Departamento de Informática
Universidad de Valladolid
Valladolid - España

Transformación de Jerarquías de Herencia Múltiple en Jerarquías de Herencia Sencilla*

Juan José Rodríguez, Yania Crespo, y José Manuel Marqués

{juanjo, yania, jmmc}@infor.uva.es

Resumen En este trabajo se analizan situaciones en las que se plantea la necesidad de transformar esquemas que utilizan jerarquías de herencia múltiple en esquemas "equivalentes" que utilicen herencia sencilla. En este contexto, se discuten diferentes aproximaciones básicas al problema de desarrollar un método automático que ejecute dicha transformación. Como resultado de la discusión, se concluye que ninguna de las estrategias básicas analizadas es completamente satisfactoria. En consecuencia, se proponen soluciones alternativas resultado de combinar adecuadamente los mecanismos básicos presentados. Estas soluciones se denominarán estrategias combinadas. Las propuestas se presentan en dos casos. El primero explica unas soluciones parciales y más comprensibles que se adaptan a lenguajes tipo Java, aprovechando la presencia de herencia simple de clases y múltiple de interfaces. Posteriormente se avanza hacia la presentación de una estrategia combinada más general.

Informe Técnico DI-2000-0002

* Este trabajo fue presentado como Informe Técnico del grupo GIRO en Octubre de 1998, identificado con el número TR-GIRO-98.

1 Introducción y Motivación

El diseño es la representación de conocimiento para modelar elementos del mundo relacionados con el dominio de aplicación. El método de Diseño Orientado a Objetos (DOO) para expresar este conocimiento del dominio se basa en modelar la realidad a través de clases y relaciones entre ellas. Una de las relaciones fundamentales que aparece en este contexto es la relación de herencia.

Las jerarquías de herencia son la piedra angular del DOO. Las metodologías más reconocidas para Análisis y Diseño Orientado a Objetos [Boo94, BRJ96, RBP91, WN95] recomiendan la utilización de herencia múltiple para modelar objetos del dominio de la aplicación.

La herencia múltiple es esencial para obtener todos los beneficios del método orientado a objetos y para diseñar modelos más cercanos al mundo real [Mey97]. Esta puede verse como un medio de aumentar la expresividad de los diseños. Santas argumenta en [San92] con ejemplos de computación simbólica la expresividad y claridad que aporta el poder modelar utilizando herencia múltiple.

Un buen modelo debe ser lo más natural posible, lo que significa: tan cercano como sea posible al sistema que debe describir. Esto es esencial si se quieren obtener descripciones que destaquen por su claridad y su legibilidad [Rüp92]. En el método Orientado a Objetos (OO) la herencia múltiple juega un importante papel para lograr esto. Aún así la herencia múltiple es controvertida y algunos autores argumentan que no es necesaria para hacer buenos diseños.

Si se aceptan la utilidad y beneficios de modelar utilizando herencia múltiple, ya sea por un diseñador o como resultado del trabajo de una herramienta, nos encontramos ante el problema de implementar los modelos. Existen varios lenguajes OO que no soportan herencia múltiple como construcción del lenguaje lo cual implica que al intentar representar estos diseños en dichos lenguajes hay que encontrar **estrategias de transformación**. Es deseable que estos mecanismos sean tales que no incrementen más aun la distancia entre el modelo de diseño y su implementación lo cual es muy importante para soportar un desarrollo reversible y sin discontinuidad[1]. En este punto también hay quienes declaran que todos los buenos lenguajes soportan herencia múltiple.

Hablar de buenos lenguaje denota un marcado "espíritu partidista". No existe un lenguaje perfecto para todas nuestras necesidades y en ocasiones se hace necesario migrar hacia un lenguaje con herencia sencilla.

Hay varios lenguajes y/o entornos de programación OO, algunos de ellos muy utilizados, que no soportan herencia múltiple como Delphi [Del], Java [AG96, GJS96] y Modula-3 [CDG+89], por ejemplo. Generalmente la causa de no incluir la herencia múltiple de clases en los lenguajes viene motivada por la complicación que puede significar, para los diseñadores y/o implementadores del lenguaje, resolver los conflictos que aparecen con esta.

En la figura 1 se representan algunos ejemplos de lenguajes categorizados según la presencia de la relación de herencia como construcción del lenguaje. Nótese que se habla de lenguajes basados en clases. CLU es un lenguaje basado

[1] del término inglés seamlessness

en clases (objetos + clases) y no incluye el mecanismo de la herencia. Por esta razón Wegner [Weg87] lo clasifica como lenguaje basado en clases y no como un lenguaje Orientado a Objetos (objetos + clases + herencia). Delphi es un entorno de trabajo basado en el lenguaje Orientado a Objetos Object Pascal del cual adquiere la característica de que sólo permite herencia simple (Modula-3 es otro ejemplo de este mismo caso). Eiffel [Mey92], C++ [Str97] y CLOS [Ste90] son ejemplos de lenguajes que soportan herencia múltiple de clases con formas diferentes de resolver los conflictos que se presentan producto de esta.

Java es un lenguaje que separa en dos estructuras, interfaces y clases, las nociones de tipo y clase como implementación de un tipo, respectivamente. Para las interfaces Java (que pueden verse como una forma muy pobre de representar tipos abstractos) se admite una derivación múltiple de interfaces mientras que para las clases se permite sólo una derivación simple[2]. Una clase puede heredar de una sola clase pero en cambio puede implementar múltiples interfaces. Por otra parte, Kaleidoscope [FBB92] es un ejemplo de lenguaje que igualmente separa la noción de clases y tipos pero soportando "herencia" múltiple de ambos.

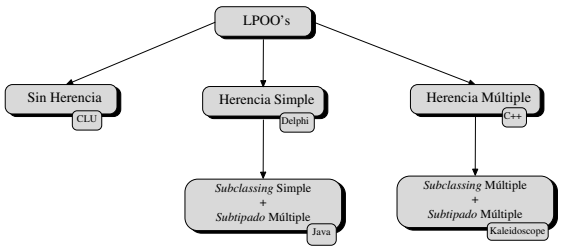

Figura1. Algunos lenguajes según la presencia de herencia.

Se pueden describir 3 situaciones generales en las cuales sería útil disponer de mecanismos de transformación de herencia múltiple a herencia sencilla:

- Para extender un lenguaje con herencia sencilla a una versión con herencia múltiple.
- Para implementar en un lenguaje con herencia sencilla un diseño que utiliza herencia múltiple cuando, por ejemplo, los requisitos del problema demandan su implementación en dicho lenguaje o cuando se intenta reutilizar un diseño.
- Para traducir de un lenguaje con herencia múltiple a uno con herencia sencilla. Actualmente se tiene un ejemplo típico de este caso: la traducción de programas o bibliotecas de C++ a Java.

En nuestra opinión la primera situación es de especial interés porque teniendo ésta resuelta, el lenguaje extendido puede utilizarse como un resultado interme-

[2] En algunas fuentes bibliográficas se utilizan los términos en inglés: *simple subclassing - multiple subtyping* para referirse a las características de Java de tener herencia simple de clases y múltiple de interfaces.

dio en los otros dos casos. Dicho lenguaje extendido puede implementarse a través de un pre-procesador que utilice alguna(s) estrategia(s) de transformación de las que aquí se proponen y obtiene una traducción al lenguaje original. Las ventajas (e inconvenientes) de los pre-procesadores en estos casos son bien conocidas. Los pre-procesadores son mucho más sencillos que los compiladores para los lenguajes extendidos. Cuando la definición del lenguaje de partida evoluciona, se necesita actualizar los compiladores pero no siempre es necesario actualizar los pre-procesadores (ver caso del pre-procesador C/C++) a no ser que la nueva definición del lenguaje invalide algunos supuestos que asuma el pre-procesador. Varios pre-procesadores pueden utilizarse para extender el lenguaje original en diferentes aspectos, pero sólo un compilador.

La segunda situación es especialmente interesante para la Ingeniería de Software por la implicación que puede tener en las herramientas CASE y los entornos de prototipación automática así como en entornos que asistan a la reutilización desde diferentes niveles de abstracción. En este caso, el pre-procesador del que se habla anteriormente sería útil porque:

- Puede incorporarse en herramientas CASE de DOO. Con una de estas herramientas, el usuario dibuja las clases y sus relaciones (incluyendo las de herencia). Los cuerpos de los métodos pueden introducirse directamente sobre la representación de las clases en un diseño detallado. Es habitual que estas herramientas permitan dibujar varios enlaces de herencia para el mismo descendiente (es decir, modelar herencia múltiple), pero se "quejan" cuando se trata de generar código para un lenguaje que no soporta herencia múltiple. Nuestra propuesta radica en que tanto la representación de las clases, como los cuerpos de los métodos para el diseño detallado en una herramienta CASE puedan incluir construcciones de herencia múltiple que serán soportadas por el pre-procesador el cual generará código para el lenguaje objeto de la extensión. Como resultado puede producirse un diseño detallado con herencia sencilla y los cuerpos de los métodos codificados en el lenguaje objetivo que no tiene herencia múltiple.
- Puede utilizarse en entornos de prototipación automática, si estos entornos utilizan herencia múltiple como una forma de especificación (o alguna otra forma de modelado que conduzca a una solución de diseño utilizando herencia múltiple) y se quiere generar prototipos para lenguajes con herencia sencilla. Un ejemplo destacado se tiene en los trabajos realizados en la Universidad Politécnica de Valencia en el marco del proyecto MENHIR donde a partir de un modelo conceptual (hasta el momento para el caso de aplicaciones de gestión) según la notación y conceptos del método OO-METHOD [PIP+97] que responde al modelo formal OASIS 3.0 [LRSP98] se genera el código de un prototipo. En [Pel98] se describen en forma de patrones de modelado y de generación de código algunas soluciones *ad-hoc* para atacar este problema.
- Puede incorporarse en entornos que ayuden a la reutilización desde diferentes niveles de abstracción. En este caso se podría reutilizar un DOO que utiliza herencia múltiple para otro proyecto en el cual el lenguaje destino sea un lenguaje OO sin herencia múltiple. Esto además potenciaría la funcionalidad

de aquellos repositorios o bibliotecas de *assets*. Un ejemplo de este caso se tendría en el repositorio basado en estructuras complejas de reutilización (Mecanos) que se desarrolla en la Universidad de Valladolid en el marco del sub-proyecto MECANO [GMM98] del proyecto MENHIR.

Es posible aplicar la transformación a esquemas de diseños no detallados, sin cuerpos de métodos, pero este caso es menos interesante y también menos manejable porque el usuario tendría que lidiar directamente en tiempo de implementación con la jerarquía resultante, lo cual podría llegar a ser bastante complejo.

En el caso de herramientas que a partir de las especificaciones intenten generar código, también el caso más útil a analizar es el que incluye no sólo el modelado de las clases como estructuras sino además la codificación de los métodos. Esto ofrece a los desarrolladores de dichas herramientas una guía de aspectos a tener en cuenta al realizar su transformación del modelo conceptual a la implementación.

En este trabajo se toma una aproximación general, es decir, se tratan de describir mecanismos independientes del lenguaje destino (con excepción del caso de Java). En todo caso, una vez claros los aspectos generales siempre se pueden llevar a cabo "optimizaciones" de acuerdo a características específicas de un lenguaje en particular.

