conforma* a un diagrama de clases, en el sentido de que los objetos y enlaces que aparecen son instancias de las correspondientes clases y asociaciones[1]. El "hecho" expresado por el enlace es el que viene significado por el nombre de la correspondiente asociación (en tanto que predicado o "relación conceptual"), particularizada para los objetos en cuestión. En la parte superior de la **Figura 1.2** podemos observar la asociación `ha-reservado` entre las clases `Cliente` y `Mesa` (modelada como clase-asociación para poder registrar la fecha de la reserva), mientras que en la parte inferior el enlace entre los objetos `Juan` y `Mesa7` significa el predicado "Juan ha reservado la Mesa 7 el día 13 de julio".

[1] Y ni siquiera esto es suficientemente preciso, ya que en un diagrama de objetos pueden aparecer instancias de clases y asociaciones especificadas en distintos diagramas de clases, con tal de que pertenezcan todas al mismo modelo.

Figura 1.2. Diagrama de clases de una clase-asociación y diagrama de objetos correspondiente

En cuanto al aspecto estático de los enlaces, el Estándar también dice que todo enlace es una instancia de una asociación [UML, pp. 2-102 y 2-113]; que un enlace es una *tupla de objetos* [UML, p. 2-20] y recíprocamente una asociación es un conjunto de tuplas que relacionan instancias de los clasificadores asociados [UML, p. 2-19]; y que no puede haber tuplas repetidas en el conjunto [UML, pp. 2-19 y 2-110]. Esto nos plantea dos tipos de problemas. En primer lugar, si el enlace es meramente una tupla, un par ordenado de referencias, o si es algo más, si tiene alguna entidad en sí mismo. Es decir, podríamos distinguir entre *identidad* y *valor* del enlace, del mismo modo que ya lo hacemos para los objetos (dos objetos pueden ser distintos aunque tengan los mismos valores en sus atributos): el valor del enlace serían las referencias a los objetos enlazados. En segundo lugar, la restricción de que no pueda haber tuplas repetidas dificulta la representación de algunas situaciones. En la **Figura 1.2**, la asociación `ha-reservado` *no* puede tener dos instancias distintas correspondientes a dos días distintos, el 13 y el 21 de julio, por ejemplo, ya que la identidad del enlace no viene determinada por sus atributos, si los tiene, sino por las identidades de los objetos que enlaza. En la imposición de la restricción de unicidad de las tuplas puede observarse una fuerte influencia del concepto de "relación" (*relation*) tal como es entendido en las metodologías de diseño de bases de datos, en las que es vital evitar la repetición de información redundante. No obstante, son varios los autores que opinan que ésta es una restricción excesiva en UML, cuyo alcance debe ser más amplio que el modelado de datos, y seguramente sería mejor dejar libertad al modelador para imponerla o no según los casos [Genilloud 99, Stevens 02].

En cuanto al aspecto dinámico de los enlaces, según el Estándar los enlaces son usados como *ruta de comunicación*: un objeto puede comunicarse con las instancias enlazadas a él (puede enviarles mensajes), y también puede usar estas instancias como argumentos o valores de respuesta en sus comunicaciones [UML, p. 2-114]. De esta forma, el enlace `Juan-Mesa7` permite que el objeto `Juan` envíe mensajes al objeto `Mesa7`, pero no a las demás instancias de la clase `Mesa`. El problema que se plantea aquí es que a menudo no parece conveniente que toda ruta de comunicación que aparece en un diagrama de colaboración sea una instancia de una asociación estructural que aparezca en un diagrama de clases, ya que, entre otras cosas, los diagramas de clases se recargarían excesivamente con asociaciones para posibilitar todas las comunicaciones necesarias. De hecho, las herramientas CASE tampoco suelen imponer esta restricción. Por otra parte, la asociación no sólo define un conjunto de enlaces, sino que también especifica sus propiedades, de tal forma que un enlace que no fuera instancia de una asociación no tendría propiedades definidas (por ejemplo, la navegabilidad), y por tanto no se podría especificar su comportamiento. Por lo tanto, hay razones a favor y en contra de considerar que todo enlace es instancia de una asociación. Es una cuestión en la que se debe profundizar más.

Todos estos problemas semánticos se agravan cuando consideramos los tipos más complejos de asociaciones: asociación cualificada, clase-asociación, y asociación n-aria. En estas asociaciones es más fácil ver el aspecto estructural que el aspecto dinámico, donde encontramos sutiles paradojas. Por ejemplo, ¿en qué sentido puede usarse un enlace n-ario como ruta de comunicación? ¿Puede un objeto-enlace (instancia de clase-asociación) recibir mensajes de los objetos que enlaza? En estos tipos de asociaciones la definición de UML solamente abarca los aspectos estáticos, dejando un gran vacío semántico para los aspectos dinámicos; en esto se nota también la gran influencia que las técnicas de modelado de datos han tenido en OMT y posteriormente en UML.

1.4.4. La multiplicidad

La aparente sencillez de la definición de multiplicidad esconde sutiles paradojas que no son fáciles de resolver, especialmente si se trata de asociaciones n-arias, para las cuales el Estándar de UML reconoce que la definición de multiplicidad no es obvia [UML, p. 3-79]. En concreto, en la multiplicidad de asociaciones n-arias encontramos dos problemas distintos: una definición ambigua, y una notación insuficiente que no permite expresar todos los valores de multiplicidad que son relevantes para comprender bien una asociación [Génova 02b].

La multiplicidad de las asociaciones binarias tampoco está exenta de dificultades, debido fundamentalmente a la restricción, mencionada en el apartado anterior, de que en una asociación no puede haber tuplas repetidas, lo que dificulta el modelado de muchos problemas comunes.

1.4.5. Direccionalidad y navegabilidad

El concepto de UML que más directamente expresa la direccionalidad de una asociación es la "navegabilidad" (*navigability*), que se expresa gráficamente mediante una flecha situada en el extremo de una asociación entre dos clases, apuntando en la dirección de recorrido (*traversal*). La navegabilidad está estrechamente relacionada con la capacidad de enviar mensajes, hasta tal punto que es muy frecuente identificar estos dos conceptos. Sin embargo, la navegabilidad no es de modo inmediato la dirección en la que pueden "viajar" los mensajes, sino más bien la dirección en la que el objeto emisor puede "mirar" o conocer a otros objetos. En todo caso, la navegabilidad tiene un impacto directo en la comunicación: un objeto puede comunicarse sólo con otros objetos de los que tiene conocimiento, es decir, objetos con los que está conectado a través de rutas navegables que se derivan de asociaciones navegables entre las correspondientes clases [Génova 03a]. Por tanto, la navegabilidad tiene que ver principalmente con el aspecto dinámico de las asociaciones.

Así pues, algunas de las cuestiones que trataremos en relación con el concepto de navegabilidad son: distinguir claramente "dirección del nombre de asociación" de "navegación de la asociación", y averiguar si hay alguna relación entre ellas; distinguir "navegabilidad" de la "capacidad de enviar mensajes", y ver en qué medida una es fundamento de la otra; intentar aplicar de modo coherente el concepto de navegabilidad a las asociaciones n-arias (el Manual de Referencia de UML limita la especificación de navegabilidad a las asociaciones binarias [RM, p. 354], pero el Estándar no hace mención de esta restricción [UML, p. 3-79]); estudiar si el concepto de navegabilidad es propio sólo de la fase de diseño dentro del proceso de desarrollo de software, o si por el contrario también tiene utilidad en la fase de análisis.

1.4.6. Dependencias y asociaciones

Si hay alguna definición vaga e imprecisa en UML, ésta es la definición de "dependencia", usada para representar cualquier cosa que no sea ni generalización ni asociación: una dependencia entre dos elementos de un modelo declara que la implementación o funcionamiento del elemento dependiente requiere la presencia del elemento independiente [UML, p. 2-33]. Ahora bien, éste es precisamente el caso de la generalización, ya que la subclase requiere la presencia de la superclase, por lo tanto es dependiente de ella; y es también el caso de la asociación navegable, ya que la clase origen requiere la presencia de la clase destino. Así pues, habría que concluir que tanto generalización como asociación son subtipos de dependencia, o al menos que *inducen* una dependencia; no obstante, en UML esto no es así [UML, pp. 2-14 y 2-15], sino que generalización, asociación y dependencia son relaciones "hermanas".

Para complicar más aún el panorama, Rumbaugh llama "asociaciones contextuales" a determinado tipo de dependencias: las asociaciones propiamente dichas serían las que expresan la estructura estática del sistema, mientras que las asociaciones dinámicas, que expresan la estructura dinámica del sistema, se representarían como "dependencias de uso" (*usage dependencies*) [Rumbaugh 98].

Por último, conviene señalar que si toda asociación navegable induce una dependencia de la clase origen hacia la clase destino, y que las dependencias dificultan la reutilización de las clases, entonces conviene evitar el uso de asociaciones bidireccionales en favor de asociaciones unidireccionales, que inducen dependencias más débiles [Génova 01].

1.4.7. Visibilidad e interfaces

En UML es posible definir cuatro "niveles" de visibilidad (`private`, `protected`, `package`, y `public`) para las propiedades estructurales y dinámicas (*structural features*, *behavioral features*) de una clase, es decir, sus atributos y operaciones [UML, p. 2-37]. Suele decirse que todos los atributos deberían ser definidos, obligatoriamente o al menos por defecto, con visibilidad `private` (o `protected`), de modo que quedasen protegidos de un acceso indebido que no respetase los invariantes[2] de la clase en cuestión. La manipulación correcta de los atributos quedaría confiada a las operaciones, que se definirían por defecto con visibilidad `public` (o `package`), pudiendo tener las otras visibilidades si es necesario.

Las operaciones públicas de una clase están a disposición de las demás clases asociadas, pero esto nos deja con el problema de que no podemos distinguir fácilmente entre distintos grupos de operaciones que estén disponibles para distintas clases asociadas. Por ejemplo, supongamos que la clase `A` tiene las operaciones `opX`, `opY`, `opZ`, y deseamos que las operaciones `opX`, `opY` sean accesibles para la clase `B`, y las operaciones `opY`, `opZ` para la clase `C`; en este caso los niveles de visibilidad nos sirven de poco, y nos vemos obligados a usar *interfaces*. Una interfaz, en UML, es exactamente esto: un grupo de operaciones con nombre colectivo [UML, p. B-11]. Mediante el empleo de interfaces es posible obtener una "visibilidad" diferente a través de cada una de las asociaciones a las que está conectada una clase: en el ejemplo precedente, la operación `opX` es accesible para la clase asociada `B`, pero no lo es para `C`.

Generalmente se reconoce que la idea de interfaz ha sido una de las contribuciones clave de Java como lenguaje de programación de propósito general, y UML toma este concepto básicamente de Java. El concepto de

---

[2] Un "invariante" es la declaración de una propiedad que debe cumplirse en el "estado estable" de un sistema, es decir, fuera del ámbito de las operaciones atómicas [Kilov 94].

interfaz está muy relacionado con el concepto de rol, hasta tal punto que se ha propuesto la fusión de ambos en uno solo [Steimann 01]. *El uso de interfaces debiera ser la forma estándar de comunicarse con una clase*, en lugar de declarar públicas todas las operaciones que otras clases requieren, ya que lo habitual no es el acceso a toda la funcionalidad que proporciona una clase, sino más bien a un subconjunto de dicha funcionalidad. En UML se pueden definir las interfaces de dos formas distintas: mediante la declaración de una interfaz como tal [UML, p. 2-41 y 3-50], que sería "realizada" por la clase en cuestión [UML, p. 2-71], o mediante un especificador de interfaz asignado al extremo de asociación [UML, pp. 2-24 y 3-72]. Este último es un concepto poco conocido, e infrautilizado por los modeladores, probablemente porque la notación misma es poco expresiva (el nombre de rol seguido por ':' y el nombre del especificador de interfaz). Tampoco está clara en UML la relación, si es que existe, entre interfaces y especificadores de interfaz.

1.4.8. Implementación de asociaciones

Como ya denunció James Rumbaugh hace tiempo [Rumbaugh 87], los lenguajes de programación orientados a objetos expresan bien la clasificación de objetos en clases y la generalización entre clases, pero *no contienen sintaxis ni semántica para expresar directamente las asociaciones entre clases*. Así pues, las asociaciones tienen que ser implementadas mediante una adecuada combinación de clases, atributos y métodos [Rumbaugh 96a, Noble 96, Noble 97, Ambler 01]. La idea más simple consiste en definir un atributo que almacene los enlaces de la asociación, así como métodos de acceso y actualización del atributo (*accessor and mutator methods*) para manipular los enlaces. Esta idea, sin embargo, proporciona una implementación muy incompleta, ya que no garantiza automáticamente el cumplimiento de las propiedades de la asociación. Otras técnicas hacen hincapié en el uso de interfaces de Java para implementar las asociaciones, lo que presenta algunas ventajas [Harrison 00].

Las herramientas CASE a menudo proporcionan algún tipo de generación de código a partir de los modelos de diseño[3], pero limitada a un *esqueleto de código* que incluye sólo generalizaciones y clases, con las signaturas de los atributos y los métodos, pero sin asociaciones[4]. El programador tiene que escribir manualmente el código para gestionar las asociaciones de modo controlado, de manera que se satisfagan todas las

---

[3] Recordemos la distinción entre modelos de análisis y modelos de diseño: un modelo de análisis es una *abstracción del problema* (el mundo real tal como es antes de que se construya el sistema propuesto), mientras que un modelo de diseño es una *abstracción de la solución* (la construcción interna del sistema propuesto) [Kaindl 99]. Por tanto la generación de código sólo tiene sentido para los modelos de diseño.

[4] Algunas herramientas son una excepción a esta regla [Fujaba, Rhapsody].

restricciones e invariantes que debe cumplir una implementación correcta. Habitualmente esto es una tarea repetitiva que podría automatizarse hasta cierto punto.

Además, el número de aspectos que el programador debe tener en cuenta al escribir el código de las asociaciones es tan grande que continuamente corre el riesgo de olvidar algún detalle esencial, especialmente cuando se trata de asociaciones múltiples (con multiplicidad mayor que 1) o bidireccionales (navegables en ambas direcciones). Es más, el código escrito final a menudo queda disperso entre el código de las clases que participan en la asociación, haciéndolo más difícil de mantener.

Uno de los objetivos de esta Tesis Doctoral es establecer una serie de principios de diseño y plantillas de código que guíen a los desarrolladores en la implementación de las asociaciones en un lenguaje de programación orientado a objetos. Estos principios y plantillas pueden emplearse para construir una herramienta que genere código automáticamente para las asociaciones a partir del modelo de diseño [Ruiz 02].

1.5. Objetivos y método

El objetivo de esta Tesis Doctoral es *clarificar y delimitar semánticamente los conceptos definidos en UML* en torno a los distintos elementos que configuran los aspectos estático y dinámico de las asociaciones: instanciación, multiplicidad, direccionalidad y navegabilidad, dependencias, visibilidad e interfaces, e implementación. El estudio de las asociaciones ocupará el núcleo de la Tesis, y los demás tipos de relaciones (generalización y dependencia) se estudiarán en la medida en que afectan a la semántica de las asociaciones. Lo que se busca es la claridad conceptual y la coherencia interna entre las distintas nociones estudiadas, que permitan la aplicación más fiable del lenguaje en la ingeniería software.

Las definiciones oficiales del UML contenidas en el Estándar y en el Manual de Referencia se caracterizan por su vaguedad, o por usar fórmulas abstractas difíciles de comprender suficientemente bien como para aplicarlas con seguridad. Por otra parte, hay una notable escasez de ejemplos, especialmente de casos conflictivos que precisamente servirían para aclarar los aspectos más sutiles del lenguaje. No se trata simplemente de buscar contradicciones internas. Se trata sobre todo de entender UML sobre el trasfondo general del paradigma de la orientación a objetos. Por tanto, la comparación con otros métodos y lenguajes será de gran utilidad.

Como ésta es una *investigación conceptual* acerca del concepto de "asociación" en UML, la principal fuente de estudio será el Estándar de UML [UML] promulgado por el *Object Management Group*, teniendo en cuenta las versiones anteriores cuando sea necesario señalar la evolución de una idea o definición [UML11, UML13]. Este documento es propiamente el único que se puede llamar "definición oficial"; no obstante, como hay

muchas cuestiones semánticas que están pobremente explicadas en el Estándar, acudir al Manual de Referencia [RM] parece la opción más obvia si se desea encontrar alguna clarificación; aunque el Manual de Referencia no haya sido adoptado por el OMG, la opinión de los autores originales de UML (Booch, Rumbaugh y Jacobson) debe considerarse particularmente autorizada. También examinamos el contenido de la Guía del Usuario [UG], de los mismos autores, no porque la consideremos una fuente particularmente fiable, sino porque posiblemente se trata de la fuente principal para muchos modeladores, de modo que es importante mostrar sus virtudes y deficiencias.

1.6. Resultados y aplicaciones previstos

De este estudio se pretenden obtener los siguientes resultados:

- Relación de ambigüedades y errores detectados en el Estándar, así como soluciones propuestas.
- Relación de reglas de coherencia entre diagramas o, equivalentemente, reglas a las que debe ajustarse un modelo bien formado.
- Relación de reglas de estilo que debe seguir un buen modelador, sin ser obligatorias para que el modelo esté bien formado, pero que sí lo harán más legible.

Así mismo, el estudio desarrollado en esta Tesis Doctoral tendrá las siguientes aplicaciones:

- Enseñanza: la aplicación más inmediata de la clarificación conceptual es una mejor comprensión del lenguaje, que capacita al docente para enseñarlo con mayor claridad y seguridad, sabiendo hacer hincapié en los puntos verdaderamente importantes para que los alumnos no se queden en un conocimiento vago, sino directamente aplicable a casos reales. Se pretende también encontrar un abundante número de ejemplos que ayudarán a explicar mejor el uso del lenguaje.
- Trazabilidad del software (análisis, diseño, implementación): una segunda aplicación es la capacidad de distinguir mejor cómo se usan las asociaciones en el análisis y en el diseño, que son respectivamente una abstracción del problema en el mundo real, y una abstracción de la solución en el mundo informático. Entendiendo mejor cómo se implementan las asociaciones, se aclara la relación que hay entre el análisis y el diseño, y cómo se pasa de uno a otro, gracias también al uso de patrones de diseño.
- Construcción y evaluación de herramientas de diseño asistido por ordenador: el conjunto de reglas de coherencia obtenidas, junto con las reglas de estilo, servirán como pautas que debe seguir una buena herramienta CASE. Estas pautas, por tanto, pueden usarse

tanto para evaluar herramientas experimentales o comerciales, como para orientar su construcción o mejora.

1.7. Experimentación y validación

El desarrollo de la experimentación será el objeto de la Tercera Parte de la Tesis. UML es un lenguaje de análisis y diseño, que no es directamente ejecutable, de modo que la validación experimental de las propuestas realizadas vendrá dada por la aplicación de los principios de modelado expuestos en esta investigación a la generación de código para asociaciones, mediante un método que generará, de modo totalmente automático, un código de calidad que implemente de modo fiable y riguroso los siguientes aspectos de las asociaciones UML:

- Multiplicidad.
- Navegabilidad.
- Visibilidad.

La clarificación de los problemas metodológicos existentes en las asociaciones UML deberá producir un beneficio directo en la perfecta definición de la semántica de un modelo basado en asociaciones. Por este motivo, la generación de un código fuente con un contenido semántico superior al código generado por las herramientas CASE actuales será la base de la experimentación de esta Tesis Doctoral.

Para el diseño de la experimentación se construirá una herramienta de generación de código a partir de modelos UML que contengan asociaciones entre clases. Esta herramienta leerá un modelo UML, almacenado en formato XMI, y generará código Java para gestionar las asociaciones UML contenidas en el mismo, de acuerdo con los principios estudiados en esta Tesis Doctoral, como se describirá en el Capítulo 6.

Además, esta herramienta formará parte de otra herramienta de mayor alcance, *UML-CAKE (Computer Aided Knowledge Engineering)*, que está destinada a facilitar la ingeniería software con modelos de análisis y diseño descritos en UML. La herramienta *CAKE* se encuentra en avanzado proceso de desarrollo en el seno del *Information Engineering Group* (Departamento de Informática, Universidad Carlos III de Madrid), del cual es miembro el autor de esta Tesis Doctoral, en colaboración con la empresa dTinf [dTinf 02].

La experimentación con estas dos herramientas pretende probar que la clarificación de los principios teóricos llevada a cabo en esta Tesis Doctoral contribuye a mejorar la automatización de los procesos de la Ingeniería del Software mediante la generación de un código más preciso, robusto y semánticamente correcto.

1.8. Estructura de esta Tesis Doctoral

Esta Tesis Doctoral está dividida en cuatro Partes, que a su vez se dividen en Capítulos, Secciones y Apartados. La Primera Parte (*Planteamiento*) consta de dos Capítulos de carácter introductorio:

1. Introducción
2. Estado de la cuestión

La Segunda Parte (*Desarrollo teórico - El concepto de asociación en UML. Aspectos estáticos y dinámicos*) constituye el núcleo de la Tesis Doctoral, dividido en tres Capítulos que se centran en los tres aspectos básicos estudiados de las asociaciones en UML:

3. La multiplicidad de las asociaciones
4. La navegabilidad de las asociaciones
5. La visibilidad de las asociaciones

La Tercera Parte (*Experimentación*) presenta en un Capítulo la aplicación de los principios descubiertos en esta investigación al desarrollo de una herramienta CASE y la consiguiente experimentación:

6. La implementación de las asociaciones

Finalmente, la Cuarta Parte (*Conclusiones y Referencias*) consta de dos Capítulos que cierran la Tesis:

7. Conclusiones
8. Referencias

2. Estado de la cuestión

En este Capítulo, tras una breve introducción histórica, vamos a presentar el estado actual de las asociaciones en la versión 1.4 del estándar del lenguaje UML (la nueva versión 2.0 aún no ha sido publicada en la fecha de redacción final de esta Tesis Doctoral). La exposición separa los aspectos notacionales y semánticos, tal como hace el Estándar, aunque en orden inverso. Consideramos que esta exposición es necesaria para facilitar la tarea al lector menos familiarizado con los detalles del lenguaje.

2.1. Breve historia de UML

UML nació como fruto del esfuerzo conjunto de tres prestigiosos metodólogos: Grady Booch, Ivar Jacobson y James Rumbaugh. Cada uno de ellos había desarrollado su propio método orientado a objetos con amplia difusión en el mundo de la industria, y habían comprendido la necesidad de un método unificado que simplificase las tareas de análisis y diseño de las aplicaciones, y facilitase el intercambio de información entre profesionales de distintos equipos de trabajo.

2.1.1. Precursores de UML

Los lenguajes de modelado orientados a objetos empezaron a surgir desde mediados de los 70 hasta finales de los 80 a medida que diversos metodólogos experimentaban con diferentes enfoques del análisis y diseño orientado a objetos, influenciados por técnicas como el modelado Entidad/Relación [Chen 76], el lenguaje SDL (*Specification & Description Language*, 1976, CCITT), y otras. Posteriormente, el número de lenguajes de modelado creció desde menos de 10 hasta más de 50 en el periodo 1989-1994, pero muchos usuarios de métodos orientados a objetos no encontraban plena satisfacción en ningún lenguaje de modelado concreto, alimentando así la "guerra de los métodos".

Durante la década de los 90 los métodos seguían evolucionando, y los unos empezaron a incorporar las técnicas de los otros. Así emergieron unos pocos métodos que sobresalían claramente entre el resto, particularmente los métodos OOSE, OMT-2 y Booch'93. Cada uno de ellos era un método completo, con reconocidas virtudes. En términos sencillos, OOSE estaba orientado a los casos de uso, proporcionando un excelente apoyo para la ingeniería de negocio y el análisis de requisitos; OMT-2 era especialmente expresivo para el análisis de sistemas de información orientados a los datos;

y Booch'93 era particularmente útil durante las fases de diseño y construcción de proyectos.

2.1.2. Booch, Rumbaugh y Jacobson unen sus fuerzas

El desarrollo de UML comenzó en octubre de 1994, cuando Grady Booch y Jim Rumbaugh, incorporados a la empresa *Rational Software Corporation*, empezaron a trabajar en la unificación de los métodos Booch [Booch 94] y OMT (*Object Modeling Technique*) [Rumbaugh 91]. Dado que estos métodos ya habían comenzado a aproximarse independientemente, y era mundialmente reconocido su liderazgo en el campo de la orientación a objetos, Booch y Rumbaugh unieron sus fuerzas para lograr la completa unificación de su trabajo. Así, la versión 0.8 del Método Unificado (*Unified Method*), como era conocido entonces, fue publicada en octubre de 1995. Poco después Ivar Jacobson y su empresa *Objectory* se unieron a *Rational* y a este esfuerzo de unificación, incorporando el método OOSE (*Object-Oriented Software Engineering*) [Jacobson 92].

Como autores principales de estos tres métodos, Booch, Rumbaugh y Jacobson tenían buenas razones para crear un lenguaje unificado de modelado:

- Primera, sus respectivos métodos ya estaban evolucionando hacia el encuentro de modo independiente. Tenía sentido continuar esta evolución de modo conjunto, en lugar de cada uno por su cuenta, eliminando así la posibilidad de diferencias innecesarias y gratuitas que no harían más que confundir a los usuarios.
- Segunda, unificando la semántica y la notación, podrían traer una cierta estabilidad al mercado de la orientación a objetos, permitiendo el asentamiento de los proyectos sobre un lenguaje de modelado maduro, y que los fabricantes de herramientas se concentrasen en funcionalidades más útiles.
- Tercera, esperaban que su colaboración mejoraría los tres métodos precedentes, aprovechando las lecciones aprendidas y enfrentándose a problemas que ninguno de los tres había resuelto bien, independientemente de las ventajas comerciales que podría reportarles.

Inventar una notación para usar en análisis y diseño orientado a objetos no es muy distinto a inventar un lenguaje de programación. Ante todo, hay que negociar un equilibrio. Primero, es necesario delimitar bien el problema: ¿Debe la notación abarcar la especificación de requisitos? Sí, en parte. ¿Debe la notación extenderse hasta convertirse en un lenguaje de programación visual? No. Segundo, debe encontrase el término medio entre expresividad y simplicidad: una notación demasiado simple limitaría el ámbito de problemas que pueden resolverse; una notación demasiado compleja abrumaría al pobre desarrollador mortal.

En el caso de la unificación de métodos existentes, debe mostrarse además sensibilidad hacia lo que ya está asentado: si se hacen demasiados cambios, se confundirá a los usuarios existentes; si, por el contrario, se impide el avance en la notación, se perderá la oportunidad de conectar con un conjunto mucho más amplio de usuarios. Los precursores de UML –Booch, Rumbaugh y Jacobson– procuraron encontrar el mejor equilibrio en estos aspectos, y sus esfuerzos resultaron en el lanzamiento de las versiones 0.9 y 0.91 de UML en junio y octubre de 1996. Durante este año habían solicitado, recibido e incorporado numerosas sugerencias, pero quedaba claro que aún se requería mayor atención.

2.1.3. Los colaboradores de UML

A lo largo de 1996 diversas organizaciones comenzaron a considerar que UML era una herramienta estratégica para sus negocios. Una Solicitud de Propuestas (*Request for Proposals - RFP*) promulgada por el *Object Management Group* (OMG) proporcionó el necesario catalizador para que estas organizaciones unieran sus fuerzas en la producción de una respuesta conjunta a la RFP.

Rational estableció un consorcio de Colaboradores de UML (*UML Partners*) con distintas organizaciones dispuestas a dedicar recursos a la elaboración de una definición fuerte de UML, la versión 1.0. Entre los que más contribuyeron a la definición de UML están *Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI,* y *Unisys.* En enero de 1997 *IBM & ObjecTime, Platinum Technology, Ptech, Taskon & Reich Technologies,* y *Softeam* enviaron respuestas individuales a la RFP del OMG. Estas empresas se unieron a los Colaboradores de UML para contribuir con sus ideas, y así surgió la versión revisada UML 1.1 [UML11], con el objetivo de mejorar la claridad de la semántica de UML 1.0 e incorporar las contribuciones de los nuevos colaboradores. El resultado de este trabajo en equipo fue que UML se vio beneficiado por las diferentes perspectivas, áreas de interés y experiencias de los distintos colaboradores.

2.1.4. La estandarización de UML

El OMG ha asumido la responsabilidad de los sucesivos desarrollos del estándar de UML, desde que en noviembre de 1997 añadiera la versión 1.1 de UML a su lista de tecnologías adoptadas, de modo que ya no es una iniciativa exclusiva de *Rational*. Posteriormente se encargó a un Equipo de Revisión (*Revision Task Force - RTF*) que aceptase comentarios del público en general, y que revisara el estándar para encontrar errores, inconsistencias, ambigüedades y omisiones menores que pudieran ser resueltas sin un cambio sustancial en el alcance de la especificación original.

Como fruto de este trabajo, se publicó la versión 1.3 en junio de 1999 [UML13]. Contenía un cierto número de cambios al metamodelo, semántica y notación de UML, pero a grandes rasgos esta versión debe ser considerada

como una actualización menor de la especificación original. La versión 1.3 fue divulgada al público en general por los autores principales de UML mediante tres libros: *The Unified Modeling Language User Guide* [UG], *The Unified Modeling Language Reference Manual* [RM], y *The Unified Software Development Process* [UP][5].

El trabajo continuó, con la publicación en septiembre de 2001 de la actual versión 1.4 [UML], que contiene pocos cambios significativos respecto a la versión 1.3, aunque elimina muchos de los defectos menores. Actualmente una nueva versión 2.0 está en pleno proceso de elaboración (se espera su publicación para finales del año 2003 o comienzos del 2004). El alcance de los cambios probablemente será mucho mayor que en las revisiones anteriores.

2.2. El estándar de UML: notación y semántica

La versión actual del estándar de UML es un documento de 566 páginas dividido en distintas secciones y anexos. Las secciones más extensas son, con mucho, la sección dedicada a la semántica y el metamodelo[6] (Sección 2, *UML Semantics*) y la sección dedicada a la notación (Sección 3, *UML Notation Guide*), con 200 y 180 páginas respectivamente. Otras secciones más breves se dedican a presentar brevemente algunos ejemplos de perfiles (Sección 4, *UML Example Profiles*), el intercambio de modelos UML (Sección 5, *UML Model Interchange*) y el lenguaje de restricción de objetos (Sección 6, *Object Constraint Language*). Estas últimas secciones no son ni pretenden ser exhaustivas, de modo que para encontrar un tratamiento completo de estas cuestiones es preciso acudir a otras fuentes [OCL, XMI]. El resto del documento está constituido por los necesarios prefacios, sumarios, glosarios e índices.

La notación empleada por el lenguaje es lo primero que le interesa al modelador que emplea UML como herramienta de trabajo. La notación es algo así como el libro de instrucciones o manual de usuario del lenguaje. Sólo en un segundo momento el modelador llega a interesarse (y no en todos los casos) por la semántica del lenguaje, es decir, el significado completamente preciso de tal o cual símbolo empleado en la notación (este significado preciso es el que se expresa en el Estándar por medio del metamodelo). Por esta razón, el presente trabajo comienza por la exposición de los distintos elementos de la notación de las asociaciones para, posteriormente, exponer su semántica. Normalmente las cuestiones

---

[5] Aunque este último se centra no tanto en la explicación del lenguaje, sino más bien en el proceso de desarrollo de software, empleando UML como lenguaje de análisis y diseño.

[6] El "metamodelo", como veremos posteriormente con más detalle, es una descripción del lenguaje UML mediante el propio lenguaje UML.

semánticas, a veces muy abstractas, se entienden mejor una vez conocidos los problemas de la notación, que son más concretos. El Estándar describe la semántica antes que la notación; este orden es adecuado al carácter formal del documento, pero para una exposición comprensiva nos ha parecido legítimo, y más conveniente, el orden inverso[7].

La Sección 3, *UML Notation Guide*, después de tratar algunas cuestiones generales acerca de la notación, describe uno por uno los elementos gráficos del lenguaje siguiendo el orden de los distintos tipos de diagramas: diagramas de clases y de objetos, diagramas de casos de uso, diagramas de secuencia y de colaboración, diagramas de estados, diagramas de actividad, y finalmente diagramas de componentes y de despliegue. Dentro de cada elemento descrito podemos encontrar uno o varios de los siguientes apartados:

- *Semantics*: breve descripción de la semántica del elemento.
- *Notation*: representación gráfica del elemento.
- *Presentation Options*: diferentes alternativas para representar el elemento.
- *Style Guidelines*: recomendaciones de buen estilo.
- *Examples*: ejemplos de uso del elemento.
- *Mapping*: correspondencia de la notación gráfica con el metamodelo descrito en la Sección 2, *UML Semantics*.

Como puede observarse, los apartados primero y último, *Semantics* y *Mapping*, sirven de nexo entre las Secciones *UML Semantics* y *UML Notation Guide* del Estándar. Especialmente importante es la correspondencia (*mapping*) entre notación y semántica, ya que un elemento gráfico sin un lugar bien definido en el metamodelo carecería de significado preciso.

Pasemos ahora a la Sección 2, *UML Semantics*. Como ya hemos visto, en esta sección se trata de describir de modo preciso el significado de cada uno de los elementos del lenguaje. La técnica empleada es la construcción de un "metamodelo", es decir, un modelo del propio lenguaje o, en otras palabras, la descripción del lenguaje UML mediante los elementos del mismo lenguaje UML, de modo análogo a como podemos encontrar una gramática del castellano escrita en castellano. Cada elemento del lenguaje (clase, asociación, atributo, operación...) está representado en el metamodelo por la correspondiente "metaclase" (`Class`, `Association`, `Attribute`, `Operation`...), y las relaciones entre los distintos elementos del lenguaje se representan mediante "meta-asociaciones" (por ejemplo, todo atributo pertenece a una clase, y esto se representa en el metamodelo mediante una meta-asociación entre las metaclases `Attribute`, y `Class`).

---

[7] Por otra parte, ni la Guía del Usuario [UG] ni el Manual de Referencia [RM] separan las cuestiones notacionales de las semánticas, sino que las estudian conjuntamente.

Como el lenguaje es extenso, el metamodelo es relativamente complejo, de modo que ha sido organizado en distintos paquetes[8] [UML, p. 2-6]. En el nivel más alto (ver **Figura 2.1**) encontramos dos grandes bloques (`Foundation` y `Behavioral Elements`) y un tercer bloque auxiliar (`Model Management`); los bloques, o paquetes, aparecen relacionados mediante flechas discontinuas que expresan las dependencias entre ellos. Los dos grandes bloques mencionados tienen a su vez una organización interna en subpaquetes (ver **Figura 2.2** y **Figura 2.3**).

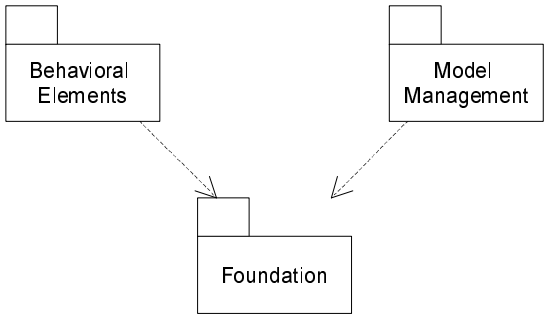

Figura 2.1. Nivel superior de organización del metamodelo

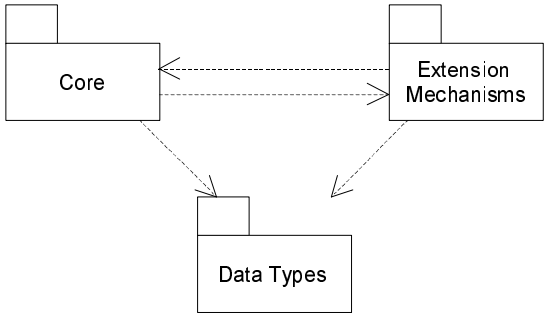

Figura 2.2. Contenido del paquete `Foundation`

[8] Los "paquetes" en UML son unidades de agrupamiento sin otra finalidad que la mejor organización de un modelo y la definición de espacios de nombres (*namespaces*).

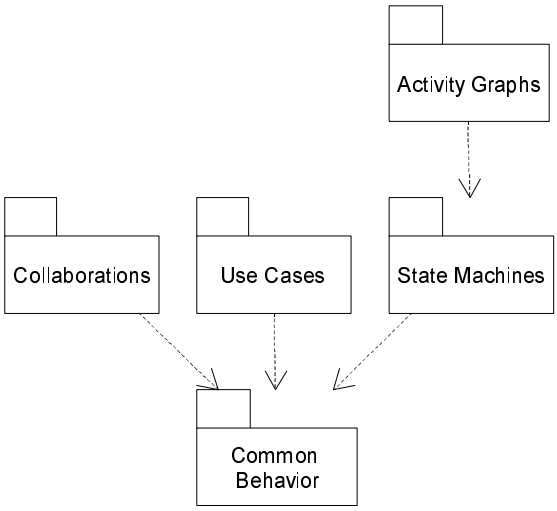

Figura 2.3. Contenido del paquete `Behavioral Elements`

La sección 2 del Estándar, *UML Semantics*, está organizada de acuerdo con la estructura de paquetes y subpaquetes del metamodelo. La descripción de cada paquete o subpaquete consta habitualmente de los siguientes apartados:

- *Abstract Syntax*: contiene uno o varios diagramas de clases descriptivos de la parte del metamodelo que se está tratando; los diagramas van seguidos de una explicación textual de cada una de las metaclases definidas en el paquete, con la explicación de sus atributos, asociaciones, estereotipos predefinidos[9], etc.
- *Well-Formedness Rules*: la notación gráfica de UML resulta insuficiente para definir de modo preciso el metamodelo, de modo que es necesario complementar los diagramas con una lista de reglas para la correcta formación de modelos; estas reglas están descritas en lenguaje natural y en el *Object Constraint Language* [OCL].

[9] Un *estereotipo* es un elemento del lenguaje UML que se utiliza para modificar el significado de otro elemento del lenguaje, aportándole un significado particular. Por ejemplo, un enlace estereotipado es un tipo peculiar de enlace que puede tener propiedades especiales en cuanto a su instanciación, multiplicidad, navegabilidad, visibilidad, etc. Como los elementos del lenguaje se representan mediante metaclases en el metamodelo, la aplicación de un estereotipo a un elemento del lenguaje es equivalente a definir una nueva metaclase en el metamodelo, que será subclase de la metaclase original. En UML hay estereotipos predefinidos, y es posible definir nuevos estereotipos en un modelo particular o en un perfil de usuario. La representación básica de un estereotipo es mediante una palabra clave encerrada entre los signos « » (comillas francesas), situados sobre el elemento modificado, aunque también es posible definir iconos gráficos específicos para un estereotipo determinado [UML, pp. 2-78 y 3-31].

- *Detailed Semantics*: los diagramas y las reglas proporcionan una descripción muy abstracta del metamodelo de UML, referida principalmente a los modelos que se *pueden* construir válidamente con UML. Este último apartado proporciona explicaciones textuales en lenguaje natural, que resultan muy útiles para comprender el sentido de algunos elementos del lenguaje, su relación con otros elementos, y cuál debe ser su *uso correcto* a la hora de utilizarlos para construir los modelos del usuario.

En conclusión, podemos observar que las dos principales secciones del Estándar, *UML Semantics* y *UML Notation Guide*, siguen una metodología bastante distinta, y a menudo no es sencillo encontrar la correspondencia entre una y otra sección, a pesar de las referencias explícitas. En cuanto al tema de las asociaciones, la información esencial la encontramos en la parte de *Notation Guide* dedicada a los diagramas estáticos (de clases y de objetos), y en la parte de *Semantics* dedicada al estudio del paquete `Core` (núcleo). No obstante, para entender bien el concepto de asociación es necesario tener una visión global del Estándar, ya que, lógicamente, hay mucha información que se encuentra diseminada por todo el documento. De modo particular, para entender el papel de las asociaciones en el dinamismo de un sistema será preciso recurrir a la semántica de los elementos dinámicos (el paquete `Behavioral Elements`) y a la notación de prácticamente todos los demás tipos de diagramas.

2.3. La notación de las asociaciones

Antes de describir la notación de las asociaciones es preciso definir la notación de otros elementos aún más básicos del lenguaje, principalmente las clases, ya que una asociación se establece entre clases[10].

2.3.1. Clases y objetos

Un *modelo* es una abstracción o simplificación de un sistema; el modelo está constituido por un conjunto de elementos, que se corresponden con los elementos del sistema real modelado [UML, p. B-12]. Una *vista* es una proyección de un modelo desde una determinada perspectiva, en la que se omiten las entidades que no son relevantes a esta perspectiva [UML, p. B-21]; un modelo puede tener muchas vistas distintas, tantas como necesiten los desarrolladores. Un *diagrama* es una presentación gráfica de una colección de elementos del modelo [UML, p. B-8]; habitualmente un diagrama se usa para representar una vista de un modelo.

[10] En realidad, una asociación se establece entre "clasificadores", un concepto algo más amplio que veremos más adelante. Por simplicidad, por el momento hablaremos de clases en lugar de clasificadores.

Una *clase* es el descriptor de un conjunto de objetos que comparten estructura, comportamiento y relaciones [UML, p. 3-35]. Una clase se dibuja como un rectángulo de trazo continuo con tres compartimentos separados por líneas horizontales. El compartimento superior contiene el nombre de la clase, mientras que los compartimentos medio e inferior, que son opcionales, contienen sendas listas de *atributos* y *operaciones* [UML, p. 3-36]. Un *diagrama de clases* es un grafo de clases conectadas por distintos tipos de relaciones estáticas [UML, p. 3-34]. Entre otras relaciones, las más usadas en un diagrama de clases son la *generalización*, la *dependencia* y la *asociación*, representadas respectivamente mediante una línea continua terminada en un triángulo blanco, una línea discontinua terminada en una punta de flecha, y una línea continua simple (ver **Figura 2.4**).

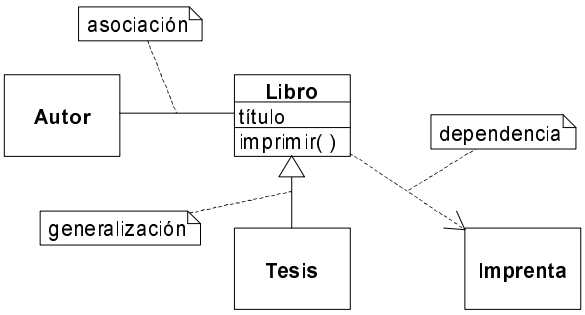

Figura 2.4. Diagrama de clases con las tres relaciones básicas

En el ejemplo de la **Figura 2.4** podemos ver un sencillo diagrama de clases con cuatro clases y tres relaciones. La clase `Libro` contiene el atributo `título` y la operación `imprimir`, y tiene relaciones de asociación, generalización y dependencia con las clases `Autor`, `Tesis` e `Imprenta` respectivamente. La relación de asociación significa que las instancias[11] de `Autor` y de `Libro` están relacionadas; la relación de generalización (o especialización, si se lee en sentido inverso) significa que el conjunto de instancias de `Tesis` es un subconjunto del conjunto de instancias de `Libro`; la relación de dependencia significa que la clase `Libro` depende de alguna manera de la clase `Imprenta`, por ejemplo, porque la operación `imprimir` requiere la especificación de una instancia de `Imprenta` como parámetro.

Un *objeto* representa una instancia particular de una clase, y tiene identidad y valores concretos para los atributos de su clase [UML, p. 3-64]. La notación de los objetos se deriva de la notación de las clases, mediante el empleo del subrayado. Un objeto se dibuja como un rectángulo con dos compartimentos. El primer compartimento contiene el nombre del objeto y

[11] El término "instancia" se toma habitualmente como sinónimo de "objeto".

de su clase, separados por el símbolo ':' y subrayados ambos, mientras que el segundo compartimento, opcional, contiene la lista de atributos de la clase con los valores concretos que tienen en ese objeto [UML, p. 3-65]. Un *enlace* es una instancia de una asociación [UML, p. 3-84], del mismo modo que un objeto es una instancia de una clase[12]. Un enlace se representa mediante una línea continua que une los objetos enlazados [UML, p. 3-84]. Un *diagrama de objetos* es un grafo de instancias, es decir, objetos, valores de datos y enlaces. Un diagrama de objetos es una instancia de un diagrama de clases; muestra una especie de fotografía del estado del sistema en un cierto instante de tiempo [UML, p. 3-35]. En la **Figura 2.5** podemos ver un sencillo ejemplo de diagrama de objetos correspondiente al diagrama de clases de la **Figura 2.4**.

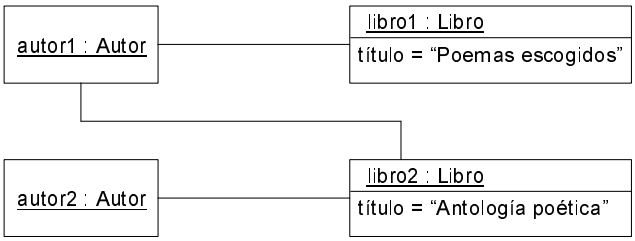

Figura 2.5. Diagrama de objetos que muestra objetos, valores y enlaces

En este diagrama podemos ver dos instancias de la clase `Autor`, dos instancias de la clase `Libro`, y tres instancias (es decir, enlaces) de la asociación `Autor-Libro`; estos enlaces significan que el primer autor es autor de dos libros, mientras que el segundo autor sólo es autor de un libro; a su vez, el primer libro tiene un autor y el segundo libro tiene dos autores.

Una vez vistos estos preliminares indispensables, podemos pasar a ver con más detalle la notación de las asociaciones. Comenzaremos por el caso más sencillo, la asociación binaria, seguida de otros casos especiales: asociación reflexiva, agregación y composición, asociación cualificada, clase-asociación, y asociación n-aria.

2.3.2. Asociación binaria

Una asociación binaria es una asociación entre dos clases [UML, p. 3-68]. Se representa mediante una línea continua que conecta las dos clases asociadas; la línea puede ser horizontal, vertical, oblicua, curva, o estar

[12] Algunas propuestas de mejora de UML de cara a la versión 2.0 proponen un cambio de terminología: "asociación" en lugar de "enlace", "tipo de asociación" en lugar de "asociación", y "tipo de objeto" en lugar de "clase" [cUML]. Con esta acertada propuesta se evitaría el uso de dos términos distintos para denominar lo que es esencialmente el mismo concepto, ya sea en el nivel de las instancias, ya sea en el nivel de los tipos. No obstante, en esta Tesis Doctoral vamos a seguir la actual terminología oficial de UML.

formada por dos o más trazos unidos. El cruce de dos asociaciones que no están conectadas se puede mostrar con un pequeño semicírculo (como en los diagramas de circuitos eléctricos) [UML, pp. 3-7 y 3-69]. La asociación en sí misma puede distinguirse de sus extremos (*association ends*), mediante los cuales se conecta a las dos clases asociadas. Las propiedades relevantes de una asociación pueden pertenecer a la asociación como tal, o a uno de sus extremos. En cada caso estas propiedades se representan mediante adornos gráficos (*graphical adornments*) situados en el centro de la asociación [UML, p. 3-68] o en el extremo correspondiente [UML, p. 3-71] (de tal forma que al mover o modificar la forma de una asociación en una herramienta CASE, el adorno debe desplazarse solidariamente con la asociación). Los adornos que podemos encontrar en una asociación binaria son:

2.3.2.1. Nombre de la asociación (*association name*)

Es opcional, y se representa mediante una cadena de caracteres situada junto al centro de la asociación, suficientemente separada de los extremos como para no ser confundida con un nombre de rol [UML, p. 3-69].

El nombre de la asociación puede contener un pequeño triángulo negro, denominado "dirección del nombre" (*name direction*), que apunta en la dirección en la que se debe leer la asociación (ver **Figura 2.6**).

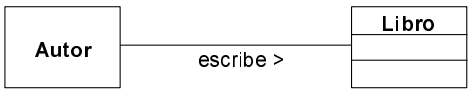

Figura 2.6. Ejemplo de asociación con nombre y dirección de nombre

2.3.2.2. Nombre de rol (*rolename*)

Se representa mediante una cadena de caracteres situada junto a un extremo de la asociación (ver **Figura 2.7**). Es opcional, pero si se especifica en el modelo ya no puede ser omitido en las vistas. Indica el rol que juega en la asociación la clase unida a ese extremo [UML, p. 3-72].

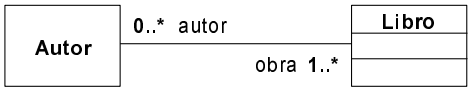

Figura 2.7. Ejemplo de asociación con nombres de rol y multiplicidades

Tanto el nombre de rol como la multiplicidad deben situarse cerca del extremo de la asociación, para que no se confundan con otra asociación distinta. Se pueden poner en cualquier lado de la línea; es tentador especificar que siempre se sitúen en el mismo lado (a mano derecha o a mano izquierda, según se mira desde la clase hacia la asociación), pero en

un diagrama repleto de símbolos debe prevalecer la claridad. El nombre de rol y la multiplicidad pueden situarse en lados contrarios del mismo extremo de asociación, o juntos en el mismo lado.

2.3.2.3. Multiplicidad (*multiplicity*)

La especificación de multiplicidad en una asociación es opcional; además, se puede omitir en una vista, aunque se haya especificado en el modelo [UML, p. 3-71]. La multiplicidad, en general, especifica el rango de cardinalidades admisibles en un conjunto; por ejemplo, roles en asociaciones, partes en compuestos, repeticiones en secuencias de operaciones, etc. [UML, p. 3-75].

Una especificación de multiplicidad es un subconjunto del conjunto de enteros no negativos, expresada mediante una secuencia de intervalos enteros separados por comas, donde cada intervalo representa un rango potencialmente infinito de enteros, con el formato:

`límite-inferior..límite-superior`,

donde `límite-inferior` y `límite-superior` son valores enteros literales que especifican un intervalo cerrado de enteros. Además, el asterisco '*' se puede usar como límite superior para denotar un valor ilimitado ('muchos').

Si se especifica un único valor, entonces el rango contiene sólo ese valor. Si la multiplicidad consiste en un asterisco solo, entonces denota el rango completo de los enteros no negativos, es decir, equivale a 0..* (cero o más). La multiplicidad 0..0 no tiene sentido, ya que indicaría que no puede haber ninguna instancia.

Ejemplos (ver **Figura 2.7**):

- 0..1
- 1
- 0..*
- *
- 1..*
- 1..6
- 1..3, 7..10, 15, 19..*

2.3.2.4. Ordenación (*ordering*)

Si la multiplicidad es mayor que uno, entonces el conjunto de elementos relacionados puede estar ordenado o no ordenado (que es la opción por defecto si no se indica nada). La ordenación se especifica mediante la propiedad `{ordered}` en el extremo correspondiente [UML, p. 3-71] (ver **Figura 2.8**).

La declaración de que un conjunto es ordenado no especifica cómo se establece o mantiene la ordenación, y en todo caso no se permiten elementos duplicados. La regla de ordenación puede estar basada o no en los valores de los elementos ordenados, y puede especificarse mediante una restricción adicional. Las operaciones que insertan nuevos elementos son responsables

de especificar su posición, ya sea implícitamente (por ejemplo, al final) o explícitamente. La estructura de datos y el algoritmo empleados para la ordenación son detalles de implementación y decisiones de diseño que no añaden nueva semántica a la asociación.

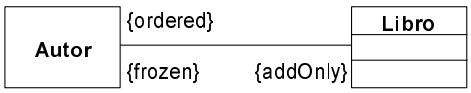

Figura 2.8. Ejemplo de asociación con especificación de ordenación y modificabilidad

2.3.2.5. Modificabilidad (*changeability*)

Si los enlaces son modificables en un extremo (pueden ser añadidos, borrados o cambiados), entonces no se necesita ninguna indicación especial [UML, p. 3-73]. Por el contrario, la propiedad `{frozen}` indica que no se pueden añadir, borrar ni cambiar enlaces de un objeto una vez que ha sido creado e inicializado. La propiedad `{addOnly}` indica que se pueden añadir enlaces, pero no se pueden borrar ni modificar los existentes (ver **Figura 2.8**).

2.3.2.6. Navegabilidad (*navigability*)

Una punta de flecha en el extremo de una asociación indica que es posible la navegación hacia la clase unida a dicho extremo (ver **Figura 2.9**). Se pueden poner flechas en uno, dos o ninguno de los extremos. Para ser totalmente explícito, se puede mostrar la flecha en todos los lugares donde la navegación es posible; en la práctica, no obstante, a menudo conviene suprimir algunas de las flechas y mostrar sólo las situaciones excepcionales. Si no se muestra la flecha en un extremo, no se puede inferir que la asociación no sea navegable en esa dirección; se trata simplemente de información omitida [UML, p. 3-72].

Conviene adoptar un estilo uniforme en la presentación de las flechas de navegabilidad en los diagramas; el estilo adoptado seguramente variará a lo largo del tiempo en función del tipo de diagrama que se trate. Algunos estilos posibles son [UML, p. 3-73]:

- *Primer estilo*: mostrar todas las flechas. La ausencia de una flecha indica que la navegación no es posible.
- *Segundo estilo*: suprimir todas las flechas. No se puede inferir nada sobre la navegabilidad de las asociaciones, de igual modo que en otras situaciones hay determinada información que se suprime en una vista por conveniencia, no porque la información no exista.
- *Tercer estilo*: suprimir las flechas con navegabilidad en ambas direcciones, mostrar las flechas sólo para asociaciones con navegabilidad en una dirección. En este caso, no se puede distinguir la asociación navegable en las dos direcciones de la

asociación que no es navegable en ninguna dirección, aunque este último caso es ciertamente muy raro en la práctica.

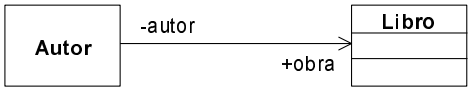

Figura 2.9. Ejemplo de asociación con especificación de navegabilidad y visibilidad

2.3.2.7. Visibilidad (*visibility*)

La visibilidad de un extremo de asociación se indica mediante un signo especial o con un nombre de propiedad explícito delante del nombre de rol (ver **Figura 2.9**), y especifica la visibilidad de la asociación al transitarla en dirección hacia ese nombre de rol [UML, p. 3-73]. Las posibles visibilidades son [UML, p. 3-42]:

- `(+) public`
- `(~) package`
- `(#) protected`
- `(-) private`

La indicación de visibilidad puede ser suprimida [UML, p. 3-42], sin que eso signifique que la visibilidad está indefinida o es `public`: simplemente, no se desea mostrar en la vista actual.

2.3.2.8. Especificador de interfaz (*interface specifier*)

El nombre de rol puede ir seguido por el signo ':' y el nombre de un clasificador[13] para especificar la interfaz de la asociación (ver **Figura 2.10**). El especificador de interfaz indica el comportamiento que un objeto espera de la instancia enlazada, en otras palabras, el comportamiento requerido para posibilitar la asociación [UML, p. 3-72]. En este caso, la clase asociada normalmente tendrá otras funcionalidades adicionales que no son requeridas para esta asociación en concreto. La clase asociada debe ser compatible con el especificador de interfaz (que puede ser una interfaz, un tipo, u otra forma específica de clasificador). Si el especificador de interfaz se omite, la asociación puede usarse para obtener un acceso completo a la clase asociada (teniendo en cuenta las restricciones impuestas por los indicadores de visibilidad de cada elemento).

[13] El termino "clasificador" denota un concepto genérico en UML que engloba conceptos tales como clase, tipo, interfaz, tipo de datos, etc. Ver Apartado 2.4.1 para una definición más precisa.

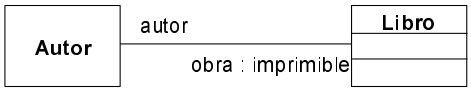

Figura 2.10. Ejemplo de asociación con especificador de interfaz

2.3.2.9. Restricción `{xor}`

Es posible establecer una restricción de exclusividad entre varias asociaciones que tengan un extremo conectado a una clase común mediante la restricción `{xor}`. Esto significa que en, un momento dado, una instancia de la clase común sólo puede participar en una de las asociaciones exclusivas [UML, p. 3-69]. La restricción se muestra mediante una línea discontinua que conecta las asociaciones exclusivas, etiquetada con la restricción `{xor}` (ver **Figura 2.11**). Los nombres de rol deben ser distintos en los extremos opuestos a la clase común. La restricción `{xor}` está predefinida en UML, y el usuario puede definir otras restricciones si lo precisa.

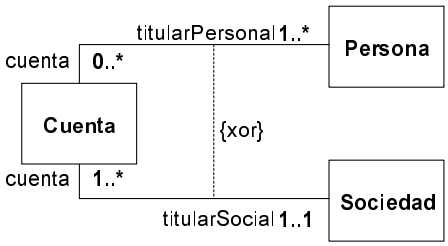

Figura 2.11. Restricción `{xor}`

2.3.3. Asociación reflexiva

Una asociación reflexiva, también denominada a veces asociación recursiva[14], es aquella en la que los dos extremos de la asociación están unidos a la misma clase (ver **Figura 2.12**). Los enlaces de una asociación reflexiva pueden conectar dos instancias diferentes de la misma clase, o incluso una instancia a sí misma. Este último caso se puede prohibir mediante una restricción si es preciso [UML, p. 3-68].

[14] El Estándar no usa los términos "asociación reflexiva" (*reflexive association*) ni "asociación recursiva" (*recursive association*) en ninguna de las dos secciones principales 2 y 3, dedicadas a la semántica y a la notación, pero sí los emplea en la sección 6, dedicada a la descripción esquemática de OCL [UML, pp. 6-14 y 6-15].

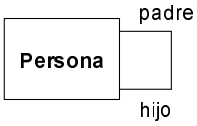

Figura 2.12. Ejemplo de asociación reflexiva en la que no debe haber ningún enlace entre una instancia y ella misma

2.3.4. Agregación y composición

La agregación es un tipo especial de asociación que se emplea para representar la relación entre un todo y sus partes; se indica mediante un rombo blanco en el extremo de la asociación unido a la clase que representa el "todo", también llamado "agregado" (*aggregate*) (ver **Figura 2.13**). Evidentemente, no se puede poner el rombo de agregación en los dos extremos de una asociación, ya que se trata de una relación antisimétrica. Si una asociación se define como agregación en el modelo, el símbolo de la agregación no se puede omitir en las vistas [UML, p. 3-72].

Dos o más agregaciones con el mismo "todo" se pueden dibujar en forma de árbol, juntando los extremos en uno solo, cuando coinciden las demás propiedades en el extremo agregado (multiplicidad, navegabilidad, etc.). La representación como árbol es meramente una opción de presentación, y no conlleva ninguna semántica especial [UML, p. 3-73].

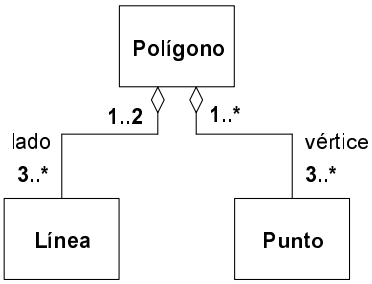

Figura 2.13. Ejemplo de dos agregaciones que no pueden dibujarse como un árbol porque no tienen la misma multiplicidad en el extremo agregado.

La composición es un tipo de agregación más fuerte, representada por un rombo negro en lugar de blanco, que se sitúa igualmente en el extremo unido al todo o "compuesto" (*composite*) (ver **Figura 2.14**). La composición implica que cada instancia-parte está incluida en un momento dado como máximo en una instancia-todo (compuesto), y que el compuesto tiene la responsabilidad exclusiva de la disposición de sus partes. La multiplicidad en el extremo compuesto no puede exceder de uno (es decir, las partes no se comparten entre distintos todos) [UML, p. 3-81].

En lugar de usar líneas terminadas en rombos negros en la forma convencional, la composición se puede representar mediante el anidamiento

de los símbolos de las partes dentro del símbolo del todo [UML, p. 3-81]. La multiplicidad de la parte respecto al compuesto se muestra en la esquina superior derecha de la clase anidada que representa la parte; si se omite, se asume que es 'muchos' por defecto. El nombre de rol de la parte respecto al todo se escribe delante del nombre de la clase anidada, separado por ':' (ver **Figura 2.14**). Nótese que la relación entre una clase y sus atributos es en la práctica una relación de composición.

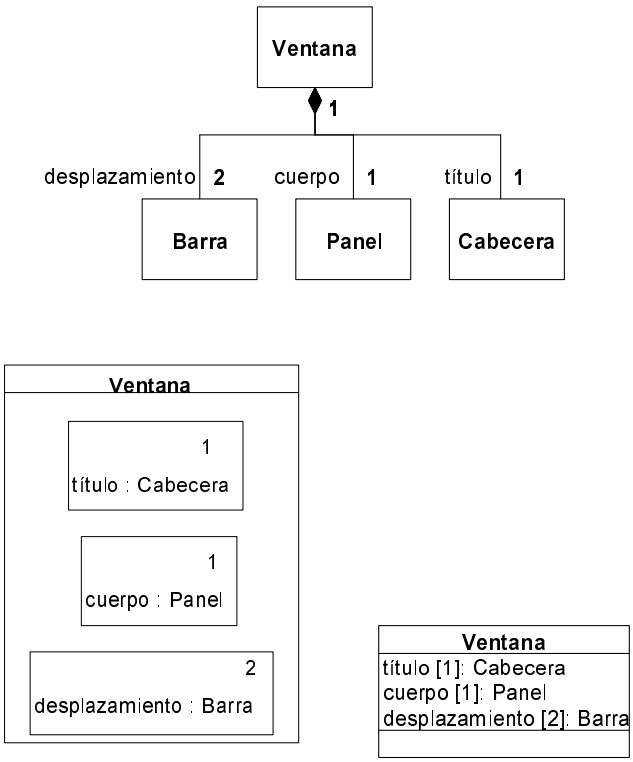

Figura 2.14. Ejemplo de composición representada en forma convencional (mediante asociaciones), en forma anidada, y en forma de atributos

Las partes de una composición pueden ser tanto clases como asociaciones. Una asociación como parte de un compuesto significa que tanto los objetos conectados como el propio enlace pertenecen todos al mismo compuesto. En la notación convencional, será necesario representar la asociación-parte como clase-asociación, para poder dibujar la composición con el todo mediante una línea terminada en rombo negro. Usando la notación anidada, una asociación entre dos partes dibujada dentro de los límites de la clase que representa el compuesto se considera que es parte de la composición. Por el contrario, si la asociación cruza el borde del compuesto, no será parte de la composición, y sus enlaces pueden establecerse entre partes de diferentes objetos compuestos.

2.3.5. Asociación cualificada

Un cualificador es un atributo (o lista de atributos) cuyos valores sirven para hacer una partición del conjunto de instancias asociadas con una instancia dada a través de una asociación. Los cualificadores son atributos de la asociación, no de las clases asociadas [UML, p. 3-76].

El cualificador se representa como un pequeño rectángulo insertado en el origen de la asociación cualificada, entre la clase y el extremo de la asociación, con el atributo (o lista de atributos) en su interior (ver **Figura 2.15**); el cualificador es parte de la asociación cualificada, no de la clase origen.

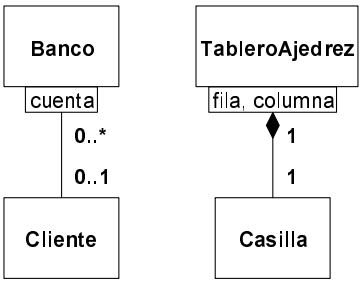

Figura 2.15. Ejemplos de asociaciones cualificadas

Una instancia de la clase origen, junto con un valor del cualificador, seleccionan una partición en el conjunto de instancias de la clase destino en el otro extremo de la asociación cualificada. La multiplicidad en el extremo destino de la asociación denota ahora la multiplicidad de la partición seleccionada; es decir, ya no denota cuántas instancias destino pueden estar asociadas con la instancia origen, sino cuántas pueden estar asociadas con la combinación de una instancia origen y un valor de cualificador.

Algunos valores típicos son:

- 0..1: se selecciona una única instancia destino, pero un valor cualquiera del cualificador no selecciona necesariamente una instancia destino.
- 1..1: cualquier posible valor del cualificador selecciona una única instancia destino; por tanto, el dominio de valores del cualificador debe ser finito.
- 0..*: el cualificador es un índice que realiza una partición de las instancias destino en subconjuntos.

Si una asociación se define con cualificador en el modelo, el símbolo del cualificador no se puede omitir en las vistas, ya que proporciona una información esencial para el carácter de la asociación [UML, p. 3-77].

2.3.6. Clase-asociación

Una clase-asociación es una asociación que también tiene propiedades de clase (o una clase que tiene propiedades de asociación). Aunque se

representa como una clase y una asociación, en realidad se trata de un único elemento del modelo [UML, p. 3-77].

Una clase-asociación se representa mediante un símbolo de clase (rectángulo) unido por una línea discontinua a un símbolo de asociación (línea continua), aproximadamente en su centro (ver **Figura 2.16**). El nombre mostrado en el rectángulo de la clase y el nombre mostrado en la línea de la asociación deben ser el mismo, ya que se trata de un único elemento; por tanto, uno de los dos nombres es redundante, aunque se pueden mostrar ambos. La línea discontinua no lleva ningún tipo de adorno.

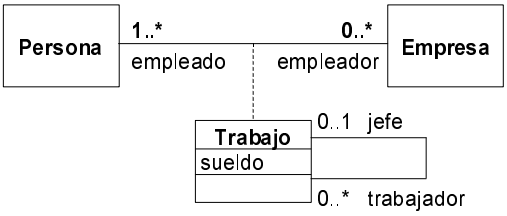

Figura 2.16. Ejemplo de clase-asociación

La clase-asociación puede tener las propiedades de cualquier asociación en sus extremos (multiplicidad, navegabilidad...), y el contenido de cualquier clase en su interior (atributos, operaciones); en cuanto que es una clase, también puede estar asociada a otras clases. Si la clase-asociación sólo tiene atributos, pero no operaciones ni asociaciones con otras clases, entonces se puede enfatizar su "naturaleza de asociación" poniendo el nombre solamente sobre la línea de la asociación; por el contrario, si tiene operaciones o asociaciones, se puede poner el nombre solamente en el rectángulo de la clase para enfatizar su "naturaleza de clase"; en ambos casos, sin embargo, la semántica es exactamente la misma.

El rectángulo de la clase puede omitirse en alguna vista, ya que la asociación sigue teniendo sentido como tal. En cambio, la línea de la asociación no puede omitirse en ninguna vista [3-78].

2.3.7. Asociación n-aria

Una asociación n-aria es una asociación entre tres o más clases; alguna de las clases puede participar más de una vez, con roles distintos, en la asociación. Cada instancia de la asociación es una n-tupla de valores de las respectivas clases [UML, p. 3-79]. Una asociación binaria se puede considerar que es un caso particular de asociación n-aria, con su propia notación.

Una asociación n-aria se representa mediante un rombo unido con una línea a cada clase participante (ver **Figura 2.17**). El nombre de la asociación se muestra junto al rombo. Cada extremo de la asociación puede tener adornos, como en una asociación binaria. En particular, se puede indicar la multiplicidad, pero ningún extremo puede tener cualificación ni agregación.

Se puede unir un símbolo de clase al rombo mediante una línea discontinua para indicar que se trata de una clase-asociación con atributos, operaciones o asociaciones con otras clases.

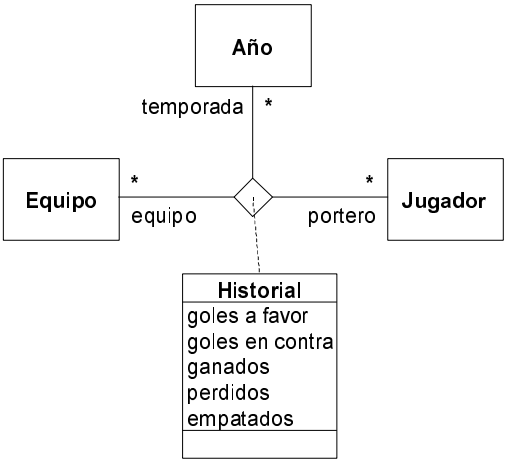

Figura 2.17. Una asociación ternaria que también es clase-asociación. En este ejemplo se muestra el historial de un equipo en cada temporada con un portero concreto. Se supone que el portero puede ser intercambiado durante la temporada y por tanto puede aparecer en varios equipos

Un enlace n-ario (instancia de asociación n-aria) se representa del mismo modo, mediante un rombo unido a cada una de las instancias participantes [UML, p. 3-85].