Nuestra posición respecto a las características de un mecanismo de transformación radica en que un mecanismo ideal debería:

- Mantener en lo posible la clasificación jerárquica a través de la herencia que se ha modelado en el diseño.
- Respetar el comportamiento y las asignaciones polimórficas, aún cuando el lenguaje sea fuertemente tipado.
- No incurrir (o al menos lo menos posible) en problemas de excesiva repetición de código y de mantenimiento de coherencia.

Estos aspectos se marcarán como objetivos a lograr cuando se define un mecanismo de transformación de jerarquías de herencia múltiple a jerarquías de herencia sencilla.

El resto del trabajo está organizado de la siguiente forma: en la sección 2 se introducen y discuten 4 aproximaciones básicas preliminares para transformar jerarquías de herencia múltiple: emancipación, composición, expansión y tipo *variant* o su simulación mediante banderas. En la sección 3 se combinan las aproximaciones anteriores para proponer soluciones que se acerquen más a los objetivos propuestos. A estas estrategias se les denominará estrategias combinadas. Primero se analiza una propuesta particular para Java como lenguaje objetivo aprovechando sus propiedades que hacen más sencilla la transformación (interfaces organizadas en jerarquías múltiples) para luego dar paso a partir de aquí a una estrategia combinada más general para lenguajes que no tienen dicha propiedad. El caso específico realizado para Java sirve de ejemplo de cómo se pueden adaptar los métodos generales para aprovechar características particulares y, por otra parte, sirve como apoyo didáctico a la presentación para

pasar de este caso al caso general. En la sección 4 se discuten algunos trabajos relacionados con este. Finalmente en la sección 5 se presentan las conclusiones remarcando algunos aspectos significativos.

2 Transformaciones básicas

El problema de transformar una jerarquía de herencia múltiple en una jerarquía de herencia sencilla "equivalente" puede atacarse utilizando las siguientes estrategias básicas:

- Emancipación: todas las relaciones de herencia de una clase se eliminan, y todas las propiedades que eran heredadas se incluyen como recursos propios. Esta estrategia es similar, pero algo más compleja porque el código es importante, a aplicar la operación de "aplastar" (*flattening*) para obtener la forma *flat* de una clase [Mey90, Mey97].
- Composición: transformar las relaciones de herencia en relaciones de composición. De acuerdo con Meyer en [Mey97] cuando una clase B necesita una propiedad que tiene otra clase A, existen dos posibilidades: £Es B un heredero de A, o un cliente de A? Aunque hay una marcada diferencia entre las relaciones *is-a* y *has-a*, algunas veces puede cambiarse una relación *is-a* por una *has-a*, con la pérdida de los beneficios de la herencia que esto implica como se ve en la sección 2.5.
- Expansión: el grafo de herencia múltiple se expande en un árbol (en un bosque, en general). Luego, en el nuevo grafo aparecen solamente relaciones de herencia sencilla. Ver una descripción y discusión más detallada en la sección 2.6.
- Tipos *Variant* o su Simulación en una clase monitora mediante banderas: a partir de las clases que son raíces de la jerarquía se crea para cada una, una estructura compleja que incluye todas las propiedades de sus clases descendientes y las despacha de acuerdo al objeto en curso. Dicha estructura consiste en un tipo *variant* (en el caso de poder expresar tipos *variant* como recurso del lenguaje) o en una clase monitora que simula un tipo *variant* utilizando banderas. A diferencia de la emancipación no se obtiene para cada clase inicial una clase "aplastada" sino una única estructura compleja a través de la cual se pueden describir objetos de una u otra clase en dependencia de una condición.

Desde nuestro punto de vista, ninguna de estas estrategias básicas son completamente satisfactorias por sí solas debido a que la transformación implica la pérdida de demasiada información de diseño y se incumplen más de uno o incluso todos los objetivos mencionados en el apartado anterior. Es por esto que en la sección 3 se pasa a unas estrategias combinadas con el fin de acercarse lo más posible a los objetivos planteados para un mecanismo de transformación como el que nos ocupa.

En la figura 2 se muestra un esquema en el que se asocian los lenguajes OO (basados en clases) con los mecanismos de transformación que pudieran aplicarse

para cada caso. La línea de puntos indica que para esos lenguajes siempre es posible aplicar los métodos anteriores aunque no se aprovechen totalmente los recursos que estos tienen. Para el primer grupo sólo pueden aplicarse los métodos directamente asociados con él, mientras que para los siguientes pudieran aplicarse los anteriores y los que referencian directamente. En la última sección se harán algunas consideraciones al respecto de la utilización de unos u otros métodos.

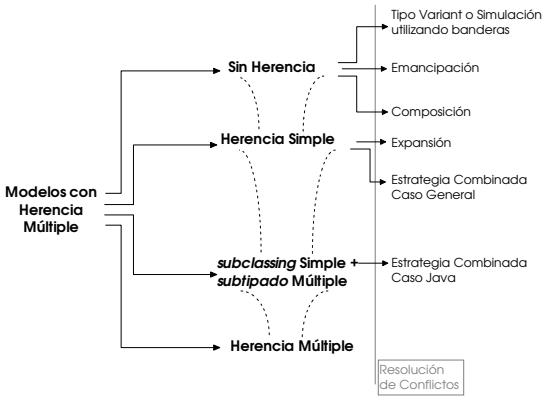

Figura2. Los lenguajes frente a los mecanismos de transformación según sus características.

2.1 Punto de Partida

El punto de partida es una jerarquía de herencia múltiple inicial, construida en un entorno de programación o de modelado que soporte herencia múltiple. A partir de esto, se obtendrá el grafo de trabajo G eliminando conflictos de la jerarquía inicial de acuerdo con una especificación de resolución de conflictos. Este grafo se definirá en el presente apartado.

Cuando se utiliza herencia múltiple siempre se debe tener en cuenta la posible presencia de conflictos tales como el dilema de la herencia repetida, la introducción de ambigüedades, la herencia de propiedades contradictorias y/o la selección polimórfica de métodos presentes en diferentes caminos de herencia [LNS91, Mey97].

En [DHHM94] las aproximaciones básicas a la resolución de conflictos se clasifican en resolución dirigida y resolución automática. En el primer caso se engloba resolución explícita (C++), renombrado y selección (Eiffel), etc. En el segundo se encuentran las técnicas de linearización (CLOS, LOOPS[SB86]). Ambos casos generales son diferentes pero tienen en común que la resolución del conflicto depende casi exclusivamente del mecanismo que tiene implementado el lenguaje y no de la decisión del diseñador/programador.

Cuando se modela utilizando herencia múltiple ya se puede detectar qué conflictos aparecen. Entonces £por qué dejar la resolución de conflictos hasta la implementación y más aún depender de lo que se pueda hacer en uno u otro lenguaje?. Deberían proponerse soluciones a estos conflictos antes de la implementación con las indicaciones del diseñador.

El esquema de la figura 3 muestra un proceso en el que se recibe una jerarquía de diseño utilizando herencia múltiple, donde habitualmente aparecen conflictos, y una especificación de cómo el diseñador quiere que se resuelvan dichos conflictos y se obtiene un primer resultado que consiste en un esquema de herencia múltiple donde el grafo de herencia puede representarse a través de una visión de las propiedades de las clases como conjuntos ya que, entre otras cosas, al resolver conflictos se garantiza la unicidad de las propiedades. Para el caso que nos ocupa de transformar jerarquías de herencia múltiple que puedan ser implementadas en lenguajes que no soporten herencia múltiple, este resultado sería la entrada del mecanismo de transformación. La notación y definiciones que se utilizan para formalizar este grafo de herencia son las que se describen a continuación.

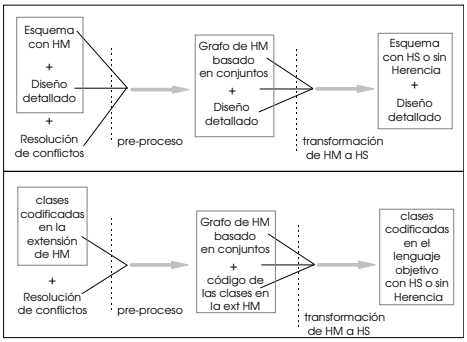

Figura3. El proceso de conversión.

2.2 Notación

La notación que se utilizará para los principales conceptos de la orientación a objetos está completamente expuesta y justificada en [Mar95]. Las clases se denotan por letras mayúsculas (X) y las propiedades por cadenas de caracteres con la primera letra en minúsculas (atr). Ocasionalmente no será necesario distinguir entre métodos y atributos por lo que ambos se unificarán en el concepto de propiedad (*feature*) [Mey97].

Para una clase C, el conjunto de propiedades definidas en esta clase se denotará por C_T. Con el fin de ver el conjunto de propiedades definidas en una clase en términos de compartición y transmisión de información (estrechamente

relacionado con los procesos de diseño con reutilización en DOO) estas se definen como propiedades heredadas o intrínsecas. Una propiedad se dice que es intrínseca de una clase -si está definida por la clase- o se dice que es heredada por la clase -si es una propiedad que la clase conoce a través de relaciones de herencia. De acuerdo con este criterio se definen el subconjunto de propiedades heredadas por la clase, denotado por C_H y el subconjunto de propiedades intrínsecas, denotado por C_I. El conjunto de propiedades de una clase define un conjunto porque han sido renombrados, unificados, etc. de acuerdo al mecanismo de resolución de conflictos de herencia múltiple que se haya planteado.

Para las propiedades de una clase se cumple que: $C_T = C_H \cup C_I$ y $C_H \cap C_I = \emptyset$. De la misma forma, los subconjuntos C_A y C_M se definen como el conjunto de todos los atributos en C, y el conjunto de todos los métodos en C, respectivamente. De acuerdo con esto, C_{AH} denotará el conjunto de todos los atributos heredados, C_{AI} el conjunto de todos los atributos intrínsecos y así C_{MH}, C_{MI}. Luego, se cumple también que $C_T = C_A \cup C_M$ y $C_A \cap C_M = \emptyset$.

Dadas dos clases X e Y que pertenecen a un conjunto finito no-vacío de clases $\mathcal{C}$, la relación de herencia definida sobre el conjunto $\mathcal{C}$ se denotará como $Y \prec X$ con X la clase padre y la clase Y su heredero. El grafo de herencia para $\mathcal{C}$ se define como el grafo $G = (\mathcal{C}, D)$, donde, $D = \{(Y, X) \mid Y \prec X\}$. G es un DAG.

Se dice, en base a esta representación del grafo de herencia, que una clase dada C es minimal si es una clase sin descendientes. Simétricamente, se dice que una clase es maximal si es una clase sin ancestros.

Las transformaciones que se describen en este trabajo se basan en una descomposición por niveles de la jerarquía, ascendente (N) o descendente (L), dependiendo del caso que sea.