La multiplicidad se puede especificar en las asociaciones n-arias, pero su significado es menos obvio que la multiplicidad binaria. La multiplicidad de un rol representa el potencial número de instancias en la asociación cuando se fijan los otros N-1 valores [UML, p. 3-79].

2.4. La semántica de las asociaciones

Una vez presentada la notación de las asociaciones según el Estándar de UML, vamos a estudiar ahora su semántica, es decir, cómo es el metamodelo subyacente y cuál es el uso adecuado de las asociaciones en modelos descritos con UML.

2.4.1. Clases, atributos y operaciones

Como ya hemos visto, cada elemento del lenguaje (clase, asociación, atributo, operación...) está representado en el metamodelo por la correspondiente "metaclase" (`Class`, `Association`, `Attribute`, `Operation`...); análogamente, las relaciones entre los distintos elementos del lenguaje (por ejemplo, entre una asociación y sus extremos, o entre una clase y sus atributos) se representan mediante "meta-asociaciones"; etc. El vértice superior de la jerarquía de clases en el metamodelo lo ocupan las

clases `Element` y `ModelElement`. La **Figura 2.18** muestra una vista simplificada de la parte del metamodelo descrita en el paquete `Core` (ver **Figura 2.2**), donde podemos observar cómo son definidos los elementos `Class`, `Attribute` y `Operation` [UML, p. 2-13].

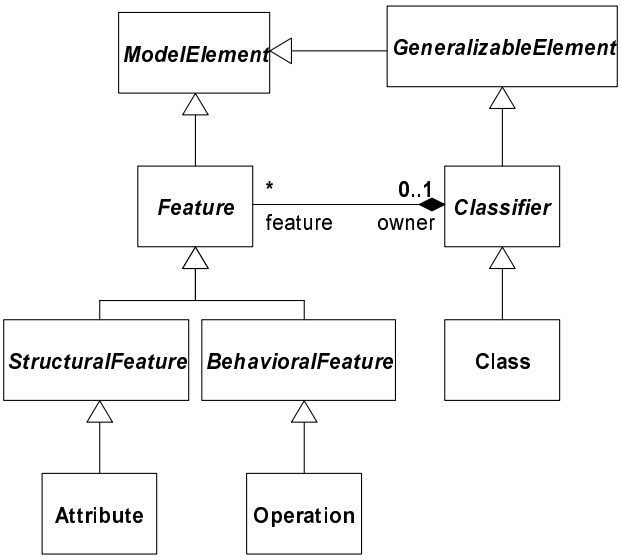

Figura 2.18. Metamodelo simplificado de `Class`, `Attribute` y `Operation`.

Como la clase y su contenido de atributos y operaciones no es el objeto principal de nuestro estudio, hemos omitido aquí muchas meta-relaciones y todos los meta-atributos, dejando sólo lo que hemos considerado indispensable para tener un conocimiento básico de su definición.

De acuerdo con la definición del Estándar [UML, p. 2-28], un clasificador (*classifier*) es un elemento que describe propiedades dinámicas y estructurales (*behavioral features*, *structural features*); se presenta en distintas formas específicas, como clase, tipo de datos, interfaz, componente, artefacto, etc.; la clase es la forma de clasificador que aparece más frecuentemente en los modelos concretos.

`Classifier` es una metaclase abstracta (esto se expresa escribiendo su nombre en cursiva), es decir, no puede tener instancias directas, sino sólo instancias indirectas a través de alguna de sus subclases que no sea a su vez abstracta; por ejemplo, `Class` (clase). Es decir, en un modelo concreto no puede haber clasificadores sin más, pero puede haber en cambio clases, interfaces, componentes, etc., que son formas concretas de clasificador. `Classifier` es subclase de `GeneralizableElement`; esto significa que los clasificadores en un modelo concreto (que son las "instancias" de la metaclase `Classifier`), pueden participar en relaciones de generalización con otros clasificadores (siempre que sean

clasificadores del mismo tipo: clase-clase, interfaz-interfaz, etc. [UML, p. 2-61]).

En el metamodelo, un `Classifier` posee por composición una colección de `Feature` (propiedad), tales como `Attribute` (atributo) y `Operation` (operación), que expresan las propiedades estructurales y dinámicas del clasificador.

2.4.2. Tipos de relaciones

El paquete `Core` contiene también la definición de los principales tipos de relaciones en UML: generalización, asociación y dependencia (hay algunos otros tipos que aquí omitimos por simplicidad). La **Figura 2.19** muestra una vista simplificada del metamodelo de estas relaciones [UML, pp. 2-14 y 2-15].

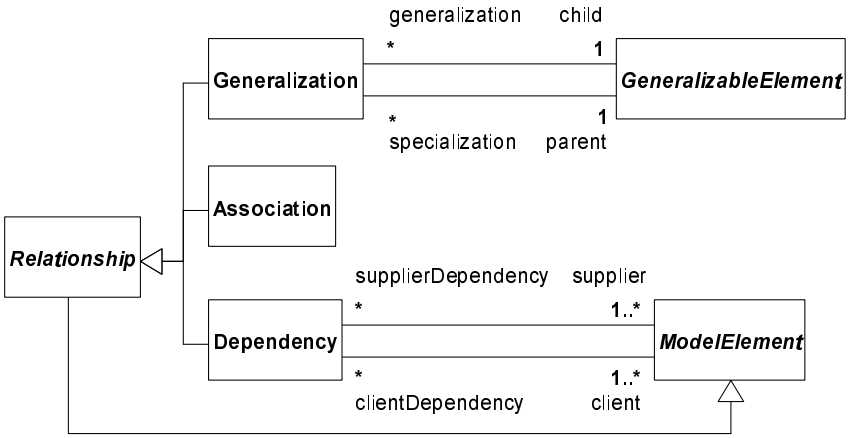

Figura 2.19. Principales tipos de relaciones en UML según el metamodelo

Podemos observar que `Relationship` (relación) es una metaclase abstracta, que es subclase de `ModelElement` y a su vez tiene tres subclases concretas: `Generalization` (generalización), `Association` (asociación) y `Dependency` (dependencia). Notemos cómo el metamodelo de una generalización y el de una dependencia son semejantes: una generalización se define entre dos elementos (que sean subclases de `GeneralizableElement`, como por ejemplo `Classifier`), uno de ellos como padre y el otro como hijo; una dependencia se define de modo semejante entre elementos cualesquiera del modelo, con la diferencia de que tanto el origen como el destino de la dependencia pueden ser múltiples.

El metamodelo de una asociación es algo más complejo, debido a que los *extremos de la asociación* tienen cierta entidad propia, y por tanto se requiere una metaclase especial `AssociationEnd` para representarlos. Gráficamente, esto se traduce en que la generalización y la dependencia se representan en un diagrama UML mediante flechas desnudas en sus

extremos, mientras que la línea de una asociación puede llevar distintos tipos de adornos gráficos en sus extremos, que contienen la información más relevante de la asociación. La **Figura 2.20** muestra una vista simplificada del metamodelo de las asociaciones [UML, p. 2-14][15].

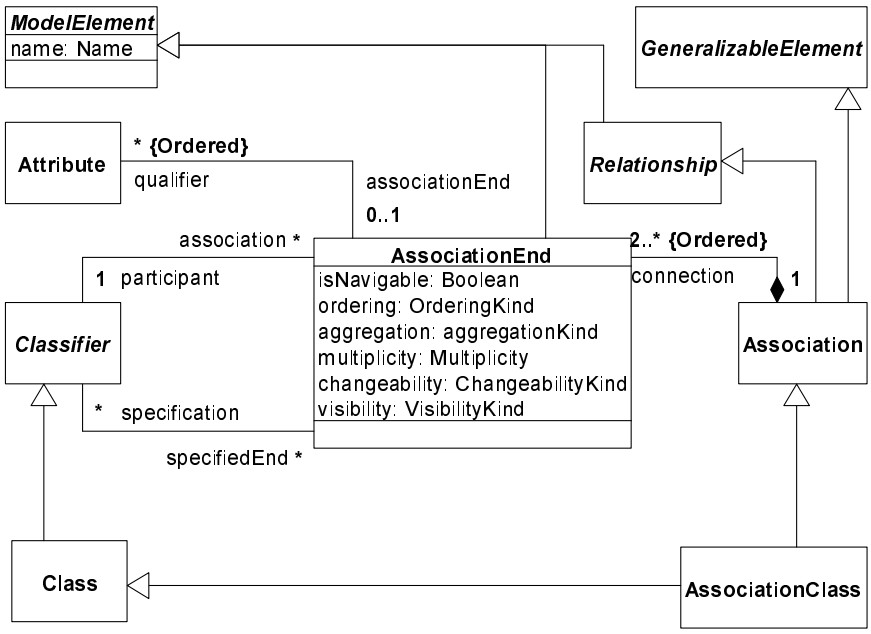

Figura 2.20. Metamodelo de las asociaciones en UML

2.4.3. La metaclase `Association`

Una asociación define una relación semántica entre dos o más clasificadores (por ejemplo, clases). La asociación representa un conjunto de conexiones entre instancias de los clasificadores. Una instancia de una asociación es un enlace (*link*), que es una tupla de instancias de los correspondientes clasificadores. Cada valor de tupla puede aparecer como máximo una vez en cada asociación [UML, pp. 2-19 y 2-20].

En el metamodelo (ver **Figura 2.20**) podemos observar que una asociación tiene al menos dos extremos (rol `connection`); serán exactamente dos para *asociaciones binarias*, y más de dos para *asociaciones n-arias*. Cada extremo está conectado a un clasificador (rol `participant`), y un clasificador puede estar conectado a varias asociaciones, o incluso a más de un extremo en la misma asociación (rol `association`); así ocurre en *asociaciones reflexivas*, y puede ocurrir también en asociaciones n-arias.

---

[15] Se ha omitido el meta-atributo `targetScope` de la metaclase `AssociationEnd`, ya que no es necesario para la presente exposición.

El *nombre de la asociación* que figura en un diagrama se corresponde con el meta-atributo `name`, heredado de la metaclase `ModelElement`. La *dirección* en la que debe ser leído el nombre en una asociación binaria se obtiene del orden de los extremos (rol `connection`): el triángulo que representa la dirección de la asociación va desde el primer extremo hacia el segundo extremo [UML, p. 3-69].

Además de ser subclase de `Relationship`, `Association` es también subclase de `GeneralizableElement` (nótese, sin embargo, que no es subclase de `Classifier`). Esto significa que se pueden establecer relaciones de generalización entre asociaciones, aunque no existe de momento una notación muy adecuada para representarlo (habría que representar las dos asociaciones involucradas como clases-asociación para poder dibujar la generalización entre ellas, aunque no tengan atributos ni operaciones). La asociación hija hereda los extremos de su madre y debe mantener el mismo número de extremos (no puede añadir ni eliminar); cada clase participante debe ser hija de la clase participante que se encuentra en la misma posición en la asociación madre. Si se trata de una clase-asociación, se heredan también los atributos y operaciones, etc. El Estándar reconoce que la generalización de asociaciones no está bien definida y se espera que en la versión 2.0 de UML se avanzará en su especificación [UML, p. 2-21].

2.4.4. La metaclase `AssociationEnd`

Un extremo de asociación es la terminación de una asociación, que conecta la asociación a un clasificador. Cada extremo de asociación es parte de una y sólo una asociación, y los extremos de una asociación están ordenados (véase la meta-composición `AssociationEnd-Association`). El extremo de la asociación especifica cómo se conecta la asociación al clasificador por medio de los distintos meta-atributos y meta-asociaciones de `AssociationEnd` (ver **Figura 2.20**), como vamos a ver a continuación [UML, pp.2-21 a 2-24]:

2.4.4.1. Nombre de rol

El meta-atributo `name`, heredado de la metaclase `ModelElement`, especifica el nombre del extremo de asociación, representado en un diagrama como nombre de rol. El nombre de rol proporciona una forma de transitar la asociación desde la instancia origen[16] hacia la instancia o conjunto de instancias destino. El nombre de rol representa un pseudo-atributo del clasificador origen, es decir, puede usarse de la misma manera

[16] En ésta y en las siguientes descripciones, "destino" (*target*) se refiere al extremo cuyas propiedades se describen, y "origen" (*source*) al extremo opuesto. En este caso concreto, el nombre de rol denota el nombre del extremo destino visto desde el extremo origen.

que un atributo, y debe ser único con respecto a otros atributos y pseudo-atributos del clasificador origen.

2.4.4.2. Multiplicidad

El meta-atributo `multiplicity` especifica el número de instancias destino que pueden estar asociadas con una única instancia origen a través de esta asociación.

El meta-atributo es de tipo `Multiplicity`, que en el Estándar se define como una colección no vacía de "rangos de multiplicidad" [UML, p. 2-90]. En cada rango se define un límite inferior (cualquier entero no negativo: 0, 1, 2...) y un límite superior (cualquier entero positivo: 1, 2, 3, ..., además del valor especial *unlimited*, que indica que no hay límite superior en el rango, representado gráficamente como un asterisco, '*').

2.4.4.3. Ordenación

El meta-atributo `ordering` especifica si el conjunto de enlaces desde la instancia origen hacia las instancias destino está ordenado. La ordenación debe ser determinada y mantenida por las operaciones que añaden enlaces. La ordenación es una información adicional que no es inherente a los objetos ni a los enlaces mismos. Los posibles valores (además de otros que pueda definir el usuario) son:

- `unordered`: los enlaces forman un conjunto no ordenado.
- `ordered`: el conjunto de enlaces puede ser recorrido en orden.

2.4.4.4. Modificabilidad

El meta-atributo `changeability` especifica si una instancia de la asociación puede ser modificada por una instancia del clasificador origen, es decir, controla el acceso a la asociación por operaciones en el extremo opuesto. Los posibles valores son:

- `changeable`: sin restricciones.
- `frozen`: las operaciones del clasificador origen no pueden añadir enlaces después de la creación de la instancia origen (las operaciones del clasificador destino podrían añadirlos en sentido contrario si la restricción fuera distinta en el otro extremo).
- `addOnly`: las operaciones del clasificador origen pueden añadir enlaces en cualquier momento, pero no pueden eliminarlos (como en el caso anterior, las operaciones del clasificador destino pueden tener una restricción distinta).

2.4.4.5. Navegabilidad

El meta-atributo `isNavigable` especifica si es posible recorrer la asociación desde una instancia origen hacia sus instancias enlazadas. Cada dirección se especifica por separado en extremos opuestos. El valor `true` significa que la asociación es navegable desde el clasificador origen, y

entonces el nombre de rol destino puede emplearse en expresiones de navegación.

2.4.4.6. Visibilidad

El meta-atributo `visibility` especifica la visibilidad del extremo de asociación desde el punto de vista del clasificador en el otro extremo. Los valores posibles son:

- `public`: otros clasificadores pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo público.
- `package`: los clasificadores que están en el mismo paquete (o subpaquete anidado, hasta cualquier nivel) que la declaración de la asociación pueden navegar la asociación y usar el nombre de rol en expresiones.
- `protected`: los descendientes del clasificador origen pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo protegido.
- `private`: sólo el clasificador origen puede navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo privado.

2.4.4.7. Agregación y composición

El meta-atributo `aggregation` especifica si el clasificador destino es una agregación con respecto al clasificador origen. Sólo un extremo puede ser agregación; si la asociación es n-aria, ningún extremo puede ser agregación [UML, p. 2-52]. Los valores posibles son:

- `none`: el clasificador destino no es un agregado.
- `aggregate`: el clasificador destino es un agregado; por tanto, el clasificador origen es una parte, y debe tener el valor `none` para la agregación. La parte puede estar contenida en otros agregados.
- `composite`: el clasificador destino es un compuesto; por tanto, el clasificador origen es una parte, y debe tener el valor `none` para la agregación. La parte es fuertemente poseída por el compuesto y no puede ser parte de ningún otro compuesto.

En UML, el significado de la agregación es intencionadamente vago para que pueda ser aplicado en distintas áreas [UML, p. 2-66]. En cambio, la composición se define de forma más precisa con las siguientes características [UML, p. 2-66]:

- Una instancia-parte puede estar incluida como mucho en una instancia-todo (multiplicidad máxima uno en el extremo del compuesto).
- El objeto compuesto tiene la responsabilidad exclusiva de la disposición de sus objetos-parte, es decir, es responsable de la creación y destrucción de las partes.

- Si un objeto compuesto es destruido, todas sus partes deben ser destruidas.
- El compuesto puede separar una parte y entregarla a otro compuesto, que entonces se hace responsable de la parte.
- Si la multiplicidad en el extremo compuesto es 0..1, el compuesto puede separar la parte y la parte puede hacerse responsable de sí misma; en caso contrario la parte no puede existir separada del compuesto.

Como consecuencia de estas reglas, la composición implica una *semántica de la propagación*, es decir, la semántica dinámica del todo es parcialmente propagada a sus partes. Por ejemplo, si se copia o destruye el todo, entonces las partes también se copian o destruyen (ya que una parte sólo puede pertenecer como máximo a un todo compuesto).

Por el contrario, la agregación normal denota una posesión débil, es decir, la parte puede pertenecer simultáneamente a varios agregados, y la destrucción del agregado no implica la destrucción de las partes.

Tanto la agregación como la composición definen una relación transitiva y antisimétrica, es decir, las instancias forman un grafo dirigido acíclico.

2.4.4.8. Especificador de interfaz

La meta-asociación desde `AssociationEnd` (rol `specifiedEnd`) hacia `Classifier` (rol `specification`) determina cero o más clasificadores que especifican las operaciones que pueden ser aplicadas a una instancia del clasificador destino que sea accedida a través de este extremo de asociación. Estos clasificadores-especificadores determinan la interfaz mínima que debe ser realizada por el clasificador que de hecho se conecte al extremo de la asociación, para cumplir con la finalidad de la asociación; estos clasificadores-especificadores no indican las clases que participan en una asociación, sino sólo las operaciones que pueden ser invocadas por medio de ella.

2.4.4.9. Asociación cualificada

La meta-asociación desde `AssociationEnd` (rol `associationEnd`) hacia `Attribute` (rol `qualifier`) determina una lista opcional de atributos cualificadores para el extremo de asociación. Si la lista está vacía, entonces la asociación no está cualificada.

El cualificador define una partición en el conjunto de instancias asociadas con respecto a una instancia dada en el extremo cualificado [UML, p. 2-67]. Una instancia del cualificador comprende un valor para cada atributo cualificador. Dado un objeto origen y una instancia del cualificador, el número de objetos en el otro extremo de la asociación queda restringido a la multiplicidad especificada (la multiplicidad "pura", sin cualificador, se asume que es 0..*). En el caso más común, cuando la

multiplicidad es 0..1, el valor del cualificador es único con respecto al objeto cualificado, y designa como máximo un objeto asociado en el otro extremo. En el caso más general de multiplicidad 0..*, el conjunto de instancias asociadas sufre una partición en subconjuntos, cada uno de los cuales es seleccionado por una instancia del cualificador.

2.4.5. La metaclase `AssociationClass`

Una clase-asociación es una asociación que a la vez es una clase. No sólo conecta dos o más clasificadores, sino que también define un conjunto de propiedades (estructurales y dinámicas) que pertenecen a la asociación en sí misma, no a los clasificadores [UML, p. 2-21].

Como podemos observar en el metamodelo (ver **Figura 2.20**), la metaclase `AssociationClass` es simultáneamente subclase de `Class` y de `Association`, por lo tanto posee por composición una colección de `Feature` (ver **Figura 2.18**) que expresan sus propiedades estructurales o dinámicas, y una colección de `AssociationEnd` (ver **Figura 2.20**) que especifican cómo se conecta a cada clasificador. Una clase-asociación no puede definirse entre ella misma y otro clasificador [UML, p. 2-53].

SEGUNDA PARTE:

DESARROLLO TEÓRICO

El concepto de asociación en UML
Aspectos estáticos y dinámicos

3. La multiplicidad de las asociaciones

3.1. Introducción

El concepto de multiplicidad en UML deriva de concepto de cardinalidad en las técnicas basadas en el modelo Entidad/Relación. La documentación de UML define este concepto pero al mismo tiempo reconoce una cierta falta de claridad en la especificación de multiplicidades para asociaciones n-arias. La multiplicidad en otros tipos de asociaciones también plantea algunos problemas de interpretación, particularmente en las asociaciones cualificadas y clases-asociación.

El modelo Entidad/Relación [Chen 76] ha sido ampliamente usado en el análisis estructurado y en el modelado conceptual, y ha evolucionado hacia los diagramas de clases orientados a objetos tales como los del Lenguaje Unificado de Modelado. Este enfoque es fácil de entender, potente para modelar problemas del mundo real, y sencillo de traducir en un esquema de base de datos, aunque otras formas de implementación, como los lenguajes orientados a objetos, no son tan simples y directas [Rumbaugh 87]. Tanto en los diagramas Entidad/Relación como en los diagramas de clases, las principales construcciones son la entidad y la relación entre entidades (clase y asociación, en la terminología de UML). En este sentido, importantes autores rechazan con fuerza el enfoque Entidad/Relación, ya que consideran que la misma distinción entre entidad y relación es poco precisa [Codd 90, Date 95].

Para muchos analistas, uno de los aspectos más problemáticos del modelado de sistemas es la correcta comprensión de las asociaciones ternarias y, en general, de las asociaciones n-arias (asociaciones con tres o más roles). Las asociaciones ternarias representan a menudo una situación compleja que los modeladores encuentran especialmente difícil de entender, en lo que se refiere tanto al modelado estructural como al dinámico. Desde el punto de vista estructural, estas dificultades a menudo están relacionadas con la cuarta y quinta formas normales [Hitchman 99]. Desde el punto de vista dinámico, las interacciones atómicas que implican más de dos objetos son otra fuente de complejidad conceptual[17].

---

[17] Algunos autores denominan estas interacciones "acciones conjuntas" (*joint actions*) o "interacciones atómicas síncronas multivía" (*atomic multiway synchronous interactions*) [Genilloud 98].

La cardinalidad de una relación (multiplicidad de una asociación, en la terminología de UML) es considerada por algunos expertos como la principal propiedad estructural de un modelo [Kilov 94]. Sin embargo, los valores de multiplicidad típicamente especificados para las asociaciones n-arias proporcionan sólo un entendimiento parcial de la estructura de objetos. Es posible incluir otras condiciones adicionales mediante descripciones textuales que acompañan a los modelos, pero siempre, en la medida de lo posible, es deseable una mejor integración. Peor aún, a menudo se comprende muy mal el significado mismo de las multiplicidades. Como veremos, la multiplicidad de las asociaciones binarias es más bien sencilla de especificar y entender en UML, pero desafortunadamente éste no es el caso para las asociaciones n-arias, para las cuales UML ha definido unas multiplicidades incompletas y poco claras.

Por otra parte, la multiplicidad en asociaciones binarias tampoco está exenta de dificultades, principalmente debido a una excesiva influencia del concepto de "relación" (*relation*)[18] tal como es entendido en las metodologías de diseño de bases de datos, que ha resultado en la incorrecta identificación de los conceptos de "enlace entre objetos" y "tupla de objetos", con la consiguiente prohibición de que existan enlaces repetidos en una asociación (es decir, enlaces que conectan los mismos dos objetos).

El propósito de este Capítulo es, en primer lugar, clarificar el significado de los valores de multiplicidad n-arios, que UML reconoce no ser claro [UML, p. 3-79], y proponer una extensión a la notación de las multiplicidades n-arias de UML que proporcione definiciones más precisas y completas para las asociaciones n-arias con la mínima carga notacional. Daremos un repaso a distintos enfoques para valores de multiplicidad e asociaciones ternarias y n-arias, mostrando las paradojas y ambigüedades de las multiplicidades n-arias. Aunque nuestro interés se centra en UML, el núcleo de nuestra exposición es suficientemente general como para ser útil en otros métodos basados en el enfoque Entidad/Relación. En segundo lugar, nos proponemos también analizar el concepto de "instancia de una asociación" desde el punto de vista de la multiplicidad, clarificando los conceptos de "enlace" y "tupla" y justificando con diversos argumentos que se deben permitir los enlaces repetidos en una asociación.

El resto de este Capítulo está organizado como se describe a continuación. La Sección 3.2 busca las raíces de la definición de multiplicidad en UML en las técnicas de modelado de datos que se derivan principalmente del enfoque Entidad/Relación de Chen. La Sección 3.3 está

---

[18] El término castellano "relación" traduce correctamente dos términos ingleses, *relation* y *relationship*; el primero tiene un sentido más matemático, y el segundo un sentido más general. A su vez, *relationship* se traduce a menudo como "interrelación", especialmente en el contexto del modelo Entidad/Relación (también llamado modelo Entidad/Interrelación).

dedicada al estudio de la multiplicidad en las asociaciones n-arias[19]: la ambigüedad de la definición de la multiplicidad mínima, que permite tres interpretaciones distintas incompatibles (cada una con sus propios problemas y consecuencias inesperadas), y la necesidad de expresar la "restricción de participación" (olvidada en UML) con una notación adecuada; se propone una notación compatible con las tres interpretaciones alternativas presentadas, y una breve propuesta de los cambios que deberían hacerse en la definición de la multiplicidad n-aria en el Estándar de UML. La Sección 3.4 trata de la multiplicidad de las asociaciones binarias "especiales" (asociación cualificada y clase-asociación). Finalmente, la Sección 3.5 se centra en el estudio de la definición de asociación en UML como "conjunto de tuplas no repetidas", partiendo de los problemas que esta definición plantea en asociaciones cualificadas y clases-asociación. El análisis conduce a una distinción más clara de los conceptos de "enlace" y "tupla", y a la recomendación de eliminar la restricción de que no pueda haber enlaces repetidos.

3.2. Definiciones de multiplicidad

3.2.1. Definición de multiplicidad en UML

UML define la "multiplicidad" como el rango de cardinalidades permitidas en un conjunto [UML, p. 3-75], donde "cardinalidad" es el número de elementos en el conjunto [UML, p. B-4]. Una cardinalidad es un valor específico ('3'), mientras que una multiplicidad es el rango de posibles cardinalidades que puede tener un conjunto ('1..10'). Las especificaciones de multiplicidad se dan principalmente para los extremos de asociación, pero también se usan con otros fines, como repeticiones de mensajes, número de instancias que puede tener una clase, etc.

La multiplicidad de una asociación binaria, situada junto al extremo destino de la asociación, especifica el número de instancias destino que pueden estar asociadas con una única instancia origen a través de la asociación dada, en otras palabras, cuántos objetos de la clase destino (*target*) pueden estar asociados con un único objeto dado de la clase origen (*source*) [RM, p. 348; UML, p. 2-23].

El clásico ejemplo de la **Figura 3.1** ilustra la multiplicidad binaria. Cada instancia de `Persona` puede trabajar para cero o una instancias de `Empresa` (0..1), mientras que cada empresa puede estar enlazada con una o más personas (1..*).

[19] Esta parte del Capítulo corresponde a un trabajo publicado por el autor de esta Tesis Doctoral [Génova 02b].

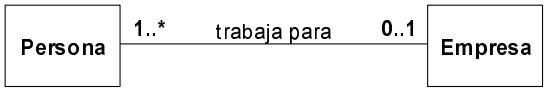

Figura 3.1. Un clásico ejemplo de asociación binaria con la expresión de las multiplicidades.

Encontramos ya aquí un problema de la multiplicidad, que es el tratamiento del tiempo, o más exactamente, el registro histórico, que se manifiesta en multitud de problemas de modelado de datos. Nótese que en la asociación del ejemplo sólo se puede representar la situación presente: "una persona *está trabajando* para 0..1 empresas", pero no "una persona *ha trabajado o trabaja* para varias empresas, aunque en un momento dado sean sólo 0..1 empresas". Al enlazar la persona con la empresa para la que trabaja actualmente, se pierde la historia de sus anteriores empresas. Para representar una relación semántica entre `Persona` y `Empresa` que incluya lógica temporal (es decir, el intervalo de tiempo en el que se considera que el predicado es válido) tendríamos que: a) sustituir la multiplicidad 0..1 del extremo derecho por 0..*; b) transformar la asociación en clase-asociación para añadirle la información relativa al periodo de validez del hecho representado; y c) añadir una restricción que impida la coexistencia de dos enlaces con periodo de tiempo superpuesto. Aun así, no podríamos tener una persona trabajando para la misma empresa en dos periodos distintos de tiempo, debido a la restricción antes mencionada de que en una asociación no puede haber tuplas repetidas (ver Apartado 1.4.3). En UML no tenemos una forma adecuada de tratar la multiplicidad en el tiempo. Volveremos sobre este problema en la Sección 3.4.

Veamos ahora lo que ocurre con las asociaciones n-arias. Una asociación n-aria es una asociación entre tres o más clases[20], y se representa mediante un rombo unido con un trazo a cada clase participante. *Cada instancia de la asociación es una n-tupla de valores de las respectivas clases* (un trío, en el caso de asociaciones ternarias). La multiplicidad para asociaciones n-arias puede ser especificada, pero es menos obvia que la multiplicidad binaria. La multiplicidad de un extremo de asociación representa *el número potencial de valores en un extremo, cuando se fijan los valores en los otros n-1 extremos* [UML, p. 3-79]. Esta definición es compatible con la de multiplicidad binaria [RM, p. 350].

El ejemplo de la **Figura 3.2**, tomado del Manual de Referencia de UML y del Estándar de UML [RM, p. 351; UML, p. 3-80], muestra el registro de un equipo de fútbol en cada temporada con un portero concreto. Se asume que el portero puede ser intercambiado durante la temporada y puede aparecer en varios equipos. Es decir, para un jugador y año dados,

[20] En realidad, *clasificadores*. Por simplicidad, frecuentemente diremos "clase" en lugar de "clasificador".

puede haber muchos equipos, y así para las otras multiplicidades establecidas en el diagrama.

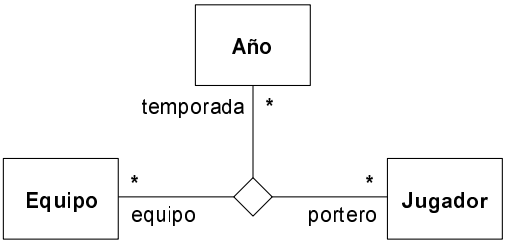

Figura 3.2. Asociación ternaria con multiplicidades muchos-muchos-muchos.

No obstante, como veremos en la Sección 3.3, detrás de la aparente claridad de estas especificaciones de multiplicidad se esconden sutiles paradojas.

3.2.2. Definición de multiplicidad en otras técnicas de modelado de datos

La definición de "multiplicidad de una asociación" en UML está basada en la definición de OMT [Rumbaugh 91], que a su vez se deriva, como generalmente es reconocido [Castellani 00, McAllister 95], de la definición de "cardinalidad de una relación" en el modelo Entidad/Relación [Chen 76][21]. De hecho, Chen no usa el término "cardinalidad" en su propuesta: sólo usa las expresiones "correspondencia 1:1", "correspondencia 1:n" y "correspondencia m:n", y explica el significado de cada una, pero sin dar ninguna definición formal del concepto de "tipo de correspondencia" (*type of mapping*). Más aún, Chen presenta un ejemplo de relación ternaria m:n:p, `proveedor-proyecto-pieza`, pero no explica cómo deben entenderse estas "cardinalidades". Nótese también que Chen sólo trata con la "cardinalidad máxima", que está estrechamente relacionada con el concepto de "dependencia funcional".

Muchas técnicas de modelado han continuado, formalizado y extendido el estilo de Chen para representar los valores de multiplicidad [De Miguel 99, Elmasri 94, Martin 95, Métrica 93, Teorey 99]. Otras notaciones, siguiendo al método francés Merise [Tardieu 85], invierten la posición de los valores de multiplicidad [Batini 92, Ceri 97, Coad 91], siguiendo el convenio "extremo cercano" en lugar del convenio "extremo lejano", que es el usado en UML. Está bien establecido que la semántica de ambas convenciones es equivalente para relaciones binarias, pero difiere

[21] La expresión "cardinalidad de una relación" en la terminología de Chen, equivale aproximadamente a "multiplicidad de una asociación" en la terminología de UML. En este Capítulo usamos una u otra terminología dependiendo del contexto: "multiplicidad de una asociación" en UML, y "cardinalidad de una relación" en otras técnicas de modelado de datos.

sustancialmente cuando se aplican a relaciones de mayor grado [Castellani 00, Martínez 99, McAllister 95, Song 95]. Trataremos con más detalle este tema en el Apartado 3.3.2.

En algunos de estos métodos encontramos una explícita y útil distinción entre los conceptos de "restricción de cardinalidad" y "restricción de participación" [Elmasri 94, Martin 95, Song 95]:

- La *restricción de cardinalidad* (*cardinality constraint*) especifica el número de instancias de la relación en las que una entidad puede participar. Tienen la forma 1:1, 1:N o M:N para expresar las dos restricciones en una relación binaria, y 1:1:1, 1:1:N, 1:M:N o M:N:P para expresar las tres restricciones en una relación ternaria. Estas restricciones se corresponden con la restricción de cardinalidad *máxima* en algunas notaciones. En el estilo de Chen, la entidad con una restricción de cardinalidad 1 es *funcionalmente dependiente* de la otra entidad (o entidades, en una relación n-aria). En el ejemplo de la **Figura 3.1**, `Empresa` es funcionalmente dependiente de `Persona`.
- La *restricción de participación* (*participation constraint*) especifica si una instancia de entidad puede existir sin participar en una relación con otra entidad. Esta restricción se corresponde con la cardinalidad *mínima* en algunas notaciones. Hay dos tipos de participación, total y parcial, también denominadas *obligatoria* y *opcional*. La participación obligatoria (*mandatory*) ocurre cuando una instancia de entidad no puede existir sin participar en una relación con otra instancia de entidad. La participación opcional (*optional*) ocurre cuando la instancia de entidad puede existir sin participar con otra instancia de entidad. En el ejemplo de la **Figura 3.1**, `Empresa` tiene participación obligatoria, mientras que `Persona` tiene participación opcional.

Otros autores definen de modo más general la restricción de co-ocurrencia (*co-occurrence constraint*), que especifica cuántos objetos (o n-tuplas de objetos) pueden co-ocurrir en una relación con otro objeto (o m-tuplas de objetos) [Embley 98]: por ejemplo, cuántos pares `producto-precio` pueden co-ocurrir con un par concreto `vendedor-comprador` en la relación cuaternaria `venta`. Este concepto generalizado de cardinalidad se estudia con más detenimiento en el Apartado 3.3.2.

Algunos métodos combinan las restricciones de cardinalidad y de participación y las representan usando restricciones mínimas y máximas con la notación `(min, max)`. Éste es el caso de UML. Sin embargo, el concepto de multiplicidad mínima no es equivalente al concepto de restricción de participación. Por ejemplo, una multiplicidad mínima 2 implica participación obligatoria, pero la participación obligatoria implica sólo una multiplicidad mínima 1. Más aún, la multiplicidad en UML puede

ser cualquier subconjunto de los enteros no negativos[22], no sólo un único intervalo como (2..*), o una lista de intervalos enteros separados por comas como (1..3, 7..10, 15, 19..*): también son válidas las especificaciones de multiplicidad tales como {números primos} o {cuadrados de enteros positivos}, aunque no hay notación estándar para ellas ni un soporte adecuado en el metamodelo, que representa internamente la multiplicidad como un rango o una lista de rangos. No obstante, en UML, como en otras técnicas de modelado, las multiplicidades más habituales son (0..1), (1..1), (0..*) y (1..*). Nuestro análisis quedará restringido a estas combinaciones de valores de multiplicidad.

3.3. La multiplicidad de las asociaciones ternarias

3.3.1. Paradojas y ambigüedades de las multiplicidades ternarias

Recordemos la definición de multiplicidad de una asociación n-aria en UML: "la multiplicidad de un extremo de asociación representa el número potencial de valores en un extremo cuando se fijan los valores en los otros n-1 extremos" [UML, p. 3-79]. Consideremos ahora la asociación ternaria "A trabaja en B usando C", que es un clásico ejemplo en la literatura [McAllister 95], definida entre empleados, proyectos y habilidades: un empleado trabaja en cierto proyecto usando cierta habilidad (ver **Figura 3.3**). La **Tabla 3.1** ilustra tres posibles conjuntos de instancias de las tres clases, mientras que la **Tabla 3.2** ilustra un posible conjunto de instancias (tríos) de la asociación.

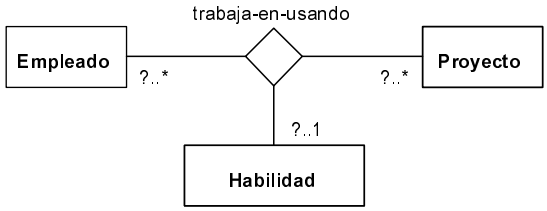

Figura 3.3. La asociación ternaria `trabaja-en-usando`, en la que se expresan sólo las multiplicidades máximas

Empleado	Proyecto	Habilidad
Alberto	Cocina	Soldadura
Benito	Laboratorio	Pintura
Clara	Garaje	Carpintería

Tabla 3.1. Tres posibles conjuntos de instancias de las tres clases `Empleado`, `Proyecto` y `Habilidad`

[22] Al menos, esto es lo que dan a entender el Manual de Referencia y el Estándar: "Esencialmente, la multiplicidad es un subconjunto (posiblemente infinito) de los enteros no negativos" [RM, p. 346; UML, pp. 3-75 y B-13].

— trabaja en — usando —		
Empleado	**Proyecto**	**Habilidad**
Alberto	Cocina	Soldadura
Alberto	Laboratorio	Soldadura
Benito	Cocina	Carpintería
Benito	Garaje	Carpintería
Clara	Cocina	Pintura

Tabla 3.2. Un posible conjunto de instancias de la asociación ternaria `trabaja-en-usando`

La **Figura 3.3** muestra un diagrama de esta asociación ternaria, con restricciones de multiplicidad máxima que, de acuerdo con la definición dada más arriba, son consistentes con los valores de la **Tabla 3.1** y la **Tabla 3.2**:

- La multiplicidad máxima para la clase `Proyecto` en la asociación `trabaja-en-usando` es *, ya que una n-tupla de instancias de Empleado-Habilidad puede estar enlazada a un máximo ilimitado de instancias de `Proyecto`: la tupla `Alberto-Soldadura` está enlazada a dos instancias diferentes de `Proyecto`, `Cocina` y `Laboratorio`, y lo mismo ocurre con `Benito-Carpintería`.
- La multiplicidad máxima para la clase `Empleado`, también *, es igualmente consistente: aunque no hay ningún par `Proyecto-Habilidad` enlazado con dos instancias diferentes de `Empleado`, el diagrama determina que una tupla tal como `Clara-Laboratorio-Soldadura`, que duplicaría el par `Laboratorio-Soldadura`, puede ser añadida al actual conjunto de tuplas.
- Finalmente, la multiplicidad máxima para la clase `Habilidad`, en este caso 1, determina que para cada par `Empleado-Proyecto` puede haber como máximo una habilidad: es decir, un empleado usa como máximo una habilidad en cada proyecto, lo cual es consistente con los valores dados, pero la restricción también prohíbe añadir una tupla como `Clara-Cocina-Soldadura` a menos que antes sea borrada la tupla `Clara-Cocina-Pintura`. En otras palabras, `Habilidad` es funcionalmente dependiente de `Empleado-Proyecto`.

Y bien, todo parece funcionar sin problemas... pero no es tan sencillo. Hasta ahora, al aplicar la definición a este ejemplo hemos considerado sólo la multiplicidad máxima. Concentrémonos ahora en la multiplicidad mínima, y veremos que hay una cierta ambigüedad en su definición. Vamos a proponer y examinar tres interpretaciones diferentes de la multiplicidad mínima de la asociación, estudiando el problema desde el punto de vista de la frase "cada par `Empleado-Proyecto`", e invitando al lector a que

compruebe cuál de ellas ha aceptado hasta ahora, probablemente de manera inconsciente. Mostraremos que cada una de estas tres interpretaciones tiene sus propios problemas y consecuencias inesperadas.

Primera interpretación (tuplas reales). "Cada par `Empleado-Proyecto`" puede ser entendido como un "par realmente existente", o par real (*actual pair*), es decir, un par de instancias que están enlazadas por algún enlace ternario perteneciente a la asociación ternaria. Los pares `Alberto-Cocina` y `Benito-Garaje` son pares reales, ya que existen de hecho algunos tríos que los contienen, mientras que los pares `Alberto-Garaje` y `Clara-Laboratorio` no lo son.

Esta interpretación de la regla parece más bien intuitiva, pero... nótese que para un par real `Empleado-Proyecto` *debe* existir siempre al menos una `Habilidad`: si es un par real, entonces existe un trío real que lo contiene, por tanto hay una instancia de `Habilidad` que también está en el trío. No puede haber un par real que no esté conectado a un tercer elemento, porque un enlace ternario es por definición un trío de valores de las clases respectivas; un enlace ternario tiene tres "patas", y ninguna de ellas puede estar vacía: los enlaces "cojos" no están permitidos.

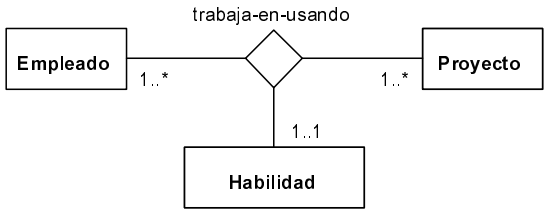

Figura 3.4. La asociación ternaria `trabaja-en-usando`, según la interpretación de *tuplas reales*

Así pues, en esta interpretación *la multiplicidad mínima siempre es al menos 1*, ya que el valor 0 no tiene sentido. Este "efecto del cero prohibido" no es consistente con la frecuente asignación de multiplicidad 0 en asociaciones ternarias (y, ante todo, con la documentación de UML, como en el ejemplo de la **Figura 3.2**, que sería incorrecto en esta interpretación). De hecho, el diagrama de la **Figura 3.3** debe ser completado y sustituido por el de la **Figura 3.4**.

Segunda interpretación (tuplas potenciales). "Cada par `Empleado-Proyecto`" puede entenderse como un "par meramente posible", o par potencial (*potential pair*), es decir, un par de instancias que pertenece al producto cartesiano de `Empleado` y `Proyecto`. Hay tres empleados y tres proyectos, de modo que hay nueve pares potenciales. Para algunos de estos pares, como `Benito-Garaje` o `Clara-Cocina`, hay una habilidad relacionada; para otros, como `Alberto-Garaje` o

`Clara-Laboratorio`, no hay ninguna. Así pues, la multiplicidad mínima 0 es consistente con la **Tabla 3.1** y la **Tabla 3.2**, y el diagrama de la **Figura 3.3** debe transformarse en el de la **Figura 3.5**.

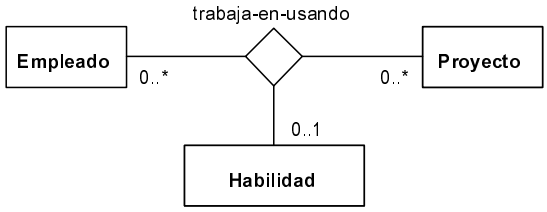

Figura 3.5. La asociación ternaria `trabaja-en-usando`, según la interpretación de *tuplas potenciales*

Bien, pero, ¿cuál sería el significado de la multiplicidad mínima 1? Supongamos que la multiplicidad de la clase `Habilidad` es 1..1, como de hecho ocurre en la **Figura 3.4**. Esto significaría que, para cada par potencial `Empleado-Proyecto`, podría haber una `Habilidad`, *pero no cero*; es decir, estarían prohibidos los pares potenciales no enlazados a ninguna habilidad. En otras palabras, cualquier par potencial debe estar enlazado a una habilidad al menos, y por tanto cualquier par potencial `Empleado-Proyecto` debe estar contenido al menos en un trío que exista de hecho en la asociación: todo empleado *debe* estar enlazado a todo proyecto al menos una vez. Esta regla y el diagrama precedente de la **Figura 3.4** no serían consistentes con los valores de la **Tabla 3.1** y la **Tabla 3.2**, ya que necesitaríamos que el entero producto cartesiano `Empleado-Proyecto` estuviera presente en la **Tabla 3.2** (de hecho, tendríamos que tener *exactamente* nueve líneas y no más, debido a la multiplicidad máxima 1 de `Habilidad`).

Así pues, en esta interpretación, *la multiplicidad mínima 1 asignada a una clase fuerza la existencia, dentro de algún trío real, de todos los pares potenciales de instancias de las clases restantes*. Esto sería un "efecto rebote del uno" que probablemente es inesperado por la mayoría de los modeladores. Sin embargo, esta interpretación parece válida, ya que está implícitamente de acuerdo con la documentación de UML y con algunos trabajos de formalización de multiplicidades [McAllister 95].

Tercera interpretación (enlaces cojos). Podríamos intentar una especie completamente distinta de interpretación, en la que se permita la existencia de *enlaces cojos* (*limping links*), es decir, enlaces ternarios que enlazan sólo dos instancias y dejan un vacío para la tercera. Entonces podríamos entender la multiplicidad 0..1 para la clase `Habilidad` como "cada par real `Empleado-Proyecto` puede estar enlazado con una o ninguna instancias de `Habilidad`", es decir, cada enlace sería, de hecho, o bien un par `Empleado-Proyecto` (perteneciente a una asociación binaria camuflada) o un trío `Empleado-Proyecto-Habilidad`

(perteneciente a la verdadera asociación ternaria). Por el contrario, si la multiplicidad fuera 1..1, entonces los enlaces cojos no estarían permitidos en el lado de la clase `Habilidad`: toda combinación real de empleado y proyecto debería estar enlazada a una habilidad. En esta interpretación, el diagrama de la **Figura 3.3** podría ser sustituido por el de la **Figura 3.6**, en el que los enlaces cojos han sido extraídos de la asociación ternaria, y se representan explícitamente mediante una asociación binaria superpuesta (esta extracción se podría ampliar a las otras dos multiplicidades cero).

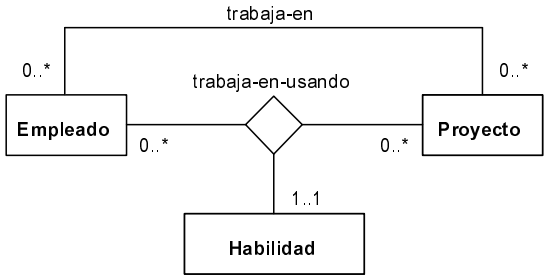

Figura 3.6. La asociación ternaria `trabaja-en-usando`, según la interpretación de *enlaces cojos*, en la que los enlaces cojos por parte de `Habilidad` se muestran como una asociación binaria explícita `trabaja-en`

En esta interpretación, *el símbolo de asociación ternaria se usa como una forma abreviada de representar una asociación ternaria genuina junto con una asociación binaria camuflada*. Estas asociaciones binarias pueden estar o no relacionadas con la asociación ternaria principal, de tal modo que algunos autores las denominan Relaciones Binarias Semánticamente Restrictivas (*Semantically Constraining Binary Relationships*) y Relaciones Binarias Semánticamente Independientes (*Semantically Unrelated Binary Relationships*) [Jones 96, Song 93]. Una asociación binaria *independiente* `Empleado-Proyecto` podría ser, por ejemplo, la asociación `es-pagado-por`, con el significado de asignación presupuestaria de empleados a proyectos; "independiente" significa aquí que el conjunto de pares es completamente independiente del conjunto de tríos. Podríamos considerar como asociación binaria *relacionada*, por otra parte, la asociación `trabaja-en` sin considerar las habilidades usadas. Pero la relación semántica entre esta asociación binaria y la asociación ternaria `trabaja-en-usando` podría tener diferentes sentidos, por ejemplo[23]:

- *Asociaciones relacionadas restrictivas.* Los pares de `trabaja-en` representan una precondición para la existencia de tríos en `trabaja-en-usando`, algo así como "para declarar que un empleado trabaja en un proyecto usando una habilidad, debe declararse primero que el mismo empleado trabaja en el mismo proyecto sin considerar la habilidad usada".

[23] Esta lista no pretende ser exhaustiva.

- *Restricciones de co-ocurrencia.* La asociación `trabaja-en` representa los pares obtenidos de los tríos al eliminar la habilidad, a los que añade alguna restricción de multiplicidad, tal como "un empleado puede trabajar, sin considerar la habilidad usada, como máximo en cuatro proyectos"; este tipo de restricción de co-ocurrencia se estudia con más detenimiento en el Apartado 3.3.2.
- *Asociaciones incompletas.* La asociación `trabaja-en` representa un conjunto de pares para los cuales el tercer elemento aún es desconocido, "este empleado trabaja en este proyecto usando cierta habilidad que aún no ha sido registrada"; representa cierta falta de información que se espera conocer tarde o temprano.

La asociación binaria en el primer sentido impone ciertamente una restricción sobre la asociación ternaria. De hecho se trata de dos asociaciones diferentes, la primera de ellas independiente y restrictiva sobre la segunda, y pensamos que es mejor, por claridad, representarlas separadamente. La asociación binaria en el segundo sentido, por el contrario, no se corresponde con ninguna asociación independiente, sino con una restricción de co-ocurrencia binaria sobre la misma asociación ternaria. Si la representamos como una asociación separada, aún necesitamos una restricción que relacione ambas asociaciones, de modo que poco ganamos con el cambio; así pues, preferimos mantenerlas juntas. Algunos autores proponen una notación de "dependencia existencial" para estos dos casos de subasociaciones binarias restrictivas (la ternaria depende de la binaria en el primer caso, o la binaria depende de la ternaria en el segundo caso) [McCarthy 97]. Finalmente, la asociación binaria en el tercer caso no es propiamente una asociación restrictiva, aunque existe una restricción que se aplica a la binaria y a la ternaria simultáneamente: si un par está en la asociación binaria explícita, no está en la asociación ternaria "expurgada", y viceversa.

En general, el procedimiento de fusionar una asociación binaria con una ternaria mediante el uso de enlaces cojos es desaconsejable. Obviamente, las *asociaciones binarias independientes* (tales como `es-pagado-por`) jamás deberían ser fusionadas con la asociación ternaria. Las *asociaciones relacionadas restrictivas* (primer sentido de *asociaciones binarias relacionadas*) también deben ser mantenidas aparte. Las *restricciones de co-ocurrencia* (segundo sentido) no son en realidad asociaciones diferentes de la ternaria a la que restringen, de modo que tampoco deberían ser representadas ni como asociaciones explícitas ni como asociaciones camufladas usando enlaces cojos. Las *asociaciones incompletas* (tercer sentido) son un caso más dudoso: parecen intuitivas, sencillas de entender y fáciles de usar; probablemente muchos modeladores usan el símbolo ternario como si significara esto.

El uso de enlaces cojos para representar *asociaciones incompletas* evita los extraños efectos vistos más arriba: el "efecto del cero prohibido"

para la interpretación de pares reales, y el "efecto rebote del uno" para la interpretación de pares potenciales. Además, evita la representación explícita de una asociación binaria auxiliar, y esto es bueno ya que ni siquiera está claro que una asociación incompleta sea conceptualmente una asociación distinta de la asociación ternaria de la que procede. Por otra parte, UML no permite las asociaciones incompletas, ya que establece que cada instancia de una asociación n-aria es una n-tupla de valores de las respectivas clases (recuérdese la definición [UML, p. 3-79]). La interpretación de "enlaces cojos" para asociaciones incompletas es una variación de la interpretación de "pares reales", en la que la multiplicidad mínima 0 significa que se permite la falta de información. No obstante, aún quedan algunas dificultades: ¿Cuántas patas pueden faltar en un enlace n-ario? ¿Una, dos, hasta n-2? ¿Cómo debe interpretarse la restricción de multiplicidad máxima cuando falta una pata en uno de los extremos opuestos?[24]

Así pues, esta interpretación puede parecer sencilla y útil a primera vista, pero aún quedan algunos puntos que no están en absoluto claros, en primer lugar la definición misma de asociación n-aria en UML. En consecuencia, en el resto de este Capítulo adoptamos la interpretación de pares potenciales por claridad, aunque los temas tratados son hasta cierto punto independientes de esta elección.

En la **Tabla 3.3** resumimos las diferentes interpretaciones y paradojas de las multiplicidades n-arias.

Nombre	**Interpretación**	**Paradoja**
Tuplas reales	Número permitido de valores en un extremo con respecto a una *combinación real* de valores en los otros n-1 extremos	"Efecto del cero prohibido": la multiplicidad mínima siempre es al menos 1
Tuplas potenciales	Número permitido de valores en un extremo con respecto a una *combinación potencial* de valores en los otros n-1 extremos	"Efecto rebote del uno": la multiplicidad mínima 1 asignada a un extremo fuerza la existencia de todas las combinaciones potenciales de valores de los otros n-1 extremos dentro de al menos una tupla
Enlaces cojos	Número permitido de valores en un extremo (*incluyendo el nulo*) con respecto a una combinación real de valores en los otros n-1 extremos	"Tuplas incompletas": las instancias de la asociación podrían conectar menos de n valores

Tabla 3.3. Tres interpretaciones diferentes de las multiplicidades n-arias

[24] Nótese la sutil diferencia entre las dos preguntas siguientes: ¿cuántos empleados pueden trabajar en un proyecto sin considerar la habilidad empleada? Y, ¿cuántos empleados pueden trabajar en un proyecto cuando la habilidad empleada aún se desconoce? La primera pregunta se refiere a una restricción de co-ocurrencia binaria sobre la asociación ternaria, mientras que la segunda pregunta se refiere a la multiplicidad estándar cuando el valor de uno de los extremos opuestos se fija como "desconocido" o "vacío".

3.3.2. La necesidad de expresar la restricción de participación

Hemos visto tres interpretaciones diferentes que podrían resolver la ambigüedad de la definición de multiplicidad mínima de las asociaciones n-arias en UML. La primera, *pares reales*, implica que la multiplicidad mínima debe ser siempre al menos 1, lo cual no es consistente con la documentación y la práctica; la segunda, *pares potenciales*, parece correcta pero produce un extraño efecto cuando el valor es 1; la tercera, *enlaces cojos*, contradice la definición de asociación n-aria, aunque parece intuitiva cuando se pretende que signifique *asociaciones incompletas*. ¿Por qué es tan escurridiza la multiplicidad mínima en las asociaciones n-arias?

McAllister ofrece una buena formalización del concepto de cardinalidad (o multiplicidad, en la terminología de UML) [McAllister 95]. Siendo `a` y `b` dos conjuntos de roles no nulos y no superpuestos en una asociación n-aria `A`, la multiplicidad `M(a, b)=(min, max)` especifica que una subtupla formada con instancias de los roles en `a` debe estar asociada mediante `A` con entre `min` y `max` subtuplas formadas con instancias de los roles en `b`[25]. Nótese que esto corresponde a la *restricción de co-ocurrencia* definida más arriba. En el ejemplo de la **Figura 3.3**, si `a = {Empleado, Habilidad}` y `b = {Proyecto}`, entonces `M(a, b) = (0..*)`. McAllister demuestra que el número total de valores de multiplicidad que se pueden definir en una asociación con N roles viene dado por $3^N-2^{N+1}+1$, y aplica este cálculo para N desde 2 hasta 5, obteniendo los resultados de la **Tabla 3.4**.

N = número de roles en A	número de M(a, b) para A
2	2
3	12
4	50
5	180

Tabla 3.4. Número total de valores de multiplicidad que se pueden definir en una asociación con N roles

A medida que N crece, hay un rápido incremento en el número de valores de multiplicidad que deberían ser analizados si se desea entender completamente la naturaleza de la asociación. Éste puede ser un factor que explique por qué muchos profesionales del modelado de datos encuentran dificultades al tratar con asociaciones n-arias, especialmente si sólo se considera un pequeño número de las multiplicidades aplicables a cada una de estas asociaciones.

Para N = 3, que es el caso de una asociación ternaria como `trabaja-en-usando`, los doce valores que pueden considerarse son: los tres valores según el estilo Chen/UML, los tres valores según el estilo

[25] Por simplicidad, restringimos el análisis a la forma más simple de multiplicidad, un único intervalo entero, aunque esto no afecta al razonamiento.

Merise, y seis valores para las tres asociaciones binarias incrustadas (es decir, implícitas) `Empleado-Proyecto`, `Empleado-Habilidad` y `Proyecto-Habilidad` (recuérdese que no son asociaciones verdaderamente independientes). McAllister prosigue definiendo un conjunto de reglas de consistencia, ya que estos valores no son completamente independientes [McAllister 95]. Para entender plenamente la estructura de una asociación n-aria, el modelador debería especificar todas estas *restricciones de co-ocurrencia*, pero la mayoría de las veces basta con los N valores de Chen y los N valores de Merise (siendo los otros habitualmente muchos-a-muchos [Jones 96]). Cuando estos valores se especifican como un simple intervalo `(min, max)`, la consistencia entre los valores de Chen y de Merise se determina asegurando que *cada valor* `min` *o* `max` *de Chen es menor o igual que los valores* `min` *o* `max` *de Merise de las otras clases* [McAllister 95]. Esta regla puede confrontarse con el ejemplo de la **Figura 3.7**, que muestra ambos conjuntos de valores: `minChen` para `Habilidad` es 0, que es igual a `minMerise` para `Empleado` (0) y menor que `minMerise` para `Proyecto` (1); `maxChen` para `Habilidad` es 1, que es menor que `maxMerise` para `Empleado` (*) y `maxMerise` para `Proyecto` (*)[26].

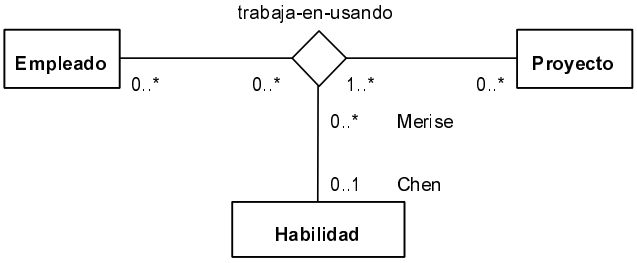

Figura 3.7. La asociación ternaria `trabaja-en-usando`, en la que se muestran los valores de multiplicidad de Chen y de Merise

Ahora podemos entender mejor los problemas semánticos de la multiplicidad mínima que se han considerado más arriba. La multiplicidad mínima está asociada con la *restricción de participación*, pero en el caso de una asociación ternaria, en el estilo de Chen, no significa la participación de instancias de la clase en la asociación, sino la participación de un par de instancias de las otras dos clases. El valor 0 para `Habilidad` no significa una participación opcional de `Habilidad` en la asociación, sino la

[26] Si, además de los valores de Chen y de Merise, los otros valores también son importantes, la representación tabular de McAllister para la multiplicidad de una asociación es una buena elección; estas restricciones de multiplicidad que son difíciles de expresar en los modelos Entidad/Relación tradicionales también pueden expresarse de modo natural utilizando aserciones [Kilov 94] (las "aserciones" son especificaciones declarativas de lo que debe ser verdadero en el "estado estable" del sistema, es decir, fuera de las operaciones atómicas; también se denominan "invariantes").

participación opcional de los pares de instancias de `Empleado-Proyecto` en la asociación con instancias de `Habilidad`. Si esto va contra la intuición, razón de más para clarificarlo. De hecho, la participación de cada clase individual queda sin expresar en el estilo de Chen, mientras que el estilo de Merise la representa adecuadamente. Por otra parte, las *dependencias funcionales* quedan sin expresar en el estilo de Merise, mientras que la multiplicidad máxima 1 las representa en el estilo de Chen. Ésta es probablemente la razón por la que OMT y UML han elegido Chen en lugar de Merise, aunque la dependencia funcional no es inherentemente más importante que la participación.

Ambos estilos, Chen y Merise, son correctos y pueden describir la misma asociación, pero establecen hechos distintos acerca de la naturaleza de la asociación. Los hechos representados por cada estilo no son especificados al usar el otro, ni pueden ser derivados del otro (excepto en el caso de asociaciones binarias, donde simplemente intercambian sus posiciones). Así pues, si los dos estilos proporcionan información útil para entender la asociación, *¿por qué no representar ambos en el mismo diagrama?* La **Figura 3.7** repite el ejemplo de la **Figura 3.5**, pero añadiendo los valores de Merise junto al rombo de la asociación. Estos valores son consistentes con los valores de la **Tabla 3.1** y la **Tabla 3.2**, y añaden aspectos semánticos nuevos y útiles a la asociación: notemos especialmente que la clase `Proyecto` es la única con participación obligatoria (`minMerise` para `Proyecto` es 1), es decir, un proyecto no puede existir sin estar enlazado a un par empleado-habilidad, aunque puede haber muchos pares (potenciales) empleado-habilidad no enlazados a ningún proyecto (`minChen` para `Proyecto` es 0); notemos también que la clase `Habilidad` puede participar en muchas instancias de la asociación (`maxMerise` para `Habilidad` es *), es decir, una cierta habilidad puede estar enlazada a muchos pares diferentes empleado-proyecto, aunque para cada par como máximo puede usarse una habilidad (`maxChen` para `Habilidad` es 1). La representación de ambos tipos de valores de multiplicidad en el mismo diagrama no es una idea enteramente nueva: otras propuestas tales como la de CDIF (*CASE Data Interchange Format*) les han dado los razonables nombres de *cardinalidad exterior* para Chen (*outer multiplicity*) y *cardinalidad interior* para Merise (*inner multiplicity*) [CDIF 96][27]. Esta notación puede parecer similar a la de reemplazar la asociación ternaria por una nueva entidad y tres asociaciones binarias que simulan la asociación ternaria, como se muestra en la **Figura 3.8**.

[27] De hecho, la idea fue presentada durante la elaboración de la versión 1 de UML, pero fue rechazada sin la debida consideración [Miller 01].

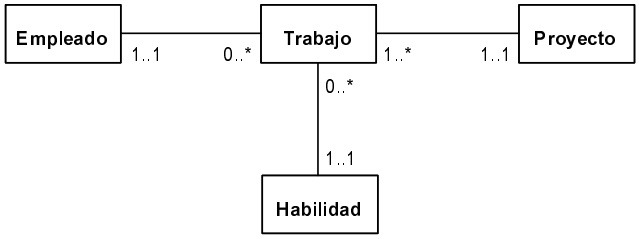

Figura 3.8. La asociación ternaria `trabaja-en-usando` sustituida por la entidad asociativa `Trabajo`. Sólo los valores de multiplicidad de Merise son preservados en la transformación

Esta nueva entidad es generalmente denominada *entidad de intersección*, *entidad asociativa* o *gerundio* [Song 95]. Notemos que los valores de multiplicidad de Merise son preservados en esta transformación, y situados de nuevo junto a la entidad asociativa, pero todos los valores de Chen han sido sustituidos por 1..1, ya que cada instancia de `Trabajo` está enlazada con una y sólo una instancia de las otras clases (esto es lo mismo que decir que todo enlace ternario tiene "tres patas"). En otras palabras, la semántica de las dependencias funcionales expresada por la asociación ternaria se pierde al simularla con un gerundio, pero la semántica de la participación sí se preserva. Hay otras diferencias entre asociaciones binarias y ternarias y, en general, las representaciones binarias de asociaciones ternarias no preservan las dependencias funcionales [Jones 95, Jones 00, Song 93].

Volviendo ahora a la **Figura 3.7**, la multiplicidad mínima de Chen, mostrada junto a la clase, será normalmente *cero*, para evitar el "efecto rebote del uno". Por otra parte, la multiplicidad máxima de Merise será normalmente *muchos*, como vamos a demostrar a continuación. Supongamos que la multiplicidad máxima de Merise fuera *uno* en lugar de *muchos*, como en la clase `Hijo` en la asociación denominada `engendra` en la **Figura 3.9**[28]. Esto significa que una instancia de `Hijo` está enlazada a un y sólo un par de instancias de `Padre` y `Madre`, que en términos de dependencias funcionales puede expresarse como `[Hijo → (Padre, Madre)]`, y por tanto `[Hijo → Padre]` e `[Hijo → Madre]`, aplicando la regla de descomposición [Teorey 99]. Esto significa que una asociación ternaria que tenga multiplicidad máxima de Merise *uno* en un lado siempre puede descomponerse en dos asociaciones binarias sin pérdida de semántica de co-ocurrencia (esto es, semántica de participación y de dependencia funcional): en otras palabras, la multiplicidad máxima de

[28] En este ejemplo usamos por simplicidad tres nombres de clases para denotar lo que más bien son tres diferentes roles de la misma clase `Persona`.

Merise normalmente será *muchos*, como queríamos demostrar. La descomposición se muestra en la **Figura 3.10**[29].

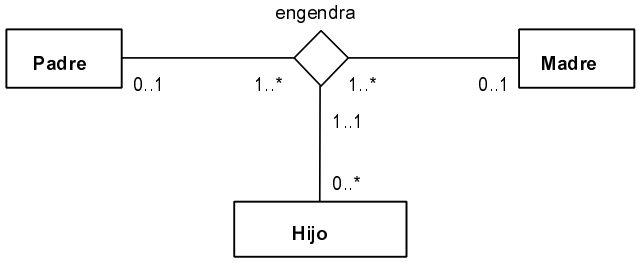

Figura 3.9. La asociación ternaria `engendra`, con las multiplicidades de Chen y Merise

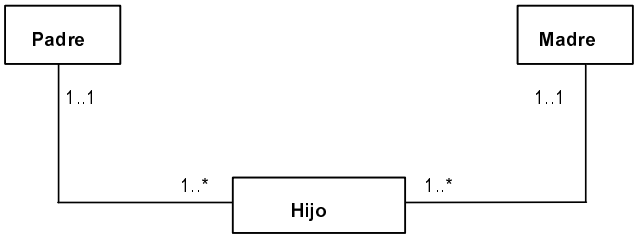

Figura 3.10. La asociación ternaria `engendra` descompuesta en dos asociaciones binarias sin pérdida de semántica de co-ocurrencia

Así pues, en una asociación ternaria los valores de multiplicidad más significativos son la *multiplicidad máxima de Chen*, que expresa la dependencia funcional de una clase respecto a las otras dos clases, y la *multiplicidad mínima de Merise*, que expresa la participación de la clase en la asociación. Podríamos mostrar sólo estos dos valores, y de hecho esto es lo que hacen algunos métodos [Dullea 98, Elmasri 94] (ver **Figura 3.11**). En nuestra opinión, sin embargo, esta notación podría resultar engañosa en el estilo de UML, de modo que preferimos mantener la expresión completa de ambas multiplicidades en nuestra propuesta.

[29] Esta descomposición plantea otro interesante problema derivado. ¿Es la co-ocurrencia entre instancias el único aspecto semántico importante de una asociación? La ternariedad en la asociación "engendra" expresa también que un padre y una madre engendran conjuntamente un hijo; que un hijo no tiene un padre sin una madre; etc. Desde el punto de vista humano, la relación de procreación es inherentemente ternaria, de modo que su proyección en dos asociaciones binarias tendrá algún inconveniente en el modelado conceptual, por muy correcta que sea desde el punto de vista formal de las co-ocurrencias. En otras palabras: el modelado orientado a objetos no es sólo modelado de datos. El modelado del comportamiento es igualmente importante que el modelado estructural, y por tanto las restricciones de co-ocurrencia, o multiplicidades, no son el criterio último para entender una asociación orientada a objetos, aunque sean importantes.

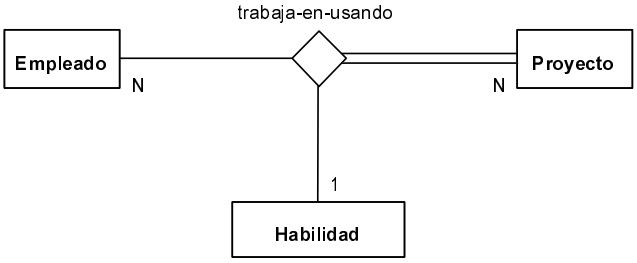

Figura 3.11. Dependencia funcional y participación expresadas conjuntamente en la notación de Elmasri y Navathe. El trazo doble indica participación obligatoria

Nótese la semántica transmitida por las multiplicidades en el caso de la asociación "engendra" en la **Figura 3.9**. En primer lugar, la participación es obligatoria para las tres clases. Esto es evidente para `Hijo`, pero en el caso de `Madre` (e igualmente para `Padre`) significa que ser madre es "haber engendrado ya a alguien", no simplemente "estar en el proceso de engendrar a alguien" o "ser capaz de engendrar a alguien". No toda persona femenina puede ser una instancia de `Madre` en este modelo (otro modelo podría usar el nombre de clase `Madre` para significar madres potenciales en lugar de madres de hecho, pero éste no es el caso). Una madre puede haber engendrado en general más de una vez (`maxMerise` para `Madre` es *), y con el mismo padre puede haber engendrado muchos hijos (`maxChen` para `Hijo` es *). También se declara que, con un padre concreto, una madre puede no haber engendrado ningún hijo (`minChen` para `Hijo` es 0), y esto no contradice el hecho de que ella debe haber engendrado al menos un hijo con algún padre (`minMerise` para `Madre` es 1). Recuérdese que en este Apartado estamos siguiendo la interpretación de los pares potenciales: si `Juan` y `Susana` no han engendrado conjuntamente ningún hijo, entonces simplemente no están enlazados por esta asociación.

Si ignoramos las multiplicidades de Merise, o peor, si asumimos un valor por defecto 0..*, la semántica de la asociación es completamente diferente (y la equivalencia entre la **Figura 3.9** y la **Figura 3.10** se pierde). Por supuesto, la participación ya no es obligatoria para ninguna de las tres clases; pero, y es lo más extraño, un hijo podría tener varios padres: ¡un padre por cada madre que tenga! (y, a la inversa, "una madre por cada padre que tenga"). Éste es un caso en el que mostrar más multiplicidades tiene una gran importancia para especificar adecuadamente una asociación. Las multiplicidades de Merise se pueden expresar en todo caso en un diagrama usando el mecanismo general existente en UML para mostrar las *restricciones*, pero pensamos que son suficientemente significativas como para merecer un lugar en la notación al mismo nivel que las multiplicidades de Chen[30].

[30] Un modelador podría también extender el lenguaje y crear un estereotipo para asociaciones que añadiera a la especificación UML del elemento de modelo

3.3.3. Propuesta sobre asociaciones n-arias para el Estándar de UML

En este Apartado proponemos brevemente los cambios requeridos para mejorar la semántica y notación de las multiplicidades n-arias. Siguiendo el enfoque de CDIF [CDIF 96], adoptamos aquí los términos multiplicidad externa/interna en lugar de Chen/Merise, ya que parecen más apropiados para una definición técnica. En nuestra propuesta adoptamos también la interpretación de "enlaces cojos" por razones pragmáticas, teniendo cuidado de especificar que representa asociaciones incompletas, pero no subasociaciones relacionadas restrictivas.

Semántica

- Una *asociación n-aria* es una asociación entre tres o más clases (más exactamente, clasificadores).
- Un *enlace n-ario* es una n-tupla de valores de las respectivas clases. Uno o más roles en el enlace (hasta n-2) pueden estar vacíos, significando una instancia incompleta de la asociación. El valor "vacío" se considera un valor concreto al aplicar las restricciones de multiplicidad.
- La *multiplicidad externa* en un extremo de asociación n-aria expresa el concepto de dependencia funcional, es decir, especifica el número potencial de valores en el extremo especificado, con respecto a una combinación de valores en los otros n-1 extremos. Una multiplicidad externa mínima 0 significa que el extremo de enlace especificado puede estar vacío; si el valor es 1 significa que el extremo de enlace especificado es funcionalmente dependiente de los otros n-1 extremos.
- La *multiplicidad interna* en un extremo de asociación n-aria expresa el concepto de participación opcional u obligatoria, es decir, especifica el número potencial de combinaciones de valores en los otros n-1 extremos con respecto a un valor en el extremo especificado. Una multiplicidad interna mínima 0 significa que los elementos en el extremo pueden no tomar parte en la asociación; si el valor es 1 o mayor significa que los elementos del extremo deben tomar parte obligatoriamente en la asociación.
- Para asociaciones binarias, la multiplicidad externa en un extremo debe ser igual a la multiplicidad interna en el otro extremo, y viceversa. Por tanto, las multiplicidades internas se omiten.

`Association` un conjunto de valores etiquetados (*tagged values*) que permitieran al modelador registrar un segundo conjunto de multiplicidades de la asociación (Joaquin Miller, mensaje a la lista de correo del *Precise UML Group* [pUML], 2 enero 2002). Pero, en la medida en que este nuevo estereotipo no es estándar, su uso no se generalizaría entre los usuarios de UML.

Notación

- Una *asociación n-aria* se representa como un rombo unido con una línea a cada clase participante.
- Un *enlace n-ario* se representa como un rombo unido con una línea a cada instancia participante. Pueden omitirse hasta n-2 líneas, pero el rombo debe mostrarse en todo caso para indicar el carácter n-ario del enlace.
- La *multiplicidad externa* en un extremo de asociación n-aria se representa junto a la clase en la línea que une el rombo con la clase.
- La *multiplicidad interna* en un extremo de asociación n-aria se representa junto al rombo en la línea que une el rombo con la clase.

Metamodelo

- La metaclase `AssociationEnd` tiene dos propiedades de multiplicidad, expresadas como dos meta-atributos diferentes: `outerMultiplicity` e `innerMultiplicity`. Si el extremo de asociación pertenece a una asociación binaria, el valor de `innerMultiplicity` no se especifica.

3.4. La multiplicidad de las asociaciones binarias especiales

3.4.1. La multiplicidad de la asociación cualificada

Una vez examinado el problema de la multiplicidad en el caso más "difícil" de las asociaciones n-arias, veamos ahora lo que ocurre en el caso, en principio más sencillo, de las asociaciones cualificadas (ver **Figura 3.12**). Este tipo de asociación fue introducida por Rumbaugh en su *Object-Relation Model* como un *tipo especial de asociación ternaria*, con la idea de disponer de algún tipo de asociación "indexada" que proporcionara un acceso más eficiente a las instancias enlazadas a través de la asociación [Rumbaugh 87]. Posteriormente fue incorporada en OMT y UML, aunque su uso aún no está muy difundido, tal vez porque su lugar en UML no está demasiado claro [Stevens 02].

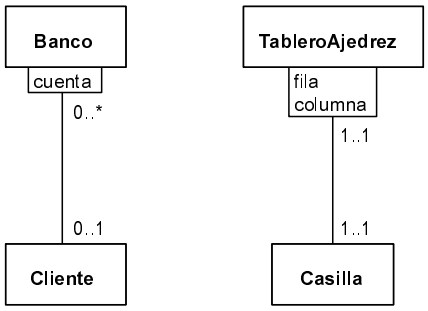

Figura 3.12. Ejemplos de asociaciones cualificadas

Recordemos la definición de multiplicidad de una asociación cualificada en el origen: "La multiplicidad del extremo destino denota las posibles cardinalidades del conjunto de instancias destino seleccionadas por el emparejamiento de una instancia origen y un valor de cualificador" [UML, p. 3-76][31]. Nótese que un cualificador puede estar formado por varios atributos, y que una instancia del cualificador comprende un valor para cada atributo cualificador [UML, p. 2-67]. Algunos valores típicos de multiplicidad cualificada son [UML, p. 3-76][32]:

- 0..1: se selecciona una única instancia destino, pero [para una instancia origen dada] un valor cualquiera del cualificador no selecciona necesariamente una instancia destino.
- 1..1: cualquier posible valor del cualificador [para una instancia origen dada] selecciona una única instancia destino; por tanto, el dominio de valores del cualificador debe ser finito.
- 0..*: el cualificador es un índice que realiza una partición de las instancias destino en subconjuntos.

La multiplicidad en el extremo origen, por el contrario, no tiene en cuenta el cualificador, sino que tiene el significado habitual de la multiplicidad en una asociación binaria sencilla. Por otra parte, *la multiplicidad "pura"* (sin tener en cuenta el cualificador) *en el extremo destino queda sin expresar*, y se asume por defecto que es 0..*. El Estándar argumenta que esto resulta casi siempre adecuado, aunque no sea completamente general, porque una multiplicidad máxima 1 se modelaría mejor sin cualificador [UML, p. 2-67]. Esto sin duda es correcto, pero queda el problema de expresar otras multiplicidades máximas distintas de * o mínimas distintas de 0.

Para entender mejor estas definiciones, podemos establecer una equivalencia parcial entre una asociación cualificada y una asociación n-aria con restricciones de co-ocurrencia, sustituyendo los atributos cualificadores por clases para obtener una representación más "familiar". La **Figura 3.13** muestra esta transformación aplicada a los ejemplos de la **Figura 3.12**.

---

[31] Una redacción algo más clara sería: "cuando el extremo origen de la asociación está cualificado, la multiplicidad en el extremo destino de la asociación denota cuántas instancias destino pueden estar asociadas con la combinación de una instancia origen y un valor de cualificador".

[32] Añadimos entre corchetes algunas palabras para aclarar la redacción del original en el Estándar, que sin ellas resulta confusa.

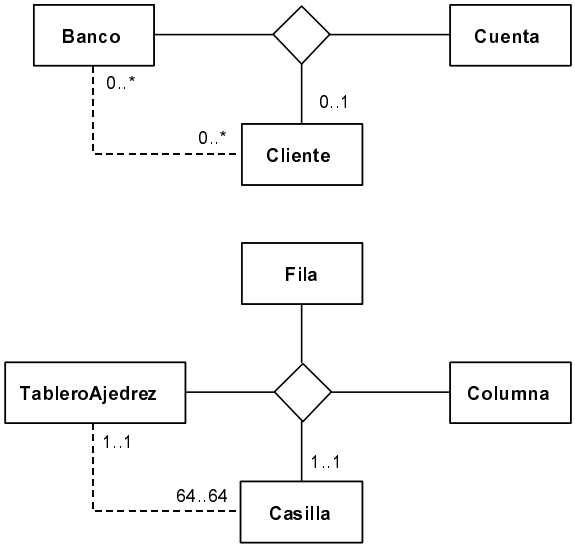

Figura 3.13. Equivalencia parcial entre las asociaciones cualificadas de la **Figura 3.12** y asociaciones n-arias con restricciones de co-ocurrencia binarias. Estas últimas están expresadas mediante líneas discontinuas para no confundirlas con asociaciones independientes (no es notación estándar de UML). La multiplicidad "pura" para la clase `Casilla` (64) no puede expresarse con la notación de asociación cualificada

Al contrario de lo que ocurre con las asociaciones n-arias, el Estándar sí explica el significado de la multiplicidad mínima en el extremo destino de una asociación cualificada: "Nótese que una multiplicidad cualificada cuyo límite inferior es cero indica que un determinado valor de cualificador puede estar ausente, mientras que un límite inferior de uno indica que cualquier posible valor de cualificador debe estar presente. Esto último sólo es razonable para cualificadores con un número finito de valores (tales como valores enumerados o rangos enteros) que representan una tabla indexada por un rango finito de valores" [UML, p. 2-67]. Esto significa, ni más ni menos, que el Estándar adopta la interpretación de *tuplas potenciales* (según la terminología que hemos empleado en el Apartado 3.3.1) para la multiplicidad en las asociaciones cualificadas. En efecto, si la multiplicidad mínima es 0, como en el caso `Banco-Cuenta-Cliente` de la **Figura 3.12** y la **Figura 3.13**, no es necesario que todas las combinaciones `Banco-Cuenta` estén instanciadas en la asociación, es decir, un banco no tiene por qué tener todas las posibles cuentas. Por el contrario, si la multiplicidad mínima es 1, como en el caso `TableroAjedrez-Fila-Columna-Casilla`, toda posible combinación de `TableroAjedrez-Fila-Columna` debe estar presente al menos en una instancia de la asociación ("efecto rebote del uno"), como ocurre de modo natural en este ejemplo del juego de ajedrez. Esto apoya la conclusión de que el Estándar asume implícitamente la interpretación de tuplas potenciales para las asociaciones n-arias, como hemos visto anteriormente.

No obstante, hemos considerado que, por razones prácticas y por ser más intuitiva, en la multiplicidad n-aria es más conveniente la interpretación de *enlaces cojos*, de modo que deberíamos intentar aplicarla también a la multiplicidad cualificada. Pero esto no resulta fácil, porque el concepto de "asociación incompleta" no tiene sentido en una asociación cualificada, ya que no hay ninguna asociación entre la clase origen y el cualificador. Recordemos que la interpretación de enlaces cojos es útil para representar asociaciones n-arias incompletas, es decir, asociaciones cuyas instancias pueden enlazar menos de n elementos, pudiendo estar vacíos hasta n-2 extremos. Si tuviera sentido en un dominio determinado la representación de una asociación incompleta entre la clase origen y el cualificador, lo propio sería emplear una asociación n-aria, no una asociación cualificada. Así pues, debemos renunciar a la interpretación de enlaces cojos para la multiplicidad cualificada.

3.4.2. La multiplicidad de la clase-asociación

Recordemos dos problemas que hemos encontrado al tratar con la multiplicidad de una clase-asociación binaria (ver Apartado 3.2.1). En primer lugar, la restricción de que *no puede haber tuplas repetidas* en una asociación [UML, pp. 2-19 y 2-110] dificulta la representación de algunas situaciones. En la **Figura 3.14** la asociación `ha-reservado` *no* puede tener dos instancias distintas que enlacen los mismos dos objetos `Juan` y `Mesa17` en dos fechas diferentes, ya que la *identidad del enlace* no viene determinada por sus atributos, si los tiene, sino por las identidades de los objetos que enlaza.

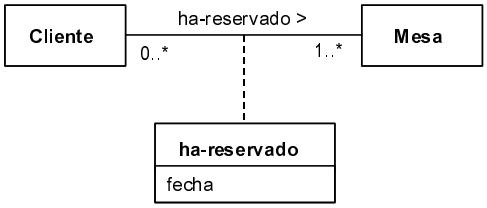

Figura 3.14. Una clase-asociación no puede contener tuplas repetidas: `ha-reservado` no puede utilizarse para representar que el mismo `Cliente` ha reservado la misma `Mesa` en dos fechas diferentes

En segundo lugar, esta misma restricción dificulta también la representación de asociaciones con *lógica temporal*, es decir, predicados que se consideran válidos en un periodo de tiempo determinado. En la **Figura 3.15** la asociación `trabaja-para` necesita una restricción adicional que impida la coexistencia de dos enlaces con periodos de tiempo superpuestos referidos a la misma persona; aún así, *no* podemos tener una persona que trabaje para la misma empresa en dos periodos de tiempo distintos.

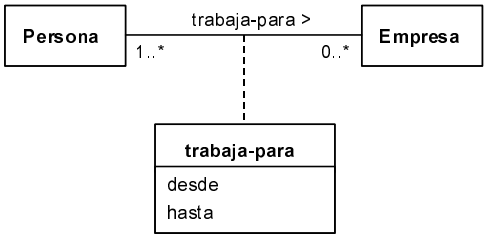

Figura 3.15. Una clase-asociación no puede representar lógica temporal: `trabaja-para` no puede utilizarse para representar que una `Persona` trabaja para más de una `Empresa` a lo largo del tiempo, pero para una sola simultáneamente

Existe una cierta analogía entre una asociación cualificada y una clase-asociación que tenga atributos pero no operaciones, también llamada asociación atribuida (*attributed association*). En ambos casos se trata de reflejar, en forma de atributos, información que es propia de la asociación, no de las clases participantes. La diferencia estriba en que la multiplicidad cualificada se ve afectada por los valores de los atributos del cualificador, que actúan como un "índice" de la asociación. Esta analogía puede servir como base para intentar resolver el primer problema expuesto. La **Figura 3.16** es una variación de la **Figura 3.14**, en donde la clase-asociación ha sido sustituida por una asociación cualificada.

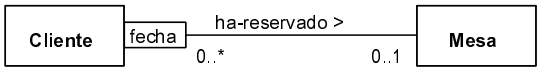

Figura 3.16. Uso de una asociación cualificada para resolver aparentemente la necesidad de tuplas repetidas

De acuerdo con la definición de multiplicidad cualificada, por cada posible pareja de valores de `Cliente[fecha]` puede haber cero o un valores de `Mesa`. Por tanto, para dos combinaciones distintas en el extremo origen aparentemente puede repetirse el mismo valor en el extremo destino. No obstante, *esta conclusión es dudosa*, ya que no deja de tratarse de una asociación binaria, y por tanto las tuplas están formadas por instancias de `Cliente-Mesa`, que no deben repetirse. La definición de asociación cualificada en UML, a medio camino entre la asociación binaria y la asociación n-aria, no deja suficientemente claro este punto: si prevalece la analogía binaria, no se pueden repetir las tuplas; si prevalece el aspecto n-ario, sí se pueden repetir.

El intento de resolver el segundo problema mediante una asociación cualificada fracasa de forma más clara. La **Figura 3.17** es una variación de la **Figura 3.15**, en donde nuevamente una asociación cualificada sustituye a la clase-asociación.

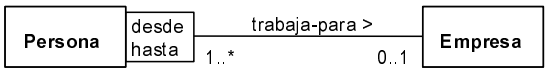

Figura 3.17. Uso de una asociación cualificada para resolver aparentemente la necesidad de lógica temporal

Con la asociación cualificada, tomando el periodo de tiempo (definido por `desde`, `hasta`) como cualificador, se puede asignar la multiplicidad 0..1 en el extremo destino y así la instancia de `Persona` puede estar enlazada con varias instancias de `Empresa` en periodos distintos, pero no en el mismo periodo. De todas formas, como el periodo está definido mediante dos atributos (es decir, a diferencia de la fecha, el periodo es un cualificador bidimensional), esto no impide que una persona esté enlazada con dos empresas en *dos periodos distintos pero superpuestos*, de modo que sigue haciendo falta una restricción adicional para evitar el solapamiento temporal. Así pues, en este caso la asociación cualificada no ofrece ventajas sobre la clase-asociación[33].

3.5. Las instancias de la asociación: ¿tuplas no repetidas?

Como hemos visto, UML define una asociación como un conjunto de tuplas no repetidas: "Las instancias de una asociación son un conjunto de tuplas que relacionan instancias de los clasificadores. Cada valor de tupla puede aparecer como máximo una vez. (...) Una instancia de una asociación es un enlace, que es una tupla de instancias tomadas de los correspondientes clasificadores" [UML, pp. 2-19 y 2-20]. En esta concepción, en la que *el enlace es meramente una tupla*, y además se impone la restricción de que *no puede haber tuplas repetidas*, puede observarse una fuerte influencia del concepto de "relación" (*relation*) tal como es entendido en las metodologías de diseño de bases de datos, en las que es vital evitar la repetición de información redundante. De hecho, Rumbaugh llama todavía "relaciones" a las asociaciones en su *Object-Relation Model* [Rumbaugh 87]. Pero en UML se ha adoptado esta restricción sin considerar todas las consecuencias.

Son varios los autores que opinan que ésta es una restricción excesiva en UML, cuyo alcance debe ser más amplio que el modelado de datos, y seguramente sería mejor dejar libertad al modelador para imponerla o no

[33] Si el problema del registro histórico se presenta con gran frecuencia en un determinado dominio, una posible solución es definir el estereotipo «`history`», que aplicado a una asociación define dos valores distintos de multiplicidad en cada extremo, multiplicidad instantánea y multiplicidad a lo largo del tiempo, y dos atributos en la asociación que representan el periodo de validez de cada enlace a efectos de multiplicidad instantánea. Fowler lo llama "patrón de correspondencia histórica" (*Historic Mapping Pattern*) [Fowler 97, Fowler 00]. Aun así, sigue presente el problema de que no puede haber tuplas repetidas, que es una restricción que no puede ser soslayada por un estereotipo, ya que *un estereotipo puede extender el lenguaje restringiendo el significado de un elemento ya existente, pero no ampliándolo*.

según los casos; en la situación actual no es posible prescindir de ella por tratarse de una restricción predefinida; en cambio, si se eliminase la restricción del lenguaje básico, sería bien fácil recuperarla en contextos determinados donde fuera necesaria [Genilloud 99, Stevens 02]. Esta restricción pone frecuentemente al analista en situaciones incómodas como las que hemos visto en el Apartado 3.4.2, y conduce a modelos innecesariamente complejos. Otros analistas no advierten las implicaciones de la restricción, especialmente en el caso de clases-asociación y asociaciones cualificadas, lo que les lleva a construir modelos erróneos (a menudo el uso de una clase-asociación está motivado por el intento de saltarse la restricción). Es más, en algunos entornos como CORBA es imposible garantizar el cumplimiento de la restricción, ya que no es posible comparar la igualdad de dos referencias a objetos, debido a la existencia de sinónimos [Genilloud 99].

Veamos más en detalle los dos aspectos del problema.

3.5.1. La asociación, ¿conjunto o saco de tuplas?

En un conjunto (*set*), por su propia definición, no puede haber elementos repetidos. Alternativamente, se suele manejar el concepto de saco (*bag*), que sí admite elementos repetidos[34]. Ya hemos visto algunos problemas causados por la definición de asociación como conjunto de tuplas (*set of tuples*) con la restricción de que no puede haber tuplas repetidas, que denominaremos abreviadamente "restricción TNR" (tuplas-no-repetidas). Veamos un ejemplo más [Stevens 02]. En la **Figura 3.18** (a) tenemos el problema del mundo real "grafos con nodos y arcos no dirigidos entre los nodos". El diagrama de clases en (b) aparentemente representa de modo adecuado la especificación de este tipo de grafos: la mayoría de la gente entendería intuitivamente que un arco se dibuja entre dos nodos, y un nodo puede ser la raíz de varios arcos. Las figuras (c) y (d) representan dos posibles diagramas de objetos correspondientes al grafo concreto mostrado en (a).

[34] El concepto de *bag* no existe en UML, aunque sí en OCL [UML, p. 6-40]. El lenguaje OCL no es parte de UML, aunque sí está estrechamente ligado a UML, y es usado en el Estándar para expresar las reglas para la correcta formación de modelos (*well-formedness rules*) [UML, p. 6-2].

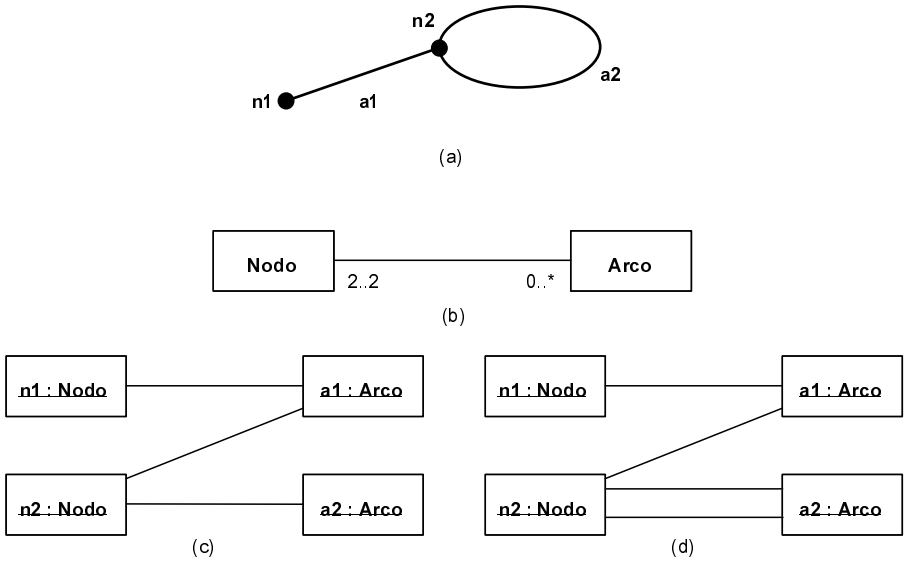

Figura 3.18. La restricción de que no puede haber tuplas repetidas en una asociación hace imposible la representación sencilla de algunos problemas: a) grafo con nodos y arcos no dirigidos; b) diagrama de clases intuitivo; c) y d) dos diagramas de objetos incorrectos

El problema es que ambos diagramas de objetos son incorrectos: en (c) no se respeta la multiplicidad mínima 2 de `Nodo` respecto a `Arco`, porque hay un solo objeto `n2` conectado al objeto `a2`; en (d) no se respeta la restricción TNR, porque hay dos enlaces `a2-n2`, es decir, dos tuplas repetidas. De modo que, *por intuitivo que parezca, de acuerdo con la semántica de UML el diagrama de clases en (b) no permite la existencia de arcos reflexivos*. Podemos considerar varias soluciones a este problema que respeten la restricción TNR, representadas en la **Figura 3.19**:

- Sustituir la multiplicidad 2 en `Nodo` por 1..2, de modo que se permita que un arco esté conectado a un solo nodo, y así el diagrama de la **Figura 3.18** (c) sería correcto. Ahora bien, esto requiere una nueva interpretación, poco intuitiva, de la asociación `Nodo-Arco`: "si una instancia de `Arco` está conectada a una sola instancia de `Nodo`, entonces se trata de un arco reflexivo". Sin esta precisión sobre el significado de la asociación, podemos llegar a la situación absurda de que un arco empiece en un nodo y no termine en ningún otro.
- Desdoblar la asociación `Nodo-Arco` en dos asociaciones distintas, que representen los extremos `inicial` y `final` del arco. Esta solución es adecuada sólo si la multiplicidad original es 2 (arcos binarios); en un caso más general (arcos ternarios, por ejemplo) no es práctico crear tantas asociaciones como sea necesario. Otra desventaja, más importante desde el punto de vista

conceptual, es que hemos introducido una direccionalidad en los arcos (al indicar el extremo inicial y el final), que resulta artificial en el problema del mundo real que estamos tratando, los grafos *no dirigidos*.

- Especializar `Arco` en dos subtipos, `ArcoReflexivo` y `ArcoBinario`, cada uno de ellos asociado independientemente con `Nodo` con su propia multiplicidad. Como en el caso anterior, esta solución es adecuada pero poco general.
- Insertar una clase ficticia (en el sentido de que no representa objetos del mundo real) `ExtremoArco` entre `Arco` y `Nodo`, de modo que todo arco tiene dos extremos, y cada extremo se conecta a exactamente un nodo. Nótese que éste es exactamente el caso de `AssociationEnd` en el metamodelo de UML (ver **Figura 2.20**).

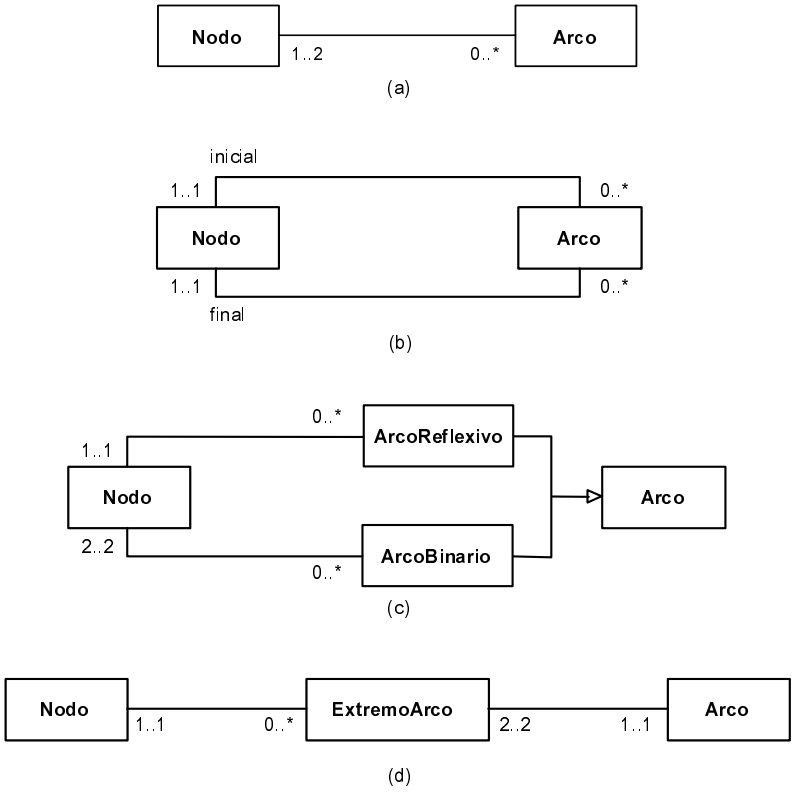

Figura 3.19. Varias formas de resolver el problema de los grafos respetando la restricción de que no puede haber tuplas repetidas en una asociación: a) modificación de la multiplicidad; b) desdoblamiento de la asociación; c) especialización de las clases participantes; d) inserción de una clase ficticia;

De las cuatro soluciones expuestas, la más típica es sin duda la última, ya que la primera es poco intuitiva, y la segunda y la tercera poco generales. De hecho, es una transformación bastante familiar para muchos modeladores cuando se enfrentan con la necesidad de representar tuplas que

enlazan los mismos dos objetos, consistente en sustituir la asociación por una "clase asociativa"[35]. Esta transformación, aplicada al ejemplo de la **Figura 3.14**, daría como resultado el diagrama de la **Figura 3.20**:

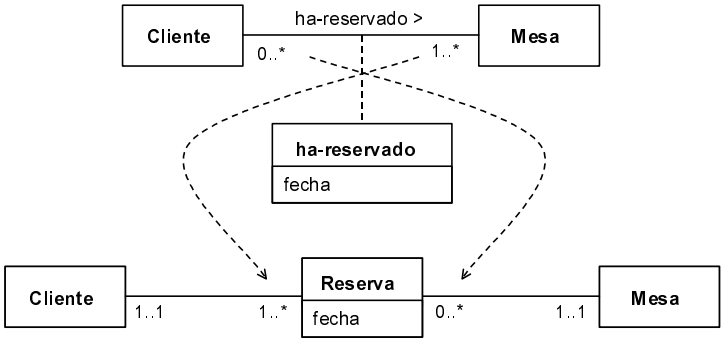

Figura 3.20. Solución al problema de reservas múltiples mediante una clase asociativa

Ya vimos, al estudiar las asociaciones n-arias, que el uso de entidades asociativas supone una pérdida de semántica de las multiplicidades. En el caso de una asociación binaria no es así. Notemos que las multiplicidades originales quedan reflejadas en este nuevo diagrama, pero en posiciones distintas: la multiplicidad 1..* de `Mesa` respecto a `Cliente` es ahora la de `Reserva` respecto a `Cliente`, y lo mismo para la multiplicidad 0..* de `Cliente` respecto a `Mesa`. Las nuevas multiplicidades de `Cliente` y `Mesa` respecto a `Reserva` son ambas 1..1, ya que una reserva está modelando en realidad un enlace entre exactamente un cliente y una mesa (esto es consecuencia de que un enlace binario tiene dos "patas").

Esta solución es válida y muy frecuentemente usada [Fowler 00, Stevens 02]. El inconveniente que presenta es que introduce una *complejidad innecesaria en el modelo*, y requiere el uso de clases ficticias que no representan objetos en el mundo real, sino más bien asociaciones. De hecho, lo que se está haciendo es *desdibujar la diferencia entre clase y asociación*, "cosificar" (*reify*) los enlaces (recordemos que una de las principales críticas al modelo Entidad/Relación es que la misma distinción entre entidad y relación es poco precisa [Codd 90, Date 95]). Una solución mucho más natural sería *prescindir de partida de la restricción TNR*, permitiendo tuplas repetidas en una asociación, es decir, definiendo una asociación como un "saco de tuplas", en lugar de cómo un "conjunto de tuplas". Pero hay una solución mejor perfilada, como vamos a ver a continuación.

[35] Esta *clase asociativa* no debe confundirse con una *clase-asociación*, que es un elemento diferente del lenguaje, que es clase y asociación a la vez. La clase asociativa no es más que una clase normal, sólo que es usada para representar lo que en realidad es una asociación.

3.5.2. En enlace, ¿tupla o elemento?

El Estándar contiene dos visiones contradictorias de lo que es un enlace. Por una parte, "un enlace *es* una tupla" [UML, p. 2-20] y además no tiene identidad propia: "los enlaces no tienen nombre de instancia, toman su identidad de las instancias que relacionan" [UML, p. 3-84]; es decir, un enlace no es nada en sí, es sólo la combinación de otros elementos. Por otra parte, en el metamodelo `Link` es subclase de `ModelElement` [UML, p. 2-98], es decir, *un enlace es un elemento del modelo, y por tanto tiene nombre e identidad propia*, heredados como cualquier otro elemento; además tiene un contenido de datos (*data content*), que es precisamente la tupla de instancias relacionadas [UML, p. 3-71]. Otro problema adicional es el siguiente: si un enlace es meramente una tupla, entonces no podemos distinguir dos enlaces entre los mismos dos objetos pertenecientes a dos asociaciones distintas[36].

Una clase se define como un conjunto de objetos que comparten las mismas propiedades [UML, p. 2-26]. Que los objetos de una clase formen un conjunto implica que *no puede haber objetos repetidos*, sin necesidad de decirlo explícitamente por la propia definición de "conjunto", pero esto no impide que dos objetos de la misma clase tengan el mismo contenido de datos, es decir, que tengan los mismos valores de atributos. Por ejemplo, en una aplicación de diseño gráfico dos instancias de la clase `Punto` pueden tener los mismos valores para el `color` y para las coordenadas `x`, `y`, pero en todo caso son distinguibles porque cada uno tiene su propia identidad. Es decir, en un objeto podemos distinguir su *identidad* de su *contenido* de datos: dos objetos son distintos (no son el mismo) aunque sean iguales (aunque tengan los mismos datos).

Análogamente, una asociación se define como un conjunto de tuplas [UML, p. 2-19] o un conjunto de enlaces [UML, p. 2-20]. ¿Es un enlace exactamente lo mismo que una tupla? Según algunas obras clásicas sobre la orientación a objetos, previas a UML, sí: la instancia de la asociación es la tupla misma, es decir, la identidad del enlace es la composición de las identidades de los objetos enlazados [Martin 95]. Por otra parte, como ya hemos visto, en UML un enlace tiene un contenido de datos (*data content*), que es precisamente la tupla de instancias relacionadas [UML, p. 3-71][37], permitiendo así la distinción en un enlace entre su identidad como elemento del modelo y su contenido de datos, en cercana analogía con los objetos. Si seguimos esta interpretación, entonces podríamos tener conjuntos de

---

[36] Desde el punto de vista de la notación, UML resuelve este problema dando la posibilidad de mostrar el nombre de la asociación junto al enlace, subrayado para indicar que se trata de una instancia [UML, p. 3-84]. En todo caso, puesto que los enlaces son elementos del modelo con nombre heredado, debería permitirse también la representación del nombre de cada enlace, opcional si se quiere, como en el caso de los objetos.

[37] Si el enlace es una instancia de una clase-asociación (un objeto-enlace), puede tener otros atributos que formen parte también de su contenido de datos, además de la tupla.

enlaces, algunos de los cuales tienen el mismo contenido de datos (relacionan los mismos objetos), pero son aun así distinguibles por su identidad. Igual que para los objetos, *no es necesario decir explícitamente que no puede haber enlaces repetidos en cuanto a su identidad*, porque forman un conjunto.

No obstante, UML va más allá e impone la restricción de que los enlaces no pueden estar repetidos en cuanto a su contenido de datos (cada tupla puede aparecer como máximo una vez en la asociación). Si admitimos que las clases son conjuntos de objetos, permitiendo que objetos distintos tengan el mismo contenido de datos, ¿por qué no admitir que las asociaciones son conjuntos de enlaces, permitiendo análogamente que enlaces diferentes tengan el mismo contenido de datos (es decir, que representen la misma tupla)? *La identidad del enlace hace innecesario discutir si las asociaciones son conjuntos o sacos de tuplas*[38].

Stevens propone abandonar la visión simplista de que un enlace meramente *es* una tupla sin identidad, y propone en su lugar esta otra definición [Stevens 02]: un enlace es una instancia de una asociación que *determina* una tupla de instancias de los clasificadores asociados, y por tanto una asociación determina un conjunto de tuplas, es decir, una relación (*relation*) en sentido matemático. Stevens añade que un enlace también *es determinado* por una tupla, pero esta afirmación debe tomarse con reservas. En primer lugar, esto es válido sólo si se admite que no puede haber enlaces repetidos en cuanto a su contenido (como es actualmente el caso en UML). Pero además una tupla, una combinación de instancias, puede determinar un enlace *sólo si se especifica también la asociación* a la que pertenece el enlace: dos enlaces de dos asociaciones distintas entre los mismos dos clasificadores determinan la misma tupla de instancias, y por tanto esta tupla no puede determinar unívocamente el enlace del que procede.

Por tanto, no es necesario definir una asociación como un "saco de tuplas" para resolver los problemas planteados anteriormente. Es mejor definir una asociación como un conjunto de enlaces; cada enlace *es una conexión* entre dos (o más) objetos, y *determina una tupla*; todos los enlaces de una asociación son distintos (en cuanto que tienen identidad propia y por tanto son distinguibles), pero dos o más enlaces pueden conectar los mismos objetos y así determinar la misma tupla (pueden tener el mismo contenido de datos). La asociación, en cuanto conjunto de enlaces, determina una relación (*relation*), que es un conjunto de tuplas. El número de tuplas en la relación puede ser, por tanto, menor que el número de enlaces en la asociación que la determina (ver **Figura 3.21**).

---

[38] Esta identidad del enlace no debe confundirse con una especie de "identificador de enlace", del mismo modo que la identidad del objeto no debe confundirse con el "identificador de objeto", un cierto atributo invisible que facilitaría la identificación del objeto [Genilloud 99].

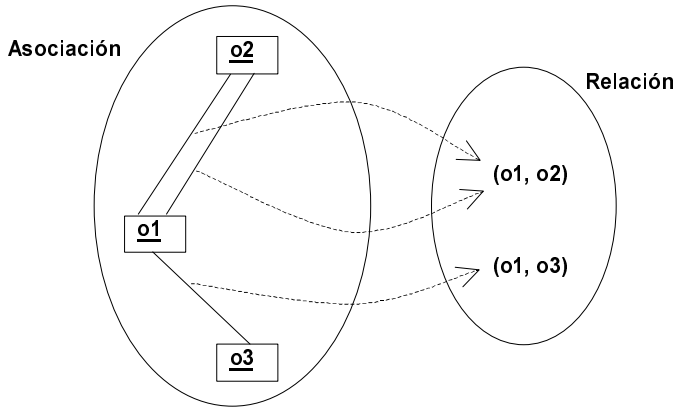

Figura 3.21. La asociación como conjunto de enlaces determina una relación como conjunto de tuplas. Dos enlaces distintos pueden determinar la misma tupla

Ya hemos visto varias *razones que justifican estas definiciones* de asociación y enlace que permiten enlaces "repetidos": a) la necesidad de modelar problemas como la reserva de mesas en un restaurante, o los grafos con nodos y arcos no dirigidos; b) la distinción entre enlace como elemento y enlace como tupla, que está presente en UML, aunque de forma contradictoria; y c) que es bien fácil, si se desea, prohibir enlaces repetidos en cuanto a la tupla que determinan, del mismo modo que se puede prohibir que los objetos de una clase repitan determinados datos. Evidentemente, en determinados contextos no tiene sentido permitir enlaces distintos que determinen la misma tupla, porque no responden a ninguna necesidad del mundo real. ¿Qué necesidad hay de expresar que Pedro es el padre de Marta con dos enlaces distintos? ¿Qué significaría cada uno de los dos enlaces? Si el significado es el mismo, no debe permitirse que coexistan los dos. Para solucionar casos semejantes puede imponerse la restricción a distintos niveles: en una asociación concreta, en un diagrama, en un paquete, en un modelo, en un perfil... Pero en el caso general es mejor dejar libertad al modelador para imponerla o no según la naturaleza del problema.

Por otra parte, la implementación típica de las asociaciones, basada en referencias cruzadas almacenadas como atributos en las clases asociadas, plantea dificultades si se desea permitir que enlaces distintos conecten los mismos objetos. En este tipo de implementación un enlace (bidireccional) se almacena como dos referencias en dos objetos distintos, comprometiendo así su unidad, y por tanto su identidad. Volveremos sobre esta cuestión al tratar de la implementación de las asociaciones en el Capítulo 6.

La propuesta de Stevens (distinguir entre enlace y tupla, definiendo que un enlace es un elemento que determina una tupla) es una clarificación conceptual muy importante, que sin duda debería ser adoptada en el Estándar, pero no implica ninguna modificación sustancial de UML. El punto clave es si debe mantenerse o no en el lenguaje la restricción de que no puede haber enlaces repetidos en cuanto a la tupla que determinan.

Stevens deja abierta esta posibilidad, y sus ideas permiten imponer o no la restricción, pero de una forma conceptualmente mucho más clara: *la restricción es algo añadido y distinto de la naturaleza del enlace*. Si se mantiene la restricción, lo peor que puede ocurrir es que el modelador se vea incomodado, pero siempre existe una salida alternativa.

Algunos analistas sugieren aplicar a la asociación una restricción `{bag}` que permitiría la existencia tuplas repetidas [Fowler 00]. Esto puede ser intuitivo, pero no es una solución válida en UML, ya que una restricción puede *restringir* el significado de un elemento, como lo dice la propia palabra, pero no puede *ampliar* su significado, es decir, "esquivar" una restricción existente en el lenguaje básico o violar otras restricciones aplicadas al elemento [UML, pp. 2-32 y 2-80]. Otros analistas, como ya hemos mencionado, pretenden saltarse la restricción TNR mediante el uso de clases-asociación: en el caso "normal", utilícese una asociación, que implícitamente cumple la restricción TNR; si, "por excepción", se desea repetir enlaces, entonces úsese una clase-asociación[39]. No cabe duda de que un objeto-enlace tiene identidad propia, independientemente de las consideraciones anteriores sobre la identidad del *enlace*, por su naturaleza de *objeto*. Y, si tiene atributos (como `fecha` en `ha-reservado`, ver **Figura 3.20**), un objeto-enlace se distingue fácilmente de otro por sus datos, además de por su identidad. No obstante, el Estándar no menciona que la clase-asociación sea una excepción a la restricción TNR, de modo que debe asumirse que la intención es que debe cumplirla como cualquier otra asociación, ya que `AssociationClass` es subtipo de `Association` en el metamodelo (ver **Figura 2.20**). Stevens y Fowler/Scott confirman también explícitamente esta interpretación [Stevens 02, Fowler 00]. ¿Sería conveniente eximir a la clase-asociación de la restricción TNR, manteniéndola para las asociaciones simples? Bien, puede parecer intuitivo, pero no es una solución "limpia", ya que violaría los principios de la generalización, que exigen que las instancias del subtipo hereden todas las propiedades del supertipo. Como ya hemos argumentado, es mejor eliminar

---

[39] Véase por ejemplo el mensaje de Daniel Jackson a la lista de correo del *Precise UML Group* [pUML] (23 enero 2001). En esta lista de correo hubo, en enero de 2001, un interesante debate sobre estas cuestiones promovido por el autor de esta Tesis Doctoral, bajo el título *"Sets & Bags / Identity of a link"*. El debate puso de manifiesto la falta de claridad en las definiciones de multiplicidad en UML y las diversas interpretaciones contradictorias que permite.

la restricción de la definición básica del lenguaje, proporcionando un mecanismo sencillo para aplicarla cuando sea necesario (por ejemplo, un estereotipo)[40].

[40] Para la definición de "estereotipo" ver la Nota 9 en la Sección 2.2.

4. La navegabilidad de las asociaciones

4.1. Introducción

El concepto de navegabilidad de las asociaciones está pobremente explicado en la documentación oficial de UML. No se distingue adecuadamente de otras asimetrías conceptuales propias de las asociaciones, especialmente de la dirección del nombre de asociación, y a veces es confundida con la visibilidad. No se explica claramente la relación y distinción entre la navegabilidad y la capacidad de enviar mensajes a través de asociaciones, provocando la frecuente confusión de estos dos conceptos. No queda claro el papel de las asociaciones como infraestructura de comunicación entre objetos, ya que en ocasiones se da a entender que puede haber enlaces de comunicación sin que exista ninguna asociación, lo cual contradice el principio de que todo enlace es instancia de una asociación. Se confunde la bidireccionalidad comunicativa de una asociación con la bidireccionalidad lógica de la relación matemática inducida, o se pretende reducir la cuestión de la navegabilidad a un problema de eficiencia de la implementación. Y se omite por completo el tratamiento de la navegabilidad de las asociaciones más complejas (clase-asociación, asociación cualificada y asociación n-aria), como si los aspectos dinámicos sólo tuvieran relevancia para las asociaciones binarias simples.

Una de las mayores dificultades al modelar con UML proviene de la pretensión de abstraer con la misma construcción, la asociación, tanto la estructura estática del sistema como la estructura de interacciones entre objetos, idea heredada del *Object-Relation Model* de Rumbaugh [Rumbaugh 87]. Desafortunadamente, en UML no está bien resuelto el conflicto entre dos nociones diferentes de asociación que mezclan relaciones entre estructuras de datos con relaciones cliente-servidor equivalentes a llamadas a procedimiento entre módulos, confundiendo así la perspectiva de modelado de datos con la perspectiva del modelado de servicios [Simons 99]. Se ha intentado distinguir entre “asociaciones estáticas” y “asociaciones dinámicas” para resolver este problema [Stevens 02], pero esta distinción no es adecuada, ya que en orientación a objetos lo estático y lo dinámico son aspectos de toda asociación, más que la característica definitoria de dos subtipos disjuntos de asociación.

Otra de las dificultades se manifiesta en el debate en torno a la bidireccionalidad de las asociaciones, enérgicamente rechazada por diversos autores. En el curso de esta controversia, los niveles lógico y de

implementación han sido mezclados demasiado a menudo, especialmente cuando el debate se ha centrado en una supuesta oposición entre el uso de asociaciones y el uso de atributos al modelar, como si pudiéramos elegir entre usar asociaciones y descartar los atributos, o viceversa [Graham 97a, Tanzer 95, Velho 94]. Tanto las asociaciones como los atributos son construcciones lógicas necesarias para desarrollar un buen análisis y diseño, y su diferencia semántica se encuentra en el nivel lógico, no en el nivel de implementación. Esta diferencia se entiende mejor desde el concepto de identidad, tal como la expresa Rumbaugh: "utilice asociaciones para mostrar referencias a objetos que tienen identidad[41], y utilice atributos para mostrar valores encapsulados sin identidad" [Rumbaugh 96b][42].

Las asociaciones son necesarias en el modelado orientado a objetos. Sin ellas no tendríamos objetos encapsulados, sino objetos aislados que no pueden interactuar. Toda asociación en un modelo rompe de alguna manera el principio de encapsulamiento, ya que introduce una dependencia entre los objetos asociados que no puede evitarse si queremos que cooperen. Una asociación bidireccional induce una dependencia mutua que es peor que la dependencia simple de una asociación unidireccional, pero, a pesar de esto, UML favorece la bidireccionalidad como opción por defecto. En este sentido las críticas por parte del *OPEN Consortium* [Graham 97b] son pertinentes y no deberían ser ignoradas: podrían servir para evitar el abuso de asociaciones bidireccionales, lo que resultaría en modelos más fácilmente reutilizables.

El resto de este Capítulo está organizado como se describe a continuación[43]. En las Secciones 4.2 y 4.3 se busca una definición "autorizada" de navegabilidad en la documentación oficial, lo cual conduce al concepto de "expresión de navegación", y se examinan otros dos conceptos de UML que están próximos a la navegabilidad, a saber, dirección del nombre de asociación y visibilidad. Las Secciones 4.4 y 4.5 se centran en el estudio de la relación entre la navegabilidad y el envío de mensajes, tocando de lleno el entrelazamiento de los aspectos estático y dinámico de las asociaciones, con una propuesta concreta de distinción entre asociaciones estructurales y contextuales que puede resolver los problemas encontrados en el Estándar. Las Secciones 4.6 y 4.7 están dedicadas al estudio de algunas consecuencias y propiedades de la navegabilidad: dependencia, invertibilidad, eficiencia y notación. Finalmente, la Sección

---

[41] El texto original dice "referencias entre objetos, las cuales tienen identidad" (*interobject references that have identity*), pero creemos que el autor no pensaba en la identidad de la referencia misma, sino en la identidad del objeto referenciado.

[42] Este principio implicaría una clara diferencia semántica entre un atributo (que no tiene identidad) y un objeto componente en una asociación de composición (que sí la tiene), contra la opinión de algunos autores que piensan que son semánticamente equivalentes e intercambiables [Muller 97].

[43] Algunas partes de este Capítulo corresponden a distintos trabajos publicados por el autor de esta Tesis Doctoral [Génova 01, Génova 03a, Génova 03c].

4.8 aplica todos estos conceptos a las asociaciones más complejas (clase-asociación, asociación cualificada y asociación n-aria), cuyos aspectos dinámicos han sido descuidados en UML.

4.2. Definición de navegabilidad

4.2.1. Traversabilidad, accesibilidad

La Guía del Usuario explica pobremente el concepto de "navegación" o "navegabilidad", y no da ninguna definición. Simplemente afirma que "dada una asociación entre dos clases (*class*), es posible navegar desde los objetos de una clase (*kind*) hacia los objetos de la otra clase (*kind*)" [UG, p. 143][44]. Otro término que la Guía del Usuario emplea como sinónimo de "navegación" (*navigation*) es "recorrido" (*traversal*). Ambos términos significan en el lenguaje ordinario algún tipo de movimiento: "navegar" es viajar o ir por el agua en embarcación o nave, transitar o trajinar de una parte a otra; "recorrer" es atravesar un espacio o lugar en toda su extensión o longitud, efectuar un trayecto [DRAE 92][45]. En orientación a objetos la idea de movimiento está relacionada con la interacción entre objetos, el paso de información, el envío de mensajes. Así pues, *el principiante en UML tiende fácilmente a pensar que la navegabilidad es la posibilidad de enviar mensajes a través de asociaciones*, es decir, que una asociación navegable desde la clase `A` hacia la clase `B` significa que los objetos de `A` pueden enviar mensajes a los objetos de `B`: una idea, en realidad, que las explicaciones de la Guía del Usuario no dan a entender tan claramente. Como simplificación razonable, esta idea no es incorrecta, e incluso algunos excelentes libros de texto la enseñan explícitamente [Stevens 00, p. 77, p. 115]. Pero la verdad es más rica, ya que los términos "navegación" y "recorrido" se usan en UML con un significado que no es primariamente el de movimiento, y que sólo secundariamente tiene que ver con el envío de mensajes.

En el Estándar, el término "recorrido" (*traversal*) se usa al definir el meta-atributo `isNavigable` de la metaclase `AssociationEnd`: "especifica si es posible el recorrido desde la instancia origen hacia sus instancias destino asociadas" [UML, p. 2-23]. La explicación de la semántica de la metaclase `Link` (enlace) nos ayuda un poco más, al decir que "el extremo opuesto [de una asociación] define el conjunto de instancias conectadas a la instancia [origen]", y que "para ser capaz de usar un extremo

---

[44] En UML la navegabilidad está definida sólo para asociaciones entre dos clases. El Estándar no menciona la cuestión, pero según el Manual de Referencia el concepto de navegabilidad no se aplica a asociaciones n-arias [RM, p. 354]. Más adelante en este Capítulo estudiaremos cómo se puede aplicar este concepto a las asociaciones de mayor grado (ver Sección 4.8).

[45] He aquí las definiciones en inglés: *"to navigate" is to travel by water, to steer a course through a medium; "to traverse" is to go or travel across or over, to move or pass along or through* [Merriam-Webster].

opuesto en concreto, el correspondiente extremo de enlace (*link end*) unido a la instancia debe ser navegable", y finalmente que un enlace es usado para "acceder" a las instancias asociadas con el fin de comunicarse con ellas, o referenciarlas como argumentos o valores de respuesta en las comunicaciones [UML, p. 2-113]. En otras palabras, *"navegabilidad" se define más o menos como "accesibilidad"* en el contexto de la comunicación, ya sea como destinatario o como contenido del mensaje. El Estándar no añade más ideas significativas acerca de la navegabilidad, al menos usando esta misma palabra. Desafortunadamente, estas pocas explicaciones son insuficientes para formar un concepto claro de "navegabilidad", un concepto que sea suficientemente comprensible como para ser usado con seguridad por modeladores y desarrolladores de herramientas.

4.2.2. Expresiones de navegación

Afortunadamente, el Manual de Referencia se extiende algo más en este tema, dedicando toda una entrada en el capítulo Enciclopedia de Términos. Allí encontramos que "la navegabilidad indica si es posible recorrer una asociación binaria" (nada nuevo hasta este punto) "en las expresiones contenidas en una clase para obtener el objeto o conjunto de objetos asociados con una instancia de la clase" [RM, p. 354]. Y un poco más adelante, "la navegabilidad indica si un nombre de rol puede ser usado en expresiones para recorrer una asociación desde un objeto hacia el objeto o conjunto de objetos de la clase unida al extremo de la asociación marcado con el nombre de rol" [RM, p. 354]. En otras palabras, *podemos usar en expresiones el nombre de rol de un extremo de asociación navegable para obtener el correspondiente objeto* (o conjunto de objetos). Estas expresiones son denominadas "expresiones de navegación" (*navigation expressions*) o "rutas de navegación" (*navigation paths*), están formadas por "secuencias de nombres de atributo o nombres de rol" [RM, p. 356], y *se expresan mediante cadenas de caracteres* (*strings*) en un lenguaje concreto que UML no especifica (puede ser el Lenguaje de Restricción de Objetos, *Object Constraint Language - OCL*, u otro lenguaje tal como un lenguaje de programación conveniente [UML, pp. 2-88 y 3-11]). Las expresiones de navegación se usan con diferentes propósitos:

- para expresar restricciones [RM, p. 354];
- para hacer corresponder un valor a un objeto (*map an object into a value*) [RM, p. 356];
- para usar el objeto referenciado a través del enlace navegable como argumento o valor de respuesta en las comunicaciones [UML, p. 2-114];
- para comunicarse con el objeto referenciado [UML, p. 2-114].

Hasta ahora hemos mostrado que "navegar" (o "recorrer") no significa directamente "enviar un mensaje", sino más bien "obtener un objeto", es

decir, *obtener una ruta o referencia hacia un objeto que permite manejar el objeto destino como un pseudo-atributo del objeto origen*. Tal como lo expresan otros autores, "en orientación a objetos, navegación significa seguir enlaces desde un objeto para *localizar* (*locate*) otro objeto" [Hamie 98]. En orientación a objetos queremos que unos objetos manipulen otros objetos, es decir, que invoquen sus operaciones *públicas*, obtengan o modifiquen sus atributos *públicos*[46], los pasen como parámetros de operaciones a otros objetos, etc. Sin embargo, *para manipular un objeto necesitamos darle un nombre*, un nombre relativo del objeto destino que es válido en el contexto del objeto origen: esto es exactamente lo que proporciona una expresión de navegación, de modo que podemos decir que *navegar es formar la expresión de una ruta que designa el objeto (o conjunto de objetos) destino desde un objeto origen*.

Así pues, la navegabilidad no es directamente la posibilidad de enviar mensajes, sino la posibilidad que un objeto origen tiene de referenciar, nombrar o designar un objeto destino, con el fin de manipularlo o acceder a él en una interacción con intercambio de mensajes[47]. *Uno de los usos* de una expresión de navegación es especificar el objeto receptor de un mensaje, de modo que, indirectamente, la navegabilidad resulta ser una precondición para enviar un mensaje; no obstante, siguen siendo conceptos diferentes.

4.3. Algunos conceptos relacionados

4.3.1. Dirección del nombre

La Guía del Usuario nos advierte que no debemos confundir la dirección del nombre[48] de la asociación (*association name direction*) con la navegación de la asociación (*association navigation*) [UG, p. 66], aunque no clarifica más la cuestión, de modo que debemos acudir al Manual de Referencia para obtener más información. La confusión es posible, ya que ambas características significan algún tipo de direccionalidad de la asociación, y ambas usan una flecha para indicarla.

Una asociación puede tener un nombre que se usa para describir la naturaleza de la relación. Para que no haya ambigüedad acerca de su significado, podemos usar un triángulo de dirección que apunta en la dirección en la que debe leerse el nombre, es decir, hacia la clase designada por la construcción verbal [UML, p. 3-69]. La navegabilidad de la

---

[46] No obstante, en general el uso de atributos públicos debería estar restringido para garantizar mejor el principio de encapsulamiento.

[47] Una definición alternativa: la navegabilidad especifica la capacidad que tiene una instancia de la clase origen de acceder a las instancias de la clase destino por medio de las instancias de la asociación que las conectan.

[48] En castellano sería más correcto traducir *direction* por "sentido" en lugar de "dirección" (una dirección tiene dos sentidos). No obstante, mantendremos la expresión "dirección del nombre" por proximidad a la terminología inglesa.

asociación, en cambio, se expresa gráficamente mediante una flecha abierta en el extremo de la línea de la asociación (ver **Figura 4.1**).

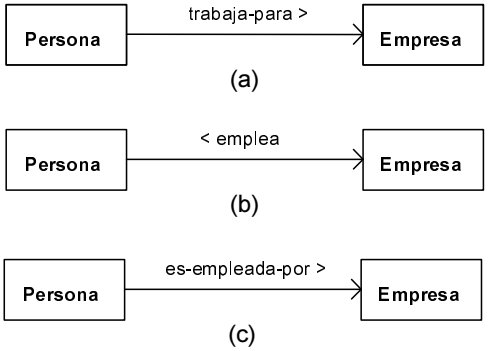

Figura 4.1. Tres asociaciones con la misma navegabilidad pero con distinto nombre y dirección de nombre: a) una persona `trabaja-para` una empresa; b) una empresa `emplea` a una persona; c) una persona `es-empleada-por` una empresa.

El uso del triángulo de dirección del nombre está relacionado con la *asimetría lingüística* o conceptual de la asociación. Esto se explica más claramente en el Manual de Referencia: "Los diferentes extremos de una asociación son distinguibles aunque dos de ellos afecten a la misma clase. (...) Como los extremos son distinguibles, una asociación no es simétrica (*excepto en casos especiales*); los extremos no pueden ser intercambiados. Esto no es más que lo que dice el sentido común en el lenguaje ordinario: el sujeto y el objeto de un verbo no son intercambiables" [RM, p. 50]. El triángulo de dirección del nombre se usa para indicar esta asimetría, de modo que el ejemplo de la **Figura 4.1** (a) puede leerse como "una persona trabaja para una empresa", pero no como "una empresa trabaja para una persona". La asimetría no significa, en este caso, que uno de los roles sea más importante que el otro, tal vez porque se trata de un todo formado por partes (agregación), o porque es un cliente que hace peticiones a un servidor (asociación unidireccional); sólo significa que la expresión verbal humana que da nombre a la asociación tiene un sujeto y un objeto que no son intercambiables.

No hay necesidad para una propiedad de dirección del nombre en el metamodelo de las asociaciones: los extremos de una asociación están inherentemente ordenados [UML, p. 2-14], de modo que la dirección del nombre es la misma ordenación de los clasificadores dentro de la asociación. Es decir, la propiedad subyacente del modelo que está relacionada con la dirección del nombre es la ordenación de los extremos de la asociación [RM, p. 155; UML, p. 3-69]. En la **Figura 4.1** (a), el extremo

de asociación unido a `Persona` es el primer extremo de la asociación, y el extremo unido a `Empresa` es el segundo extremo[49].

Podemos pensar que esos "casos especiales" en los que el Manual de Referencia dice que una asociación es simétrica surgen cuando la asociación especifica algún tipo de equivalencia entre los objetos asociados, tal como `es-igual-a`, `es-amigo-de`, etc.; evidentemente, los dos extremos de la asociación, tienen que estar unidos a la misma clase para que los objetos sean intercambiables, como en los ejemplos de la **Figura 4.2**. No obstante, aunque la asociación sea conceptualmente simétrica, el metamodelo prohíbe la intercambiabilidad de los roles de una asociación: *los roles siempre son distinguibles* gracias a la ordenación inherente de los extremos de la asociación, aunque la dirección del nombre no esté representada. En otras palabras, el metamodelo de UML impide la verdadera representación de asociaciones simétricas, aunque la presentación gráfica pueda inducir a pensar lo contrario, como en la **Figura 4.2** (a). En efecto, "Juan es amigo de María" no implica en UML que "María es amiga de Juan". Esto queda más claro si, en lugar de "es amigo de" decimos "ama a": "Juan ama a María" no implica que "María ama a Juan", ¡sólo que "María es amada por Juan"! En la **Figura 4.2** (b) podemos invertir la dirección del nombre de la asociación `ama` y obtener la asociación recíproca `es-amado-por`, pero no podemos intercambiar los roles `amante` y `amado`; son dos roles distintos, y si una instancia desempeña un rol en un enlace, no desempeña el otro, de modo que la asociación no es simétrica. Por esta razón, el Manual de Referencia no debería decir que hay casos especiales en los que una asociación UML puede ser simétrica.

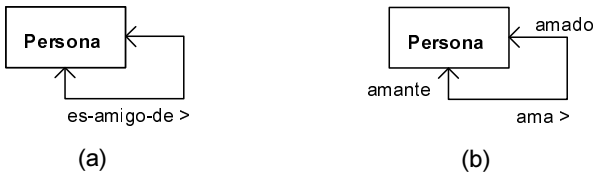

Figura 4.2. Dos asociaciones bidireccionales reflexivas aparentemente simétricas

La especificación de la dirección del nombre es directa cuando el nombre de la asociación es una expresión verbal, que requiere habitualmente un sujeto y un objeto, como en los ejemplos de la **Figura 4.1**. Una expresión verbal en voz pasiva, tal como `es-empleada-por` en la **Figura 4.1** (c), es menos clara: debe recordarse que la dirección del nombre representa la diferencia entre el sujeto y el objeto desde el punto de vista lingüístico, de modo que la cola de la flecha está unida al sujeto

[49] Los clasificadores en una asociación n-aria también están inherentemente ordenados, pero evidentemente esta ordenación no puede expresarse gráficamente mediante una flecha de dirección de nombre. Desafortunadamente, el Estándar no proporciona ninguna notación alternativa para la ordenación de los extremos en una asociación n-aria [UML, p. 3-69].

"gramatical" (en este ejemplo, la persona), sin considerar el sujeto "real" de la acción que se representa (la empresa, sujeto real de la acción "emplear"). Muchos modeladores prefieren el uso de nombres de rol en lugar de nombres de asociación para evitar estos problemas. Cuando el nombre de la asociación es un sustantivo, tal como `trabajo` o `empleo`, puede ser hasta imposible decidir adecuadamente la dirección del nombre de la asociación, ya que un sustantivo no tiene sujeto gramatical (en tanto que nombre de acción, un sustantivo puede tener un sujeto real, pero ya hemos dejado claro que esta no es la regla adecuada para identificar la dirección del nombre). Por supuesto, deberíamos evitar nombrar asociaciones con sustantivos en lugar de verbos, pero esto es un verdadero problema que tiene mala solución en el caso de una clase-asociación, que tiene un único nombre en tanto que asociación y en tanto que clase [UML, p. 3-78], y por tanto su nombre debería ser una expresión verbal y un sustantivo a la vez.

El Estándar dice que "la flecha de dirección del nombre no es semánticamente significativa, sino que es meramente descriptiva" [UML, p. 3-69], una afirmación que consideramos más bien confusa, ya que la interpretación humana de la asociación es ciertamente algún tipo de "significación semántica". Probablemente, esto debe entenderse en el sentido de que la dirección del nombre no impone otras restricciones en el modelo que la ordenación de los extremos de la asociación (aunque esta ordenación es en sí misma también un tipo de significación semántica); específicamente, la flecha de dirección del nombre y la flecha de navegabilidad pueden apuntar en direcciones opuestas, como en el ejemplo de la **Figura 4.1** (b). En otras palabras, *la dirección del nombre no tiene nada que ver con la navegabilidad* en un modelo de objetos, y en consecuencia no tiene nada que ver con la comunicación o interacción entre objetos.

Este "no tiene nada que ver" ha quedado bien establecido en los párrafos precedentes. Sin embargo, desde el punto de vista conceptual ambos tipos de asimetría, dirección del nombre y navegabilidad, están estrechamente relacionados, puesto que el nombre de la asociación denota también la interacción que ocurre entre las clases a través de la asociación. En el caso de una asociación unidireccional, una dirección del nombre que apunte en dirección contraria, como en el ejemplo de la **Figura 4.1** (b), puede ser engañosa, de modo que el nombre de la **Figura 4.1** (a) es preferible. En consecuencia, podemos inferir la siguiente regla de estilo para hacer más legibles los modelos: "cuando una asociación es navegable sólo en una dirección, la dirección del nombre de la asociación debería apuntar en la misma dirección que la asociación misma".

4.3.2. Visibilidad

Navegabilidad y visibilidad son conceptos diferentes, pero es posible confundirlos ya que ambos se refieren en cierto modo a la capacidad que un

objeto tiene de ver y usar las propiedades de otro objeto[50]. Curiosamente, el Manual de Referencia es víctima de la proximidad semántica de estos dos conceptos, como podemos observar en dos lugares: "La falta de navegabilidad implica que la clase opuesta al nombre de rol no puede "ver" la asociación y por tanto no puede usarla para formar una expresión" [RM, p. 355]: esto no es propiamente un error, aunque el verbo "ver" debería ser usado para definir la visibilidad más bien que la navegabilidad. Otro lugar: "El nombre de rol puede llevar un indicador de visibilidad –una punta de flecha– que indica si el elemento en el extremo opuesto de la asociación puede ver el elemento designado por el nombre de rol" [RM, p. 415]: éste es un verdadero error, ya que una *punta de flecha* es un indicador de navegabilidad, no de visibilidad.

Según la Guía del Usuario, la visibilidad de una propiedad (*feature*) de una clase especifica *si puede ser usada por otras clases*. Cada propiedad (atributo u operación) puede estar marcada como "pública" `(+)`, "protegida" `(#)` o "privada" `(-)`; esto significa que la propiedad puede ser usada, respectivamente, por cualquier clase externa (*outside class*) con visibilidad hacia la clase en cuestión[51], por un descendiente de la clase, o sólo por la clase misma [UG, p. 123]. La información dada por el Manual de Referencia [RM, p. 497] y el Estándar [UML, pp. 2-37 y 3-42] es sustancialmente la misma. La versión 1.4 del Estándar añade un nuevo tipo de visibilidad, "paquete" `(~)`, que significa que la propiedad puede ser usada por cualquier clase contenida en el mismo paquete[52].

La visibilidad de un extremo de asociación, que es referenciado por su nombre de rol, es definida exactamente de la misma manera que la de una propiedad [UML, p. 2-23], pero sólo tiene sentido cuando el extremo de asociación es navegable: "*Si la navegabilidad es verdadera*, entonces la asociación define un pseudo-atributo de la clase que está en el extremo opuesto al nombre de rol, es decir, el nombre de rol puede ser usado en expresiones para obtener valores de modo semejante a un atributo de la clase" [RM, p. 354]. En la **Figura 4.3**, por ejemplo, los nombres de rol `cerrojo` y `mango` definen dos pseudo-atributos de la clase `Puerta`, de modo que podemos referirnos a ellos como `Puerta.cerrojo` y

[50] En este Apartado sólo vamos a introducir brevemente el concepto de visibilidad en relación con el de navegabilidad. Para un tratamiento más extenso del problema de la visibilidad, ver el Capítulo 5.

[51] Además de visibilidad, la clase externa *necesita también navegabilidad*, es decir, estar conectada a través de una asociación navegable con la clase en cuestión, como mostraremos en las páginas siguientes.

[52] Bien entendido, cualquier clase en el mismo paquete *y con navegabilidad* hacia la clase dada, como en el caso de visibilidad pública. De forma negativa, la propiedad con visibilidad "paquete" no puede ser usada por una clase externa al paquete, aunque la clase externa esté asociada con la clase poseedora de la propiedad [Stevens 01]. Así pues, la visibilidad "de paquete" es más que "privada", pero menos que "pública". Pensamos que la forma de expresarse del Estándar podría mejorarse para aclarar estos puntos.