La descomposición ascendente del grafo de diseño $G = (\mathcal{C}, D)$ se define como sigue:

$$N_0 = \{X \in \mathcal{C} \mid X \text{ es minimal en } G\}$$

$$N_p = \{X \in \mathcal{C} - \bigcup_{r=0}^{p-1} N_r \mid X \text{ es minimal en } (\mathcal{C} - \bigcup_{r=0}^{p-1} N_r, D/(\mathcal{C} - \bigcup_{r=0}^{p-1} N_r))\}$$

En las ecuaciones anteriores, $(\mathcal{C}', D/\mathcal{C}')$ se refiere al subgrafo inducido por el grafo $(\mathcal{C}, D)$ siendo $\mathcal{C}' \subset \mathcal{C}$ tal que $D/\mathcal{C}' = D \cap (\mathcal{C}' \times \mathcal{C}')$. $dim(G)$ denota el número de niveles en un grafo de herencia G. Por otra parte, de acuerdo a la definición de los conjuntos N_i, si $k = \max\{p \mid N_p \neq \emptyset\}$ entonces $\mathcal{C} = \bigcup_{r=0}^{k} N_r \Longleftrightarrow dim(G) = k$.

Si en la definición anterior, el criterio de minimal se cambia por maximal, se tiene la descomposición por niveles descendente (L). La descomposición por niveles conduce a la idea de generaciones.

Una vez que se ha realizado la descomposición por niveles del grafo, y como el número de clases en una biblioteca de clases es finito, se puede establecer una

función uno-a-uno ι desde el conjunto de clases $\mathcal{C}$ al conjunto $I = \{1, 2, \ldots, n\}$. La función ι actúa como un proceso de identificación numérica para el conjunto de clases $\mathcal{C}$. Debido a que se tiene que el grafo de herencia es un DAG y está dividido por niveles, si $n_r = \mathrm{C}ard(N_r)$ (lo mismo que $l_r = \mathrm{C}ard(L_r)$) entonces la asignación de identificadores puede realizarse de modo que:

$$\forall X, Y \in \mathcal{C}, X \neq Y \Rightarrow \iota(X) \neq \iota(Y)$$
$$\forall X \in N_p, \sum_{r=0}^{p-1} n_r < \iota(X) \leq \sum_{r=0}^{p} n_r$$

Establecido el proceso de identificación numérica, se puede hacer referencia a las clases utilizando la notación X_i. La notación X_i significa que la imagen de una clase $A \in \mathcal{C}$ a través de la aplicación ι se corresponde con el identificador numérico i.

Las notaciones y convenciones anteriores se utilizarán principalmente para presentar las transformaciones de las clases, atributos y relaciones de herencia de forma algebraica. Sin embargo, las representaciones gráficas de los ejemplos se harán utilizando notación UML [BRJ96]. De esta forma, algunos de los conceptos más usuales en la orientación a objetos, como por ejemplo la composición, se mostrarán explícitamente en los diagramas UML e implícitamente en las formulaciones algebraicas.

2.3 El ejemplo guía

Las transformaciones que se describen y sus problemas se ilustrarán a través del ejemplo clásico de la jerarquía de diamante, figura 4. El diagrama de la parte izquierda de la figura es un esquema simplificado de lo que podría ser un ejemplo real que se esboza paralelamente en la parte derecha de la figura.

Hay cuatro clases en el ejemplo. Cada clase introduce un nuevo atributo y un nuevo método. Por ejemplo, la clase B introduce el atributo *atrB* y el método *metB*. En la figura, por simplicidad, no se especifica completamente la signatura del método. Se adoptarán las siguientes convenciones para este ejemplo:

Todos los métodos tienen igual signatura (excepto el nombre). Cuando un nombre de método aparece en una clase y el mismo nombre está en una clase ancestro, esto significa que el método se redefine. Cuando se redefine un método, se asume que en la nueva definición se utiliza el método correspondiente a su ancestro (puesto que es el caso más interesante). Por ejemplo, el método *metA* se introduce en la clase A y se redefine en el resto de las clases y el método *metA* de la clase D utiliza los métodos correspondientes a *metA* que definieron sus ancestros B and C.

Ahora se pasará a dar una descripción detallada de cada estrategia y los problemas que presenta.

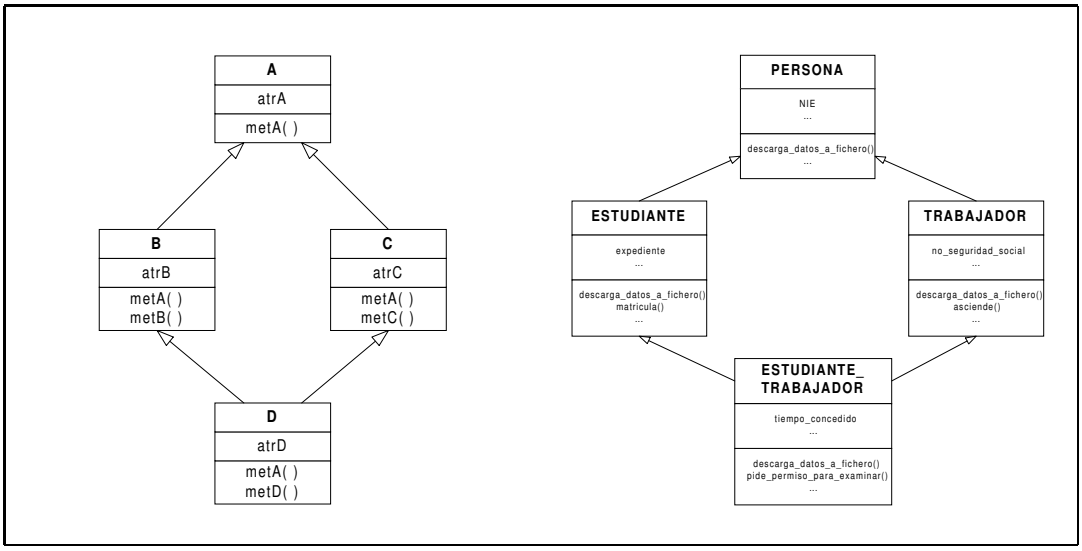

Figura4. Una jerarquía de herencia múltiple.

2.4 Emancipación

La aplicación de este proceso a una jerarquía de herencia múltiple se lleva a cabo utilizando la descomposición por niveles descendente del grafo de herencia y dos operadores $E()$ y ∇.

El primer operador a ser definido es el operador $Y.E()$ (Y se Emancipa). El operador $Y.E()$ se define como: $Y' = Y.E()$ donde

$$
\begin{aligned}
Y'_{AI} &= Y_{AI} \cup Y_{AH} \\
Y'_{AH} &= \emptyset \\
Y'_{MI} &= \nabla\,(Y_{MI} \cup Y_{MH}) \\
Y'_{MH} &= \emptyset
\end{aligned}
$$

La primera ecuación significa que en la nueva clase se conservan todos los atributos intrínsecos y heredados de que disponía la clase original. Esta junto a la segunda ecuación indica que, en cuanto a atributos, la clase se prepara para liberarse de sus ancestros.

La tercera ecuación indica que en el proceso de emancipación, los métodos de los ancestros se traen a la nueva clase. De esta forma, es posible que aparezcan diferentes "versiones" del mismo método en una clase, debido a que el método haya sido redefinido. Entonces, se hace necesario renombrarlos (para que puedan coexistir) y modificar el cuerpo de aquellos métodos que lo utilizan, como es el caso de las llamadas a un método *super*. Esto está dado porque todas las llamadas

super ahora serán llamadas locales al método renombrado e introducido en la clase resultante. ∇ es un operador que abstrae este tipo de operaciones que se describen en [Opd92] como re-factorizaciones primitivas o atómicas[3]. La última ecuación complementa este planteamiento dejando claro que en la nueva clase no aparecen ya las dependencias de los ancestros pues $Y'_H = Y'_{AH} \cup Y'_{MH} = \emptyset$.

Dado un grafo G, ($dim(G) = k$), con relaciones de herencia múltiple y su descomposición descendente L, el proceso de emancipación se define según el siguiente algoritmo:

Algoritmo 1

Desde $p \leftarrow 0$ *hasta* k *hacer*

Para cada $Y \in L_p$ *hacer*

- $Y' = Y.E()$

El grafo resultante sería $G_E = (\mathcal{C}_E, D_E)$ definido como:

$$D_E = \emptyset$$
$$\mathcal{C}_E = \{Y' \mid \exists Y \in \mathcal{C} \text{ con } Y' = Y.E()\}$$

La figura 5 muestra el resultado de aplicar el proceso de emancipación al diseño que aparece en la figura 4, con una resolución de conflictos dada (en este caso, herencia no repetida). Se obtienen cuatro clases aisladas. Para cada clase en el diagrama original se tiene su correspondiente clase emancipada. Es necesario resaltar que en la clase resultante B' aparecen dos "versiones" de *metA* (una renombrada) porque se conoce que el método *metA* de la clase B utiliza el método *metA* de la clase A. Podrían necesitarse incluso todas las "versiones" del método en sus ancestros, como es el caso de la clase D.

Si se hubiera especificado herencia repetida para el atributo *atrA* (heredado en D a través de diferentes caminos), estos hubieran tenido que ser renombrados e incluidos (*atrAofB* y *atrAofC*), en el momento de la resolución de conflictos.

Haciendo un análisis de esta estrategia siguiendo los objetivos declarados en la sección 1, salta a la vista la excesiva duplicación de código a la que conduce esta transformación. No se conserva de ninguna forma la clasificación jerárquica inicial pues se rompen los enlaces de herencia. De aquí mismo surge el principal problema de esta aproximación que viene del polimorfismo (más exactamente, polimorfismo de inclusión [CW85]). A menudo un objeto de una clase heredera se quiere utilizar en el lugar donde se espera un objeto de la clase padre. Las definiciones dadas en [CW85] se basan en un formalismo de tipo-subtipo basado en conjuntos. Según esto, una instancia de la clase D' podría reemplazar a una instancia de la clase C' porque D' tiene las mismas propiedades que C' y más. Pero esto no soluciona el problema. Uno de los objetivos principales que se plantearon fue obtener una estrategia implementable en cualquier lenguaje OO con las menores pérdidas posibles. El problema es que la mayoría de los lenguajes OO permiten el polimorfismo de inclusión solamente basado en la existencia de

[3] primitive re-factorings

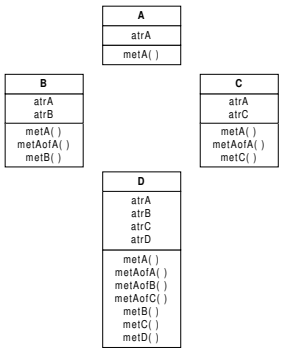

Figura5. La jerarquía después de transformarse según la emancipación.

una relación de herencia. A favor de esta estrategia hay que decir que precisamente por sus características de adapta a lenguajes en los que no existe ninguna presencia de construcciones de herencia.

Para resolver el problema de las asignaciones se pueden definir mecanismos de conversión de una clase a todas las nuevas clases correspondientes a sus ancestros. Pero esto conduce a problemas de eficiencia (copiar todos los atributos) y de coherencia (los atributos de los objetos deben mantenerse iguales a lo largo de los cambios realizados a través de una u otra entidad).

2.5 Composición

La base de esta estrategia es transformar los enlaces de herencia en enlaces de composición. Todo lo que se resolvía con herencia será resuelto con delegación [Ste87].

Podría pensarse en la posibilidad de no aplicar el proceso para todos los enlaces de herencia que tiene una clase. Es decir, si una clase tiene n relaciones de herencia, entonces transformar sólo $n-1$. Debido a que esto conduce a un tratamiento no uniforme se asumirá que se transforman todos. Esto mismo se asume para el caso anterior y los siguientes.