`Puerta.mango`. En otras palabras, *la navegabilidad y la visibilidad de las asociaciones son conceptos independientes que se especifican también de modo independiente, pero la capacidad de utilizar una determinada clase, propiedad o extremo de asociación debe ser permitida conjuntamente por ambas características, primariamente por la navegabilidad, y secundariamente por la visibilidad.* Es decir, el extremo de asociación navegable equivale a un pseudo-atributo cualificado por su visibilidad.

Para ilustrar el entrelazamiento entre visibilidad y navegabilidad, consideremos el ejemplo[53] de la **Figura 4.3**:

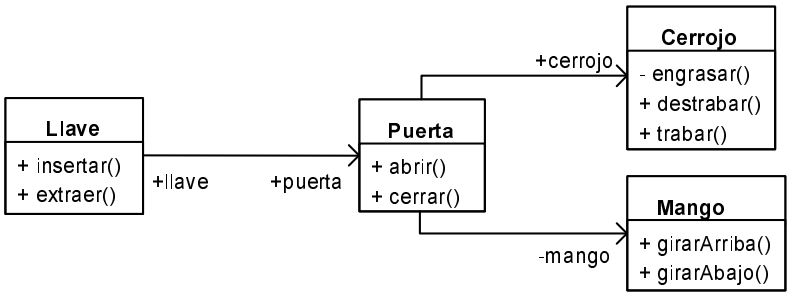

Figura 4.3. Entrelazamiento de visibilidad y navegabilidad

- Un objeto de clase **`Puerta`** puede usar las operaciones `self.abrir`, `self.cerrar`, `cerrojo.destrabar`, `cerrojo.trabar`, `mango.girarAbajo` y `mango.girarArriba`; no puede usar `llave.insertar` ni `llave.extraer` porque la asociación no es navegable hacia la clase `Llave`; no puede usar la operación `cerrojo.engrasar` debido a su visibilidad privada; puede usar las operaciones `mango.girarAbajo` y `mango.girarArriba`, aparentemente a pesar de la visibilidad privada del extremo de asociación mango, porque la clase `Puerta` es la poseedora de este extremo de asociación, exactamente igual que si el rol fuera uno de sus atributos, de modo que la visibilidad privada no le afecta.
- Un objeto de clase **`Cerrojo`** puede usar las operaciones `self.engrasar`, `self.trabar` y `self.destrabar`; no puede usar otras operaciones debido a la falta de navegabilidad en su asociación con `Puerta`.
- De modo similar, un objeto de clase **`Mango`** puede usar sólo las operaciones `self.girarAbajo` y `self.girarArriba`.

[53] Obviamente, éste es sólo un ejemplo simplificado que sirve a nuestros propósitos en esta cuestión. En un modelo más realista, la clase `Llave` probablemente estaría asociada con la clase `Cerrojo`. Hemos omitido muchos detalles por simplicidad, tales como las indicaciones de multiplicidad.

- Finalmente, un objeto de clase **`Llave`** puede usar las operaciones `self.insertar`, `self.extraer`, `puerta.abrir`, `puerta.cerrar` y, gracias a la visibilidad pública del rol `cerrojo`, puede también usar `puerta.cerrojo.destrabar` y `puerta.cerrojo.trabar`; no puede usar `puerta.cerrojo.engrasar` debido a la visibilidad privada de la operación; no puede usar las operaciones `puerta.mango.girarAbajo` y `puerta.mango.girarArriba` porque el rol `mango` es privado: sólo objetos de la clase `Puerta` pueden usar este extremo de asociación.

Como hemos visto, *la navegabilidad actúa como una especie de precondición para la visibilidad de una asociación*, y de los atributos y operaciones de la clase asociada. No tiene sentido especificar que un extremo de asociación tiene visibilidad pública[54] sin ser navegable (véase `Puerta.llave` en el ejemplo, que es público en vano por no ser navegable), y además el modelo se vuelve menos legible. Probablemente, en este caso el rol debería quedarse sin nombre: *un extremo de asociación que no es navegable implica que el nombre de rol no tiene ninguna utilidad*. En consecuencia, podemos inferir la siguiente regla de estilo: "cuando una asociación es navegable sólo en una dirección, el extremo de asociación que no es navegable no debería tener especificada la visibilidad, e incluso el rol debería quedar sin nombre".

4.4. El envío de mensajes

4.4.1. Ruta navegable y operación visible

Consideremos ahora más de cerca la relación entre navegabilidad y comunicabilidad. Puesto que un mensaje puede ser una señal o, con más frecuencia, la invocación de una operación, hay dos formas de enviar un mensaje [RM, p. 333]:

- Un objeto (el *emisor*) envía una señal a uno o más objetos (los *receptores*).
- Un objeto (el *emisor* o *llamante*) invoca una operación en otro objeto (el *receptor*).

El envío de un mensaje es el resultado de una acción en el objeto emisor[55]. En el metamodelo (véase la **Figura 4.4**), la metaclase `Action`

[54] De hecho, no tiene sentido especificar ningún tipo de visibilidad en un extremo de asociación no navegable.

[55] No debe confundirse la acción de enviar un mensaje (que tiene lugar en el objeto emisor) con la acción que típicamente ocurre como resultado de la recepción del mensaje (y que tiene lugar en el objeto receptor). Así pues, un mensaje habitualmente implica dos acciones, una en el emisor y otra en el receptor.

especifica *el destinatario de la acción* (*target*) por medio de una expresión de conjunto de objetos (*object set expression*) que se resuelve en cero o más instancias [UML, pp. 2-99 y 2-115], de modo que, en principio, tanto para una señal como para la invocación de una operación el receptor puede ser un conjunto de objetos [RM, p. 334] (como hemos visto en el párrafo precedente, sin embargo, la documentación a veces sugiere que el receptor de una invocación de operación debe ser un único objeto, mientras que los receptores de una señal pueden ser múltiples).

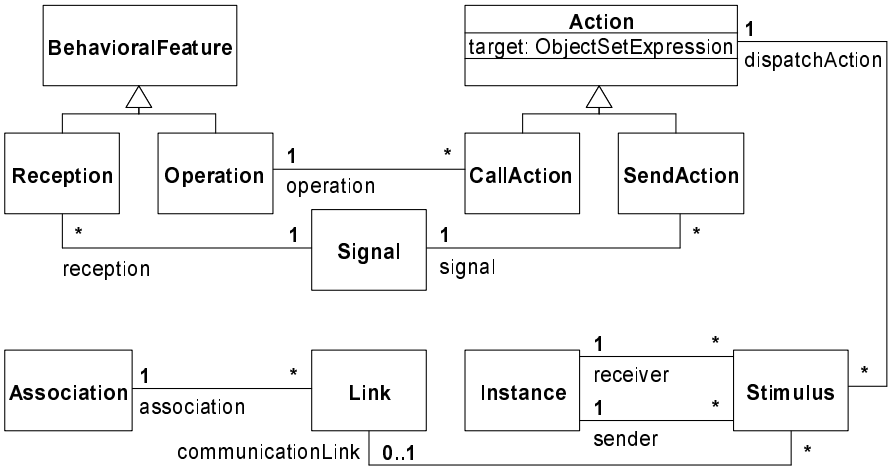

Figura 4.4. Metamodelo de los mensajes, extraído de las Figuras 2-5, 2-14, 2-15, 2-16 y 2-17 en el Estándar

La especificación de un mensaje, por tanto, consiste en la expresión de la *acción* que se desea ejecutar[56], con parámetros si es necesario, así como la expresión de la lista deseada de *destinatarios*, es decir, el objeto u objetos que se espera que ejecuten el servicio requerido. Por tanto, para que un objeto envíe un mensaje a otro objeto necesitamos expresar la ruta y la acción:

- La ruta (*path*) se expresa como una *ruta navegable* desde el emisor hacia el receptor, por medio de la cual el objeto emisor "tiene conocimiento" del objeto receptor y puede especificarlo. Esta ruta es el resultado de una expresión de navegación, que es el valor del meta-atributo `Action.target` [UML, p. 2-99], que es heredado tanto por `CallAction` como por `SendAction` [UML, pp. 2-100 y 2-105]. Así pues, la lista deseada de destinatarios es

[56] De hecho, acción o secuencia de acciones: "En el metamodelo una `ActionSequence` es una `Action`, que a su vez es una agregación de otras `Actions`" [UML 2-99].

especificada mediante la expresión de navegación que se resuelve en una expresión de conjunto de objetos[57].

- La acción (*action*) se expresa como una *operación visible* en la clase del objeto receptor, que puede ser invocada desde la clase del objeto emisor, es decir, una operación pública[58]. Alternativamente, para una señal necesitamos una *recepción* (¿pública?) en la clase del objeto receptor, que declare que el objeto está preparado para reaccionar ante la llegada de la señal[59]. Así pues, en el metamodelo la acción que se desea ejecutar en el receptor es *directamente* especificada en el caso de una `CallAction` mediante una `Operation` asociada [UML, p. 2-100], e *indirectamente* en el caso de una `SendAction` por medio de una `Signal` asociada [UML, p. 2-105][60].

Como hemos visto, para que tenga lugar la comunicación entre objetos se requiere tanto navegabilidad como visibilidad (podemos expresarlo así: accesibilidad = navegabilidad + visibilidad). El envío de mensajes requiere rutas navegables y operaciones visibles: *un objeto puede comunicarse sólo con otros objetos de los que tiene conocimiento, y que han puesto las operaciones (o recepciones) deseadas a disposición de sus clientes en sus interfaces.* Estas ideas son más bien simples, pero encontramos que no están clara y concisamente expresadas en la documentación oficial de UML.

4.4.2. Representación de los mensajes en los diagramas de UML

¿Cómo representamos el envío de mensajes en los diagramas de UML? Veámoslo en los distintos tipos de diagramas dinámicos. En un **diagrama de estados** el envío de un mensaje puede expresarse textualmente como *una acción vinculada a una transición* entre dos estados (ver **Figura**

---

[57] Una expresión de conjunto de objetos también puede obtenerse como valor de retorno de una operación. En este caso tendríamos una especie de "acción anidada" para seleccionar las instancias destino, que a su vez ejecutarían la "acción principal".

[58] Excepto en el caso de que el emisor y el receptor pertenezcan a la misma clase, porque entonces la operación es visible sin necesidad de ser pública. Nótese que en UML un objeto puede acceder a las propiedades privadas, tanto atributos como operaciones, de otros objetos de la misma clase con los que esté enlazados. Ver Apartado 5.2.1.

[59] La recepción designa una señal y especifica el comportamiento esperado como respuesta; en el metamodelo `Reception` es, como `Operation`, una subclase de `BehavioralFeature` [UML, p. 2-104]. Desafortunadamente, las explicaciones sobre `Reception` son bastante pobres en la documentación. No es seguro que la recepción deba ser pública -el signo de interrogación es intencionado- ya que el objeto emisor realmente no necesita referenciarla cuando envía la señal.

[60] Por otra parte, y esto tampoco lo dice el Estándar, la operación (o, en su caso, la recepción) debe estar definida en al menos uno de los clasificadores a los que pertenece la instancia para que ésta pueda responder al mensaje. En UML una instancia puede pertenecer simultáneamente a uno o más clasificadores [UML, p. 2-102], lo que se conoce como *multiclasificación*.

4.5), o también dentro de un estado como transición interna, o como acción de entrada o de salida. En esta notación textual la ruta desde el emisor hacia el receptor queda expresada explícitamente en la cláusula de destino de la expresión de la acción.

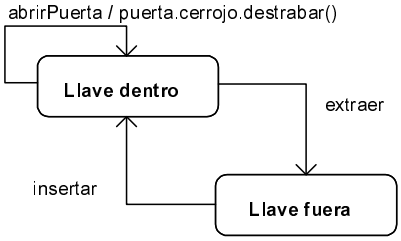

Figura 4.5. Diagrama de estados de la clase `Llave` que muestra el mensaje `destrabar()` enviado a un objeto asociado de la clase `Cerrojo` como consecuencia de la transición `abrirPuerta`. La ruta hacia el objeto receptor se muestra explícitamente mediante la expresión de navegación `puerta.cerrojo`

El Manual de Referencia presenta una notación gráfica alternativa en la que se dibuja una flecha discontinua desde la línea de la transición hacia un recuadro que alberga la máquina de estados del receptor [RM, p. 420], aunque esta notación es completamente ignorada por el Estándar. Para los **diagramas de actividad** existe una notación gráfica similar (ver **Figura 4.6**), en la que "el envío de una señal puede mostrarse mediante un pentágono convexo (...). Puede dibujarse una *flecha discontinua* desde el vértice del pentágono hacia un símbolo de objeto para mostrar el receptor de la señal" [UML, p. 3-166]. En esta representación gráfica no se muestra la expresión de navegación, y la existencia de la ruta es implícita: la flecha discontinua no significa la ruta, sino la acción misma de enviar.

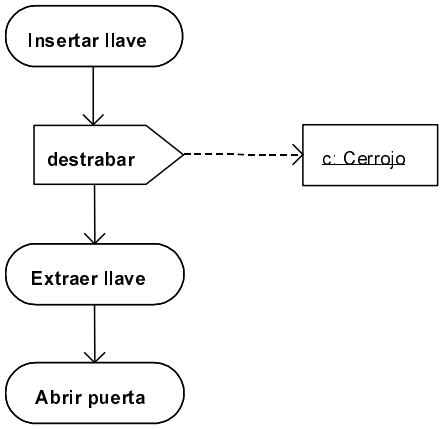

Figura 4.6. Diagrama de actividad que muestra la apertura de una puerta, en donde el mensaje `destrabar` es enviado a un objeto de clase `Cerrojo`. La existencia de la ruta hacia el receptor es implícita, y su expresión permanece desconocida

En un **diagrama de secuencia** el envío de un mensaje se representa mediante una *flecha continua* que comienza en la línea de vida (*lifeline*) del

emisor y termina en la línea de vida del receptor (ver **Figura 4.7**). Nótese que, como en el diagrama de actividad, la existencia y expresión de la ruta desde el emisor hacia el receptor es implícita: cada flecha representa un mensaje individual, y la ruta misma no se muestra.

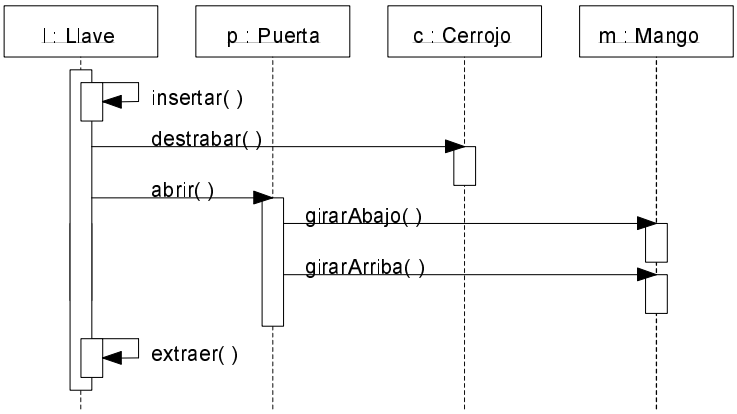

Figura 4.7. Diagrama de secuencia que muestra la interacción necesaria para abrir una puerta. Aquí la existencia de la ruta hacia el objeto receptor también es implícita

Finalmente, en un **diagrama de colaboración** se representan explícitamente *tanto el mensaje enviado como la ruta que sigue el mensaje* desde el emisor hacia el receptor (ver **Figura 4.8**). Esta ruta se muestra mediante un enlace que según el Estándar es "una instancia de una asociación" [UML, pp. 2-102 y 2-113], y que puede tener una punta de flecha para indicar que es navegable sólo en un sentido [UML, p. 3-115] (supuestamente, porque la asociación que lo especifica también tiene este tipo de navegabilidad). El mensaje se muestra mediante una pequeña flecha junto al enlace entre instancias, apuntando en la dirección que corresponda; puede haber más de un mensaje junto a un único enlace, si la ruta es usada varias veces en la interacción.

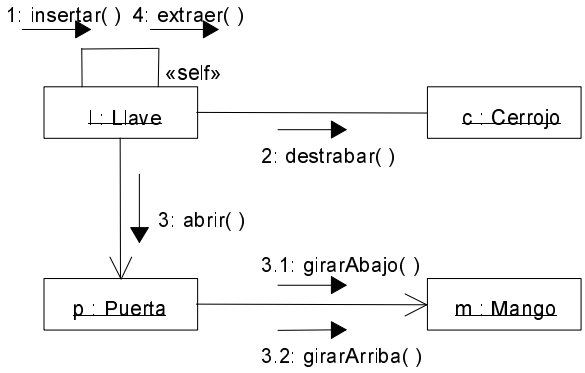

Figura 4.8. Diagrama de colaboración que muestra la misma interacción de la **Figura 4.7**. Las rutas seguidas por los mensajes se muestran explícitamente en forma de enlaces entre objetos.

Las diferentes notaciones que hemos examinado se resumen en la **Tabla 4.1**. Nótese que la acción (la acción que se desea ejecutar) se representa siempre textualmente mediante el nombre del mensaje enviado, que puede ser una invocación de operación o una señal, mientras que la ruta (la lista deseada de destinatarios) a veces está implícita y a veces se expresa explícitamente, ya sea textual o gráficamente.

Diagrama	Ruta	Acción
Diagrama de estados	Expresión de navegación (textual)	Nombre de mensaje (textual)
Diagrama de actividad	Implícita	
Diagrama de secuencia	Implícita	
Diagrama de colaboración	Enlace de comunicación (gráfico)	

Tabla 4.1. Resumen de las notaciones para el envío de mensajes en los diferentes tipos de diagramas dinámicos en UML

Esperamos que la representación coherente del envío de mensajes en los diferentes diagramas de UML y el seguimiento desde las expresiones de navegación hasta los enlaces de comunicación haya ayudado a comprender mejor el metamodelo subyacente de los mensajes y enlaces. Con todo, la representación implícita de la ruta en los diagramas de actividad y de secuencia puede ser problemática, especialmente cuando hay una ambigüedad acerca del enlace de conexión, que puede llevar a los desarrolladores a interpretar mal un diseño. Esta ambigüedad puede darse incluso cuando la ruta se representa explícitamente, pero de modo gráfico, como ocurre en un diagrama de colaboración, ya que la notación gráfica no transmite suficiente información para identificar unívocamente la procedencia de la ruta o enlace[61].

4.4.3. ¿Es todo enlace de comunicación una instancia de una asociación?

En el ejemplo de la **Figura 4.8**, los enlaces `l→p` y `p→m` tienen un indicador de navegabilidad que debe estar de acuerdo con las asociaciones de la **Figura 4.3**. Por otra parte, la ruta entre los objetos `l` y `c` no es instancia de ninguna de las asociaciones mostradas en la **Figura 4.3**, ni tampoco es un enlace estereotipado[62], sino más bien una combinación de los

[61] En UML es posible expresar el nombre de la asociación junto al enlace, subrayado para indicar que se trata de una instancia [UML, p. 3-84]. En cambio, no hay ninguna manera estándar de expresar que el enlace procede de una expresión de navegación más compleja. Una posible solución sería unir al enlace una nota que contenga la expresión.

[62] Para la definición de "estereotipo" ver la Nota 9 en la Sección 2.2.

enlaces `l→p` y `p→c` (este último no mostrado en el diagrama). ¿Cuál sería el indicador de navegabilidad en el "enlace" `l—c`?[63]

El Estándar dice que "obviamente esta flecha [la flecha del mensaje] no puede apuntar hacia atrás en una ruta unidireccional" [UML, p. 3-115]. Bien, esto puede parecer obvio en este momento, cuando ya hemos establecido que un objeto puede comunicarse sólo con otros objetos de los que tiene conocimiento (ver Apartado 4.4.1), pero también hemos mostrado que las ideas dispersas en la documentación oficial no explican suficientemente bien las implicaciones de la navegabilidad para el envío de mensajes (ver Sección 4.2). Aun más, admitimos la verdad de que un mensaje no puede viajar en contra de la navegabilidad del enlace, pero advirtiendo que la navegabilidad del enlace puede ser diferente de la navegabilidad de la asociación. Consideremos los dos ejemplos siguientes en la **Figura 4.9** y la **Figura 4.10** (los ejemplos son de Stevens, a los que añadimos nuestra interpretación):

- En la **Figura 4.9** existe una asociación unidireccional desde la clase `A` hacia la clase `B`; un objeto de clase `A` envía un mensaje que lo contiene a sí mismo como argumento a un objeto de clase `B`, y el objeto `B` usa el argumento para devolver un mensaje al objeto `A` [Stevens 01]. El mensaje desde el objeto `A` hacia el objeto `B` usa un enlace unidireccional que es una instancia de la asociación unidireccional; en cambio, el mensaje desde el `B` hacia el `A` usa un enlace unidireccional en dirección opuesta, que no es instancia de esta asociación, sino un enlace con el estereotipo «`parameter`».
- En la **Figura 4.10** existe una asociación unidireccional desde la clase `A` hacia la clase `B`, y otra asociación unidireccional desde la clase `A` hacia la clase `C`; un objeto de clase `A` envía un mensaje que contiene un objeto de clase `B` como argumento a un objeto de clase `C`, y el objeto `C` usa este argumento para enviar un mensaje al objeto `B` [Stevens 02]. Una vez más, el mensaje desde el `C` hacia el `B` usa un enlace con el estereotipo «`parameter`» donde no existe ninguna asociación, ni siquiera con navegabilidad invertida.

[63] Más aún, ¿es la ruta desde el objeto `l` hacia el objeto `c` un "enlace"? Bien, en UML 1.4, es un enlace en el sentido de que conecta dos objetos en un diagrama de colaboración, y es usado para transmitir mensajes; no es un enlace en el sentido de que no es una instancia de una asociación. Una contradicción que es el *leit motiv* de esta Sección, e intentaremos resolver en la Sección 4.5.

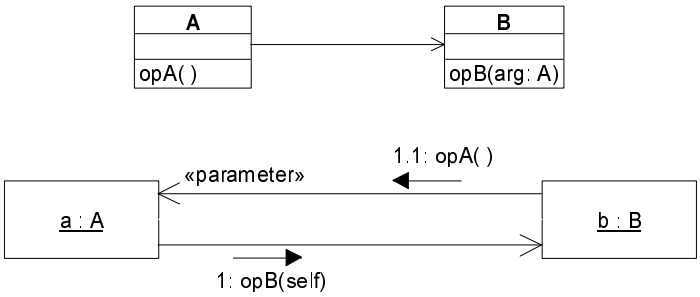

Figura 4.9. Diagrama de colaboración que usa un enlace con el estereotipo «`parameter`» en contra de la navegabilidad de la asociación `A`→`B`

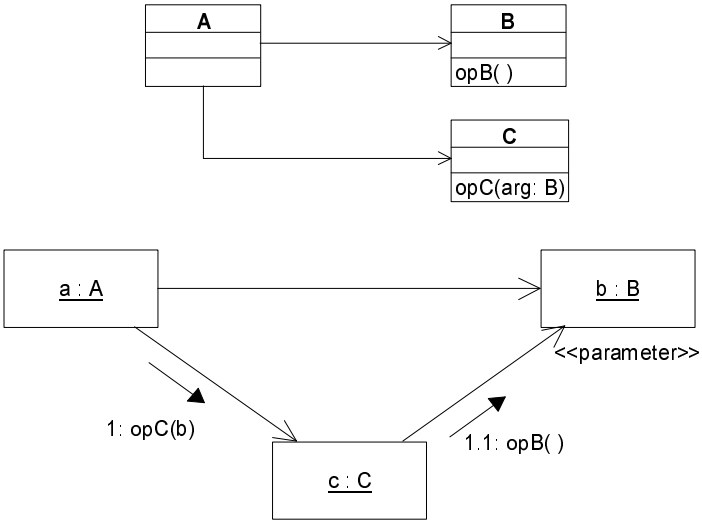

Figura 4.10. Diagrama de colaboración que usa un enlace con el estereotipo «`parameter`» sin que exista ninguna asociación `C`→`B`

Estos ejemplos, aparentemente, muestran que *un enlace no es siempre una instancia de una asociación*. En el primer ejemplo, el enlace tiene su propia navegabilidad, opuesta a la navegabilidad de una asociación que existe entre las clases correspondientes. En el segundo ejemplo, hay un enlace navegable donde no existe ninguna asociación. No obstante, los enlaces usados para la comunicación pueden ser puestos en relación con enlaces que sí son instancias de asociaciones. Podemos concluir que *hay múltiples formas de construir rutas navegables, pero todas ellas están basadas en último término en asociaciones navegables.*

Como hemos visto, la afirmación de que "un enlace en un diagrama de colaboración es una instancia de una asociación" plantea dos tipos diferentes de problemas. En primer lugar, puede haber *enlaces estereotipados* [UG, p. 210; UML, p. 2-103] que aparentemente no corresponden a ninguna asociación en el modelo (por ejemplo, un enlace entre un objeto y él mismo, véase el enlace `l`—`l` en la **Figura 4.8**; o el

enlace entre un objeto y el argumento recibido en un mensaje, véase el enlace `b→a` en la **Figura 4.9** y el enlace `c→b` en la **Figura 4.10**), pero que constituyen a pesar de todo una conexión entre instancias a través de la cual se pueden enviar mensajes. Esta cuestión está lejos de haber sido clarificada, como demuestran los continuos debates e investigaciones recientes [Stevens 02][64]. En segundo lugar, la ruta desde el emisor hacia el receptor puede representarse mediante una expresión de navegación que es una "secuencia de nombres de atributos o nombres de rol" [RM, p. 356], es decir, una ruta podría ser una *combinación de enlaces* más bien que un único enlace, de modo que obviamente no puede ser la instancia de una única asociación. Ahora bien, en un diagrama de colaboración nos encontramos con un dilema: o bien representamos esta ruta compuesta como una única línea que conecta las instancias del emisor y del receptor (véase el enlace `l—c` en la **Figura 4.8**), o bien fragmentamos la ruta en sus enlaces componentes. Pero en este último caso podríamos encontrarnos con que no hay ninguna operación en las clases intermedias que consista simplemente en retransmitir el mensaje hasta que alcance su destino final. Y si estamos obligados a fragmentar la ruta y retransmitir el mensaje a través de los objetos intermedios, ¿qué sentido tiene tener asociaciones visibles y permitir expresiones de navegación compuestas? De la existencia de enlaces estereotipados y rutas compuestas debemos concluir, aparentemente, que *no toda ruta o enlace en un diagrama de colaboración es una instancia de una asociación*: hay rutas que son enlaces estereotipados, y hay rutas que son combinaciones de enlaces.

El Estándar es más bien contradictorio en relación con esta cuestión, y da dos soluciones distintas al problema:

- *A veces un mensaje no usa ningún enlace de comunicación*. Después de afirmar que una instancia de mensaje, también llamado "estímulo" (*stimulus*), "utiliza un enlace entre el emisor y el receptor para la comunicación", el Estándar reconoce *algunas situaciones especiales en las que puede faltar este enlace de comunicación*: "si el receptor es un argumento dentro de la activación en curso, o una variable local o global, o si el estímulo es enviado a la misma instancia emisora" [UML, p. 2-114]. Así pues, los enlaces `b→a` en la **Figura 4.9** y `c→b` en la **Figura 4.10** serían "enlaces ficticios", y no se requieren las asociaciones `B→A` ni `C→B`.
- *A veces un enlace no es instancia de una asociación*. El Estándar define *cinco estereotipos estándar* para `LinkEnd` (`«global»`,

[64] Ver también las contribuciones a la lista de correo del *Precise UML Group* [pUML] durante los años 2000-2001 sobre los temas *"Links & messages", "Link as instance, tuple, path", "Sets and bags",* y *"Dependencies and associations",* en donde el autor de esta Tesis Doctoral desempeñó un activo papel.

«`local`», «`parameter`», y «`self`», además del redundante «`association`») para manejar estas mismas situaciones especiales [UML, p. 2-103], en las que encontramos comunicación sin asociaciones[65]. Por tanto, los enlaces `b→a` y `c→b` serían verdaderos enlaces, pero no derivados de la existencia de asociaciones `B→A` o `C→B`, sino de "otras circunstancias". De nuevo, no se requiere que existan esas asociaciones.

La primera solución es coherente con la afirmación de que un enlace *es* una instancia de una asociación, representado en el metamodelo por la meta-asociación obligatoria que especifica el enlace (multiplicidad 1 en el rol `Link.association`, ver **Figura 4.11**), y también es coherente con la afirmación de que el enlace es utilizado *opcionalmente* por el mensaje para la comunicación (como se ve por la multiplicidad 0..1 en el rol `Stimulus.communicationLink`): algunas veces el mensaje utiliza un enlace (que es instancia de una asociación), y a veces el mensaje no usa ningún enlace (así que ninguna asociación se ve implicada).

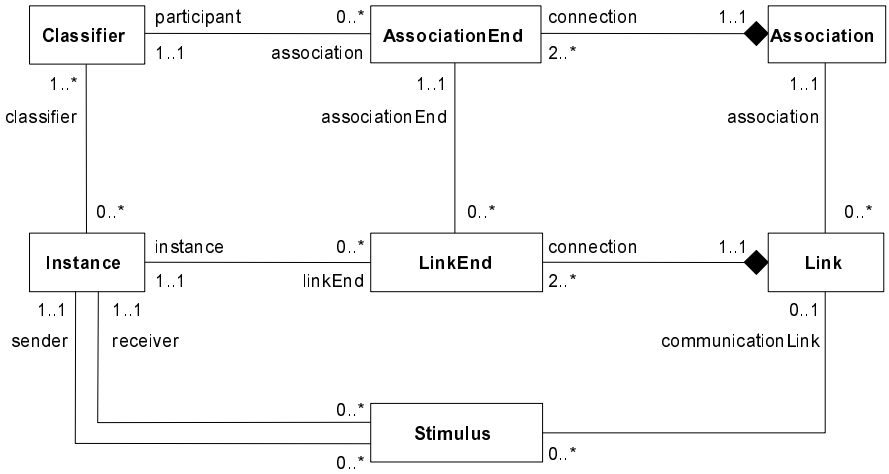

Figura 4.11. Metamodelo de los enlaces de comunicación, extraído de las Figuras 2-6, 2-16 y 2-17 en el Estándar

La segunda solución rompe el principio de que todo enlace es instancia de una asociación, y contradice además la primera solución: si el enlace de comunicación es opcional, ¿qué sentido tiene definir estos enlaces especiales? En cambio, es coherente con la representación habitual de interacciones en los diagramas de colaboración, donde un mensaje *siempre* usa un enlace.

Ninguna de las dos soluciones es satisfactoria. Si los enlaces son opcionales, ¿cuál es la representación de un mensaje que es enviado a través de un enlace inexistente en un diagrama de colaboración? Admitir enlaces

[65] En el Apartado 4.5.2 veremos la definición concreta de estos estereotipos.

ficticios no es una buena idea. Pero afirmar que hay enlaces que no son instancias de asociaciones no es mejor. Al igual que un objeto, un enlace es "algo concreto" (una instancia); por tanto un enlace, como un objeto, necesita conformarse a un "tipo" que especifique sus propiedades. El tipo de un objeto es una clase, y el "tipo" de un enlace debería ser una asociación, que especifica, entre otras propiedades, las clases de los objetos enlazados, la navegabilidad, modificabilidad, etc. Si un enlace no tuviera tipo no seríamos capaces de decir cuáles son sus propiedades y cómo debe comportarse[66]. Por tanto, en la siguiente Sección intentaremos buscar otra solución que sea conceptualmente mejor y logre evitar los "enlaces sin fundamento" (*baseless links*), como han sido denominados [Stevens 02].

4.5. Asociaciones estructurales y contextuales

4.5.1. Asociaciones estáticas y dinámicas

UML define la asociación como una "relación semántica" entre clasificadores [UML, p. 2-19]. Pero ¿qué tipo de relación semántica? ¿Qué *significa* una asociación? ¿Cómo identificamos ese factor en el dominio del problema o de la solución que debe ser modelado como una asociación? Por ejemplo, ¿qué asociaciones deberían existir en la **Figura 4.10**? ¿Tendríamos que añadir una asociación desde la clase `C` hacia la clase `B` para representar la relación de comunicación?

Stevens ha propuesto la interesante distinción entre asociaciones estáticas y dinámicas [Stevens 02], que llevaría a definir dos subtipos de un concepto genérico de "asociación" en el metamodelo:

- Una asociación estática (*static association*) expresa una *relación estructural* entre clases (o, en general, clasificadores), típicamente implementada mediante referencias mutuas, es decir, el código de cada una de las clases asociadas incluirá al menos un atributo que contenga referencias a objetos de la otra clase (el número de referencias permitidas dependerá de la multiplicidad).
- Una asociación dinámica (*dynamic association*) expresa una *relación de comportamiento* entre clases, que típicamente implica que en el sistema implementado habrá algún tipo de intercambio de mensajes entre objetos de las clases asociadas (por lo que puede denominarse también *relación comunicativa*).

Esta distinción se representaría en el metamodelo mediante dos nuevas metaclases `StaticAssociation` y `DynamicAssociation`, que serían subtipos de la metaclase `Association`. Para distinguirlas en

[66] Existen lenguajes de programación orientados a objetos no tipados, como Eiffel o Smalltalk, y aunque es completamente legítimo emplear UML para modelar sistemas que se implementen con estos lenguajes, UML en sí mismo es fuertemente tipado, más en la línea de lenguajes como Java.

un diagrama, Stevens propone el uso de los estereotipos «`static`» y «`dynamic`». En el ejemplo de la **Figura 4.10**, la asociación `A`→`B` sería estática, la asociación `A`→`C` sería estática y dinámica a la vez (o simplemente estática), y debería existir además una asociación dinámica `C`→`B`. No obstante, esta clasificación de asociaciones requiere hacer algunas precisiones:

Asociación estática

- Que una asociación sea "estática" no significa que sus enlaces sean fijos y no cambien durante el funcionamiento del sistema. En el ejemplo típico que ya hemos considerado varias veces, la asociación `trabaja-para` entre `Persona` y `Empresa`, lo habitual es que cambien los enlaces entre personas y empresas, incluso muchas veces y muy frecuentemente. Lo que significa una asociación estática no es la "estaticidad" de los enlaces, sino que cada clase contiene una referencia a la otra clase en su propia estructura estática, independientemente de los mensajes que reciba. Ciertamente, en un problema del mundo real puede haber asociaciones cuyos enlaces no cambien (salvo para crearlos al principio y destruirlos al final), pero esta característica ya se expresa en UML con la propiedad de "modificabilidad" (ver Secciones 2.3.2.5 y 2.4.4.4).
- Una asociación estática existe independientemente de la comunicación, pero también permite el intercambio de mensajes, como hemos visto en los ejemplos del Apartado 4.4.3; es más, no tiene sentido establecer una asociación estática entre clases si no se usa de alguna manera para la comunicación, tal vez no directamente enviando mensajes a los objetos enlazados, pero sí al menos usándolos como argumentos o valores de retorno de los mensajes (ver Apartado 4.2.2). Es el caso de la **Figura 4.10**, en donde el objeto `a:A` usa el enlace con `b:B` no como destinatario de un mensaje, sino como argumento en el mensaje enviado a `c:C`. Sin embargo, las asociaciones estáticas son insuficientes para modelar los enlaces de comunicación, porque existen enlaces que no son instancias de asociaciones estáticas.
- La existencia de una asociación estática puede deducirse de la implementación, sin necesidad de ejecutar el código para comprobar si existe o no la asociación. No obstante, como veremos en el Capítulo 6, la ingeniería directa e inversa de las asociaciones estáticas no es tampoco trivial.

Asociación dinámica

- La relación de una asociación dinámica con la implementación es mucho más oscura. No sería correcto definir que existe una asociación dinámica cuando *de hecho* hay intercambio de mensajes. En primer lugar, determinar la existencia de una asociación dinámica exigiría la observación del código en ejecución en todos los casos posibles, teniendo en cuenta además la interacción con el entorno, lo que conllevaría considerables dificultades técnicas, y privaría a la noción de asociación dinámica de utilidad práctica, ya que nos interesa establecer si existe o no una tal asociación ya en las fases de análisis y diseño (ingeniería directa), y no sólo comprobarlo tras el examen de la implementación en ejecución (ingeniería inversa). En segundo lugar, en determinados casos la comprobación sería una cuestión indecidible por principio: es decir, en general no podemos decidir, a partir de la inspección del código, si dos instancias de dos clases determinadas intercambiarán o no un mensaje, ya que éste es un problema de dificultad semejante al Problema de la Parada de Turing[67]. Por lo tanto, debemos conformarnos con definir que existe una asociación dinámica cuando *potencialmente* hay intercambio de mensajes.
- Como ya hemos demostrado en la Sección 4.4, para que pueda haber un intercambio de mensajes es necesario que el emisor tenga conocimiento del receptor, es decir, el emisor deberá tener alguna referencia que señale al receptor. No obstante, no es necesario que esta referencia esté almacenada en un atributo de la clase emisora (es decir, no es necesario que la referencia proceda de una asociación estática), sino que puede tratarse de un argumento o valor de retorno recibido en un mensaje previo, o de un objeto creado en el curso de la respuesta a un mensaje. En todo caso, la implementación de una asociación dinámica también requiere el uso de algún tipo de referencia, es decir, el intercambio potencial de mensajes implica la existencia de una cierta estructura en la clase del objeto emisor.

---

[67] Consideremos las familias de clases A_1, A_2, ... A_n y B_1, B_2, ... B_n, donde las instancias de la clase A_i simulan el comportamiento de la i-ésima máquina de Turing, y envían un mensaje a una instancia de B_i al terminar la simulación. Del mismo modo que es indecidible, en general, si se parará o no la i-ésima máquina de Turing [Turing 36], es indecidible si la instancia de A_i enviará o no un mensaje a la instancia de B_i. Por tanto, si ligamos la existencia de una asociación dinámica al intercambio real de mensajes, no podemos decidir si existe o no una asociación dinámica entre A_i y B_i. Este argumento es una adaptación del que da Stevens [Stevens 02].

Resumiendo, las asociaciones estáticas son insuficientes para modelar los enlaces de comunicación, de modo que el concepto de asociación dinámica, o algún otro semejante, es necesario. Pero la distinción entre ambas no puede basarse en que "las asociaciones estáticas se implementan como referencias, mientras que las asociaciones dinámicas se implementan como intercambio de mensajes". Así definidas, no constituyen dos subtipos disjuntos de asociación: como hemos visto, las asociaciones estáticas permiten el intercambio de mensajes, y las asociaciones dinámicas requieren el uso de referencias[68].

4.5.2. El contexto de las asociaciones

Si la diferencia estática/dinámica no constituye una adecuada clasificación de las asociaciones, ¿cómo podemos distinguir las asociaciones "normales" de otros tipos de asociación que parecen ser relevantes en el modelado? Ya hemos mencionado que en UML existen cinco estereotipos predefinidos para los extremos de enlace (metaclase `LinkEnd` en el metamodelo), con los que se pretende resolver la cuestión de las asociaciones dinámicas, es decir, cómo puede una instancia comunicarse con otra sin que exista una asociación (estática) entre las respectivas clases. Los cinco estereotipos especifican distintas formas en que la instancia puede ser visible[69] [UML, p. 2-103]:

- «`association`»: la instancia es visible vía asociación.
- «`global`»: la instancia es visible porque está en un contexto (*scope*) global relativo al enlace.
- «`local`»: la instancia es visible porque está en un contexto local relativo al enlace.
- «`parameter`»: la instancia es visible porque es un parámetro relativo al enlace.
- «`self`»: la instancia es visible porque es el originador (*dispatcher*) de una petición.

Para los extremos de asociación (metaclase `AssociationEnd` en el metamodelo) existen los mismos cinco estereotipos, aunque con definiciones ligeramente distintas que vale la pena recoger y comparar, ya que las anteriores no son muy explicativas [UML, p. 2-24]:

---

[68] Es más, según Stevens podríamos tener incluso asociaciones que son estáticas en un extremo y dinámicas en el otro: en el ejemplo de la **Figura 4.9**, en lugar de establecer dos asociaciones unidireccionales, una estática `A→B` y otra dinámica `B→A`, podríamos establecer una única asociación bidireccional, estática en el extremo `B` y dinámica en el extremo `A`.

[69] Nótese cómo el Estándar incurre en imprecisión terminológica al mezclar el concepto de visibilidad en estas definiciones. En lugar de "visible" debería decir "accesible".

- «`association`»: especifica una asociación real (*real association*); es la opción por defecto y es redundante, aunque puede usarse como énfasis.
- «`global`»: el destino (*target*) es un valor global conocido por todos los elementos, más que una verdadera asociación (*actual association*).
- «`local`»: la relación representa una variable local dentro del procedimiento (*procedure*), más que una verdadera asociación.
- «`parameter`»: la relación representa un parámetro del procedimiento, más que una verdadera asociación.
- «`self`»: la relación representa una referencia al objeto que posee (*owns*) la operación o acción, más que una verdadera asociación.

No está muy clara la intención del Estándar al definir estos cinco estereotipos. Por una parte, parece que debemos suponer una regla de coherencia, en el sentido de que si un extremo de enlace lleva un estereotipo, el extremo de asociación correspondiente debe llevar el mismo estereotipo; pero el Estándar no impone esta restricción. Por otra parte, parece que la intención de los cuatro estereotipos «`global`», «`local`», «`parameter`», y «`self`» es dar un acceso que no se deriva propiamente de una asociación [UML, p. 2-103], sino de otras circunstancias, apoyando la tesis de que no todo enlace de comunicación es instancia de una asociación; es decir, que un extremo de enlace estereotipado no se correspondería con ningún extremo de asociación, ni siquiera con uno que tenga el mismo estereotipo, incurriendo en contradicción con la regla de coherencia sugerida, y además haciendo superfluos los estereotipos para extremos de asociación (es decir, sólo serían necesarios para extremos de *enlace*)[70].

Posiblemente esta paradoja se deba más a una redacción descuidada del Estándar que a una verdadera inconsistencia. Podemos suponer que la intención era afirmar que todo enlace es instancia de una asociación, pero que hay algunos enlaces "especiales" que no son instancias de asociaciones normales, sino de asociaciones especiales *implícitas*, que existen sin que sea necesario declararlas en el modelo, aunque se pueda hacer por claridad.

Podemos relacionar el uso de estereotipos de asociación y enlace con un principio de la Ingeniería del Software que conviene respetar para comprender y minimizar mejor las dependencias entre distintos elementos del diseño, conocido como "Ley de Demetria" (*Law of Demeter*)[71], que se

---

[70] Ya hemos mencionado otra contradicción en el Estándar, cuando afirma que en determinadas situaciones especiales (las mismas para las que se definen estos cuatro estereotipos) no es necesario el enlace de comunicación [UML, p. 2-114].

[71] En alusión a la diosa griega Démeter o Demetria. Se trata de una regla de buen estilo para diseñar sistemas orientados a objetos, inventada en 1987 por Karl Lieberherr y Ian Holland [Lieberherr 89], mientras trabajaban en el *Demeter Research Group*

puede resumir así: "no hable con desconocidos". En otras palabras, cuando objetos de dos clases tienen la posibilidad de intercambiar un mensaje (existe un enlace entre ellos), el código de la clase emisora debe ponerlo claramente de manifiesto (el enlace se conforma a una asociación declarada entre las dos clases). Es decir, todo enlace de comunicación es instancia de una asociación (estática o dinámica), y toda asociación debe ser declarada en el código. La Ley de Demetria establece que, en respuesta a un mensaje $m(a_1, a_2, \ldots a_n)$, un objeto o sólo debe enviar mensajes a los siguientes objetos (añadimos la equivalencia con los estereotipos):

- Objetos directamente enlazados con o, es decir, referenciados por sus atributos (estereotipo «association»).
- El objeto o mismo (estereotipo «self»).
- Un objeto global conocido por todos los objetos (estereotipo «global»)[72].
- Cualquier objeto recibido como argumento en el mensaje m, es decir, los a_1, a_2, ... a_n que sean objetos más que valores elementales (estereotipo «parameter»).
- Objetos creados y destruidos por o en el curso de su respuesta a m (estereotipo «local»)[73].

El estereotipo por defecto «association» equivale más o menos a una *asociación estática explícita*, según la terminología de Stevens. Los estereotipos «self» y «global», por otra parte, serían *asociaciones estáticas implícitas* (existen independientemente del intercambio de mensajes, es decir, del comportamiento). Finalmente, los estereotipos «parameter» y «local» corresponderían a *asociaciones dinámicas implícitas* (están directamente relacionados con la invocación de operaciones, es decir, mensajes). No obstante, ya hemos hecho notar que no nos satisfacen plenamente los términos "asociación estática" y "asociación dinámica" de Stevens, ya que toda asociación tiene propiedades estáticas y dinámicas (es decir, toda asociación está implicada en la estructura y en el comportamiento del sistema), por tanto lo estático y lo dinámico son dos aspectos de toda asociación, de modo que no son criterios adecuados para

---

(Northeastern University, Boston, USA), y popularizada por diversos autores como Booch, Budd, Coleman, Larman, Page-Jones, Rumbaugh, etc.

[72] Stevens omite la mención de los objetos globales en su exposición de la Ley de Demetria [Stevens 02], aunque Lieberherr y Holland los incluyen "por razones pragmáticas". Este "valor global que es conocido por todos los elementos" [UML, p. 2-24] sería equivalente a una clase pública con una única instancia, en aplicación del patrón de diseño *Singleton* [Gamma 94], de modo que puede ser referenciada sin necesidad de instanciar un enlace. Stevens tampoco relaciona la Ley de Demetria con los estereotipos.

[73] Este último caso no debe confundirse con la creación de un objeto que siga existiendo, enlazado con o, al término de la ejecución de la operación; el enlace con este objeto correspondería al estereotipo «association».

clasificarlas[74]. Además, estos términos pueden ocultar el hecho de que todo enlace es "dinámico", en el sentido de que es creado y destruido dinámicamente (los enlaces "estáticos" de hecho cambian).

En su lugar, proponemos la clasificación de asociaciones en *asociaciones estructurales* (estereotipo «`association`») y *asociaciones contextuales* (estereotipos «`self`», «`global`», «`parameter`» y «`local`»), es decir, asociaciones que son válidas dependiendo del contexto[75]. Recuperamos así el término que acuñó Rumbaugh (*contextual association*) para referirse a determinadas dependencias de uso entre clases (*usage dependencies*) [Rumbaugh 98]. La **Tabla 4.2** resume ambas clasificaciones.

<table>
<tr><th>Estereotipo</th><th>Stevens
(comportamiento)</th><th>Génova
(contexto)</th></tr>
<tr><td>«association»</td><td rowspan="3">Asociación estática</td><td>Asociación estructural</td></tr>
<tr><td>«self»</td><td rowspan="4">Asociación contextual</td></tr>
<tr><td>«global»</td></tr>
<tr><td>«parameter»</td><td rowspan="2">Asociación dinámica</td></tr>
<tr><td>«local»</td></tr>
</table>

Tabla 4.2. Dos posibles clasificaciones para asociaciones

Las asociaciones contextuales serán, en general, unidireccionales:

- Esto es bastante claro en el caso de «`global`», ya que la variable global no tendrá ninguna referencia hacia los objetos que la conocen por el contexto.
- Así mismo, la dirección de una asociación «`self`» no tiene relevancia (el objeto está enlazado consigo mismo en cualquiera de las dos direcciones), por lo que por simplicidad es mejor considerar que es siempre unidireccional.
- Cuando un objeto es pasado como argumento en un mensaje, sólo el objeto receptor del mensaje tiene conocimiento del enlace que se

[74] Otra posible nomenclatura es *asociaciones persistentes* (asociaciones cuyos enlaces persisten entre distintas invocaciones de las operaciones de una clase) frente a *asociaciones transitorias* (asociaciones cuyos enlaces existen sólo dentro de una invocación de operación). El concepto es el mismo, pero la terminología es aún menos adecuada porque hace excesivo hincapié en la duración temporal del enlace, que tanto en un caso como en el otro puede ser larga o breve.

[75] Otra importante diferencia entre asociaciones estructurales y contextuales: aparentemente, el *grado* de las asociaciones contextuales debe ser siempre 2, es decir, sólo las asociaciones estructurales pueden ser n-arias con n>2. El uso de asociaciones n-arias para la comunicación es tratado en el Apartado 4.8.2.

crea en el contexto de la respuesta al mensaje, por tanto la asociación «`parameter`» también es unidireccional.

- Finalmente, es posible que un objeto cree una asociación bidireccional con una variable local dentro de un método, pero esto resulta un tanto oscuro, ya que la variable local y la asociación sólo existen dentro del contexto de la respuesta al mensaje; si la variable local usara la asociación para enviar mensajes al objeto que la ha creado, éste debería responderlos sin haber concluido la respuesta al primer mensaje, lo cual puede ser confuso; por tanto, es recomendable que la asociación «`local`» sea también unidireccional.

Por último, queda el problema de los enlaces originados en una expresión de navegación que combine varias asociaciones (estructurales o contextuales). Según Stevens, un enlace de este tipo sería instancia de una *asociación derivada*, que debe considerarse dinámica (o contextual, en nuestra clasificación), puesto que no está declarada en la estructura de las clases participantes. Pero notemos que el uso de asociaciones derivadas viola la Ley de Demetria enunciada más arriba, puesto que el objeto emisor no tiene conocimiento directo del objeto receptor (no están conectados mediante un enlace "físico"). En concreto, a diferencia de lo que ocurre con los otros tipos de asociaciones contextuales, en una asociación derivada la clase del objeto emisor no conoce cuál es la clase[76] del objeto receptor, es decir, no conoce su interfaz, qué mensajes puede enviarle. Si el código empleara directamente asociaciones derivadas para enviar mensajes introduciría oscuras dependencias entre clases, lo que indica un diseño inseguro, y es posible que algunos lenguajes de programación orientados a objetos ni siquiera permitan la compilación del código. Esto no impide por completo el uso de asociaciones derivadas (es decir, expresiones de navegación compuestas) para la comunicación: en lugar de enviar el mensaje directamente al objeto (o conjunto de objetos) seleccionado por la expresión de navegación, puede almacenarse primero la referencia al objeto destino en una variable local (con lo que ya es necesario especificar su clase y la dependencia es explícita), y después enviar el mensaje al objeto referenciado por la variable local, de modo que estaríamos en el caso de un enlace con el estereotipo «`local`».

4.5.3. Propuesta sobre estereotipos de asociaciones para el Estándar de UML

En esta Sección proponemos brevemente los cambios requeridos para mejorar la semántica y notación de los estereotipos de asociaciones y enlaces, aplicando las ideas desarrolladas en la Sección precedente,

[76] En realidad, el "tipo", no la "clase". No podemos analizar aquí la confusión entre tipos y clases que hay en UML [Simons 02].

especialmente la distinción entre asociaciones estructurales y contextuales, y la Ley de Demetria.

Esta propuesta implica la corrección de la multiplicidad del rol `Stimulus.communicationLink` [UML, p. 2-98] (cambiar de 0..1 a 1..1), una nueva definición de los estereotipos de las metaclases `AssociationEnd` y `LinkEnd` [UML, pp. 2-24 y 2-103], y una nueva explicación de cómo los mensajes utilizan los enlaces para la comunicación [UML, p. 2-114].

Asociaciones

- Se pueden distinguir dos tipos de asociaciones. Una *asociación estructural* especifica una relación entre clasificadores que es definida en la propia estructura estática de los clasificadores asociados. Una *asociación contextual* especifica una relación entre clasificadores que es válida sólo en determinados contextos de los clasificadores asociados.
- Toda asociación debe ser declarada en un modelo bien formado, para especificar sus propiedades y para evitar interacciones incoherentes con el resto del modelo.
- No es necesario que toda asociación aparezca en algún diagrama de clases. Basta que la asociación esté representada en el modelo subyacente.
- Los distintos tipos de asociaciones se distinguen por el estereotipo aplicado a cada asociación.
- Una asociación derivada no puede tener instancias directas; si se desea acceder a objetos accesibles mediante expresiones de navegación compuestas, es necesario asignar antes el valor de la expresión de navegación a una variable local.

Enlaces

- En un modelo bien formado, todo enlace es instancia de una asociación.
- En fases iniciales del desarrollo de un modelo está permitido representar enlaces sin especificar su asociación.
- Un enlace tiene el mismo estereotipo que su asociación.
- Todo estímulo o instancia de mensaje requiere obligatoriamente un enlace de comunicación.

Estereotipos

Se proponen cinco estereotipos predefinidos para las metaclases `Association` y `Link`. El primero corresponde a asociaciones estructurales, y el resto a asociaciones contextuales. Nótese que estos

estereotipos ya no se aplican a los extremos de asociación y enlace, representados en el metamodelo por las metaclases `AssociationEnd` y `LinkEnd`, sino a las asociaciones y enlaces mismos. Para el primer estereotipo proponemos una nueva palabra clave (*keyword*) que nos parece más adecuada que la vigente «`association`»[77].

- «`structural`»: se aplica a una asociación que especifica enlaces estructurales; opción por defecto y redundante, pero puede usarse como énfasis.
- «`self`»: se aplica a una asociación contextual reflexiva, implícita en todo clasificador, que especifica el enlace implícito que toda instancia tiene consigo misma; esta asociación es unidireccional y con multiplicidad 1..1 en el extremo destino, que lleva el nombre de rol `selfTarget`.
- «`global`»: se aplica a la asociación contextual que especifica el enlace entre una instancia y un objeto global conocido en el contexto de la instancia; la asociación es unidireccional desde el clasificador de la instancia hacia el clasificador del objeto global.
- «`parameter`»: se aplica a las asociaciones contextuales que especifican los enlaces entre una instancia y los parámetros de sus propiedades dinámicas (*behavioral features*); estos enlaces sólo existen en el contexto de la ejecución de las propiedades dinámicas, en respuesta a los mensajes recibidos por la instancia; la asociación es unidireccional desde el clasificador de la instancia hacia el clasificador del parámetro, y el nombre de rol en el extremo destino es el nombre del parámetro.
- «`local`»: se aplica a las asociaciones contextuales que especifican los enlaces entre una instancia y las instancias creadas localmente por sus propiedades dinámicas (*behavioral features*) como variables locales; estos enlaces sólo existen en el contexto de la ejecución de las propiedades dinámicas, en respuesta a los mensajes recibidos por la instancia; la asociación es ordinariamente unidireccional desde el clasificador de la instancia hacia el clasificador de la variable local, y el nombre de rol en el extremo destino es el nombre de la variable local.

4.5.4. Representación de asociaciones contextuales

Llegados a este punto, debemos examinar la representación de las asociaciones contextuales, en concreto, cómo se representan estas

[77] Por otra parte, sería imposible conservar la palabra clave «`association`» en esta propuesta, ya que un estereotipo no puede tener el mismo nombre que la metaclase a la que se aplica [UML, p. 2-81]. En la versión actual de UML, aunque la palabra clave «`association`» sea inadecuada y confusa, esta regla no es formalmente violada, ya que el estereotipo no se aplica a `Association` sino a `AssociationEnd`.

asociaciones y sus enlaces en diagramas de clases, diagramas de objetos, y diagramas de colaboración.

4.5.4.1. Diagramas de clases y de objetos

En un *diagrama de clases* que represente la estructura global del sistema o subsistema seguramente será conveniente omitir las asociaciones contextuales y mostrar sólo las asociaciones estructurales, para no recargarlo excesivamente. No obstante, también es posible, y a veces conveniente, representar en un diagrama de clases un contexto más particular, empleando los distintos estereotipos que hemos definido en el Apartado precedente para hacer explícitas las asociaciones contextuales. Rumbaugh propone representar las asociaciones contextuales en un diagrama de clases como dependencias de uso (*usage dependencies*) [Rumbaugh 98], pero esto presenta ciertas desventajas. En primer lugar, como veremos en el Apartado 4.6.1, cualquier asociación induce de por sí una dependencia, de modo que no está claro por qué deban representarse las asociaciones estructurales como asociaciones, y las contextuales como dependencias. En segundo lugar, si una asociación contextual se representa como dependencia, se oculta su carácter de "asociación" con todas sus implicaciones y propiedades: instanciación, multiplicidad, modificabilidad, navegabilidad, visibilidad, etc. En tercer lugar, si el objetivo es no recargar el diagrama, usar dependencias en lugar de asociaciones no sirve de gran ayuda.

Un buen método para explicitar todas las asociaciones que afectan a una clase puede ser representar, además del diagrama de clases de estructura global en el que se define la clase, un diagrama de clases contextual para cada una de las operaciones de la clase. En la **Figura 4.12** podemos ver el diagrama de clases estructural de parte de un sistema bancario, que muestra sólo las asociaciones estructurales conectadas a la clase `Cuenta`. Esta clase define los atributos `número` y `saldo` (de tipos `NumeroCuenta` y `Moneda`, considerados como tipos de datos básicos) y la operación `emitirTransferencia`, de cuya signatura podemos inferir una asociación contextual con la misma clase `Cuenta`, pero ninguna otra asociación contextual.

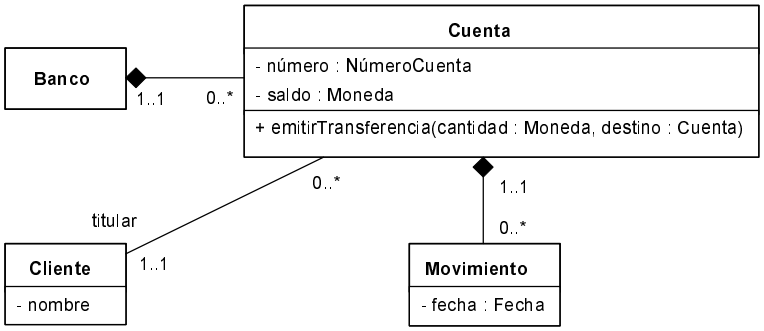

Figura 4.12. Diagrama de clases estructural de una cuenta bancaria

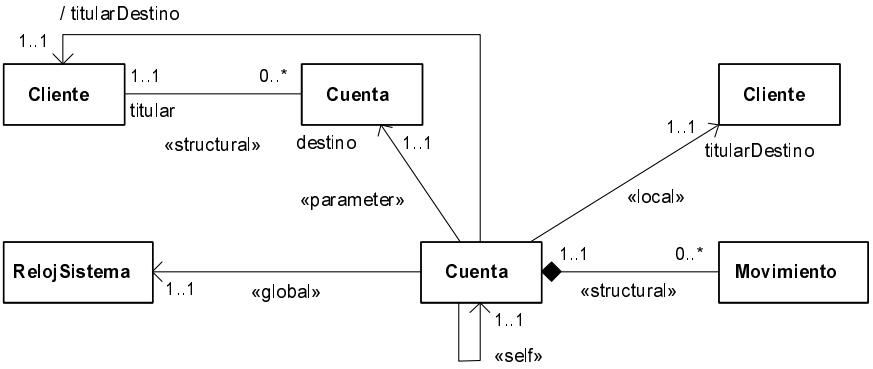

Figura 4.13. Diagrama de clases contextual de la operación `emitirTransferencia`

La operación `emitirTransferencia` almacena los datos relevantes de la transferencia (fecha actual, número de la cuenta destino, nombre del propietario de la cuenta destino y cantidad transferida) como un nuevo movimiento en la colección de movimientos de la cuenta, actualiza el saldo de la cuenta, y advierte a la cuenta destino para que pueda actualizar sus datos. En la **Figura 4.13** se muestra el diagrama de clases contextual de esta operación, con el resto de asociaciones contextuales explicitadas. Se repite la asociación estructural con `Movimiento`, ya que es relevante en el contexto de la operación. Además se representan las siguientes asociaciones contextuales:

- Una asociación «`self`» con la cuenta misma.
- Una asociación «`global`» con `RelojSistema`, del que habrá que leer la fecha para almacenarla en el movimiento correspondiente a la transferencia efectuada.
- Una asociación «`parameter`» con el argumento `destino` de la operación, de clase `Cuenta`, cuyo nombre de rol es el nombre del argumento.
- La clase que representa la cuenta destino muestra una asociación estructural con su `titular`, de clase `Cliente`, de modo que

existe una asociación contextual derivada que permite construir la expresión de navegación `destino.titular`[78].

- Para observar la Ley de Demetria, esta asociación derivada no es utilizada directamente, sino que se lee su valor y se almacena en la variable local `titularDestino`, de donde surge una asociación «`local`» con `Cliente`.

La **Figura 4.14** representa un estilo diferente de diagrama de clases estructural en el que, siguiendo la sugerencia de Rumbaugh, las asociaciones contextuales que no son estructurales al mismo tiempo se expresan como dependencias de uso. Así pues, se omiten las dependencias hacia `Cuenta`, `Movimiento` y `Cliente`. En nuestra opinión, nuestra propuesta es mucho más expresiva y útil para entender las relaciones de `Cuenta` con las otras clases.

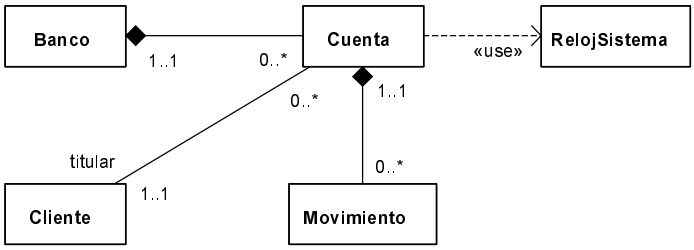

Figura 4.14. Diagrama de clases estructural de una cuenta bancaria, con las asociaciones contextuales de la operación `emitirTransferencia` expresadas como dependencias

En un *diagrama de objetos* se representa una situación concreta del sistema, una especie de fotografía del estado del sistema en un cierto instante de tiempo [UML, p. 3-35], con los objetos y enlaces implicados. Un diagrama de objetos, como un diagrama de clases, es "estático", en el sentido de que no representa un comportamiento sino una estructura. Pero esto no impide que se puedan mostrar enlaces contextuales en un diagrama de objetos. Al igual que ocurre con el diagrama de clases, si se desea representar la situación global del sistema se mostrarán sólo enlaces estructurales, pero si se desea representar un contexto particular no hay ningún inconveniente en mostrar juntos enlaces estructurales y enlaces contextuales, con los necesarios estereotipos para distinguirlos convenientemente. Ésta es una razón más para preferir nuestra terminología, "enlace contextual", frente a la de Stevens, "enlace dinámico", porque parecería contradictorio mostrar enlaces *dinámicos* en un diagrama de objetos *estático*.

4.5.4.2. Diagramas de colaboración

Un *diagrama de colaboración en el nivel de instancia* es similar a un diagrama de objetos al que se añadieran los mensajes intercambiados en la

[78] El nombre de rol se muestra precedido por una barra (/) para indicar que se trata de una asociación derivada [UML, p. 3-93].

interacción, por tanto valen las mismas consideraciones del párrafo precedente. En concreto, aunque en un diagrama de colaboración los enlaces contextuales tengan especial importancia, también es posible representar enlaces estructurales, ya que estos forman parte esencial del contexto de la colaboración y también pueden ser usados para enviar mensajes, o como argumentos o valores de retorno en los mensajes. La **Figura 4.15** muestra el diagrama de colaboración de la operación `emitirTransferencia`, con los objetos, enlaces y mensajes concretos que son intercambiados durante la ejecución de la operación. El último mensaje de la secuencia es enviado a la cuenta destino de la transferencia para que actualice de forma recíproca sus movimientos (los argumentos se omiten por simplicidad).

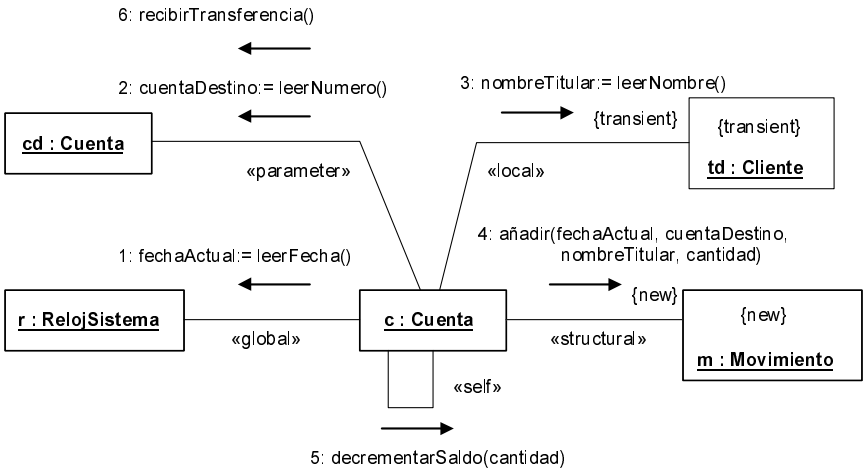

Figura 4.15. Diagrama de colaboración de la operación `emitirTransferencia`

Por último, un *diagrama de colaboración en el nivel de especificación* es similar a un diagrama de clases, con la peculiaridad de que representa roles-clasificador y roles-asociación (metaclases `ClassifierRole` y `AssociationRole`) en lugar de clasificadores y asociaciones[79], y tipos de mensajes (metaclase `Message`) en lugar de instancias de mensajes (metaclase `Stimulus`), con el fin de mostrar tipos o patrones de interacción, en lugar de interacciones concretas. Teniendo en cuenta las consideraciones precedentes, parece claro que en estos diagramas se pueden representar tanto asociaciones estructurales como contextuales, aunque estas últimas tengan especial relevancia.

---

[79] En el metamodelo, `ClassifierRole` es subtipo de `Classifier`, `AssociationRole` es subtipo de `Association`, y `AssociationEndRole` es subtipo de `AssociationEnd` [UML, p. 2-118]. No vamos a entrar en el estudio detallado de los diagramas de colaboración en el nivel de especificación, tal vez uno de los puntos más oscuros y peor resueltos del Estándar [Steimann 00]. En todo caso, nuestros diagramas de clases contextuales son en cierto modo similares a diagramas de colaboración en el nivel de especificación, pero con un enfoque más simple que evita las metaclases de rol.

4.5.4.3. Herramientas CASE

Las herramientas CASE podrían tener en cuenta todas estas indicaciones. Seguramente no conviene, de modo general, exigir en un modelo que cualquier enlace que aparezca en un diagrama de objetos o de colaboración se corresponda con alguna asociación en un diagrama de clases. En nuestra propuesta todo enlace es instancia de una asociación, pero no es necesario que todas las asociaciones se representen en algún diagrama: basta con que esas asociaciones estén representadas en el modelo subyacente, aunque no aparezcan en ningún diagrama. Por otra parte, la declaración de asociaciones «`self`» es superflua, ya que todo objeto tiene por principio un enlace «`self`» consigo mismo, y por tanto toda clase tiene por defecto una asociación «`self`» unidireccional consigo misma con multiplicidad 1..1: especificarlas no aportaría ninguna información útil al modelo. En cuanto al resto de enlaces estructurales y contextuales, *sí* es conveniente que la herramienta permita (e incluso exija) especificar las asociaciones estructurales o contextuales correspondientes, del mismo modo que para un objeto se especifica su clase[80]: así se evita, especialmente en el caso de las asociaciones contextuales implícitas, que sus propiedades queden sin especificar o que el modelador pretenda especificar una interacción que resulta incoherente con el resto del modelo. En determinados contextos puede ser relativamente fácil sugerir cuáles sean estas asociaciones. Por ejemplo, si se está desarrollando un diagrama de colaboración que representa la ejecución de una operación de la clase, el contexto indica que el objeto receptor del mensaje tiene, además de las asociaciones estructurales propias de la clase, una asociación contextual con cada uno de los parámetros de la operación y con cada una de sus variables locales.

En cuanto a la correspondencia entre un modelo y una posible implementación, trataremos este tema con más profundidad en el Capítulo 6. Baste decir, por el momento, que las asociaciones estructurales se traducirán de alguna manera en atributos de clase, y las asociaciones contextuales «`parameter`» y «`local`» en argumentos o variables locales de los métodos de clase. Las asociaciones contextuales «`self`» y «`global`» no tienen ningún reflejo en la implementación, porque no es necesario.

[80] En determinadas fases del desarrollo de un proyecto software, especialmente en las iniciales, conviene no especificar la clase de un objeto, y del mismo modo no habrá que exigir que se especifique la asociación de un enlace. Las herramientas CASE pueden habilitar o deshabilitar esta característica según convenga al modelador.