Para la transformación por composición a una jerarquía de herencia múltiple se utiliza la descomposición por niveles descendente del grafo, y los operadores $C()$ y ξ.

El operador $Y.C()$ se define como: $Y' = Y.C()$ donde

$$Y'_{AI} = Y_{AI} \cup \left(\bigcup_i \{attX_i\} \right) \ \forall X_i \text{ con } Y \prec X_i$$
$$Y'_{AH} = \emptyset$$
$$Y'_{MI} = \xi\,(Y_{MI} \cup Y_{MH})$$
$$Y'_{MH} = \emptyset$$

La primera de las ecuaciones anteriores significa que en la nueva clase se conservan todos los atributos intrínsecos y se añade un atributo nuevo por cada clase padre. En la definición anterior, cuando se incluye $attX_i$ se está indicando que se añade una relación de composición con la clase X_i y esto se representa a través de un atributo ($attX_i$) de tipo X_i.

La tercera ecuación indica que en este proceso la nueva clase tendrá intrínsecamente todos los métodos (heredados e intrínsecos) de que disponía. El operador ξ, análogamente al operador ∇ definido en la sección anterior, abstrae algunas operaciones que tienen que ver con cambiar el cuerpo de algunos métodos de la clase que está siendo transformada, al igual que ∇ consiste en un conjunto de re-factorizaciones primitivas. En este sentido, debido a que las relaciones de herencia se reemplazan por relaciones de composición, ξ realiza tres modificaciones: las llamadas a los *super* métodos deben modificarse delegando en el objeto componente, el acceso a los atributos que antes eran heredados ahora se hace a través de los objetos compuestos, los métodos heredados pero no redefinidos, en la jerarquía original, se incluyen en la nueva clase cambiando su cuerpo. El nuevo cuerpo tendrá una única acción: delegar en el objeto componente.

La segunda ecuación y la última indican que la nueva clase rompe la dependencia con los padres que le correspondían en la jerarquía original.

Dado un grafo G, ($dim(G) = k$), con relaciones de herencia múltiple y su descomposición descendente L, el proceso de composición se define según el siguiente algoritmo:

Algoritmo 2

Desde $p \leftarrow 0$ *hasta* k *hacer*

Para cada $Y \in L_p$ *hacer*

- $Y' = Y.C()$

El grafo que se obtiene sería $G_C = (\mathcal{C}_C, D_C)$ definido como:

$$D_C = \emptyset$$
$$\mathcal{C}_C = \{Y' \mid \exists Y \in \mathcal{C} \text{ con } Y' = Y.C()\}$$

En la nueva jerarquía resultante, cada clase tendrá todos los métodos intrínsecos y heredados de la clase correspondiente en la jerarquía original. La definición de los métodos que en la clase original hayan sido heredados pero no redefinidos, y los métodos redefinidos que utilizan sus correspondientes métodos provenientes de los ancestros, se resuelve delegando en el método de la clase componente.

La figura 6 muestra el resultado de aplicar el proceso de composición al diseño que aparece en la figura 4. Cuando se quiera acceder a un atributo heredado, ahora habría que especificar el camino completo hasta el atributo (a través del objeto compuesto). Para evitar esto podría definirse en la nueva clase métodos de acceso para cada atributo heredado que realizaran este paso.

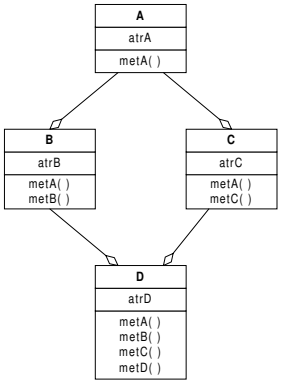

Figura6. La jerarquía después de transformarse según la composición.

En esta estrategia, no aparecen problemas de duplicación de código pero, al igual que el caso anterior, rompe con todos los enlaces de herencia eliminando completamente la clasificación jerárquica. Esto mismo es lo que la hace buena para lenguajes sin herencia. Por otra parte, aún se presentan los problemas con el polimorfismo, al igual que en la transformación por emancipación. Pero aquí se tiene una alternativa para la asignación polimórfica. Si es necesario que un objeto de una clase actúe como un objeto de su clase ancestro (en la jerarquía original), puede utilizarse en su lugar el correspondiente objeto compuesto (esto es, delegar en uno de los objetos compuestos). Esto resuelve la asignación polimórfica pero lo más importante: el comportamiento polimórfico, queda como problema fundamental. Por ejemplo, en la nueva clase D' el método *metB* se redefine. Si se utiliza en una asignación polimórfica el objeto compuesto B' de D', cuando se llame al método *metB* a través de él, se invocará la "versión" de *metB* correspondiente a la clase B original y no la correspondiente a la clase D original, lo cual sería el efecto deseado. Se pierde el comportamiento polimórfico debido a la ausencia de la referencia a sí mismo (*self-reference* [CP89]).

2.6 Expansión: transformando el grafo de herencia en un árbol

Las jerarquías de herencia basadas en herencia múltiple se transforman a jerarquías con sólo herencia sencilla a través de un mecanismo de expansión de grafos sin pérdida de información. El proceso aquí descrito no distingue entre métodos y atributos, se interesa solamente por propiedades en general.

El principal objetivo de la expansión es obtener un nuevo grafo donde: cada componente conectada, i.e., componente sin elementos aislados, sea un árbol. El nuevo grafo será generado obteniendo por cada clase original con relaciones de herencia múltiple, tantas clases nuevas con relaciones de herencia sencilla como relaciones de herencia tenía la original. Este proceso se lleva a cabo sobre la base de una descomposición por niveles ascendente del grafo de herencia (sección 2.2).

Una vez que se tiene hecha la descomposición ascendente (N_i) del grafo y la función de identificación numérica aplicada, el proceso de conversión se describe utilizando un operador $\mathcal{J}$. La notación $Y' = \mathcal{J}(Y \mid X)$ significa que la clase Y' se obtiene de la clase Y y la relación de herencia que existe entre las clases X y Y ($Y \prec X$). La clase Y' se define por las siguientes ecuaciones:

$$Y'_I = Y_I \cup (Y_H - X)$$
$$Y'_H = Y_H \cap X$$

La primera ecuación declara que el subconjunto de propiedades intrínsecas en la nueva clase se construye con las propiedades intrínsecas de la clase Y y las propiedades que pertenecen a la parte heredada que no se obtiene de X, i.e., las propiedades que se heredan a través de los otros padres de Y.

La segunda ecuación declara que las propiedades de Y que se heredan a través de la relación de herencia $Y \prec X$ se colocan en el subconjunto de propiedades heredadas de Y'.

En la transformación que describen las ecuaciones anteriores, cada clase en la jerarquía original se expande en tantas clases como padres tiene. La expansión se realiza utilizando el operador $\mathcal{J}$, y el principal objetivo del proceso es: no perder información. Las nuevas clases definidas mantendrán toda la información de la clase original correspondiente, sólo cambia la forma en que la información aparece en la clase: de heredada a intrínseca. Esto es resultado de replicar las propiedades.

Definición 1 *Sea $G = (\mathcal{C}, D)$, un grafo de herencia. El conjunto réplica de una clase $X_i \in \mathcal{C}$ se denota como $\ddot{X}_i$ y es el conjunto de clases que se define como*

$$\ddot{X}_i = \{X_{ij} = \mathcal{J}(X_i \mid X_j) \mid (X_i, X_j) \in D\}$$

Según la definición de $\mathcal{J}$ todas las clases X_{ij} tendrán las mismas propiedades que X_i.

La conversión del grafo será un proceso paso-a-paso donde la conversión se lleva a cabo por niveles, comenzando por el nivel N_0.

En lo que queda de esta sección se supondrá que el grafo de herencia G tiene $Card(\mathcal{C}) = n$, $dim(G) = k$, y está etiquetado por el conjunto de identificadores numéricos $I = \{1, 2, ...n\}$ utilizando la función ι (sección 2.2).

Definición 2 *Sea G un grafo de herencia. La conversión del grafo G por las clases del nivel N_0 producirá como resultado el grafo $G_1 = (\mathcal{C}_1, D_1)$ definido como:*

$$\begin{aligned} \mathcal{C}_1 &= (\mathcal{C} - N_0) \cup \\ &\quad \{X_{ij} \mid X_i \in N_0 \wedge X_{ij} \in \ddot{X}_i\} \cup M_0 \\ D_1 &= (D - \{(X_i, X_j) \in G \mid X_{ij} \in \mathcal{C}_1\}) \cup \\ &\quad \{(X_{ij}, X_j) \mid X_{ij} \in \mathcal{C}_1\} \end{aligned}$$

En las ecuaciones anteriores, M_0 representa el conjunto de clases aisladas i.e., clases sin relación de herencia.

En este proceso, las clases en el nivel que se está convirtiendo se reemplazan dependiendo de su multiplicidad local (número de padres) pero manteniendo la misma información. Cada clase con multiplicidad local mayor que 1 se sustituirá por un conjunto de clases con multiplicidad 1. Mientras aquellas clases con multiplicidad local igual o menor que 1 se quedan como están. La conversión conduce a hacer desaparecer todas las relaciones de herencia múltiple dejando solamente relaciones de herencia sencilla.

El proceso definido se aplicará sucesivamente para el resto de los niveles, en orden ascendente. Así, la conversión para un nivel dado solamente puede obtenerse si todas las clases que pertenecen a los niveles inferiores han sido expandidas. Si todas las clases bajo el nivel N_p están expandidas, i.e. sólo existen bajo ese nivel relaciones de herencia sencilla, se dice que el grafo G es un grafo expandido de nivel N_p.

Por otra parte, el proceso de expansión de nivel p, $p \neq 0$, no puede ejecutarse meramente reemplazando cada clase con un conjunto de nuevas clases como se hizo en el nivel 0. Cuando se realiza el proceso de conversión del nivel p es necesario determinar, no solamente cómo replicar las clases de dicho nivel, sino las repercusiones en las clases que pertenecen a niveles inferiores. Las transformaciones para las clases en los niveles p, $0 < p \leq k$, están condicionadas por el subgrafo al que pertenecen.

Definición 3 *Sea X_i una clase del nivel N_p, $X_i \in N_p$. El grafo $G^i = (\bar{X}_i, D/\bar{X}_i)$ es el grafo derivado para la clase X_i, siendo $\bar{X}$ el conjunto de clases descendientes de X; $\bar{X} = \{Y \mid$ existe un camino en el grafo de herencia de Y a $X\ \} \cup \{X\}$*

Como resultado de la definición anterior, si G es un grafo expandido de nivel N_p, X_i es una clase que pertenece al nivel N_p y G^i su grafo derivado, entonces el grafo G^i es un árbol que tiene a la clase X_i como clase máxima o raíz [Mar95].

El proceso de conversión expuesto para un nivel dado N_p define en primer lugar los conjuntos réplica $\ddot{X}_i$ para cada clase $X_i \in N_p$. Luego, para cada nueva clase X_{ij} se define un árbol asociado G^{ij}, con X_{ij} como raíz. El grafo G^{ij} se construye con arreglo a G^i, definido por la clase X_i. Finalmente, el conjunto de árboles definidos se conectarán al resto del grafo.

Definición 4 *Sea $G = (\mathcal{C}, D)$ un grafo de herencia expandido hasta el nivel N_{p-1}, con $0 < p < k$. Dada una clase $X_i \in N_p$, su conjunto réplica $\ddot{X}_i = \{X_{ij} = \mathcal{J}(X_i \mid X_j) \mid (X_i, X_j) \in D\}$, y su grafo derivado $G^i = (\bar{X}_i, D/\bar{X}_i)$. El grafo $G^{ij} = (\mathcal{C}^{ij}, D^{ij})$ se dice grafo derivado de la clase $X_{ij} \in \ddot{X}_i$ con:*

$$\mathcal{C}^{ij} = \{X_{lj} \mid X_l \in \bar{X}_i\}$$
$$D^{ij} = \{(X_{uj}, X_{vj}) \mid (X_u, X_v) \in D/\bar{X}_i\}$$

y las clases X_{lj} se definen como:

$$X_{ij} = \mathcal{J}(X_i \mid X_j)$$
$$X_{lj} = X_l \quad \forall X_l \in \bar{X}_i \text{ y } l \neq i$$

La definición del proceso de conversión para un nivel dado N_p se basa en los conjuntos réplica $\ddot{X}_i$ definidos y en los grafos asociados G^{ij}.

Definición 5 *Sea $G = (\mathcal{C}, D)$ un grafo de herencia expandido hasta el nivel N_{p-1}, con $0 < p < k$. El grafo de conversión G para las clases en el nivel N_p se denota como $G_{p+1} = (\mathcal{C}_{p+1}, D_{p+1})$ y se define según:*

$$\begin{aligned}
\mathcal{C}_{p+1} &= ((\mathcal{C} - N_p) - \{X_l \mid X_l \in \bar{X}_i \wedge X_i \in N_p\}) \cup \\
&\quad \{X_{lj} \mid X_l \in \bar{X}_i,\ X_i \in N_p \wedge X_{ij} \in \ddot{X}_i\} \\
D_{p+1} &= [(D - \{(X_i, X_j) \mid X_i \in N_p\}) - \\
&\quad \{(X_u, X_v) \in D \mid X_{uj}, X_{vj} \in \mathcal{C}_{p+1}\}] \cup \\
&\quad \{(X_{rj}, X_{sj}) \mid (X_{rj}, X_{sj}) \in G^{ij} \wedge X_i \in N_p\}) \cup \\
&\quad \{(X_{ij}, X_j) \mid X_i \in N_p \wedge X_{ij} \in \mathcal{C}_{p+1}\}
\end{aligned}$$

Resumiendo, dado un grafo de herencia $G = (\mathcal{C}, D)$, su descomposición por niveles ascendente ($N_r \quad r = \{0, 1, 2, \ldots, k\}$) y la asignación de identificadores numéricos para todas las clases según la función ι, el proceso completo de conversión para este grafo G se describe con el siguiente algoritmo.

Algoritmo 3

1. *Definir el conjunto réplica $\ddot{X}_i$ para todas las clases $X_i \in N_0$ utilizando el operador $\mathcal{J}$.*
2. *Convertir el grafo G para obtener el grafo G_1.*
3. *Desde $p \leftarrow 1$ hasta $k - 1$ hacer*
   (a) *Definir el conjunto réplica $\ddot{X}_i$ para todas las clases $X_i \in N_p$ utilizando el operador $\mathcal{J}$.*
   (b) *Construir los grafos derivados $G^{ij} = (\mathcal{C}^{ij}, D^{ij})$ para todas las clases $X_{ij} \in \ddot{X}_i$ obtenidas.*
   (c) *Convertir el grafo G_p para obtener el grafo $G_{p+1} = (\mathcal{C}_{p+1}, D_{p+1})$.*

De la aplicación de este algoritmo se obtiene la conversión de una jerarquía de herencia múltiple en una jerarquía de herencia sencilla que mantiene toda la información que tenía la anterior. Una descripción detallada y con ejemplos de este mecanismo se puede encontrar en [Mar95, Capítulo 4].

Nuevamente, el problema que se encuentra al aplicar esta transformación radica en el polimorfismo. Las clases que heredaban múltiple se han expandido en varias clases (en el ejemplo, de D se obtuvieron *D1* y *D2*, ver figura 7). Con esta transformación no habría ningún problema si, según el diseño, se requiere que un objeto que pertenece a una de las clases –que en la jerarquía original tenía relaciones de herencia múltiple– actúe en el lugar de uno de sus ancestros (por ejemplo un objeto de D actuando en el lugar de uno de B). Siempre se podría escoger declarar el objeto como un objeto perteneciente a la clase apropiada, es decir, que en la nueva jerarquía hereda de la clase de dicho ancestro (*D1* hereda de B en la nueva jerarquía). El problema real aparecería si se deseara que un mismo objeto actúe indistintamente en el lugar de objetos de más de

uno de sus ancestros que no se encuentran en el mismo camino de herencia en la jerarquía original (en el ejemplo, unas veces un objeto de D actúa en lugar de uno de C y de uno de B). Para poder obtener este efecto, se necesitan dos objetos diferentes (*d1*: *D1* y *d2*: *D2*) tratados como uno. Esto conlleva a serios problemas de mantener la coherencia entre uno y otro. Por otra parte, otro aspecto negativo, en relación con el cumplimiento de los objetivos propuestos, que se aprecia inmediatamente con esta transformación es la duplicación de código que es consecuencia precisamente de la réplica de clases.

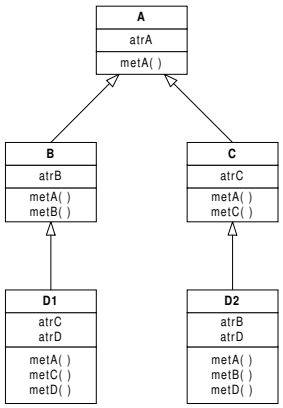

Figura7. La jerarquía después de transformarse según la expansión.

2.7 Tipos *variant* o simulación mediante banderas

En esta estrategia se crea una clase monitora en la idea de los tipos *variant* o *union*. Los tipos que se conocen como tipos *record* son productos cartesianos de elementos de otros tipos, al contrario de los tipos *variant* o *union* que son una suma disjunta de elementos de otros tipos. *records* y *variants* son las construcciones básicas de tipos no-funcionales en la semántica denotacional. Al igual que en los *records*, las partes de la suma disjunta en los *variants* son elementos etiquetados. Las operaciones básicas sobre *variants* son `is` y `as`. La primera, pregunta si el objeto *variant* está determinado por una etiqueta dada (o sea por la parte en la suma disjunta que representa dicha etiqueta). La segunda, extrae el contenido de un objeto *variant* asumiendo que es de la forma que tiene el elemento representado por la etiqueta dada. Ambas operaciones se complementan. Si `obj is etiq` es verdadero, `obj as etiq` es una operación válida a través de la cual se obtiene al objeto en su forma correcta [Car84].

En algunos lenguajes existen construcciones que asemejan la definición de tipo *variant* o *union* como es el caso de Pascal con el *record case* y C/C++ con el *union*. Pero aún cuando no existen construcciones específicas para representar

un tipo *variant* se puede lograr simular la idea básicamente a través de un tipo *record* cuyas componentes están ocultas y que se muestran a través de operaciones que se definen simulando is y as. Esto conduce a la idea de una clase que contiene componentes ocultas, ciertas banderas y que ofrece como interfaz unos métodos que se encargan de simular las operaciones is y as mencionadas.

La estrategia que aquí se define trata de representar una clase y sus subclases en un único tipo *variant*. Llevando esto a la idea de la simulación mediante clases y banderas se tiene que de lo que se trata es de representar en una nueva clase lo que está contenido en una clase y sus subclases en la jerarquía original.

Para llevar a cabo esta transformación se utiliza la descomposición descendente del grafo de herencia y dos operadores $\triangle$ y ρ. El algoritmo que se describe supone que no hay ocultación ni cancelación de propiedades a través de la herencia.

Sea L la descomposición descendente del grafo G. Para cada subgrafo S que sea una componente desconectada en el grafo G obtener la clase Y' de forma que:

Algoritmo 4

Inicializar Y' *según:*

$Y'_H = \emptyset$

- $Y'_{AI} = \{whoAmI\}$ *(1)*

Para cada $Y \in A$ *tal que* $Y \in L_0$ *, inicializar* Y' *según:*

- $Y'_{AI} = Y'_{AI} \cup Y_A$
- $Y'_{MI} = \triangle(Y_M) \cup \left(\bigcup_{X \in Y_A} \{SetX, GetX\}\right)$ *(2)*

Desde $p \leftarrow 1$ *hasta* k *hacer*

Para cada $Y \in L_p$ *hacer*

- $Y'_{AI} = Y'_{AI} \cup Y_{AI}$
- $Y'_{MI} = Y'_{MI} \cup \triangle(Y_{MI}) \cup \left(\bigcup_{X \in Y_{AI}} \{SetX, GetX\}\right)$ *(3)*

Para cada $Y \in L_k$ *hacer*

- $Y'_{MI} = Y'_{MI} \cup \rho(Y_M)$ *(4)*

La clase Y' que se obtiene para cada árbol desconectado en G no tiene relaciones de herencia ($Y'_H = \emptyset$). A la clase Y' se le denomina clase monitora de las clases de dicho árbol. En (1) se introduce en Y' un atributo $whoAmI$ que representa el identificador del tipo del objeto en curso (la etiqueta del *variant*). A dicho identificador se le dará valor en el momento de la construcción del objeto. Los posibles valores de $whoAmI$ se corresponden con identificadores de las clases de la jerarquía original.

El operador $\triangle$ (ver (2) y (3)) renombra los métodos de una forma que podría ser nombreOriginal_of_nombreClase por ejemplo $metAofA$ y los oculta. Los métodos $SetX$, $GetX$ que se introducen en (3) preguntan por $whoAmI$, si $whoAmI$ no se corresponde con el identificador para la clase Y o sus descendientes se obtiene una excepción. No ocurre así para los métodos de $SetX$, $GetX$ que se introducen en (2) ya que estos atributos pertenecen a la clase raíz y por tanto se heredan en todas las subclases (recordar la suposición inicial de no ocultación-cancelación). En (4) se añaden a Y' los métodos que serán la interfaz de la clase

ya que los demás están ocultos, estos métodos serán los encargados de despachar a los métodos adecuados o impedir la ejecución del método.

El operador ρ modifica los cuerpos de los métodos. El objetivo es que en dichos cuerpos se haga una instrucción case sobre $whoAmI$ y se invoque al método que se ocultó y se renombró en (2) o en (3) correspondiente a la clase en la jerarquía original que $whoAmI$ identifique o se lance una excepción si el método no se corresponde con dicha clase. Los operadores $\triangle$ y ρ se emplean para encapsular conjuntos de re-factorizaciones primitivas al igual que ∇ y ξ en las secciones anteriores.

En la figura 8 se ve un esquema de la clase resultado de transformar mediante esta estrategia el ejemplo guía. Los métodos que se listan a partir de $SetA$ constituyen la interfaz de la clase. El resto de métodos y atributos forman la representación interna de la clase monitora.

Figura8. La jerarquía después de transformarse a una clase monitora.