4.6. Propiedades de la navegabilidad

4.6.1. Dependencia

La primera consecuencia de lo expuesto sobre la navegabilidad es que una asociación navegable *posibilita la comunicación*: si la clase `A` tiene una asociación navegable hacia la clase `B`, entonces los objetos de la clase `A` pueden tener conocimiento de los objetos de la clase `B` y pueden enviarles mensajes. La segunda consecuencia de la navegabilidad que vamos a ver ahora es que una asociación navegable *crea una dependencia*: si la clase `A` tiene una asociación navegable hacia la clase `B`, entonces la clase `A` depende de la clase `B`[81]. La navegabilidad significa conocimiento, y conocer significa tanto comunicabilidad como dependencia[82].

En UML, "dependencia" es un término de conveniencia para representar una relación que no sea ni asociación, ni generalización, ni flujo, ni meta-relación (tal como la relación entre un clasificador y una de sus instancias) [UML, p. 2-33]. Esta relación agrupa tipos muy distintos de relaciones, de modo que una definición que las abarque a todas resulta bastante abstracta. En la documentación oficial encontramos varias de estas definiciones de dependencia:

- Una relación semántica (*semantic relationship*) entre dos cosas (*things*) en la que el cambio en una cosa (la independiente) puede afectar a la semántica de la otra cosa (la dependiente) [UG, pp. 23 y 460].
- Una relación de uso (*using relationship*) que determina que el cambio en la especificación de una cosa puede afectar a otra cosa que la utiliza, pero no necesariamente a la inversa [UG, pp. 63 y 137].
- Una relación entre dos elementos en la que el cambio en un elemento (el suministrador - *supplier*) puede afectar o suministrar información necesaria para el otro elemento (el cliente - *client*) [RM, p. 250].

---

[81] El Manual de Referencia afirma esto sólo de forma negativa: "Una asociación sin navegabilidad no crea una dependencia desde la clase origen hacia la clase destino [RM, p. 355].

[82] Los buenos diseños se caracterizan por conseguir navegabilidad sin crear dependencias. Por ejemplo, supongamos que la clase `A` tiene una asociación navegable hacia la clase `B`, y la clase `C` es subclase de la clase `B`. En tiempo de compilación, `A` depende de `B`, pero no depende de `C`. En tiempo de ejecución, podemos tener instancias de `A` enlazadas a instancias de `C`, por tanto podemos navegar desde instancias de `A` hacia instancias de `C`. Hemos conseguido navegabilidad de `A` hacia `C` al nivel de las instancias sin que haya dependencia desde `A` hacia `C` en el nivel de la especificación [Jackson 00]. No obstante, la asociación desde `A` hacia `B` sí que crea una dependencia desde `A` hacia `B`.

- La dependencia determina que la implementación o funcionamiento de uno o más elementos requiere la presencia de otros elementos adicionales [UML, p. 2-33].
- La dependencia especifica que la semántica de un conjunto de elementos del modelo (*model elements*) requiere la presencia de otro conjunto de elementos del modelo. Esto implica que si el origen es modificado de alguna manera, los dependientes probablemente deban ser modificados [UML, p. 2-73].
- Una relación entre dos elementos de modelado (*modeling elements*) en la que un cambio en un elemento de modelado (el elemento independiente) afectará al otro elemento de modelado (el elemento dependiente) [UML, p. B-7]

Queremos resaltar dos ideas de estas definiciones: "los cambios pueden afectar...", "la presencia es requerida...". Para una asociación navegable estas ideas significan: si `A` conoce y usa `B`, entonces *la presencia de `B`* es requerida para que `A` funcione, y *un cambio en `B`* puede ser reconocido por `A`, de modo que puede tener que adaptarse al cambio.

Las dependencias son malas para la reutilización, ya que el elemento dependiente no puede ser reutilizado sin reutilizar también el elemento independiente. En términos del clases, si la clase `A` tiene conocimiento de la clase `B`, entonces es imposible reutilizar la clase `A` sin reutilizar la clase `B` [Stevens 00, p. 78]. No podemos eliminar completamente las dependencias entre elementos, pero deberíamos intentar minimizarlas en un proyecto software. Cuando las asociaciones en un modelo son predominantemente unidireccionales, *la reutilización de pequeñas porciones del modelo resulta más sencilla*: si la clase `A` depende de la clase `B`, entonces no podemos reutilizar `A` sin `B`, pero al menos podemos reutilizar `B` sin `A`[83].

La navegabilidad bidireccional en una asociación induce una dependencia más fuerte que la navegabilidad unidireccional: induce dos dependencias en lugar de una sola. La consecuencia es clara: no deberíamos introducir la navegabilidad bidireccional a menos que sea verdaderamente requerida, o que haya una buena razón para pensar que será requerida en el futuro. A veces es esencial permitir la navegabilidad bidireccional en una asociación, pero estas decisiones deberían ser justificadas una a una, más bien que ser la opción por defecto [Stevens 00, p. 78].

La dependencia inducida por una asociación estructural se detecta inmediatamente por simple inspección: allí donde hay una asociación estructural, hay también una dependencia en la misma dirección en la que la asociación es navegable (mutua, por tanto, si es bidireccional). Las

[83] Pero, atención, un ciclo de asociaciones unidireccionales impide la reutilización de modo tan eficaz como una asociación bidireccional. De modo que es inútil intentar solucionar la dependencia mutua introduciendo dependencias circulares, que son aún peores [Stevens 01].

asociaciones contextuales no presentan mayor dificultad si están expresadas explícitamente. En caso contrario, como veremos a continuación, puede ser imposible detectar las dependencias, lo que demuestra la conveniencia de explicitar las asociaciones contextuales de un modelo. En primer lugar, hay que examinar los argumentos y variables locales de las operaciones para identificar las asociaciones contextuales implícitas «`parameter`» y «`local`»; naturalmente, esto sólo es posible si el modelo incluye la signatura de la operación y sus variables locales; esto último es poco frecuente, de modo que no es fácil identificar una asociación contextual «`local`» implícita. Análogamente, una asociación contextual implícita «`global`» se corresponde con la mención de la variable global en algún lugar del cuerpo de las operaciones de la clase, de modo que tampoco es fácil identificarla en un modelo que no incluya el cuerpo de las operaciones. Una asociación contextual «`self`» no induce ninguna dependencia, a menos que se considere la dependencia trivial de una clase consigo misma. Por último, las asociaciones derivadas no inducen dependencia por sí mismas si se usan de acuerdo con la Ley de Demetria, tal como hemos explicado en el Apartado 4.5.2, transformándolas en asociaciones contextuales «`local`».

4.6.2. Invertibilidad y bidireccionalidad

Vamos a examinar ahora la propiedad de invertibilidad de las asociaciones. En principio podemos asumir que una asociación es invertible cuando es bidireccional[84], es decir, cuando es navegable en ambos sentidos. Así pues, desde nuestro punto de vista, invertibilidad (*invertibility*), bidireccionalidad (*bidirectionality*) y navegabilidad en ambos sentidos (*two-way navigability*) son sinónimos. Una asociación que puede ser navegada (o recorrida) en ambos sentidos implica *conocimiento mutuo*, y por tanto comunicabilidad y dependencia mutuas. Sin embargo, no estamos seguros de que ésta sea la intención de UML, o que UML sea plenamente coherente con el paradigma de orientación a objetos en este tema.

De hecho, el Manual de Referencia es más bien confuso al respecto: "A veces se dice que una asociación es bidireccional (*bidirectional*). Esto significa que la relación lógica funciona en ambos sentidos. Esta afirmación es con frecuencia mal entendida, incluso por algunos metodólogos. No significa que cada clase "conozca" (*know*) a la otra clase, o que, en una implementación, sea posible acceder (*access*) a cada clase desde la otra. Simplemente significa que toda relación lógica tiene una relación inversa, sea o no la inversa fácil de calcular. Para afirmar la capacidad de recorrer una asociación en una dirección pero no en la otra, en tanto que decisión de diseño, se puede usar la marca de navegabilidad" [RM, p. 50].

De acuerdo con este párrafo, parece ser que:

---

[84] Recuérdese que bidireccional no significa simétrica, ver Apartado 4.3.1.

- La direccionalidad lógica no significa conocimiento, ni capacidad de acceder o referenciar otro elemento; direccionalidad lógica no es navegabilidad.
- La bidireccionalidad, o invertibilidad, es una propiedad puramente lógica que se aplica necesariamente a todas las relaciones lógicas, no sólo a algunas de ellas, porque toda relación lógica funciona en ambos sentidos, es decir, tiene una relación inversa aunque no sea fácil de calcular.
- La navegabilidad es una propiedad de la implementación que puede especificarse para algunas asociaciones; la navegabilidad es una cuestión de diseño; definir la navegabilidad de una asociación sólo tiene sentido en el nivel de diseño.
- Invertibilidad no es navegabilidad en ambos sentidos; la primera es un concepto lógico, la segunda es un concepto de implementación.

Hemos intentado encontrar un sentido a la idea de bidireccionalidad, o invertibilidad, tal como es expresada en el Manual de Referencia: de acuerdo con este punto de vista, invertibilidad y bidireccionalidad son sinónimos, pero no tienen nada que ver con la navegabilidad en ambos sentidos. Sin embargo, no estamos satisfechos con el resultado: esta interpretación nos parece artificial. Si direccionalidad no es navegabilidad, entonces, ¿qué es? Más aún, aunque el texto citado del Manual de Referencia separa el concepto de bidireccionalidad-invertibilidad (concepto lógico) del concepto de navegabilidad (concepto de implementación), otros textos no presentan una distinción tan cuidadosa: "a menos que se especifique de otra manera, la navegación de una asociación es bidireccional" [UG, p. 143], "si no se ha decidido la navegabilidad, entonces es bidireccional en el caso general" [RM, p. 355], etc. El Estándar mismo no dice nada acerca de la invertibilidad o bidireccionalidad de las asociaciones. Una interpretación mucho más natural del entrelazamiento de invertibilidad, direccionalidad y navegabilidad nos lleva al núcleo del concepto estático y dinámico de asociación. A partir de este momento, adoptamos de nuevo nuestro punto de vista, en el cual estos tres términos son sinónimos.

Es bien conocido que el concepto de asociación (*association*) en UML, como en sus más cercanos predecesores, *Object Modeling Technique* (OMT) [Rumbaugh 91] y *Object-Relation Model* (ORM) [Rumbaugh 87], deriva del concepto de relación (*relationship*) en el Modelo Entidad/Relación (ERM) [Chen 76]. En el modelado ER la existencia de una relación entre dos entidades es *la expresión de que un cierto predicado se verifica en el mundo real*, de que hay una propiedad que se verifica para el par de objetos enlazados. Almacenamos la tupla `(Juan, Acme)` en la tabla `Trabaja-para` con el fin de registrar el hecho en el mundo real de que "Juan trabaja para Acme".

En el modelado ER las relaciones no tienen dirección[85]. Las relaciones son "neutrales" para las entidades relacionadas, expresan un hecho o predicado que puede ser consultado (*queried*) sólo por alguien desde fuera de la relación misma (el motor de base de datos, por ejemplo), pero no por las mismas entidades relacionadas. Las entidades no tienen comportamiento, son datos puramente pasivos, no ejecutan ninguna operación, no consultan a otras entidades relacionadas. Si "Juan trabaja para Acme", esto es un hecho que no conocen ni `Juan` ni `Acme`. En otras palabras, las entidades no son objetos. Estamos acostumbrados a pensar en asociaciones sin dirección probablemente porque estamos acostumbrados a pensar en términos de *estructuras de datos pasivas que son consultadas desde el exterior*.

En el modelado orientado a objetos este enfoque ya no funciona. El estado de un objeto se define por los valores de sus atributos y por los enlaces que tiene con otros objetos. El estado de un objeto es modificado por su comportamiento, pero el estado también delimita los posibles comportamientos que un objeto puede emprender en un determinado instante de tiempo. En esto consiste, al fin y al cabo, el estado: posibilidades de comportamiento[86]. En este sentido, una instancia de una asociación caracteriza el estado de los objetos enlazados y determina sus posibilidades de comunicación. Un enlace entre dos objetos, como una relación entre dos entidades, expresa un hecho, pero los mismos objetos enlazados son los responsables de informar de la existencia de ese hecho a quienquiera que lo consulte. Una asociación navegable representa una información de estado accesible a una clase [RM, p. 49]. La navegabilidad de una asociación indica, entre otras cosas, *quién tiene la responsabilidad de informar sobre el estado de la asociación* [Fowler 00, p. 55], quién conoce ese estado, y quién puede usar la asociación para comunicarse. En la **Figura 4.1** la asociación `trabaja-para` es navegable sólo desde `Persona` hacia `Empresa`: esto significa que `Juan` es capaz y responsable de decir si trabaja o no para `Acme`; `Acme` no tiene esta posibilidad.

Si el estado de una asociación puede ser consultado y actualizado por ambos extremos, entonces la asociación es bidireccional (*two-way*); si sólo por un extremo, es unidireccional (*one-way*). Según el Estándar, una asociación sin navegabilidad (*no-way*) es normalmente rara o inexistente en la práctica [UML, p. 3-73]. Nosotros lo afirmamos con mayor rotundidad aún: *una asociación no navegable no es aceptable*, porque nadie sería responsable de ella[87].

---

[85] Por supuesto, son asimétricas desde el punto de vista lingüístico, pero ésta es otra cuestión, como ya hemos explicado anteriormente. Ver Apartado 4.3.1.

[86] Como algunos autores lo expresan, las propiedades son una forma indirecta de especificar el comportamiento [Genilloud 00].

[87] Excepto, tal vez, el caso de una clase-asociación, que teniendo su propio comportamiento, podría ser ella misma responsable de mantener su estado. Ver Apartado 4.8.1.

Ahora bien, ¿es verdad que toda relación lógica es bidireccional, invertible? La invertibilidad de una asociación significa, aproximadamente, que funciona en ambos sentidos, que es significativa en ambas direcciones [RM, p. 50]. Cuando consideramos solamente una dirección de la asociación, obtenemos una correspondencia (*mapping*). Según Martin y Odell, una correspondencia asigna a un objeto de un tipo un objeto o conjunto de objetos de otro tipo [Martin 95, p. 42]. Martin y Odell sugieren también que un par de correspondencias asociadas que apuntan en direcciones opuestas son inversas una de la otra (en el sentido de la teoría de conjuntos), y, por tanto, una asociación consiste en dos correspondencias inversas. Otros autores han demostrado que esto es formalmente incorrecto, aunque suficientemente cercano a la verdad, y han tratado de desarrollar un enfoque más riguroso basado en la teoría de las categorías [Graham 97a].

Stevens ha propuesto una distinción pertinente entre "asociación" (*association*) y "relación" (*relation*): una asociación no es simplemente una relación, un conjunto de tuplas; más bien, una asociación determina y es determinada por una relación [Stevens 02] (ver Apartado 3.5.2). Desde nuestro punto de vista, la *invertibilidad de una relación* es una cuestión matemática. Podemos definir así el inverso matemático de una relación `R`: la tupla `(y, x)` está en R^{-1} si y sólo si `(x, y)` está en `R`. Por el contrario, la *invertibilidad de una asociación* es una cuestión computacional: una asociación es invertible si existe un proceso computacional mediante el cual los objetos asociados en un extremo pueden conocer los objetos asociados en el otro extremo a través de la asociación. La existencia de un conjunto de tuplas en la relación `R`, de las cuales podemos inferir las tuplas recíprocas en la relación inversa R^{-1}, no implica que este conocimiento sea compartido por ambos extremos de la asociación. En otras palabras, *la invertibilidad de la relación no implica la invertibilidad de la asociación*.

En contra del Manual de Referencia, aparentemente, afirmamos que *tiene pleno sentido decir que una asociación es significativa sólo en una dirección*, y que esto es una propiedad lógica del modelo, no una propiedad de la implementación (continuaremos con esto en el siguiente Apartado). Por ejemplo, podemos considerar, como es el caso de la **Figura 4.3**, que un objeto `Llave` conoce el objeto (u objetos) `Puerta` que puede abrir, pero no al revés: podemos hacer copias de la llave sin que la puerta se entere. ¿Por qué nos vemos privados de la posibilidad de especificar que una asociación lógica es navegable sólo en una dirección?

Concedemos que en los primeros estadios del análisis no hay gran necesidad de poner dirección a las asociaciones [RM, p. 49], y que la navegabilidad es principalmente una cuestión de diseño [Fowler 00, p. 55] que a menudo no puede ser decidida durante el análisis [Stevens 00, p. 78], pero no estamos de acuerdo con que toda asociación deba ser lógicamente invertible. *La esencia de una asociación*, afirmamos, *es el conocimiento*, y

el conocimiento puede ser unidireccional, no por una cuestión de eficiencia, sino por una cuestión de principio.

4.6.3. Eficiencia

El Manual de Referencia da una definición técnica de la eficiencia de navegación: "Una asociación binaria es eficientemente navegable si el coste medio de obtener el conjunto de objetos asociados es proporcional al número de objetos en el conjunto (no al límite superior de la multiplicidad, que puede ser ilimitado), más una constante fija" [RM, p. 356]. Sin duda la eficiencia es una cuestión que debe tratarse durante la fase de diseño, y en este sentido una definición basada en el coste de recorrido es perfectamente adecuada.

Ahora bien, aunque el Manual de Referencia afirma claramente que la eficiencia de navegación (*navigation efficiency*) no es la propiedad definitoria de la navegabilidad [RM, p. 356], no obstante declara que la navegación habitualmente lleva consigo la connotación de eficiencia de navegación [RM, p. 355]. Esto no sería tan malo si la Guía del Usuario y el Manual de Referencia no sugirieran en algunos lugares que el indicador de navegabilidad no indica navegabilidad *lógica*, sino navegabilidad *eficiente*, y a la inversa, que la navegación contra la marca de navegabilidad no resulta *imposible*, sino tan sólo *ineficiente*:

- "La especificación de una dirección de recorrido (*direction of traversal*) no significa necesariamente que no sea posible para los objetos de un extremo obtener los objetos del otro extremo. Más bien, la navegación es una declaración de la eficiencia del recorrido (*efficiency of traversal*)" [UG, p. 144].
- "Un enlace puede ser usado para la navegación. En otras palabras, un objeto que aparece en una posición de un enlace puede obtener el conjunto de objetos que aparecen en otra posición. Puede enviarles mensajes (lo que se llama "enviar un mensaje a través de una asociación"). Este proceso es eficiente si la asociación tiene la propiedad de navegabilidad en la dirección del destino. El acceso puede ser o no ser factible si la asociación no es navegable, pero será probablemente ineficiente" [RM, p. 325].
- "La falta de navegabilidad no implica que no exista forma de recorrer (*traverse*) la asociación. Si es posible recorrer la asociación en la otra dirección, puede ser factible buscar todas las instancias de la otra clase para encontrar aquellas que conducen al objeto, invirtiendo así la asociación [RM, p. 355].
- "Si una asociación no es navegable en una dirección dada, esto no significa que no pueda ser recorrida en absoluto, sino que el coste del recorrido puede ser significativo (por ejemplo, que requiera buscar en una gran lista" [RM, p. 357].

Aparentemente, pues, la flecha de navegabilidad puede tener dos significados diferentes: posibilidad lógica de navegación, o bien eficiencia de navegación. Esta ambigüedad deja al modelador en una posición difícil, como cualquier otra ambigüedad del lenguaje, porque obstaculiza la comunicación con otros modeladores y desarrolladores. El Manual de Referencia favorece la versión de "eficiencia" (que es defendida por Rumbaugh también en otros lugares [Rumbaugh 96a]), aunque esta noción está ausente en el Estándar, donde la navegabilidad se define como pura posibilidad de recorrido (*traversal*), sin ninguna consideración de eficiencia [UML, p. 2-23]. La posibilidad lógica de navegación es un concepto importante tanto en el análisis como en el diseño, mientras que la eficiencia de la navegación sólo es relevante para el diseño. Por tanto, en nuestra opinión, y contra el Manual de Referencia, *la flecha de navegabilidad no debe usarse nunca para significar navegación eficiente*, especialmente porque esto hace imposible especificar que una asociación no es navegable en absoluto en una dirección. Más bien deberíamos utilizar otra notación para expresar la eficiencia de navegación cuando sea necesario.

Si una asociación es lógicamente no navegable en una dirección, entonces ciertamente no es invertible (en el Apartado anterior ya hemos distinguido entre invertir una asociación e invertir la relación matemática inducida). Esto no significa que sea absolutamente imposible conocer el objeto asociado; significa sólo que es imposible conocerlo *a través de* la asociación. Puede haber otras clases y asociaciones en el modelo que permitan una ruta alternativa que podría ser navegable e incluso eficientemente navegable, pero esto no hace a la asociación invertible en sí misma, ni siquiera ineficientemente invertible.

Consideremos el ejemplo de la **Figura 4.16**. La asociación `Llave-Puerta` no es navegable hacia la clase `Llave`. Esto significa que una puerta no puede saber a través de esta misma asociación qué llaves pueden abrirla o cerrarla: la asociación no es invertible. Afortunadamente, para cada puerta existe una lista de llaves asociadas con la puerta, de modo que una búsqueda en la lista proporciona una ruta (probablemente) ineficiente hacia sus llaves. La asociación `Llave-Puerta` sigue siendo no invertible en sí misma, pero hemos encontrado una ruta alternativa a través de la lista, y podemos "invertirla" indirectamente (advertimos que no nos gusta esta forma de hablar de "inversión indirecta", ya que puede sugerir que se trata de una verdadera inversión, cuando no lo es). Ahora supongamos que no existiera la lista de llaves: no habría ninguna ruta alternativa desde la puerta hacia las llaves, e "invertir" la asociación resultaría completamente imposible, ni directa ni indirectamente. Esta situación no es irrazonable, ya que no podemos imponer que toda implementación imaginable mantenga una lista de los objetos que pertenezcan a una determinada clase.

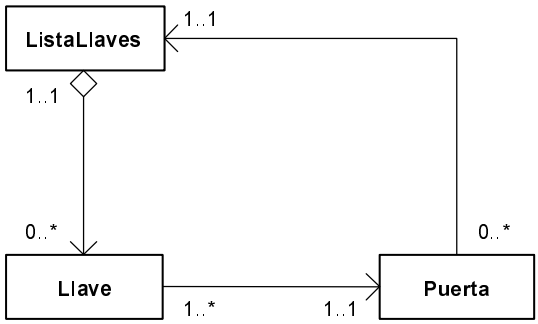

Figura 4.16. Una asociación unidireccional `Llave-Puerta` con una ruta alternativa desde `Puerta` hacia `Llave` a través de otra clase `ListaLlaves`

4.7. Notación de la navegabilidad

El Estándar da las siguientes reglas básicas para la notación de la navegabilidad: "Puede ponerse una flecha en el extremo de la línea para indicar que se soporta la navegación hacia el clasificador unido a la flecha. Pueden ponerse flechas en ninguno, uno, o los dos extremos de la línea. Para ser totalmente explícito, pueden mostrarse las flechas siempre que la navegabilidad esté soportada en una dirección dada. En la práctica, a menudo es conveniente suprimir algunas de las flechas y mostrar sólo las situaciones excepcionales" [UML, p. 3-72].

En la notación de UML hay una regla general: la supresión de algún símbolo notacional no significa la negación de la existencia del elemento no simbolizado[88]. Por ejemplo, la supresión del compartimento de atributos en una clase no significa que la clase no tenga atributos: sólo significa que el modelador no desea mostrarlos en una vista particular. De acuerdo con esta regla, la supresión de la flecha de navegabilidad no indica que el extremo de asociación no sea navegable, sino que solamente deja esta propiedad sin especificar.

Desafortunadamente, esto puede conducir a modelos ambiguos, porque no podemos distinguir entre "no especificada" y "no navegable", aunque los modeladores pueden evitar esta ambigüedad adhiriéndose a un buen estilo de notación. El Estándar propone tres estilos diferentes u "opciones de presentación" [UML, p. 3-73] (ver **Figura 4.17**):

- *Mostrar todas las flechas.* La notación de la navegabilidad está totalmente explícita, y la ausencia de una flecha indica que la navegación no está soportada.

[88] Cuando algún elemento de la notación no se puede omitir, el Estándar lo indica explícitamente. Por ejemplo, el símbolo de agregación o el cualificador de una asociación [UML, pp. 3-72 y 3-77].

- *Suprimir todas las flechas.* La notación de la navegabilidad está totalmente sin especificar, de modo que no se puede inferir nada acerca de la navegación.
- *Mostrar las flechas sólo para las asociaciones unidireccionales.* En esta notación "económica" las flechas de la notación se suprimen para las asociaciones con navegabilidad en ambas direcciones.

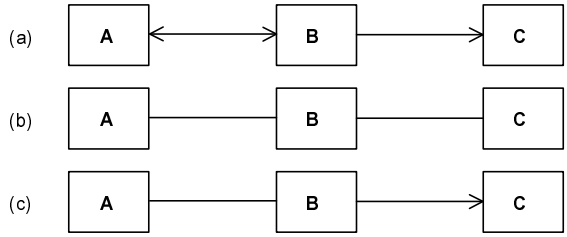

Figura 4.17. Opciones de presentación de la navegabilidad: a) mostrar todas las flechas; b) suprimir todas las flechas; c) mostrar sólo las flechas de asociaciones unidireccionales

Un mal estilo sería aquél en el que la supresión de las flechas es arbitraria: se muestran algunas flechas en asociaciones unidireccionales, otras no; algunas asociaciones bidireccionales se muestran con dos flechas, otras sin ninguna flecha, algunas con una sola flecha. En un mal estilo no hay forma de saber si un extremo de asociación no es navegable.

Tanto la primera como la tercera de las opciones de presentación propuestas por el Estándar sirven para mostrar que un determinado extremo de asociación no es navegable, aunque la tercera opción es menos clara, porque no distingue adecuadamente entre navegabilidad bidireccional y navegabilidad sin decidir. Además, en la primera opción las dependencias inducidas quedan expresadas por las mismas flechas situadas en los extremos de asociación, mientras que en la tercera opción se omiten para las asociaciones bidireccionales. También existe el peligro de que las asociaciones unidireccionales sean consideradas "situaciones excepcionales", lo que desde nuestro punto de vista es erróneo[89]: como ya hemos mostrado, las asociaciones unidireccionales deberían considerarse la opción por defecto, ya que minimizan las dependencias entre clases.

Así pues, encontramos preferible una notación que sea totalmente explícita o totalmente no especificada, de modo que si un diagrama de clases muestra alguna flecha de navegabilidad asumimos que se muestran todas las flechas; si no se muestra ninguna flecha, no estamos especificando la navegabilidad [Stevens 00, p. 78]. En general, el estilo "suprimir todo" sería adecuado para las primeras fases del análisis, mientras que el estilo "mostrar todo" sería mejor para un análisis detallado y para el diseño.

[89] La afirmación de que una asociación es bidireccional por defecto [UG, p. 143; RM, p. 355] refuerza esta tendencia.

Como nota final, deseamos subrayar que *la notación de la navegabilidad debe usarse para expresar navegabilidad lógica, no navegabilidad eficiente*. Si la eficiencia es importante, y sin duda lo es en el diseño, debe emplearse una notación diferente, tal como una restricción, un estereotipo, o un tipo diferente de flecha (una línea con dos puntas de flecha en el mismo extremo, por ejemplo).

4.8. La navegabilidad en las asociaciones más complejas

Una vez que hemos analizado extensamente la cuestión de la navegabilidad en asociaciones binarias simples, vamos a aplicar ahora los conceptos estudiados a otros tipos de asociación más complejos: la clase-asociación, la asociación cualificada y la asociación n-aria. Con frecuencia este tipo de asociaciones sólo se considera desde el punto de vista estructural, como si sólo tuvieran relevancia para el modelado de datos, pero si se trata de verdaderas asociaciones es imprescindible aclarar también sus aspectos dinámicos. Sin duda es un tema bastante descuidado en UML.

4.8.1. Expresiones de navegación

Hemos definido la *navegabilidad* de una asociación binaria como la posibilidad de que un objeto origen designe un objeto destino a través de una asociación, con el fin de manipularlo o acceder a él en una interacción con intercambio de mensajes. Esta designación se lleva a cabo mediante una *expresión de navegación*, que da la posibilidad de obtener una ruta o referencia hacia un objeto que permite manejar el objeto (o conjunto de objetos) destino como un pseudo-atributo del objeto origen.

En una *clase-asociación* (binaria) hay tres clases involucradas: las dos clases asociadas y la clase que las asocia. En tanto que asociación binaria, es válido todo lo que hemos dicho hasta el momento. Si la asociación es bidireccional el acceso es mutuo entre las dos clases, y si es unidireccional sólo está permitido desde la clase origen hacia la clase destino. ¿Cómo es el acceso desde y hacia la clase-asociación? Ante todo, hay que hacer notar que la línea discontinua que une el símbolo de la clase con el símbolo de la asociación no permite ningún adorno adicional [UML, p. 3-78], en particular, no se puede añadir una punta de flecha en ninguno de los extremos para indicar una supuesta navegabilidad hacia la clase-asociación o hacia las clases asociadas. Por tanto, habrá que deducir las posibilidades de acceso a partir de la navegabilidad de la propia clase-asociación en tanto que asociación:

- **Clase-asociación bidireccional.** Parece lógico pensar que es posible acceder a la clase-asociación desde ambas clases asociadas y viceversa. Aunque el Estándar no es explícito en este punto, el capítulo dedicado a OCL proporciona algunas indicaciones sobre cómo serían las expresiones de navegación correspondientes [UML, p. 6-15]. En el ejemplo de la **Figura 4.18** (a) podemos

acceder desde la clase `Persona` a la fecha de su matrimonio con la expresión de navegación `Matrimonio.fecha`, y podemos acceder desde la clase `Matrimonio` a los nombres de los esposos mediante las expresiones `esposo.nombre` y `esposa.nombre`. Nótese que estas últimas son las mismas expresiones que emplearíamos para acceder desde un extremo hacia el otro. Existe, no obstante, una diferencia: la expresión de navegación evaluada desde un extremo resulta, si la multiplicidad fuera mayor que uno, en un conjunto de objetos, mientras que evaluada desde la clase-asociación resulta, por su propia definición, en un objeto único sea cual sea la multiplicidad [UML, p. 6-16]. Así pues, si la multiplicidad fuera mayor que uno, en un caso la expresión `esposo.nombre` produciría el conjunto de nombres de los esposos de una persona, y en el otro caso produciría el nombre del esposo en un matrimonio concreto.

- **Clase-asociación unidireccional.** Este caso no es tan obvio como el anterior, y el capítulo del Estándar dedicado a OCL no da ninguna indicación útil para resolverlo. No hay duda de que desde el extremo origen (no navegable) será posible el acceso tanto hacia el extremo destino (navegable) como hacia la propia clase-asociación, y también parece claro que no será posible ningún acceso desde el extremo destino de la asociación, ya que no tiene conocimiento de la asociación. Desde la clase-asociación se podrá acceder al extremo navegable, pero es más dudoso si el acceso al extremo no navegable está permitido o no, aunque nos parece más coherente pensar en una respuesta negativa. Así pues, en el ejemplo de la **Figura 4.18** (b), podemos acceder desde la clase `Cliente` y desde la clase `Reserva` al número de la mesa reservada con la misma expresión de navegación en ambos casos, `mesaReservada.número`. Ningún otro acceso es posible en este ejemplo. Al igual que en el caso de la asociación bidireccional, la expresión puede resultar en un conjunto de objetos o en un objeto único, según que se evalúe desde el extremo origen o desde la clase-asociación.

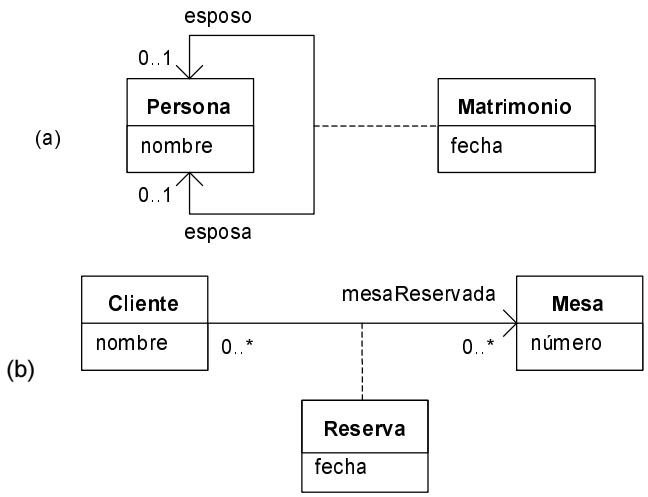

Figura 4.18. Navegabilidad en clases-asociación: a) asociación bidireccional; b) asociación unidireccional

¿Tiene sentido considerar una *clase-asociación no direccional*? En el Apartado 4.6.2 hemos descartado como un sinsentido el caso de una asociación que no sea navegable en ningún sentido (distinto de aquélla que tenga la navegabilidad sin especificar). Si lo que se desea es que la clase-asociación tenga acceso a las dos clases asociadas, pero no viceversa, entonces será mejor sustituirla por una clase con dos asociaciones unidireccionales hacia los dos extremos.

En una *asociación cualificada* hay sólo dos clases involucradas, ya que el cualificador no es propiamente una clase distinta, sino un atributo (o conjunto de atributos) de la propia asociación. La intención de la asociación cualificada es disponer de algún tipo de asociación "indexada" que proporcione un acceso más eficiente a las instancias enlazadas a través de la asociación [Rumbaugh 87] (eficiente en el sentido de que reduce la multiplicidad efectiva de la asociación). La cualificación sólo afecta a una de las dos direcciones de navegación (a menos que el otro extremo también esté cualificado, lo cual no es muy frecuente). Una asociación cualificada puede ser bidireccional o unidireccional, aunque no tiene mucho sentido que el extremo opuesto al cualificador no sea navegable, porque entonces el cualificador sería completamente inútil. En la expresión de navegación se puede incluir opcionalmente un valor de cualificador entre corchetes después del nombre de rol [UML, p. 6-16], de modo que en el ejemplo de la **Figura 4.19** podemos acceder desde la clase `TableroAjedrez` a todas las casillas con la expresión `casilla`, y a una casilla concreta con la expresión `casilla[5, 2]`, por ejemplo[90]; por el contrario, desde la clase

[90] No está permitido especificar parcialmente el valor del cualificador: o no se especifica el valor de ninguno de los atributos, o se especifican todos [UML, p. 6-17]. No está muy claro cuál es el sentido de esta restricción que OCL impone, ya que sería útil

`Casilla` sólo podemos acceder al tablero con la expresión `tableroAjedrez`.

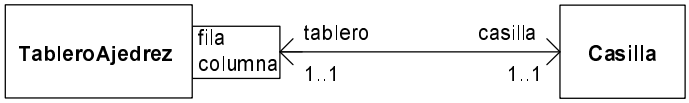

Figura 4.19. Navegabilidad en asociación cualificada

En una *asociación n-aria* el problema se torna conceptualmente mucho más complejo. Según el Manual de Referencia, el concepto de navegabilidad no se aplica a asociaciones n-arias [RM, p. 354]. El Estándar tampoco menciona la navegabilidad en las asociaciones n-arias, aunque no la prohíbe explícitamente. Solamente se dice que en cada extremo pueden mostrarse las mismas propiedades que en una asociación binaria, excepto agregación y cualificación [UML, p. 3-79]. El capítulo dedicado a OCL no dice cómo serían las expresiones de navegación en una asociación n-aria.

Sin embargo, no parece que se pueda renunciar a este concepto, ya que si una asociación n-aria no es navegable, ¿cuál es su utilidad? Si han de tener algún papel en un modelo orientado a objetos, no pueden quedar relegadas al puro modelado estático de datos. Por tanto, en lugar de rehuir la cuestión como hacen el Manual de Referencia y el Estándar, intentaremos encontrar algún sentido a su navegabilidad.

Supongamos, en primer lugar, que es posible especificar la navegabilidad de cualquiera de los extremos de una asociación n-aria. Esto permitiría construir expresiones de navegación desde los otros extremos. La **Figura 4.20** muestra la asociación `trabaja-en-usando` con dos extremos navegables que permiten construir las expresiones de navegación `proyecto` y `habilidad` desde la clase `Empleado`, la expresión `habilidad` desde la clase `Proyecto`, y la expresión `proyecto` desde la clase `Habilidad`. En cambio, la clase `Empleado` no es accesible desde ninguna de las otras dos. Observemos que con esta notación no es posible especificar, por ejemplo, que `Proyecto` sea accesible desde `Empleado`, pero no desde `Habilidad`, y viceversa.

---

poder referirse desde el tablero de ajedrez a todas las casillas que están en una fila o columna determinadas.

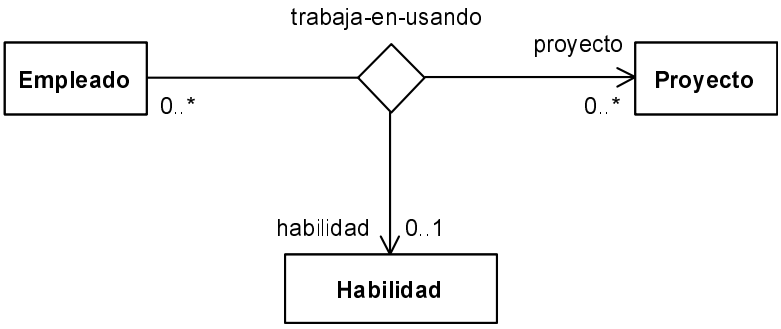

Figura 4.20. La asociación ternaria `trabaja-en-usando`, con dos extremos navegables

Nótese también que, por la propia definición de multiplicidad n-aria, el resultado de la expresión `habilidad` no será un único objeto (o ninguno), a pesar de la multiplicidad 0..1, sino más bien un conjunto de objetos. Se obtendría un único objeto para una combinación de instancias de `Empleado` y `Proyecto`, pero para una instancia sola de `Empleado`, o una instancia sola de `Proyecto`, en general puede haber varias instancias de `Habilidad` asociadas[91]. Para aprovechar las ventajas de la multiplicidad n-aria sería conveniente disponer de algún tipo de expresión que combine instancias de varias clases para acceder a otra. Podemos emplear una notación semejante a la que existe para asociaciones cualificadas, poniendo entre corchetes el valor de una instancia de cada una de las otras clases que intervienen en la asociación después del nombre de rol de la clase destino. En la **Tabla 4.3** se reproduce el ejemplo empleado en el Capítulo 3, de modo que, cuando se evalúan desde la instancia `Alberto`, la expresión `habilidad[Cocina]` da como resultado `{Soldadura}` y la expresión `proyecto[Soldadura]` da como resultado `{Cocina, Laboratorio}`.

— trabaja en — usando —		
Empleado	**Proyecto**	**Habilidad**
Alberto	Cocina	Soldadura
Alberto	Laboratorio	Soldadura
Benito	Cocina	Carpintería
Benito	Garaje	Carpintería
Clara	Cocina	Pintura

Tabla 4.3. Un posible conjunto de instancias de la asociación ternaria `trabaja-en-usando`

Una segunda forma de navegabilidad n-aria que podemos considerar en la **Figura 4.21** es la navegabilidad hacia la asociación misma, que daría la posibilidad de obtener subtuplas de instancias de las otras clases. Así, la

[91] La definición de multiplicidad n-aria y las restricciones de co-ocurrencia han sido tratadas extensamente en el Capítulo 3.

expresión `trabaja-en-usando` daría como resultado `{(Cocina, Carpintería), (Garaje, Carpintería)}` al ser evaluada desde la instancia `Benito`.

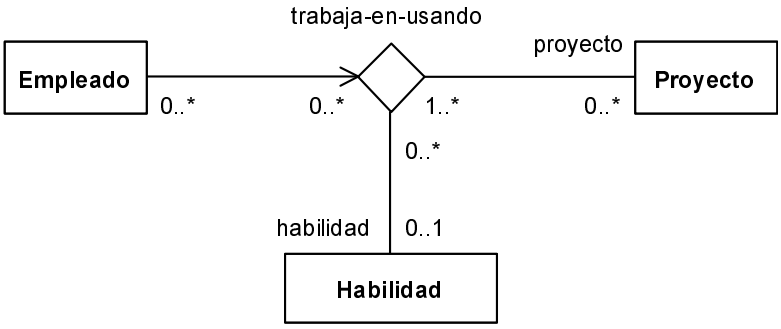

Figura 4.21. La asociación ternaria `trabaja-en-usando`, con la navegabilidad expresada hacia la asociación misma. Se han representado también los valores de multiplicidad externa e interna

No obstante, la utilidad de esta segunda forma de navegabilidad es más limitada, como veremos a continuación.

4.8.2. Envío de mensajes

Una vez establecido cómo se forman las expresiones de navegación en clases-asociación, asociaciones cualificadas y asociaciones n-arias, el envío de mensajes no ofrece mayor problema que para las asociaciones binarias simples. Si es posible establecer una ruta navegable hacia el objeto receptor, y éste dispone de operaciones visibles, entonces el objeto emisor puede enviarle el mensaje correspondiente. Solamente queremos hacer notar que, si se permitieran enlaces "repetidos" en una asociación, tal como hemos propuesto en la Sección 3.5, entonces el resultado de una expresión de navegación podría incluir más de una vez un mismo objeto, con lo que recibiría varias veces el mensaje enviado a ese conjunto de destinatarios. Esta idea no tiene en sí nada de extraño, una vez aceptada la posibilidad de que una asociación contenga enlaces repetidos[92].

Las asociaciones n-arias presentan algunas peculiaridades al respecto. Aunque tenga sentido hablar de enlaces n-arios que representan un predicado que se cumple entre los objetos enlazados, y a la vez una posibilidad de comunicación entre ellos, la comunicación en sí misma es un fenómeno intrínsecamente binario. En el metamodelo de UML esto se refleja en que una instancia de mensaje, o estímulo (*stimulus*), tiene exactamente un emisor y un receptor (ver **Figura 4.4**). Esto impide, por una

[92] No obstante, en el metamodelo el resultado de una expresión de navegación es de tipo `ObjectSetExpression`, con lo que al evaluar la expresión se deberían eliminar las repeticiones, aunque estuvieran permitidas en la misma asociación. Para permitir destinatarios repetidos sería necesario definir un nuevo tipo `ObjectBagExpression` o similar.

parte, la emisión conjunta de un mensaje por parte de dos o más objetos, y, por otra parte, la recepción conjunta de mensajes[93].

Algunos autores han analizado el concepto de "acción conjunta" (*joint action*) [Genilloud 98] o el concepto de "contrato de cooperación" (*cooperation contract*) [Schrefl 91, Schrefl 96] para intentar explicar, dentro del paradigma de la orientación a objetos, pero ampliando el esquema tradicional de "paso de mensajes", la noción de un mensaje que es recibido conjuntamente por varios objetos, y cuyo comportamiento cooperativo da respuesta al mensaje. Esta idea es interesante para modelar un comportamiento que no pertenezca exclusivamente a ninguno de los objetos receptores del mensaje, y es análoga en cierto modo a la clase-asociación, que modela datos y operaciones que no son propiamente de los extremos, sino de la asociación en cuanto tal[94]. Desgraciadamente, la noción de "mensaje conjunto" no tiene cabida en UML, de modo que no la vamos a analizar con más detalle.

Volviendo, pues, al concepto de mensaje que UML contempla, es decir, aquél que tiene exactamente un emisor y un receptor, la representación de un mensaje "binario" enviado a través de un enlace n-ario en un diagrama de colaboración es algo problemática. En efecto, un enlace n-ario se representa como un rombo unido a cada una de las instancias participantes [UML, p. 3-85]. En un enlace binario la flecha del mensaje se representa junto al enlace y en posición paralela, pero es indiferente que esté situada en uno u otro lado del enlace. Para expresar gráficamente quiénes son el emisor y el receptor, la cola de la flecha debe estar orientada hacia el emisor, y la cabeza hacia el receptor. ¿Cómo conseguir lo mismo en un enlace n-ario? Si se trata de un enlace ternario, podemos dividir la región circundante al enlace en tres subregiones, de modo que por la situación de la flecha y su orientación podamos inferir el objeto emisor y el receptor, como en el ejemplo de la **Figura 4.22**. Sin embargo, si se trata de un enlace de grado superior (cuaternario, etc.), no disponemos de subregiones suficientes para expresar gráficamente la misma información. Una posible solución sería emplear flechas más alargadas de lo habitual, que puedan "saltar" por encima del enlace si es necesario, de modo que la cola y la cabeza indiquen de forma no ambigua, por cercanía, el emisor y el receptor del mensaje.

---

[93] El concepto de "difusión" (*broadcasting*) es distinto y legítimo en UML, ya que no se trata de enviar un único mensaje a un conjunto de objetos, sino un mensaje a cada uno de los objetos de un conjunto.

[94] Una diferencia importante con la clase-asociación es que, según estos autores, los objetos receptores del mensaje conjunto no tienen necesariamente que estar enlazados.

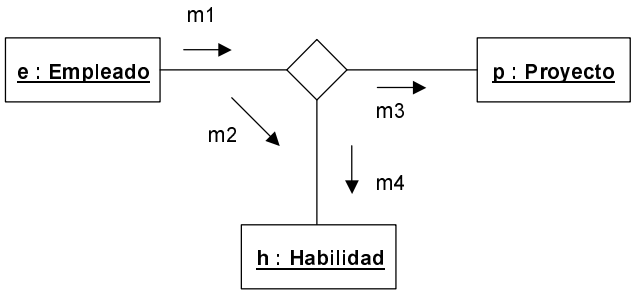

Figura 4.22. Mensajes binarios enviados a través de un enlace n-ario en un diagrama de colaboración. De la situación y orientación de cada flecha se infieren los siguientes emisores y receptores: `m1(e→p)`, `m2(e→h)`, `m3(h→p)`, `m4(p→h)`

La navegabilidad y el envío de mensajes a través de una clase-asociación n-aria se resolvería aplicando lo dicho para clases-asociación y asociaciones n-arias. Por último, las asociaciones contextuales son, en principio, binarias y unidireccionales; en el caso del estereotipo «`local`» cabría la posibilidad de emplear asociaciones más complejas, aunque no es probable que la complejidad añadida compense su utilidad[95].

[95] Según una reciente propuesta, la invocación de una operación *es* una instancia de una asociación dinámica n-aria establecida entre el objeto emisor, el objeto receptor y los argumentos del mensaje [Steimann 02]. Se trata de una concepción muy novedosa, incluso "revolucionaria", de las asociaciones estáticas y dinámicas, que no podemos detenernos a examinar con más profundidad.

5. La visibilidad de las asociaciones

5.1. Introducción

El concepto de visibilidad en UML indica en general si un elemento es visible fuera del elemento que lo contiene. En UML hay cuatro tipos de visibilidad:

Primero, la visibilidad que una propiedad (*feature*), es decir, un atributo o una operación, tiene desde fuera del *clasificador propietario*; o, análogamente, un extremo de asociación desde fuera del clasificador conectado al extremo opuesto. Esta visibilidad se representa en el metamodelo mediante el meta-atributo `visibility` de las metaclases `Feature` [UML, p. 2-37] y `AssociationEnd` [UML, p. 2-23].

Segundo, la visibilidad que un elemento tiene desde fuera del *espacio de nombres propietario*. Un espacio de nombres (*namespace*) es una parte de un modelo que contiene un conjunto de elementos designados por un nombre único dentro del espacio de nombres; cada elemento de un modelo es poseído como máximo por un espacio de nombres [UML, p. 2-44]. El caso más típico de espacio de nombres es el paquete (*package*)[96]. La relación entre un espacio de nombres y los elementos contenidos se representa en el metamodelo mediante la metaclase `ElementOwnership` (en realidad es una clase-asociación), que contiene el meta-atributo `visibility` [UML, p. 2-34].

Tercero, la visibilidad que un elemento tiene desde fuera de un *paquete que lo ha importado*. Esta visibilidad puede ser distinta que la visibilidad original en el paquete propietario. La relación entre un paquete y los elementos importados se representa en el metamodelo mediante la metaclase `ElementImport`, que contiene el meta-atributo `visibility` [UML, p. 2-189].

Cuarto, la visibilidad que un elemento tiene desde fuera de un *componente en el que reside*. La relación entre un componente y los elementos que residen en él se representa en el metamodelo mediante la metaclase `ElementResidence`, que contiene el meta-atributo `visibility` [UML, p. 2-35].

---

[96] Los otros tipos de espacio de nombres son el clasificador y la colaboración. Un clasificador es además un espacio de nombres virtual para sus propiedades [UML, p. 2-43].

En este Capítulo nos vamos a centrar en el primer tipo, la visibilidad de atributos, operaciones y extremos de asociación, por la estrecha relación que tiene con el principio de *encapsulamiento* o *separación de interfaz e implementación*, es decir, la norma de programar (o diseñar) teniendo en cuenta sólo la interfaz de una clase, no su implementación [Gamma 94, Steimann 01]. La visibilidad combinada de atributos, operaciones y asociaciones proporciona una primera forma de separar la *interfaz pública* del clasificador de su *implementación privada*. En el Capítulo anterior hemos distinguido entre asociaciones estructurales y asociaciones contextuales[97]. En ambos casos una clase asociada con otra normalmente no necesita acceder a toda su funcionalidad sino sólo a una pequeña parte de ella, de modo que conviene *restringir el acceso* a las propiedades verdaderamente requeridas por la clase cliente y así lograr un *desacoplamiento* más efectivo de las clases participantes en una asociación[98]. Es decir, conviene especificar *una interfaz distinta para cada asociación*, para lo cual resulta insuficiente la visibilidad de las propiedades, ya que `public` y `package` no discriminan entre las distintas asociaciones a las que está conectado un clasificador. En UML hay dos mecanismos (especificadores de interfaz e interfaces propiamente dichas) que permiten abordar este problema de forma más adecuada. Cuando las interfaces se usan en los extremos de la asociación para desacoplar las clases participantes, resaltando un uso o aspecto particular de una clase, es decir, como *especificaciones parciales* de las clases participantes, algunos autores las denominan *roles* [Steimann 01, D'Souza 99, Firesmith 98].

Por tanto, y siguiendo a Steimann, propondremos una *nueva definición de asociación*, de modo que la asociación ya no se establece entre clasificadores sino entre interfaces, o roles. No obstante, la definición actual de interfaz en UML como especificación de un conjunto de operaciones sin estructura alguna no permite usarlas en asociaciones bidireccionales. Por ello también será necesario *redefinir el concepto de interfaz*, de modo que incluya también atributos y asociaciones navegables. Estos cambios conceptuales irán acompañados de *modificaciones en la notación*, que permitan expresar las propiedades de una asociación con distintos niveles de complejidad, de acuerdo con las necesidades del modelador.

El resto de este Capítulo está organizado como se describe a continuación. La Sección 5.2 está dedicada a analizar la definición de

---

[97] Recordemos que las hemos definido así: una *asociación estructural* especifica una relación entre clasificadores que es definida en la propia estructura estática de los clasificadores asociados; una *asociación contextual* especifica una relación entre clasificadores que es válida sólo en determinados contextos de los clasificadores asociados, por ejemplo, dentro de una invocación de operación.

[98] Insistimos una vez más en que, además de la *interfaz visible*, será necesaria una *ruta navegable* desde la clase origen hacia la clase destino, es decir, una asociación estructural o contextual. Ver la Sección 4.4 para un tratamiento extenso y detallado de la cuestión.

visibilidad de atributos, operaciones y extremos de asociación. La Sección 5.3 profundiza en otras formas de especificar la interfaz de una asociación, como son el especificador de interfaz y la interfaz propiamente dicha. Finalmente, la Sección 5.4 presenta una propuesta concreta para solucionar los problemas planteados.

5.2. Definición de visibilidad

5.2.1. La visibilidad de atributos y operaciones

Atributos y operaciones comparten la misma definición de visibilidad, ya que en el metamodelo ambos son subtipos del concepto genérico "propiedad" (*feature*) (ver **Figura 2.18**)[99]. Una propiedad está encapsulada dentro de un clasificador, y se define como una característica de la estructura o del comportamiento (*behavioral or structural characteristic*) de una instancia del clasificador, o del clasificador mismo [UML, p. 2-36][100].

Según el Estándar, la visibilidad de una propiedad especifica si ésta "puede ser usada por otros clasificadores", y da cuatro posibilidades (se muestra entre paréntesis el símbolo utilizado) [UML, p. 2-37]:

- `(+)public`: cualquier clasificador externo (*outside classifier*) con visibilidad hacia el clasificador puede usar la propiedad.
- `(#)protected`: cualquier descendiente del clasificador puede usar la propiedad.
- `(-)private`: sólo el clasificador mismo puede usar la propiedad.
- `(~)package`: cualquier clasificador declarado en el mismo paquete (o subpaquete anidado, hasta cualquier nivel) que el poseedor de la propiedad puede usar la propiedad.

Podemos considerar que los dos tipos básicos de visibilidad son `public` y `private`, y su utilidad está en permitir una primera separación entre la *interfaz pública* del clasificador y su *implementación privada*. Suele recomendarse que, para respetar el principio de encapsulamiento, todos los atributos sean declarados privados, y las operaciones sean declaradas privadas o públicas según se desee que estén o no disponibles en la interfaz del clasificador. Así, *un objeto puede recibir mensajes que correspondan a operaciones públicas, pero no privadas*; éstas últimas sólo estarían

---

[99] En el metamodelo `Reception` es también subtipo de `BehavioralFeature`, junto con `Operation`. Un mensaje ordinariamente es una invocación de operación o el envío de una señal, que se corresponden respectivamente con operaciones y recepciones. Ver Apartado 4.4.1.

[100] Para distinguir ambos casos se emplea el meta-atributo `ownerScope`, que especifica si la propiedad es distinta para cada instancia del clasificador, o común para todas ellas. Éste último caso, cuando la propiedad es del clasificador mismo, se expresa en la notación subrayando el nombre de la propiedad, que equivale al uso de la palabra clave `static` en lenguajes como Java o `shared` en el entorno .NET.

disponibles para el propio objeto, como parte de su funcionamiento interno. Por supuesto, además de la visibilidad de la operación también es necesario que el objeto emisor tenga conocimiento del objeto receptor, es decir, que exista una ruta navegable o enlace de comunicación entre ellos, como ya hemos tratado extensamente en el Capítulo 4. No obstante, en el caso de que los objetos emisor y receptor pertenezcan a la misma clase, es decir, que estén conectados por una asociación reflexiva, el funcionamiento puede ser distinto del esperado, ya que ambos pueden ver las operaciones privadas, y por tanto pueden enviarse mensajes que no correspondan a la interfaz pública de la clase. Nótese que en UML la visibilidad de los atributos y operaciones es una característica especificada *para los clasificadores*, no para las instancias, es decir, la visibilidad afecta a las relaciones entre clases, no entre objetos[101].

La visibilidad `protected` puede considerarse como una *variante de la visibilidad privada*. Las propiedades protegidas son visibles en los descendientes, es decir, las subclases; las propiedades privadas, en cambio, están restringidas a la clase que las define, de forma que se consigue reducir la dependencia desde la subclase hacia la superclase, ya que no depende de sus propiedades privadas; de esta forma una clase puede esconder detalles de implementación a sus descendientes. Como en el caso de la visibilidad privada, si los objetos emisor y receptor pertenecen a la misma clase (están conectados por una asociación reflexiva), pueden enviarse mensajes que correspondan a operaciones protegidas, es decir, que no estén en la interfaz pública de la clase. En cambio, no pueden enviarse mensajes que correspondan a operaciones privadas de la superclase. Esto puede dar lugar a situaciones un tanto extrañas, aunque propiamente no cabe calificarlas de incorrectas. Consideremos el ejemplo de la **Figura 5.1**, en el que la clase `Rectángulo` define la operación pública `probar` y es subclase de `Polígono`, que a su vez tiene las operaciones `mostrar` y `ocultar`, protegida y privada respectivamente; `Polígono` tiene además una asociación reflexiva unidireccional con el nombre de rol `siguiente` en el extremo navegable (de la visibilidad de los extremos de asociación trataremos posteriormente, por el momento consideramos que es pública y por tanto es visible en la subclase).

---

[101] Para complicar más las cosas, en este tema puede haber variaciones semánticas entre distintos lenguajes de programación, según tengan un carácter más bien "orientado a objetos" u "orientado a clases", como es el caso de UML. En C++ y Java, por ejemplo, que también son "orientados a clases", un objeto puede invocar una operación privada (o protegida) de otro objeto de la misma clase. En el entorno .NET ocurre lo mismo [Barwell 02]. En cambio en Smalltalk y Eiffel, que son más propiamente "orientados a objetos", las propiedades privadas de un objeto sólo son accesibles a él mismo [Mylopoulos 02, Switzer 93].

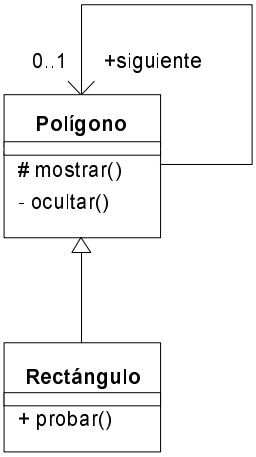

Figura 5.1. Visibilidad privada y protegida de las operaciones

La **Figura 5.2** ilustra esta situación mediante dos diagramas de colaboración. *En el contexto de la subclase* (por ejemplo, en el cuerpo de la operación `probar`, primer diagrama), un objeto `r1:Rectángulo` puede enviar el mensaje `siguiente.mostrar` a otro objeto `r2` de la misma clase que esté enlazado con él, pero no puede enviarle el mensaje `siguiente.ocultar`, porque esta operación no es visible. En cambio, el mismo objeto `r1`, *en el contexto de la superclase* (por ejemplo, en el cuerpo de la operación `mostrar`, segundo diagrama), puede enviarle ambos mensajes a `r2`.

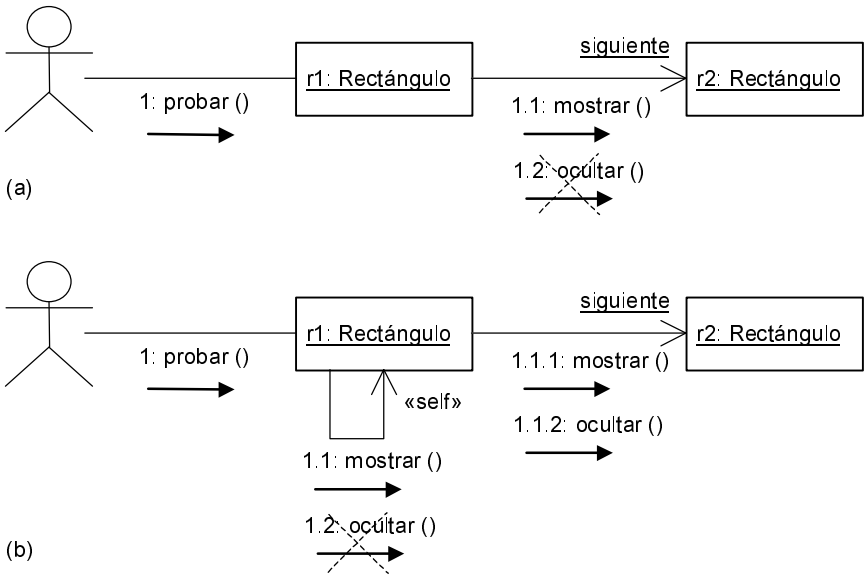

Figura 5.2. El envío de mensajes que corresponden a operaciones protegidas o privadas está permitido o prohibido dependiendo del contexto (los envíos prohibidos se indican mediante un aspa). Los números de secuencia indican el orden de activación de las operaciones y su nivel de anidamiento, es decir, el contexto en el que son invocadas

Esto es consecuencia de que, como ya hemos comentado más arriba, en UML la visibilidad de atributos y operaciones es una característica especificada para las clases, no para los objetos. La visibilidad no especifica si un objeto puede ver otro objeto, sino más bien si una propiedad de una clase puede ver otra propiedad de la misma o de otra clase; por tanto, el cuerpo de una operación puede hacer referencia a otras operaciones privadas de la misma clase. En la **Figura 5.2** (a), si el objeto `r1` recibe un mensaje que invoca la operación `probar` (estamos en el contexto de la subclase), esta operación puede invocar a su vez la operación `siguiente.mostrar`, pero no puede invocar `siguiente.ocultar`. Supongamos ahora, ver **Figura 5.2** (b), que la operación `probar` invoca `self.mostrar`, y que `mostrar` invoca a su vez (ya en el contexto de la superclase) `siguiente.ocultar`; entonces `r1` está enviando de hecho el mensaje `ocultar` a `r2`, por medio de la operación `mostrar` invocada en él mismo. Una última observación: nótese que, en el contexto de la subclase, `r1` no puede invocar `self.ocultar`, como se indica en la **Figura 5.2** (b): es una operación privada que no le pertenece en cuanto objeto, sino que pertenece a la clase que la define. Una vez más: la visibilidad no es una característica de las instancias, sino de las especificaciones, y el hecho de que una operación sea privada no significa sin más que el objeto pueda invocarla sobre sí mismo.

De modo semejante a la visibilidad protegida y privada, podemos considerar que la visibilidad `package`, que ha sido añadida en la versión 1.4 del Estándar (y por tanto no se encuentra en el Manual de Referencia ni en la Guía del Usuario, que se corresponden aproximadamente con la versión 1.3), es una *variante de la visibilidad pública*: las propiedades de paquete son "públicas" en el ámbito de un paquete, es decir, son visibles para cualquier otro clasificador del mismo paquete[102], y por eso podemos

[102] El *concepto de paquete* ya existía en versiones anteriores de UML, lo que ha sido añadido en la versión 1.4 es la *visibilidad de paquete*. Un paquete es una estructura de agrupamiento de otros elementos (incluso otros paquetes), y se representa como un rectángulo con una pestaña, el símbolo habitual de "carpeta". El nombre del paquete puede estar en la pestaña o dentro del paquete, si éste se presenta vacío. El contenido de un paquete puede mostrarse dentro del paquete, o unido al paquete mediante una línea con el símbolo '⊕' (*anchor icon*) en el extremo conectado al paquete. Los elementos contenidos en el paquete pueden tener los cuatro tipos de visibilidad, aunque en realidad no tienen sentido más que `public` y `package`: en efecto, un elemento `private` o `protected` no sería visible para otros elementos del paquete, por tanto el elemento no tendría ninguna utilidad para los demás, por lo que no tiene sentido definirlo con esa visibilidad. Es posible mostrar relaciones (generalizaciones, dependencias, asociaciones, etc.) con elementos públicos de otros paquetes, así como relaciones entre paquetes. Un elemento externo referenciado puede representarse fuera del paquete, o dentro del mismo mediante un alias que consiste en el nombre del elemento, precedido del nombre del paquete propietario, separados por el símbolo '::' [UML, pp. 3-16 a 3-18 y 3-62]. No obstante, el propio Estándar representa los elementos referenciados añadiendo la cláusula '(from *package-*

llamarlas "semipúblicas". Un paquete constituye una barrera de visibilidad "hacia dentro", pero no "hacia fuera", de modo que los clasificadores del paquete `B`, anidado dentro del paquete `A`, pueden ver los mismos elementos que los clasificadores del paquete `A`, tanto `public` como `package`; en cambio, los clasificadores de `A` no pueden ver los elementos de `B` que tengan visibilidad `package`, sólo los que sean `public`; y así hasta cualquier nivel de anidamiento. Al hablar de "subpaquetes" es importante no confundir los términos "anidado" (*nested*) con "descendiente" (*descendant*), ya que son conceptos completamente distintos. Seguramente sería mejor reservar el término "subpaquete" para referirse a la especialización de paquetes, por analogía con "subclase", y en este sentido puede decirse que el Estándar es poco riguroso en el uso de estos términos (véase la definición de visibilidad de paquete más arriba, donde se habla de "subpaquete anidado").

Es fácil ver que `package` es más restrictivo que `public`, así como `private` es más restrictivo que `protected`. En cambio, de acuerdo con las definiciones del Estándar, no puede decirse que la visibilidad protegida sea más restrictiva que la de paquete, ni viceversa. En efecto, consideremos el ejemplo de la **Figura 5.3**, donde las clases `Polígono` y `Rectángulo` se encuentran en paquetes distintos `A` y `B`, y las operaciones `mostrar` y `mover` tienen ahora las visibilidades protegida y de paquete respectivamente. La operación `mostrar` es visible dentro de `Rectángulo`, porque éste es un descendiente de `Polígono`, pero `mover` no es visible porque las clases están en paquetes distintos. Por tanto, en este caso `package` es más restrictiva que `protected`. Evidentemente, otras clases que estuvieran dentro del paquete `A` y que no fueran descendientes de `Polígono`, no tendrían acceso a la operación `mostrar`, pero sí a la operación `mover`, de modo que para ellas `protected` sería más restrictiva que `package`.

---

name)' bajo el nombre del elemento, que es la notación no estándar empleada por la herramienta comercial *Rational Rose*.

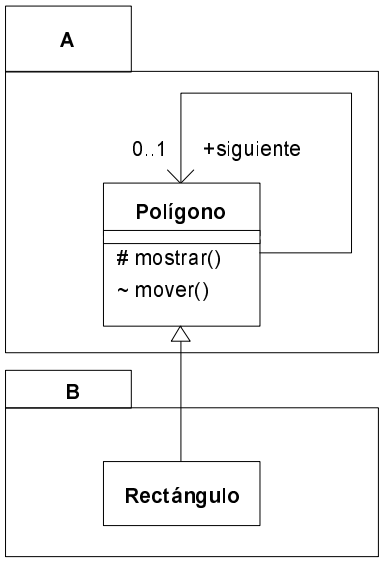

Figura 5.3. Visibilidad protegida y de paquete de las operaciones

En consecuencia, no es adecuado hablar de "niveles de visibilidad" (*visibility levels*), como se hace frecuentemente, dando a entender que se trata de cuatro niveles progresivamente más restrictivos[103]. En realidad, no podemos establecer un orden estricto entre los cuatro tipos de visibilidad, ya que se establecen simultáneamente sobre dos dimensiones distintas: la dimensión de *anidamiento* o encapsulamiento (`public` ⊂ `package` ⊂ `private`) y la dimensión de *especialización* (`protected` ⊂ `private`). La **Figura 5.4** expresa gráficamente estas ideas.

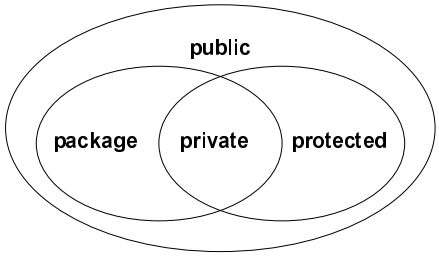

Figura 5.4. Representación gráfica de los cuatro tipos de visibilidad en UML

El Estándar reconoce que todas las formas de visibilidad no pública son de hecho dependientes de la implementación [UML, p. 3-42]. En Java, por ejemplo, donde también hay cuatro tipos de visibilidad (denominadas *access control levels*), el control de acceso `protected` significa la combinación de las visibilidades `protected` y `package` de UML, es decir, el elemento protegido es accesible tanto a los descendientes como a

[103] El Estándar usa esta expresión una vez [UML, p. 3-17].

otros elementos del mismo paquete. En cambio, el control de acceso por defecto, al que no corresponde ninguna palabra clave, aunque se conoce habitualmente como *friendly* [Gosling 96, Arnold 00, Lemay 00], equivale a la visibilidad `package` en UML. Por tanto en Java sí hay una jerarquía de niveles de visibilidad en sentido estricto (*public, protected, friendly, private*), pero al precio de no disponer de una verdadera visibilidad protegida como en UML. En el entorno .NET, por ver otro planteamiento más moderno, existen cinco tipos de visibilidad (denominadas *scope options*): `Private`, `Friend`, `Protected`, `Protected-Friend` y `Public` [Barwell 02]. `Friend` equivale a `package` en UML, `Protected` equivale a `protected` en UML, y `Protected-Friend` es la combinación de `protected` y `package` (equivale por tanto a `protected` de Java). Con esto se salvan en cierto modo los inconvenientes de Java, aunque los cinco tipos tampoco forman una jerarquía estricta. La **Tabla 5.1** resume esta comparación de los tipos de visibilidad en UML, Java y el entorno .NET.

<table>
<tr><th colspan="2">UML</th><th colspan="2">Java</th><th colspan="2">.NET</th></tr>
<tr><td colspan="2">Public</td><td colspan="2">Public</td><td colspan="2">Public</td></tr>
<tr><td rowspan="2">Package</td><td rowspan="2">Protected</td><td colspan="2">Protected</td><td colspan="2">Protected-Friend</td></tr>
<tr><td>Friendly</td><td></td><td>Friend</td><td>Protected</td></tr>
<tr><td colspan="2">Private</td><td colspan="2">Private</td><td colspan="2">Private</td></tr>
</table>

Tabla 5.1. Comparación de los tipos de visibilidad en UML, Java y el entorno .NET

Una observación final antes de concluir este Apartado. Suele recomendarse, y muchas herramientas CASE funcionan con este criterio, que la opción por defecto para la visibilidad de los atributos sea `private`, y `public` para las operaciones[104]. De esta forma los atributos quedarían protegidos de un acceso indebido que no respetase los invariantes[105] de la clase en cuestión, y su manipulación correcta quedaría confiada a las operaciones. Pero, en realidad, para preservar el principio de encapsulamiento no es *necesario* que los atributos sean privados, ni es *suficiente* que el acceso a ellos pase por el control de operaciones públicas. Pensemos en un atributo para el que se proporcionan operaciones que pueden consultar o actualizar su valor sin comprobar ninguna restricción. Si este acceso se considera correcto en un caso concreto, entonces lo mismo da consultar o actualizar el valor del atributo directamente que a través de las operaciones, por tanto las operaciones no son necesarias; y, si no es correcto, entonces las operaciones deben comprobar las restricciones pertinentes, por tanto el mero acceso a través de operaciones no es

[104] Otra posibilidad, quizás más razonable, sería `protected` para atributos y `package` para operaciones.

[105] Un "invariante" es la declaración de una propiedad que debe cumplirse en el "estado estable" de un sistema [Kilov 94].

suficiente, si estas operaciones no llevan a cabo esa comprobación. La clave no está en la visibilidad de atributos y operaciones, sino en la *conservación de los invariantes*; la visibilidad no es más que un instrumento para facilitar esto último, pero de por sí no lo garantiza si no se utiliza adecuadamente.

5.2.2. La visibilidad de los extremos de asociación

En UML la visibilidad no se especifica propiamente para las asociaciones como tales, sino para cada uno de sus extremos, referenciados por sus respectivos nombres de rol, que representan "pseudo-atributos" del clasificador en el otro extremo [UML, p. 2-23]. Según el Manual de Referencia, el uso de un extremo de asociación sólo tiene sentido cuando la asociación es navegable en esa dirección: "*Si la navegabilidad es verdadera*, entonces la asociación define un pseudo-atributo de la clase que está en el extremo opuesto al nombre de rol, es decir, el nombre de rol puede ser usado en expresiones para obtener valores de modo semejante a un atributo de la clase" [RM, p. 354]. La indicación de visibilidad puede ser suprimida, sin que eso signifique que la visibilidad está indefinida o es `public` [UML, p. 3-42]: simplemente, no se desea mostrar en la vista actual. No obstante, la Guía del Usuario afirma que por defecto la visibilidad de un rol es pública [UG, p. 145].

La visibilidad de un extremo de asociación se indica delante del nombre de rol, con la misma notación que la visibilidad de atributos y operaciones, y especifica "la visibilidad del extremo de asociación desde el punto de vista del clasificador en el otro extremo" [UML, p. 2-23], o bien, "la visibilidad de la asociación al transitarla en dirección hacia ese nombre de rol" [UML, p. 3-73]. Nótese que estas dos definiciones, tomadas respectivamente de las Secciones *UML Semantics* y *UML Notation Guide* del Estándar, no son equivalentes, ya que una se refiere a la "visibilidad del extremo" y la otra a la "visibilidad de la asociación". Además, la primera definición, tomada "desde el punto de vista del clasificador en el otro extremo", no es coherente en cuanto al *punto de vista* con las definiciones de cada uno de los cuatro tipos de visibilidad recogidas a renglón seguido en el propio Estándar (y que veremos en seguida): en efecto, el punto de vista adecuado no es el del "clasificador en el otro extremo", sino el de los "clasificadores que ven el clasificador en el otro extremo", exactamente igual que para los atributos y operaciones. Proponemos en su lugar la siguiente definición, más rigurosa en su expresión que las dos anteriores: *la visibilidad de un extremo de asociación especifica la visibilidad de la asociación desde el punto de vista de otros clasificadores al navegar la asociación en dirección hacia ese extremo*.

Del mismo modo que el Estándar equipara un extremo de asociación a un pseudo-atributo, asimila también la visibilidad de un extremo de

asociación a la visibilidad de un atributo, y da las mismas cuatro posibilidades [UML, p. 2-23][106]:

- `(+)public`: otros clasificadores pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo público[107].
- `(#)protected`: los descendientes del clasificador origen pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo protegido.
- `(-)private`: sólo el clasificador origen puede navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo privado.
- `(~)package`: los clasificadores que están en el mismo paquete (o paquete anidado, hasta cualquier nivel) que *la declaración de la asociación* pueden navegar la asociación y usar el nombre de rol en expresiones.

En este último tipo de visibilidad, añadido en la versión 1.4 del Estándar, se pierde el paralelismo con la definición de visibilidad para atributos, y no está muy claro por qué los autores han cambiado el estilo de la redacción. A primera vista esta definición puede parecer ambigua con respecto al *paquete que contiene la declaración de la asociación*, puesto que podría declararse una asociación entre clasificadores de dos paquetes diferentes. Entonces, ¿a qué paquete pertenece la declaración de la asociación? El Estándar contiene en otro lugar distinto la siguiente afirmación: "la asociación es poseída por el paquete que contiene el diagrama" [UML, p. 3-70]. Desgraciadamente, esta afirmación es incorrecta y no resulta de gran ayuda, ya que el concepto de "diagrama" no está en el metamodelo de UML, es decir, no existe ninguna metaclase `Diagram`, y por tanto no es posible determinar qué paquete contiene un determinado diagrama en un modelo. Los diagramas son meras "vistas" o representaciones gráficas de un modelo, y no forman parte propiamente del modelo. Por otra parte, una asociación puede aparecer en varios diagramas.

En todo caso, una asociación es un elemento del modelo, y como tal será declarada en un paquete, que puede ser o no el paquete de los clasificadores asociados (en el caso de una clase-asociación esto es aún más claro). Ahora bien, esto plantea un problema acerca de la *naturaleza misma de una asociación y sus extremos*, que no tiene fácil solución.

---

[106] Nótese que, respecto a la definición de visibilidad de atributos y operaciones vista más arriba, en lugar de "usar la propiedad" se dice "navegar la asociación y usar el nombre de rol". La expresión "navegar la asociación hacia un extremo" debe tomarse como equivalente de "usar el extremo de asociación como si fuera un pseudo-atributo". Por supuesto, para navegar la asociación se requiere como condición previa que ésta sea navegable.

[107] Nótese cómo esta definición de la visibilidad pública da pie a las expresiones de navegación compuestas, examinadas en detalle en la Sección 4.4.

En efecto, el Estándar (y el Manual de Referencia) equipara un extremo de asociación a un pseudo-atributo del clasificador origen (el que está en el otro extremo de la asociación) [UML, p. 2-23]. Si se mantuviera este paralelismo también en la visibilidad de paquete, su definición debería ser:

- `(~)package`: los clasificadores que están en el mismo paquete (o paquete anidado, hasta cualquier nivel) que *el clasificador origen* pueden navegar la asociación y usar el nombre de rol en expresiones, *de modo similar al uso de un atributo con visibilidad de paquete*.

Es decir, la visibilidad no dependería de dónde se declara la asociación, sino de dónde se declaran los clasificadores asociados, del mismo modo que para los otros tres tipos de visibilidad. Esta solución parece sencilla, pero en realidad el planteamiento general de equiparar extremos de asociación a pseudo-atributos está atentando contra la *unidad del concepto de asociación* en sí. En efecto, en lugar de una única declaración de la asociación, tenemos tres declaraciones: la de cada uno de los extremos, que "pertenecen" al clasificador opuesto como "pseudo-atributos", y la de la asociación en cuanto tal[108]. En UML hay una cierta ambivalencia a este respecto, ya que los extremos pertenecen por composición a la asociación [UML, p. 2-14] (ver **Figura 2.20**), pero a la vez se quiere que pertenezcan a los clasificadores asociados en tanto que pseudo-atributos. En definitiva, *no se escoge claramente entre "asociación" como concepto independiente de los clasificadores asociados, y "referencia" como elemento incluido en ellos y más o menos equivalente a un atributo*. De todas formas, en la definición de las visibilidades de un extremo de asociación debería adoptarse un esquema uniforme para los cuatro tipos, válido también para la visibilidad de paquete, como el que hemos propuesto.

5.2.3. Asociaciones bidireccionales entre clases de distintos paquetes

La definición de asociaciones bidireccionales entre clasificadores pertenecientes a distintos paquetes resulta problemática, sea cual sea la visibilidad de los extremos de la asociación, como vamos a mostrar en seguida[109].

En primer lugar, en UML *un elemento se define en un único paquete*, aunque puede ser importado y usado en varios paquetes[110]. Un elemento

---

[108] Si la asociación es n-aria, habrá n+1 declaraciones.

[109] Obviamente, los elementos asociados tendrán que ser declarados públicos en su paquete, para que puedan ser referenciados desde fuera.

[110] Técnicamente un elemento pertenece a un único espacio de nombres, que puede ser un paquete, un clasificador o una colaboración. En el metamodelo, `Package`, `Classifier` y `Collaboration` son subtipos de la metaclase abstracta `Namespace`. La meta-asociación `ElementOwnership` desde `ModelElement` hacia `Namespace` tiene multiplicidad 0..1 (un elemento es *definido* en un único espacio de nombres al cual

puede ser usado en paquetes distintos de aquél en el que es definido pero, evidentemente, *un elemento no puede ser modificado fuera de su paquete propietario*. Este principio no está claramente enunciado en el Estándar, donde a veces parece que el paquete es propietario sólo del *nombre* del elemento contenido, pero no de la *definición* del elemento. En todo caso, parece un principio de sentido común, ya que, si la definición de un elemento estuviera repartida entre varios paquetes, podríamos llegar a situaciones absurdas. Por ejemplo, no es posible definir la clase `A` en el paquete `P`, y posteriormente definir en el paquete `Q` que `A` es subclase de `B`. En cambio, sería posible importar `B` desde `Q` y definir la generalización en el propio paquete `P` (ver **Figura 5.5**).

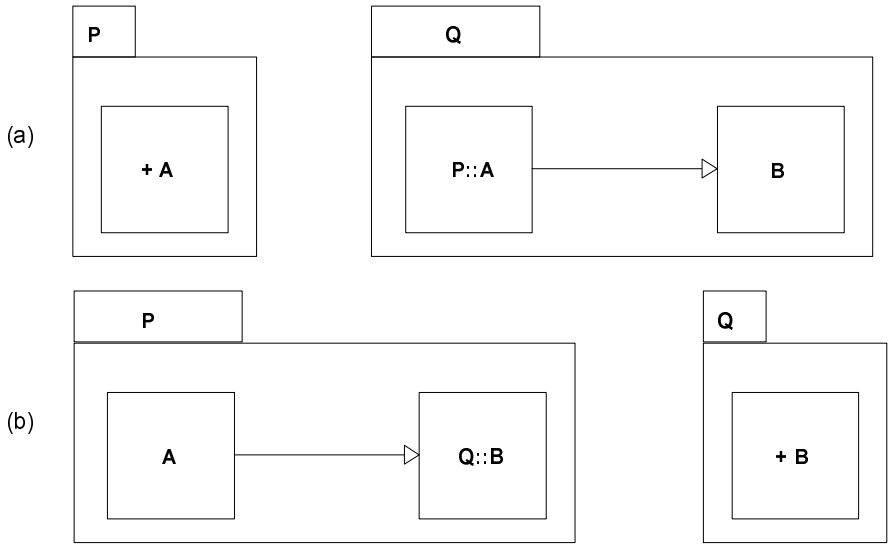

Figura 5.5. Generalización entre clases de dos paquetes distintos: a) incorrecta; b) correcta

El segundo paso del razonamiento es el siguiente. Como ya hemos demostrado en el Apartado 4.6.1, *una asociación navegable induce una dependencia* desde el clasificador origen hacia el clasificador destino: si la clase `A` tiene una asociación navegable hacia la clase `B`, entonces la clase `A` depende de la clase `B`. *La dependencia afecta a la definición del elemento*, independientemente de cómo se implemente la asociación. En efecto, una posible implementación[111], bastante típica, consiste en definir en la clase `A` un atributo que almacene una referencia a un objeto de la clase `B`, lo que convierte a la clase `A` en dependiente de la clase `B`. En principio, puede haber otros esquemas de implementación que no estén basados en el uso de

---

pertenece), y la meta-asociación `ElementImport` desde `ModelElement` hacia `Package` tiene multiplicidad * (un elemento puede ser *usado* en varios paquetes, que deben importarlo para ello) [UML, p. 2-188].

[111] En el Capítulo 6 trataremos con más detalle de la implementación de asociaciones.

atributos, dependiendo del lenguaje de programación empleado, pero en todo caso la dependencia existe de una forma u otra, ya que la clase `A` tendrá conocimiento de la clase `B`. Como la dependencia modifica el elemento dependiente, debe ser definida en el mismo paquete, es decir, no es posible añadir dependencias a un elemento fuera de su paquete original. Por tanto, *el extremo navegable de una asociación debe ser definido en el mismo paquete que el clasificador origen*.

Tercer paso. No hay problema en definir una asociación bidireccional entre dos clasificadores pertenecientes al mismo paquete, ver **Figura 5.6** (a), ya que la asociación y sus extremos estarían definidos en el mismo paquete que los clasificadores asociados (y estos podrían tener visibilidad pública o de paquete). Tampoco hay problema en definir una asociación unidireccional entre una clase `A` de un paquete `P` y otra clase `B` pública de otro paquete `Q`, ver **Figura 5.6** (b), siempre que la asociación (y por tanto su extremo navegable) se defina en el mismo paquete que la clase origen. En cambio, no es posible definir una asociación bidireccional entre estas dos clases, ver **Figura 5.6** (c), ya que esto exigiría definir la asociación en cada uno de los dos paquetes, y *una asociación debe ser definida en un único paquete*, como cualquier otro elemento del modelo. Aún peor sería el caso en que la asociación estuviera definida en un paquete distinto de los paquetes de los clasificadores asociados, como en la **Figura 5.6** (d). Por lo tanto, *una asociación bidireccional sólo puede definirse entre clasificadores que pertenezcan al mismo paquete*[112].

No debe pensarse que esto es una grave limitación de UML. En realidad se trata de una consecuencia natural, aunque poco evidente, del concepto de paquete. He aquí otras configuraciones que sí son posibles en UML:

- Definir dos asociaciones unidireccionales recíprocas entre clases de dos paquetes distintos, ver **Figura 5.7** (a). Esto induce una dependencia mutua entre las clases y por ende entre los paquetes, pero no es incorrecto. En la mencionada figura se representan explícitamente las dependencias[113], que en los casos anteriormente vistos también existen.
- Definir, dentro de un paquete, una asociación bidireccional entre una clase y una subclase de una clase importada de otro paquete, como podemos observar en la **Figura 5.7** (b). Esto es, ni más ni

[112] El Estándar adopta una perspectiva más "blanda" a este respecto, ya que sólo exige que los clasificadores asociados estén *accesibles* o hayan sido *importados* desde el espacio de nombres donde se define la asociación [UML, p. 2-52]. Ya hemos argumentado que esto implica modificar la definición de los clasificadores asociados fuera de su espacio de nombres propietario, y por tanto no debería permitirse.

[113] Técnicamente son relaciones de dependencia `Permission` con el estereotipo «`access`», «`import`» o «`friend`» [UML, p. 2-47]. Para la definición de "estereotipo" ver la Nota 9 en la Sección 2.2.

menos, la práctica habitual en la que los elementos de un paquete de librería son importados y especializados en el paquete cliente.

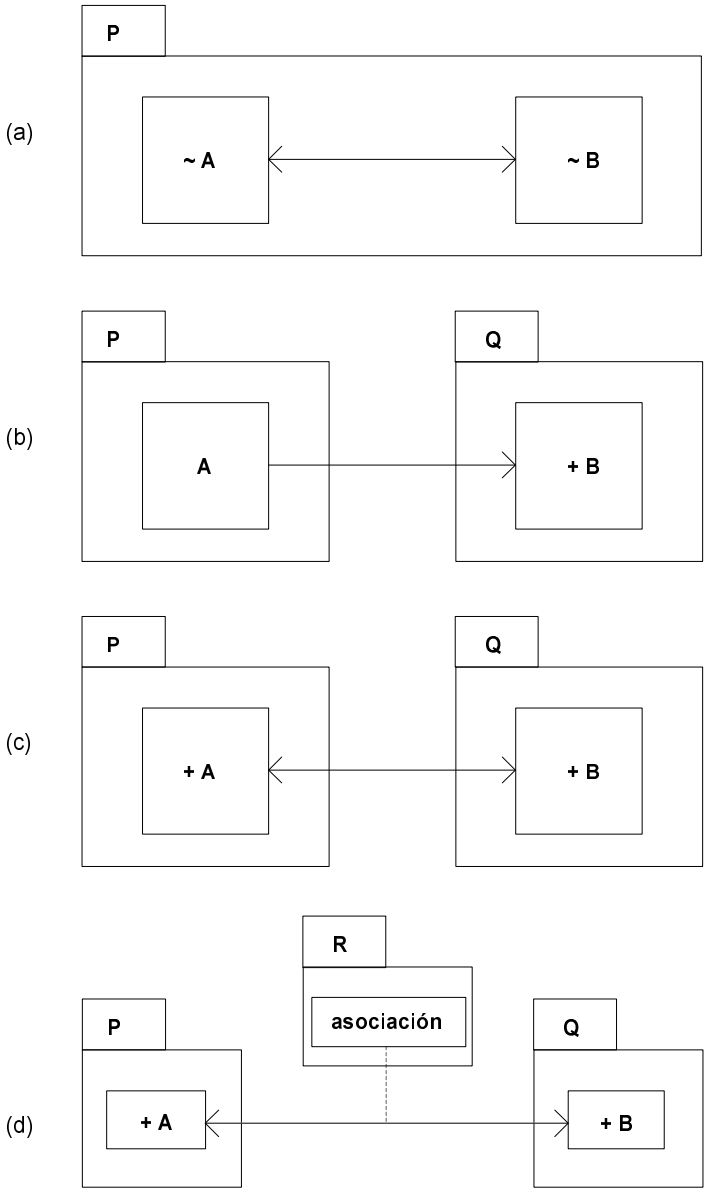

Figura 5.6. Asociaciones entre clases de distintos paquetes: a) asociación bidireccional entre clases de un mismo paquete; b) asociación unidireccional entre clases de dos paquetes distintos; c) asociación bidireccional (incorrecta) entre clases de dos paquetes distintos; d) asociación bidireccional (también incorrecta) entre clases de dos paquetes distintos, definida en un tercer paquete (se representa como clase-asociación por claridad)

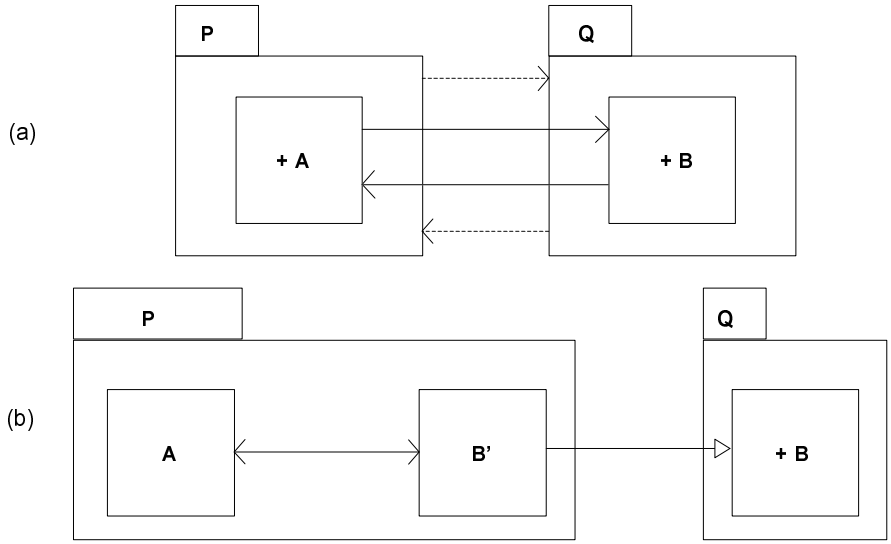

Figura 5.7. Configuraciones permitidas: a) dos asociaciones unidireccionales recíprocas entre clases de dos paquetes distintos; b) asociación bidireccional entre una clase y una subclase de una clase de otro paquete

5.3. Otras formas de especificar la interfaz de una asociación

La visibilidad combinada de atributos, operaciones y asociaciones proporciona una primera forma de especificar la interfaz de una clase: una *interfaz pública* disponible para todos los elementos del modelo (que podemos denominar *interfaz nativa*), y una *interfaz semipública* para todos los elementos contenidos en el mismo paquete[114]. Como primera aproximación puede ser suficiente para resolver problemas sencillos, ¿pero cómo conseguir que determinadas propiedades sean visibles a través de una asociación, sin serlo a través de las demás, y así lograr un desacoplamiento más efectivo de las clases participantes en una asociación? Es decir, ¿cómo podemos conseguir una interfaz distinta para cada asociación? Las visibilidades `public` y `package` son claramente insuficientes para conseguir este comportamiento, ya que no discriminan entre las distintas asociaciones a las que está conectado un clasificador[115].

---

[114] Insistimos una vez más en que, además de la *interfaz visible*, será necesaria una *ruta navegable* desde la clase origen hacia la clase destino, es decir, una asociación estructural o contextual.

[115] En UML un espacio de nombres puede dar permiso de acceso «`friend`» a un elemento externo, lo que significa que el elemento externo puede acceder a los elementos contenidos en el espacio de nombres sea cual sea su visibilidad [UML, p. 2-47]. Como puede observarse, este permiso especial tampoco sirve para conseguir un acceso distinto a través de cada asociación.

En UML hay dos mecanismos que permiten abordar este problema: especificadores de interfaz e interfaces propiamente dichas. Veamos en detalle cada uno de ellos.

5.3.1. Especificadores de interfaz

En la *notación* de UML, el nombre de rol puede ir seguido por el signo ':' y el nombre de un clasificador para especificar la interfaz del extremo de asociación (ver **Figura 5.8**). El especificador de interfaz indica el comportamiento que un objeto espera de la instancia asociada, en otras palabras, el comportamiento requerido para posibilitar la asociación. En este caso, la clase asociada normalmente tendrá otras funcionalidades adicionales que no son requeridas para esta asociación en concreto. La clase destino debe ser *compatible* con el especificador de interfaz[116] (que puede ser una interfaz propiamente dicha, un tipo, u otra forma específica de clasificador). Si el especificador de interfaz se omite, la asociación puede usarse para obtener un acceso completo a la clase asociada [UML, p. 3-72].

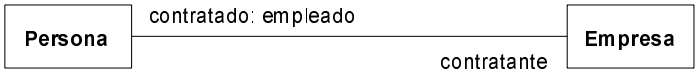

Figura 5.8. Ejemplo de asociación con especificador de interfaz

En el *metamodelo* de UML, esto se traduce en la meta-asociación desde `AssociationEnd` (rol `specifiedEnd`) hacia `Classifier` (rol `specification`), que determina cero o más clasificadores que especifican las operaciones que pueden ser aplicadas a una instancia del clasificador destino que sea accedida a través de este extremo de asociación (ver **Figura 2.20**). Estos clasificadores-especificadores determinan la interfaz mínima que debe ser realizada por el clasificador que de hecho se conecte al extremo de la asociación, para cumplir con la finalidad de la asociación; *no indican ni restringen las clases que participan en una asociación, sino sólo las operaciones que pueden ser invocadas por medio de ella* [UML, p. 2-24].

Hasta aquí lo que dice el Estándar. En el ejemplo de la **Figura 5.8**, el *nombre de rol* `contratado` va seguido por el *especificador de interfaz* `empleado`, que es el nombre de otro clasificador que no aparece como tal en el diagrama. El acceso desde `Empresa` hacia `Persona` a través de la

[116] Se dice que un clasificador es *compatible con una especificación* cuando la realiza, es decir, cuando implementa todas las operaciones y recepciones de señales contenidas en la especificación; la compatibilidad *afecta al comportamiento* (operaciones y recepciones), *pero no a la estructura* (atributos y asociaciones estructurales); además, da la impresión de que la compatibilidad sólo se refiere a las operaciones y recepciones *públicas*, aunque el Estándar no lo dice claramente [UML, pp. 2-18 y 3-51]. La compatibilidad del clasificador destino con el especificador de interfaz se expresa en UML mediante una relación de "realización" (*realization relationship*), como veremos con más detalle en el siguiente Apartado. Esta relación está implícita en la notación del especificador de interfaz.

asociación `contratante-contratado` está limitado a lo que permite la interfaz especificada por `empleado`. En sentido inverso, por el contrario, no se ha especificado ninguna limitación de acceso. Aparentemente, el especificador de interfaz soluciona exactamente el problema que hemos dejado pendiente en la Sección anterior, es decir, cómo conseguir una interfaz distinta para cada asociación. La ventaja esencial del uso de especificadores de interfaz es que *limita la dependencia inducida por la asociación* desde la clase origen hacia la clase destino, puesto que la clase origen ahora sólo depende de las operaciones incluidas en el especificador, y no de otras propiedades de la clase destino. Veamos ahora los problemas que plantea la definición del especificador de interfaz en la versión actual de UML.

En primer lugar, un inconveniente menor, pero que puede ser revelador: es fácil que la notación para el especificador de interfaz resulte en una *inflación de nombres*, puesto que tanto el nombre de rol como el especificador de interfaz expresan conceptos muy similares, a saber, el rol que desempeña el clasificador destino en la asociación. Steimann ha propuesto fundir ambos conceptos de modo que el nombre de rol sea utilizado para designar la interfaz del extremo de asociación [Steimann 00, Steimann 01], idea que recogemos más adelante en nuestra propuesta.

En segundo lugar, el especificador de interfaz puede ser *cualquier tipo de clasificador*, no sólo una interfaz propiamente dicha. Esta generalidad parece excesiva; en efecto, supongamos que el especificador es una clase; aunque esta clase defina atributos, el clasificador destino, de acuerdo con la noción de compatibilidad en UML, sólo está obligado a realizar sus operaciones, que son las que verá el clasificador origen a través de la asociación; por tanto no hay razón para permitir que el especificador de interfaz pueda ser una clase con atributos; sería mucho más adecuado limitar los tipos de clasificadores que pueden ser especificadores, y modificar el metamodelo de modo que el especificador de interfaz sólo pueda ser una interfaz propiamente dicha (metaclase `Interface`) [Steimann 00].

En tercer lugar, es sorprendente que el Estándar permita *cero o más* especificadores de interfaz para un extremo de asociación dado. La notación sólo indica cómo expresar un único especificador, aunque no sería muy problemático extenderla para permitir varios especificadores separados por comas [Steimann 00], por ejemplo, pero ¿cuál sería su significado? Es decir, ¿cuál sería la interfaz resultante? Como el clasificador destino debe ser compatible con todos y cada uno de los especificadores del extremo, la interfaz resultante sería la unión de todos ellos. Esto equivale en la práctica a la definición de un nuevo especificador de interfaz implícito, que será subtipo de todos los especificadores explícitos. Parece mucho más sencillo y conveniente limitar el número máximo de especificadores a uno.

En cuarto lugar, *si el especificador se omite*[117], se dice que la asociación permite el "acceso completo" a la clase asociada; es decir, el acceso a aquellas propiedades (atributos y operaciones, e incluso extremos de asociación) cuya visibilidad permite de por sí el acceso desde el clasificador origen: en principio, las propiedades públicas o semipúblicas (de paquete) proporcionadas por la clase, y también las propiedades privadas o protegidas en el caso de asociaciones reflexivas (ver Apartado 5.2.1). Por el contrario, *si el especificador no se omite*, su mero uso excluye el acceso a los atributos de la clase asociada; en cuanto a las operaciones, parece que la intención es limitar el acceso a las que tienen visibilidad pública (por analogía con las interfaces propiamente dichas, que sólo definen operaciones públicas), aunque esto no está dicho explícitamente en el Estándar, que no explica claramente la relación, si es que la hay, entre el especificador de interfaz y la visibilidad de las operaciones. Notemos, pues, que el acceso completo puede dar mucho más de lo que proporcionaría un especificador de interfaz con todas las operaciones públicas del clasificador destino (lo que antes hemos denominado "interfaz nativa").

Estos inconvenientes se podrían resolver hasta cierto punto matizando la definición de especificador de interfaz de la siguiente manera:

El nombre de rol puede ir seguido por el signo ':' y el nombre de una interfaz que especifica el subconjunto de las operaciones públicas del clasificador destino que son accesibles a través de la asociación. El especificador de interfaz puede ser suprimido en una vista aunque exista en el modelo subyacente. El clasificador destino debe ser compatible con la interfaz, es decir, existe una relación de realización implícita desde el clasificador destino hacia la interfaz especificada. Si no se especifica ninguna interfaz, la asociación puede usarse para acceder a todas las propiedades visibles del clasificador destino.

De acuerdo con esta definición, habría que modificar ligeramente el metamodelo, sustituyendo `Classifier` por `Interface` en el rol `specification` de `AssociationEnd`, con multiplicidad 0..1.

El problema fundamental del especificador de interfaz es que se trata sustancialmente de lo mismo que una interfaz propiamente dicha, por tanto resulta un concepto redundante y superfluo.

5.3.2. Interfaces

Uno de los elementos que pueden figurar en un modelo UML es la interfaz (metaclase `Interface`, subtipo de `Classifier` en el

[117] Aquí nos referimos a que sea omitido en el modelo subyacente, no meramente suprimido en la notación. El Estándar no dice explícitamente si el especificador de interfaz puede ser suprimido en la notación, es decir, si se puede omitir su presentación en un diagrama aunque exista en el modelo. Por tanto, hay que aplicar la regla general, y suponer que es suprimible al igual que el nombre de rol, la multiplicidad, la navegabilidad, la visibilidad, etc.

metamodelo). El propósito de una interfaz es expresar *un grupo de operaciones que definen conjuntamente un servicio coherente* ofrecido por un clasificador. Una interfaz es sólo un conjunto de operaciones[118] con un nombre colectivo, *sin ninguna estructura interna*; en concreto, una interfaz no tiene atributos. El clasificador que ofrece el servicio (una clase, un componente, un subsistema, etc.) se dice que "realiza" (*realize*) o "implementa" (*implement*) la interfaz. La interfaz especifica un conjunto de operaciones visibles externamente sin especificar la estructura interna del clasificador. *Todas las operaciones definidas en una interfaz son públicas.* Un clasificador puede ofrecer distintos servicios, es decir, puede realizar varias interfaces, y varios clasificadores pueden realizar la misma interfaz. El clasificador que realiza la interfaz debe realizar al menos todas las operaciones especificadas en ella[119]. La especificación de una operación incluye su signatura (nombre, tipos de los parámetros y del valor de retorno), pero no su implementación, aunque puede incluir una especificación de los efectos de su invocación (mediante pre- y post-condiciones, por ejemplo, o simplemente en lenguaje natural) [UML, pp. 2-41, 2-61, 2-71 y 3-50].

Existen dos formas básicas de representar una interfaz [UML, p. 3-51]. En primer lugar, una interfaz se puede representar como un *rectángulo de clasificador* con la palabra clave «`interface`»[120] y compartimentos para el nombre de interfaz y la lista de operaciones. El compartimento de los atributos puede omitirse porque siempre está vacío. El clasificador que realiza la interfaz se muestra unido a ella mediante el *símbolo de realización*, que es una línea discontinua terminada en un triángulo (como el símbolo de generalización, pero con línea discontinua)[121]. En el ejemplo de la **Figura 5.9**, `Persona` realiza la interfaz `empleado`, que define la operación `contratar`, por tanto `Persona` también debe definir e

---

[118] No está nada claro por qué el Estándar omite toda referencia a las recepciones de señales en su definición de interfaz. Recuérdese que `Reception` también es, junto con `Operation`, subtipo de `BehavioralFeature` en el metamodelo [UML, p. 2-104]. La única referencia a las recepciones se encuentra en la primera regla para la correcta formación de interfaces (*well-formedness rules*), que además resulta ser contradictoria: el enunciado en lenguaje natural afirma que una interfaz sólo contiene operaciones, pero la expresión OCL incluye tanto operaciones como recepciones [UML, p. 2-61].

[119] Lo que no está claro es si las operaciones deben tener la misma visibilidad pública en el clasificador y en la interfaz. En el entorno .NET, por ejemplo, esto no es necesario, de modo que una operación pública de una interfaz podría ser implementada por una operación privada del clasificador [Barwell 02].

[120] Atención: según el Estándar, «`interface`» es una "palabra clave" (*keyword*), no un "estereotipo" (*stereotype*), por tanto una interfaz no es una clase estereotipada, aunque la notación lo sugiera. Sin duda el Estándar es particularmente confuso en este punto.

[121] Técnicamente la realización es una dependencia de abstracción (*abstraction dependency*) con el estereotipo «`realize`» [UML, p. 3-52].

implementar esta operación, aunque no esté representada en el diagrama. En segundo lugar, una interfaz también puede representarse de forma más compacta como un *círculo* con el nombre de la interfaz situado debajo de él, sin expresar la lista de operaciones. Si la interfaz se representa como un círculo, la relación de realización puede representarse como una línea continua sin punta de flecha[122]. En el ejemplo de la **Figura 5.9**, tanto `Persona` como `Empresa` realizan la interfaz `titular` (de una cuenta corriente).

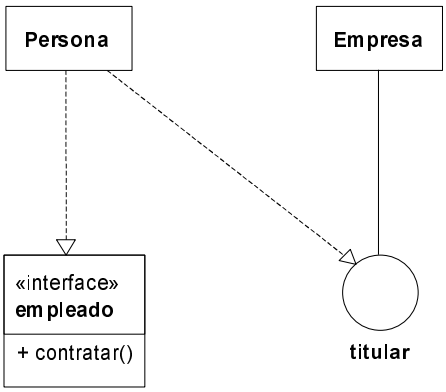

Figura 5.9. Ejemplo de representación de interfaces y su relación con las clases que las implementan. La clase `Persona` realiza las interfaces `empleado` y `titular`, la clase `Empresa` realiza la interfaz `titular`

Una interfaz, como cualquier otro clasificador, es un *elemento generalizable*, es decir, es posible generalizarla o especializarla, y construir así jerarquías de interfaces. Un clasificador que realice una interfaz, por tanto, debe implementar tanto sus operaciones como las de sus generalizaciones [UML, p. 2-71]. Según el Estándar, una interfaz es *formalmente equivalente a una clase abstracta* con operaciones abstractas y sin atributos [UML, p. 3-51]. Una clase abstracta no tiene instancias directas, sino sólo a través de sus subclases; una operación abstracta consiste sólo en la signatura de la operación, sin proporcionar implementación alguna. Al igual que la clase abstracta, una interfaz no es directamente instanciable [UML, p. 2-71], pero puede decirse que tiene *instancias indirectas*, que son las instancias de los clasificadores que la realizan. En este sentido, no parece casual que la notación elegida para la realización sea visualmente similar a una generalización, es decir, se usa un

[122] En general, en UML puede haber más de una representación gráfica para un mismo elemento del lenguaje [UML, p. 2-17]; no obstante, en el caso de un estereotipo se define un único icono gráfico [UML, p. 2-76], de modo que la posibilidad de representar la realización de interfaces de dos formas distintas no parece estar de acuerdo con el metamodelo. Además, el uso de la línea continua puede ser ambiguo, ya que podría significar también una asociación (sin la navegabilidad expresada) entre el clasificador y la interfaz.

símbolo de generalización discontinuo (*dashed generalization symbol*) [UML, p. 3-51] para sugerir que la realización de una interfaz es "una forma débil de herencia" [Stevens 00].

Los clasificadores que *usan* una interfaz (por contraste con los que la *realizan*) pueden representarse de dos formas distintas (ver **Figura 5.10**). En primer lugar, mediante una *flecha discontinua* dibujada desde la clase que usa las operaciones hacia la interfaz que las proporciona[123] [UML, p. 3-51]. En segundo lugar, mediante una *asociación unidireccional* dibujada de la misma manera [UML, p. 3-50]. En este último caso las instancias de la asociación serán enlaces entre instancias de la clase origen e instancias de la clase que realiza la interfaz, que son instancias indirectas de la interfaz. Téngase en cuenta además que la primera representación, en forma de dependencia de uso, esconde en realidad una asociación, tal como se ha explicado detalladamente en el Apartado 4.5.3, de modo que en el fondo las dos representaciones corresponden a una misma realidad.

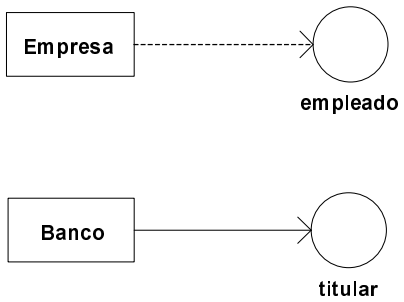

Figura 5.10. Uso de interfaces, representado como dependencia o asociación unidireccional

5.3.3. Comparación

Las dos construcciones analizadas, especificadores de interfaz e interfaces propiamente dichas, persiguen el mismo objetivo: *minimizar las dependencias* creadas por las asociaciones entre clases. Por otra parte, se trata de dos conceptos tan similares que probablemente se podrían *unificar para simplificar el lenguaje*. No obstante, hay dos diferencias importantes que vamos a estudiar a continuación.

En primer lugar, una interfaz puede ser realizada por varias clases, lo que permite que, si tenemos una asociación conectada a una interfaz, en el extremo de la asociación pueda haber instancias de distintas clases, todas ellas sujetas a la misma definición de asociación (por ejemplo respecto a la multiplicidad); es decir, *la interfaz permite compartir el extremo de asociación entre varias clases* sin necesidad de crear una superclase[124]. Por

[123] Técnicamente se trata de una dependencia de uso (*usage dependency*) [UML, p. 3-52].

[124] En la interpretación de Steimann, una interfaz es un *rol* que puede ser desempeñado por instancias de varias clases distintas; desde este punto de vista puede

el contrario, un especificador de interfaz puede ser compartido por varias clases, pero en asociaciones distintas, no en la misma asociación. Así pues, en la **Figura 5.11** (a) podemos observar cómo el uso de interfaces permite definir fácilmente que una `Cuenta` es poseída exactamente por un `titular`, que puede ser una `Persona` o una `Empresa`; en cambio, el uso de especificadores de interfaz en la **Figura 5.11** (b) permite definir que una `Persona` y una `Empresa` presentan la misma interfaz `titular` para una `Cuenta`, pero en asociaciones distintas, de modo que no es posible definir una restricción de multiplicidad común para ambas clases[125].

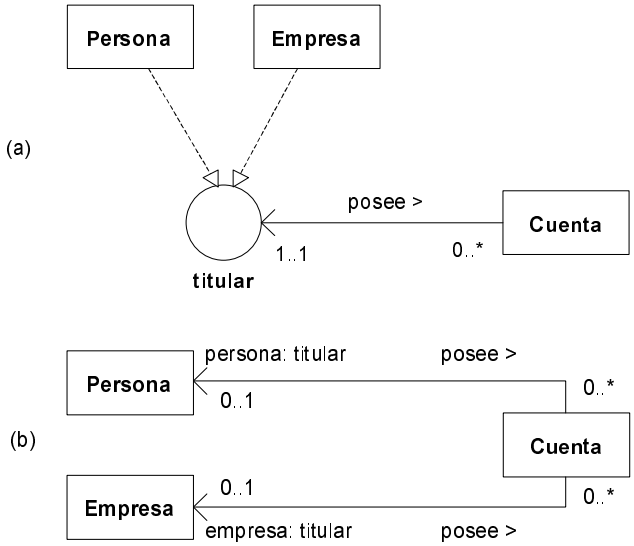

Figura 5.11. Comparación de interfaces y especificadores de interfaz: a) una interfaz puede ser realizada por varias clases que comparten el extremo de asociación; b) un especificador de interfaz no puede ser compartido por varias clases en el mismo extremo de asociación

Tenemos aquí, por tanto, una importante diferencia conceptual entre interfaces y especificadores: no es lo mismo *conectar* un extremo de asociación directamente a una interfaz, que *determinar la funcionalidad* requerida en el extremo mediante un especificador de interfaz. En este sentido el especificador de interfaz es menos potente que la interfaz propiamente dicha, porque no permite mostrar varias clases que desempeñen el mismo rol en una asociación. Así pues, desde este punto de

---

considerarse que el rol es un *supertipo* de las clases que lo realizan [Steimann 01], lo cual encaja con la idea de que la realización es una *forma débil de generalización* [Stevens 00]. La ventaja del concepto de rol como supertipo es que es más flexible que una *superclase*. Además, algunos lenguajes de programación, como Java, no permiten la generalización múltiple, pero sí permiten la realización de múltiples interfaces.

[125] Una posible solución es el uso de la restricción `{xor}` entre las dos asociaciones, pero sin duda es mucho menos versátil y elegante que el uso de interfaces. Nótese cómo en este ejemplo el uso de interfaces no sólo ayuda a expresar mejor la *interacción* requerida a través de una asociación, sino también la *estructura* requerida.

vista, las interfaces son preferibles, porque a la vez que especifican la funcionalidad requerida, permiten compartir el extremo de asociación entre varias clases[126].

La segunda diferencia importante entre especificadores e interfaces es que *una interfaz no puede estar conectada a una asociación bidireccional*. Es decir, en UML una interfaz puede participar en una asociación sólo si la asociación es navegable *hacia* la interfaz, pero no *desde* la interfaz [UML, pp. 2-41 y 3-50]. A primera vista, esta restricción de UML resulta sorprendente e incómoda, ya que impide aplicar las ventajas de las interfaces a las asociaciones bidireccionales. Es más, si no tuviéramos esta restricción, podríamos prescindir por completo de los especificadores de interfaz y sustituirlos por interfaces propiamente dichas, que como hemos visto tienen mayor potencia expresiva, simplificando así el lenguaje. Algunos autores ya han propuesto eliminar esta restricción, para la que no encuentran ninguna justificación plausible [Steimann 00, Steimann 02][127]. Se potenciaría así el uso de interfaces frente al abuso de clasificadores en general que se observa actualmente en UML.

¿Cuál es la razón de ser de esta restricción? Al parecer, el motivo es que una interfaz en UML no es más que un conjunto de operaciones con nombre colectivo, que especifican un *comportamiento externo* sin especificar la *estructura interna* necesaria para implementarlo. Según esto, es correcto concluir que una interfaz no tiene *implementación* propia, pero probablemente UML va más allá de lo necesario al afirmar que una interfaz tampoco tiene *estado* (atributos y asociaciones) [UML, p. 3-50]. Es decir, se ha asumido acríticamente que el estado de un clasificador pertenece a su estructura interna, y por tanto no puede ser especificado por una interfaz. Por el contrario, es perfectamente razonable *una noción de interfaz que incluya tanto la especificación del comportamiento como la especificación del estado*. Con esto se pone de manifiesto, además, que el comportamiento y el estado son en realidad inseparables. El clasificador que realice la interfaz tendrá que implementar el comportamiento (operaciones) junto con el estado (atributos y asociaciones), mediante la necesaria estructura interna. El comportamiento y el estado especificados en la interfaz serán visibles externamente a través de esta misma interfaz, mientras que la estructura interna no lo será[128].

---

[126] Por otra parte, cualquier clasificador que realice la interfaz compartirá el extremo de asociación (aunque no se muestre explícitamente), lo cual puede ser considerado también un inconveniente de las interfaces en determinadas circunstancias.

[127] Steimann es bastante radical al afirmar que la raíz del problema está en que UML ha adoptado servilmente la noción de interfaz de Java, mientras que otros estándares del OMG como CORBA o IDL son mucho más flexibles en permitir asociaciones bidireccionales entre interfaces [Steimann 00].

[128] Esta noción de interfaz sería más o menos equivalente al concepto de *tipo* en UML [Stevens 00, p. 86], al que no corresponde ninguna metaclase específica, sino una clase con el estereotipo predefinido «`type`» [UML, pp. 2-27 y 3-49]. Es decir, en UML

Una última observación relativa a esta noción de interfaz que permite asociaciones bidireccionales. El estado especificado por una interfaz incluye las asociaciones navegables *desde* la interfaz, pero no las asociaciones unidireccionales que son sólo navegables *hacia* la interfaz, y por tanto desconocidas para ella. En efecto, el estado de un objeto consiste en los valores de sus atributos junto con los enlaces que tiene con otros objetos, pero sólo aquellos enlaces que el objeto en cuestión conoce, es decir, los que son navegables desde él mismo hacia los otros objetos. Si un objeto es destino de enlaces unidireccionales, él mismo no tiene conocimiento de esos enlaces, así que no son parte de su estado. Por tanto una asociación unidireccional no forma parte de la estructura de la clase destino. El resto de asociaciones (bidireccionales, y unidireccionales para la clase origen) sí forman parte de la estructura de la clase, y análogamente de la interfaz.

5.4. Solución propuesta: asociaciones entre interfaces

A la vista de los problemas encontrados en las definiciones de visibilidad de atributos y operaciones, visibilidad de extremos de asociación, especificadores de interfaz, e interfaces propiamente dichas, vamos a proponer una solución que, con el mínimo número de cambios en el lenguaje UML, sea capaz de dar respuesta a esas dificultades.

El *objetivo* de esta solución es doble (en realidad no se trata de dos objetivos distintos, sino de las dos caras de un mismo objetivo): deseamos *minimizar las dependencias entre las clases participantes* en una asociación, de modo que se reduzca su acoplamiento; y deseamos también *proporcionar una interfaz distinta para cada asociación*, es decir, un acceso distinto a las funcionalidades de una clase a través de cada una de las asociaciones en las que participa. Lo segundo es una forma directa, quizás no la única, de conseguir lo primero. Para conseguir este objetivo *no basta con asignar una interfaz a cada extremo de asociación*, al modo de un especificador de interfaz, porque así la clase origen sigue dependiendo de la clase destino concreta, aunque sólo conozca de ella la parte que revela el especificador de interfaz.

La solución que vamos a proponer, en consonancia con ideas de otros autores [Steimann 00, Steimann 01], unifica los conceptos de nombre de rol, especificador de interfaz e interfaz propiamente dicha, introduciendo un cambio importante en el concepto de asociación en UML. En la versión actual una asociación es básicamente una relación definida entre

---

un tipo es una clase estereotipada, pero una interfaz no, como hemos aclarado más arriba. Un tipo puede tener atributos y asociaciones además de operaciones, pero siempre sin definir su implementación. No obstante, el clasificador que realice el tipo no está obligado a implementar sus atributos y asociaciones; en este sentido, la nueva noción de interfaz implica así mismo *una nueva noción de compatibilidad o realización*, que incluya tanto la estructura como el comportamiento.

clasificadores, teniendo opcionalmente un especificador de interfaz asignado a cada extremo. En nuestra solución, *una asociación es directamente una relación entre interfaces*, pudiendo estar cada una de ellas realizada por uno o más clasificadores y permitiendo así gran flexibilidad en el diseño, ya que la asociación es definida independientemente de los clasificadores conectados. En cada extremo *la interfaz especifica la estructura y el comportamiento* que es posible conocer navegando hacia ese extremo a través de la asociación, y que deben satisfacer los objetos asociados. Por tanto, una interfaz especifica no sólo un comportamiento, sino también una estructura, es decir, atributos y asociaciones navegables.

Definir las asociaciones entre interfaces es una forma conveniente de llevar a la práctica el *principio de encapsulamiento*, es decir, separar la interfaz de la implementación, y programar (o diseñar) teniendo en cuenta sólo la interfaz, no la implementación [Steimann 01]. Este principio, incorporado por ejemplo en la teoría de *patrones de diseño* [Gamma 94], es uno de los pilares la orientación a objetos y ha dado importantes frutos que avalan su aplicación.

A continuación explicamos en Apartados distintos las consecuencias que esta solución tiene para la notación, el metamodelo, y la visibilidad de atributos y operaciones.

5.4.1. Notación

Como ya hemos establecido anteriormente, en nuestra propuesta *una asociación es una relación directamente definida entre interfaces*. Ahora bien, conviene facilitar un medio para que la notación no se recargue excesivamente y permita vistas simplificadas del modelo, de modo que el lenguaje siga siendo un instrumento práctico y manejable. Así pues, *la notación que presentamos permite distintos niveles de complejidad* para expresar un mismo modelo con mayor o menor grado de detalle. En los niveles más sencillos las interfaces no se muestran, o se muestran de forma abreviada; en los niveles sucesivos se muestran con mayor detalle y se realza su papel en los extremos de la asociación. Así pues, *la interfaz siempre existe* en el extremo de la asociación, *aunque no se muestre* en un diagrama concreto.

La propuesta presenta dos cambios importantes en la notación:

- Desaparece el *especificador de interfaz*, cuya misión es ahora innecesaria, ya que es asumida por la interfaz conectada al extremo de la asociación.
- Desaparece también el *nombre de rol* como tal, aunque en los niveles sencillos de notación puede usarse para representar la interfaz de modo abreviado.

En la **Figura 5.12** podemos observar tres formas equivalentes de expresar, con mayor o menor detalle, las interfaces de una asociación bidireccional entre las clases `Persona` y `Libro`. En el caso (a) se han

omitido las interfaces; esto puede significar que, o bien existen en el modelo pero no se desea mostrarlas en el diagrama, o bien no existen en el modelo, en cuyo caso hay que suponer una *interfaz implícita* en cada extremo de asociación, que permite el acceso a todas las propiedades públicas de la clase conectada (lo que hemos denominado *interfaz nativa*). En el caso (b) se han representado las interfaces `autor` y `publicación` explícitamente como nombres de rol, es decir, cada "nombre de rol" es en realidad el nombre de una interfaz que es realizada por el clasificador. En el caso (c) tenemos exactamente la misma información, pero representada de modo más explícito y menos compacto.

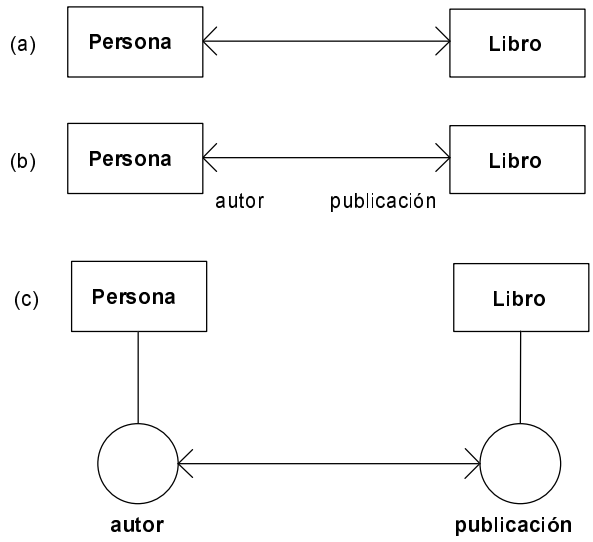

Figura 5.12. Tres formas equivalentes de expresar, con mayor o menor detalle, las interfaces de una asociación bidireccional: a) interfaces omitidas; b) interfaces expresadas como nombres de rol; c) interfaces explícitas

En la **Figura 5.13** podemos observar tres formas algo más complejas de expresar las mismas interfaces, que además están compartidas por varias clases. El caso (a) es igual al de la **Figura 5.12** (c), salvo que las interfaces `autor` y `publicación` son realizadas también respectivamente por las clases `Institución` y `Artículo`. En el caso (b), en lugar de la línea continua, se usa el símbolo propio de la realización, que resulta más claro cuando hay interfaces compartidas, como puede observarse comparando con el caso (a). En el caso (c) se usa el rectángulo de clasificador con la palabra clave `«interface»`, lo que es adecuado para expresar la lista de propiedades que especifica la interfaz. Obsérvese que el rectángulo contiene compartimentos tanto para atributos como para operaciones.

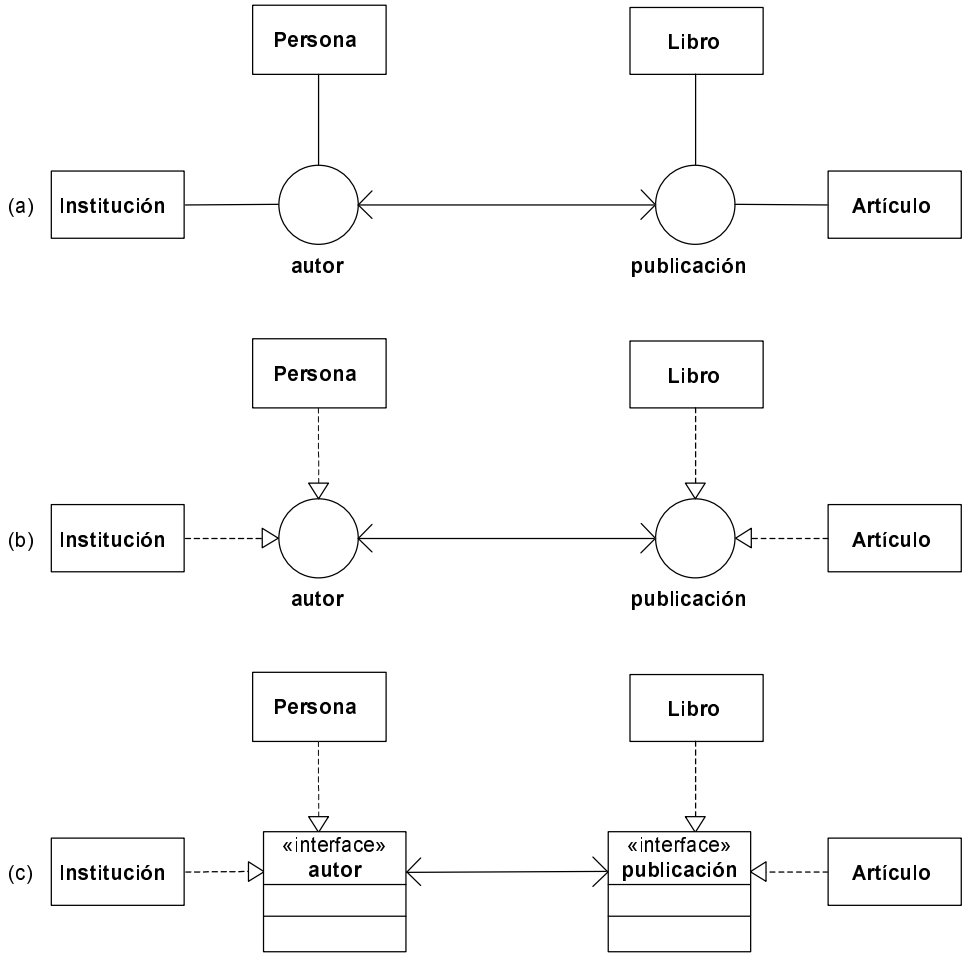

Figura 5.13. Tres formas equivalentes de expresar que la interfaz de un extremo de asociación es compartida por varias clases: a) círculo unido por línea continua a las clases que realizan la interfaz; b) círculo unido mediante símbolo de realización; c) rectángulo de clasificador con la palabra clave «`interface`»

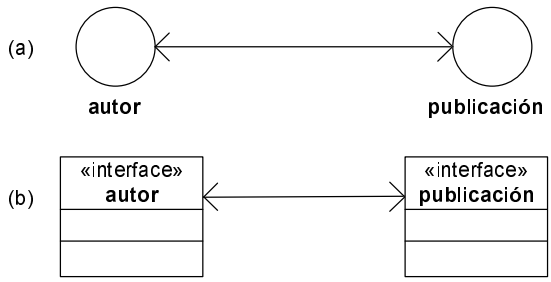

Figura 5.14. Representación de una asociación independientemente de las clases participantes: a) círculos de interfaz en los extremos; b) rectángulos de clasificador con la palabra clave «`interface`»

También tendría pleno sentido modelar la asociación con independencia de las clases que realicen las interfaces de los extremos, como se representa en la **Figura 5.14**.

5.4.2. Metamodelo

Veamos ahora el metamodelo que correspondería a esta propuesta (ver

Figura 5.15), comparado con el metamodelo original (**Figura 2.20**). Podemos observar las siguientes diferencias fundamentales:

- La metaclase `Interface` sustituye a `Classifier` en los extremos de asociación (`AssociationEnd`).
- Desaparece la meta-asociación desde `AssociationEnd` hacia `Classifier` (rol `specification`), que representaba los especificadores de interfaz.
- Como todo elemento del modelo, `AssociationEnd` sigue heredando el meta-atributo `name` de `ModelElement`; no obstante, el "nombre de rol" en la notación ya no corresponde al nombre del extremo de la asociación, sino al nombre de la interfaz.
- No hay ninguna meta-asociación explícita entre `Interface` y `Classifier` que represente la relación de realización, ya que ésta técnicamente es una dependencia de abstracción (*abstraction dependency*) con el estereotipo «`realize`» [UML, p. 3-52], y por tanto no precisa una meta-asociación propia para ser representada[129].

Figura 5.15. Metamodelo correspondiente a la propuesta de sustituir clasificadores por interfaces en los extremos de asociación

5.4.3. Simplificación de la visibilidad de atributos y operaciones

Como hemos mencionado más arriba, *la interfaz siempre existe* en el extremo de la asociación, *aunque no se muestre* en un diagrama concreto. Si la interfaz no se define explícitamente en algún lugar del modelo, entonces hay que suponer una *interfaz implícita* que permite el acceso a todas las

[129] Esta meta-asociación sí es explícita en la solución de Steimann, que además introduce otras variaciones para intentar resolver la generalización de asociaciones [Steimann 00, Steimann 01], problema que no ha sido tratado en esta Tesis Doctoral.

propiedades públicas de la clase conectada (la "interfaz nativa"). Nótese que esta interfaz nativa implícita *no proporciona un acceso completo* en el sentido explicado en el Apartado 5.3.1, es decir, acceso a las propiedades y asociaciones del clasificador destino que sean visibles desde el clasificador origen, que pueden tener cualquier visibilidad, incluso privada, en el caso de asociaciones reflexivas. Por el contrario, en nuestra propuesta hemos restringido el acceso en todo caso (con interfaz implícita o explícita) a las *propiedades y asociaciones públicas*, excluyendo el resto de visibilidades.

La ventaja más evidente de adoptar esta decisión es que desaparecerían las *paradojas del acceso a propiedades privadas y protegidas* a través de asociaciones reflexivas, ya que solamente sería posible el acceso a través de una asociación a las propiedades incluidas en la interfaz correspondiente. Por tanto, un objeto nunca tendría acceso a las propiedades privadas o protegidas de otro objeto, ni siquiera si es de su misma clase.

Esta decisión también implica que las propiedades de paquete no serían accesibles de ninguna manera para otras clases, es decir, la *visibilidad de paquete* prácticamente dejaría de tener sentido para las propiedades de un clasificador. En realidad, el uso adecuado de interfaces hace innecesaria la distinción de tantos tipos de visibilidad en las propiedades de los clasificadores. En efecto, bastaría con especificar que un atributo u operación es *privado o no*. Si es privado, no es visible más que para el clasificador propietario. Si no es privado, es visible para los descendientes, y para otros clasificadores a través de las asociaciones que lo incluyan en su interfaz.

En este enfoque las visibilidades `public` y `package` son sustituidas, con ventaja, por la inclusión de la propiedad correspondiente en las interfaces que lo requieran; la visibilidad `protected` equivale a no incluir la propiedad en ninguna interfaz y no declararla privada; y la visibilidad `private` sería equivalente a la definición actual, excluyendo el acceso a través de asociaciones reflexivas.

Así, en el ejemplo de la **Figura 5.16** solamente se ha especificado la visibilidad privada de la propiedad `e`, que así es visible únicamente para la clase propietaria `Z` (es más, *sólo para la instancia propietaria*, ya que otras instancias de la misma clase accederían a través de una asociación reflexiva que requeriría el uso de interfaces). Las propiedades `a`, `b` y `c` son visibles a otras clases asociadas a través de las correspondientes interfaces `ab` y `ac`, por lo que pueden tener la consideración de propiedades en cierto modo "públicas". La propiedad `d` no es privada ni tampoco es visible a través ninguna interfaz, de modo que puede considerarse "protegida", es decir, visible para los descendientes pero no para otras clases asociadas (las propiedades `a`, `b` y `c` también son visibles para los descendientes).

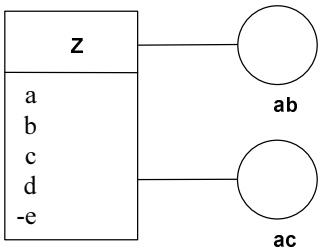

Figura 5.16. Simplificación de la visibilidad de propiedades

5.4.4. Interfaces con múltiples asociaciones

Para que una interfaz pueda conectarse a asociaciones bidireccionales ha sido necesario incluir la especificación de estructura (atributos y asociaciones navegables) dentro de ella. Esto implica que una interfaz puede *especificar una asociación hacia otra interfaz*, no sólo la que está en el extremo opuesto de la asociación original.

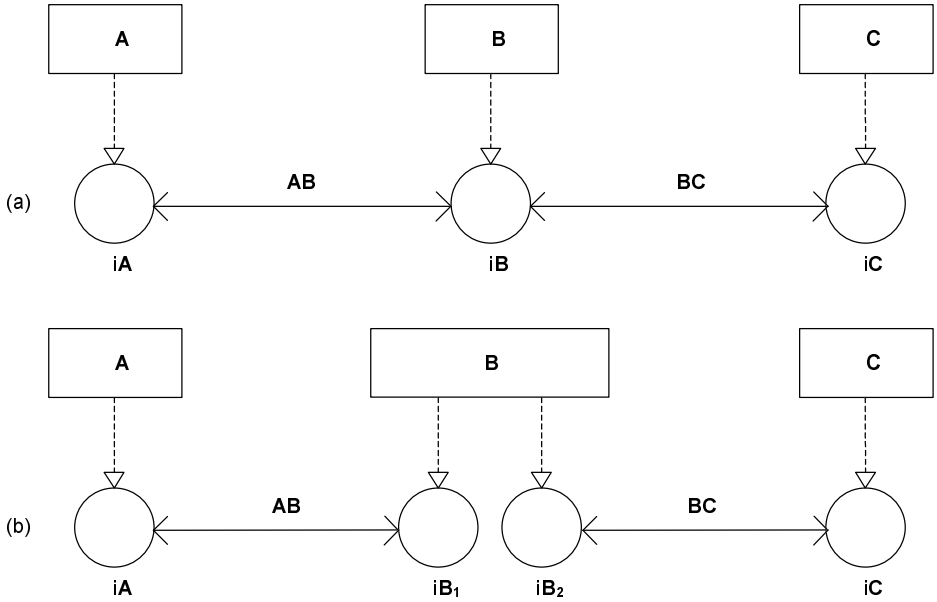

Figura 5.17. Participación de una interfaz en más de una asociación: a) `iA` e `iC` se ven mutuamente a través de `iB`; b) `iA` e `iC` ya no se ven porque iB_1 e iB_2 separan la participación de `B` en las dos asociaciones

Consideremos en la **Figura 5.17** (a) la asociación bidireccional `AB` definida entre las interfaces `iA` e `iB`, y la asociación `BC` definida entre `iB` e `iC`. Sean `A`, `B` y `C` las clases que implementan `iA`, `iB` e `iC` respectivamente. La interfaz `iB` está conectada a dos asociaciones distintas, `AB` y `BC`, lo que implica que:

- La clase B debe implementar o realizar el comportamiento y estado especificado en la interfaz iB, lo que incluye las asociaciones AB y BC, además de otros atributos y operaciones especificados por iB.
- La interfaz iA ve todo el contenido de iB, por tanto iA ve iC a través de la asociación BC. Y viceversa para iC.
- La interfaz iB, entendida como rol, ya no significa "cómo una clase participa en *una* asociación" (AB o BC), sino más bien "cómo una clase participa en *una o más* asociaciones" (AB y BC).

Por el contrario, en la **Figura 5.17** (b) las asociaciones AB y BC están conectadas a dos interfaces distintas realizadas por la clase B, denominadas iB_1 e iB_2, que separan la participación de B en las dos asociaciones. En este caso las interfaces iA e iC ya no pueden verse mutuamente a través de un elemento común.

Esta diferencia es sin duda importante, pero la notación propuesta es suficientemente expresiva como para que no pase desapercibida. Por el contrario, si se usa el estilo más compacto de la **Figura 5.18**, en el que las interfaces se expresan como nombres de rol, puede ser más difícil advertir las implicaciones de conectar una asociación a la misma o a dos interfaces distintas.

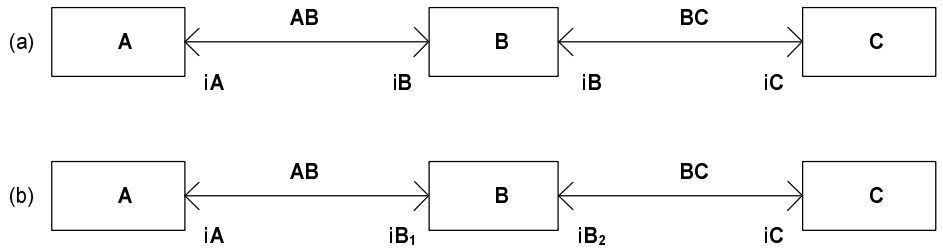

Figura 5.18. El mismo ejemplo de la figura precedente, expresado con notación más compacta. Nótese la crucial importancia de los nombres de rol que representan interfaces: a) el mismo nombre iB, visibilidad mutua entre A y C; b) nombres distintos iB_1 e iB_2, no hay visibilidad entre A y C

TERCERA PARTE:

EXPERIMENTACIÓN

6. La implementación de las asociaciones

6.1. Introducción

Este Capítulo concluye la investigación realizada en esta Tesis Doctoral con la experimentación de las ideas examinadas en los Capítulos precedentes, mediante su aplicación a la implementación de asociaciones. En él nos planteamos dos objetivos principales y un tercer objetivo que queda para futuros trabajos. En primer lugar, escribir una serie de *plantillas de código* que ayuden a los programadores a traducir las asociaciones UML al lenguaje orientado a objetos destino. El lenguaje escogido es Java, aunque los principios que hemos seguido son aplicables a otros lenguajes próximos como C++ o el entorno .NET. En segundo lugar, construir una *herramienta* que usando estas plantillas genere código para las asociaciones, que serán leídas de un modelo almacenado en formato XMI[130]. Un tercer objetivo será posibilitar la *ingeniería inversa*, es decir, obtener las asociaciones entre clases a partir del análisis del código que las implementa. Nuestra herramienta no lleva a cabo actualmente esta tarea, aunque es muy sencilla y directa si el código ha sido escrito con nuestras plantillas[131].

Las asociaciones en UML pueden tener una gran variedad de propiedades. El presente Capítulo se limita al análisis e implementación de las propiedades de multiplicidad, navegabilidad y visibilidad en asociaciones binarias. Excluye, por tanto, algunos tipos más complejos de asociaciones como asociaciones reflexivas, asociaciones todo/parte (agregaciones y composiciones), asociaciones cualificadas, clases-asociación, y asociaciones n-arias. Excluye también propiedades tales como ordenación (*ordering*), modificabilidad (*changeability*), etc.

El resto de este Capítulo está organizado como se describe a continuación. Las Secciones 6.2, 6.3 y 6.4 están dedicadas al estudio de las propiedades de multiplicidad, navegabilidad y visibilidad de las asociaciones, con un análisis detallado de los posibles problemas y

---

[130] *XML Metadata Interchange* [XMI], un formato basado en XML diseñado para almacenar e intercambiar modelos UML entre distintas herramientas.