En el código, donde quiera que se necesita declarar una entidad de una clase se declara como la clase monitora correspondiente al subgrafo del grafo de herencia al que pertenece la clase. En las asignaciones entre entidades hay que tener cuidado y preguntar si el objeto se puede asignar o no (dependiendo de $whoAmI$). La otra opción es no preguntar y dejar que se lance una excepción (en el mejor de los casos) cuando suceda alguna violación.

En esta estrategia no se tienen problemas de mantenimiento de coherencia ni de duplicación de código. Tiene una ventaja importante: que es aplicable para todas las "categorías" de lenguajes con respecto a la herencia (figura 2). Pero en cambio se destacan unas cuantas características negativas. No respeta en ninguna medida la jerarquía del modelo inicial. Por otra parte, cuando el lenguaje es fuertemente tipado se pierden las ventajas del chequeo de tipos en tiempo de compilación para las asociaciones polimórficas y lo que es peor para la

invocación de métodos o el acceso a atributos. Intentar asociar un objeto de un tipo que no es permitido en un momento dado se traduce en una excepción (en el mejor de los casos o en un comportamiento imprevisto en el peor) en lugar de poderse detectar mediante el chequeo de conformancia en la compilación. Esto hace el código menos seguro y por tanto menos fiable. Por otra parte, aunque el comportamiento polimórfico se garantiza, esto se hace a través de instrucciones de tipo case que se encargan de despachar los métodos correctos y de rechazar las llamadas no permitidas. El resultado es una clase compleja que contiene todos los métodos y atributos de la clase raíz de una jerarquía y sus descendientes y a la que hay que manejar con excesivo cuidado.

3 Combinación de estrategias básicas para el proceso automatizable de transformación de jerarquías

Un análisis de las estrategias básicas nos conduce a la conclusión de que ninguna de ellas es completamente satisfactoria por sí sola debido a que la transformación implica la pérdida de demasiada información de diseño y se incumplen más de una o incluso todas las características ideales mencionadas en la sección 1 para una transformación como la que nos ocupa. Es por esto que aquí se introducen además otras dos propuestas para el mecanismo de transformación de jerarquías que son producto una combinación de estrategias básicas con el fin de acercarse lo más posible a los objetivos planteados.

Las estrategias combinadas, al igual que las básicas, se describen partiendo de una aproximación general. Es decir, se trata de describir los mecanismos con independencia del lenguaje destino (con excepción del caso de Java). En todo caso, una vez claros los aspectos generales siempre se pueden llevar a cabo "optimizaciones" de acuerdo a características específicas de un lenguaje en particular.

Las estrategias combinadas, que son dos, se clasifican en: caso Java y caso General. La distinción especial para lenguajes Java se justifica porque, aunque no deja de ser un caso particular, es muy interesante y representativo porque sirve de base para pensar en la estrategia más general.

Lo que hace especial el caso Java es que se pueden obtener dos jerarquías separadas, una de clases con herencia sencilla y otra de interfaces con herencia múltiple y se pueden aprovechar otras características del lenguaje. Esta particularización sirve de ejemplo de cómo se pueden adaptar los métodos generales para aprovechar características particulares y, por otra parte, sirve como apoyo para pasar al caso general en el que básicamente se sustituye la presencia de interfaces por unas clases (que son sólo caparazones) organizadas en una jerarquía de herencia simple producto de aplicar la idea de la estrategia definida como expansión (sección 2.6).

3.1 Estrategia combinada para Java

Java incluye el concepto de interfaces. Las clases implementan interfaces, y pueden implementar más de una interface. En Java, existe herencia sencilla de clases

y múltiple de interfaces. En las interfaces pueden describirse solamente constantes y declaraciones de métodos. Un objeto de una clase que implementa una interfaz puede actuar bajo la declaración de una variable de dicha interfaz. Una variable declarada de una interfaz puede sustituir a cualquier variable de las interfaces que son ancestros de ella, así mismo un objeto de una clase puede actuar en el lugar de un objeto de una de las clases que es ancestro de ella [GJS96].

Dada una jerarquía de diseño, esta se puede representar utilizando interfaces Java, donde se mantienen todas las relaciones de herencia. En estas interfaces no se pueden incluir ni atributos no-constantes ni los cuerpos de los métodos. Por tanto, se necesita que algunas clases implementen dichas interfaces. Para poder modelar la existencia de atributos a partir de las interfaces se deben definir las signaturas de métodos de acceso a estos (para recuperación y transformación).

Debido a la aplicación del proceso de emancipación a la jerarquía original, se obtienen nuevas clases equivalentes a las originales. Para Java puede especificarse que cada clase resultante implementa a su correspondiente interfaz, i.e. la interfaz que se obtuvo a partir de la misma clase que se transformó por emancipación.

En la figura 9 se muestra la aplicación de esta solución al ejemplo. Por simplificar se muestran solamente los atributos y métodos de las interfaces y de la clase *AClass*. Los atributos y métodos de las otras clases están dados por el resultado del proceso de emancipación. A dicho resultado se le añade un grupo de métodos para obtener y transformar los atributos.

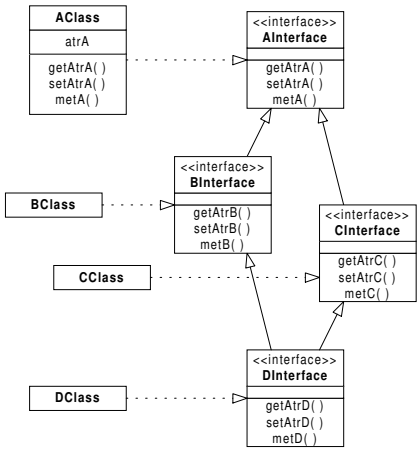

Figura9. Una vista de la primera variante en la estrategia combinada para Java.

Cada variable (atributo, parámetro o variable local) que se declaraba como una entidad de la clase original se declara como la interfaz correspondiente, excepto en la construcción del objeto. En este caso se construye un objeto de la clase resultado de la emancipación.

El hecho de manipular todos los objetos a través de variables declaradas como sus correspondientes interfaces permite la asignación polimórfica y el correspondiente comportamiento polimórfico se obtiene como resultado de la creación de los objetos a partir de las clases emancipadas. Hasta aquí se logran cumplir dos de los objetivos propuestos: se mantiene la clasificación jerárquica original a través de la jerarquía de interfaces. Se respetan las asignaciones y el comportamiento polimórficos gracias al manejo de los objetos a través de entidades declaradas como interfaces y a la construcción de los objetos mediante clases emancipadas que implementan dichas interfaces.

Pero el proceso de emancipación genera duplicación de métodos en el conjunto de clases resultantes (el código correspondiente al *metA* de la jerarquía original está en la clase *AClass*, y en la clase *BClass* bajo el nombre de *metAofA*, etc.). Para evitar esta redundancia podría pasarse a una segunda variante que resulta más compleja. Se introduce un nuevo grupo de clases, resultado de aplicar la composición a la jerarquía original. El proceso completo incluye filtrar las clases resultantes de la emancipación y de la composición de la siguiente forma: en las clases que se obtienen por composición dejar solamente los métodos; de acuerdo con esto, en las clases que se obtienen de la emancipación dejar sólo los atributos; y mantener idéntica la obtención de las interfaces a partir de la jerarquía original.

Finalmente, dada una jerarquía de herencia múltiple, de la transformación se obtendrán tres grupos de clases (en realidad, dos de clases y uno de interfaces) relacionadas:

- Primero se obtiene una nueva jerarquía de interfaces, isomorfa con la original. Para cada clase original habrá una nueva interfaz. Para cada enlace de herencia en la clase original, se tendrán las correspondientes interfaces enlazadas (en Java se dice que una interfaz extiende a la otra: `interface B extends A`). Cada interfaz tendrá la signatura de los métodos de su correspondiente clase en la jerarquía original. Para cada atributo en la clase original, se definen dos métodos, uno para obtener y el otro para modificar el atributo. Esto es así porque en las interfaces sólo aparecen signaturas de métodos y atributos constantes. De esta forma para poder manipular los objetos siempre a través de una variable declarada como la interfaz, se hace necesario disponer en esta última de dichos métodos.
- En segundo lugar se da ubicación a los atributos de las clases originales en otro grupo de clases. Dichas clases se obtienen aplicando un proceso de emancipación a las clases originales y eliminando sus métodos. A estas clases se les nombrará como clases de atributos. Para utilizar a tercera ecuación y la última que definen el operador $Y.E()$ en 2.4 se funden para este caso en $Y'_M = \emptyset$.
- Luego se obtiene otro grupo de clases utilizando la transformación por composición. Los atributos se sustituyen por el correspondiente par de métodos para obtenerlos y modificarlos. Se mantienen los cuerpos de los métodos. Cada clase de este grupo tendrá una clase componente que no es más que su correspondiente clase de atributos. A las clases que se obtie-

nen en este grupo se les nombrará como clases de métodos. Para utilizar la primera ecuación que definen el operador $Y.C()$ en 2.5 se cambia por $Y'_{AI} = Y_{AI} \cup \{attY Attributes\}$ donde $attY Attributes$ es un atributo de tipo $Y Attributes$ que es la clase de atributos resultado de aplicar la emancipación a la clase de partida Y como se expone en el punto anterior. Para utilizar la tercera ecuación se añade la unión con un par de métodos de acceso y transformación para cada atributo $a \in Y_{AH}$ (los atributos heredados en la clase original).

Entrelazando los 3 grupos se tiene que: cada clase de métodos implementa su correspondiente interfaz en la jerarquía de interfaces (en Java se dice que class AMethods implements AInterface) y cada clase de atributos es un componente de la correspondiente clase de métodos (en Java se tendrá que la clase de métodos tiene un atributo que es un objeto de la clase de atributos). Cuando se crea un objeto de una clase de métodos, los objetos de atributos que pertenecen a las clases de métodos agregadas a ella se mantienen nulos, es decir, sólo se crea el objeto de atributos asociado directamente con ella. Las interfaces constituirán caparazones para despachar las llamadas al objeto.

La figura 10 muestra una vista de la jerarquía que se obtiene al transformar la jerarquía del ejemplo presentado en la figura 4. Las clases abstractas *AttributesClass* y *MethodsClass* se utilizan para ilustrar la interrelación entre los grupos de clases. En el esquema se utiliza el concepto de asociaciones abstractas como se describe en [McC97]. La composición entre *DMethods* y *DAttributes* (ver figura 10) es una herencia de la relación de composición declarada entre las clases *MethodsClass* y *AttributesClass*. Por razones de brevedad se muestran solamente las asociaciones, atributos y métodos para la clase *D*.

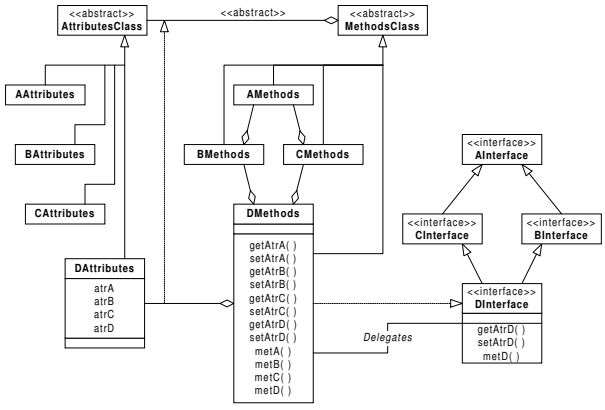

Figura10. Una vista de la segunda variante en la estrategia combinada para Java..