[131] La herramienta ha sido desarrollada, de acuerdo con los principios presentados en este Capítulo, en el ámbito de un Proyecto Fin de Carrera dirigido por el autor de esta Tesis Doctoral. La descripción completa de las plantillas de código y de la herramienta puede encontrarse en la documentación de dicho Proyecto [Ruiz 02]. El núcleo del Capítulo corresponde, además, a un trabajo publicado por el autor de esta Tesis Doctoral [Génova 03b].

soluciones propuestas. La Sección 6.5 contiene la descripción de una interfaz uniforme para todos los tipos de asociaciones desde el punto de vista de las clases participantes, tal como son implementadas con nuestras plantillas y herramientas. Finalmente, la Sección 6.6 describe brevemente cómo funciona nuestra herramienta.

6.2. Experimentación con la multiplicidad

Recordemos la definición de multiplicidad: la multiplicidad de una asociación binaria, situada junto al extremo de la asociación (el extremo destino), especifica el número de instancias destino que pueden estar asociadas a una única instancia origen a través de la asociación en cuestión, en otras palabras, cuántos objetos de una clase (la clase destino) pueden estar asociados con un único objeto dado de la otra clase (la clase origen) [RM, p. 348; UML, p. 2-23]. El clásico ejemplo de la **Figura 6.1**, que ya hemos empleado anteriormente, ilustra la multiplicidad binaria. Cada instancia de `Persona` puede trabajar para cero o una instancias de `Empresa` (0..1), mientras que cada empresa puede estar enlazada a una o más personas (1..*). Nuestro análisis quedará restringido a multiplicidades que puedan expresarse como un único intervalo entero en la forma `(min..max)`, aunque en UML las expresiones de multiplicidad pueden ser más complejas.

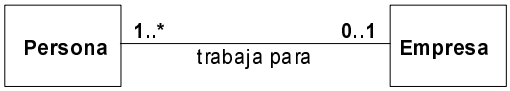

Figura 6.1. Un clásico ejemplo de asociación binaria con la expresión de multiplicidades

La restricción de multiplicidad es un tipo de *invariante*, es decir, una condición que debe ser satisfecha por el sistema. Una posible práctica al programar es no comprobar el invariante en todo momento, sino sólo cuando lo pida el programador, después de completar un conjunto de operaciones que se supone dejan el sistema en un estado válido (una *transacción*). Esta práctica es más eficiente en tiempo de ejecución, y da al programador más libertad y responsabilidad en la escritura del código, con el correspondiente riesgo de que se olvide de incluir las necesarias comprobaciones y deje al sistema por descuido en un estado erróneo. Por otra parte, pensamos que comprobar las restricciones de multiplicidad no es excesivamente ineficiente en el consumo de tiempo, especialmente cuando se compara con el tiempo requerido para gestionar colecciones o sincronizar asociaciones bidireccionales (ver Apartado 6.3.2). Así pues, pensamos que vale la pena hacer todo lo que podamos por el programador, de modo que el primer objetivo de este Capítulo es analizar la posibilidad de realizar comprobaciones automáticas para las restricciones de multiplicidad.

6.2.1. Asociaciones opcionales y obligatorias

El valor de la multiplicidad mínima puede ser cualquier entero positivo, aunque los valores más comunes son 0 y 1. Cuando el valor es 0 decimos que la asociación es opcional (*optional*) para la clase en el extremo opuesto (clase `Persona` en la **Figura 6.1**), y decimos que es obligatoria (*mandatory*) cuando el valor es 1 o mayor (clase `Empresa`). Las asociaciones opcionales no presentan problemas especiales para la implementación, pero las obligatorias sí. Desde el punto de vista conceptual, un objeto que participa en una asociación obligatoria tiene que estar enlazado *en todo momento* con un objeto (o más) al otro lado de la asociación, en caso contrario el sistema está en un estado erróneo. En el ejemplo de la **Figura 6.1**, una instancia de `Empresa` necesita siempre una instancia de `Persona`. Por tanto, en el mismo momento en que se crea una instancia de `Empresa` hay que enlazarla a una instancia de `Persona`.

Esto puede ocurrir de tres formas distintas:

- Una instancia de `Empresa` es creada por una instancia de `Persona` y enlazada a su creador.
- Una instancia de `Empresa` es creada con una instancia de `Persona` suministrada como parámetro.
- Una instancia de `Empresa` es creada y solicita la creación de una instancia de `Persona`.

El tercer caso plantea problemas adicionales. La creación de una persona probablemente requerirá datos adicionales, como el nombre, la dirección, etc., y no parece muy razonable suministrarlos en la creación de una empresa. Este problema empeora si `Persona` tiene otras asociaciones obligatorias, por ejemplo con el país donde vive: si éste fuera el caso, la creación de una empresa requeriría suministrar datos para crear una persona, un país, etc.

La solución más obvia es permitir sólo las dos primeras formas de instanciación. Pero supóngase entonces que la asociación es obligatoria en los dos extremos. ¿Cuál de las dos instancias hay que crear primero? No tenemos una elección satisfactoria, ya que dejaríamos al sistema en un estado erróneo hasta que las dos creaciones hayan terminado. Podemos pensar en una *creación atómica* de las dos instancias, pero esto vale sólo para el caso más sencillo en el que sólo hay dos clases implicadas. ¿Tendremos que definir creaciones atómicas para dos, tres, cualquier número de clases? En la destrucción de objetos surgen problemas similares.

Imaginemos ahora que no vamos a crear o destruir instancias, sino que vamos a cambiar enlaces entre instancias. Si queremos cambiar la instancia de `Empresa` que está enlazada con una determinada instancia de `Persona`, podemos borrar simplemente el enlace con la antigua empresa y añadir un nuevo enlace a la nueva empresa. Esto funciona siempre que la antigua empresa esté enlazada con otras instancias de `Persona`; incluso

podemos borrar el enlace y no añadir ninguno, ya que la asociación es opcional para `Persona`. Si sólo tuviéramos una persona enlazada a una empresa determinada, tendríamos que suministrar una nueva persona a la empresa antes de borrar el enlace con la antigua persona, pero esto no es más que el comportamiento especificado (la asociación es obligatoria para `Empresa`) y no podemos quejarnos por eso.

No obstante, aquí encontramos nuevos problemas. Si la asociación con `Empresa` fuera obligatoria también para `Persona` (es decir, multiplicidad 1..1 en lugar de la presente 0..1), la instancia de `Persona` no podría borrar el antiguo enlace con la empresa y luego añadir el nuevo enlace, ni podría hacerlo en el orden inverso, "primero añadir, luego borrar", porque pasaría por un estado erróneo del sistema. Un *cambio atómico* de enlaces sería válido sólo para los casos más sencillos, pero no para casos más complicados como el siguiente, que es más bien retorcido (ver **Figura 6.2**): consideremos las clases `A` y `B`, asociadas con multiplicidad 1..1 en ambos extremos, y las correspondientes instancias `a1`, `a2`, `b1` y `b2`. En el estado inicial tenemos los enlaces `a1-b1` y `a2-b2`. En el estado final queremos tener los enlaces `a1-b2` y `a2-b1`. Aunque pudiéramos cambiar atómicamente `a1-b1` por `a1-b2` sin violar la restricción de multiplicidad para `a1`, esto dejaría a `b1` sin ningún enlace y a `b2` con dos enlaces hasta que se alcanzase el estado final. Tendríamos que llevar a cabo todo el cambio de modo atómico por medio de un *trueque atómico* implementado en una única operación.

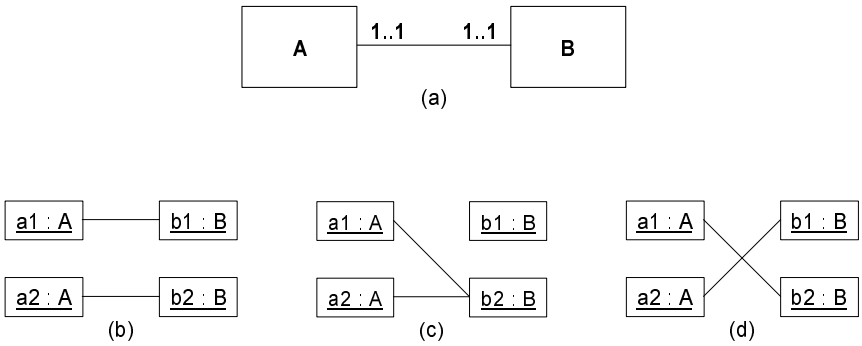

Figura 6.2. Las restricciones de multiplicidad pueden hacer muy difícil el cambio de enlaces entre instancias sin pasar por un estado erróneo del sistema: a) diagrama de clases; b) estado inicial; c) estado intermedio erróneo; d) estado final deseado

Evidentemente, no podemos definir una nueva operación para evitar cualquier estado erróneo concebible que implique varias instancias. En consecuencia, pensamos que las asociaciones obligatorias plantean problemas insolubles en lo que se refiere a la creación y destrucción de instancias y enlaces: no podemos conseguir con unas pocas operaciones

primitivas que una asociación obligatoria sea obedecida *en todo momento*, y no podemos aislar, dentro de operaciones atómicas, los momentos en los que la restricción no es obedecida. Por tanto, tenemos que relajar las implicaciones de las asociaciones obligatorias para la implementación, como hacen otros métodos [Harrison 00]. Nuestra propuesta es la siguiente: *no comprobar la restricción de multiplicidad mínima al modificar los enlaces de la asociación* (métodos mutadores), *sino sólo al acceder a los enlaces* (métodos lectores). El programador será responsable de usar las primitivas de modo consistente, de modo que se alcance cuanto antes un estado válido del sistema.

Por ejemplo, estará permitido crear una empresa sin enlazarla a ninguna persona, y estará permitido borrar todos los enlaces de una empresa con instancias de `Persona`; pero antes de acceder, con otros fines, a los enlaces de esa instancia particular de `Empresa` con instancias de `Persona`, será necesario restaurarlos a un estado válido; en caso contrario se obtendrá una excepción de multiplicidad inválida (*invalid multiplicity exception*), definida en el código que implemente las asociaciones de acuerdo con nuestra propuesta.

6.2.2. Asociaciones sencillas y múltiples

El valor de la multiplicidad máxima en un extremo de asociación puede ser cualquier entero mayor o igual que 1, aunque los valores más comunes son 1 y *. Cuando el valor es 1 decimos que la asociación es sencilla (*single*) para la clase en el extremo opuesto (clase `Persona` en la **Figura 6.1**), y decimos que es múltiple (*multiple*) cuando el valor es 2 o mayor (clase `Empresa`). Las asociaciones sencillas son más fáciles de implementar que las asociaciones múltiples: para almacenar la única instancia posible de una asociación sencilla habitualmente empleamos un atributo que tenga como tipo la correspondiente clase destino, pero para almacenar los muchos enlaces potenciales de una asociación múltiple tenemos que usar algún tipo de colección de objetos, tal como las predefinidas en Java `Vector`, `HashSet`, etc. En el caso general no podemos usar un *array* de objetos, porque su tamaño queda fijado al instanciarlo. Como las colecciones en Java pueden tener cualquier número de elementos, la restricción de multiplicidad máxima no puede ser establecida en la declaración de la colección en el código Java, de modo que habrá que comprobarla en algún otro lugar en tiempo de ejecución.

Necesitamos dos tipos de métodos mutadores, `add` (añadir) y `remove` (borrar), que aceptarán como parámetro un único objeto o una colección. Debido a los problemas con la multiplicidad mínima explicados más arriba, el *método borrador* a veces tendrá que dejar a la instancia origen en un estado erróneo; no podemos evitar esta situación. El *método añadidor*, en cambio, nos permite un cierto grado de elección. Si intentamos añadir algunos enlaces por encima de la restricción de multiplicidad máxima,

podemos elegir entre rechazar la adición o llevarla a cabo; en el último caso violamos temporalmente la restricción hasta que una llamada al *método borrador* restaure la instancia origen a un estado seguro; el estado erróneo sólo sería detectado por los *métodos lectores*, tal como establecimos en el caso de la multiplicidad mínima. Sin embargo, esto vale sólo para asociaciones múltiples implementadas mediante una colección; en asociaciones sencillas simplemente no podemos violar la restricción de multiplicidad máxima: estamos obligados a rechazar la adición.

Si decidimos rechazar la adición, en cambio, además de tener un comportamiento asimétrico entre el método borrador y el método añadidor, podemos encontrar problemas de precedencia al invocar el añadidor y el borrador uno detrás de otro. Consideremos la clase `Juego` asociada con la clase `Participante` con multiplicidad 2..4 (ver **Figura 6.3**), y supongamos que una instancia `j1` de `Juego` está enlazada con dos instancias `p1`, `p2` de `Participante`. Queremos reemplazar estos dos participantes por cuatro nuevos participantes diferentes, `q1`, `q2`, `q3`, `q4`. Si ejecutamos "primero borrar, luego añadir", obtenemos finalmente lo que queremos; si ejecutamos "primero añadir, luego borrar", la adición es rechazada y el borrado deja la instancia de `Juego` en un estado erróneo.

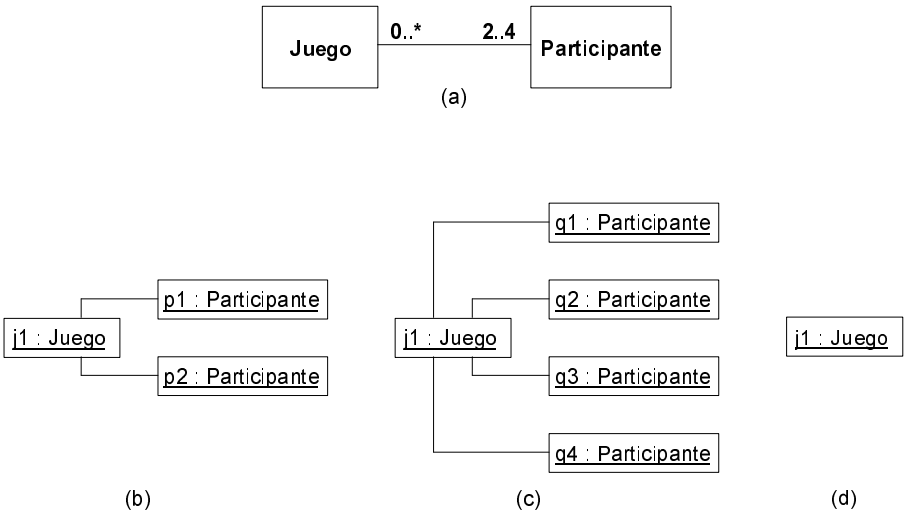

Figura 6.3. Problemas de precedencia al invocar sucesivamente el método añadidor y el método borrador: a) diagrama de clases de la asociación `Juego-Participante`; b) estado inicial con participantes `p1`, `p2`; c) estado final deseado después de borrar los participantes `p1`, `p2` y luego añadir los participantes `q1`, `q2`, `q3`, `q4`; d) estado final incorrecto después de intentar añadir sin éxito los participantes `q1`, `q2`, `q3`, `q4` y luego borrar los participantes `p1`, `p2`.

Al final hemos preferido *rechazar la adición si viola el máximo permitido*, y pedir a los usuarios de los métodos mutadores que los usen

siempre en el orden correcto, "primero borrar, luego añadir", de modo que obtengamos un comportamiento análogo para asociaciones sencillas y múltiples. Así pues, *el método borrador no comprueba la restricción de multiplicidad mínima* (tal vez dejando vacía una asociación obligatoria), *el método añadidor sí comprueba la restricción de multiplicidad máxima, y el método lector levanta una excepción si cualquiera de las restricciones no se satisface.*

Los métodos lectores de las asociaciones múltiples tienen otra peculiaridad, cuando se comparan con los de las asociaciones sencillas: devuelven una colección de objetos, no un único objeto, y por tanto el tipo del valor de retorno es el de la colección, no el de la clase destino. En nuestra implementación el tipo devuelto es la interfaz `Collection` de Java, implementada por todas las colecciones estándar. Internamente usamos una colección `HashSet`, que asegura que no hay enlaces duplicados en una asociación, tal como exige UML [UML, p. 2-19][132].

Finalmente, las colecciones estándar en Java están especificadas para contener instancias de la clase estándar `Object`, que es superclase de todas las demás clases en Java. No es posible especializar estas colecciones para que almacenen objetos que pertenezcan sólo a una clase determinada[133]. Esto significa que, si utilizamos un `HashSet` dentro de `Empresa` para almacenar los enlaces con `Persona`, debemos asegurar por nuestra cuenta que nadie pone un enlace a una instancia de otra clase como `Perro` o `Informe` (esto podría ocurrir si se pasa como parámetro una colección de objetos al método `add`). Así pues, los métodos mutadores deben realizar una comprobación de tipos por medio de un moldeado (*casting*) explícito en tiempo de ejecución. Si falla la comprobación de tipos, entonces no se actualiza el enlace a ese objeto, y se levanta una excepción de molde de clase (*class cast exception*), predefinida en Java.

6.3. Experimentación con la navegabilidad

Recordemos la definición de navegabilidad: la direccionalidad o navegabilidad de una asociación binaria, expresada gráficamente mediante una flecha abierta en el extremo de la línea de la asociación que conecta las dos clases, especifica la capacidad que tiene una instancia de la clase origen de acceder a las instancias de la clase destino por medio de las instancias de la asociación (enlaces) que las conectan[134]. Si la asociación puede ser

---

[132] Aunque en la Sección 3.5 hemos dado argumentos concluyentes en contra de esta restricción, en la generación de código hemos respetado la definición actual de UML.

[133] Es decir, no es posible especializarlas para modificar su estructura de almacenamiento, pero sí es posible modificar su funcionamiento para que, de hecho, sólo almacenen los objetos requeridos, precisamente mediante el método que describimos.

[134] Una definición alternativa: la posibilidad que tiene un objeto origen de designar un objeto destino a través de una instancia de asociación (enlace), con el fin de manipularlo

recorrida en ambas direcciones, entonces es bidireccional (*two-way*), en caso contrario es unidireccional (*one-way*).

Un extremo de asociación navegable, referenciado por su nombre de rol, define un pseudo-atributo de la clase origen, de modo que la instancia origen puede usar el nombre de rol en expresiones, del mismo modo que usa sus propios atributos [RM, p. 354]. Una instancia puede comunicarse (enviando mensajes) con las instancias conectadas en los extremos opuestos navegables, y puede usar referencias a éstas como argumentos o valores de respuesta en las comunicaciones [UML, p. 2-114]. De modo similar, si el extremo de asociación es navegable, la instancia origen puede consultar (*query*) y actualizar (*update*) los enlaces que la conectan a las instancias destino.

Los ejemplos de la **Figura 6.4** ilustran la navegabilidad. La asociación `Llave-Puerta` es unidireccional, lo que quiere decir que una llave puede acceder a la puerta que abre, pero una instancia de `Puerta` no conoce el conjunto de instancias de `Llave` que la abren: la puerta no puede recorrer las conexiones o enlaces en contra de la navegabilidad de la asociación. Por otra parte, la asociación `Hombre-Mujer` es bidireccional, lo que quiere decir que las instancias conectadas de estas clases se conocen mutuamente.

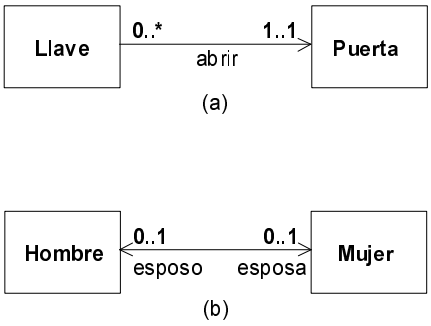

Figura 6.4. Ejemplos de a) asociación unidireccional y b) asociación bidireccional

Las puntas de flecha pueden mostrarse u omitirse en una asociación bidireccional [UML, p. 3-73]. Desafortunadamente, esto conduce a una ambigüedad en la notación gráfica, porque no podemos distinguir entre asociaciones bidireccionales y asociaciones sin navegabilidad especificada. O, peor aún, se asume que las asociaciones sin navegabilidad especificada son bidireccionales sin posterior análisis.

---

o acceder a él en una interacción con intercambio de mensajes. El Estándar no contiene una definición clara de navegabilidad, como hemos mostrado en el Capítulo 4. Nótese que seguimos tomando navegabilidad y direccionalidad como sinónimos.

6.3.1. Asociaciones unidireccionales

Una *asociación unidireccional sencilla* es muy similar a un atributo de valor único (*single valued attribute*) en la clase origen, del tipo de la clase destino: una referencia incrustada, puntero, o como quiera denominárselo. La equivalencia, no obstante, no es completa. Mientras que el *valor del atributo* es "poseído" por la instancia de la clase y no tiene identidad, un *objeto externo referenciado* tiene identidad y puede ser compartido por instancias de otras clases que tengan referencias a ese mismo objeto [Rumbaugh 96b] (ver **Figura 6.5**). En todo caso, la equivalencia es suficientemente satisfactoria como para servir de base para la implementación de este tipo de asociaciones. En realidad, en Java no hay ninguna diferencia: excepto en el caso de los valores primitivos, los atributos en Java son objetos con identidad, y si son públicos no se puede evitar que sean referenciados y compartidos por otros objetos.

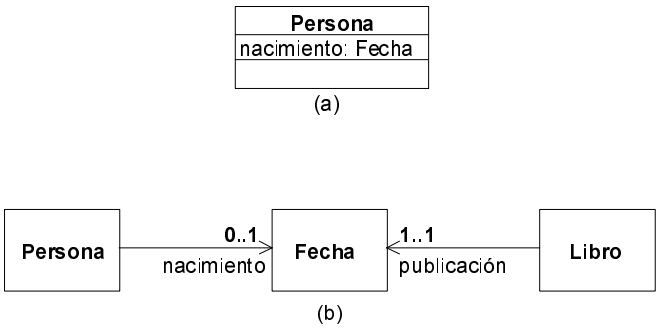

Figura 6.5. Equivalencia parcial entre a) atributo y b) asociación unidireccional sencilla

Una *asociación unidireccional múltiple* es algo más complicada, aunque la implementación puede basarse en los mismos principios, ya que puede asimilarse a un atributo multivaluado (*multivalued attribute*)[135]. Sin embargo, para gestionar la colección de objetos en el extremo navegable necesitamos un objeto adicional de una clase colección estándar, que en nuestra implementación es la clase `HashSet` (ver **Figura 6.6**).

[135] UML permite la multiplicidad en atributos, y por tanto los atributos multivaluados [UML, p. 2-50].

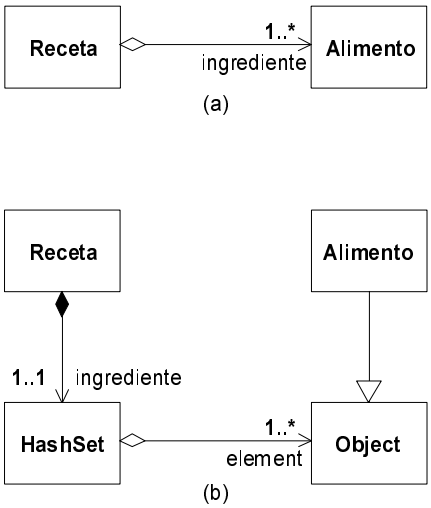

Figura 6.6. Asociación unidireccional múltiple: a) diagrama de análisis y b) diagrama de diseño. Es necesario insertar un nuevo objeto para gestionar la colección de objetos destino. Las colecciones estándar en Java, tales como `HashSet`, están definidas para la clase estándar `Object`, que es superclase de todas las clases; por tanto, los métodos mutadores deben asegurar que los objetos contenidos en el parámetro colección son del tipo apropiado antes de añadirlos al atributo colección.

Así pues, el tipo del atributo usado para implementar la asociación dentro de la clase origen ya no es la clase destino misma, sino la clase `HashSet` u otra clase colección conveniente. Los métodos para gestionar la asociación tendrán que llevar al cabo algunas tareas adicionales. Los *métodos mutadores* pueden añadir o borrar no sólo objetos singulares de la clase destino, sino también enteras colecciones; así, el tipo del parámetro será o bien la clase destino de la asociación o bien la clase colección intermedia. En este caso, los métodos mutadores deben asegurar que los objetos contenidos en el parámetro colección son del tipo apropiado antes de añadirlos al atributo colección. Los *métodos lectores*, como ya hemos explicado (ver Apartado 6.2.2), no devuelven un objeto singular, sino una colección de objetos, aunque esté formada por un solo elemento. El objeto colección devuelto no es idénticamente el mismo que el que está almacenado dentro de la clase origen, sino un clon (un nuevo objeto con una colección de referencias a los mismos elementos destino), porque el objeto colección original debe permanecer completamente encapsulado dentro del objeto origen (esto se representa como una composición en la **Figura 7.6**).

Como muestran los diagramas de la **Figura 7.4** y la **Figura 6.5**, en nuestra opinión *la restricción de multiplicidad en un diagrama de diseño*

sólo puede especificarse en el extremo de asociación navegable[136]. En efecto, la multiplicidad es una restricción que debe ser evaluada en el contexto de la clase que posee el extremo de la asociación; si esta clase conoce la restricción, entonces conoce el extremo de asociación, es decir, el extremo es navegable. No se puede restringir el número de objetos conectados a una instancia dada a menos que esta instancia tenga algún conocimiento de los objetos conectados, es decir, a menos que se haga navegable el extremo de asociación. Por tanto, *la necesidad de una restricción de multiplicidad distinta de 0..** (es decir, no restringida) *es una indicación de que el extremo de asociación debe ser navegable*. En consecuencia, las asociaciones unidireccionales con restricciones de multiplicidad en el extremo de asociación no navegable deben ser rechazadas en la generación de código.

6.3.2. Asociaciones bidireccionales

La equivalencia parcial entre atributos y asociaciones unidireccionales ya no la encontramos entre las asociaciones bidireccionales. En su lugar, una instancia de una asociación bidireccional es más parecida a una *tupla de elementos* [UML, p. 2-19]. Combinando las multiplicidades en ambos extremos de asociación, podemos tener tres casos: sencilla-sencilla, sencilla-múltiple, y múltiple-múltiple.

Una forma fácil de implementar una asociación bidireccional *sencilla-sencilla* es por medio de dos asociaciones unidireccionales sencillas sincronizadas (ver **Figura 6.7**). La sincronización de las dos mitades debe ser preservada por los métodos mutadores en cada lado: cada vez que se solicita una actualización en un lado, el lado opuesto debe ser informado para realizar la correspondiente actualización; ésta es llevada a cabo sólo si ambos lados están de acuerdo en que pueden ejecutarla manteniendo las restricciones de multiplicidad máxima[137] (ver **Figura 6.8**).

---

[136] Este principio no es aplicable a los modelos de análisis, que a menudo no tratan con la navegabilidad [Fowler 00, Stevens 00]. Evidentemente, la generación de código sólo tiene sentido a partir de diagramas de diseño.

[137] Ya hemos justificado que debe permitirse a los métodos mutadores que violen la restricción de multiplicidad mínima, ver Apartado 6.2.1.

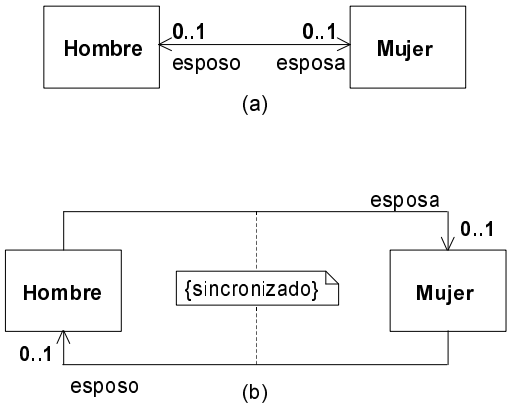

Figura 6.7. Asociación bidireccional sencilla-sencilla: a) diagrama de análisis y b) diagrama de diseño. La implementación de los métodos mutadores de la asociación debe asegurar que el marido de la esposa de un cierto hombre es ese hombre mismo, y viceversa

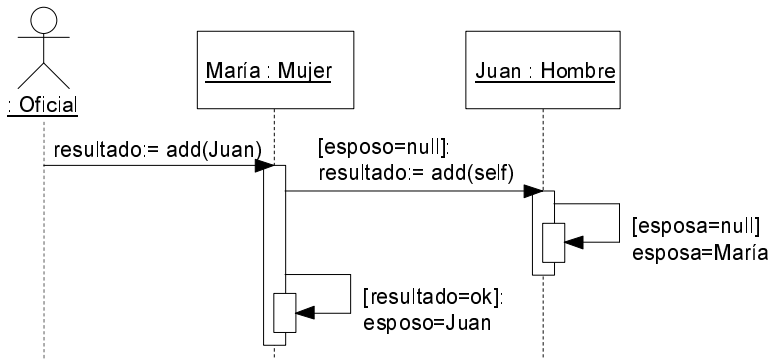

Figura 6.8. Diagrama de secuencia que ilustra la sincronización de una asociación bidireccional. La actualización del atributo `Mujer.marido` a `Juan` (última operación) sólo tiene lugar después de que la actualización del atributo `Hombre.esposa` ha sido correctamente realizada. Si la mujer ya estuviera casada, entonces no solicitaría al hombre que actualizara la asociación de matrimonio en su lado; si la actualización en el lado del hombre falla (porque ya está casado), entonces la mujer no actualiza su lado. Para lograr este comportamiento, el método `add` devuelve un resultado conveniente que es comprobado por el objeto cliente

Una asociación bidireccional *sencilla-múltiple* puede implementarse de modo similar, combinando una asociación unidireccional sencilla con una asociación unidireccional múltiple. Y, finalmente, una asociación bidireccional *múltiple-múltiple* se consigue mediante dos asociaciones unidireccionales múltiples (ver **Figura 6.9**).

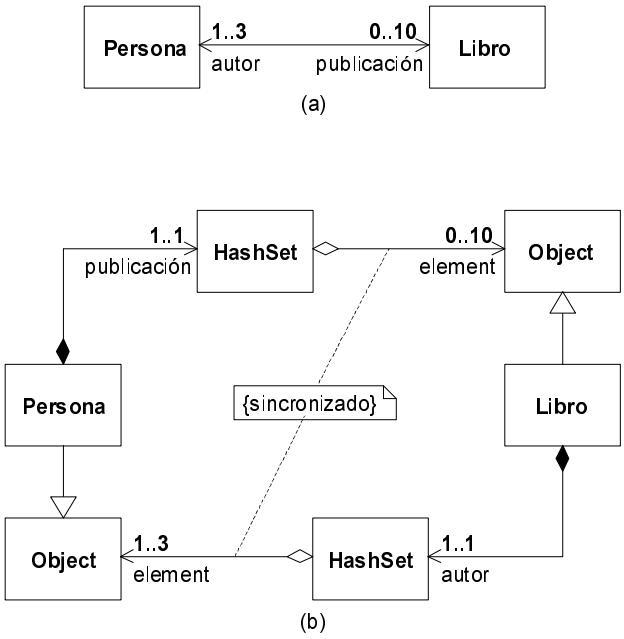

Figura 6.9. Asociación bidireccional múltiple-múltiple: a) diagrama de análisis y b) diagrama de diseño

La sincronización se vuelve progresivamente más complicada cuando uno o dos de los extremos de asociación es múltiple. Consideremos el ejemplo de la **Figura 6.9**. Supongamos que queremos añadir un autor a cierta instancia de `Libro`; para ello invocamos el método `add` en la instancia de `Libro`, pasando una instancia de `Persona` como parámetro. Si el libro puede tener más autores sin violar su multiplicidad máxima (que es 3), entonces solicita al autor que añada el libro mismo a la colección de publicaciones de la persona; esto puede fallar si la restricción de multiplicidad máxima para el número de publicaciones (en este caso 10) es violada. Si la solicitud al autor tiene éxito, entonces el libro actualiza su lado.

Ahora bien, podemos intentar también añadir una colección de autores a un libro. Como podríamos esperar, el libro solicita a cada uno de los autores que añada el libro mismo como publicación; si cualquiera de los autores falla en añadir el libro, entonces toda la operación debe ser deshecha, ya que una actualización debe ser atómica: todo o nada.

Consideraciones similares afectan al método mutador `remove`, teniendo en cuenta que éste es ejecutado aunque la restricción de multiplicidad mínima no sea satisfecha, y por tanto puede dejar a la instancia origen o alguna de las instancias destino afectadas en un estado inválido.

En UML una asociación se define como un "conjunto de tuplas" [UML, p. 2-19], lo que quiere decir que no podemos tener dos veces la misma tupla en la colección de enlaces de una asociación[138]. Esto queda automáticamente salvaguardado si seguimos el esquema de implementación explicado más arriba. En todo caso, también sugiere un tipo diferente de implementación que podría tener algunas ventajas. En lugar de sincronizar dos asociaciones unidireccionales para obtener una bidireccional, podemos almacenar directamente la colección de enlaces bidireccionales (ver **Figura 6.10**).

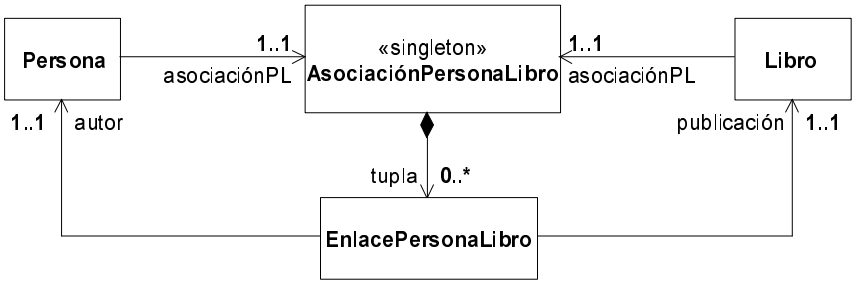

Figura 6.10. Un esquema alternativo para implementar asociaciones bidireccionales por medio de una colección de tuplas "cosificadas"

En este esquema alternativo los enlaces están "cosificados" y se transforman en objetos en sí mismos [Rumbaugh 87]. Para gestionar la colección de enlaces, o tuplas, necesitamos un objeto, que sea la única instancia de una clase (aplicando el patrón de diseño *Singleton* [Gamma 94]) y que representa a la asociación misma. La principal ventaja de este enfoque es que evita la dispersión de la información acerca de las instancias de la asociación (enlaces), de modo que las actualizaciones se efectúan en un único sitio, sin problemas de sincronización. Se puede extender fácilmente para implementar clases-asociación y asociaciones de mayor grado (ternarias, etc.). Sin embargo, estas ventajas tienen un alto coste, como podemos apreciar comparando la **Figura 6.9** y la **Figura 6.10**.

Nótese que *las restricciones de multiplicidad originales no están expresadas en este esquema*: la multiplicidad de los roles `Persona.asociaciónPL` y `Libro.asociaciónPL` deben ser obviamente 1..1, ya que sólo hay una instancia del objeto que gestiona la asociación considerada como colección de enlaces; además, un enlace es la conexión de dos instancias, por tanto un enlace tiene exactamente una "pata" en cada lado (ver Sección 3.3), es decir, las multiplicidades deben ser 1..1 en los roles `autor` y `publicación`; finalmente, el rol `tupla` tiene multiplicidad 0..* sea cual sea la multiplicidad de la asociación original,

[138] La inconveniencia de esta restricción, heredada del Modelo Entidad/Relación, ya ha sido discutida en detalle en la Sección 3.5.

aunque fuera sencilla-sencilla, porque almacena todos los enlaces que pueden existir entre cualesquiera dos instancias en cada lado. En consecuencia, las restricciones de multiplicidad resultan más difíciles de mantener, puesto que el control no puede consistir simplemente en "contar enlaces".

Otro problema que encontramos es que *la unicidad de cada tupla, exigida por la restricción "conjunto de tuplas", no queda automáticamente salvaguardada*. Supongamos que implementamos la colección de tuplas mediante un `HashSet` de objetos, donde cada objeto almacena dos referencias, `autor` y `publicación`. Como cada objeto tupla tiene su propia identidad, dos tuplas diferentes que referencien los mismos dos objetos destino serían consideradas como objetos diferentes, de modo que la colección `HashSet` no comprobaría por nosotros la unicidad de cada tupla[139].

Considerando todos estos factores, en nuestra implementación hemos descartado el enfoque de "tuplas cosificadas" en favor del anterior esquema de "referencias cruzadas sincronizadas". En futuros trabajos se afrontarán otros enfoques.

6.4. Experimentación con la visibilidad

Hasta ahora hemos tratado sólo con la implementación en Java de dos propiedades de las asociaciones UML: multiplicidad y navegabilidad (o direccionalidad), pero también estamos interesados en la implementación de la visibilidad. Recordemos su definición: la visibilidad de un extremo de asociación especifica la visibilidad de la asociación desde el punto de vista de otros clasificadores al navegar la asociación en dirección hacia ese extremo[140]. El Estándar asimila la visibilidad de un extremo de asociación a la visibilidad de un atributo, y da las mismas cuatro posibilidades [UML, p. 2-23]:

- `public` - otros clasificadores pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo público.
- `protected` - los descendientes del clasificador origen pueden navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo protegido.

---

[139] En todo caso, no es una dificultad insalvable. Para que dos tuplas "iguales" sean reconocidas como tales y así garantizar su unicidad, es necesario redefinir los métodos `equals` y `hashCode`, heredados de la clase `Object`, y empleados por la colección `HashSet` con este fin [Eckel 00].

[140] Ésta es nuestra definición. La definición que da el Estándar no es suficientemente precisa, como ya hemos argumentado en el Capítulo precedente (ver Apartado 5.2.2).

- `private` - sólo el clasificador origen puede navegar la asociación y usar el nombre de rol en expresiones, de modo similar al uso de un atributo privado.
- `package` - los clasificadores que están en el mismo paquete (o paquete anidado, hasta cualquier nivel) que la declaración de la asociación pueden navegar la asociación y usar el nombre de rol en expresiones[141].

En Java encontramos los mismos cuatro tipos de visibilidad para atributos y métodos (no es casualidad, por supuesto), denominados niveles de control de acceso (*access control levels*), aunque su significado no es exactamente el mismo que en UML[142]. La opción por defecto, obtenida cuando no se especifica el control de acceso, es la visibilidad de paquete (habitualmente conocida en Java como `friendly`). Como hemos implementado las asociaciones UML mediante atributos y métodos Java, parece que no deberíamos encontrar especiales problemas con la implementación de la visibilidad[143]; al contrario, debería ser más bien fácil.

Esto es verdad para asociaciones unidireccionales: si declaramos los atributos y métodos Java con el mismo control de acceso que el extremo de asociación UML que queremos implementar, automáticamente obtenemos el comportamiento deseado. Pero la historia es diferente para asociaciones bidireccionales. En principio, parece razonable declarar privado uno o dos de los extremos de una asociación binaria. Podemos pensar en una asociación con dos extremos privados como una relación "secreta" que no es conocida fuera de las clases participantes, tal como una asociación `Banco-Cliente`, por ejemplo. De modo similar, una asociación con un extremo público y otro privado sería sólo parcialmente conocida desde el exterior. Pero hay problemas. Consideremos la asociación bidireccional `Profesor-Materia` con visibilidades pública y privada (ver **Figura 6.11**).

---

[141] Ya hemos visto los problemas que plantea la definición de este último tipo de visibilidad en el Apartado 5.2.2, donde hemos sugerido esta otra redacción (adiciones en cursiva): "Los clasificadores que están en el mismo paquete (o paquete anidado, hasta cualquier nivel) que *el clasificador origen* pueden navegar la asociación y usar el nombre de rol en expresiones, *de modo similar al uso de un atributo con visibilidad de paquete*".

[142] El control de acceso `protected` significa en Java la unión de las visibilidades `protected` y `package` de UML, es decir, el elemento protegido es accesible tanto a los descendientes como a otros elementos del mismo paquete [Arnold 00, Eckel 00, Gosling 96, Lemay 00]. Ver Apartado 5.2.1.

[143] Salvo que el significado de `protected` será el de Java, no el de UML. El Estándar reconoce que todas las formas de visibilidad no pública son de hecho dependientes de la implementación [UML, p. 3-42].

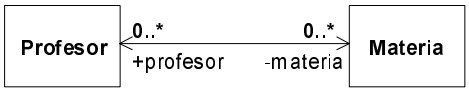

Figura 6.11. Una asociación bidireccional con un extremo de asociación público y otro privado

El extremo de asociación público, `profesor`, puede ser usado por cualquier otra clase en el modelo que tenga visibilidad hacia la clase `Materia`, es decir, la colección de profesores que enseñan una determinada materia puede ser consultada y actualizada directamente por cualquier clase en el modelo que vea la materia. En cambio, el extremo de asociación privado, `materia`, que significa la colección de materias que enseña un determinado profesor, es conocido sólo por el profesor mismo, exactamente igual que un atributo privado. La clase `Profesor` podría declarar públicos otros métodos que internamente se refirieran al nombre de rol `materia`, proporcionando de esta manera *acceso indirecto* al extremo de asociación privado, pero *el acceso directo está restringido a la clase poseedora misma*. Esto no es más que la idea de declarar privada alguna cosa[144].

Ahora bien, aquí tenemos una paradoja en relación con la bidireccionalidad de la asociación. El extremo de asociación con el nombre de rol privado `materia` sólo es conocido por su poseedor, es decir, la clase `Profesor`. Repetimos, *sólo por su poseedor*. ¡Esto significa que la clase `Materia` no conoce el extremo de asociación `materia`! La clase `Materia` conoce que está asociada con la clase `Profesor`, pero no conoce que la clase `Profesor` está a su vez asociada con ella. *¿Es esto realmente una asociación bidireccional?*

En nuestra implementación, basada en referencias cruzadas sincronizadas como hemos explicado más arriba, esta paradoja se manifiesta en la imposibilidad de actualización recíproca de los extremos de asociación. Recuérdese que, cuando una instancia de `Materia` intenta añadir una instancia `nuevoProfesor` de la clase `Profesor` a su colección de profesores (`profesor.add(nuevoProfesor)`), primero tiene que invocar el método `add` en el lado recíproco (`profesor.materia.add(self)`)[145]; pero esto ahora resulta imposible debido a la visibilidad privada del extremo de asociación

[144] Recuérdese que en UML la visibilidad de los atributos y operaciones es una característica especificada para los clasificadores, no para las instancias, de modo que un objeto puede ver las propiedades privadas de otro objeto de la misma clase, y análogamente sus extremos de asociación. Ver Apartado 5.2.1.

[145] Un objeto se refiere a sí mismo en UML/OCL mediante la palabra clave `self`; el equivalente en Java es `this`.

`materia`. Lo mismo ocurre con el método `remove`. Por el contrario, si una instancia de `Profesor` intenta actualizar la asociación, puede invocar el método de actualización en el lado opuesto, porque es público, y puede actualizar su propio lado privado, de modo que toda la operación tiene éxito. A primera vista, pues, parecía que la asociación podía ser gestionada a través de la clase poseedora del extremo de asociación público (en este caso, la clase `Materia`), pero esto ha resultado ser falso: en realidad, sólo la clase que posee el extremo de asociación privado (`Profesor`) puede gestionar la asociación, y el acceso directo desde fuera de las dos clases participantes es imposible. No obstante, como hemos explicado más arriba, la clase `Profesor` podría declarar métodos públicos para proporcionar acceso indirecto al extremo de asociación privado desde el exterior.

Aún peor, si ambos extremos de asociación fueran privados, como en el ejemplo `Banco-Cliente`, la asociación resultaría inaccesible desde ambos lados[146]. El enfoque basado en "tuplas cosificadas" tampoco soluciona el problema, ya que implica clases auxiliares que no pueden proporcionar acceso "privado" a las clases principales, excluyendo a todas las demás clases.

Resumiendo, *una asociación bidireccional pública-privada puede ser gestionada sólo desde la clase que posee el extremo privado*, y todas las demás clases, incluyendo la clase al otro extremo de la asociación, sólo pueden tener acceso indirecto si esta clase proporciona los métodos públicos adecuados. *Una asociación bidireccional privada-privada, por el contrario, no puede ser gestionada de ninguna manera*. Se pueden hacer consideraciones similares para las visibilidades de paquete y protegida, que se comportan a estos efectos respectivamente como las visibilidades pública y privada. En consecuencia, las asociaciones bidireccionales con visibilidad distinta de `public` o `package` en ambos extremos deben ser rechazadas en la generación de código.

Pensamos que este resultado no sólo está influenciado por nuestra particular implementación, sino que se trata de una *dificultad semántica real de la definición de visibilidad en asociaciones bidireccionales*. La visibilidad en UML no se especifica para asociaciones sino para extremos de asociación, y está asimilada a la visibilidad de los atributos [UML, p. 2-23]. Necesitamos en UML una definición de visibilidad que encaje mejor con el concepto de asociación bidireccional.

---

[146] Una asociación reflexiva (una asociación entre instancias de la misma clase) es una excepción a esta regla, ya que los extremos de asociación privados son visibles dentro de la clase, es decir, para ambos lados de la asociación, salvo que se usen interfaces en los extremos, como hemos propuesto en la Sección 5.4.

6.5. Descripción del código que implementa la solución

El código requerido para implementar una asociación UML consiste básicamente en una adecuada combinación de atributos y métodos Java, pero esta combinación depende de la multiplicidad, navegabilidad y visibilidad de la asociación. Con respecto a la *multiplicidad*, para almacenar la única instancia posible de una asociación sencilla podemos emplear un atributo de la correspondiente clase destino, pero para almacenar los muchos enlaces potenciales de una asociación múltiple necesitamos algún tipo de colección de objetos: hemos elegido `HashSet`, porque asegura que no hay objetos duplicados en la colección. Con respecto a la *navegabilidad*, una asociación unidireccional se implementa sólo en la clase origen, mientras que una asociación bidireccional se implementa en las dos clases, con código que asegura la sincronización de ambos extremos. La *visibilidad* de los extremos de asociación se traduce directamente en la visibilidad de los métodos requeridos; los atributos, en cambio, serán siempre privados, para mantener el acceso controlado a través de los métodos de la interfaz.

Hemos diseñado una *interfaz uniforme* para todo tipo de asociaciones, es decir, una interfaz tan independiente como sea posible de la multiplicidad, navegabilidad y visibilidad de los extremos de asociación. La interfaz comprende métodos lectores y mutadores, así como otros métodos auxiliares para averiguar el estado y definición de la asociación. Nuestra intención es que el código cliente pueda usar la interfaz de la asociación sin conocer *a priori*, cuando sea posible, de qué tipo de asociación se trata; esto hará el código cliente mucho más estable frente a cambios en el diseño (por ejemplo, una asociación unidireccional que se transforma en bidireccional).

En realidad la implementación de asociaciones unidireccionales y bidireccionales es diferente, porque sólo las bidireccionales tienen que estar sincronizadas, pero ambos tipos presentan exactamente la misma interfaz en cada extremo. Por el contrario, las asociaciones sencillas y múltiples no sólo tienen implementaciones diferentes, sino también una interfaz ligeramente distinta, porque las asociaciones sencillas no manejan colecciones de objetos como parámetros o valores de retorno. Podríamos tratar las asociaciones sencillas como un caso particular de las asociaciones múltiples y no proporcionar ninguna implementación o interfaz especiales para ellas, pero consideramos que se usan con tanta frecuencia que los beneficios en eficiencia son proporcionados a las pérdidas en uniformidad de interfaz.

El los párrafos que siguen, `%Target` significa el nombre de la clase destino en la asociación, que tendrá que ser sustituido por su nombre real cuando el código sea generado para cada asociación en el modelo de diseño. La clase origen de una asociación unidireccional presenta una interfaz para consultar y actualizar el extremo de asociación opuesto; la clase destino no presenta ninguna interfaz, porque no tiene conocimiento de la asociación. Por el contrario, los dos lados de una asociación bidireccional presentan una

interfaz; en este caso los términos "origen" y "destino" son relativos a la clase que está viendo el extremo de asociación opuesto.

6.5.1. Métodos lectores

Tenemos dos métodos lectores, `test` y `get`, con las signaturas siguientes:

- `boolean test(%Target query_link);`
- `boolean test(Collection query_links);`
- `%Target get();`
- `Collection get();`

El método `test` comprueba si una instancia dada de la clase destino (el parámetro `query_link`) está enlazada con la instancia de la clase origen que recibe la invocación del método. La segunda versión de este método se define sólo cuando el extremo de asociación es múltiple; en este caso el método comprueba si todas las instancias contenidas en el parámetro de tipo `Collection` están enlazadas con la instancia origen[147].

El método `get` devuelve las instancias destino que están enlazadas con la instancia origen. La primera versión es para un extremo de asociación sencillo y devuelve un valor único cuyo tipo es la clase destino, mientras que la segunda versión es para un extremo de asociación múltiple, y por tanto devuelve un valor de tipo `Collection`. De acuerdo con la justificación dada más arriba al tratar con los problemas de la multiplicidad, los métodos mutadores garantizan que la multiplicidad máxima no es violada, pero en cuanto a la multiplicidad mínima, puede ocurrir que el número de instancias enlazadas sea menor que el mínimo requerido por el diseño, en cuyo caso se levanta la *excepción de multiplicidad inválida*[148].

6.5.2. Métodos mutadores

Tenemos también dos métodos mutadores, `remove` y `add`, con las signaturas siguientes:

- `int remove();`
- `int remove(%Target old_link);`
- `int remove(Collection old_links);`
- `boolean add(%Target new_link);`
- `boolean add(Collection new_links);`

---

[147] El parámetro se define de tipo `Collection` para tener mayor generalidad. `Collection` es una abstracción de los conceptos de lista (*list*), conjunto (*set*) y correspondencia (*map*). Técnicamente, es una interfaz realizada por clases de librería tales como `Vector` (obsoleta), `ArrayList`, `HashSet`, `TreeMap` y otras.

[148] De hecho, la comprobación de la multiplicidad, llevada a cabo por el método `isValid`, puede ser algo más elaborada que la simple verificación de que el número de instancias enlazadas no es menor que el mínimo requerido; el programador puede modificar manualmente el código de `isValid` para implementar una restricción más compleja. Ver Apartado 6.5.3.

El método `remove` borra las instancias destino del extremo de asociación opuesto, y devuelve un código de error conveniente. Puede borrar todas las instancias (primera versión sin parámetros), una instancia (segunda versión), o una colección de instancias (tercera versión, disponible sólo cuando el extremo de asociación opuesto es múltiple). En la tercera versión, si cualquiera de las instancias en el parámetro colección no es de tipo `%Target`, entonces no se borra ningún enlace, siguiendo la semántica de "todo o nada", y se levanta una excepción de molde de clase (*class cast exception*). Por el contrario, si alguna de las instancias de la colección (o la única instancia, en la segunda versión del método) simplemente no está enlazada a la instancia origen, entonces la operación prosigue sin considerarlo un error. En una asociación bidireccional el método invoca un `remove` recíproco en cada una de las instancias que hay que borrar. El método `remove` puede dejar la instancia origen o alguna de las instancias destino en un estado inválido respecto a las restricciones de multiplicidad mínima, en cuyo caso se devuelve un código de error, pero no se levanta ninguna excepción. Si a continuación se invocara el método `get`, se levantaría la *excepción de multiplicidad inválida*.

El método `add` añade una nueva instancia o colección de instancias destino en el extremo de asociación opuesto, y devuelve un valor `boolean` para indicar si la operación fue ejecutada o no. La segunda versión de este método se define sólo cuando el extremo de asociación es múltiple; si cualquier instancia del parámetro colección no es de tipo `%Target`, entonces no se añade ningún enlace, siguiendo la semántica de "todo o nada" y se levanta una excepción de molde de clase (*class cast exception*). Por el contrario, si alguna de las instancias de la colección (o la única instancia, en la primera versión del método) ya está enlazada a la instancia origen, entonces la operación prosigue sin considerarlo un error (y, por supuesto, sin añadir un duplicado). En una asociación bidireccional, el método invoca un `add` recíproco en cada una de las instancias que hay que añadir. El método `add` comprueba si la instancia origen, o alguna de las instancias destino, sería dejada en un estado inválido respecto a las restricciones de multiplicidad máxima, en cuyo caso la operación es cancelada, no se añade ningún enlace, y se devuelve un valor `False`[149].

Si se desea sustituir algunas instancias destino por otras instancias destino, hay que invocar primero el método `remove` y luego el método `add`; en caso contrario el resultado podría ser diferente del esperado (ver Apartado 6.2.2). Atención, esto vale *incluso para asociaciones sencillas*: no hay ningún `remove` implícito de la instancia antigua cuando se añade una

[149] Como en el caso de `get` y `remove`, la comprobación es ejecutada por el método `isValid`, que puede conseguir un comportamiento más general.

nueva (se hace así para obtener un comportamiento lo más parecido posible entre asociaciones sencillas y múltiples).

6.5.3. Métodos auxiliares para averiguar el estado de la asociación

Tenemos dos métodos auxiliares para conocer el estado de la asociación desde el punto de vista de una instancia origen particular:

- `boolean isValid();`
- `long numberOfLinks();`

El método `isValid` determina si la instancia origen ve el número correcto de instancias destino en el lado opuesto de la asociación, de acuerdo con las restricciones de multiplicidad especificadas en el modelo de diseño. La herramienta genera código sólo para el caso más sencillo, cuando la restricción de multiplicidad consiste en un único intervalo `MIN..MAX`. En todo caso, el programador puede modificar manualmente el código para implementar una restricción más compleja, y los cambios afectarán a la ejecución de los métodos lectores y mutadores, ya que ellos comprueban las restricciones de multiplicidad por medio de este método. Este método también es útil cuando la comprobación automática de las restricciones de multiplicidad está desactivada y el programador asume la responsabilidad de comprobarlas manualmente en puntos específicos del código fuente.

El método `numberOfLinks` devuelve el número de instancias destino enlazadas a la instancia origen.

6.5.4. Métodos auxiliares para averiguar la definición de la asociación

Tenemos cinco métodos auxiliares para conocer la definición de la asociación desde un lado de la misma, que pueden ser útiles para el código cliente:

- `boolean isBidirectional();`
- `boolean isMandatory();`
- `boolean isMultiple();`
- `long getMIN();`
- `long getMAX();`

El método `isBidirectional` determina si el extremo de asociación recíproco también es navegable.

El método `isMandatory` determina si la multiplicidad mínima es mayor que cero.

El método `isMultiple` determina si la multiplicidad máxima es mayor que uno.

El método `getMIN` devuelve el valor de la restricción de multiplicidad mínima.

El método `getMAX` devuelve el valor de la restricción de multiplicidad máxima. Cuándo ésta es *muchos* ('*') se devuelve un valor especial. La interfaz también define el valor constante `int MANY` (de hecho -1).

6.6. La herramienta de generación de código

En esta Sección vamos a presentar brevemente la herramienta que hemos desarrollado: *JUMLA* (*Java code generator for UML Associations*). Esta herramienta lee un modelo UML, almacenado en formato XMI, y genera el código Java para gestionar las asociaciones UML contenidas en el modelo de entrada, de acuerdo con la técnica descrita en este Capítulo. La herramienta *sólo genera código para asociaciones*: ignora cualquier otro artefacto UML que no esté directamente relacionado con las asociaciones, como generalizaciones entre clases, atributos y operaciones de clase, etc. La herramienta presenta las clases y asociaciones encontradas en el modelo, y el usuario puede seleccionar de qué asociaciones quiere que se genere el código.

La herramienta crea ficheros de salida Java para las clases implicadas e inserta en ellas el código para las asociaciones, con etiquetas convenientes que marcan el principio y fin del código generado. Si el fichero de la clase ya existe, el código es insertado al final del fichero, respetando cualquier otro código de la clase que el programador pueda haber escrito manualmente (por el contrario, si el programador cambia el código de la asociación y luego lo vuelve a generar, los cambios manuales se pierden).

La **Figura 6.12** muestra un modelo de ejemplo y la **Figura 6.13** muestra cómo se presenta en la ventana principal de la herramienta JUMLA. El panel izquierdo de la herramienta muestra las clases contenidas en el modelo, en una estructura en árbol correspondiente a la estructura de paquetes del modelo. El panel derecho muestra las asociaciones contenidas en el modelo. Para cada asociación se presenta la siguiente información: clases origen y destino; nombre de rol (opcional), multiplicidad, navegabilidad y visibilidad de los extremos de asociación origen y destino; nombre de asociación (opcional). El usuario puede seleccionar con una casilla de verificación las asociaciones cuyo código quiere generar.

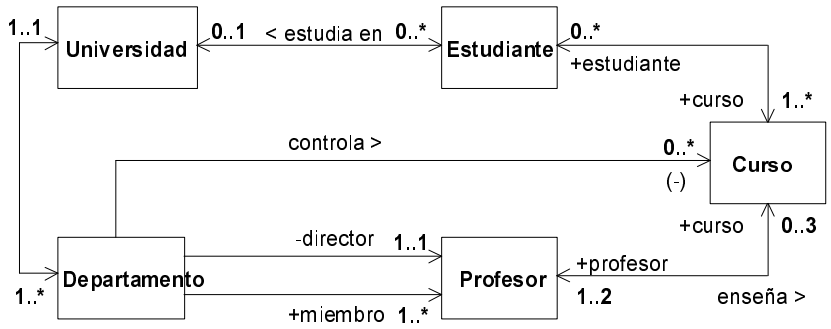

Figura 6.12. Un modelo de ejemplo con algunas clases y asociaciones entre las clases

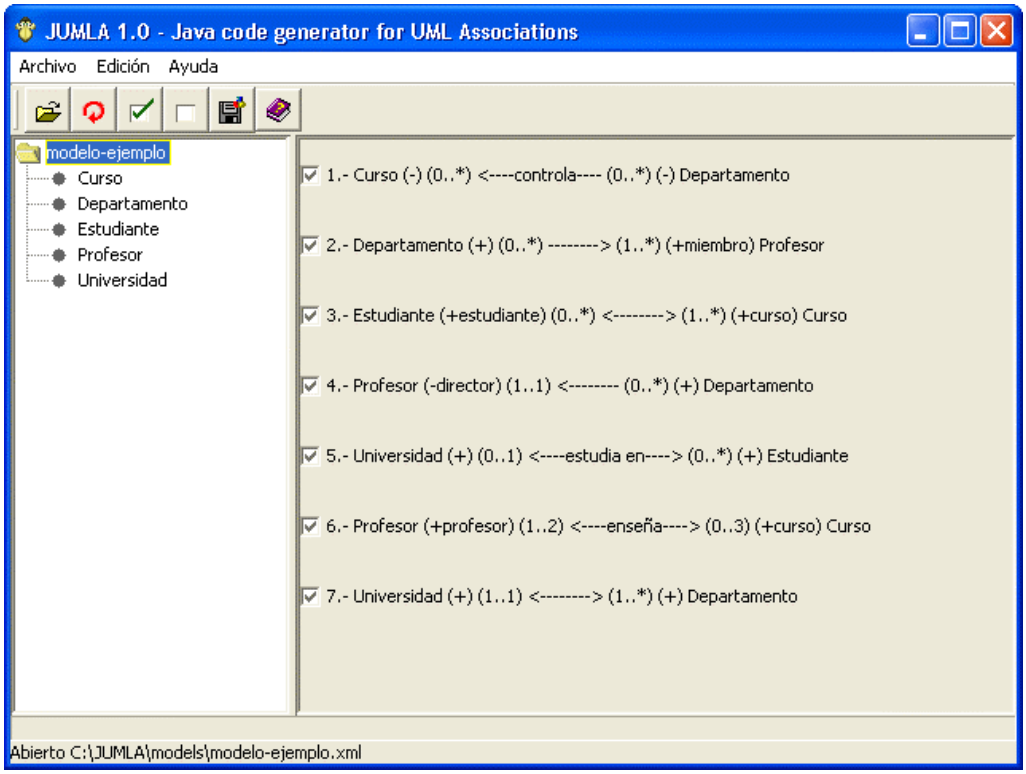

Figura 6.13. Una instantánea de la herramienta JUMLA.

La herramienta funciona de acuerdo con *cinco opciones predefinidas* que pueden ser deshabilitadas por el usuario para obtener mayor flexibilidad en la generación de código o en el tratamiento del modelo de entrada. La **Tabla 6.1** resume las opciones de la herramienta.

Opción	**Valor por defecto**
Comprobar las restricciones de multiplicidad mínima y máxima en el método `get`	Sí
Comprobar la restricción de multiplicidad máxima en el método `add`	Sí
Comprobar el tipo de los objetos en parámetros `Collection` en los métodos `add` y `remove`	Sí
Rechazar asociaciones unidireccionales con restricción de multiplicidad en el extremo origen	Sí
Rechazar asociaciones bidireccionales con un extremo privado o protegido	Sí

Tabla 6.1. Resumen de opciones de la herramienta

Las dos primeras opciones se refieren a la *comprobación automática de restricciones de multiplicidad* en los métodos mutadores y lectores mediante el método auxiliar `isValid`. De acuerdo con la justificación dada en la Sección 6.2, el funcionamiento predefinido es así: los *métodos*

lectores (`get`) levantan una excepción de multiplicidad inválida (*invalid multiplicity exception*), definida en el código que implementa las asociaciones, si no se satisfacen las restricciones de multiplicidad; los *métodos añadidores* (`add`) rechazan la adición de nuevos enlaces si no se satisfacen estas restricciones, pero no levantan ninguna excepción; y los *métodos borradores* (`remove`) no hacen ninguna comprobación. Estas dos primeras opciones permiten generar un código simplificado que omite estas comprobaciones, asumiendo el usuario la responsabilidad del control de la multiplicidad.

La tercera opción de la herramienta se refieren a la *comprobación automática de tipos* en los métodos mutadores (`add` y `remove`) para asociaciones múltiples, que manejan parámetros de tipo `Collection`, por medio de un moldeado (*casting*) explícito en tiempo de ejecución. De acuerdo con la justificación dada en la Sección 6.2, si falla la comprobación de tipos, entonces no se actualizan los enlaces, y se levanta una excepción de molde de clase (*class cast exception*), predefinida en Java. Esta tercera opción permite generar un código simplificado que no comprueba el tipo de los objetos recibidos en el parámetro `Collection`, y no levanta ninguna excepción.

Las dos últimas opciones se refieren a la *comprobación de la validez del modelo de entrada*. En el funcionamiento predefinido se rechazan las asociaciones unidireccionales con restricciones de multiplicidad en el extremo de asociación no navegable (ver Sección 6.3), y las asociaciones bidireccionales con visibilidad distinta de `public` o `package` en ambos extremos (ver Sección 6.4). La cuarta opción permite generar el código sin comprobar la multiplicidad en el extremo no navegable, en lugar de rechazar la asociación. La quinta opción permite generar el código, en lugar de rechazar la asociación, cuando uno de los dos extremos tiene visibilidad `protected` o `private` y el otro extremo `public` o `package`, advirtiendo al usuario de que debe proporcionar acceso indirecto por medio de otros métodos. Cuando ambos extremos son `protected` o `private`, la asociación siempre es rechazada, porque el código generado no podría funcionar correctamente.

6.7. Comparación de resultados

En esta Sección vamos a comparar los resultados de nuestra herramienta con los de otras herramientas existentes en el mercado. Hemos escogido dos herramientas representativas, que muestran además las diferencias existentes entre unas y otras: *Rational Rose* y *MEGA*. Como banco de pruebas utilizaremos la asociación `trabaja-para` entre `Persona` y `Empresa`, con los respectivos nombres de rol `contratado` y `contratante` (ver **Figura 6.14**). Como puede observarse, se trata de

una asociación bidireccional, sencilla en un extremo y múltiple en el otro, y con un extremo público y otro privado.

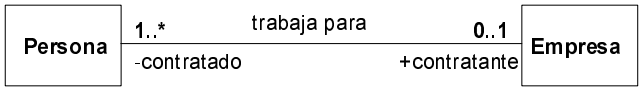

Figura 6.14. Ejemplo de asociación utilizado para comparar el código generado por diversas herramientas

Las herramientas analizadas tienen en común la creación de las clases `Persona` y `Empresa` como contenedores de los extremos de la asociación; las diferencias quedarán manifiestas en la forma de implementar los extremos.

6.7.1. Código generado por *Rational Rose*

Rational Rose [RationalRose] es probablemente la herramienta más conocida de entre las existentes en el mercado, ya que *Rational Software Corporation* es la principal empresa promotora del uso de UML. El código generado es:

```
public class Persona
{
   public Empresa contratante;

}

public class Empresa
{
   private Persona contratado[];

}
```

Como puede observarse, los dos extremos de la asociación se han traducido simplemente en dos atributos, uno en cada una de las dos clases participantes en la asociación, y con la misma visibilidad que tienen los extremos de la asociación. El atributo `contratante` es un valor simple de clase `Empresa`, y el atributo `contratado` es un *array* de clase `Persona`.

La simple inspección de este código pone de manifiesto sus muchas limitaciones:

- No hay ningún control sobre la multiplicidad mínima o máxima.
- El extremo múltiple de la asociación, `contratado`, está implementado mediante un *array*, cuyo tamaño quedará fijado en el momento de instanciarse, por lo que no es posible gestionar un número variable de enlaces.
- No hay ningún mecanismo de sincronización que garantice la actualización simultánea de ambos extremos.

Es decir, que la herramienta *Rational Rose* genera un código para la asociación que es totalmente inservible por sí mismo. Muchas otras

herramientas, como *Magic Draw* [MagicDraw], *Visual Paradigm* [VisualParadigm], etc., tienen un funcionamiento semejante a éste, limitado a la simple traducción de los extremos de la asociación en atributos, sin ningún otro mecanismo de control.

6.7.2. Código generado por *MEGA*

MEGA [MegaSuite] es una herramienta que da un gran salto adelante en la generación de código para asociaciones, incomparablemente superior a las herramientas mencionadas en el Apartado precedente. Veamos el código generado para la asociación de prueba:

```
class Persona
{
  // Association contratante, multiplicity = 0..1
  protected Empresa m_contratante;

  /** Reads the association to a Empresa object.
   *   @returns a Empresa object reference representing the contratante
association. */
  public Empresa getContratante ()
  {
    return m_contratante;
  }

  /** Sets an association to a Empresa object.
   *   @param v_contratante a new associated object for the contratante
association. */
  public void setContratante (Empresa v_contratante)
  {
    m_contratante = v_contratante;
  }

}


class Empresa
{
  // Association contratado, multiplicity = 1..*
  protected int m_nbcontratado = 0;
  protected Persona[] m_arrayOfcontratado;

  /** Reads the association to a Persona object.
   *   @returns all Persona object references representing the contratado
association. */
  public Persona[] getContratado ()
  {
    return m_arrayOfcontratado;
  }

  /** Reads the association to a Persona object.
   *  @returns one Persona object including in the contratado association.
*/
  public Persona getIndexContratado (int index)
  {
    if (index >= m_nbcontratado)
    {
      throw new ArrayIndexOutOfBoundsException (index + " >= " +
m_nbcontratado);
    }
    return m_arrayOfcontratado[index];
  }
```

```
  /** Sets an association to a Persona object.
   *  @param v_arrayOfcontratado a new associated collection of objects for
the contratado association. */
  public void setContratado (Persona[] v_arrayOfcontratado)
  {
    m_arrayOfcontratado = v_arrayOfcontratado;
  }

  /** @return returns the number of associated object in the contratado
association. */
  public int nbContratado ()
  {
    return m_nbcontratado;
  }

  /** Adds a Persona object.
   *   @param v_contratado a new associated objects for the contratado
association. */
  public void addContratado (Persona v_contratado)
  {
    Persona[] oldData = m_arrayOfcontratado;
    m_nbcontratado++;
    m_arrayOfcontratado = new Persona[m_nbcontratado];
    if    (m_nbcontratado    >    1)    System.arraycopy    (oldData,    0,
m_arrayOfcontratado, 0, m_nbcontratado-1);
    m_arrayOfcontratado[m_nbcontratado-1] = v_contratado;
  }

}
```

Este código es mucho más interesante y requiere algunas explicaciones:

- El extremo sencillo se implementa mediante un atributo de valor simple, y sendos métodos `get` y `set` para acceder a su valor.
- El extremo múltiple se implementa mediante dos atributos que almacenan respectivamente un *array* y un contador de elementos, así como métodos `get`, `getIndex`, `set`, `nb` y `add` para manipular su contenido.

No obstante, a persar de sus evidentes mejoras, presenta también algunos inconvenientes:

- El método `add` puede aumentar el tamaño del *array* para almacenar un elemento nuevo, creando un nuevo *array* de tamaño mayor y copiando en él todo el contenido del antiguo *array*, técnica que es altamente ineficiente. Lo propio sería el uso de colecciones estándar, que además de proporcionar mayor riqueza de métodos de acceso, gestionan de modo robusto y flexible un número variable de elementos.
- No hay ningún método para eliminar elementos, reduciendo así el tamaño del *array*.
- No hay ningún control de multiplicidad mínima o máxima.
- No hay ningún mecanismo de sicronización de los extremos.
- La visibilidad especificada para los extremos de asociación no queda reflejada en la implementación, ya que todos los atributos son protegidos y todos los métodos son públicos.

En resumen, la gestión de asociaciones múltiples es ineficiente y otros aspectos básicos, especialmente la gestión de asociaciones bidireccionales, quedan sin implementar.

6.7.3. Código generado por *JUMLA*

El código generado por *JUMLA* es considerablemente más exhaustivo, dado que afronta la implementación rigurosa de la multiplicidad, navegabilidad y visibilidad de las asociaciones. Para la asociación de prueba el resultado es:

```
public class Persona {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  public final PersonaEmpresa_SingleBidirectional_Association
         contratante = new PersonaEmpresa_SingleBidirectional_Association();

  public class PersonaEmpresa_SingleBidirectional_Association {

    /** 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
     */
    public final static int MANY = -1;

    /** Enlace de Persona a Empresa obligatorio (true) u opcional (false) */
    private final static boolean mandatory = false;

    /** Almacena el enlace con Empresa */
    private Empresa link = null;

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

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

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

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    public 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 *)
     */
    public long getMAX() {
      return 1;
    }

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

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

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

    /** Devuelve el enlace Empresa 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)
     */
    public Empresa get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase Persona"
        + " no cumple con los requisitos de la asociación");
      }
      return link;
    }

    /** Añade un enlace con una instancia de Empresa
     *  @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
     */
    public boolean add(Empresa 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 Persona a Empresa
        link = new_link;
        // Si en el otro extremo no está establecido el enlace y
        if ((! new_link.contratado.test(Persona.this)) &&
            // no se puede establecer de Empresa a Persona
            (! new_link.contratado.add(Persona.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 Persona y link
     *   siguen  cumpliendo  con  las  restricciones  de  multiplicidad,  1  si
Persona
     *  deja de cumplir con las restricciones, 2 si link deja de cumplir
     *  y 3 si ambos
     */
    public int remove() {
      synchronized (this) {
        boolean invalid_link = false;
        Empresa 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.contratado.test(Persona.this)) &&
               //  se  llama  a  contratante  para  que  elimine  su  enlace  con
Persona
               (old_link.contratado.remove(Persona.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 Empresa
     *  @param old_link Supuesto enlace de tipo Empresa
     *  @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
     */
    public int remove(Empresa old_link) {
      if ((old_link != null) && test(old_link)) {
        return remove();
      }
      return -1;
    }

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

```
}

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

public class Empresa {

  /** !Código generado por JUMLA */
  /** Atributo por el cual se gestiona la asociación */
  private final EmpresaPersona_MultipleBidirectional_Association
       contratado = new EmpresaPersona_MultipleBidirectional_Association();

  private class EmpresaPersona_MultipleBidirectional_Association {

    /** Multiplicidad mínima del enlace */
    private final static long MIN = 0;

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

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

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

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

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

    /** Devuelve la multiplicidad mínima del final de la asociación
     *  @return multiplicidad mínima
     */
    private 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 *)
     */
    private long getMAX() {
      return MAX;
    }

    /** Dice si Empresa cumple con las restricciones de la asociación
     *  @return true si sí las cumple, false en caso contrario
```

```
     */
    private 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
     */
    private long numberOfLinks() {
      return links_hs.size();
    }

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

    /** Comprueba si Empresa está asociado con una colección de objetos
Persona
     *  @param query_links Colección de supuestos enlaces
     *  @return true si todos los elementos de links_hs están enlazados
     *  con Empresa o si links_hs está vacío y query_links también o es null
     */
    private 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 Persona
     *  contenidos en links_hs. Lanza la excepción UnsetAssociationException
si
     *  la asociación no cumple con los requisitos de la asociación
     *  @return Copia de links_hs
     */
    private Collection get() throws UnsetAssociationException {
      if (! isValid()) {
        throw new UnsetAssociationException ("El objeto de clase Empresa"
        + " no cumple con los requisitos de la asociación");
      }
      return (Collection) links_hs.clone();
    }

    /** Añade un enlace con una instancia de Persona
     *  @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
     */
    private boolean add(Persona 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 Empresa a Persona
            links_hs.add(new_link);
            // Si en el otro extremo no está establecido el enlace y
            if ((! new_link.contratante.test(Empresa.this)) &&
                // no se puede establecer de Persona a Empresa
                (! new_link.contratante.add(Empresa.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 Persona 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
     */
    private boolean add(Collection new_links) {
      boolean stop = false; // Parada forzosa
      Persona last_link_i = null;
      Persona last_link_ii = null;
      Iterator i;

      synchronized (this) {
        if (new_links != null) {
          // Sobre una copia de new_links se eliminan aquellos objetos
          // Persona 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 = (Persona) i.next();
            } catch (ClassCastException e) {
              // Si algún elemento no es de tipo Persona 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 = (Persona) i.next();
              // de Empresa a Persona y de
              links_hs.add(last_link_i);
              // de Persona a Empresa
              if (! last_link_i.contratante.add(Empresa.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 = (Persona) i.next();
              while (last_link_i != last_link_ii) {
                // de Empresa a Persona
                links_hs.remove(last_link_ii);
                // y de Persona a Empresa
                last_link_ii.contratante.remove(Empresa.this);
                last_link_ii = (Persona) 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 Empresa y las
instancias
     *    de links_hs siguen cumpliendo con las restricciones de la
asociación,
     *  1 si Empresa deja de cumplir con las restricciones, 2 si al menos un
     *  elemento de links_hs deja de cumplirlas y 3 si ambos (Empresa y al
     *  menos un elemento de links_hs)
     */
    private 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()) {
          Persona last_link = (Persona) i.next();
          // se elimina el enlace de Empresa a last_link
          links_hs.remove(last_link);
          // y de last_link a Empresa
          if (last_link.contratante.remove(Empresa.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 Persona
     *  @param old_link Enlace de tipo Persona 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
     *   Empresa, 2 si en contratado, 3 si en ambos y -1 si no existe el
enlace
     *  con contratado u old_link es null
     */
    private int remove(Persona old_link) {
      synchronized (this) {
```

```
        boolean invalid_old_link = false;
        if ((old_link != null) && test(old_link)) {
          // Elimina el enlace de Empresa a Persona
          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.contratante.test(Empresa.this)) &&
               // se llama a old_link para que elimine su enlace con Empresa
               (old_link.contratante.remove(Empresa.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 Persona
     *  @param old_links Colección de objetos Persona cuyo enlace con
     *  Empresa se quiere borrar
     *   @return  0  si  se  eliminan  los  enlaces  solicitados  y  se  sigue
cumpliendo
     *  con las restricciones de la asociación, 1 si Empresa deja de cumplir
con
     *   las restricciones, 2 si al menos un elemento de old_links deja de
cumplir
     *  y 3 si ambos (Empresa y al menos un elemento de old_links_hs). -1 si
     *   algún elemento de old_links no es de tipo Persona u old_links es
null y
     *  por tanto, no se hace nada
     */
    private int remove(Collection old_links) {
      boolean invalid_old_link = false;
      Persona last_link = null;
      Iterator i;

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

      synchronized (this) {
        // Se borran los enlaces
        i = old_links.iterator();
        while (i.hasNext()) {
          last_link = (Persona) i.next();
          // de Empresa a last_link
          links_hs.remove(last_link);
          // y de last_link a Empresa
```

```
          if (last_link.contratante.remove(Empresa.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 */

}
```

Sobre este código podemos hacer las siguientes precisiones:

- En cada clase participante, el código de control de la asociación queda encapsulado dentro de una clase interna con la misma visibilidad que el extremo correspondiente. Se declara un atributo cuyo tipo es esta clase, con su misma visibilidad, pero con la cláusula `final`, de modo que se evitan manipulaciones indebidas del mismo. De este modo, si la clase participa en varias asociaciones, el código de control de cada una de ellas queda perfectamente aislado y distinguible del resto.
- Dentro de la clase interna, todos los atributos son privados, y los métodos de control tienen la visibilidad del extremo correspondiente.
- La sincronización de los extremos en asociaciones bidireccionales se logra mediante la técnica de actualización recíproca, como se ha descrito en el Apartado 6.3.2. Este control es la principal fuente de complejidad del código generado.
- La multiplicidad de los extremos se controla mediante comparaciones con los valores tomados del modelo, tal como se ha descrito en el Apartado 6.2.2. Los extremos múltiples se gestionan mediante las colecciones estándar de Java, lo que requiere importar las librerías `Collection`, `HashSet` e `Iterator`.

6.7.4. Valoración final

El código generado por *JUMLA* es ciertamente exhaustivo, y supone añadir, en cada clase participante y según los casos, entre 150 y 330 líneas de código (comentarios incluidos) por cada asociación. Posiblemente éste es un aspecto mejorable, y nuestras investigaciones continúan en esta línea. En todo caso, el gran mérito está en haber dado una solución a los problemas que otras herramientas de generación de código, sencillamente, ignoran.

CUARTA PARTE:

CONCLUSIONES Y REFERENCIAS

7. Conclusiones

En este Capítulo final presentamos un breve resumen del contenido de esta Tesis Doctoral, resaltando aportaciones originales, propuestas concretas de modificación del Estándar de UML, y posibles trabajos futuros.

7.1. Aportaciones originales

En esta Tesis Doctoral hemos desarrollado una investigación acerca del *concepto de asociación* en el Lenguaje Unificado de Modelado, centrada en tres grandes *aspectos teóricos* (la multiplicidad, la navegabilidad y la visibilidad) y buscando siempre las consecuencias de su *aplicación práctica* (la implementación).

7.1.1. ¿Qué es una asociación?

Tal vez uno de los frutos más importantes de esta Tesis Doctoral es la clarificación del concepto mismo de asociación. La definición de *asociación* que da el Estándar en el Glosario de Términos es la siguiente: "una asociación es la relación semántica entre dos o más clasificadores que especifica conexiones entre sus instancias" [UML, p. B-3]. A su vez, un *enlace* se define como "una conexión semántica entre una tupla de objetos; una instancia de una asociación" [UML, p. B-11]. El objetivo de este trabajo ha sido precisar el sentido de la palabra "semántica" cuando se usa para definir una asociación o un enlace. ¿Qué es una *relación semántica*, qué es una *conexión semántica*? ¿Es lo mismo un *enlace* que una *tupla*?

Hemos llegado a la conclusión de que la semántica o *significado* de toda asociación incluye dos aspectos que están íntimamente entrelazados: el *aspecto estático* y el *aspecto dinámico*, relacionados respectivamente con la estructura y con el comportamiento del sistema; estos dos aspectos sirven de base para una nueva clasificación de las asociaciones. También hemos argumentado que, para lograr un mayor desacoplamiento entre los participantes en una asociación, conviene definir una asociación no entre clasificadores, sino entre interfaces (redefiniendo también el concepto de interfaz, ya que su definición actual no lo permite).

El aspecto estático se manifiesta en el "hecho" expresado por el enlace; la existencia del enlace implica la verificación del predicado significado por el nombre de la asociación, y este hecho es parte del *estado del sistema*. El aspecto dinámico se manifiesta en la posibilidad de comunicación proporcionada por el enlace; la existencia del enlace implica que las instancias se conocen en la dirección en que el enlace es navegable,

y por tanto pueden *intercambiar mensajes* en una interacción, teniendo en cuenta también la visibilidad de las operaciones invocadas. El uso de interfaces flexibiliza el modelado con asociaciones, ya que la asociación puede definirse independientemente de los clasificadores conectados.

Así pues, he aquí nuestra definición: *Una asociación es una relación definida entre dos o más interfaces. En cada extremo, la interfaz especifica la estructura y el comportamiento que es posible conocer navegando hacia ese extremo a través de la asociación. La asociación especifica un conjunto de enlaces entre instancias de los clasificadores que realicen las respectivas interfaces en cada extremo. Cada enlace es una conexión entre instancias que declara un hecho (la verificación de un predicado) y proporciona una posibilidad de comunicación (una ruta navegable).*

7.1.2. La multiplicidad de las asociaciones

En el Capítulo 3 hemos abordado dos temas principales: la multiplicidad en asociaciones n-arias, y la multiplicidad en asociaciones cualificadas y clases-asociación.

En primer lugar, hemos considerado algunos problemas semánticos de la multiplicidad mínima en las asociaciones n-arias, tal como se expresa actualmente en UML; no obstante, nuestras ideas son suficientemente generales como para ser aplicables a otras técnicas de modelado más o menos basadas en el enfoque Entidad/Relación. La *multiplicidad mínima* está estrechamente relacionada con la restricción de participación, aunque en el caso de asociaciones n-arias no significa la participación de la clase en la asociación, sino la *participación de tuplas de las otras n-1 clases*. Más aún, hemos descubierto que esta última participación está definida con incerteza, permitiendo *tres interpretaciones conflictivas*: participación de tuplas reales, participación de tuplas potenciales, y participación con enlaces cojos.

La única que está (implícitamente) de acuerdo con la documentación de UML es la segunda interpretación, tuplas potenciales, a pesar del efecto rebote de la multiplicidad mínima 1. El Estándar debería clarificar esta cuestión, sin resignarse a una falta de claridad en la definición. Además, si esta segunda interpretación fuera elegida, el Estándar también debería advertir, puesto que este resultado no es intuitivo en absoluto, que una multiplicidad mínima 1 o mayor asignada a una clase fuerza a que todas las potenciales subtuplas de instancias de las clases restantes existan contenidas en alguna n-tupla; por tanto, la multiplicidad mínima sería 0 en casi todas las asociaciones n-arias.

La tercera interpretación, *enlaces cojos*, que es una variación de la primera, parece intuitiva y tiene también algunas ventajas pragmáticas, aunque está en contradicción con la definición de asociación n-aria en UML (tal vez más con la letra que con el espíritu). Nosotros nos inclinamos a apoyar esta interpretación, teniendo cuidado de especificar que *representa*

asociaciones incompletas, pero no subasociaciones relacionadas restrictivas. Podría permitirse que faltasen hasta n-2 patas, y el valor "desconocido", "vacío" o "nulo" debería ser considerado como un valor concreto a la hora de aplicar las restricciones impuestas por los valores de multiplicidad. No obstante, este tema merece ser proseguido en una investigación que excede el ámbito de este trabajo.

La eventual clarificación de este punto deja otro problema sin resolver: la participación de cada clase queda inexpresada en el estilo Chen de representar las multiplicidades (que es también el estilo de UML), mientras que el estilo Merise las muestra adecuadamente. *Ambos estilos Chen y Merise son correctos, pero describen características diferentes de la misma asociación*, que no pueden ser derivadas una de la otra en el caso n-ario, aunque sí están relacionadas por una simple regla de consistencia.

Siendo útiles ambos estilos para entender la naturaleza de las asociaciones, proponemos una sencilla *extensión a la notación de las multiplicidades n-arias de UML*, que permite la representación de la participación y la dependencia funcional (esto es, los estilos Merise y Chen, o multiplicidad interna y externa en la terminología de CDIF). Puesto que esta notación es compatible con las tres interpretaciones alternativas de la multiplicidad de Chen, su uso no evita en sí mismo la ambigüedad de la definición de multiplicidad: son problemas independientes. Si esta notación fuera aceptada, el Estándar debería *modificar también el metamodelo* en este sentido, ya que sólo tiene prevista una especificación de multiplicidad en la metaclase `AssociationEnd`. Si éste no fuera el caso, al menos podría reconocerse que las multiplicidades de Chen no son las únicas restricciones de co-ocurrencia razonables que pueden definirse en una asociación n-aria.

Entender las asociaciones n-arias es un problema difícil en sí mismo. Si las reglas del lenguaje usadas para representarlas no están claras, esta tarea puede hacerse inaccesible. Si la interpretación de las asociaciones n-arias es incierta, la comunicación directa entre los modeladores se hace imposible. Si las implicaciones semánticas de un modelo son ambiguas, los implementadores tendrán que tomar decisiones que no les corresponden, y posiblemente decisiones erróneas. Estas razones son más que suficientes para esperar una definición más precisa de estos temas por parte de UML, que tal vez se consiga en la versión 2.0[150].

El segundo tema que hemos abordado es la *definición de multiplicidad en asociaciones cualificadas y clases-asociación*. Al contrario de lo que ocurre con las asociaciones n-arias, el Estándar sí explica el significado de la multiplicidad mínima en el extremo destino de una asociación cualificada, adoptando el equivalente a la interpretación de tuplas potenciales. Esto apoya la conclusión de que el Estándar asume implícitamente la

[150] De hecho, la propuesta 3C (*Clear, Clean, Concise*) para la elaboración de la versión 2.0, promovida por la empresa *Financial Systems Architects* (Nueva York, EE.UU.) [cUML], ya ha asumido nuestras ideas sobre las asociaciones n-arias.

interpretación de tuplas potenciales para las asociaciones n-arias. Como hemos considerado que, por razones prácticas y por ser más intuitiva, en la multiplicidad n-aria es más conveniente la interpretación de enlaces cojos, hemos intentado aplicarla también a la multiplicidad cualificada. No ha sido posible, porque el concepto de "asociación incompleta" no tiene sentido en una asociación cualificada, ya que no hay ninguna asociación entre la clase origen y el cualificador. Si tuviera sentido en un dominio determinado la representación de una asociación incompleta entre la clase origen y el cualificador, lo propio sería emplear una asociación n-aria, no una asociación cualificada. Así pues, debemos renunciar a la interpretación de enlaces cojos para la multiplicidad cualificada.

La multiplicidad en las clases-asociación se ve afectada por la restricción de que *no puede haber tuplas repetidas*, como en cualquier otra asociación, lo que dificulta el modelado de algunas situaciones comunes, y específicamente la representación de asociaciones con "lógica temporal", es decir, predicados que se consideran válidos en un periodo de tiempo determinado. El uso de asociaciones cualificadas en lugar de clases-asociación no parece que ayude a resolver estos problemas, porque en principio también están sometidas a esta restricción. La definición de asociación cualificada en UML, a medio camino entre la asociación binaria y la asociación n-aria, no deja suficientemente claro este punto: si prevalece la analogía binaria, no se pueden repetir las tuplas; si prevalece el aspecto n-ario, sí se pueden repetir.

Finalmente, hemos analizado en detalle la raíz de estas dificultades, que es la *definición de asociación como conjunto de tuplas no repetidas*, una definición excesivamente influenciada por las metodologías de diseño de bases de datos, que ha sido adoptada en UML sin considerar todas las consecuencias. Hemos mostrado una posible solución para escapar de esta restricción, consistente en insertar una clase asociativa ficticia que permita la repetición de tuplas, pero introduciendo una complejidad innecesaria en los modelos. En todo caso, el análisis del metamodelo, salvando sus contradicciones internas, deja claro que un enlace no es exactamente lo mismo que una tupla: un enlace *es una conexión* entre dos (o más) objetos, y *determina una tupla*; todos los enlaces de una asociación son distintos (en cuanto que tienen identidad propia y por tanto son distinguibles), pero dos o más enlaces pueden conectar los mismos objetos y así determinar la misma tupla (pueden tener el mismo contenido de datos). Por tanto, *la restricción de que no puede haber tuplas repetidas no se deriva de la propia naturaleza de los enlaces*, sino que es una restricción adicional que podría suprimirse sin violentar los principios del lenguaje, siendo fácil recuperarla cuando lo requiera la naturaleza del problema modelado.

7.1.3. La navegabilidad de las asociaciones

En el Capítulo 4 hemos considerado algunos problemas semánticos de las asociaciones y la navegabilidad en UML. Hemos intentado clarificar algunas definiciones, y hemos propuesto soluciones para algunos problemas. Hemos buscado una *definición de navegabilidad que falta en la documentación oficial*, hemos explorado la relación de la navegabilidad con el envío de mensajes, y hemos examinado en detalle la cuestión de los enlaces de comunicación, subrayando algunos malentendidos y conflictos en la presente definición de UML. Hemos reafirmado el principio de que todo enlace es instancia de una asociación, y nuestro análisis nos ha llevado a la distinción entre *asociaciones estructurales y contextuales*, y a una nueva definición y aplicación de los *estereotipos de asociaciones y enlaces*. Hemos señalado la relación entre navegabilidad y dependencia, y hemos examinado la invertibilidad, eficiencia y notación de las asociaciones navegables. También hemos aplicado el *concepto de navegabilidad a las asociaciones más complejas* (clase-asociación, asociación cualificada y asociación n-aria), un tema que ha sido descuidado en la documentación de UML hasta el momento.

Las asociaciones en los modelos orientados a objetos, y más específicamente en los modelos UML, no son simétricas. Las principales asimetrías que podemos encontrar en las asociaciones son la *asimetría lingüística*, que consiste básicamente en la no intercambiabilidad entre sujeto y objeto en la expresión verbal que da nombre a la asociación, y que se expresa gráficamente mediante el triángulo de dirección del nombre de la asociación; la *asimetría todo-parte*, expresada por la propiedad de agregación o composición de la asociación; y la *asimetría de comunicación*, que significa la dirección en la que se puede obtener conocimiento a través de la asociación, y que está estrechamente relacionada con los conceptos de visibilidad, referencia y navegación. Una asociación puede ser bidireccional en este último sentido (navegable en ambas direcciones), pero esto no la hace simétrica. Estos tres tipos de asimetría son independientes, aunque estén conceptualmente relacionados. Las generalizaciones y dependencias, que son otros tipos de relaciones junto con las asociaciones, son también asimétricas.

"Navegar" o "recorrer" una asociación es obtener, a través de la asociación, una ruta o referencia al objeto opuesto que permita manejarlo; en otras palabras, *formar la expresión de una ruta que designa el objeto (o conjunto de objetos) destino desde un objeto origen*. Una vez que el objeto origen tiene un nombre relativo del objeto destino que es válido en el contexto del origen, el origen puede manipular el destino, es decir, puede invocar sus operaciones públicas, obtener o modificar sus atributos públicos, pasarlo como parámetro en mensajes a otros objetos, etc. La navegabilidad, por tanto, es (nuestra definición) *la posibilidad de que un objeto origen designe un objeto destino a través de una asociación*, con el fin de

manipularlo o acceder a él en una interacción con intercambio de mensajes. Esta definición u otra similar debería ser incorporada al Estándar.

La dirección de navegabilidad indica que el objeto en el extremo origen puede conocer otros objetos en el extremo destino a través de la asociación. El objeto que tiene conocimiento de la asociación es responsable de *mantener el estado* de la asociación y *controlar la interacción* que puede tener lugar a través de ella. Si ambos extremos tienen conocimiento y son responsables de la asociación, entonces se dice que la asociación es bidireccional (*two-way*), en caso contrario la asociación es unidireccional (*one-way*). Una asociación sin navegabilidad (*no-way*) no tiene sentido.

Para que tenga lugar la comunicación entre objetos se requiere tanto navegabilidad como visibilidad: *un objeto puede comunicarse sólo con otros objetos de los que tiene conocimiento, y que han puesto las operaciones deseadas a disposición de sus clientes en sus interfaces*. Esta idea debería ser expresada clara y concisamente en el Estándar. La navegabilidad está tan estrechamente relacionada con la capacidad de enviar mensajes, que muy a menudo se identifican estos dos conceptos.

Los *distintos tipos de enlaces de comunicación* que pueden existir en un modelo plantean la cuestión de si todo enlace es o no instancia de una asociación, y si es necesario que exista una asociación siempre que haya comunicación entre objetos. La distinción entre asociaciones estáticas y dinámicas resulta inadecuada para resolver este problema, ya que en orientación a objetos *lo estático y lo dinámico son aspectos de toda asociación*, más que la característica definitoria de dos subtipos disjuntos de asociación. En su lugar hemos propuesto la *distinción entre asociaciones estructurales y contextuales*, que, con una adecuada redefinición de los estereotipos de asociaciones y enlaces, permite mantener el principio de que todo enlace es instancia de una asociación. Esta distinción no está basada en las propiedades estáticas o dinámicas de las asociaciones, puesto que toda asociación está (o al menos puede estar) implicada en la estructura y el comportamiento del sistema modelado. En cambio, nuestra clasificación se basa en el contexto en el que son válidas las asociaciones. La distinción se expresa gráficamente en los diagramas usando los estereotipos tradicionales de asociación y enlace, aunque ya no se aplican a los extremos de asociación y enlace, sino a las asociaciones y enlaces mismos.

Hemos examinado tres propiedades de las asociaciones que dependen de la navegabilidad: *dependencia*, *invertibilidad* y *eficiencia*. Puesto que *navegabilidad significa conocimiento*, y conocimiento significa tanto comunicabilidad como dependencia, la navegabilidad *crea una dependencia* desde el origen hacia el destino. Cuando las asociaciones en un modelo son predominantemente unidireccionales, la reutilización de pequeñas partes del modelo resulta más sencilla. Este es el principal argumento en favor de asociaciones unidireccionales como opción por defecto, en lugar de asociaciones bidireccionales como promueve UML. En todo caso, las

asociaciones bidireccionales no pueden ser descartadas completamente, ya que a veces las requiere la naturaleza del problema o de la solución.

En algunos lugares del Manual de Referencia la invertibilidad-bidireccionalidad parece ser una propiedad lógica de una asociación (incluso de toda asociación), diferente del hecho de que la asociación sea navegable en ambas direcciones. La navegabilidad *no* sería una propiedad lógica, sino una propiedad de la implementación que significaría más o menos lo mismo que eficiencia de navegación. Nosotros consideramos, por el contrario, que la posibilidad lógica de navegación es un concepto importante tanto en el análisis como en el diseño. Desde nuestro punto de vista, invertibilidad, bidireccionalidad y navegabilidad en ambas direcciones son sinónimos. *La esencia de una asociación es el conocimiento*, y el conocimiento puede ser unidireccional, no por cuestión de eficiencia, sino por cuestión de principio. Así pues, la flecha de navegabilidad no debería nunca usarse para significar eficiencia de navegación, especialmente porque esto hace imposible especificar que una asociación no es navegable en absoluto en una dirección.

Entre las tres *opciones de presentación* recomendadas por el Estándar, pensamos que la mejor práctica es usar sólo el estilo "suprimir todo" para las primeras fases del análisis, y el estilo "mostrar todo" para el análisis detallado y para el diseño. Una conexión sin flechas no debe usarse para significar navegabilidad bidireccional, sino navegabilidad sin decidir o sin especificar.

El concepto de navegabilidad, que en la documentación sólo es tratado en relación con la asociación binaria simple, se puede extender sin excesiva dificultad a la *clase-asociación* y la *asociación cualificada*. Por el contrario, no resulta tan sencillo para la *asociación n-aria*. Hemos sugerido varios tipos de expresiones de navegación, con el fin de aprovechar las ventajas de la multiplicidad n-aria: desde un extremo hacia otro extremo, desde una combinación de n-1 extremos hacia otro extremo (usando una notación semejante a la que existe para asociaciones cualificadas), y desde un extremo hacia la asociación misma.

Aunque tenga sentido utilizar un *enlace n-ario como infraestructura de comunicación* entre los objetos enlazados, la comunicación en sí misma es un fenómeno intrínsecamente binario, es decir, un mensaje tiene exactamente un emisor y un receptor. Esto impide, por una parte, la emisión conjunta de un mensaje por parte de dos o más objetos, y, por otra parte, la recepción conjunta de mensajes. La representación de un mensaje "binario" enviado a través de un enlace n-ario en un diagrama de colaboración es algo problemática, pero hemos dado algunas reglas sencillas que pueden resolver el problema.

7.1.4. La visibilidad de las asociaciones

En el Capítulo 5 hemos considerado algunos problemas relativos a la visibilidad de las asociaciones. Como ésta está basada en la visibilidad de atributos y operaciones, ha sido necesario aclarar antes un par de cuestiones relativas a la visibilidad en UML en general. En primer lugar, *la visibilidad de los atributos y operaciones es una característica especificada para los clasificadores, no para las instancias*; es decir, la visibilidad no especifica si un objeto puede ver otro objeto, sino más bien si una propiedad de una clase puede ver otra propiedad de la misma o de otra clase. Por tanto, en el caso de que los objetos emisor y receptor pertenezcan a la misma clase, es posible enviar un mensaje que corresponda a una operación privada, aun cuando esta operación no pertenece a la "interfaz nativa" de la clase, formada por sus operaciones públicas. En segundo lugar, *los cuatro tipos de visibilidad existentes no son cuatro niveles progresivamente más restrictivos*, aunque la expresión "niveles de visibilidad" lo dé a entender, ya que a veces `protected` es más restrictivo que `package`, y en otras ocasiones es al revés.

Las definiciones de visibilidad de extremos de asociación contenidas en el Estándar son poco rigurosas; en su lugar hemos propuesto la siguiente: la *visibilidad de un extremo de asociación* especifica la visibilidad de la asociación desde el *punto de vista de otros clasificadores* al navegar la asociación en dirección hacia ese extremo. En UML los extremos de una asociación están equiparados a pseudo-atributos de los clasificadores participantes en la asociación, comprometiendo así la unidad de la asociación como elemento constructivo de los modelos. Consecuentemente, la visibilidad de los extremos de asociación se define de modo similar a la visibilidad de atributos y operaciones, con las mismas cuatro posibilidades, excepto en el caso de la visibilidad de paquete, para la cual se pierde el paralelismo. Para recuperar este paralelismo hemos propuesto una *nueva definición de visibilidad de paquete para extremos de asociación*, que no depende de dónde se define la asociación, sino de dónde se definen los clasificadores asociados. En todo caso, queda sin resolver el problema de la *ambivalencia* entre la "asociación" como concepto independiente de los clasificadores asociados, y la "referencia" como elemento incluido en ellos y más o menos equivalente a un atributo.

Por otra parte, la definición de asociaciones bidireccionales entre clasificadores pertenecientes a distintos paquetes resulta problemática. En efecto: en primer lugar, un elemento se define en un único paquete, y puede ser usado fuera de su paquete propietario, pero no modificado. En segundo lugar, una asociación navegable induce una dependencia desde el clasificador origen hacia el clasificador destino, dependencia que afecta a la definición del clasificador origen, de modo que debe ser definida en su mismo paquete; por tanto, el extremo navegable de una asociación debe ser definido en el mismo paquete que el clasificador opuesto. En tercer lugar,

no es posible definir una asociación bidireccional entre clasificadores de dos paquetes distintos, ya que esto exigiría definir la asociación (es decir, sus extremos) en cada uno de los dos paquetes, y una asociación debe ser definida en un único paquete, como cualquier otro elemento del modelo. Por lo tanto, *una asociación bidireccional sólo puede definirse entre clasificadores que pertenezcan al mismo paquete*. Aunque a primera vista puede parecer que esto es una grave limitación de UML, en realidad se trata de una consecuencia natural, aunque poco evidente, del concepto de paquete.

El recurso a la visibilidad de atributos y operaciones es insuficiente para conseguir un eficaz desacoplamiento entre los clasificadores de un modelo, ya que la visibilidad no discrimina entre las distintas asociaciones a las que está conectado un clasificador. Lo ideal sería disponer de *una interfaz distinta para cada asociación*, de modo que el clasificador conectado en el extremo opuesto tuviera una dependencia limitada a las propiedades incluidas en la interfaz. En UML hay dos mecanismos muy similares (incluso demasiado similares para que tenga sentido conservar la distinción) que permiten abordar este problema: especificador de interfaz, e interfaz propiamente dicha.

El *especificador de interfaz* da una solución parcialmente adecuada a este problema, especialmente si se adopta la definición matizada que hemos propuesto en este Capítulo; no obstante, la clase origen sigue dependiendo de la clase destino concreta, aunque sólo conozca de ella la parte que revela el especificador de interfaz. En cambio, mediante el uso de una *interfaz propiamente dicha* se consigue una plena independencia respecto a la clase destino concreta, que puede ser cualquier clase que realice la interfaz; así se consigue, además de especificar la funcionalidad requerida a través de la asociación, compartir el extremo de asociación entre varias clases sin necesidad de crear una superclase. Sin embargo, como una interfaz en UML especifica el comportamiento pero no la estructura del clasificador que la realiza, no está permitido que una interfaz participe en asociaciones bidireccionales, porque esto exigiría que la interfaz tuviera estructura. En todo caso, es razonable extender UML con una *noción menos restrictiva de interfaz* (algunos la denominan "rol") *que incluya tanto la especificación del comportamiento como la especificación del estado*. Efectivamente, el uso de interfaces con comportamiento y estado no sólo ayuda a expresar mejor la *interacción* requerida a través de una asociación, sino también la *estructura* requerida, logrando así una mejor integración de los aspectos estático y dinámico de la asociación. Esta nueva noción de interfaz implica así mismo *una nueva noción de compatibilidad o realización*, que incluya ambos aspectos.

En este sentido, y siguiendo a Steimann, hemos propuesto una *nueva definición de asociación que ya no se establece entre clasificadores sino entre interfaces*, pudiendo estar cada una de ellas realizada por uno o más

clasificadores y permitiendo así gran flexibilidad en el diseño. La interfaz en cada extremo especifica la estructura y el comportamiento que es posible conocer a través de la asociación en ese extremo concreto, y que deben satisfacer los objetos asociados. También hemos presentado una *notación que permite distintos niveles de complejidad* para expresar una misma asociación con mayor o menor grado de detalle; en los niveles más sencillos las interfaces no se muestran, o se muestran de forma reducida; en los niveles sucesivos se muestran con mayor detalle y se realza su papel en los extremos de la asociación.

Finalmente, hemos aplicado este nuevo concepto de asociación, mostrando cómo su uso permite la *simplificación de la visibilidad de atributos y operaciones*, y resuelve los inconvenientes de la visibilidad privada para asociaciones reflexivas. También hemos mostrado la relevancia de que una interfaz esté *conectada sólo a una asociación, o a más de una*.

7.1.5. La implementación de las asociaciones

En el Capítulo 6 hemos desarrollado una forma concreta de traducir las asociaciones UML a código Java: hemos escrito plantillas de código específicas, y hemos construido una herramienta que lee un modelo de diseño UML almacenado en formato XMI y genera los ficheros Java necesarios. Hemos prestado especial atención a tres propiedades principales de las asociaciones: multiplicidad, navegabilidad y visibilidad. Nuestro análisis ha encontrado dificultades que pueden revelar algunas debilidades de la especificación de UML.

En lo que respecta a la *multiplicidad*, hemos mostrado que en la práctica es imposible satisfacer en todo momento la restricción de multiplicidad mínima en un extremo de asociación obligatorio con unas pocas operaciones primitivas; nuestra propuesta es comprobar esta restricción sólo al acceder a los enlaces, pero no al modificarlos. El programador será responsable de usar las primitivas de modo consistente, de modo que se alcance cuanto antes un estado válido del sistema. Por el contrario, es posible asegurar el cumplimiento de la restricción de multiplicidad máxima en tiempo de ejecución, de modo que nuestra implementación la hace cumplir. Los extremos de asociación sencillos se almacenan fácilmente en atributos que tengan como tipo la clase destino, pero los extremos de asociación múltiples requieren el uso de colecciones para almacenar el correspondiente conjunto de enlaces; como las colecciones en Java están basadas en la clase estándar `Object`, es necesario realizar comprobación de tipos en tiempo de ejecución por medio de moldeado (*casting*) explícito cuando se usan colecciones como parámetros en los métodos mutadores.

En lo que respecta a la *navegabilidad*, las asociaciones unidireccionales son más fáciles de implementar por medio de atributos que las asociaciones bidireccionales, debido a las dificultades en la

sincronización de los dos extremos de la asociación. Una actualización en una asociación bidireccional debe ser ejecutada atómicamente en ambos extremos para mantenerla consistente; esto se consigue en el objeto origen solicitando una actualización recíproca en el objeto destino. Hemos considerado los pros y contras de una implementación alternativa, basada en el almacenamiento de "tuplas cosificadas", y finalmente la hemos descartado en favor de nuestro esquema de "referencias cruzadas sincronizadas". Una consecuencia derivada de nuestro análisis es que la restricción de multiplicidad en un modelo de diseño sólo puede ser especificada para un extremo de asociación navegable.

En lo que respecta a la *visibilidad*, en el caso de asociaciones unidireccionales se implementa con cierta facilidad simplemente traduciendo la visibilidad del extremo de la asociación a la visibilidad de los correspondientes métodos lectores y mutadores, porque los niveles de visibilidad de UML y Java tienen aproximadamente el mismo significado (salvo el caso de `protected` en Java, que equivale a la unión de las visibilidades `protected` y `package` de UML). Sin embargo, las asociaciones bidireccionales con uno o dos extremos privados (o protegidos) se comportan de modo paradójico, porque la actualización recíproca resulta imposible.

El código generado para cada asociación se localiza fácilmente dentro de las clases Java implicadas. Cada extremo de asociación presenta una *interfaz uniforme* para el programador. La interfaz es exactamente la misma para extremos de asociación unidireccionales y bidireccionales, pero hay ligeras diferencias para extremos de asociación sencillos y múltiples.

Nuestro enfoque es más bien exhaustivo en cuanto a las comprobaciones de invariantes. Pensamos que vale la pena hacer todo lo que se pueda por el programador, de modo que nuestra herramienta inserta código que realiza *comprobaciones de multiplicidad y tipo en tiempo de ejecución* y, por supuesto, que solicita las *actualizaciones recíprocas* en asociaciones bidireccionales. No obstante, la herramienta tiene distintas opciones que permitirán al usuario omitir las comprobaciones automáticas de multiplicidad y tipo al generar el código, en favor de la eficiencia. Además, hemos argumentado que las asociaciones unidireccionales no deberían tener restricciones de multiplicidad en el extremo origen en un modelo de diseño, y que las asociaciones bidireccionales no deberían tener ambos extremos con visibilidad privada (o protegida); por tanto, nuestra herramienta rechazará la generación de código para estas asociaciones. Una vez más, el usuario podrá desactivar esta comprobación de corrección del modelo y generar el código por su cuenta y riesgo.

7.2. Propuestas de modificación del Estándar de UML

Resumimos aquí brevemente las propuestas más importantes de modificación del Estándar de UML contenidas en esta Tesis Doctoral, remitiendo a la Sección correspondiente donde se detalla cada propuesta y su justificación.

7.2.1. Sobre la multiplicidad de las asociaciones

- **Definición de multiplicidad para las asociaciones n-arias (Sección 3.3).** Interpretación de *enlaces cojos* para la multiplicidad. Notación ampliada con *multiplicidades externa e interna*.
- **Supresión de la restricción de que no puede haber tuplas repetidas (Sección 3.5).** Un enlace *es una conexión* entre dos (o más) objetos, y *determina una tupla*; todos los enlaces de una asociación son distintos (en cuanto que tienen identidad propia y por tanto son distinguibles), pero dos o más enlaces pueden conectar los mismos objetos y así determinar la misma tupla (pueden tener el mismo contenido de datos).

7.2.2. Sobre la navegabilidad de las asociaciones

- **Definición de navegabilidad (Sección 4.2).** La navegabilidad es *la posibilidad de que un objeto origen designe un objeto destino a través de una asociación*, con el fin de manipularlo o acceder a él en una interacción con intercambio de mensajes. Alternativamente, la navegabilidad especifica *la capacidad que tiene una instancia de la clase origen de acceder a las instancias de la clase destino por medio de las instancias de la asociación* que las conectan.
- **Estereotipos de asociaciones y enlaces (Sección 4.5).** Distinción entre *asociaciones estructurales y contextuales*. Todo enlace es instancia de una asociación (estructural o contextual), y todo mensaje requiere un enlace de comunicación. Nuevas definiciones de los cinco estereotipos predefinidos para asociaciones y enlaces.

7.2.3. Sobre la visibilidad de las asociaciones

- **Definición de visibilidad de una asociación (Sección 5.2).** La visibilidad de un extremo de asociación especifica la visibilidad de la asociación *desde el punto de vista de otros clasificadores* al navegar la asociación en dirección hacia ese extremo. Definición de visibilidad de paquete para extremos de asociación. Prohibición de asociaciones bidireccionales entre clasificadores de distintos paquetes.
- **Asociaciones entre interfaces (Sección 5.4).** Una asociación es una relación definida entre interfaces; en cada extremo, la interfaz especifica la estructura y el comportamiento que es posible conocer navegando hacia ese extremo a través de la asociación, y que deben satisfacer los objetos asociados. Unificación de los conceptos de

nombre de rol, especificador de interfaz e interfaz propiamente dicha. Simplificación de la visibilidad de atributos y operaciones.

7.3. Otros resultados obtenidos

A continuación se resumen otros resultados menores que se han obtenido en esta Tesis Doctoral, remitiendo al Apartado correspondiente donde se explica cada problema en su contexto y la solución propuesta.

Ap.	**Problema**	**Solución**
3.4.3.	Un diagrama de objetos es una instancia de un diagrama de clases.	Los objetos y enlaces de un diagrama de objetos se conforman a clases y asociaciones del modelo, que pueden estar definidas en distintos diagramas.
4.2.2.	Una expresión de multiplicidad es una lista de intervalos enteros.	Las multiplicidad debería permitir la expresión de cualquier subconjunto de los enteros no negativos, tales como {números primos} o {cuadrados de enteros positivos}.
4.4.1.	La multiplicidad del extremo destino [de una asociación cualificada] denota las posibles cardinalidades del conjunto de instancias destino seleccionadas por el emparejamiento de una instancia origen y un valor de cualificador.	Redacción mejorada: Cuando el extremo origen de la asociación está cualificado, la multiplicidad en el extremo destino de la asociación denota cuántas instancias destino pueden estar asociadas con la combinación de una instancia origen y un valor de cualificador.
4.5.2.	No existe notación para los nombres de enlaces, pero un enlace es un elemento del modelo, con nombre heredado de la metaclase `ModelElement`.	Permitir la representación del nombre de cada enlace, como en el caso de los objetos.
5.3.1.	La dirección del nombre de la asociación representa la inherente ordenación de los extremos de la asociación. Los clasificadores en una asociación n-aria también están inherentemente ordenados, pero esta ordenación no puede expresarse gráficamente mediante una flecha de dirección de nombre.	Completar la notación de las asociaciones n-arias de modo que pueda expresarse la ordenación de los extremos.
5.3.1.	La dirección del nombre de la asociación puede apuntar en la dirección opuesta a la navegabilidad de la asociación.	Regla de estilo: En una asociación unidireccional la dirección del nombre de la asociación debe apuntar en la misma dirección que la navegabilidad.
5.3.2.	Se permite la especificación de multiplicidad y visibilidad en extremos de asociación no navegables.	Regla de estilo: El extremo origen de una asociación unidireccional no debería tener especificada ninguna propiedad, tal como multiplicidad, visibilidad, o nombre de rol.
6.2.1.	Un subpaquete es un paquete anidado dentro de otro paquete. Un subpaquete es un paquete descendiente de otro paquete.	Reservar el término “subpaquete” para referirse a la especialización de paquetes.

6.2.2.	Una asociación es poseída por el paquete que contiene el diagrama. Pero los diagramas no son elementos como tales de un modelo, y por tanto no están contenidos en ningún paquete.	Una asociación es poseída por el paquete que contiene su definición, y puede aparecer en diversos diagramas.
6.3.2.	El clasificador que realiza la interfaz se muestra unido a ella mediante el símbolo de realización, que es una línea discontinua terminada en un triángulo. Si la interfaz se representa como un círculo, la relación de realización puede representarse como una línea continua sin punta de flecha. Pero la realización es una dependencia estereotipada, y como tal admite sólo una representación gráfica.	Elegir una de las dos representaciones, o bien, suprimir la restricción de que un estereotipo sólo puede tener una representación gráfica.
6.3.2.	Una interfaz se puede representar como un rectángulo de clasificador con la palabra clave «`interface`». En este caso «`interface`» es una "palabra clave", no un "estereotipo", por tanto una interfaz no es una clase estereotipada, aunque la notación lo sugiera.	No usar la notación de estereotipos para las palabras clave.
6.3.2.	Una interfaz sólo contiene operaciones. Una interfaz sólo contiene operaciones y recepciones.	Resolver esta contradicción optando por la segunda definición, o mejor aún incluyendo también atributos y asociaciones.

7.4. Trabajos futuros

Esta Tesis Doctoral deja varios caminos abiertos que pueden ser proseguidos en ulteriores investigaciones. Ante todo, además de las tres propiedades principales de las asociaciones que han sido estudiadas en este trabajo (multiplicidad, navegabilidad y visibilidad), las asociaciones en UML tienen *otras características* que también pueden estudiarse en profundidad, tales como ordenación, modificabilidad, restricciones entre asociaciones, etc.

Las *asociaciones todo-parte* (agregación y composición) también plantean algunas dificultades que no han sido aún satisfactoriamente resueltas [Henderson-Sellers 99b]. Podemos señalar, en concreto, el posible carácter n-ario de una agregación sugerido por la representación en árbol [Génova 00], y la esquiva relación entre composición y encapsulamiento, donde entra en juego también el concepto de visibilidad. (¿Tiene el compuesto acceso exclusivo a los servicios que proporcionan las partes? O, por el contrario, ¿pueden otros objetos externos acceder a las partes "saltándose" la barrera del todo? En otras palabras, ¿las partes son visibles desde fuera del todo?)

Las *relaciones entre actores y casos de uso* también se modelan en UML como asociaciones (tanto actores como casos de uso son clasificadores), aunque son asociaciones de un carácter bastante distinto al de las asociaciones "normales" entre clases, ya que en ellas está ausente el aspecto estático. El autor de esta Tesis Doctoral ya tiene un trabajo preliminar sobre el tema [Génova 02a], en el que sin duda se puede profundizar más.

La *generalización de asociaciones* es tal vez un tema suficientemente amplio y complejo como para ocupar una entera tesis doctoral. Algunos trabajos ya han estudiado aspectos parciales del problema [McCarthy 97, Steimann 00, Steimann 01, Stevens 01], pero falta aún una solución global y elegante.

Respecto a la *multiplicidad de las asociaciones*, ya hemos mencionado (ver Apartado 7.1.2) que la interpretación de "enlaces cojos" deja abiertas algunas cuestiones, como cuántas patas pueden faltar en un enlace, cómo se deben interpretar las restricciones de multiplicidad en presencia de enlaces cojos, y si puede variar el grado de una asociación (es decir, el número de clasificadores asociados). La asociación cualificada también ha dejado algunas lagunas de interpretación.

La *navegabilidad de las asociaciones* ha sido estudiada principalmente en relación con el envío de mensajes que sean invocaciones de operaciones, pero en UML hay otra gran clase de mensajes, las señales, que merecen una mayor atención para conseguir una visión más completa de la cuestión. Así mismo, los mensajes de creación y destrucción de objetos plantean dificultades específicas que no hemos tratado (por ejemplo, en una asociación unidireccional, ¿cómo se entera el objeto origen que el objeto destino ha sido destruido?). El uso de enlaces n-arios como infraestructura de comunicación también deja planteadas preguntas muy interesantes (¿tienen sentido la emisión conjunta o la recepción conjunta de mensajes?).

En cuanto a la *visibilidad de las asociaciones*, recordemos que en UML hay cuatro tipos distintos de visibilidad, y en este trabajo nos hemos limitado al estudio de una sola de ellas, por tanto queda abierta la posibilidad de continuar el estudio también en esta línea. Nuestro análisis de la visibilidad de los extremos de asociación ha puesto de manifiesto la ambivalencia que existe en UML entre la "asociación" como concepto independiente de los clasificadores asociados, y la "referencia" como elemento incluido en ellos y más o menos equivalente a un atributo, lo que pone en peligro la unidad de las asociaciones bidireccionales; la investigación puede continuarse también en esta línea, enriquecida con la noción de asociación entre interfaces.

El trabajo realizado sobre la *implementación de asociaciones* puede continuarse en distintas líneas. Primero, implementación de otras propiedades de los extremos de asociación, tales como ordenación, modificabilidad, especificador de interfaz, asociaciones exclusivas, etc.

Segundo, implementación específica de algunos tipos particulares de asociaciones binarias tales como asociaciones reflexivas, agregaciones y composiciones. Tercero, implementación de asociaciones más complejas: asociaciones cualificadas, clases-asociación, y asociaciones n-arias, teniendo en cuenta la posibilidad de suprimir la restricción de que no puede haber tuplas repetidas. Cuarto, tener en cuenta la propuesta presentada en el Capítulo 5, donde una asociación no se define entre clases sino entre interfaces. Quinto, mejorar la herramienta para que lleve a cabo ingeniería inversa, es decir, obtener las asociaciones entre clases analizando el código que las implementa. Nuestra herramienta no lleva a cabo actualmente esta tarea, aunque es muy sencilla y directa si el código ha sido escrito con nuestras plantillas. Finalmente, adaptar la herramienta y las plantillas de modo que sigan la nueva *Java Metadata Interface (JMI) Specification* [JMI].

Finalmente, antes de un año se espera que aparezca la *versión 2.0* de UML. Es posible que algunas de las dificultades analizadas en esta Tesis queden resueltas en la versión 2.0, ya sea de modo independiente a nuestro trabajo, o bien como resultado directo de éste a través de la influencia de la iniciativa 3C [cUML] en la elaboración del nuevo Estándar. En todo caso, es más que probable que la nueva versión de UML deje cuestiones sin resolver o plantee otras nuevas.

8. Referencias

[Arnold 00] Ken Arnold, James Gosling, David Holmes. *The Java Programming Language.* Addison-Wesley, 3rd ed., 1998.

[Ambler 01] Scott W. Ambler. "An Overview of Object Relationships", "Unidirectional Object Relationships", "Implementing One-to-Many Object Relationships", "Implementing Many-to-Many Object Relationships". A series of tips to be found at *IBM Developer Works* (http://www-106.ibm.com/developerworks/).

[Batini 92] Carlo Batini, Stefano Ceri, Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin-Cummings, 1992.

[Barwell 02] Fred Barwell et al. *Professional VB.NET*, 2nd Edition. Wrox Press, 2002.

[Booch 94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, 1994.

[Castellani 00] Xabier Castellani, Henri Habrias, Philippe Perrin. "A Synthesis on the Definitions and Notations of Cardinalities of Relationships", *Journal of Object Oriented Programming*, 13(6):32-35, 2000.

[CDIF 96] Electronic Industries Association: *CASE Data Interchange Format, CDIF Integrated Meta-model, Data Modeling Subject Area*, EIA/IS-114, December 1996.

[Ceri 97] Stefano Ceri, Piero Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, 1997.

[Chen 76] Peter P. Chen. "The Entity-Relationship Model", *ACM Transactions on Database Systems*, 1(1):9-36, 1976.

[Coad 91] Peter Coad, Edward Yourdon. *Object-Oriented Analysis*, 2nd ed. Prentice-Hall, 1991.

[Codd 90] Edgar F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley, 1990.

[cUML] Financial Systems Architects (New York, U.S.A.). *3C-Clear Clean Concise*. Submission to OMG (http://www.community-ml.org/).

[Date 95] C.J. Date. *An Introduction to Database Systems*, 6th ed. Addison-Wesley, 1995.

[De Miguel 99] Adoración De Miguel, Marco Piattini, Esperanza Marcos. *Diseño de bases de datos relacionales*. Ra-Ma, Madrid, 1999.

[DRAE 92] Real Academia Española. *Diccionario de la Lengua Española*, 21ª ed. Madrid, 1992.

[D'Souza 99] Desmond F. D'Souza, Alan C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[dTinf 02] dTinf, Desarrollos para las Tecnologías de la Investigación. *CAKE-Computer Aided Knowledge Engineering*. Proyecto de Investigación I+D en colaboración con la Universidad Carlos III de Madrid (Art. 11 de la Ley 11/83 de 25 de Agosto de Reforma Universitaria). Investigador responsable: Juan Llorens Morillo. Código de proyecto 1269. Años: 2001-2003.

[Dullea 98] James Dullea, Il-Yeol Song. "An Analysis of Structural Validity of Ternary Relationships in Entity-Relationship Modeling", *Proceedings of the 7th International Conference on Information and Knowledge Management*, 331-339, Washington, D.C., November 3-7, 1998.

[Eckel 00] Bruce Eckel. *Thinking in Java*, 2nd ed. Prentice-Hall, 2000.

[Elmasri 94] Ramez Elmasri, Shamkant B. Navathe. *Fundamentals of Database Systems*, 2nd ed. Benjamin-Cummings, 1994.

[Embley 98] David W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, 1998.

[Firesmith 98] Donald G. Firesmith, Brian Henderson-Sellers. "Upgrading OML to Version 1.1: Part 2-Additional Concepts and Notation". *Journal of Object Oriented Programming*, 11(15):61-67, 1998.

[Fowler 97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[Fowler 00] Martin Fowler, Kendall Scott. *UML Distilled, A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2000. First edition published as: *UML Distilled, Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[Fujaba] *The Fujaba CASE Tool*, University of Paderborn (http://www.fujaba.de/).

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable Object-Oriented software.* Addison-Wesley, 1994.

[Genilloud 98] Guy Genilloud. "Common Domain Objects in the RM-ODP Viewpoints", *Computer Standards and Interfaces*, 19(7):361-374, 1998.

[Genilloud 99] Guy Genilloud. "Informal UML 1.3 - Remarks, Questions, and some Answers". *UML Semantics FAQ Workshop* (held at ECOOP'99), Lisbon, Portugal, June 12, 1999.

[Genilloud 00] Guy Genilloud, Alain Wegman. "A New Definition for the Concept of Role, and Why it Makes Sense". In *Proceedings of the Ninth OOPSLA Workshop on Behavioral Semantics, OOPSLA'2000*, Minneapolis, Minnesota, October 15-19, 2000.

[Génova 00] Gonzalo Génova, Juan Llorens, José Miguel Fuentes, Jorge Morato, Paloma Martínez. "Conceptual Hierarchies in UML: Comparing ISO 2788 Standard with the UML Metamodel". *Workshop on Defining Precise Semantics for UML, The 14th European Conference on Object-Oriented Programming-ECOOP'2000*, June 12-16, 2000, Sophia Antipolis-Cannes, France.

[Génova 01] Gonzalo Génova. "Semantics of Navigability in UML Associations". *Informe Técnico UC3M-TR-CS-2001-06*, Departamento de Informática de la Universidad Carlos III de Madrid, noviembre 2001, pp. 233-251.

[Génova 02a] Gonzalo Génova, Juan Llorens, Víctor Quintana. "Digging into use case relationships". *The 5th International Conference on the Unified Modeling Language-UML'2002*, September 30-October 4 2002, Dresden, Germany. Published in *Lecture Notes in Computer Science 2460*, Springer 2002, pp. 115-127.

[Génova 02b] Gonzalo Génova, Juan Llorens, Paloma Martínez. "The Meaning of Multiplicity of N-ary Associations in UML", *Journal on Software and Systems Modeling*, 1(2): 86-97, 2002. Una versión preliminar de este artículo fue publicada como: Gonzalo Génova, Juan Llorens, Paloma Martínez. "Semantics of the Minimum Multiplicity in Ternary Associations in UML". *The 4th International Conference on the Unified Modeling Language-UML'2001*, October 1-5 2001, Toronto, Ontario, Canada. Publicado en *Lecture Notes in Computer Science 2185*, Springer 2001, pp. 329-341.

[Génova 03a] Gonzalo Génova, Juan Llorens, Vicente Palacios. "Sending Messages in UML", *Journal of Object Technology*, vol.2, no.1, Jan-Feb 2003, pp. 99-115, http://www.jot.fm/issues/issue_2003_01/article3. El contenido de este artículo se encuentra en forma primitiva en [Génova 01], que también trata de otros temas.

[Génova 03b] Gonzalo Génova, Juan Llorens, Carlos Ruiz del Castillo. "Mapping UML Associations into Java Code", *Journal of Object Technology* (to be published in the Sep/Oct 2003 issue).

[Génova 03c] Gonzalo Génova, Juan Llorens, José Miguel Fuentes. "UML Associations: A Structural and Contextual View". *The 6th International Conference on the Unified Modeling Language-UML'2003*, 20-24 Octubre, 2003, San Francisco, California, Estados Unidos. Enviado para revisión.

[Gosling 96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Graham 97a] Ian Graham, Julia Bischof, Brian Henderson-Sellers. "Associations Considered a Bad Thing". Journal of Object Oriented Programming, 9(9):41-48, 1997.

[Graham 97b] Ian Graham, Brian Henderson-Sellers, Houman Younessi. *The OPEN Process Specification*. Addison-Wesley, 1997.

[Hamie 98] Ali Hamie, John Howse, Stuart Kent: "Navigation Expressions in Object-Oriented Modelling". In *Proceedings of FASE'98 in ETAPS'98.* Published in *Lecture Notes in Computer Science 1382*, pp. 123-137, Springer-Verlag, 1998.

[Harrison 00] William Harrison, Charles Barton, Mukund Raghavachari. "Mapping UML designs to Java". *The 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications-OOPSLA'2000*, October 15-19 2000, Minneapolis, Minnesota, United States. *ACM SIGPLAN Notices*, 35(10): 178-187. ACM Press, New York, NY, USA.

[Henderson-Sellers 99a] Brian Henderson-Sellers, Donald G. Firesmith. "Comparing OPEN and UML: The Two Third-Generation OO Development Approaches", *Information and Software Technology*, 41(3):139-156, 1999.

[Henderson-Sellers 99b] Brian Henderson-Sellers, Frank Barbier, "Black and White Diamonds", *Proc 2nd IEEE Conference on UML: UML'99*, LNCS 1723, pp. 550-565, 1999.

[Hitchman 99] Steve Hitchman. "Ternary Relationships--To Three or not to Three, Is there a Question?" *European Journal of Information Systems*, 8:224-231, 1999.

[Jackson 00] Daniel Jackson. *Lab in Software Engineering--Lecture Notes*. Massachusetts Institute of Technology, 2000.

[Jacobson 92] Ivar Jacobson. *Object-Oriented Software Engineering: a Use Case Driven Approach*. Addison-Wesley, 1992.

[JMI] Java Community Process. *Java Metadata Interface (JMI) Specification*, Version 1.0, June 2002. Available at http://www.jcp.org/.

[Jones 95] Trevor H. Jones, Il-Yeol Song. "Binary Representations of Ternary Relationships in ER Conceptual Modeling", *14th International Conference on Object-oriented and Entity-Relationship Approach*, pp. 216-225, Gold Coast, Australia, December 12-15, 1995.

[Jones 96] Trevor H. Jones, Il-Yeol Song. "Analysis of Binary/Ternary Cardinality Combinations in Entity-Relationship Modeling", *Data & Knowledge Engineering*, 19(1):39-64, 1996.

[Jones 00] Trevor H. Jones, Il-Yeol Song. "Binary Equivalents of Ternary Relationships in Entity-Relationship Modeling: a Logical Decomposition Approach", *Journal of Database Management*, April-June:12-19, 2000.

[Kaindl 99] Hermann Kaindl. "Difficulties in the Transition from OO Analysis to Design". *IEEE Software*, 16(5):94-102, 1999.

[Kilov 94] Haim Kilov, James Ross. *Information Modeling: An Object-Oriented Approach*. Prentice Hall, 1994.

[Lemay 00] Laura Lemay, Rogers Cadenhead. *Sams Teach Yourself Java 2 in 21 Days*, 2nd ed. Sams Publishing, 2000.

[Lieberherr 89] Karl J. Lieberherr, Ian M. Holland. "Assuring Good Style for Object-Oriented Programs". *IEEE Software*, 6(5): 38-48, 1989.

[MagicDraw] *MagicDraw UML Professional 7.0*, No Magic, Inc. (http://www.magicdraw.com/).

[Martin 95] James Martin, James Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall, 1995.

[Martínez 99] Paloma Martínez, Carlos Nieto, Dolores Cuadra, Adoración De Miguel. "Profundizando en la semántica de las cardinalidades en el modelo E/R extendido", *IV Jornadas de Ingeniería del Software y Bases de Datos,* pp. 53-54, Cáceres, Spain, November 24-26, 1999.

[McAllister 95] Andrew McAllister. "Modeling N-ary Data Relationships in CASE Environments", *Proceedings of the 7th International Workshop on Computer Aided Software Engineering*, pp. 132-140, Toronto, Canada, 1995. A more recent version in: "Complete Rules for N-Ary Relationship Cardinality Constraints", *Data & Knowledge Engineering*, 27(3):255-288, 1998.

[McCarthy 97] Brendan McCarthy. "Association Inheritance and Composition", *Journal of Object Oriented Programming*, 10(4): 69-81, 1997.

[MegaSuite] *Mega Suite 6.0*, Mega International (http://www.mega.com/).

[Merriam-Webster] Merriam-Webster OnLine Dictionary (http://www.m-w.com/).

[Métrica 93] *Metodología de planificación y desarrollo de sistemas de información, METRICA versión 2. Tomo 3: Guía de técnicas*. Instituto Nacional de Administración Pública, España. Madrid, 1993.

[Miller 01] Joaquin Miller. Comunicación personal al autor, 13 diciembre, 2001.

[Muller 97] Pierre A. Muller. Instant UML. Wrox Press, 1997.

[Mylopoulos 02] John Mylopoulos. *Conceptual Modeling, The Unified Modeling Language--Lecture Notes*. University of Toronto, 2002 (http://www.cs.toronto.edu/~jm/2510S/Lectures.html).

[Noble 96] James Noble. "Some Patterns for Relationships". In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*, Melbourne, Australia, 1996.

[Noble 97] James Noble. "Basic Relationship Patterns". In *Proceedings of the European Conference on Pattern Languages of Program Design (EuroPLOP'97)*. Irsee, Germany, 1997.

[OCL] Jos B. Warmer, Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[OMG] Object Management Group (http://www.omg.org/).

[pUML] The Precise UML Group (http://www.cs.york.ac.uk/puml/).

[RationalRose] *Rational Rose Professional 2000*, Rational Software Corporation (http://www.rational.com/).

[Rhapsody] *The Rhapsody CASE Tool*, ILogix (http://www.ilogix.com/).

[RM] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Ruiz 02] Carlos Ruiz del Castillo. *Implementación en Java de asociaciones binarias UML*. Universidad Carlos III de Madrid, Proyecto Fin de Carrera, Ingeniería Informática (Segundo Ciclo), Julio 2002. Tutor: Gonzalo Génova.

[Rumbaugh 87] James Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language", In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pp. 466-481, Orlando, Florida, 1987.

[Rumbaugh 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[Rumbaugh 96a] James Rumbaugh. "Models for Design: Generating Code for Associations". *Journal of Object Oriented Programming*, 8(9):13-17, 1996.

[Rumbaugh 96b] James Rumbaugh. "A Search for Values: Attributes and Associations". *Journal of Object Oriented Programming*, 9(3):6-8, 1996.

[Rumbaugh 98] James Rumbaugh. "Depending on Collaborations: Dependencies as Contextual Associations". *Journal of Object Oriented Programming*, 11(4):5-9, 1998.

[Schrefl 91] Michael Schrefl, Gerti Kappel. "Cooperation Contracts". In T.J. Teorey (ed.), *Proceedings of The 10th International Conference on the Entity Relationship Approach*, San Mateo, California, October 1991, pp. 285-307.

[Schrefl 96] Michael Schrefl, Gerti Kappel, Peter Lang. "Modeling Cooperative Behavior Using Cooperation Contracts". Technical Report 02.96, Institut für Wirtschaftsinformatik, Johannes Kepler Universität Linz, Austria, 1996.

[Simons 99] Anthony J.H. Simons, Ian Graham: "30 Things That Go Wrong in Object Modelling with UML 1.3". Chapter 17 in Haim Kilov, Bernhard Rumpe, Ian Simmonds (eds.): *Behavioral Specifications of Businesses and Systems*, pp. 237-257. Kluwer Academic Publishers, 1999.

[Simons 02] Anthony J.H. Simons. "The Theory of Classification, Part 1: Perspectives on Type Compatibility". *Journal of Object Technology*, vol. 1, no. 1, May-June 2002, pp. 55-61, http://www.jot.fm/issues/issue_2002_05/column5.

[Song 93] Il-Yeol Song, Trevor H. Jones. "Analysis of binary relationships within ternary relationships in ER Modeling", *Proceedings of the 12th International Conference on Entity-Relationship Approach*, pp. 265-276, Dallas, Texas, December 15-17, 1993.

[Song 95] Il-Yeol Song, Mary Evans, E.K. Park. "A Comparative Analysis of Entity-Relationship Diagrams", *Journal of Computer and Software Engineering*, 3(4):427-459, 1995.

[Steimann 00] Friedrich Steimann. "A Radical Revision of UML's Role Concept". *The Third International Conference on the Unified Modeling Language-UML'2000*, October 2-6, 2000, York, United Kingdom. Published in *Lecture Notes in Computer Science 1939*, Springer 2000, pp. 194-209.

[Steimann 01] Friedrich Steimann. "Role = Interface: A Merger of Concepts". *Journal of Object Oriented Programming*, 14(4):23-32, 2001.

[Steimann 02] Friedrich Steimann, Thomas Kühne. "A Radical Reduction of UML's Core Semantics". *The Fifth International Conference on the Unified Modeling Language-UML'2002*, September 30 - October 4, 2002, Dresden, Germany. Published in *Lecture Notes in Computer Science 2460*, Springer 2002, pp. 34-48.

[Stevens 00] Perdita Stevens, Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 2000.

[Stevens 01] Perdita Stevens. Comunicación personal al autor, 1 Agosto 2001.

[Stevens 02] Perdita Stevens. "On the Interpretation of Binary Associations in the Unified Modelling Language", *Journal on Software and Systems Modeling*, 1(1):68-79, 2002. A preliminar version in: Perdita Stevens. "On Associations in the Unified Modeling Language". *The Fourth International Conference on the Unified Modeling Language-UML'2001*, October 1-5, 2001, Toronto, Ontario, Canada. Published in *Lecture Notes in Computer Science 2185*, Springer 2001, pp. 361-375.

[Switzer 93] Robert Switzer. *Eiffel, an introduction*. Prentice-Hall, 1993.

[Tanzer 95] Christian Tanzer. "Remarks on object-oriented modeling of associations". Journal of Object Oriented Programming, 7(9):43-46, 1995.

[Tardieu 85] Hubert Tardieu, Arnold Rochfeld, René Coletti. *La Méthode MERISE. Tome 1: Principles et Outils*. Les Editions d'Organisation, Paris, 1983, 1985.

[Teorey 99] Toby J. Teorey. *Database Modeling and Design*, 3rd ed, Morgan Kaufmann Publishers, 1999.

[Turing 36] Alan Turing. "On Computable Numbers, With an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society*, Series 2, Volume 42, 1936. Reprinted in M. David (ed.), *The Undecidable*, Hewlett, NY: Raven Press, 1965.

[UG] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[UML] Object Management Group. *Unified Modeling Language Specification*, Version 1.4, September 2001 (http://www.omg.org/).

[UML11] Rational Software Corporation, *Unified Modeling Language Semantics,* Version 1.1, September 1997. Rational Software Corporation, *Unified Modeling Language Notation Guide,* Version 1.1, September 1997.

[UML13] Object Management Group. *Unified Modeling Language Specification*, Version 1.3, June 1999 (http://www.omg.org/).

[UMLConf] International Conferences on the Unified Modeling Language (http://www.umlconference.org/).

[UP] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Velho 94] Amândio V. Velho, Rogerio Carapuça. "Attribute: A Semantic and Seamless Construct". In TOOLS 13 (ed. B. Magnusson et al.), Prentice Hall, 1994.

[VisualParadigm] *Visual Paradigm for UML 2.1*, Visual Paradigm (http://www.visual-paradigm.com/).

[XMI] Object Management Group. *XML Metadata Interchage (XMI) Specification*, Version 1.2, January 2002 (http://www.omg.org/).