Como muestra la figura 11 un objeto de la clase B se representará mediante objetos de las clases *BAttributes*, *AMethods* y *BMethods*. Un objeto de la case D de la jerarquía original se representará por objetos de las clases *DAttributes*, *AMethods*, *BMethods*, *CMethods* y *DMethods*.

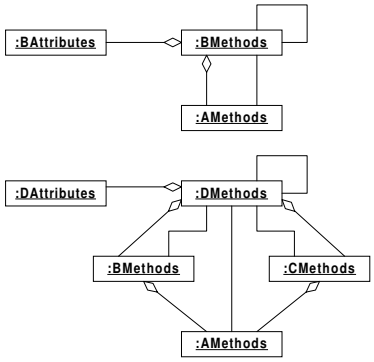

Figura11. Objetos de las clases B y D según la segunda variante de la estrategia combinada para Java.

Entre las clases de métodos y las interfaces se establece una asociación de nombre *Delegates* que indica qué objeto está delegando en la correspondiente clase de métodos (como se ve en la figura 10). En la figura 11, por simplificar, se muestra esta asociación sin nombre porque todas las asociaciones que aparecen en la figura son de este tipo. Dado un objeto de la clase original D, se necesitan cuatro objetos de las clases de métodos. Uno de estos, *DMethods*, se asocia con el resto y consigo mismo a través de la relación *Delegates*. La asociación se puede hacer efectiva gracias a la declaración de entidades como las interfaces y a la creación de objetos de las clases de métodos.

Esta asociación se necesita para resolver el problema con la referencia a sí mismo (*self-reference*) que se planteó en 2.5 cuando se presentó la composición. Básicamente se resuelve el trabajo con las funciones virtuales (en Java, todas las funciones son virtuales). Una llamada a un método f desde otro método g de la misma clase (por ejemplo en la clase B) no puede traducirse directamente porque quedaría como una llamada local en la clase de métodos y porque podría intentar trabajar con un objeto de la clase de atributos que es nulo (el método f de *BMethods* siempre sería el resultado de la ejecución aunque el objeto original fuera de la clase D, y D tuviera su propia "versión" de f). Si se quiere un comportamiento polimórfico esto no puede ocurrir. Por tanto, las llamadas a los métodos de la misma clase se envían al objeto que delegó en el actual.

Como se podía observar en la figura 2 para un lenguaje como Java serían aplicables todas las estrategias tanto básicas como combinadas. En esto se incluyen las dos variantes que se describieron en este apartado que son las que mejor

aprovechan las características del lenguaje. La segunda variante permite lograr una solución que cumple con todos los objetivos planteados.

3.2 Estrategia combinada: caso general

Cuando no se tienen interfaces o un concepto similar, las variantes de estrategia combinada que se expusieron en el apartado anterior para el caso Java no son aplicables. Como ya se ha anticipado, esta estrategia a la que se ha nombrado caso general sigue la misma idea del caso Java pero el papel de la jerarquía de interfaces será sustituido por una jerarquía de clases que se obtiene producto de una transformación por expansión de la jerarquía original. Las dos variantes que se presentaron en el caso Java se transformarán para obtener el caso general.

Para la primera variante se muestra el diagrama de clases resultante en la figura 12. Nuevamente se detallan sólo los elementos relacionados con la clase D de la jerarquía original presentada en el ejemplo guía (figura 4).

El proceso consiste básicamente en:

- Se obtiene una jerarquía expandida a partir de la original y se filtra para eliminar los atributos.
- Se obtienen las clases de atributos (al igual que en la sección 3.1) a partir de una transformación por emancipación y un filtrado para dejar solamente los atributos.
- Se añade una relación de agregación entre cada clase de la jerarquía expandida y la clase de atributos correspondiente. Todas las clases que pertenecen al conjunto réplica de una clase de la jerarquía original, se relacionan con la misma clase de atributos (la que se generó por emancipación a partir de dicha clase).
- Se añaden a las clases expandidas los pares de métodos para obtener y modificar los atributos. Estos métodos siempre tendrán que redefinirse puesto que de padres a hijos el objeto de atributos cambia y por tanto cambia la forma de hacer referencia a un atributo en específico.

Los objetos se declaran y se crean como objetos de las clases de la jerarquía expandida. Como cada clase tiene una relación de agregación con su clase de atributos correspondiente, para una clase dada se crea el objeto de atributos intrínseco y los objetos de atributos heredados permanecen nulos.

Cuando es necesario que un objeto de una clase D actúe en el lugar donde se espera un objeto de una clase ancestro C (clases de la jerarquía original) se escoge para declarar y construir el objeto, a una clase D' en la jerarquía expandida tal que $D' \in \ddot{D}$ (pertenece al conjunto réplica de D) y $D \in G^C$ (pertenece al grafo derivado de C) según las definiciones dadas en 2.6. Si D y C son las clases del ejemplo guía entonces $D' = D2$ en la figura 12.

En cambio si se requiere que un objeto de una clase D actúe en lugares diferentes donde se esperan objetos de clases ancestros C y B que forman grafos derivados (G^C y G^B) no inclusivos (es decir, que no existe un camino de herencia de una a otra) se puede asumir lo siguiente. Se escogen para declarar y construir

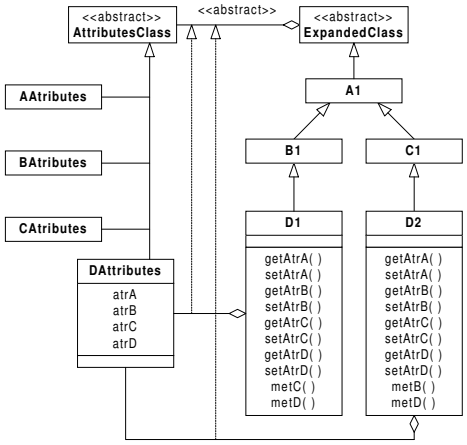

Figura12. Una vista de la estrategia combinada general "equivalente" a la primera variante del caso Java.

objetos a clases D' y D'' de la jerarquía expandida tales que D' y $D'' \in \ddot{D}$; $D' \in G^B$; $D'' \in G^C$. Todo esto con una salvedad, hay que crear un único objeto de atributos y se debe asociar a los objetos de las clases expandidas con el mismo objeto de atributos. Si B, C y D son las clases del ejemplo guía entonces $D' = D1$ y $D'' = D2$ en la figura 12.

Si no se requiere ningún comportamiento o asignación polimórfica da lo mismo qué clase del conjunto réplica de la clase original se escoja. En el caso del ejemplo sería lo mismo que se cree un objeto de la clase expandida *D1* o de la clase expandida *D2* porque ambas tienen las mismas propiedades (por construcción del grafo según la transformación por expansión 2.6).

De esto se tiene que un objeto de las clases emancipadas, puede ser un agregado de más de un objeto de las clases expandidas (por eso la relación es de agregación y no de composición). De esta forma se resuelve el problema planteado en 2.6 acerca de mantener la coherencia.

Con esta estrategia se cumple en gran medida el primer objetivo acerca de mantener la clasificación jerárquica. No se cumple de manera total pues no es posible ya que no existe ningún recurso (como las interfaces en el caso Java) que sustituya la herencia múltiple. Pero los caminos de herencia sencilla se mantienen. El segundo objetivo acerca de conservar las asignaciones y el comportamiento polimórfico se logra como se ha explicado en los párrafos anteriores. Del tercer objetivo se logra solamente la parte de mantener la coherencia pero una vez más, este esquema tiene duplicación de código. Al igual que se hizo en el caso Java se puede pasar a una segunda variante siguiendo la misma idea de introducir las clases de métodos. En la figura 13 se muestra una vista de las clases resultantes de esta transformación correspondientes al ejemplo de la figura 4.

De forma resumida el proceso consiste en:

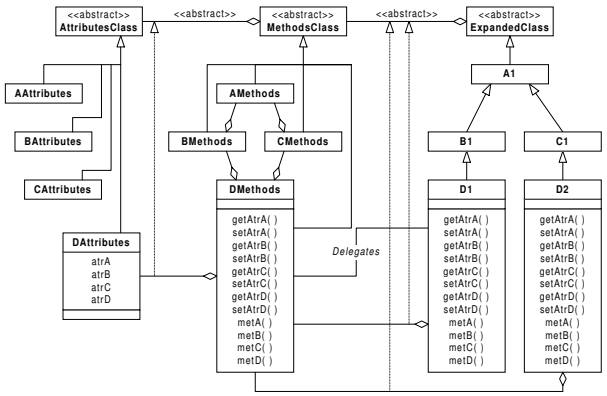

Figura13. Una vista de la estrategia combinada general "equivalente" a la segunda variante del caso Java..

- Obtener una jerarquía expandida a partir de la original filtrada para eliminar los atributos.
- Obtener las clases de atributos (al igual que en la sección 3.1) a partir de una transformación por emancipación y un filtrado para dejar solamente los atributos.
- Obtener las clases de métodos a partir de una transformación por composición filtrada para eliminar los atributos (al igual que en la sección 3.1).
- Se añaden a las clases de métodos los pares de métodos para obtener y modificar los atributos.
- Se añade una relación de agregación entre cada clase de métodos y la clase de atributos correspondiente (la que se generó por emancipación a partir de la misma clase).
- Se añaden a las clases expandidas los pares de métodos para obtener y modificar los atributos. Estos métodos siempre tendrán que redefinirse puesto que la forma de hacer referencia a dichos atributos va a depender del objeto de métodos que se corresponde con el atributo de métodos intrínseco a la clase.
- Se añade una relación de composición entre cada clase expandida y la clase de métodos correspondiente. Todas las clases que pertenecen al conjunto réplica de una clase en la jerarquía original estarán relacionadas con la misma clase de métodos: la que se originó por composición a partir de dicha clase original.
- Los métodos de las clases expandidas se convierten en métodos simples[4] que delegan en el método correspondiente del objeto de métodos intrínseco asociado.
- Se añade la relación *Delegates*

[4] métodos simples sólo hacen referencia una variable de instancia y su única sentencia es enviar un mensaje a esta variable [Cha96]

Ahora los cuerpos de los métodos residen en las clases de métodos resultantes de la transformación por composición. Los métodos que se encuentran en las clases expandidas delegan en los correspondientes métodos de las clases de métodos. Todas las clases que en la jerarquía expandida provienen del mismo conjunto réplica están relacionadas con la misma clase de métodos.

Al igual que en el caso de Java, se utiliza la asociación *Delegates*. Ahora esta asociación ocurre entre una de las clases de métodos y una de las clases expandidas.

La figura 14 muestra los objetos necesarios para simular un objeto de la clase de la jerarquía original. El objeto de la clase *D2* puede eliminarse si no se necesita que el objeto de la clase *D* actúe en el lugar donde se espera a un objeto de la clase *C*. Nuevamente la asociación *Delegates* se muestra sin nombre.

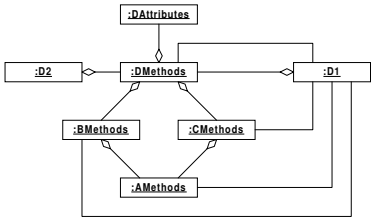

Figura14. Objetos de la clase *D* según la estrategia combinada general "equivalente" a la segunda variante del caso Java..

4 Trabajos relacionados

Jamie [VTB98] es un pre-procesador para Java que brinda delegación automática como una alternativa a la herencia múltiple. En esta aproximación el programador tiene que decidir en cual objeto delega. Su propósito no es la traducción automática pero pudiera ser útil en la implementación de estas propuestas.

J2C++[5] permite acceder a objetos C++ desde Java. Para esto asocia a cada clase C++ una clase Java representante. J2C++ solamente soporta clases C++ con herencia sencilla. Las estrategias que aquí se describen podrían utilizarse para resolver esta limitación.

Bruce[6] es un traductor de Eiffel a Java. Hasta el momento se tiene poca información disponible, pero ellos plantean que el procedimiento consiste en "transformar cada clase Eiffel no-genérica a una interfaz Java" y "seleccionar un sub-árbol maximal de la jerarquía Eiffel de clases efectivas para traducirlas directamente a una jerarquía de clases en Java, las restantes clases efectivas

[5] http://www.alphaworks.ibm.com/formula.nsf/toolpreview
[6] http://www.mri.mq.edu.au/dotg/projects/bruce

necesitan ser aplastadas (*flattened*)". Podría analizarse utilizar la aplicación de la estrategia combinada para el caso Java que se describe en este trabajo y así evitar la duplicación de código.

OO-METHOD CASE es una herramienta de generación automática de código a partir de un modelo conceptual en OO-METHOD. En el modelado conceptual de OO-METHOD aparecen varias situaciones que conducen a un diseño con herencia múltiple. Estos casos se describen en forma de patrones en [Pel98]. En dicha herramienta se genera código para estos diseños y las decisiones de implementación para codificar estos también se describen en [Pel98]. Situaciones similares a la primera variante del caso Java y a las estrategias básicas que en el presente trabajo se denominan como composición y "tipo *variant...*" son las que se destacan en la generación de código para OO-METHOD CASE en la versión actual del generador. Por esta cercanía, una fase de trabajo futuro acerca de este tema pasa por un trabajo integrador que tienda a fundir de alguna forma las tendencias seguidas en OO-METHOD CASE con el estudio que aquí se presenta.

La estrategia que se denomina expansión fue propuesta por Marqués en [Mar95]. La estrategia nombrada emancipación toma la idea de la noción de forma "aplastada" de una clase [Mey90], asimismo como la estrategia de composición tiene su base en los trabajos de Lynn Stein sobre delegación [Ste87]. La estrategia que a la que se denomina tipos *variant...* es una idea que parte del trabajo de codificación automática a partir del modelado conceptual en OO-METHOD CASE y que fue sugerida por el grupo de Valencia a los autores en comunicación personal cuando el presente trabajo no incluía dicho caso. Los trabajos sobre tipos de Cardelli [Car84] y Wegner [CW85] también fueron una base para su conformación. Otros antecedentes importantes para este trabajo se encuentran en los algoritmos de linearización que se han propuesto para los lenguajes de la familia de CLOS[Ste90]. Aunque en este caso no se ha considerado clasificarlos como estrategias de transformación de herencia múltiple a herencia simple puesto que es más apropiado ubicarlos entre los mecanismos de resolución de conflictos cuando la herencia múltiple está presente.

5 Conclusiones

Del estudio de diseños con herencia múltiple se pueden extraer algunos aspectos positivos y negativos. Los diseños con herencia múltiple son importantes para representar los conceptos del mundo en el modelo de diseño. Por otra parte, no todos los lenguajes OO permiten herencia múltiple pero cuando sí lo permiten con la herencia múltiple aparecen algunos conflictos. Luego, sería deseable primero: promover el diseño con herencia múltiple y segundo: tener disponibles métodos para transformar jerarquías de herencia múltiple en jerarquías de herencia sencilla, pasando por un filtro que reciba la orden adecuada de resolución de conflictos.

En este trabajo se analizan varias aproximaciones al problema de la transformación de jerarquías de herencia múltiple a jerarquías de herencia sencilla.

Dichas aproximaciones pueden dividirse a groso modo en estrategias básicas y estrategias combinadas.

De un estudio previo se concluyó que las principales debilidades que presentan algunas estrategias están relacionadas con problemas de mantenimiento de coherencia, duplicación excesiva de código y de asegurar las asignaciones y el comportamiento polimórfico. Otro aspecto importante que se analiza es la falta de conservación de la clasificación jerárquica a través de la herencia que se tenía originalmente. Resolver estas debilidades se ha planteado en forma de objetivos a cumplir por una estrategia de transformación.

En consecuencia con esto, en cada sección donde se describe una estrategia se valora el método a la luz de dichos objetivos. Esto nos permite concluir que las estrategias básicas son buenas por sencillas de aplicar pero inclumplen en un alto grado con los objetivos propuestos.

Las estrategias que se proponen bajo el nombre de estrategias combinadas se agrupan en dos casos: el caso general y el caso Java. Estas estrategias como su nombre los indica combinan algunas de las estrategias básicas (con modificaciones) y evitan los problemas de coherencia, disminuyen la duplicación de código y están preparados para que los objetos actúen polimórficamente. Otro aspecto destacable es cómo conservan en alta medida la clasificación a través de la herencia. Estas estrategias tienen como inconvenientes que son más complejas de manejar y que no son universalmente aplicables porque el caso Java es aplicable solamente para lenguajes con características similares a este y el caso general es aplicable a cualquier lenguaje basado en clases que soporte herencia sencilla pero no a un lenguaje que no tenga el recurso de la herencia. Las estrategias nombradas como emancipación, composición y tipos *variant* si serían aplicables en este caso pero se cargaría con los problemas que introducen.

La propuesta se particuliza para implementar en Java diseños con herencia múltiple aprovechando la presencia en Java de herencia múltiple de interfaces. Podría ser interesante tener traductores automáticos de algunas bibliotecas (independientes de plataforma) de C++ o Eiffel a Java, aplicando esta estrategia para lograr la traducción completa.

Un trabajo inmediato pasa por analizar casos de uso de estas estrategias utilizando ejemplos reales que nos permitirán hacer una valoración de aplicabilidad en dependencia, no sólo de las características de los lenguajes objetivos y de las ventajas e inconvenientes que plantean las estrategias con respecto al cumplimiento de los objetivos que es el análisis que se ha realizado en este trabajo, sino del tipo de modelo que se quiera codificar. Para esto se pretende trabajar en coordinación con el grupo de Valencia dentro del marco del proyecto MENHIR.

En los trabajos de experimentación y pruebas realizados o en curso se han tenido en cuenta otros aspectos importantes como: los métodos de creación, los controles de acceso y las clases abstractas que no se han tenido en cuenta aquí a la hora de presentar las estrategias.

Agradecimientos

Este trabajo ha sido financiado parcialmente por el proyecto CICYT TIC97-0593-C05-05. Yania Crespo es estudiante de doctorado gracias al Instituto de Cooperación Iberoamericana (ICI). Los autores agradecen al resto de los miembros del grupo GIRO (Grupo de Investigación en Reutilización y Orientación a Objetos) por su apoyo y preocupación y por nuestras ricas e interminables discusiones cada viernes.

Referencias

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Java Series. Sun Microsystems, 1996.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Pub. Co., 2nd edition, 1994.

[BRJ96] G. Booch, J. Rumbaugh, and I. Jacobson. UML. Technical report, Rational Software Corporation, 1996. Available on the web: http://www.rational.com/ot/uml.html.

[Car84] L. Cardelli. A semantics of multiple inheritance. *Semantics of Data Types, LNCS 173*, pages 51–68, 1984. also in Information and Computation 76, 1988.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G.Ñelson. Modula-3 report (revised). Technical Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, 1989.

[Cha96] H.S. Chae. Restructuring of classes and inheritance hierarchy in object-oriented systems. Master's thesis, Software Engineering Laboratory, Computer Science Depart. Korea Advanced Institute of Science and Technology (KAIST), 1996.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of OOPSLA'89, ACM SIGPLAN Notices*, pages 433–443. 24(10), October 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–523, Dec 1985.

[Del] Delphi 4. Object pascal language guide. Inprise Corporation, available on the web: http:// www.inprise.com/ delphi.

[DHHM94] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. *ACM SIGPLAN notices*, 29(10), Oct 1994.

[FBB92] B.N. Freeman-Benson and A. Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society International Conference on Programming Languages*, pages 174–180, April 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GMM98] F.J. García, J.M. Marqués, and J.M. Maudes. Mecanos as basis of compositional/generative mixed reuse model. In *Proceedings of II European Reuse Workshop, Madrid, Spain*, November 1998.

[LNS91] M. Lenzerini, D.Ñardi, and M. Simi, editors. *Inheritance hierarchies in knowledge representation and Programming Languages*. John Wiley & Sons, 1991.

[LRSP98] P. Letelier, I. Ramos, P. Sánchez, and O. Pastor. *OASIS versión 3.0. Un enfoque formal para el modelado conceptual Orientado a Objeto*. Servicio de Publicaciones Universidad Politécnica de Valencia, 1998.

[Mar95] J.M. Marqués. *Jerarquías de herencia en el diseño de software orientado al objeto*. PhD thesis, Universidad de Valladolid, 1995.

[McC97] B. McCarthy. Association inheritance and composition. *JOOP: Journal of Object Oriented Programming*, 10(4), July/August 1997.

[Mey90] B. Meyer. Tools for a new culture: Lessons from the design of the Eiffel libraries. *CACM*, 33(9):68–88, Sept 1990.

[Mey92] B. Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice-Hall, 1992. second revised printing.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. also Technical Report UIUCDCS-R-92-1759.

[Pel98] V. Pelechano. Fundamentos metodológicos para el tratamiento de la herencia múltiple en un entorno de producción automática de software. Aspectos de notación, semántica y generación automática de código. Technical report, Departamento de Sistemas Informáticos y Computación (DSIC). Universidad Politécnica de Valencia, 1998.

[PIP+97] O. Pastor, E. Insfrán, V. Pelechano, J. Romero, and J. Merseguer. OO-METHOD: An OO software production environment combining conventional and formal methods. In *Conference on Advanced Information Systems Engineering (CAiSE'97). Barcelona, Spain*, 1997.

[RBP91] J. Rumbaugh, M. Blaha, and W. et al Premerlan. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rüp92] A. Rüping. Why multiple inheritance is a natural description technique. In *ECOOP'92 "Workshop Multiple Inheritance and Multiple Subtyping"*. [Se92].

[San92] Ph. Santas. Multiple subclassing & subtyping for symbolic computation. In *ECOOP'92 "Workshop Multiple Inheritance and Multiple Subtyping"*. [Se92].

[SB86] M. Stefik and D.G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 6(4), 1986.

[Se92] M. Sakkinen (ed). Workshop W1. In *ECOOP'92 "Workshop Multiple Inheritance and Multiple Subtyping"*., 1992.

[Ste87] L. Stein. Delegation is inheritance. In *Proceedings of OOPSLA'87*, 1987.

[Ste90] G.L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[VTB98] J. Viega, B. Tutt, and R. Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, University of Virginia, 1998. http:// www.list.org/ jamie.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87*, 1987.

[WN95] K. Waldén and J.M. Nerson. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems.* Object-Oriented Series. Prentice Hall, 1995.